Noname manuscript No.
(will be inserted by the editor)

A Method for Detecting Symmetries in Constraint Models and its Generalisation

Christopher Mears · Maria Garcia de la Banda · Mark Wallace · Bart Demoen

the date of receipt and acceptance should be inserted later

Abstract The symmetries that appear in many constraint problems can be used to significantly speed up the search for solutions to these problems. While the accurate detection of symmetries in instances of a given constraint problem is possible, current methods tend to be impractical for real-sized instances. On the other hand, methods capable of detecting properties for a problem model – and thus all its instances – are efficient but not accurate enough. This paper presents a new method for inferring symmetries in constraint satisfaction models that combines the high accuracy of instance-based methods with the efficiency of model-based methods; the key insight is that symmetries detected for small instances of the model can be generalised to the model itself. Experimental evaluation shows that this approach is able to find all symmetries in almost all the benchmark problems used. The generality of our method is then illustrated by showing how it can be applied to infer other properties.

1 Introduction

Many constraint satisfaction and optimisation problems have symmetries, which can be used to significantly shorten the search time of traditional tree-search algorithms by applying symmetry breaking techniques [8, 14, 10]. These techniques avoid searching symmetric parts of the search space, yet still find all non-symmetric solutions.

The potential gains of symmetry breaking have motivated interest in analysis methods that can detect symmetries in a constraint problem (e.g. [11, 2, 30, 27]). Most of these symmetry detection methods are run for every constraint problem (or *instance*) even if the

This research was partly sponsored by the Australian Research Council through grant DP110102258, and by the Research Foundation Flanders (FWO) through projects *WOG: Declarative Methods in Computer Science* and *G.0221.07*.

Christopher Mears · Maria Garcia de la Banda · Mark Wallace
E-mail: {chris.mears / maria.garciadelabanda / mark.wallace}@monash.edu
Monash University, Australia

Bart Demoen
E-mail: bart.demoen@cs.kuleuven.be
KU Leuven, Belgium

instances have the same *model* and only differ in the particular data used as input. The instance data is used during the detection process to increase the accuracy of the detection methods and, therefore, the detected symmetries apply only to that instance. As a result, while instance-based symmetry detection methods can be very accurate, their detection process is often so costly that it nullifies or even outweighs the savings achieved by exploiting the detected symmetries. This usually makes instance-based symmetry detection methods impractical for large instances.

The time taken to detect symmetries is easier to offset if it is performed only once for a given problem model in such a way that the detected symmetries apply to all instances of that model; this way, we no longer need to analyse each instance individually. While this approach misses any symmetry that applies to some – but not all – instances of the model, it makes symmetry detection practical. Unfortunately, analysing the problem model without the instance data often results in a substantial loss of accuracy. Furthermore, current model-based symmetry detection methods either strongly depend on the constraints used to specify the problem model to gain accuracy [36], or require manual intervention by the user [30, 18].

We propose a new symmetry detection method that combines the high accuracy of instance-based detection methods with the performance advantages of the model-based methods. Figure 1 provides an overview of the method, which has the following steps: (1) use accurate instance-based methods to detect the symmetries of several small instances of the model, (2) lift these instance symmetries to the model level, obtaining a set of candidate model symmetries, (3) filter these candidates to eliminate those that clearly do not hold for the model, thus obtaining a set of likely candidates, and (4) prove that these likely candidates are symmetries of the model. These steps correspond to an inductive reasoning process, where the fourth step attempts to formally prove the veracity of the induced symmetries. Thus, it is similar to, for example, the approach of Charnley et al. [3] where inductive reasoning is used to find implied constraints and a theorem prover is then used to prove that the induced constraints hold. On a common set of benchmarks, our current (relatively ad hoc) implementation of the method is capable of detecting all symmetries as likely candidates in all but two benchmarks. As this paper shows, our method can also be used for inferring other model properties, such as subproblem equivalence.

The main contributions of this paper are as follows. First, we introduce a new method for accurately detecting symmetries in constraint models. For this we provide (a) an abstract definition of a symmetry acting on a model that can be handled algorithmically — called a *parameterised symmetry*, — given in Definition 1, and an extension of this definition to sets of symmetries acting on a model, given in Definition 2, (b) a simple and effective (although ad hoc) method of lifting symmetries from instances to models, given in Section 4.2, and (c) a method for combining the symmetries found for several instances and recovering missing model symmetries. Second, we present an experimental evaluation on a set of benchmarks showing that our current implementation is practical and accurate (Section 7). And third, we offer a generalisation of our symmetry detection method to detect other properties of constraint models (Section 8).

This paper extends and revises earlier work [23] by generalising the method to other properties, providing a more extensive experimental evaluation, and providing a much more detailed explanation and formalisation of the patterns and algorithms used for lifting symmetries from instances to models. The paper is organised as follows. Section 2 provides the necessary background on constraint satisfaction problems, their symmetries and problem models. Section 3 defines model symmetries. Section 4 presents the new symmetry detection method and describe its steps in detail. Section 5 discusses the limitations of the method. Section 6 illustrates our implementation via several examples. Section 7 presents and dis-

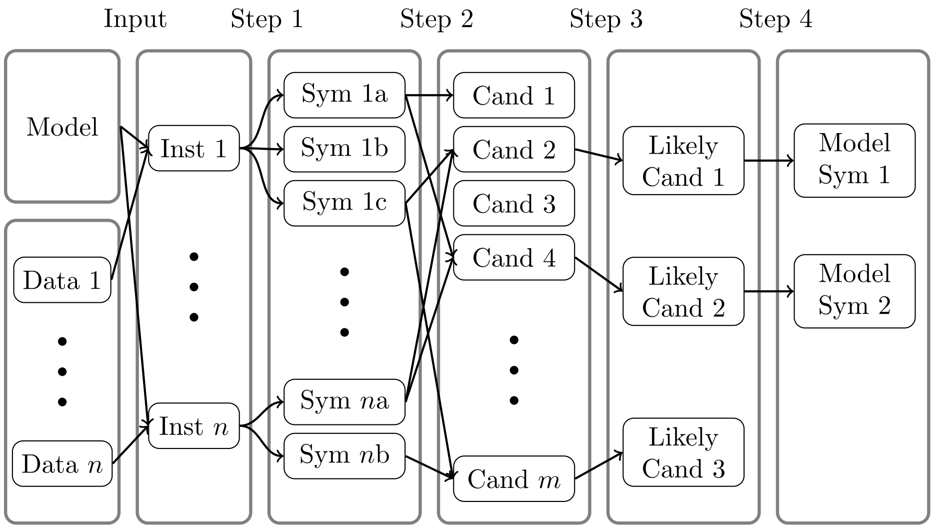

Fig. 1 Method overview.

cusses an experimental evaluation of the method. Section 8 presents a generalised form of the method. Section 9 discusses related work. Finally, Section 10 concludes the paper.

2 Background

2.1 Constraint Satisfaction Problems (CSPs)

Let $vars(O)$ denote the set of variables of object O. A constraint satisfaction problem (CSP) is a tuple (X,D,C) where X is a set of variables, D is a function that maps each variable in X to its domain (a finite set of values), and C is a set of constraints such that $vars(C) \subseteq X$. For brevity, we write $D = \{a_1,\ldots,a_n\}$ when all variables in X have the same domain $\{a_1,\ldots,a_n\}$, and we denote the set of consecutive integers $\{i,i+1,i+2,\ldots,j\}$ by $i..j$.

Consider the CSP $P = (X,D,C)$. A *literal* of P is a pair (x,d), written as $x = d$, where $x \in X$ and $d \in D(x)$. We denote the set of all literals of P by $lit(P)$. An *assignment of P over* $V \subseteq X$ is a subset of $lit(P)$ with exactly one literal per variable in V. A constraint $c \in C$ is a set of assignments over $vars(c) \subseteq X$, usually denoted by a formula. An assignment A of P over $V \subseteq X$ satisfies c if and only if both $vars(c) \subseteq V$ and the projection of A over $vars(c)$ (defined as $\{(x = d) \mid (x = d) \in A \wedge x \in vars(c)\}$) is a member of c. Finally, a *solution* of P is an assignment of P over X that satisfies every constraint in C. Where the identity of P is clear, we will omit the “of P” part.

Example 1 Consider the Latin square problem of size N whose aim is to find an $N \times N$ matrix of values from 1 to N such that each value occurs exactly once in each row and exactly once in each column. The Latin square problem of size 3 can be represented as a CSP with 9 integer[1] variables $\{x_{ij} \mid i,j \in 1..3\}$ where x_{ij} represents the cell in row i and

[1] In general the Latin square problem does not require the values in the cells to be integers. We have used integers as they are well supported by constraint solvers.

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

1	3	2
2	1	3
3	2	1

1	2	3
3	1	2
2	3	1

Fig. 2 Variables in the CSP of the Latin square problem of size 3 and two possible solutions.

column j, as shown in the left hand side of Figure 2. The domain of each variable is 1..3 and C contains 18 disequality constraints that ensure each value occurs exactly once in each row and exactly once in each column. Formally, the CSP is defined as $P = (X, D, C)$ where:

$$\begin{aligned}
X = \ & \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\
D = \ & 1..3 \\
C = \ & \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, x_{22} \neq x_{23}, \\
& x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, x_{11} \neq x_{31}, x_{21} \neq x_{31}, \\
& x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\}
\end{aligned}$$

Assignment $\{x_{11} = 1, x_{21} = 3\}$ satisfies constraint $x_{11} \neq x_{21}$, while assignment $\{x_{11} = 1, x_{21} = 1\}$ does not. Assignment $\{x_{11} = 1, x_{21} = 3, x_{31} = 1\}$ satisfies $x_{11} \neq x_{21}$ but does not satisfy $x_{11} \neq x_{31}$. Figure 2 also shows two possible solutions of P. □

For simplicity, we focus on satisfaction problems. However, the results can be extended to optimisation problems by representing the optimisation function as an additional constraint. To be precise, an optimisation function $opt(x_1, x_2, \ldots, x_n)$ is treated (for symmetry detection purposes) as the constraint $z = opt(x_1, x_2, \ldots, x_n)$, where z is a new variable.

2.2 Symmetries of a CSP

A *solution symmetry* of a CSP $P = (X, D, C)$ is a permutation on $lit(P)$ that preserves the set of solutions of P [5]. Two important kinds of solution symmetries are induced by permuting either the variables in X or the values in the range of D.

A permutation f on X induces a permutation p_f on $lit(P)$ defined as $p_f(x = d) = (f(x) = d)$, where $x \in X$ and $d \in D(x)$, if $\forall x : D(x) = D(f(x))$. A *variable symmetry* is a permutation on X whose induced permutation on $lit(P)$ is a solution symmetry [26]. A set of value permutations f_i, one on each $D(x_i), x_i \in X$, induces a permutation p_{f_i} on $lit(P)$ by defining $p_{f_i}(x_i = d) = (x_i = f_i(d))$. A *value symmetry* is a set of value permutations whose induced permutation on $lit(P)$ is a solution symmetry [26]. We will use $\langle xd_1, \ldots, xd_n \rangle \leftrightarrow \langle xd_{1'}, \ldots, xd_{n'} \rangle$, where $xd_1, \ldots, xd_n, xd_{1'}, \ldots, xd_{n'}$ are either variables in X or values in the range of D, to denote a permutation that maps each xd_i to $xd_{i'}$, and vice versa, leaving the unmentioned variables or values unchanged. As a special case, when all permutations of a set of variables (resp. values) are solution symmetries, we say that the set of variables (values) is *interchangeable* [11].

A *variable-value symmetry* is any solution symmetry that is not a variable symmetry or a value symmetry. Note that a variable-value symmetry is not necessarily a composition of variable and value symmetries.

Example 2 The CSP given in Example 1 to represent the Latin square problem of size 3 has, among others, the following symmetries:

- Value symmetries that swap any two values: $\langle 1\rangle \leftrightarrow \langle 2\rangle$, $\langle 1\rangle \leftrightarrow \langle 3\rangle$, and $\langle 2\rangle \leftrightarrow \langle 3\rangle$ (therefore all three values are interchangeable)
- Variable symmetries that swap any two columns in the square:
$\langle x_{11},x_{21},x_{31}\rangle \leftrightarrow \langle x_{12},x_{22},x_{32}\rangle$, $\langle x_{11},x_{21},x_{31}\rangle \leftrightarrow \langle x_{13},x_{23},x_{33}\rangle$, and
$\langle x_{12},x_{22},x_{32}\rangle \leftrightarrow \langle x_{13},x_{23},x_{33}\rangle$
- Similar variable symmetries that swap any two rows in the square.
- Variable symmetries across a diagonal: $(x_{ij} = k) \mapsto (x_{ji} = k), \forall i, j, k \in 1..3$; this symmetry (which corresponds to a reflection of the board) is shown in Figure 2 mapping one solution to another.
- Variable-value symmetries that swap the rows (or columns) with the values, e.g. the symmetry $(x_{ij} = k) \mapsto (x_{ik} = j), \forall i, j, k \in 1..3$. Note that these symmetries are not compositions of any variable and value symmetries.

