Proving Symmetries by Model Transformation

C. Mears[1], T. Niven[1], M. Jackson[2], and M. Wallace[1]

{Chris.Mears,Todd.Niven,Mark.Wallace}@monash.edu,
M.G.Jackson@latrobe.edu.au

[1] Faculty of IT, Monash University, Australia
[2] Department of Mathematics, La Trobe University, Australia

Abstract. The presence of symmetries in a constraint satisfaction problem gives an opportunity for more efficient search. Within the class of matrix models, we show that the problem of deciding whether some well known permutations are model symmetries (solution symmetries on every instance) is undecidable. We then provide a new approach to proving the model symmetries by way of model transformations. Given a model M and a candidate symmetry σ, the approach first syntactically applies σ to M and then shows that the resulting model $\sigma(M)$ is semantically equivalent to M. We demonstrate this approach with an implementation that reduces equivalence to a sentence in Presburger arithmetic, using the modelling language MiniZinc and the term re-writing language Cadmium, and show that it is capable of proving common symmetries in models.

1 Introduction

Solving a constraint satisfaction problem (CSP) can be made more efficient by exploiting the symmetries of the problem. In short the efficiency is gained by omitting symmetric regions of the search space. The automated detection of symmetries in CSPs has recently become a topic of great interest. However, the majority of research into this area has been directed at individual instances of CSPs where the exact set of variables, constraints and domains are known before the detection takes place. The most accurate and complete methods for detecting solution symmetries are computationally expensive and so limited in the size of problem they can tackle (e.g. [10, 8, 1]).

A CSP *model* represents a class of CSPs and is defined in terms of some parameters. An *instance* is generated from the model by assigning values to the parameters. There are automatic symmetry detection methods for CSP models, as described in [13, 14]. However they are problem-specific or can only detect a very small collection of simple symmetries, namely piecewise value and piecewise variable interchangeability.

All four authors were supported under Australian Research Council's Discovery Projects funding scheme (project number DP110102258 for the first, second and fourth authors and project number DP1094578 for the third author).

Mears et al. [9] proposed a broader framework to detect model symmetries which only requires explicitly detecting solution symmetries on small instances. The framework can be described as performing the following steps:

1. Detect symmetries on some collection of small instances of the model,
2. Lift the detected symmetries to model permutations,
3. Filter the model permutations to keep only those that are likely to be symmetries for all instances of the model (candidate model symmetries),
4. Prove that the selected model permutations are indeed symmetries for every instance of the model (model symmetries).

Mears et al. [9] developed an automated implementation of this framework on matrix models (i.e. the variables of each instance have an underlying matrix structure) that tackles steps 1, 2 and 3 whilst preliminary attempts at 4, using graph techniques, can be found in [7]. These graph theoretic approaches were, however, ad hoc and not automated. Automating step 4 can be approached by way of automated theorem proving as in [6], where the authors represent their models in existential second order logic and use a theorem proving application to verify that a candidate model symmetry is a model symmetry. Whilst potentially quite powerful, this approach requires a large amount of work to translate a practical model into the required form.

The distinction between constraint symmetry and solution symmetry proves to be critical. This paper studies the problem of proving whether a given candidate symmetry is a model symmetry. One result that we provide is that, in general, deciding whether or not some well known candidate symmetries are model "solution" symmetries is undecidable and indeed undecidable under quite weak assumptions on the models. These results consider models that can be viewed as tiling problems and then utilises the standard method of encoding Turing machines into tiling problems introduced by Robinson [12].

From the other direction, when restricting to constraint symmetries, we provide a new method that can prove when a given candidate symmetry is a model symmetry by way of model transformations. Specifically, if we apply our candidate symmetry to our model and obtain an "equivalent" model in return, then we can deduce that our candidate symmetry is indeed a model symmetry (and indeed a constraint symmetry on every instance of the model). We implement this idea by attempting to reduce the problem of model equivalence to a first order sentence in some decidable theory.

Our implementation uses MiniZinc as the modelling language and Cadmium to perform model transformations and our method focuses on proving simple matrix symmetries (swapping dimensions, inverting dimensions and permutations of a dimension) on arbitrary matrix models. Two benefits of our method are:

1. we act directly on the MiniZinc model, being the same model that could be used in solving a given instance, and
2. the theoretical steps to transform the model are closely matched to the Cadmium rules that transform the MiniZinc model.

We present an application of our method to a set of well known bench marks.

2 Background

A CSP is a tuple (X, D, C) where X represents a set of variables, D a set of values and C a set of constraints. For a given CSP, a *literal* is defined to be an ordered pair $(x, d) \in X \times D$ and represents the expression $x = d$. We denote the set of all literals of a CSP P by $lit(P)$ and define $var(x, d) = x$, for all $(x, d) \in lit(P)$. An *assignment* A is a set of literals. An *assignment over a set of variables* $V \subseteq X$ has precisely one literal (x, d) for each variable $x \in V$. An assignment over X is called a *complete* assignment.

A constraint c is defined over a set of variables, denoted by $vars(c)$, and specifies a set of *allowed* assignments over $vars(c)$. An assignment A over $V \subseteq X$ *satisfies* constraint c if $vars(c) \subseteq V$ and the set $\{(x, d) \in A \mid x \in vars(c)\}$ is allowed by c. A *solution* is a complete assignment that satisfies every constraint in C.

