# Proving Symmetries by Model Transformation

C. Mears* and T. Niven**

{Chris.Mears,Todd.Niven}@monash.edu

Faculty of IT, Monash University, Australia

**Abstract.** The presence of symmetries in a constraint satisfaction problem gives an opportunity for more efficient search. We provide a new way of proving the existence of solution symmetries in all instances of a class of matrix CSPs by way of model transformations. Given a model $M$ and a candidate symmetry $\sigma$, the approach first syntactically applies $\sigma$ to $M$ and then shows that the resulting model $\sigma(M)$ is semantically equivalent to $M$. We show an implementation of the method using the modelling language MiniZinc and the term re-writing language Cadmium and show that it is able to prove certain kinds of symmetry.

## 1 Introduction

Solving a constraint satisfaction problem (CSP) can be made more efficient by exploiting the symmetries of the problem. Roughly speaking 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. [8, 6, 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 [9, 10]. However they are problem-specific or can only detect a very small collection of simple symmetries, namely piecewise value and piecewise variable interchangeability.

Mears et al. [5] proposed a broader framework to detect model symmetries which only requires explicitly detecting solution symmetries on small instances. The global framework can be described as performing the following steps:

1. Detect symmetries on some collection of small instances of the model,
2. Extend the detected symmetries to model permutations,

* This author was supported under Australian Research Council's Discovery Projects funding scheme (project number DP0879710).

** This author was supported by Constraint Technologies International.

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. [5] 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 [4]. 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 [3], 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.

This paper describes a new method for proving that 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 solution symmetry. Our implementation uses MiniZinc as the modelling language and Cadmium to perform model transformations. Our method focuses on proving simple variable 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.

The limitations of our method are that we only consider matrix models and particular matrix permutations as candidate symmetries. Our results, however, show that our method has potential and we believe that we can extend our implementation to deal with many other kinds of model symmetries.

## 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 literals* $V \subseteq X$ 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* $\sigma$ of a CSP $P$ is a permutation on $lit(P)$ that preserves the set of solutions [1], i.e. $\sigma$ 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 $\sigma_f$ 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.

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* $\sigma$ 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 symmetry* $\sigma$ of a CSP model $M$ is a model permutation such that $\sigma(P)$ is a solution symmetry, for all instances $P$ of $M$. A *matrix model* is a model $M$ such that the variables of $M$ form an $n$-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 may be determined by the parameters of the model.

*Example 1.* The *Latin square* problem of size $N$ is to construct an $N \times N$ square where each of the $N^2$ cells has an integer value from 1 to $N$, and each value appears exactly once in each row and exactly once in each column.

Below is a model of the Latin square problem of size $N$. The model uses $N^3$ zero-one variables – one for each combination of row, column and value – where the variable $x[i, j, k]$ is one if and only if the cell at row $i$ and column $j$ has the value $k$.

$$\begin{aligned}
X[N] =& \{x[i, j, k] \mid i, j, k \in \{1, 2, \ldots, N\}\} \\
D[N] =& \{0, 1\} \\
C[N] =& \{\sum_{1 \leq k \leq N} x[i, j, k] = 1 \mid i, j \in \{1, 2, \ldots, N\}\} \cup \\
& \{\sum_{1 \leq j \leq N} x[i, j, k] = 1 \mid i, k \in \{1, 2, \ldots, N\}\} \cup \\
& \{\sum_{1 \leq i \leq N} x[i, j, k] = 1 \mid j, k \in \{1, 2, \ldots, N\}\}.
\end{aligned}$$

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

Our implementation utilises the modelling language MiniZinc [7] and the rule-based term rewriting language Cadmium [2]. MiniZinc allows the specification of

parametrised models and supports quantification of constraints over parameters. These languages are explained briefly in Section 4.