Other common symmetries, such as board rotations, can be obtained by composing those mentioned above. □

A permutation group is a set of permutations that is closed under composition and inverses. The solution symmetries of a CSP P form a permutation group, where each element is a permutation on the set of literals $lit(P)$. Given permutations $\{f_1, f_2, \ldots, f_n\}$ on $lit(P)$, we denote by $[f_1, f_2, \ldots, f_n]$ the closure under composition and inverses of $\{f_1, f_2, \ldots, f_n\}$. Given a permutation group G, if $[f_1, f_2, \ldots, f_n] = G$ then $\{f_1, f_2, \ldots, f_n\}$ is called a *generating set* of G and is said to *generate* G. A generating set is *minimal* if any proper (i.e. strictly smaller) subset of the generating set generates a proper subgroup.

Example 3 The symmetries of the CSP given in Example 1 to represent the Latin square problem of size 3 form a group G of cardinality $6(3!)^3 = 1296$ that can be generated by, among others, the following four symmetries:

- Swap rows 1 and 2: $\langle x_{11},x_{12},x_{13}\rangle \leftrightarrow \langle x_{21},x_{22},x_{23}\rangle$.
- Swap rows 2 and 3: $\langle x_{21},x_{22},x_{23}\rangle \leftrightarrow \langle x_{31},x_{32},x_{33}\rangle$.
- Reflect the square diagonally ($x_{ij} = k \mapsto x_{ji} = k$, $\forall i, j, k \in 1..3$).
- Swap the row index with the value ($x_{ij} = k \mapsto x_{kj} = i$, $\forall i, j, k \in 1..3$).

Therefore, these four symmetries form a generating set for G. The generating set is minimal since any proper subset is not a generating set for G. □

2.3 Detecting symmetries of a CSP via graph automorphism

A hyper-graph is a pair (V,E) where V is a set of vertices and E is a set of hyper-edges, each of which is a non-empty subset of V. An *automorphism* (or symmetry) f of a hyper-graph (V,E) is a permutation of V that preserves E, i.e. a permutation such that $\forall\{v_i, \ldots, v_j\} \in E$: $\{f(v_i), \ldots, f(v_j)\} \in E$.

Several methods have been defined for automatically finding the symmetries of a CSP P by representing it as a (hyper-)graph G in such a way that each automorphism of G corresponds to a solution symmetry of P (see, for example, [27, 29, 5, 21]). The main difference among these methods is in how the elements of P are mapped to the vertices and hyper-edges of G.

In this paper we will use the full assignments graph representation defined in [21]. Briefly, the full assignments graph is built from a given CSP $P = (X,D,C)$ by (a) representing every literal in $lit(P)$ as a vertex; (b) representing every constraint $c \in C$ by a set

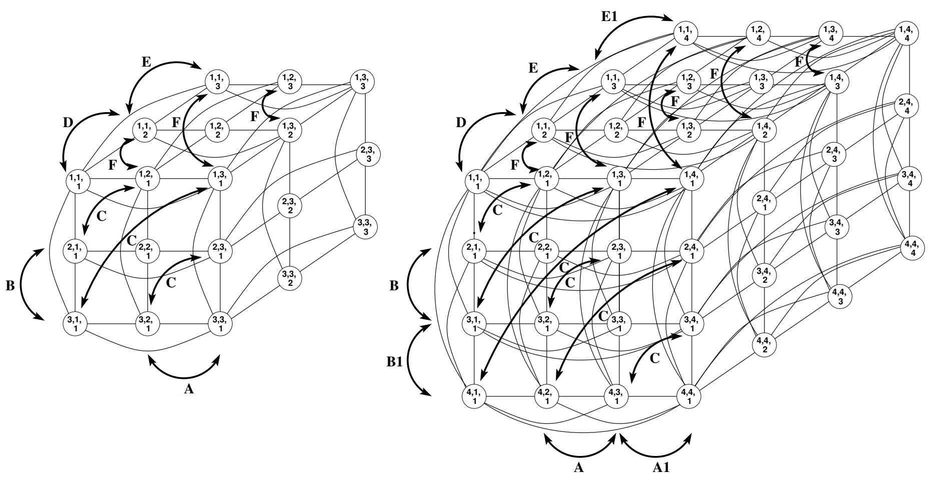

Fig. 3 Full assignments graphs and generating sets for LatinSquare[3] and LatinSquare[4]. Note that parts of the graph are omitted for legibility.

of hyper-edges: either a hyper-edge for every assignment that does not satisfy c or a hyper-edge for every assignment that satisfies c; and (c) adding an edge between every two literals that assign different values to the same variable. The choice of whether to use satisfying or unsatisfying assignments can be made independently for each constraint (often choosing the one that would result in the least amount of edges).

Example 4 The full assignments graph for the CSP given in Example 1 to represent the Latin square problem of size 3 is shown in the left hand side of Figure 3. The $9 \times 3 = 27$ literals in the instance $x_{ij} = k$, where $i, j, k \in 1..3$, are represented by the 27 vertices in the graph, each labelled $\mathbf{x}_{ijk}$, where the $\mathbf{x}$ has been omitted in the graph for clarity. The graph also has (18×3) edges representing the 3 assignments that do not satisfy each of the 18 constraints, plus (9×3) edges connecting the 3 different values of each of the 9 variables. The front-most face of the cube contains the vertices representing the literals that assign value 1 to each variable, while the middle slice assigns value 2 and the back face assigns value 3. Each variable can be seen as a line formed by the three vertices perpendicular to the front-most face. Note that much of the graph is omitted for the purpose of legibility. The bold arrows correspond to the elements of a generating set of symmetries and will be described later. $\square$

We chose to use the full assignments graph representation in our implementation because it is relatively simple and often more powerful than that of Puget [27] without being as computationally demanding as that of Cohen et al [5]. However, our method can use any graph representation whose automorphisms correspond to the symmetries of the CSP[2].

Once the CSP is represented as a graph, standard tools such as Saucy [9] can be used to compute the automorphisms of the graph and return its symmetry group via a generating set. Note that such tools may return any generating set, including a non-minimal one.

[2] Further, as we will see in Section 4, our method can use any instance-based method capable of inferring a generating set of symmetries for a given CSP. We focused on graph-based methods because they are currently the most accurate ones.

Example 5 Consider the graph built in Example 4 for the CSP representing the Latin square problem of size 3 and recall that vertex $\mathbf{x}_{ijk}$ represents literal $x_{ij} = k$. The automorphism detection tool Saucy returns the generating set $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$, where the elements (illustrated as bold arrows in the left hand side of Figure 3) are as follows:

A $\langle \mathbf{x}_{121}, \mathbf{x}_{122}, \mathbf{x}_{123}, \mathbf{x}_{221}, \mathbf{x}_{222}, \mathbf{x}_{223}, \mathbf{x}_{321}, \mathbf{x}_{322}, \mathbf{x}_{323} \rangle \leftrightarrow$
$\langle \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{133}, \mathbf{x}_{231}, \mathbf{x}_{232}, \mathbf{x}_{233}, \mathbf{x}_{331}, \mathbf{x}_{332}, \mathbf{x}_{333} \rangle$
B $\langle \mathbf{x}_{211}, \mathbf{x}_{212}, \mathbf{x}_{213}, \mathbf{x}_{221}, \mathbf{x}_{222}, \mathbf{x}_{223}, \mathbf{x}_{231}, \mathbf{x}_{232}, \mathbf{x}_{233} \rangle \leftrightarrow$
$\langle \mathbf{x}_{311}, \mathbf{x}_{312}, \mathbf{x}_{313}, \mathbf{x}_{321}, \mathbf{x}_{322}, \mathbf{x}_{323}, \mathbf{x}_{331}, \mathbf{x}_{332}, \mathbf{x}_{333} \rangle$
C $\langle \mathbf{x}_{121}, \mathbf{x}_{122}, \mathbf{x}_{123}, \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{133}, \mathbf{x}_{231}, \mathbf{x}_{232}, \mathbf{x}_{233} \rangle \leftrightarrow$
$\langle \mathbf{x}_{211}, \mathbf{x}_{212}, \mathbf{x}_{213}, \mathbf{x}_{311}, \mathbf{x}_{312}, \mathbf{x}_{313}, \mathbf{x}_{321}, \mathbf{x}_{322}, \mathbf{x}_{323} \rangle$
D $\langle \mathbf{x}_{111}, \mathbf{x}_{121}, \mathbf{x}_{131}, \mathbf{x}_{211}, \mathbf{x}_{221}, \mathbf{x}_{231}, \mathbf{x}_{311}, \mathbf{x}_{321}, \mathbf{x}_{331} \rangle \leftrightarrow$
$\langle \mathbf{x}_{112}, \mathbf{x}_{122}, \mathbf{x}_{132}, \mathbf{x}_{212}, \mathbf{x}_{222}, \mathbf{x}_{232}, \mathbf{x}_{312}, \mathbf{x}_{322}, \mathbf{x}_{332} \rangle$
E $\langle \mathbf{x}_{112}, \mathbf{x}_{122}, \mathbf{x}_{132}, \mathbf{x}_{212}, \mathbf{x}_{222}, \mathbf{x}_{232}, \mathbf{x}_{312}, \mathbf{x}_{322}, \mathbf{x}_{333} \rangle \leftrightarrow$
$\langle \mathbf{x}_{113}, \mathbf{x}_{123}, \mathbf{x}_{133}, \mathbf{x}_{213}, \mathbf{x}_{223}, \mathbf{x}_{233}, \mathbf{x}_{313}, \mathbf{x}_{323}, \mathbf{x}_{333} \rangle$
F $\langle \mathbf{x}_{112}, \mathbf{x}_{113}, \mathbf{x}_{123}, \mathbf{x}_{212}, \mathbf{x}_{213}, \mathbf{x}_{223}, \mathbf{x}_{312}, \mathbf{x}_{313}, \mathbf{x}_{323} \rangle \leftrightarrow$
$\langle \mathbf{x}_{121}, \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{221}, \mathbf{x}_{231}, \mathbf{x}_{232}, \mathbf{x}_{321}, \mathbf{x}_{331}, \mathbf{x}_{332} \rangle$

Symmetry **A** has the effect of swapping the variables in columns 2 and 3 of the Latin square, since it maps literal $x_{i2} = k$ to $x_{i3} = k$, and vice versa. Similarly, **B** swaps those in rows 2 and 3, **C** reflects the variables in the square across the top-left/bottom-right diagonal, **D** swaps values 1 and 2, **E** swaps values 2 and 3, and **F** swaps the second dimension of the cube with the value dimension (i.e. it maps literal $x_{ij} = k$ to $x_{ik} = j$). Their combination results in a group that contains the symmetries detailed in Example 2, e.g. to swap columns 1 and 2 of the Latin square (symmetry $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32} \rangle$ when defined in terms of the variables, and $\langle \mathbf{x}_{111}, \mathbf{x}_{211}, \mathbf{x}_{311}, \mathbf{x}_{112}, \mathbf{x}_{212}, \mathbf{x}_{312}, \mathbf{x}_{113}, \mathbf{x}_{213}, \mathbf{x}_{313} \rangle \leftrightarrow$
$\langle \mathbf{x}_{121}, \mathbf{x}_{221}, \mathbf{x}_{321}, \mathbf{x}_{122}, \mathbf{x}_{222}, \mathbf{x}_{322}, \mathbf{x}_{123}, \mathbf{x}_{223}, \mathbf{x}_{323} \rangle$ when defined in terms of the literals) one can apply first **F**, then **D**, and then **F** again. Note that the generating set returned by Saucy is not minimal, since the set $\{\mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$ generates the same symmetry group.

Consider now the right hand side of Figure 3, which represents the Latin square problem of size 4. Saucy returns the generating set $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{A1}, \mathbf{B1}, \mathbf{E1}\}$ where the first six symmetries are simple extensions of those found for size 3. For example, the symmetry **A** obtained for size 3 extends to size 4 as:

A $\langle \mathbf{x}_{121}, \mathbf{x}_{122}, \mathbf{x}_{123}, \mathbf{x}_{124}, \mathbf{x}_{221}, \mathbf{x}_{222}, \ldots, \mathbf{x}_{321}, \ldots, \mathbf{x}_{421}, \ldots \rangle \leftrightarrow$
$\langle \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{133}, \mathbf{x}_{134}, \mathbf{x}_{231}, \mathbf{x}_{232}, \ldots, \mathbf{x}_{331}, \ldots, \mathbf{x}_{431}, \ldots \rangle$

and similarly for **B, C, D, E** and **F**. The other three are defined as:

A1 $\langle \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{133}, \mathbf{x}_{134}, \mathbf{x}_{231}, \mathbf{x}_{232}, \ldots, \mathbf{x}_{331}, \ldots, \mathbf{x}_{431}, \ldots \rangle \leftrightarrow$
$\langle \mathbf{x}_{141}, \mathbf{x}_{142}, \mathbf{x}_{143}, \mathbf{x}_{144}, \mathbf{x}_{241}, \mathbf{x}_{242}, \ldots, \mathbf{x}_{341}, \ldots, \mathbf{x}_{441}, \ldots \rangle$
B1 $\langle \mathbf{x}_{311}, \mathbf{x}_{312}, \mathbf{x}_{313}, \mathbf{x}_{314}, \mathbf{x}_{321}, \mathbf{x}_{322}, \ldots, \mathbf{x}_{331}, \ldots, \mathbf{x}_{341}, \ldots \rangle \leftrightarrow$
$\langle \mathbf{x}_{411}, \mathbf{x}_{412}, \mathbf{x}_{413}, \mathbf{x}_{414}, \mathbf{x}_{421}, \mathbf{x}_{422}, \ldots, \mathbf{x}_{431}, \ldots, \mathbf{x}_{441}, \ldots \rangle$
E1 $\langle \mathbf{x}_{113}, \mathbf{x}_{123}, \mathbf{x}_{133}, \mathbf{x}_{143}, \mathbf{x}_{213}, \mathbf{x}_{223}, \ldots, \mathbf{x}_{313}, \ldots, \mathbf{x}_{413}, \ldots \rangle \leftrightarrow$
$\langle \mathbf{x}_{114}, \mathbf{x}_{124}, \mathbf{x}_{134}, \mathbf{x}_{144}, \mathbf{x}_{214}, \mathbf{x}_{224}, \ldots, \mathbf{x}_{314}, \ldots, \mathbf{x}_{414}, \ldots \rangle$ □

A1 swaps columns 3 and 4, **B1** swaps rows 3 and 4, and **E1** swaps values 3 and 4. □

3 From instance symmetries to model symmetries

3.1 CSP instances and CSP models

Our method distinguishes between a problem model (or CSP model) and a problem instance (or CSP instance), something for which there is no standard, formal notation even though the concepts are present in many CP languages, such as MiniZinc [25]. For the purpose of this paper, it is enough to define a CSP model as any specification that can be expressed as a MiniZinc program with at least one parameter whose value is not given, where a parameter can be an integer, a set of integers or a sequence of integers. A CSP instance of a CSP model is then defined as the result of extending the model by providing values to all its parameters. Therefore, a CSP instance also corresponds to a MiniZinc program, but one where all the parameters are given values. We base our definitions of model and instance data on MiniZinc because it is a well-defined language, well-known, reasonably expressive, and has a clear correspondence with the formal CSP notation previously (and commonly) used. In addition to this, MiniZinc has multi-dimensional arrays of variables and constants and supports iteration, two features that are used by our method.