A *solution symmetry* σ of a CSP P is a permutation on $lit(P)$ that preserves the set of solutions [1], i.e. σ is a bijection from $lit(P)$ to $lit(P)$ that maps solutions to solutions. A permutation f on the set of variables X induces a permutation σ_f on the set of literals $lit(P)$ in the obvious way, i.e. $\sigma_f(x, d) = (f(x), d)$. A *variable symmetry* is a permutation of the variables whose induced literal permutation is a solution symmetry. Similarly, a *value symmetry* is a solution symmetry $\sigma_f(x, d) = (x, f(d))$, for some permutation f on D. If d is a set, then f is a permutation on all possible elements of d. A *variable-value* symmetry is a solution symmetry that is neither a variable nor a value symmetry.

The *microstructure complement* of a CSP P is a graph with vertices $X \times D$, and a hyperedge between a set of vertices if that set represents an assignment disallowed by some constraint, or disallowed because it assigns distinct values to one variable. A *constraint symmetry* of a CSP P is an automorphism of the microstructure of P [1]. Note that every constraint symmetry of a problem is also a solution symmetry.

A CSP *model* is a parametrised form of CSP, where the overall structure of the problem is specified, but particular details such as size are omitted. A *model permutation* σ of a CSP model M is a function that takes an instance P of the model M and produces a permutation on $lit(P)$, i.e. $\sigma(P)$ is a permutation on $lit(P)$, for all instances P of M. A *model (constraint) symmetry* σ of a CSP model M is a model permutation such that $\sigma(P)$ is a solution (constraint) symmetry, for all instances P of M. For the purposes of this paper, a *matrix model* is a model M such that the variables of M form a single n-dimensional matrix of the following form:

$$\{x[i_1, i_2, \ldots, i_n] \mid 1 \leq i_j \leq d_j \text{ for all } 1 \leq j \leq n\},$$

where the d_j's indicate the size of each dimension and may be determined by the parameters of the model. See [3] and [4] for more on matrix models.

The models that we will be interested in are those that have parameters consisting of integers $p_1, p_2, \ldots, p_n$ and a fixed number of quantified constraints

of the form

$$(\forall k_1, k_2, \ldots k_l)\Phi(k_1, k_2, \ldots, k_l, p_1, p_2, \ldots, p_n)$$
$$G(\{x[I_1], x[I_2], \ldots, x[I_m] \mid \Psi(I_1, I_2, \ldots, I_m, k_1, k_2, \ldots, k_l, p_1, p_2, \ldots, p_n)\}) \qquad (\star)$$

where

- G is a constraint (possibly global),
- each I_i represents a list of r (not necessarily distinct) variables for some fixed r (note that we do not allow nested indexing), and
- Φ and Ψ are arithmetic formulæ, with free variables among those within the corresponding parenthesis.

Example 1. The *N-Queens* problem of size N is to construct an $N \times N$ board where we are required to place N mutually non-attacking queens.

Below is a model of the Boolean N-queens problem of size N, where N is the parameter of the model. The model uses N^2 zero-one variables – one for each combination of row and column – where the variable $x[i, j]$ is one if and only if row i and column j has a queen placed.

$$\begin{aligned}
X[N] =& \{x[i,j] \mid i,j \in R = \{1,2,\ldots,N\}\} \\
D[N] =& \{0,1\} \\
C[N] =& \{(\forall j \in R) \sum_{1 \le i \le N} x[i,j] = 1, \\
& \quad (\forall i \in R) \sum_{1 \le j \le N} x[i,j] = 1, \\
& \quad (\forall k \in \{3,\ldots,2N-1\}) \sum \{x[i,j] \mid i,j \in R, i+j=k\} \le 1, \\
& \quad (\forall k \in \{2-N,\ldots,N-2\}) \sum \{x[i,j] \mid i,j \in R, i-j=k\} \le 1\}.
\end{aligned}$$

This problem has many model symmetries; one of them is that the i and j dimensions can be interchanged (diagonal reflection of the square).

The quantified constraints in the above example are equivalent to those of $(\star)$. Indeed,

$$(\forall j \in R) \sum_{1 \le i \le N} x[i,j] = 1$$

from above is equivalent to

$$(\forall k) 1 \le k \wedge k \le N \sum \{x[i,j] \mid 1 \le i \wedge i \le N \wedge j = k\} = 1,$$

where the constraint G in this case is the global constraint of the sum equaling 1.

3 Undecidability of model symmetries

In this section we show that two common permutations cannot be algorithmically recognised as solution symmetries of CSP models. More precisely we show that the class of CSP models (with a single 2-dimensional matrix variable x)

admitting the given solution symmetry is not recursively enumerable. As the consequences of a proof system form a recursively enumerable set, a sort of incompleteness theorem follows: any proof system for proving the existence of solution symmetries from CSP models is incomplete. We prove this for the domain inversion symmetry and the dimension swap symmetry, though other cases (those in Section 4.1) can be treated using similar ideas.

Mancini and Cadoli [6] have provided similar undecidability results with respect to *problem specifications* which formulates classes of CSPs via scond order logic. The results given here differ in that we only require a single matrix of variables, the constraints have extremely simple structure and the models relate to natural geometric constraint satisfaction problems of independent interest (symmetric tilings of the plane for example).