*Example 2.* The following is the Latin Square model from Example 1 written in MiniZinc.

```
set of int: range = 1..N;

array[range, range, range] of var 0..1: x;

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

□

## 3 Proving Symmetries by Model Transformation

Motivated by the kinds of matrix model permutations investigated in [5] (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 symmetries. Specifically, given a common matrix permutation $\sigma$ on the variables of a model $M$, we prove that $\sigma$ is a symmetry of $M$ by showing that $\sigma(M)$ can be rewritten to a semantically equivalent form that is syntactically equal to $M$. In other words, the symmetry $\sigma$ is applied to $M$, and then $\sigma(M)$ is shown to be equivalent to $M$, which proves that $\sigma$ is a model symmetry.

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

1. Compute $\sigma(c)$ for each constraint $c \in C$, giving a set $C' = \{\sigma(c) | c \in C\}$.
2. Normalise every constraint $c \in C$ and every constraint $c' \in C'$ as follows:
   (a) Reduce the expressions used as array indices to single variables by substitution.
   (b) Simplify arithmetic expressions.
   (c) Simplify quantifications.
   (d) Reorder consecutive quantifications.
3. Find a one-to-one matching between the constraints in $C$ and the constraints in $C'$.

We now describe each of these steps in detail.

### 3.1 Computing $\sigma(c)$

We consider the following three types of permutations acting on a matrix of variables:

- 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 $\varphi$ 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 $\varphi$ 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 correspond to symmetries considered by Mears et al. [5] and occur commonly in matrix models. The constraint $\sigma(c)$ is constructed 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 3.* One of the constraints of the Latin square problem (Example 1) is:

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

where $R = \{1, 2, \ldots, N\}$. Let $\sigma$ be the symmetry that swaps dimensions 1 and 2:

$$x[i, j, k] \mapsto x[j, i, k]$$

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

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

□

### 3.2 Substituting complex expressions

The goal of this and the next 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 $j < k$.

In this step we introduce variables $i_j$ that will ultimately take the place of the expressions $e_j$. We assume that an expression $e_j$ is an arithmetic expression built from addition and subtraction and only involves one quantified variable $q_j$. Thus, there is a corresponding linear function $f$ such that $e_j = f(q_j)$.

If $e_j$ is a constant, nothing needs to be done and $i_j = e_j$. Otherwise, $e_j$ is a function of a quantified variable $q_j$ and therefore $e_j = f(q_j)$ for some linear function $f$. 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 $f^{-1}(i_j)$. As a result, every $e_j = f(q_j)$ becomes $e_j = f(f^{-1}(i_j)) = i_j$.

We assume that 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 4.* Consider again one of the constraints of the Latin square problem (Example 1):

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

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

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

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

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

Let us now substitute the first expression in the array access. We introduce a new variable $\alpha = N - i + 1$. We see that $\alpha$ 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 $N - \alpha + 1$, giving:

$$(\forall (N - \alpha + 1), j \in R) \sum_{k \in R} x[N - (N - \alpha + 1) + 1, j, k] = 1$$

□

### 3.3 Simplifying expressions

In this step basic arithmetic and permutation simplifications are performed. These are (where $\varphi$ is an arbitrary permutation):

- $a - b \Rightarrow a + (-b)$
- $-(a + b) \Rightarrow (-a) + (-b)$
- $-(-(a)) \Rightarrow a$
- $a + -(a) \Rightarrow 0$
- $-(0) \Rightarrow 0$
- $a + 0 \Rightarrow a$
- $a - b = c \Rightarrow b - a = -c$ (if $b <_{\text{lex}} a$)
- $\varphi(x) \neq \varphi(y) \Rightarrow x \neq y$
- $\varphi(\varphi^{-1}(a)) \Rightarrow a$
- $\varphi^{-1}(\varphi(a)) \Rightarrow a$
- $\text{alldifferent}(\varphi(a)) \Rightarrow \text{alldifferent}(a)$
- $\text{alldisjoint}(\varphi(a)) \Rightarrow \text{alldisjoint}(a)$
- $\text{card}(\varphi(a)) \Rightarrow \text{card}(a)$
- $a \in \varphi(b) \Rightarrow \varphi^{-1}(a) \in b$

In addition, any terms of the form $\varphi(a)$ are substituted (à la Section 3.2) so that the $\varphi$ permutations appear only in the quantification heads to be dealt with in the next section.

*Example 5.* In Example 4 we ended with the constraint:

$$(\forall(N-\alpha+1), j \in R) \sum_{k \in R} x[N-(N-\alpha+1)+1, j, k] = 1$$

This is simplified to:

$$(\forall(N-\alpha+1), j \in R) \sum_{k \in R} x[\alpha, j, k] = 1$$

As a result, all indices in the array access are single variables. □

### 3.4 Simplifying quantifications

The naive substitutions of the previous steps may leave quantifications in an ambiguous state where the object being quantified is not an isolated variable. In this step we attempt to repair these quantifications.

Consider a quantification $\forall(f(i) \in R)$ where $i$ is a variable, $f$ a function and $R$ is a set. We interpret this as meaning $\{f(i)|f(i) \in R\}$, i.e. the set of all values $f(i)$ that lie within $R$. For particular functions $f$, such quantifications can be simplified into an unambiguous form. We apply transformations for two kinds of function.

- ⋆ $\forall((d_j - i + 1) \in \{1, 2, \ldots, d_j\}) \rightmodels \forall(i \in \{1, 2, \ldots, d_j\})$.
- ⋆ $\forall(\varphi(i) \in R) \rightmodels \forall(i \in R)$ where $\varphi$ is a permutation on $R$.

(Note that the first is a special case of the second.)

*Example 6.* In Example 5 we ended with the constraint:

$$(\forall(N-\alpha+1), j \in R) \sum_{k \in R} x[\alpha, j, k] = 1$$

The quantification is simplified so that the constraint becomes:

$$(\forall\alpha, j \in R) \sum_{k \in R} x[\alpha, j, k] = 1$$

since $R = \{1, 2, \ldots, N\}$. □

### 3.5 Reordering of consecutive quantifications

The last step of constraint normalisation is to reorder consecutive quantifications. In this step, any two quantifications that appear immediately next to one another are reordered so that the names of the quantified variables are in lexicographical order.

- ⋆ $\forall j \in R_j, \forall i \in R_i \rightmodels \forall i \in R_i, \forall j \in R_j$ if $i$ is lexicographically less than $j$.

### 3.6 Matching constraints

The final step of the method is to match the constraints in $C$ with the constraints in $C'$. A constraint $c \in C$ is considered to match a constraint $c' \in C'$ if the two constraints are identical up to variable renaming. If there exists a constraint $c \in C$ for every $c' \in C'$, then the symmetry is deemed to hold on the model.

## 4 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 of interest for us 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:

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

where $R = \{1, 2, \ldots, N\}$.

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.

### 4.1 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 3.1.