A problem model can thus be seen as a CSP P parameterised by *Data*, written herein as $P[Data]$; i.e. $P[Data]$ is an indexed family of instances $P[d]$ for every index $d \in Data$.[3] This parameterisation also applies to its components, so $P[Data] = (X[Data], D[Data], C[Data])$, and a CSP instance $P[d]$ is then the tuple $(X[d], D[d], C[d])$. For the purposes of this paper, the parameter *Data* will be a tuple of integers.

Example 6 Consider the Latin square problem defined in Example 1. Its model can be written as a parameterised CSP LatinSquare$[N]$ as follows:

$$\begin{aligned} X[N] : N \mapsto \quad & \{x_{ij} \mid i,j \in 1..N\} \\ D[N] : N \mapsto \quad & 1..N \\ C[N] : N \mapsto \quad & \{x_{ij} \neq x_{ik} \mid i,j \in 1..N, k \in j+1..N\} \cup \\ & \{x_{ji} \neq x_{ki} \mid i,j \in 1..N, k \in j+1..N\} \end{aligned}$$

This model defines N^2 integer decision variables (all named x and distinguished by their subscripts i, j) with values in $1..N$ and two sets of constraints, the first ensuring that the values in each row are different, while the second does the same for the columns. The difference in the number of variables, values and constraints appearing in different instances of LatinSquare$[N]$ is simply a consequence of the different values given to N. Clearly, the CSP instance obtained by instantiating N to 3 (LatinSquare[3]) is the one given in Example 1. The corresponding MiniZinc program for this model is as follows:

```
int : N ;
array [1..N, 1..N] of var 1..N : x ;
constraint forall (i,j in 1..N, k in j+1..N) (x[i,j] != x[i,k]) ;
constraint forall (i,j in 1..N, k in j+1..N) (x[j,i] != x[k,i]) ;
```

This model defines N^2 integer decision variables `x[i,j]` with the same domain as before, and has two quantified constraints that mimic those in the parameterised CSP LatinSquare$[N]$ above. □

In the rest of the paper we will provide examples using parameterised CSP notation rather than MiniZinc notation, because we believe mathematical notation is more intuitive to readers unfamiliar with MiniZinc notation, and because it makes explicit the parameters of the CSP model that do not have values. However, we restrict our CSPs to those that are direct translations of MiniZinc models. The original MiniZinc models for all the examples in this paper are given in Appendix A.

[3] We describe $P[Data]$ as a family of instances indexed by *Data*, rather than an ordinary function from *Data* to instances, to emphasise that a model is usually viewed as a collection of instances.

Note that, for any problem model $P[Data]$ there is a fixed set of m variable names, which without loss of generality we can call $x^1, x^2, \ldots, x^m$ (or just x if $m = 1$, as for LatinSquare[N]). Each variable x^r has a fixed number of subscripts n_r and the set of x^r variables in any instance $P[d]$ of $P[Data]$ is determined by the ranges of its subscripts, which often depend on d (as subscripts i and j in LatinSquare[N] where determined by N). We can thus characterise the set of variables in any instance of a model by giving the range of each subscript of each variable. Further, since without loss of generality we can start each range at the lower bound of 1, the set of variables in each instance can be specified by just the upper bound of each subscript. The same applies to the variable's values.

3.2 Model symmetries

As stated in the introduction, our method requires us to lift the symmetries from instances to models, i.e. it requires us to describe the symmetries in terms of the model rather than as permutations of the literals in a particular instance of that model. This is possible because symmetries that occur across different instances of the same model often share a common structure, or pattern.

Example 7 Consider the LatinSquare[4] instance from the LatinSquare[N] model instantiated with $N = 4$. As discussed in Example 5, its associated full assignments graph is shown in the right hand side of Figure 3, together with the generating set found by the automorphism detection tool Saucy. The elements of this generating set are very similar to those found for $N = 3$ (left hand side of Figure 3). Six of them (**A**, **B**, **C**, **D**, **E**, and **F**) follow exactly the same pattern as those described in Example 2, even though they are not identical. For example, both symmetries denoted as **B** follow the pattern "swap rows 2 and 3", even though **B** maps literal $\mathbf{x}_{241}$ to literal $\mathbf{x}_{341}$ in LatinSquare[4] but not in LatinSquare[3], since those literals do not even exist in its graph. The other three symmetries for LatinSquare[4] (**A1**, **B1**, and **E1**) are new. □

In order to describe such patterns, we need to describe permutations in terms of the model's input parameters. To achieve this we introduce the notion of *parameterised* permutation. (See Fig. 4 for an overview of the relationship between permutations, symmetries and their parameterised versions.)

Definition 1 Given a model $P[Data]$, a parameterised permutation is an indexed family $f[Data]$ that maps each $d \in Data$ to a permutation of the literals in $lit(P[d])$. A model symmetry of $P[Data]$ is a parameterised permutation $f[Data]$ such that for all values $d \in Data$, $f[d]$ is a solution symmetry of $P[d]$.

To define a parameterised permutation of model $P[Data]$, we need to identify *literals* across its instances. To achieve this we will describe a literal as we did for vertices in the full assignments graph, that is, by using the name of its variable (in bold) subscripted by a tuple containing the indices specified in the model for the variable and the value assigned to it. The components of this tuple will be referred to as its *dimensions*. In this way, literals can be naturally arranged into one or more multi-dimensional matrices, each named by the variables occurring in its literals. The sizes (number of dimensions and number of elements in each dimension) of the matrices can be obtained by the function *Dims*, where for a given $d \in Data$:

$$Dims(d) = \langle x^1(d^1_1, d^1_2, \ldots, d^1_{n_1}), \ldots, x^m(d^m_1, d^m_2, \ldots, d^m_{n_m})\rangle$$

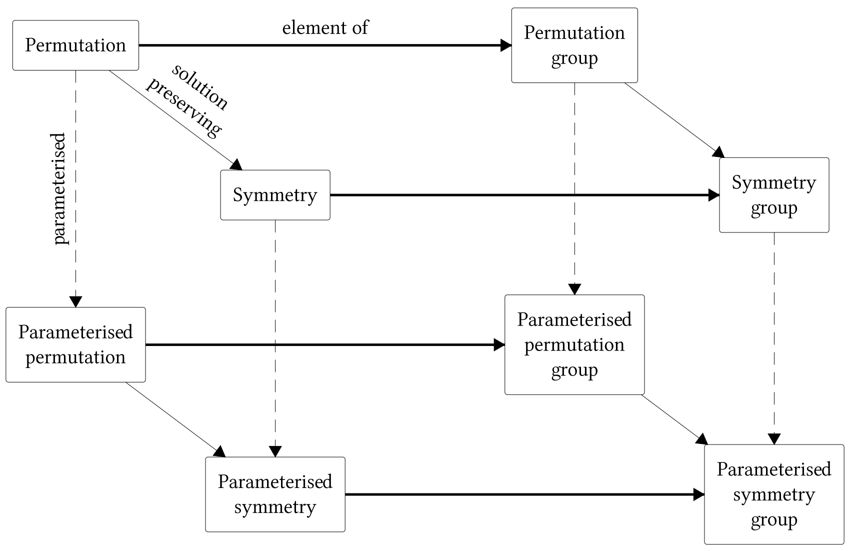

Fig. 4 Relationship between permutations and symmetries, and their parameterised versions.

where m is the number of matrices, x^r is the name of the r^{th} matrix, n_r is the number of dimensions in x^r and d^r_k is the number of elements in the k^{th} dimension of matrix x^r for $k \in 1..n_r, r \in 1..m$. Note that the number m of matrices and the number n_r of dimensions in each matrix, are fixed for a given model and do not depend on the data, since they are uniquely determined by the name of the variables and the associated indices given in the model. The only thing that changes is the number of elements in each dimension.

Example 8 The literals in any instance of the LatinSquare[N] model can be arranged into a single 3-dimensional matrix named $\mathbf{x}$ where literals are uniquely identified by $\mathbf{x}_{ijv}$, where $i, j, v \in 1..N$ (recall that, in the graphs of Figure 3, we eliminated the name $\mathbf{x}$ for clarity). Therefore, $\mathit{Dims}(N) = \langle \mathbf{x}(N,N,N) \rangle$. As is apparent in the figure, the only difference between the vertices of the graphs of LatinSquare[3] and LatinSquare[4] is the number of elements in each dimension of this matrix: $\mathit{Dims}(3) = \langle \mathbf{x}(3,3,3) \rangle$ while $\mathit{Dims}(4) = \langle \mathbf{x}(4,4,4) \rangle$. □

Example 9 Consider the Golomb ruler problem, where the task is to find a set of N integer marks on a ruler, such that the absolute values of the differences between distinct marks are different. The following model Golomb[N] has two sets of variables, one holding N integer variables (the marks) with domain $0..N^2$, and the other holding $\frac{N(N-1)}{2}$ integer variables (the differences) with domain $1..N^2$.

$$\begin{array}{ll}
X[N] : N \mapsto & \{m_i \mid i \in 1..N\} \cup \{d_{ij} \mid i, j \in 1..N, i < j\} \\
D[N] : N \mapsto & \{0..N^2\} \text{ for all } m_i \text{ and } \{1..N^2\} \text{ for all } d_{ij} \\
C[N] : N \mapsto & \{m_j - m_i = d_{ij} \mid i, j \in 1..N \wedge i < j\} \cup \\
& \{d_{ij} \neq d_{kl} \mid i, j, k, l \in 1..N \wedge i < j \wedge k < l \wedge (i \neq k \vee j \neq l)\}
\end{array}$$

The literals in every instance of the above model can be arranged into two matrices. The first matrix has the literals of the mark variables, which are identified by $\mathbf{m}_{ij}$, where $i \in 1..N, j \in 1..N^2+1$ such that $\mathbf{m}_{ij}$ represents $m_i = j-1$. The difference variables d_{ij} are flattened into a one-dimensional array with indices from 1 to $\frac{N(N-1)}{2}$. This flattening is performed because the d_{ij} variables in the model are only defined when $i < j$. Thus, the second matrix has the literals of the flattened array of difference variables, identified by $\mathbf{d}_{ij}$, where $i \in 1..\frac{N(N-1)}{2}, j \in 1..N^2$. Therefore, $Dims(N) = \langle \mathbf{m}(N, N^2+1), \mathbf{d}(\frac{N(N-1)}{2}, N^2)\rangle$. □

As we will see later (Section 4.2), the symmetries detected by our system are those that perform relatively simple permutations on these matrices, such as swapping rows or columns. Without loss of generality, we assume all variables of a given name have the same domain (otherwise, we can always use a domain computed by the union of their domains and modify the constraints accordingly).

Example 10 Consider a CSP model $P[N]$ with $Dims(N) = \langle \mathbf{x}(N, N, N)\rangle$. A parameterised permutation $f[N]$ to swap the first and second dimensions can be specified as $\mathbf{x}_{i_1 i_2 i_3} \mapsto \mathbf{x}_{i_2 i_1 i_3}$, $\forall i_1, i_2, i_3 \in 1..N$, where f maps an N equal to, say, 3 to the symmetry $\langle \mathbf{x}_{121}, \mathbf{x}_{122}, \mathbf{x}_{123}, \mathbf{x}_{131}, \mathbf{x}_{132}, \mathbf{x}_{133}, \mathbf{x}_{231}, \mathbf{x}_{232}, \mathbf{x}_{233}\rangle \leftrightarrow \langle \mathbf{x}_{211}, \mathbf{x}_{212}, \mathbf{x}_{213}, \mathbf{x}_{311}, \mathbf{x}_{312}, \mathbf{x}_{313}, \mathbf{x}_{321}, \mathbf{x}_{322}, \mathbf{x}_{323}\rangle$. A parameterised permutation to reflect the third dimension is specified as $\mathbf{x}_{i_1 i_2 i_3} \mapsto \mathbf{x}_{i_1 i_2 (N-i_3+1)}$, for $i_1, i_2, i_3 \in 1..N$. Note that the indices must remain within the bounds dictated by $Dims(N)$. □

The notion of parameterised permutation extends naturally to a *parameterised set of permutations*. Note that a parameterised set of permutations is not simply a set of parameterised permutations, as the size of the set may depend on the parameter. To simplify later explanation, we call a parameterised set of permutations a *pattern*.