We consider CSP models related to tiling grids by square tiles with matching conditions dictating which tile can be placed in horizontal adjacency and which can be placed in vertical adjacency. This situation may be viewed as a kind of directed graph except with two binary relations $\sim_h$ and $\sim_v$ (representing allowed horizontal and vertical adjacencies resp.) instead of one.

The basic problem of tiling an $N \times N$ grid with tiles from $\{0, \ldots, n-1\}$ is as follows (here `x[i,j]` $= k$ represents tile k being placed at position (i, j)).

$$\begin{aligned} X[N] &= \{x[i,j] \mid 0 \leq i, j \leq N-1\} \\ D[N] &= \{0, 1, \ldots, n-1\} \\ C[N] &= \{(\forall i, j \in \{0, \ldots, N-1\}, i > 0) x[i-1, j] \sim_h x[i, j] \\ &\qquad (\forall i, j \in \{0, \ldots, N-1\}, j > 0) x[i, j-1] \sim_v x[i, j]\} \end{aligned}$$

It is well known that the problem of deciding if the full positive quadrant of the plane may be tiled starting from an arbitrary finite $\mathcal{T}$ is undecidable. This is usually proved by a simple encoding of a Turing machine program into the tiles so that successfully tiled rows of the plane correspond to successful steps of computation by the Turing machine. If the Turing machine eventually halts then the tiling cannot be completed. We exploit this idea by instead allowing completion of the tiling, but only in a way that violates some symmetry always present in nonhalting situations.

The results are proved using variations of this CSP model. Note that failure to exhibit a given solution symmetry is a Σ_1^0 property. Thus it remains to show Σ_1^0-hardness. We reduce the Halting Problem to the failure of a solution symmetry.

3.1 Basic strategy

The arguments are extensions of the following idea. We use an easy variation of the Halting problem for deterministic Turing machine programs: given a Turing machine program T with no halting states, but with some distinguished state q, is the state q ever reached when T is started on the blank tape? This problem is Σ_1^0-complete.

Step 1. The program T can be encoded into a set of tiles $\mathcal{T}_T$ in a standard way (see Robinson [12] or Harel [5] for example). Successive tiled rows correspond

to successive configurations of the machine running T. Let us assume that tile 0 encodes the Turing machine in initial state reading the blank symbol and that tile 1 encodes a transition into state q. Then the following problem is Σ_1^0-complete: given T, is there a number N and a tiling of the $N \times N$ grid using $\mathcal{T}_T$ with tile 0 at position $(0, 0)$ and involving placement of tile 1?

Step 2. Duplicate some part (or all of) $\mathcal{T}_T$, creating the solution symmetry of interest. Then adjust the adjacency conditions applying to tile 1 and compared to its duplicate so that the solution symmetry is violated in sufficiently large models if and only if state q is reached by T.

3.2 Swapping of dimension: $x[i, j] \mapsto x[j, i]$.

Aside from the domain structure and the input parameter N, the constraints used in this construction involve only order and successor on indices. First, the tiling created at step 1 of the basic strategy is adjusted so that it exhibits the constraint symmetry $x[i, j] \mapsto x[j, i]$: moreover in every solution, the value of $x[i, j]$ is equal to the value of $x[j, i]$, and in no solution can tile 1 be placed on the diagonal. Step 2 of the basic strategy involves duplication of the distinguished tile 1: say that tile 2 is the exact duplicate of tile 1, with every adjacency condition for 1 applying identically to tile 2. Add one further constraint c dictating that tile 1 cannot be placed above the diagonal and that tile 2 cannot be placed below the diagonal.

If the Turing machine program $\mathcal{T}$ does not reach state q, then tile 1 cannot be placed, and the final constraint c is redundant. In this case the dimension swapping solution symmetry holds. However if T does eventually reach state q, then for sufficiently large N, a tiling of the $N \times N$ grid will involve, for some $i > j$, placement of 1 at some position $x[i, j]$ and 2 at position $x[j, i]$. This violates the solution symmetry.

3.3 Inversion of domain: $x[i, j] \mapsto n - 1 - x[i, j]$.

Aside from the domain structure and the input parameter N, the constraints used in this construction involve only successor on indices. Let us assume that the tiling created at step 1 of the basic strategy has tiles $0, \ldots, n-1$. For step 2 we duplicate these tiles to produce tiles $0, \ldots, n-1, n, \ldots, 2n-1$, with tile $i \leq n-1$ corresponding to tile $2n-1-i$: there are no adjacencies allowed between tiles from $0, \ldots, n-1$ and those from $n, \ldots, 2n-1$, but within these two blocks, the adjacency patterns are identical (except in reverse order).

Weaken the constraint $x[0, 0] = 0$ to $x[0, 0] \in \{0, 2n-1\}$: this CSP model exhibits inversion of domain as a constraint symmetry, and every solution corresponds to a tiling of an $N \times N$ grid with either the tiles $0, \ldots, n-1$ or the tiles $n, \ldots, 2n-1$. Now remove all adjacency capabilities for tile $2n-2$ (the duplicate of the special tile 1): so $2n-2$ cannot be placed. The inversion of domain solution symmetry can hold if and only if no solution involves placement of tile 1, which is equivalent to program T reaching state q.