```
% 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.

### 4.2 Substituting complex expressions

In this step we find the expressions used in array accesses and replace them with single variables. The first part is to 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 3.2, 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.

### 4.3 Simplifying expressions and quantifications

The arithmetic identities in Section 3.3 translate naturally into Cadmium rules:

```
% Arithmetic simplification rules.
X-Y <=> X + -(Y).
-(X+Y) <=> -(X) + -(Y).
-(-(X)) <=> X.
% Et cetera.
```

Likewise, the reordering of consecutive quantifications is simple (although it requires some knowledge of how MiniZinc quantifications appear as terms in Cadmium):

```
gen('$cc'(decl(int,Var1,no,VarKind1,A1),
    gen('$cc'(decl(int,Var2,no,VarKind2,A2),Body)))) <=>
        Var1 `$>` Var2 |
    gen('$cc'(decl(int,Var2,no,VarKind2,A2),
        gen('$cc'(decl(int,Var1,no,VarKind1,A1),Body)))).
```

A `gen` term represents a generator such as `(i in 1..n)`. Briefly, this rule states that if there are two consecutive generators over variables `Var1` and `Var2` respectively, and `Var2` comes lexicographically before `Var1`, then the order of the generators should be swapped.

### 4.4 Matching constraints

After all normalisation is done, the model has the set of normalised original constraints $C$, represented by `constraint_orig` terms, and the set of normalised symmetric constraints $C'$, represented by `constraint_sym` terms. In the final step we attempt to match each `constraint_sym` term with a `constraint_orig` term.

```
constraint_orig(ID1,C1) /\ constraint_sym(ID2,C2) <=>
        identical_up_to_renaming(C1,C2) | true.
```

If two constraints are identical up to renaming (as tested by a standard Cadmium rule), they are eliminated from the model. If at the end of execution no constraints remain in the model, the symmetry is deemed to hold. If there are any remaining constraints, they are the ones that could not be matched and help to explain why the symmetry could not be proved to hold.

## 5 Results

We have tested our model transformation approach for symmetries found by Mears et al. [5] in a suite of benchmark problems. Below are the MiniZinc models that we tested using our method, followed by Table 1 which lists, for each problem and symmetry, whether our implementation was able to show that the symmetry holds.

**Latin Square (integer).**

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

constraint forall (i in 1..n) (alldifferent (j in 1..n) (x[i,j]));
constraint forall (j in 1..n) (alldifferent (i in 1..n) (x[i,j]));
```

The variable `x[i,j]` being equal to `k` represents the cell at row `i` and column `j` having the value `k`.

**Steiner Triples.**

```
int: nb = n * (n-1) div 6;
array[1..nb,1..n] of var 0..1: x;

constraint forall (i in 1..nb)
  (sum (j in 1..n) (x[i,j])= 3);
constraint forall (i,j in 1..nb where i!=j)
  (sum (k in 1..n) (x[i,k] * x[j,k]) <= 1);
```

The variable `x[i,j]` being equal to 1 represents the element `j` belonging to the triple `i`.

**BIBD.**

```
int: v;
int: k;
int: lambda;
int: b = (lambda * v * (v - 1)) div (k * (k - 1));
int: r = (lambda * (v - 1)) div (k - 1);

array [1..v, 1..b] of var bool: m;

constraint forall (i in 1..v) (sum (j in 1..b) (bool2int(m[i, j])) = r);

constraint forall (j in 1..b) (sum (i in 1..v) (bool2int(m[i, j])) = k);

constraint forall (i1, i2 in 1..v where i1 != i2)
             (sum (j in 1..b) (bool2int(m[i1,j] /\ m[i2,j])) = lambda);
```

The variable `m[i,j]` being equal to 1 represents the object `i` belonging to block `j`.

**Social Golfers.**

```
int: n_groups;
int: n_per_group;
int: n_rounds;
int: n_golfers = n_groups * n_per_group;

set of int: groups = 1..n_groups;
set of int: group = 1..n_per_group;
set of int: rounds = 1..n_rounds;
set of int: golfers = 1..n_golfers;
```

```
array [rounds, groups] of var set of golfers: x :: is_output;

constraint forall (r in rounds, g in groups)
    (card(x[r, g]) = n_per_group);

constraint forall (r in rounds)
    (all_disjoint (g in groups) (x[r, g]));

constraint forall (a, b in golfers where a < b)
    (sum (r in rounds, g in groups)
        (bool2int(a in x[r, g] /\ b in x[r,g])) <= 1);
```

The variable `x[i,j]` is the set of players who are in Group `j` in Week `i`.

**N-Queens (Boolean).**

```
array[1..N,1..N] of var 0..1: x;

constraint forall (i in 1..N) (sum (j in 1..N) (x[i,j]) = 1);
constraint forall (j in 1..N) (sum (i in 1..N) (x[i,j]) = 1);
constraint forall (k in 2-N..N-2)
  (sum (i,j in 1..N where i-j= k) (x[i,j]) <= 1);
constraint forall (k in 3..N+N-1)
  (sum (i,j in 1..N where i+j= k) (x[i,j]) <= 1);
```

The variable `x[i,j]` being equal to 1 represents a queen being placed in square `(i,j)`.

**N-Queens (integer).**

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

constraint
    forall (i,j in 1..n where i != j) (
        x[i] != x[j] /\ x[i]+i != x[j]+j /\ x[i]-i != x[j]-j
    );
```

The variable `x[i]` taking the value `j` means that the queen in column `i` is in row `j`.