Definition 2 Given a model $P[Data]$, a parameterised set of permutations (or pattern) is an indexed family $\overline{f}[Data]$ that maps each $d \in Data$ to a set of permutations of the literals in $lit(P[d])$. A parameterised set of symmetries of $P[Data]$ is a parameterised set of permutations $\overline{f}[Data]$ such that for all values $d \in Data$, every element in the set $\overline{f}[d]$ is a solution symmetry of $P[d]$.

Just as a set of permutations generates a permutation group, a parameterised set of permutations generates a parameterised permutation group. That is, a pattern $\overline{f}[Data]$ generates a parameterised permutation group $G[Data]$ such that for each possible parameter $d \in Data$, the group $G[d]$ is a permutation group of $lit(P[d])$.

Example 11 (Continued from Example 10.) Consider a pattern $\overline{f}[N]$ containing all adjacent swaps of the rows of LatinSquare$[N]$; that is, $\overline{f}[N] = \{f_{12}, f_{23}, \ldots, f_{(N-1)(N)}\}$ where f_{ij} swaps rows i and j. The pattern $\overline{f}[N]$ is a parameterised set of symmetries of LatinSquare$[N]$. It generates the parameterised permutation group $G[N]$, which for each instance is the group containing all $N!$ permutations of the rows. □

Note that the combination of model and data to create an instance often results in the addition of auxiliary variables during the transformation of a high-level MiniZinc model into a low-level instance, where constraints are flattened. These variables are (a) functionally dependent on the variables in the model, and (b) typically not used as search variables. If the latter, they are not needed when detecting symmetries. If so, we can simply eliminate auxiliary variables from the domain and codomain of the symmetry. If a symmetry maps a non-auxiliary variable to an auxiliary variable, then that symmetry can be discarded altogether. In fact, functionally dependent variables that are not used during the search might already be present in the model, as they are often used to propagate information. For example, in the Golomb ruler model of Example 9, the difference variables are functionally dependent on the mark variables and the search is often performed on the mark variables. Thus, the instance (and model) symmetries only need to consider the m_i variables.

4 A method for detecting model symmetries

Our approach to automatic model symmetry detection is based on a method that takes as input a model $P[Data]$ and a small set $V \subset Data$ of input values that result in small instances, and combines them to perform the following steps:

1. For each $d \in V$, detect a generating set $Generating_d$ of a symmetry group in instance $P[d]$,
2. For each $d \in V$, lift each $\sigma \in Generating_d$ to a parameterised permutation of $P[Data]$, obtaining the set of candidate patterns $Candidates_d$,
3. Create a subset *Likely* of $\bigcup_{d \in V} Candidates_d$ containing only those elements that are *likely* to be parameterised sets of symmetries of the model,
4. Prove whether the elements of *Likely* are indeed parameterised sets of symmetries of the model, or not.

Note that in the first step we would like $Generating_d$ to be small (for efficiency) and to generate the entire symmetry group of $P[d]$ (for accuracy). Also note that the third step is not strictly necessary but can make the proof step more efficient by eliminating candidates that are known not to hold for at least one instance. The rest of this section explains the above steps in detail, and describes how each of them is implemented in the system.

4.1 Step one: Detecting instance symmetries

In this step, our implementation of the method combines the MiniZinc model $P[Data]$ with each element $d \in V$ to create the instances $P[d]$, obtains the full assignments graph of every $P[d]$, and then uses Saucy to return the generating set $Generating_d$ of its symmetry group, as described in Section 2.2. In practice, it might be useful to integrate several graph representations with different accuracy/complexity trade-offs. This would allow the implementation to use a less demanding representation whenever the size of the graph required by a more accurate one was deemed too high. For example, while the size of the full assignments graphs obtained during our experimental evaluation was manageable, it might become unmanageable for problems with constraints that contain large numbers of satisfied and not satisfied assignments.

If the set V is not provided as input, our implementation can automatically generate it if the elements of *Data* are tuples of k integers $(p_1, p_2, \ldots, p_k)$. This is done by starting from some user-defined base tuple, typically the smallest meaningful instance of the model, and increasing each p_i individually until there are enough instances to ensure diversity. Note that although this process may generate an unsatisfiable instance, this does not cause symmetries to be missed since all valid permutations are solution symmetries of unsatisfiable instances. It might, however, result in spurious parameterised permutations being added as candidates in step two.

Example 12 In the model of the Latin square problem given in Example 6, *Data* has a single component: the board size N. If the user provides (2) as the base tuple, we increment the component three times obtaining four values for d: (2), (3), (4), and (5). In the model of the Social Golfers problem given later in Section 6.2, *Data* has three components: the number of weeks, groups per week and players per group. If the user provides $(2,2,2)$ as the base tuple, we increment each component twice to get seven distinct values for d: $(2,2,2), (3,2,2), (4,2,2)$, $(2,3,2), (2,4,2)$, $(2,2,3)$, and $(2,2,4)$. □

Algorithm 1 MATCH(σ, $f[Data]$, d) returns *true* iff symmetry σ of instance $P[d]$ matches parameterised permutation $f[Data]$.

```
function MATCH($\sigma$, $f[Data]$, $d$)
    for $\ell \in lit(P[d])$ do
        if $\sigma(\ell) \neq f[d](\ell)$ then return false
    end for
    return true
end function
```

4.2 Step two: Lifting symmetries to parameterised permutations

The aim of this step is to construct, for each $d \in V$, a set $Candidates_d$ of patterns derived from the symmetries $Generating_d$ detected in step one for $P[d]$. The construction process has two separate phases.

4.2.1 Phase One

For each $\sigma \in Generating_d$, phase one tries to find one or more patterns $f[Data]$ that *match* σ for d (i.e. for which $f[d] = \{\sigma\}$) so that it can add $f[Data]$ to $Candidates_d$.

Note that in Phase one we consider only patterns that give a singleton set for all instances. Therefore, in this phase we identify a pattern with its sole parameterised permutation.

A Boolean function to check whether a given σ matches a given $f[Data]$ for d is shown in Algorithm 1. It returns *false* as soon as the results of applying $f[d]$ and σ to a literal of $P[d]$ differ, and *true* if they are the same for all its literals. Note that the set of literals in $P[d]$ is automatically (and lazily) generated by the implementation each time the function is executed.

Example 13 Consider again the LatinSquare[3] instance which, as shown in Example 8, has dimension sizes $Dims(3) = \langle \mathbf{x}(3,3,3) \rangle$. The generating set found by Saucy for this instance (given in Example 5) includes symmetry **F**, which swaps dimensions 2 and 3. For $N = 3$, this symmetry matches a parameterised permutation $f[N]$ defined as $\mathbf{x}_{i_1 i_2 i_3} \mapsto \mathbf{x}_{i_1 i_3 i_2}, \forall i_1, i_2, i_3 \in 1..N$. This is because the result of applying $f[3]$ to any literal l in LatinSquare[3] is equal to that obtained when applying **F** to l. Therefore, we would like $f[N]$ to be added to $Candidates_3$. □

While a symmetry might match many different parameterised permutations, most of them are unlikely to occur in other instances of the model. Therefore, we would like to restrict the space of parameterised permutations to those that are likely to apply across all instances.

We look for patterns that can be built by a fixed set of *pattern constructors*. These represent the kinds of symmetry that occur commonly in CSP models. Each pattern constructor takes some set set of arguments as input and produces a pattern. For a given model $P[Data]$, we construct a set called *Patterns* which contains all the instantiations of the pattern constructors that make sense for that model.

We first describe how the *basic patterns* are built. For a given model $P[Data]$, with $Dims(d) = \langle \mathbf{x}(d_1, d_2, \ldots, d_n) \rangle$ for $d \in Data$, the set of basic patterns contains the following patterns:

– **Dimension invert** ($\mathbb{DI}$) on the k^{th} dimension:

$$\mathbb{DI}[k] : \mathbf{x}_{i_1 \ldots i_n} \mapsto \mathbf{x}_{i_1 \ldots i_{k-1}\ (d_k - i_k + 1)\ i_{k+1} \ldots i_n}$$

– **Dimension swap** ($\mathbb{DS}$) on dimensions $k < k'$:

$$\mathbb{DS}[k,k'] : \mathbf{x}_{i_1 \ldots i_n} \mapsto \mathbf{x}_{i_1 \ldots i_{k-1}\ i_{k'}\ i_{k+1} \ldots i_{k'-1}\ i_k\ i_{k'+1} \ldots i_n}$$

– **Index swap** ($\mathbb{IS}$) in the k^{th} dimension for index values $v < v' \leq d_k$:

$$\mathbb{IS}[k,v,v'] : \mathbf{x}_{i_1 \ldots i_n} \mapsto \begin{cases} \mathbf{x}_{i_1 \ldots i_{k-1}\ v'\ i_{k+1} \ldots i_n} & \text{if } i_k = v \\ \mathbf{x}_{i_1 \ldots i_{k-1}\ v\ i_{k+1} \ldots i_n} & \text{if } i_k = v', \\ \textit{unchanged} & \text{otherwise.} \end{cases}$$

A pattern built by the $\mathbb{DI}$ pattern constructor corresponds to a parameterised permutation representing the common case of either variable (if the instantiated k is smaller than n) or value (if $k = n$) reflection symmetries. Consider, for example, the value reflection **A** (Figure 5) about the horizontal axis of the board of the $N = 8$ instance of the NQueens[N] model introduced later in Section 6.1. This symmetry matches the pattern $\mathbb{DI}[2]$ which *simultaneously* reflects each of the sequences of literals $\langle \mathbf{q}_{11}, \ldots, \mathbf{q}_{1N} \rangle, \langle \mathbf{q}_{21}, \ldots, \mathbf{q}_{2N} \rangle, \ldots, \langle \mathbf{q}_{N1}, \ldots, \mathbf{q}_{NN} \rangle$ and, thus, corresponds to the value reflection symmetry $\langle 1, \ldots, N \rangle \leftrightarrow \langle N, \ldots, 1 \rangle$. A reflection symmetry also exists in every instance of LatinSquare[N] even though they are not part of the generating set returned by Saucy, only of the generated group. In this case, the vertical axis simultaneously reflects each of the N^2 sequences of literals $\langle \mathbf{x}_{111}, \ldots, \mathbf{x}_{1N1} \rangle$, $\langle \mathbf{x}_{112}, \ldots, \mathbf{x}_{1N2} \rangle, \ldots,$ $\langle \mathbf{x}_{N1N}, \ldots, \mathbf{x}_{NNN} \rangle$.

A pattern built by the $\mathbb{DS}$ pattern constructor corresponds to a parameterised permutation representing another common reflection, that about a diagonal. For example, symmetry **C** in both Latin square instances of Figure 3 matches the pattern $\mathbb{DS}[1,2]$. Note that $\mathbb{DS}[k,k']$ represents a variable symmetry if $k' < n$ and a variable-value symmetry if $k' = n$.

An pattern built by the $\mathbb{IS}$ permutation pattern corresponds to a parameterised permutation representing the common case of symmetries that swap either two values (if the instantiating k is equal to n) or two sequences of variables (if $k < n$), such as two rows or two columns. The condition $v' \leq d_k$ ensures that $\mathbb{IS}[k,v,v']$ only applies to instances where both v and v' are within the range of dimension k. Examples include the variable symmetry **A** in both Latin square instances of Figure 3, which matches $\mathbb{IS}[2,2,3]$, and the value symmetry **D** also in both Latin square instances, which matches $\mathbb{IS}[3,1,2]$. When $k < n$, the $\mathbb{IS}[k,v,v']$ pattern is an extension of the definition of column (or row) permutation by Van Hentenryck et al. [36], for a bijection that swaps columns (or rows) v and v', to the case of an n-dimensional matrix (rather than a 2-dimensional one). We will often represent this pattern using the notation $\mathbf{x}_{i_1 \ldots i_{k-1}\ v\ i_{k+1} \ldots i_n} \leftrightarrow \mathbf{x}_{i_1 \ldots i_{k-1}\ v'\ i_{k+1} \ldots i_n}$ (which indicates the swapping of those literals leaving the rest unchanged), rather than the more verbose functional notation used above.