4 Proving Symmetries by Model Transformation

Motivated by the matrix model permutations investigated in [9] (i.e. dimension swap, dimension inversion and dimension permutations), we describe an automated method that is capable of proving when such permutations are indeed model constraint symmetries. Specifically, given a common matrix permutation σ on the variables of a model M, we prove that σ is a symmetry of M by showing that $\sigma(M)$ is semantically equivalent to M. We say that a quantified constraint c in a model M is equivalent to a quantified constraint c' in the model $\sigma(M)$ if, for every instance of M, the quantified constraint c is equal to c' in the corresponding instance of $\sigma(M)$. Note that we are concerned here with "constraint symmetries" in contrast to the previous section where we found that it is undecidable to determine if such permutations are model solution symmetries.

Given a model with a set of quantified constraints C, and a symmetry σ, the method has the following steps:

1. Partition the quantified constraints into equivalence classes Θ_G where a quantified constraint $c \in \Theta_G$ if the constraint (refer to $(\star)$ in Section 2) in c is G.
2. Compute $\sigma(c)$ for each quantified constraint $c \in C$, giving the set $C' = \{\sigma(c) \mid c \in C\}$ with equivalence classes $\Theta'_G = \sigma(\Theta_G)$ (we assume that σ satisfies: if G is the constraint in c then G is also the constraint in $\sigma(c)$).
3. Normalise every quantified constraint $c' \in C'$ by reducing the expressions used as array indices to single variables by substitution.
4. For each bijection $\varphi : C \to C'$, that preserves the equivalence classes Θ_G, produce a sentence $\Phi_\varphi(c)$ (in some decidable theory) that expresses (if true) the equivalence of $c \in C$ with its matched constraint $\varphi(c) \in C'$.
5. Prove that the sentence $\bigvee_\varphi \bigwedge_{c \in C} \Phi_\varphi(c)$ is true.

Since item 4 requires two quantified constraints have the same constraint G before we attempt to match them, this method will not be complete for model constraint symmetries.

We now describe steps 2–4 in detail. We restrict ourselves to models in which the variables have integer, or set of integer, domains and the quantified constraints are of the form $(\star)$ where Φ and Ψ are first order formulæ in Presburger arithmetic (considered here to be the first order theory of $+, -, \leq, 0, 1$ over the integers; a well known decidable theory, see e.g. [2]).

4.1 Computing $\sigma(c)$

We consider the following five types of permutations acting on a matrix of variables (these include permutations from Section 3.)

- Swapping of dimensions j and k:
  $x[i_1, i_2, \ldots, i_j, \ldots, i_k, \ldots, i_n] \mapsto x[i_1, i_2, \ldots, i_k, \ldots, i_j, \ldots, i_n]$,
  where $j < k$ and $d_j = d_k$,
- Inverting of dimension j:
  $x[i_1, i_2, \ldots, i_j, \ldots, i_n] \mapsto x[i_1, i_2, \ldots, d_j - i_j + 1, \ldots, i_n]$,

- All permutations of dimension j:
  $x[i_1, i_2, \ldots, i_j, \ldots, i_n] \mapsto x[i_1, i_2, \ldots, \varphi(i_j), \ldots, i_n]$, where φ represents an arbitrary permutation on $\{1, 2, \ldots, d_j\}$,
- All permutations of values:
  $x[i_1, i_2, \ldots, i_n] \mapsto \varphi(x[i_1, i_2, \ldots, i_n])$, where φ represents an arbitrary permutation on the domain of values.
- Inverting of values:
  $x[i_1, i_2, \ldots, i_n] \mapsto u - (x[i_1, i_2, \ldots, i_n]) + l$, where l and u are the lower and upper bounds of the value domain.

These permutations appear commonly in matrix models. We define the quantified constraint $\sigma(c)$ by replacing each occurrence of $x[i_1, i_2, \ldots, i_n]$ in c with its image $\sigma(x[i_1, i_2, \ldots, i_n])$ as given above.

Example 2. One of the constraints of the N-Queens problem (Example 1) is:

$$(\forall k \in \{2 - N, \ldots, N - 2\}) \sum \{x[i, j] \mid i, j \in R, i - j = k\} \leq 1$$

where $R = \{1, 2, \ldots, N\}$. Let σ be the symmetry that swaps dimensions 1 and 2: $x[i, j] \mapsto x[j, i]$ By substituting $x[j, i]$ for $x[i, j]$, we see that $\sigma(c)$ is:

$$(\forall k \in \{2 - N, \ldots, N - 2\}) \sum \{x[j, i] \mid i, j \in R, i - j = k\} \leq 1$$

4.2 Substituting complex expressions

The goal of this step is to reduce all array accesses $x[e_1, e_2, \ldots, e_n]$, where each e_j is an expression, to the form $x[i_1, i_2, \ldots, i_n]$ where each i_j is a single variable (or constant) and the name of the variables i_j are in lexicographical order. In particular, the name of a variable i_j is lexicographically less than the name of i_k if $j < k$.

We introduce variables i_j that will ultimately take the place of the expressions e_j. We assume an expression e_j is a permutation f of a quantified variable q_j. For example, e_j could be the expression $\varphi(q_j)$ where $1 \leq q_j \leq N$ and φ is a permutation on the set $\{1, 2, \ldots, N\}$.