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.

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 $\sigma(x[i], j) = (x[j], i)$ in $N$-Queens (integer). 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. Our implementation does not, as yet, handle models that contain arithmetic in "where" clauses e.g. Boolean $N$-Queens.

## 6 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 automated theorem provers. This paper provides a new way of proving the existence of model symmetries by way

**Table 1.** Summary of Symmetries Proved

| Problem | Variable Symmetries | Proved |
|---|---|---|
| Latin Squares (Boolean) | `x[i,j,k]` $\mapsto$ `x[j,i,k]` | Yes |
| | `x[i,j,k]` $\mapsto$ `x[i,k,j]` | Yes |
| | `x[i,j,k]` $\mapsto$ `x[N-i+1,j,k]` | Yes |
| | `x[i,j,k]` $\mapsto$ `x[`$\varphi$`(i),j,k]` | Yes |
| | `x[i,j,k]` $\mapsto$ `x[i,`$\varphi$`(j),k]` | Yes |
| | `x[i,j,k]` $\mapsto$ `x[i,j,`$\varphi$`(k)]` | Yes |
| Latin Squares (integer) | `x[i,j]` $\mapsto$ `x[j,i]` | Yes |
| | `x[i,j]` $\mapsto$ `x[`$\varphi$`(i),j]` | Yes |
| | `x[i,j]` $\mapsto$ `x[i,`$\varphi$`(j)]` | Yes |
| | `x[i,j]` $\mapsto$ $\varphi$`(x[i,j])` | Yes |
| Steiner Triples | `x[i,j]` $\mapsto$ `x[`$\varphi$`(i),j]` | Yes |
| | `x[i,j]` $\mapsto$ `x[i,`$\varphi$`(j)]` | Yes |
| BIBD | `x[i,j]` $\mapsto$ `x[`$\varphi$`(i),j]` | Yes |
| | `x[i,j]` $\mapsto$ `x[i,`$\varphi$`(j)]` | Yes |
| Social Golfers | `x[i,j]` $\mapsto$ `x[`$\varphi$`(i),j]` | Yes |
| | `x[i,j]` $\mapsto$ `x[i,`$\varphi$`(j)]` | Yes |
| | `x[i,j]` $\mapsto$ $\varphi$`(x[i,j])` | Yes |
| $N$-Queens (Boolean) | `x[i,j]` $\mapsto$ `x[j,i]` | No |
| | `x[i,j]` $\mapsto$ `x[N-i+1,j]` | No |
| $N$-Queens (integer) | `x[i]` $\mapsto$ `x[N-i+1]` | Yes |
| | `x[i]` $\mapsto$ `N-x[i]+1` | Yes |

of model transformations on parametrised matrix models. 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.

Our implementation is at present somewhat ad hoc and cannot yet deal with general arithemetic expressions e.g. Boolean $N$-queens. We are currently extending our implementation to deal more generally with matrix models.

## 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.

# Bibliography


[1] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M. Smith. Symmetry definitions for constraint satisfaction problems. In *Principles and Practice of Constraint Programming - CP 2005*, 2005.

[2] G.J. Duck, L. De Koninck, and P.J. Stuckey. Cadmium: An implementation of ACD term rewriting. In *Logic Programming - ICLP 2008*, 2008.

[3] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.

[4] C. Mears. *Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming*. PhD thesis, Monash University, 2009.

[5] C. Mears, M. Garcia de la Banda, M. Wallace, and B. Demoen. 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.

[6] C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. *Constraints*, 14, 2009.

[7] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007*, 2007.

[8] J.-F. Puget. Automatic detection of variable and value symmetries. In *Principles and Practice of Constraint Programming - CP 2005*, 2005.

[9] P. Roy and F. Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 Workshop on Nonbinary Constraints*, 1998.

[10] P. Van Hentenryck, P. Flener, J. Pearson, and M. Agren. Compositional derivation of symmetries for constraint satisfaction. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.