In addition to the three basic patterns introduced above, our set of patterns used in Phase one contains a *projection* variant where a given pattern p only applies to value v of dimension l. This fourth pattern constructor is defined as follows:

– **Projection** (Π) of pattern p on dimension l and index value $a \leq d_l$:

$$\Pi[l,a,p] : \mathbf{x}_{i_1 \ldots i_n} \mapsto \begin{cases} p(\mathbf{x}_{i_1 \ldots i_n}) & \text{if } i_l = a \\ \textit{unchanged} & \text{otherwise.} \end{cases}$$

Example 14 Consider a problem model $P[Data]$ where each element of $Data$ is an integer N, where $Dims(N) = \langle x(N,N)\rangle$, and where the values of the first variable x_1 can be inverted. This symmetry is a projection of the $\mathbb{DI}$ pattern, $\Pi[1,1,\mathbb{DI}[2]]$, and maps literal $\mathbf{x}_{ij}$ to:

$$\begin{cases} \mathbf{x}_{i(N-j+1)} & \text{if } i = 1 \\ \mathbf{x}_{ij} & \text{otherwise.} \end{cases}$$

which, as for $\mathbb{IS}$, we will represent using the notation $\mathbf{x}_{1j} \leftrightarrow \mathbf{x}_{1(N-j+1)}$, rather than the more verbose functional notation used above. □

A pattern of the form $\Pi(p,l,a)$ constructor corresponds to a parameterised permutation representing the common case where a symmetry represented by the pattern p occurs only for a particular subset of the variables or the values. Consider, for example, the variable symmetry $\mathbf{D}$ for the instances of the Golf$[W,G,P]$ model of the Social Golfers problem, introduced later in Section 6.2. This symmetry swaps groups 1 and 2 of players but only within week 1. It matches the parameterised permutation $\Pi[1,1,\mathbb{IS}[2,1,2]]$, which corresponds to the $\mathbb{IS}$ pattern swapping values 1 and 2 of the second dimension (groups of players 1 and 2) projected onto value 1 of the first dimension (week 1).

Phase one, described by Algorithm 2, starts the construction of $Candidates_d$ by trying to match every symmetry σ in $Generating_d$ against every parameterised permutation in the set $PermPatterns_d$ and collecting the matching parameterised permutations in $Candidates_d$. In our implementation, given a model $P[Data]$ with $Dims(d) = \langle \mathbf{x}(d_1,d_2,\ldots,d_n)\rangle$ for $d \in Data$, $PermPatterns_d$ contains all instantiations of the basic pattern constructors, as well as all instantiations of the projection pattern constructor. It is formally defined as:

$$PermPatterns_d = Basic_d \cup \{\Pi[l,a,p] \mid l \in 1..n, a \in 1..d_l, p \in Basic_d\}$$

where

$$\begin{aligned} Basic_d = &\{\mathbb{DI}[k] \mid k \in 1..n\} \\ &\cup \{\mathbb{IS}[k,v,v'] \mid k \in 1..n,\ v,v' \in 1..d_k,\ v < v'\} \\ &\cup \{\mathbb{DS}[k,k'] \mid k,k' \in 1..n,\ k < k'\} \end{aligned}$$

One could easily increase the accuracy of the implementation by either adding other basic pattern constructors, or allowing nested projections (that is, allowing p in $\Pi[l,a,p]$ to be a projection pattern itself). Of course, this would have an associated cost (see Section 4.2.3 for details).

Note that at the end of phase one the system is able to inform the user of possibly missing patterns: if every $Generating_d$ set contains a symmetry that could not be matched, this might indicate the existence of a missing pattern that would account for those unmatched symmetries. Thus, the system produces a list of unmatched symmetries for the user's inspection.

Example 15 Consider the generating set $Generating_3 = \{\mathbf{A},\mathbf{B},\mathbf{C},\mathbf{D},\mathbf{E},\mathbf{F}\}$ returned by Saucy for the LatinSquare[3] in Example 5. During phase one of the detection process for $d = 3$, its elements are automatically matched to the following elements of $PermPatterns_3$:

A	matches $\mathbb{IS}[2,2,3]$:	$\mathbf{x}_{i2l} \leftrightarrow \mathbf{x}_{i3l}, \forall i,l \in 1..N$
B	matches $\mathbb{IS}[1,2,3]$:	$\mathbf{x}_{2jl} \leftrightarrow \mathbf{x}_{3jl}, \forall j,l \in 1..N$
C	matches $\mathbb{DS}[1,2]$:	$\mathbf{x}_{ijl} \mapsto \mathbf{x}_{jil}, \forall i,j,l \in 1..N$
D	matches $\mathbb{IS}[3,1,2]$:	$\mathbf{x}_{ij1} \leftrightarrow \mathbf{x}_{ij2}, \forall i,j \in 1..N$
E	matches $\mathbb{IS}[3,2,3]$:	$\mathbf{x}_{ij2} \leftrightarrow \mathbf{x}_{ij3}, \forall i,j \in 1..N$
F	matches $\mathbb{DS}[2,3]$:	$\mathbf{x}_{ijl} \mapsto \mathbf{x}_{ilj}, \forall i,j,l \in 1..N$

Algorithm 2 Phase one of the construction of $Candidates_d$.

```
function PHASE-ONE($d$, $Generating_d$)
    $Candidates_d \leftarrow \emptyset$
    for $\sigma \in Generating_d$ do
        for $p \in PermPatterns_d$ do
            if MATCH$(\sigma, p, d)$ then $Candidates_d \leftarrow Candidates_d \cup \{p\}$
        end for
    end for
    return $Candidates_d$
end function
```

Since no other element of $PermPatterns_3$ is matched, at the end of phase one we have $Candidates_3 = \{\mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[1,2,3], \mathbb{IS}[2,2,3], \mathbb{IS}[3,1,2], \mathbb{IS}[3,2,3]\}$ and the algorithm moves to $d = 4$. The first six symmetries in the set $Generating_4 = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{A1}, \mathbf{B1}, \mathbf{E1}\}$ returned by Saucy for $d = 4$ match the same patterns as for $d = 3$. This causes the same six elements to be added to $Candidates_4$. In addition, **A1, B1** and **E1** match the following elements:

A1 matches $\mathbb{IS}[2,3,4] : \mathbf{x}_{i3l} \leftrightarrow \mathbf{x}_{i4l}, \forall i,l \in 1..N$
B1 matches $\mathbb{IS}[1,3,4] : \mathbf{x}_{3jl} \leftrightarrow \mathbf{x}_{4jl}, \forall j,l \in 1..N$
E1 matches $\mathbb{IS}[3,3,4] : \mathbf{x}_{ij3} \leftrightarrow \mathbf{x}_{ij4}, \forall i,j \in 1..N$

Therefore, at the end of phase one $Candidates_4$ contains the original six elements plus the above three new patterns.

The algorithm also analyses the instances $d = 2$ and $d = 5$, obtaining similar results. In particular, at the end of phase one we have:

$Candidates_2 = \{\mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[3,1,2], \mathbb{DI}[3]\}$
$Candidates_3 = \{\mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[1,2,3], \mathbb{IS}[2,2,3], \mathbb{IS}[3,1,2], \mathbb{IS}[3,2,3]\}$
$Candidates_4 = Candidates_3 \cup \{\mathbb{IS}[1,3,4], \mathbb{IS}[2,3,4], \mathbb{IS}[3,3,4]\}$
$Candidates_5 = Candidates_4 \cup \{\mathbb{IS}[1,4,5], \mathbb{IS}[2,4,5], \mathbb{IS}[3,4,5]\}$.

Since every symmetry in every generating set has been matched, no list of unmatched symmetries is printed for the user's inspection. $\square$

In the following we will say that pattern $f[Data]$ *occurs* in instance $P[d]$ if the symmetry group of $P[d]$ contains at least one symmetry that matches $f[Data]$. Note that while all parameterised permutations in $Candidates_d$ must occur in $P[d]$, there might be parameterised permutations that occur in $P[d]$ but do not appear in $Candidates_d$.

4.2.2 Phase Two

While the elements in $PermPatterns_d$ (and, therefore, those in the $Candidates_d$ sets computed by phase one) are patterns that give singleton parameterised permutations, we are also interested in detecting parameterised sets of permutations. We introduce the following two pattern constructors, $\overline{\mathbb{IS}}$ and $\overline{\Pi}$. Since these parameterised sets generate parameterised permutation groups, we define them in terms of the occurrence in $P[d]$ of the parameterised permutations that generate the target group.

– **All index swap** ($\overline{\mathbb{IS}}$) in the k^{th} dimension: $\overline{\mathbb{IS}}[k]$ occurs in $P[d]$ if there exists a path between any two pairs of values in the k^{th} dimension, such that for each edge connecting values v and v' in the path, $\mathbb{IS}[k,v,v']$ or $\mathbb{IS}[k,v',v]$ occurs in $P[d]$.

- **All index projection ($\overline{\Pi}$)** of pattern p on dimension l: $\overline{\Pi}[l,p]$ occurs in $P[d]$ if $\Pi[l,a,p]$ occurs in $P[d]$ for every value a in $1..d_l$.

A pattern of the form $\overline{\mathbb{IS}}[k]$ corresponds to a commonly occurring symmetry group in constraint models. It represents a set of interchangeable values (if $k = n$) or interchangeable sequences of variables (if $k < n$), which often occurs as *column (or row) interchangeability* [36]. For example, the pattern $\overline{\mathbb{IS}}[3]$ represents the interchangeability of all values in the Latin square model. We know that $\overline{\mathbb{IS}}[3]$ occurs in LatinSquare[4] because the parameterised permutations $\mathbb{IS}[3,1,2]$, $\mathbb{IS}[3,2,3]$ and $\mathbb{IS}[3,3,4]$ (representing symmetries **D**, **E** and **E1**, respectively) appear in $Candidates_4$ and, therefore, occur in LatinSquare[4]. Note that in order for $\overline{\mathbb{IS}}[3]$ to occur in LatinSquare[5] we need a new parameterised permutation ($\mathbb{IS}[3,4,5]$) to appear in $Candidates_5$. In fact, for the generating set returned by Saucy, every $Candidates_d$ needs to have at least one more pattern than $Candidates_{d-1}$, making it difficult to express the symmetry group. The $\overline{\mathbb{IS}}$ pattern constructor allows us to capture this without the need to mention explicit values.

A pattern of the form $\overline{\Pi}[l,p]$ pattern corresponds to another commonly occurring symmetry group: that generated by several patterns $\Pi[l,v,p]$, one for each element v of dimension l. For example, consider a model with a matrix of variables where the variables of each row can be interchanged, independently of the other rows. The literals form a matrix of three dimensions; dimension 1 for the rows, dimension 2 for the columns, and dimension 3 for the values. The symmetry can be represented by $\overline{\Pi}[1,\overline{\mathbb{IS}}[2]]$; this indicates that $\Pi[1,v,\overline{\mathbb{IS}}[2]]$, which interchanges the columns only in row v, is present for every row. Again, if the number of values of the first dimension varies with each $d \in Data$, every $Candidates_d$ would require different elements in order to detect the parameterised group. The $\overline{\Pi}$ pattern unifies these patterns across the different data values.

The p in $\overline{\Pi}[l,p]$ will often be a non-singleton parameterised set of permutations itself. In a group-theoretic sense, for a given instance if p generates parameterised permutation group G and l has n elements, then $\overline{\Pi}[l,p]$ corresponds to the direct product G^n. Similarly, the combination of $\overline{\Pi}[l,p]$ and any other pattern that generates symmetry group H that acts also on l (such as $\mathbb{DI}[l]$ or $\overline{\mathbb{IS}}[l]$) represents the wreath product of G^n and H.

Note that the singleton versions of pattern constructors $\mathbb{DS}$ and $\mathbb{DI}$ are sufficient; there is no need for a parameterised set version. This is because they only have dimensions as parameters and the number of dimensions is constant for all instances of a model. Therefore, the relevant $\mathbb{DS}$ and $\mathbb{DI}$ patterns can appear in every $Candidates_d$ set (e.g. $\mathbb{DS}[1,2]$ and $\mathbb{DS}[2,3]$ in Latin square appear in all the $Candidates_d$ sets computed by phase one).

Phase two continues the construction process of a given $Candidates_d$ set by adding parameterised set patterns to it. Our implementation, described by Algorithm 3, only adds the following three kinds of patterns. First, it adds any $\overline{\mathbb{IS}}[k]$ returned by INFER-$\overline{\text{IS}}(Candidates_d)$, i.e. any $\overline{\mathbb{IS}}[k]$ whose occurrence in $P[d]$ can be detected by the appearance in $Candidates_d$ of the necessary $\mathbb{IS}[k,v,v']$ elements. Second, it adds any $\Pi[k,v,\overline{\mathbb{IS}}[l]]$ for which $\overline{\mathbb{IS}}[l]$ is in the set returned by INFER-$\overline{\text{IS}}(kvp)$, where kvp is the set containing all patterns p for which $\Pi[k,v,p]$ appears in $Candidates_d$. In other words, any $\Pi[k,v,\overline{\mathbb{IS}}[l]]$ for which the $\Pi[k,v,\mathbb{IS}[l,a,a']]$ elements for all necessary a and a' appear in kvp (and, thus, in $Candidates_d$). And third, it adds any $\overline{\Pi}[k,p]$ for which $\overline{\Pi}[k,v,p]$ appears in $Candidates_d$ for every value v in dimension k.

Algorithm 3 is essentially a brute force search over all possible values for the $\overline{\Pi}$ and $\overline{\mathbb{IS}}$ patterns, and so finds all such patterns as long as their component patterns are found in phase one.