We introduce a new variable i_j and let $i_j = e_j$; therefore $q_j = f^{-1}(i_j)$. Using this identity, we replace all occurrences of q_j throughout the constraint with $f^{-1}(i_j)$ and as a result, e_j becomes i_j.

With the names of the introduced variables are generated in lexicographical order, we perform the substitution of the expressions e_j in order that they appear in the array access; this ensures that after simplification, the names of the i_j variables in $x[i_1, i_2, \ldots, i_n]$ are in lexicographical order.

Example 3. Consider again one of the constraints of the N-Queens problem (Example 1):

$$(\forall k \in \{2 - N, \ldots, N - 2\}) \sum \{x[i, j] \mid i, j \in R, i - j = k\} \leq 1$$

where $R = \{1, 2, \ldots, N\}$. Let σ be the symmetry that inverts dimension 1:

$$x[i, j] \mapsto x[N - i + 1, j]$$

By substituting $x[N-i+1,j]$ for $x[i,j]$, we see that $\sigma(c)$ is:

$$(\forall k \in \{2-N,\ldots,N-2\}) \sum \{x[N-i+1,j] \mid i,j \in R, i-j=k\} \leq 1$$

Let us now substitute the first expression in the array access. We introduce a new variables $\alpha = N-i+1$ and $\beta = j$. We see that α is a function of the quantified variable i, and that $i = N-\alpha+1$. Next, we replace each occurrence of the quantified variable i with $N-\alpha+1$ and each occurrence of j with β, giving:

$$(\forall k \in S) \sum \{x[\alpha,\beta] \mid N-\alpha+1, \beta \in R, N-\alpha+1-\beta=k\} \leq 1$$

where $S = \{2-N,\ldots,N-2\}$.

4.3 Equivalence via Presburger forumlæ

In the previous subsections we described how we apply a candidate symmetry to the quantified constraints and to rewrite them into a reduced form that matches the form of one or more of the original constraints. We now want to determine if two model constraints are equivalent; if checking this can be formulated into a first order statement in some decidable theory, then a theorem prover can prove or disprove equivalence.

Example 4. In Example 3 we obtained the quantified constraint

$$(\forall k \in S) \sum \big\{x[\alpha,\beta] \mid N-\alpha+1, \beta \in R \text{ and } N-\alpha+1-\beta=k\big\} \leq 1$$

where $R = \{1,\ldots,N\}$ and $S = \{2-N,\ldots,N-2\}$. Renaming α to i and β to j we obtain

$$(\forall k \in S) \sum \big\{x[i,j] \mid N-i+1, j \in R \text{ and } N-i+1-j=k\big\} \leq 1.$$

It so happens that this quantified constraint is equivalent to one from the original model, namely

$$(\forall l \in \big\{3,\ldots,2N-1\}) \sum \big\{x[i,j] \mid i,j \in R \text{ and } i+j=l\big\} \leq 1.$$

Since the relations corresponding to the sum global constraint are identical in the two constraints whenever they have the same arity, the equivalence of the quantified constraints is equivalent to their scopes being the same; which corresponds to the following sentences holding in the integers:

$$\begin{aligned}(\forall i,j,N)(\forall k \in S)(N-i+1 \in R) \wedge (j \in R) \wedge (N-i+1-j=k) \\ \Rightarrow (\exists l \in \{3,\ldots,2N-1\})(i \in R) \wedge (j \in R) \wedge (i+j=l)\end{aligned}$$

and

$$\begin{aligned}(\forall i,j,N)(\forall l \in \{3,\ldots,2N-1\})(i \in R) \wedge (j \in R) \wedge (i+j=l) \\ \Rightarrow (\exists k \in S)(N-i+1 \in R) \wedge (j \in R) \wedge (N-i+1-j=k)\end{aligned}$$

These sentences are both true and can easily be seen to be equivalent to sentences in Presburger arithmetic.

4.4 An exploration of N-Queens

In this section we further explore the model symmetry $\sigma(i,j) = (j,i)$ for the N-queens model described in Section 2.

This model involves two global constraints, both involving $\sum$. This gives us the two equivalence classes

$$\begin{aligned}\Theta_{(=1)} =&\{(\forall j \in R) \sum_{1\leq i\leq N} x[i,j] = 1, (\forall i \in R) \sum_{1\leq j\leq N} x[i,j] = 1\} \text{ and}\\ \Theta_{(\leq 1)} =&\{(\forall k \in \{3,\ldots,2N-1\}) \sum\{x[i,j] \mid i,j \in R, i+j=k\} \leq 1,\\ &(\forall k \in \{2-N,\ldots,N-2\}) \sum\{x[i,j] \mid i,j \in R, i-j=k\} \leq 1\}.\end{aligned}$$

Applying σ to C we obtain $C' = \sigma(\Theta_{(=1)}) \cup \sigma(\Theta_{(\leq 1)})$, where

$$\begin{aligned}\sigma(\Theta_{(=1)}) =&\{(\forall j \in R) \sum_{1\leq i\leq N} x[j,i] = 1, (\forall i \in R) \sum_{1\leq j\leq N} x[j,i] = 1\}\\ &\text{and}\\ \sigma(\Theta_{(\leq 1)}) =&\{(\forall k \in \{3,\ldots,2N-1\}) \sum\{x[j,i] \mid i,j \in R, i+j=k\} \leq 1,\\ &(\forall k \in \{2-N,\ldots,N-2\}) \sum\{x[j,i] \mid i,j \in R, i-j=k\} \leq 1\}.\end{aligned}$$

Notice that, the constraints in C' are already essentially normalized (the indices are not lexicographically ordered, however this will not concern us), so we just need to find a bijection φ from C to C' that maps $\Theta_{(=1)}$ to $\sigma(\Theta_{(=1)})$ and $\Theta_{(\leq 1)}$ to $\sigma(\Theta_{(\leq 1)})$ such that for all $c \in C$, the quantified constraint $\varphi(c)$ is equivalent to c.

Let φ be a bijection from C to C' such that $\varphi(\Theta_{(=1)}) = \sigma(\Theta_{(=1)})$ and $\varphi(\Theta_{(\leq 1)}) = \sigma(\Theta_{(\leq 1)})$ (there are only 4 such maps). Since our quantified constraints are equivalent to those of the form $(\star)$, to determine if $c \in C$ is equivalent to $\varphi(c) \in C'$ amounts to proving sentences in Presburger like those we found in Section 4. In this case, the map φ that matches $c \in \Theta_{(\leq 1)}$ with its corresponding $\sigma(c)$ and matches $c \in \Theta_{(=1)}$ with $c' \in \Theta_{(\leq 1)}$ where $c' \neq \sigma(c)$, will produce true sentences, whilst all other φ will produce a sentence that is false.

5 Implementation

The transformations described in the previous section are implemented as Cadmium rules that act on a MiniZinc model. Before showing the details of our implementation, we describe briefly MiniZinc and Cadmium.

A MiniZinc model is a set of items. The items we are interested in are *constraint* items: it is these that we will be manipulating. Consider this example constraint item:

```
constraint forall (i,j in 1..N) ((sum (k in 1..N) (x[i,j,k]) = 1));
```

The token `constraint` introduces a constraint item. The `forall` indicates a quantification of some variable(s) over some range(s) of values. The first parenthesised part `(i,j in 1..n)` is called a generator and introduces the two variables that are to be quantified, and that both range over the set of integers from 1 to N inclusive. The body of the quantification is the second parenthesised part. The left hand side of the `=` constraint is a `sum` expression that introduces an index variable `k` which also ranges over the set 1 to N, and the expression as a whole evaluates to the sum of `x[i,j,k]` for a given `i` and `j` over those values of `k`. The right hand side is simply the constant 1. This constraint item therefore represents the constraint:

$$(\forall i, j \in R) \sum_{k \in R} x[i, j, k] = 1 \text{ where } R = \{1, 2, \ldots, N\}.$$

Since we only consider quantified constraints of the form ($\star$) from Section 2, we only operate on a subset of Minizinc.

MiniZinc models are translated into terms to be manipulated by Cadmium rules. A Cadmium rule has the following form:

```
Context \ Head <=> Guard | Body.
```

The meaning of a rule is that wherever `Head` occurs in the model it should be replaced by `Body`, but only if `Guard` is satisfied and if `Context` appears in the conjunctive context of `Head`. Roughly, the conjunctive context of a term is the set of all terms that are joined to it by conjunction. The `Context` and `Guard` parts are optional. Consider the following example Cadmium rules:

```
-(-(X)) <=> X.
constraint(C) <=> ID := unique_id("con") |
                  (constraint_orig(ID,C) /\ constraint_to_sym(ID,C)).
```

The first rule implements a basic arithmetic identity. Identifiers such as `X` that begin with an uppercase letter are variables and can match any term. The head `-(-(X))` matches any term `X` that is immediately preceded by two negations, and such a term is replaced by the body `X`. The second rule is more complex. It matches any constraint item `constraint(C)` and replaces it with the conjunction `constraint_orig(ID,C) /\ constraint_to_sym(ID,C)`. The body of the constraint item `C` is duplicated into two items `constraint_orig(ID,C)` and `constraint_to_sym(ID,C)`, where the new names `constraint_orig` and `constraint_to_sym` are arbitrary and do not have any interpretation in MiniZinc. The guard `ID := unique_id("con")` calls the standard Cadmium function `unique_id` to supply a unique identifier to be attached to the constraints. This guard always succeeds; its purpose is to assign a value to `ID`.

Each step of the method corresponds to a set of Cadmium rules. In this section we show excerpts of the relevant parts of the Cadmium rules that implement these steps. Particular details of Cadmium will be explained as necessary.