Example 16 Consider phase two of the algorithm for the $\mathit{Candidates}_d$ sets generated in Example 15. For $d = 2$ the presence of $\mathbb{IS}[3,1,2]$ in $\mathit{Candidates}_2$ causes INFER-$\overline{\text{IS}}$($\mathit{Candidates}_3$) to return $R = \{\overline{\mathbb{IS}}[3]\}$ representing the interchangeability of all values in the Latin Square. Therefore, $\overline{\mathbb{IS}}[3]$ is added to $\mathit{Candidates}_2$. Since $\mathit{Candidates}_2$ does not contain any projection pattern, the algorithm moves to $d = 3$. For $d = 3$ the presence of $\mathbb{IS}[3,1,2]$ and $\mathbb{IS}[3,2,3]$ in $\mathit{Candidates}_3$ again causes $\overline{\mathbb{IS}}[3]$ to be added to $\mathit{Candidates}_3$ and, as before, the algorithm proceeds to $d = 4$. For $d = 4$, the presence of $\mathbb{IS}[3,1,2]$, $\mathbb{IS}[3,2,3]$ and $\mathbb{IS}[3,3,4]$ in $\mathit{Candidates}_4$ again causes $\overline{\mathbb{IS}}[3]$ to be added to $\mathit{Candidates}_4$ and the algorithm moves to $d = 5$. Similarly, the presence of $\mathbb{IS}[3,1,2]$, $\mathbb{IS}[3,2,3]$, $\mathbb{IS}[3,3,4]$, and $\mathbb{IS}[3,4,5]$ in $\mathit{Candidates}_5$ causes $\overline{\mathbb{IS}}[3]$ to be added to $\mathit{Candidates}_5$, and phase two finishes. □

Algorithm 3 Phase two of construction of $\mathit{Candidates}_d$.

```
function INFER-$\overline{\text{IS}}$($\mathit{Matched}$)
    $R \leftarrow \emptyset$
    for $k \in 1..n$ do
        $S \leftarrow \{1\}$
        for $v' \in 2..d_k$ do
            if $\exists v \in S.(\mathbb{IS}[k,v,v'] \in \mathit{Matched})$ then $S \leftarrow S \cup \{v'\}$
        end for
        if $S = 1..d_k$ then $R \leftarrow R \cup \overline{\mathbb{IS}}[k]$
    end for
    return $R$
end function

function PHASE-TWO($d$, $\mathit{Candidates}_d$)
    $\mathit{Candidates}_d \leftarrow \mathit{Candidates}_d \cup$ INFER-$\overline{\text{IS}}$($\mathit{Candidates}_d$)
    for $k \in 1..n$ do
        for $v \in 1..d_k$ do
            let $kvp = \{p \mid \Pi[k,v,p] \in \mathit{Candidates}_d\}$
            $S \leftarrow$ INFER-$\overline{\text{IS}}$($kvp$)
            $\mathit{Candidates}_d \leftarrow \mathit{Candidates}_d \cup \{\Pi[k,v,p] \mid p \in S\}$
        end for
    end for
    for $p \in \mathit{Basic}_d \cup \{\overline{\mathbb{IS}}[k] \mid k \in 1..n\}$ do
        for $k \in 1..n$ do
            if $\forall v \in 1..d_k.\Pi[k,v,p] \in \mathit{Candidates}_d$ then
                $\mathit{Candidates}_d \leftarrow \mathit{Candidates}_d \cup \{\overline{\Pi}[k,p]\}$
            end if
        end for
    end for
    return $\mathit{Candidates}_d$
end function
```

4.2.3 Properties of Step Two

Let us first discuss the complexity of step two. In phase one, matching a symmetry σ of instance $P[d]$ against a parameterised permutation requires at worst $O(|lit(P[d])|)$ operations, that is, linear in the number of literals in the instance. In practice, because the matching test fails at the first literal whose image according to the parameterised permutation is different from that obtained by σ, most tested parameterised permutations can be discarded quickly.

The number of patterns to be tested grows with the number and sizes of the dimensions. Let d_k be the largest dimension size. Since there are $O(n)$ $\mathbb{DI}$ patterns, $O(nd_k^2)$ $\mathbb{IS}$ patterns, and $O(n^2)$ $\mathbb{DS}$ patterns, we have $|Basic_d| = O(n(d_k^2+n))$. And since there are $O(nd_k|Basic_d|)$ Π patterns, we have $|PermPatterns_d| = O(|Basic_d| + nd_k|Basic_d|) = O(n^2d_k(d_k^2+n))$.

In phase two, the function INFER-$\overline{\text{IS}}$ requires $O(nd_k)$ operations, assuming that the set operations are done in constant time. Consider now the function PHASE-TWO, which has three parts: the initial call to INFER-$\overline{\text{IS}}$, the first nested loop, and the second nested loop. The complexity of the initial call is $O(nd_k)$ and that of the first nested loop is $O(n^2d_k^2)$. Both are dominated by the cost of the second nested loop, which is $O((|Basic_d|+n) \times nd_k) = O(n(d_k^2+n)(nd_k)) = O(n^2d_k(d_k^2+n)) = O(n^2d_k^3+n^3d_k)$.

Let us now discuss the expressive power of our patterns by considering how commonly occurring symmetries are captured (or not) by our current implementation. Let us start with value interchangeability. We can represent the interchangeability of either two values v and v' with $v < v'$ ($\mathbb{IS}[n,v,v']$), or of all values ($\overline{\mathbb{IS}}[n]$). Note that, for those patterns, the values need to be interchangeable for all variables in the matrix. We cannot represent interchangeability of any other set of values that depends on d_n, such as set $\{v \mid 1 \leq v < (d_n/2)\}$. We would need a new pattern for capturing the $1 \leq v < (d_n/2)$ relationship. Piecewise value interchangeability [17] (that is, disjoint sets of interchangeable values) can be represented if the values belong to different matrices, or if all values appear in all instances (through the composition of the appropriate $\mathbb{IS}$ patterns).

Regarding variable interchangeability, $\mathbb{IS}[k,v,v']$ for $k < n$ allows us to capture (a general version of) column and row permutation symmetries [36]. Further, we can represent the interchangeability of either two variables x_i and x_j with $i < j$ ($\mathbb{IS}[1,i,j]$), or of all variables ($\overline{\mathbb{IS}}[1]$) if the matrix has only two dimensions. We can also represent the interchangeability of a given subset of variables, if the subset corresponds to the variables represented by a particular dimension $k < n$ ($\overline{\mathbb{IS}}[k]$). Piecewise variable interchangeability [17] (that is, disjoint sets of interchangeable variables) can be represented if each subset corresponds to a different dimension (or different matrices), through the appropriate instantiations of the $\overline{\mathbb{IS}}$ pattern constructor.

In addition, we can represent inversions of either values or variables through instantiations of the $\mathbb{DI}$ pattern constructor, and diagonal reflections of either variables or variable/values through instantiations of the $\mathbb{DS}$ pattern constructor. Also, the Π pattern constructor allows us to represent symmetries that only apply to certain dimensions. For example, it allows us to capture the interchangeability of all values for a given subset of variables, if that subset corresponds to the variables represented by a particular value a of dimension l ($\Pi[l,a,\mathbb{IS}[n,v,v']]$ and $\Pi[l,a,\overline{\mathbb{IS}}[n]]$). Furthermore, we can easily represent the direct product of two symmetry groups through the composition of groups that apply to different dimensions (such as $\overline{\mathbb{IS}}[i]$ and $\overline{\mathbb{IS}}[j]$, $i \neq j$), and the wreath product of symmetry groups through the composition of a $\overline{\Pi}[l,p]$ pattern and a pattern that acts on l (such as $\overline{\mathbb{IS}}[l]$ and $\mathbb{DI}[l]$).

Note that symmetries that require the simultaneous application of two or more patterns will not be matched and, therefore, they will only be detected as part of the symmetry group if all the instantiated patterns occur in all the instances. This is clear when considering, for example, a model with 4 dimensions and symmetry $\sigma(\mathbf{x}_{i_1i_2i_3i_4}) = \mathbf{x}_{i_2i_1i_4i_3}$. This symmetry does not match any pattern and, therefore, it is only detected as a symmetry of the model if both $\mathbb{DS}[1,2]$ and $\mathbb{DS}[3,4]$ are parameterised model symmetries and are detected as candidates for all instances of the problem. While this is a significant limitation of our current implementation, it considerably reduces the complexity of the detection algorithm.

4.3 Step three: Filtering candidate symmetries

The aim of this step is to create a set *Likely* of patterns that are likely to be proved as model symmetries in the final step. Algorithm 4 constructs this *Likely* set as follows. It first adds to $Likely_0$ any element p of any candidates set, such that $\forall d \in V$, p occurs in every instance $P[d]$ either directly (p is a member of $Candidates_d$), or indirectly (the permutation group represented by $p[d]$ belongs to the symmetry group of $P[d]$). The indirect check is represented by expression $p \in Group(Candidates_d)$ in the algorithm and performed using the GAP system for computational group theory [33].

The above two checks exclude from $Likely_0$ candidates that do not occur in all instances of the model. Further, the indirect check partly mitigates the problem of Saucy returning arbitrary generating sets; with this check, candidates cannot be overlooked due to Saucy producing generating sets of symmetries that match different patterns in different instances.

Once $Likely_0$ is computed, the algorithm disregards any element that is redundant due to the addition of *occluders*, that is, patterns that subsume (or "occlude") others:

$$\begin{aligned} occluders(\mathbb{IS}[k,v,v']) &= \{\overline{\mathbb{IS}}[k]\} \\ occluders(\Pi[k,v,p]) &= \{\overline{\Pi}[k,p]\} \cup \{\Pi[k,v,p'] \mid p' \in occluders(p)\} \\ occluders(\mathbb{DI}[k]) &= \{\overline{\mathbb{IS}}[k]\} \\ occluders(\overline{\Pi}[k,p]) &= \{\overline{\Pi}[k,p'] \mid p' \in occluders(p)\} \end{aligned}$$

Specifically, the algorithm removes from $Likely_0$ any $p \in Likely_0$ such that $occluders(p) \subseteq Likely_0$. Note that the removal of occluded elements might result in a loss of accuracy if the occluding symmetry group cannot be proved while the occluded element could.

Algorithm 4 Step three of the symmetry detection algorithm.

```
function OCCURS($d, p$)
    return $p \in Candidates_d \vee p \in Group(Candidates_d)$
end function

function STEP-THREE($V, \{Candidates_d \mid d \in V\}$)
    $Likely_0 \leftarrow \emptyset$
    for $p \in \bigcup_{d \in V} Candidates_d$ do
        if $\forall d \in V$ OCCURS($d, p$) then $Likely_0 \leftarrow Likely_0 \cup \{p\}$
    end for
    $Likely \leftarrow Likely_0$
    for $p \in Likely_0$ do
        if $occluders(p) \subseteq Likely_0$ then
            $Likely \leftarrow Likely \setminus \{p\}$
        end if
    end for
end function
```

Example 17 Recall that, for LatinSquare[N] and instance values $d = 2$, $d = 3$, $d = 4$ and $d = 5$, the implementation returns the following sets after step two:

$Candidates_2 = \{\mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[3,1,2], \overline{\mathbb{IS}}[3], \mathbb{DI}[3]\}$
$Candidates_3 = (Candidates_2 \setminus \{\mathbb{DI}[3]\}) \cup \{\mathbb{IS}[1,2,3], \mathbb{IS}[2,2,3], \mathbb{IS}[3,2,3]\}$
$Candidates_4 = Candidates_3 \cup \{\mathbb{IS}[1,3,4], \mathbb{IS}[2,3,4], \mathbb{IS}[3,3,4]\}$
$Candidates_5 = Candidates_4 \cup \{\mathbb{IS}[1,4,5], \mathbb{IS}[2,4,5], \mathbb{IS}[3,4,5]\}$.

Elements $\mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[3,1,2], \overline{\mathbb{IS}}[3]$ appear in all candidate sets and are thus added to $Likely_0$. Element $\mathbb{DI}[3]$ only appears in $Candidates_2$ but occurs in all other instances (as indicated by GAP). Thus, it is also added to $Likely_0$. All other elements are of the form $\mathbb{IS}[i,j,k]$ and do not occur in LatinSquare[2] (since k is outside the range of dimension i). Thus, *Likely* is initially computed as $\{\mathbb{DI}[3], \mathbb{DS}[1,2], \mathbb{DS}[2,3], \mathbb{IS}[3,1,2], \overline{\mathbb{IS}}[3]\}$. Then, $\mathbb{DI}[3]$ and $\mathbb{IS}[3,1,2]$ are eliminated from it, since they are occluded by $\overline{\mathbb{IS}}[3]$. □

4.4 Step four: Proving model symmetries

The final step of our method checks whether the patterns returned in the set *Likely* by step three are model symmetries, i.e. whether, in every instance of the model, they map to sets whose elements are symmetries. The fourth step is vital if the symmetries are to be later used by symmetry breaking techniques to make the search more efficient, for if the permutations are not symmetries then the search may fail to find some solutions.

Proving that a candidate parameterised permutation is indeed a model symmetry can be achieved, for example, by first representing both the model and the candidate parameterised permutation in the logic formalism described by Mancini and Cadoli [18], and then making use of theorem proving techniques. Of course, such a technique is in general undecidable and requires the user to find an equivalent logical expression for their constraint model. In this sense, such an approach is not truly automatic.