5.1 Computing $\sigma(c)$

First, the constraints are duplicated and the symmetry is applied.

```
% Every constraint C is given a unique ID and is duplicated.
constraint(C) <=> ID := unique_id("con") |
                  (constraint_orig(ID,C) /\ constraint_to_sym(ID,C)).
% Every constraint in the duplicated set has the symmetry applied.
constraint_to_sym(ID,C) <=> constraint_sym(ID,sigma(C)).
```

The rule for `sigma` depends on the particular symmetry to be tested. Here are three possible definitions, corresponding to the first three kinds of permutation in Section 4.1. The "all-permutations" symmetries are represented by a syntactic construct that represents an arbitrary permutation.

```
% Dimensions 1 and 2 swap:  x[i,j,k] -> x[j,i,k]
sigma(aa(id("x"), t([I,J,K]))) <=> aa(id("x"), t([J,I,K])).
% Inverting of dimension 1:  x[i,j,k] -> x[n-i+1,j,k]
sigma(aa(id("x"), t([I,J,K]))) <=>
                                  aa(id("x"), t([id("n")+(-I)+i(1),J,K])).
% All permutations of dimension 1:  x[i,j,k] -> x[phi(i),j,k]
sigma(aa(id("x"), t([I,J,K]))) <=>
                                 aa(id("x"), t([permutation(phi,I),J,K])).

% Traverse the entire constraint term to apply the symmetry.
sigma(E) <=> '$arity'(E) `$==` 0 | E.
sigma(E) <=> [F|A] := '$deconstruct'(E) |
             '$construct'([F | list_map(sigma, A)]).
```

The term `aa(id("x"), t([I,J,K]))` represents a MiniZinc array access of the form `x[I,J,K]`, where `I`, `J` and `K` are arbitrary terms. The `id(S)` term represents an identifier with name `S` (a string), and the `t([...])` term represents a tuple (in this case the indices of the array).

The final two rules implement a top-down traversal of a term. Zero-arity terms, such as strings, are handled in the first rule: they are left unchanged. Compound terms, such as `constraint_to_sym(ID,C)`, are broken into their functor (`constraint_to_sym`) and their arguments (`ID` and `C`), and the symmetry is applied recursively to the arguments. The special `$deconstruct` and `$construct` functions respectively break a term into its parts or reconstruct a term from its parts.

5.2 Substituting complex expressions

In this step we find the expressions used in array accesses and replace them with single variables. Firstly, we find those expressions used in the array accesses.

```
% Extract array indices in the order that they are used.
% I,J,K may be complex expressions.
extract_indices(aa(_Array, t([I,J,K]))) <=> [I,J,K].
% (Traversal omitted.)
```

The result is a list of expressions that should be replaced with single variables. This list is passed as the first argument to the `rename_list` rule. Note that the order that the expressions were found in the array access is also the order in which they are renamed.

```
rename_list([], T) <=> T.
% Replace in term T the complex expression X with a fresh variable Y.
rename_list([X|Xs], T) <=>
        Y := unique_id("index") /\
        renaming(From, To) := compute_renaming(X, id(Y)) |
              substitute_ids([From `maps_to` To], T).
```

The term `X` is the expression e_j to be replaced. The first part of the guard `Y := unique_id("index")` generates the fresh variable i_j. As described in Section 4.2, we assume that $e_j = f(q_j)$ and replace all occurrences of q_j with $f^{-1}(i_j)$. The rule `compute_renaming` computes this replacement $f^{-1}(i_j)$; the standard Cadmium rule `substitute_ids` performs the replacement throughout the term `T`.

The `compute_renaming` begins with the complex expression e_j as the first argument, and the replacement variable i_j as the second argument. Parts of the expression are moved to the second argument until the first argument is a single variable (a bare identifier).

```
% The inverse of phi(X) is invphi(X).
compute_renaming(permutation(Phi, X), Y) <=>
  compute_renaming(X, inverse_permutation(Theta, Y)).

% If X is a global variable (e.g. a parameter), then move it to
% the right hand side.
% X + Y = Z  -->  Y = Z - X.
decl(int,id(X),_,global_var,_) \
  compute_renaming(id(X)+Y, Z) <=> compute_renaming(Y, Z + (-id(X))).
% -X = Y  ->  X = -Y.
compute_renaming(-(id(X)), Y) <=> compute_renaming(id(X), -(Y)).
% X + Y = Z  -->  X = Z - Y.
compute_renaming(id(X)+Y, Z) <=> compute_renaming(id(X), Z + -(Y)).
% -X + Y = Z  -->  X - Y = -Z.
compute_renaming(-(id(X))+Y, Z) <=> compute_renaming(id(X) + -(Y), -(Z)).

% When the left hand side is a mere identifier, the right hand side
% is the expression to replace it with.
compute_renaming(id(X), Y) <=> renaming(id(X), Y).
```

Note the use of the contextual guard `decl(int,id(X),_,global_var,_)` in the second rule. This means that the identifier `X` is moved to the second argument only if it is declared as a global variable somewhere in the conjunctive context of the term being matched to the head. This contextual matching feature of Cadmium allows parts of the model that occur in distant parts of the model to be used when determining if a rule should apply. Also note that a pattern such as `id(X)+Y` exploits the commutativity and associativity of addition; Cadmium rearranges the expression as needed to make the pattern match.

5.3 Producing and proving Presburger formulæ

Finally, we attempt to match each quantified constraint in C with a quantified constraint in C' that is in the same equivalence class Θ_G. Ensuring that the two quantified constraints are in the same class is done by inspecting the structure of the terms. The test for equivalence of the quantified constraints is then reduced to a Presburger sentence. We extract from each quantified constraint the expressions for Φ and Ψ (see $(\star)$) and construct a Presburger sentence as described in Section 4.3. The sentence is then passed to the Presburger solver *Omega*, which uses the omega test [11] to prove or disprove the sentence.

For a given symmetry σ, if all constraints in $\sigma(C)$ can be shown to match a constraint in C, then we state that σ is a model symmetry.