An alternative approach to proving the existence of a symmetry on a model is to show that the symmetry, when applied to the model, leaves the model itself unchanged. In other words, a model M is transformed into a new model $f(M)$, by applying the symmetry f to every constraint in M. If $f(M)$ can be shown to be equivalent to M, then the symmetry f is a model symmetry. This approach has been described and implemented by Mears et al. [24], who show that it can be applied successfully for various kinds of symmetry. The approach involves applying a parameterised permutation to the constraints of a MiniZinc model M obtaining a new model M', and then normalising the constraints of both M and M' in such a way that each constraint in M is syntactically identical (up to Presburger arithmetic) to one in M'.

Example 18 Of the likely candidates found for the Latin square problem (Example 17), the proof technique in [24] can prove that the $\mathbb{DS}[1,2]$ candidate is a model symmetry and that the elements of $\overline{\mathbb{IS}}[3]$ are model symmetries, but is unable to prove that the $\mathbb{DS}[2,3]$ candidate is also a model symmetry. □

Given an approach to verify the occurrence in every instantiation of the parameterised permutations, one may ask whether we could bypass step 2 and simply mark as a likely candidate every instantiation of a pattern that GAP confirmed as occurring. While this is indeed possible, one would end up marking as likely candidates not only the generating set of the symmetry group but also all other instantiations of the pattern constructors. This has two main disadvantages. First, one would have to prove many more likely candidates. And second, many redundant symmetries may be found, which might slow down the search for solutions for the instances of the problem, since the performance of symmetry breaking approaches is often reduced when dealing with many symmetries.

5 Limitations of the Method

Clearly, the way in which the model is specified naturally affects the effectiveness of the method. For example, a model symmetry must occur in every instance of the model and, therefore, must be determined by information explicitly represented in the model itself. A more complex *Data* parameter may lead to fewer model symmetries and more instance-specific symmetries. Also, specifying only some parameters — restricting a model to another more specific model — may cause new model symmetries to emerge.

Theoretically, the method is not affected by the particular choice of constraints used in the model, since it can be implemented using a complete instance-based detection method that does not depend on this. However, complete instance-based methods tend to be computationally expensive, even for small instances. Even so, it is more likely for our method to be less dependent on the choice of constraints than other model symmetry detection approaches, since it can use information from the instance to reduce its dependence. For example, while our method can be implemented using complete instance symmetry detection methods (e.g. [5]), the instance-based detection method that we chose (full assignments graph [21]) makes the resulting model method less affected by the constraint syntax used in the model than previous approaches, since it does not require the use of global constraints and can cope with some degree of syntax variation.

The method is sensitive, however, to the way in which variables are specified in the model, since they determine the way in which the structure of the problem is represented (i.e. its dimensions and index values). This structure is the basis to define the patterns, and some structures might make it easier to detect patterns than others.

Example 19 The variable symmetries given in Example 2 that swap a pair of columns of LatinSquare$[N]$ can easily be recognised because the tuple of indices that identify each variable in the model contains an index j corresponding to each column. Similarly, row symmetries are recognisable because there is an index i corresponding to each row. The situation is different in a model where each variable is uniquely identified by a single index ranging from $1..N^2$:

$$\begin{array}{ll} X[N] : N \mapsto & \{x_i \mid i \in 1..N^2\} \\ D[N] : N \mapsto & 1..N \\ C[N] : N \mapsto & \{x_i \neq x_j \mid i,j \in 1..N^2 \text{ where } i < j \wedge ((i-1) \bmod N) = ((j-1) \bmod N)\} \cup \\ & \{x_i \neq x_j \mid i,j \in 1..N^2 \text{ where } i < j \wedge ((i-1) \operatorname{div} N) = ((j-1) \operatorname{div} N)\} \end{array}$$

This model is less intuitive and the symmetries are correspondingly more difficult to express. Consider, for example, the "swap rows 2 and 3" variable symmetry. This is represented in the original model as $\langle x_{21},\ldots,x_{2N}\rangle \leftrightarrow \langle x_{31},\ldots,x_{3N}\rangle$ (which corresponds to $\mathbb{IS}[1,2,3]$), and in the above model as $\langle x_{N+1},\ldots,x_{2N}\rangle \leftrightarrow \langle x_{2N+1},\ldots,x_{3N}\rangle$, arguably a less intuitive form.

$\square$

Different choices of variable names also lead to different models and, thus, to different structures.

Example 20 Consider the Golomb ruler problem and the associated model introduced in Example 9. A different model could have been constructed with a single set of variables:

$$\begin{array}{ll} X[N] : N \mapsto & \{d_a \mid a \in 1..N-1\} \\ D[N] : N \mapsto & \{1..N^2\} \\ C[N] : N \mapsto & \{\sum_{i \le a < j} d_a \neq \sum_{k \le a < l} d_a \mid i,j,k,l \in 1..N \wedge i < j \wedge k < l \wedge (i \neq k \vee j \neq l)\} \end{array}$$

The values of the marks on the ruler are the cumulative sums of the d_a variables; the n^{th} mark is equal to $\sum_{a=1}^{N-1} d_a$. The literals in every instance of the above model can be arranged into a single matrix named d, where literals are uniquely identified by the name **d** and the (i,j) tuple, where $i \in 1..N-1, j \in 1..N^2$. Therefore, $Dims(N) = \langle \mathbf{d}(N-1, N^2) \rangle$. This model has different symmetry detection opportunities to those of Example 9: the symmetry that reflects the differences now acts only on the d_a variables, affects a single dimension and matches the $\mathbb{DI}[1]$ pattern. $\square$

In summary, our method can miss model properties if the choice of variable specification, be it its variable name or its associated indices, does not reflect the structure of the property we are trying to detect.

6 Detailed Examples

We have discussed how our implementation works with the Latin square example. Here we illustrate this further with three detailed examples: N-queens and Social Golfers, for which our method detects all model symmetries as likely candidates; and Golomb ruler, for which it fails to detect any likely candidate.

6.1 N-queens

The N-queens problem is to position N queens on an $N \times N$ chess board without attacking each other. The following NQueens[N] model uses N integer variables q_i where $q_i = j$ if the queen in column i appears in row j.

$$\begin{aligned}
X[N] : N \mapsto\ & \{q_i \mid i \in 1..N\} \\
D[N] : N \mapsto\ & 1..N \\
C[N] : N \mapsto\ & \{q_i \neq q_j \mid i \in 1..N, j \in i+1..N\} \cup \\
& \{q_i + i \neq q_j + j \mid i \in 1..N, j \in i+1..N\} \cup \\
& \{q_i - i \neq q_j - j \mid i \in 1..N, j \in i+1..N\}
\end{aligned}$$

The $q_i \neq q_j$ constraint ensures that no two queens share a row, while the other two constraints ensure that no two queens share a diagonal in either direction (rising or falling). The literals in every instance of the model can be arranged into a single 2-dimensional matrix with $Dims(N) = \langle \mathbf{q}(N,N) \rangle$, where each literal $q_i = j$ is uniquely identified as $\mathbf{q}_{ij}$. During the first step of the analysis, and given an initial user-defined value $N = 8$, the implementation generates the full assignment graphs for $N = 8, N = 9, N = 10$ and $N = 11$ and uses Saucy to compute the generating sets of the symmetry group of each graph. Figure 5 shows the graph for $N = 8$, with the elements of its generating set marked as bold arrows. Again, each vertex in the graph represents a literal $\mathbf{q}_{ij}$ where the **q** has been omitted for clarity. Each edge, shown as a thin black line, indicates that the two literals at the end-points are incompatible.

For $N = 8$ Saucy returns $Generating_8 = \{\mathbf{A}, \mathbf{B}\}$ where:

A $\langle \mathbf{q}_{11}, \mathbf{q}_{12}, \mathbf{q}_{13}, \mathbf{q}_{14}, \mathbf{q}_{21}, \mathbf{q}_{22}, \mathbf{q}_{23}, \mathbf{q}_{24}, \cdots, \mathbf{q}_{81}, \mathbf{q}_{82}, \mathbf{q}_{83}, \mathbf{q}_{84} \rangle \leftrightarrow$
$\langle \mathbf{q}_{18}, \mathbf{q}_{17}, \mathbf{q}_{16}, \mathbf{q}_{15}, \mathbf{q}_{28}, \mathbf{q}_{27}, \mathbf{q}_{26}, \mathbf{q}_{25}, \cdots, \mathbf{q}_{88}, \mathbf{q}_{87}, \mathbf{q}_{86}, \mathbf{q}_{85} \rangle$

B $\langle \mathbf{q}_{12}, \mathbf{q}_{13}, \mathbf{q}_{14}, \mathbf{q}_{15}, \mathbf{q}_{16}, \mathbf{q}_{17}, \mathbf{q}_{18}, \mathbf{q}_{23}, \mathbf{q}_{24}, \mathbf{q}_{25}, \mathbf{q}_{26}, \mathbf{q}_{27}, \mathbf{q}_{28}, \cdots, \mathbf{q}_{78} \rangle \leftrightarrow$
$\langle \mathbf{q}_{21}, \mathbf{q}_{31}, \mathbf{q}_{41}, \mathbf{q}_{51}, \mathbf{q}_{61}, \mathbf{q}_{71}, \mathbf{q}_{81}, \mathbf{q}_{32}, \mathbf{q}_{42}, \mathbf{q}_{52}, \mathbf{q}_{62}, \mathbf{q}_{72}, \mathbf{q}_{82}, \cdots, \mathbf{q}_{87} \rangle$

Symmetry **A** maps each value j in the second dimension to $N - j + 1$. This is the reflection of the chessboard around the central horizontal axis. Symmetry **B** swaps the variable and value dimensions. This is the reflection of the chessboard around the main diagonal axis. During phase one of the second step of the analysis, these symmetries are automatically matched to the following patterns:

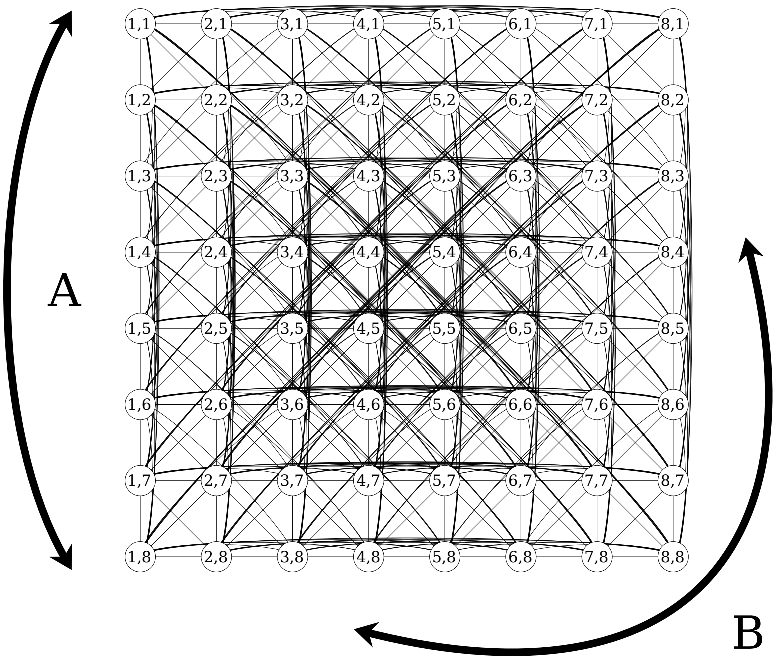

Fig. 5 Full assignments graph of instance NQueens[8]

A matches $\mathbb{DI}[2]$: $\mathbf{q}_{ij} \mapsto \mathbf{q}_{i(N-j+1)} \forall i, j \in 1..N$
B matches $\mathbb{DS}[1,2]$: $\mathbf{q}_{ij} \mapsto \mathbf{q}_{ji}, \forall i, j \in 1..N$

No extra patterns are found during phase two and, therefore, $\mathit{Candidates}_8 = \{\mathbb{DI}[2], \mathbb{DS}[1,2]\}$. For $N = 9$ Saucy returns $\mathit{Generating}_9 = \{\mathbf{A1}, \mathbf{B1}\}$ where:

A1 $\langle \mathbf{q}_{11}, \mathbf{q}_{12}, \mathbf{q}_{13}, \mathbf{q}_{14}, \mathbf{q}_{21}, \mathbf{q}_{22}, \mathbf{q}_{23}, \mathbf{q}_{24}, \cdots, \mathbf{q}_{91}, \mathbf{q}_{92}, \mathbf{q}_{93}, \mathbf{q}_{94} \rangle \leftrightarrow$
$\langle \mathbf{q}_{19}, \mathbf{q}_{18}, \mathbf{q}_{17}, \mathbf{q}_{16}, \mathbf{q}_{29}, \mathbf{q}_{28}, \mathbf{q}_{27}, \mathbf{q}_{26}, \cdots, \mathbf{q}_{99}, \mathbf{q}_{98}, \mathbf{q}_{97}, \mathbf{q}_{96} \rangle$
B1 $\langle \mathbf{q}_{12}, \mathbf{q}_{13}, \mathbf{q}_{14}, \mathbf{q}_{15}, \mathbf{q}_{16}, \mathbf{q}_{17}, \mathbf{q}_{18}, \mathbf{q}_{19}, \mathbf{q}_{23}, \mathbf{q}_{24}, \mathbf{q}_{25}, \mathbf{q}_{26}, \mathbf{q}_{27}, \mathbf{q}_{28}, \mathbf{q}_{29}, \cdots, \mathbf{q}_{89} \rangle \leftrightarrow$
$\langle \mathbf{q}_{21}, \mathbf{q}_{31}, \mathbf{q}_{41}, \mathbf{q}_{51}, \mathbf{q}_{61}, \mathbf{q}_{71}, \mathbf{q}_{81}, \mathbf{q}_{91}, \mathbf{q}_{32}, \mathbf{q}_{42}, \mathbf{q}_{52}, \mathbf{q}_{62}, \mathbf{q}_{72}, \mathbf{q}_{82}, \mathbf{q}_{92}, \cdots, \mathbf{q}_{98} \rangle$

Note that **B1** is an extension of symmetry **B** found for NQueens[8] that matches the same pattern $\mathbb{DS}[1,2]$, and **A1** is a new symmetry that can be matched to the same pattern ($\mathbb{DI}[2]$) as **A**. Since again no extra patterns are found during phase two of the second step of the analysis, $\mathit{Candidates}_9 = \mathit{Candidates}_8 = \{\mathbb{DI}[2], \mathbb{DS}[1,2]\}$.