Table 1. Summary of Symmetries Proved

Problem	Variable Symmetries	
Latin Squares (Boolean)	`x[i,j,k]` $\mapsto$ `x[j,i,k]`	`x[i,j,k]` $\mapsto$ `x[i,k,j]`
	`x[i,j,k]` $\mapsto$ `x[N-i+1,j,k]`	`x[i,j,k]` $\mapsto$ `x[`φ`(i),j,k]`
	`x[i,j,k]` $\mapsto$ `x[i,`φ`(j),k]`	`x[i,j,k]` $\mapsto$ `x[i,j,`φ`(k)]`
Latin Squares (integer)	`x[i,j]` $\mapsto$ `x[j,i]`	`x[i,j]` $\mapsto$ `x[`φ`(i),j]`
	`x[i,j]` $\mapsto$ `x[i,`φ`(j)]`	`x[i,j]` $\mapsto$ φ`(x[i,j])`
Steiner Triples	`x[i,j]` $\mapsto$ `x[`φ`(i),j]`	`x[i,j]` $\mapsto$ `x[i,`φ`(j)]`
BIBD	`x[i,j]` $\mapsto$ `x[`φ`(i),j]`	`x[i,j]` $\mapsto$ `x[i,`φ`(j)]`
Social Golfers	`x[i,j]` $\mapsto$ `x[`φ`(i),j]`	`x[i,j]` $\mapsto$ `x[i,`φ`(j)]`
	`x[i,j]` $\mapsto$ φ`(x[i,j])`	
N-Queens (Boolean)	`x[i,j]` $\mapsto$ `x[j,i]`	`x[i,j]` $\mapsto$ `x[N-i+1,j]`
N-Queens (integer)	`x[i]` $\mapsto$ `x[N-i+1]`	`x[i]` $\mapsto$ `N-x[i]+1`

6 Results

We have tested our model transformation approach for symmetries found by Mears et al. [9] in a suite of benchmark problems modelled in MiniZinc. Table 1 lists, for each problem, the symmetries that our implementation was able to prove hold.

The results in Table 1 show that we can verify the existence of some common variable and value symmetries in selected well-known matrix models. In addition, we have tested symmetries that are known not to hold on the models and verified that the implementation fails to prove them. Note that for the set variables in the Social Golfers problem, the value symmetry acts on the elements of the set rather than the sets themselves. The running time is negligible; for each benchmark, the execution takes around one second.

Our implementation does not deal with variable-value symmetries that cannot be expressed as a composition of a variable symmetry with a value symmetry e.g. the solution symmetry σ taking the literal $x[i] = j$ to $x[j] = i$. One way to step around this problem is to translate one's model into a Boolean model (in an appropriate way), where now the value symmetries and variable-value symmetries are simply variable symmetries.

7 Conclusion

The automatic detection of CSP symmetries is currently either restricted to problem instances, is limited to the class of symmetries that can be inferred from the global constraints present in the model, or requires the use of (incomplete) automated theorem provers. This paper, whilst showing that the fundamental problem is undecidable, provides a new way of proving the existence of model symmetries by way of model transformations. We show that simple matrix permutations, such as swapping and inverting dimensions, can be shown to be model symmetries using this method. Pleasingly, our method has also been successful in showing that an arbitrary permutation (which represents a large

group of symmetries) applied to a dimension of the matrix of variables is a model symmetry.

Acknowledgements

The authors would like to thank Leslie De Koninck and Sebastian Brand for assisting with the Cadmium development, and the G12 Project for supplying some of the MiniZinc models. We also thank Maria Garcia de la Banda for her invaluable suggestions.

References

1. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: Symmetry definitions for constraint satisfaction problems. In: Principles and Practice of Constraint Programming - CP 2005 (2005)
2. Enderton, H.: A Mathematical Introduction to Logic (second edition). Academic Press, Inc (2001)
3. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: Proc. Formul01, CP01 Workshop on Modelling and Problem Formulation (2001)
4. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Principles and Practice of Constraint Programming - CP 2002 (2002)
5. Harel, D.: Effective transformations on infinite trees with applications to high undecidability, dominoes and fairness. J. ACM pp. 224–248 (1986)
6. Mancini, T., Cadoli, M.: Detecting and breaking symmetries by reasoning on problem specifications. In: Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005) (2005)
7. Mears, C.: Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming. Ph.D. thesis, Monash University (2009)
8. Mears, C., Garcia de la Banda, M., Wallace, M.: On implementing symmetry detection. Constraints 14 (2009)
9. Mears, C., Garcia de la Banda, M., Wallace, M., Demoen, B.: A novel approach for detecting symmetries in CSP models. In: Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (2008)
10. Puget, J.F.: Automatic detection of variable and value symmetries. In: Principles and Practice of Constraint Programming - CP 2005 (2005)
11. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM pp. 102–114 (1992)
12. Robinson, R.: Undecidability and nonperiodicity for tilings of the plane. Inventiones Math. 12, 177–209 (1971)
13. Roy, P., Pachet, F.: Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In: ECAI98 Workshop on Non-binary Constraints (1998)
14. Van Hentenryck, P., Flener, P., Pearson, J., Agren, M.: Compositional derivation of symmetries for constraint satisfaction. In: Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005) (2005)