The candidate sets found for $N = 10$ and $N = 11$ are, however, different because Saucy returns different generating sets. In both instances, one of the elements of the generating set does not match any of our patterns, resulting in $\mathit{Candidates}_{10} = \{\mathbb{DI}[2]\}$ and $\mathit{Candidates}_{11} = \{\mathbb{DI}[1], \mathbb{DI}[2]\}$.

In phase three, the algorithm checks the elements in $\bigcup_{d \in V} \mathit{Candidates}_d = \{\mathbb{DI}[1], \mathbb{DI}[2], \mathbb{DS}[1,2]\}$ to see whether they occur in every instance's symmetry group. For $\mathbb{DI}[2]$ this test is trivial because it is found in every $\mathit{Candidates}_d$ set and, thus, it is added to Likely_0. The other two patterns require using GAP to test for membership in the symmetry group of each instance. Since both $\mathbb{DI}[1]$ and $\mathbb{DS}[1,2]$ are found to be members of these groups,

$Likely_0 = \{\mathbb{DI}[1], \mathbb{DI}[2], \mathbb{DS}[1,2]\}$. Finally, since none of these elements is occluded by the others, *Likely* is the same as $Likely_0$.

These three patterns together generate all symmetries of the problem. All three can be proved by the approach of [24] to be model symmetries.

6.2 Social Golfers

The Social Golfers problem is to build a schedule of W weeks, with G equally-sized groups per week, and N golf players per group, such that each pair of players may play in the same group at most once. The following Golf$[W,G,N]$ model uses $W * G$ set variables p_{wg} where p_{wg} represents the set of players that play in group g during week w:

$$\begin{array}{ll} X[W,G,N] : W \times G \times N \mapsto & \{p_{wg} \mid w \in 1..W, g \in 1..G\} \\ D[W,G,N] : W \times G \times N \mapsto & \wp(1..(N \times G)) \\ C[W,G,N] : W \times G \times N \mapsto & \{|p_{wg}| = N \mid w \in 1..W, g \in 1..G\} \cup \\ & \{|p_{wg_1} \cap p_{wg_2}| = 0 \mid w \in 1..W, g_1, g_2 \in 1..G, g_1 < g_2\} \cup \\ & \{|p_{w_1g_1} \cap p_{w_2g_2}| \leq 1 \mid w_1, w_2 \in 1..W, w_1 < w_2, g_1, g_2 \in 1..G\} \end{array}$$

where $\wp$ denotes the powerset. The first constraint ensures the groups are equally-sized, the second that players play in a single group each week, and the last that each pair of players play in the same group (across the weeks) at most once. The literals in every instance of the model can be arranged into a single 3-dimensional matrix with $Dims(W,G,N) = \langle \mathbf{p}(W,G,N*G) \rangle$, where each literal $p_{wg} = k$ is uniquely identified as $\mathbf{p}_{wgk}$. For set variables, the full assignments graph uses an alternative Boolean representation, where a node is created for every possible value a in the set variable x, plus an extra node for a dummy value representing the empty set. Then, literal $x = a$ represents that value a appears in the set assigned to variable x. Therefore, for Social Golfers, a literal identified by $\mathbf{p}_{wgx}$ represents $x \in p_{wg}$, i.e. that player x is in group g of week w. The dummy value is 0, e.g. $\mathbf{p}_{110}$ represents the literal $p_{11} = \emptyset$.[4]

As mentioned in Example 12, during the first step of the analysis, and given the initial base tuple $(2,2,2)$, the implementation generates graphs for seven values of d: $(2,2,2)$, $(3,2,2), (4,2,2), (2,2,2), (2,3,2), (2,4,2), (2,2,2), (2,2,3)$, and $(2,2,4)$. For tuple $(2,2,2)$ Saucy returns the set $Generating_{(2,2,2)} = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}\}$ where:

A $\langle \mathbf{p}_{113}, \mathbf{p}_{123}, \mathbf{p}_{213}, \mathbf{p}_{223} \rangle \leftrightarrow \langle \mathbf{p}_{114}, \mathbf{p}_{124}, \mathbf{p}_{214}, \mathbf{p}_{224} \rangle$
B $\langle \mathbf{p}_{112}, \mathbf{p}_{122}, \mathbf{p}_{212}, \mathbf{p}_{222} \rangle \leftrightarrow \langle \mathbf{p}_{113}, \mathbf{p}_{123}, \mathbf{p}_{213}, \mathbf{p}_{223} \rangle$
C $\langle \mathbf{p}_{111}, \mathbf{p}_{121}, \mathbf{p}_{211}, \mathbf{p}_{221} \rangle \leftrightarrow \langle \mathbf{p}_{112}, \mathbf{p}_{122}, \mathbf{p}_{212}, \mathbf{p}_{222} \rangle$
D $\langle \mathbf{p}_{110}, \mathbf{p}_{111}, \mathbf{p}_{112}, \mathbf{p}_{113}, \mathbf{p}_{114} \rangle \leftrightarrow \langle \mathbf{p}_{120}, \mathbf{p}_{121}, \mathbf{p}_{122}, \mathbf{p}_{123}, \mathbf{p}_{124} \rangle$
E $\langle \mathbf{p}_{210}, \mathbf{p}_{211}, \mathbf{p}_{212}, \mathbf{p}_{213}, \mathbf{p}_{214} \rangle \leftrightarrow \langle \mathbf{p}_{220}, \mathbf{p}_{221}, \mathbf{p}_{222}, \mathbf{p}_{223}, \mathbf{p}_{224} \rangle$
F $\langle \mathbf{p}_{110}, \mathbf{p}_{111}, \mathbf{p}_{112}, \mathbf{p}_{113}, \mathbf{p}_{114}, \mathbf{p}_{120}, \mathbf{p}_{121}, \mathbf{p}_{122}, \mathbf{p}_{123}, \mathbf{p}_{124} \rangle \leftrightarrow$
$\langle \mathbf{p}_{210}, \mathbf{p}_{211}, \mathbf{p}_{212}, \mathbf{p}_{213}, \mathbf{p}_{214}, \mathbf{p}_{220}, \mathbf{p}_{221}, \mathbf{p}_{222}, \mathbf{p}_{223}, \mathbf{p}_{224} \rangle$

Symmetries **A**, **B** and **C** state that players 3 and 4 can be swapped, and so can 2 and 3, and 1 and 2, respectively. Symmetries **D** and **E** state that groups 1 and 2 can be swapped within week 1 and within week 2, respectively. Finally, symmetry **F** states that weeks 1 and 2 can be swapped.

In phase one of the second step of the analysis, these symmetries are matched to the following patterns:

[4] The dummy value is required for the instance symmetry detection method; without it, a literal with the value of empty set would have no edges in the symmetry detection graph and would not appear to participate in any constraint.

A	matches $\mathbb{IS}[3,3,4]$:	$\mathbf{p}_{ij3} \leftrightarrow \mathbf{p}_{ij4}, \forall i \in 1..W, j \in 1..G$
B	matches $\mathbb{IS}[3,2,3]$:	$\mathbf{p}_{ij2} \leftrightarrow \mathbf{p}_{ij3}, \forall i \in 1..W, j \in 1..G$
C	matches $\mathbb{IS}[3,1,2]$:	$\mathbf{p}_{ij1} \leftrightarrow \mathbf{p}_{ij2}, \forall i \in 1..W, j \in 1..G$
D	matches $\mathit{\Pi}[1,1,\mathbb{IS}[2,1,2]]$:	$\mathbf{p}_{11k} \leftrightarrow \mathbf{p}_{12k}, \forall k \in 1..N \times G$
E	matches $\mathit{\Pi}[1,2,\mathbb{IS}[2,1,2]]$:	$\mathbf{p}_{21k} \leftrightarrow \mathbf{p}_{22k}, \forall k \in 1..N \times G$
F	matches $\mathbb{IS}[1,1,2]$:	$\mathbf{p}_{1jk} \leftrightarrow \mathbf{p}_{2jk}, \forall j \in 1..G, k \in 1..N \times G$

Since for tuple $(2,2,2)$ the first two dimensions have only two values, any symmetry that matches an $\mathbb{IS}$ pattern on dimension 1 or 2 also matches a $\mathbb{DI}$ pattern on the same dimension. Thus, **D**, **E** and **F** also match $\mathit{\Pi}[1,1,\mathbb{DI}[2]]$, $\mathit{\Pi}[1,2,\mathbb{DI}[2]]$ and $\mathbb{DI}[1]$, respectively. We will call this a redundant $\mathbb{DI}$ match. As a result, at the end of phase one $Candidates_{(2,2,2)}$ is defined as $\{\mathbb{IS}[1,1,2], \mathbb{DI}[1], \mathbb{IS}[3,1,2], \mathbb{IS}[3,2,3], \mathbb{IS}[3,3,4], \mathit{\Pi}[1,1,\mathbb{IS}[2,1,2]], \mathit{\Pi}[1,1,\mathbb{DI}[2]], \mathit{\Pi}[1,2,\mathbb{IS}[2,1,2]], \mathit{\Pi}[1,2,\mathbb{DI}[2]]\}$.

In phase two, the patterns matched for **A**, **B** and **C** are used to add $\overline{\mathbb{IS}}[3]$ to $Candidates_{(2,2,2)}$, stating that all players are interchangeable. The pattern $\mathbb{IS}[1,1,2]$ matched for **F** causes the addition of $\overline{\mathbb{IS}}[1]$, which states that all weeks are interchangeable. The patterns $\mathit{\Pi}[1,1,\mathbb{IS}[2,1,2]]$ and $\mathit{\Pi}[1,2,\mathbb{IS}[2,1,2]]$, matched for **D** and **E** respectively, each cause the addition of $\mathit{\Pi}[1,1,\overline{\mathbb{IS}}[2]]$ and $\mathit{\Pi}[1,2,\overline{\mathbb{IS}}[2]]$. And these two projections cause $\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$ to be added, stating the interchangeability of groups within each week. Therefore, at the end of phase two $\overline{\mathbb{IS}}[1], \overline{\mathbb{IS}}[3]$, $\mathit{\Pi}[1,1,\overline{\mathbb{IS}}[2]], \mathit{\Pi}[1,2,\overline{\mathbb{IS}}[2]]$ and $\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$ are added to $Candidates_{(2,2,2)}$.

The generating sets for the other instances Golf$[3,2,2]$, Golf$[4,2,2]$, Golf$[2,3,2]$, Golf$[2,2,3]$, Golf$[2,4,2]$ and Golf$[2,2,4]$ include the extended versions of symmetries **A** to **F** above (with identical matches except for the lack of redundant $\mathbb{DI}[2]$ patterns for Golf$[2,3,2]$ and Golf$[2,4,2]$), plus additional symmetries representing the interchangeability of the extra weeks, extra players, and extra groups. These additional symmetries allow the implementation to detect the same extra patterns as before. For example, at the end of phase two, $Candidates_{(2,3,2)} = \{\mathbb{IS}[1,1,2], \mathbb{DI}[1], \mathbb{IS}[3,1,2], \mathbb{IS}[3,2,3], \mathbb{IS}[3,3,4], \mathbb{IS}[3,4,5], \mathbb{IS}[3,5,6], \mathit{\Pi}[1,1,\mathbb{IS}[2,1,2]], \mathit{\Pi}[1,1,\mathbb{IS}[2,2,3]], \mathit{\Pi}[1,2,\mathbb{IS}[2,1,2]], \mathit{\Pi}[1,2,\mathbb{IS}[2,2,3]], \overline{\mathbb{IS}}[1], \overline{\mathbb{IS}}[3], \mathit{\Pi}[1,1,\overline{\mathbb{IS}}[2]], \mathit{\Pi}[1,2,\overline{\mathbb{IS}}[2]], \overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]\}$.

During step three, all patterns of the form $\mathbb{IS}[1,\_,\_]$ or $\mathbb{IS}[3,\_,\_]$ that pass the OCCURS check are eliminated since they are occluded by $\overline{\mathbb{IS}}[1]$ and $\overline{\mathbb{IS}}[3]$, respectively. Similarly, all patterns of the form $\mathit{\Pi}[1,1,\mathbb{IS}[2,\_,\_]]$ or $\mathit{\Pi}[1,2,\mathbb{IS}[2,\_,\_]]$ are eliminated due to $\mathit{\Pi}[1,1,\overline{\mathbb{IS}}[2]]$ and $\mathit{\Pi}[1,2,\overline{\mathbb{IS}}[2]]$, respectively. Finally, all patterns of the form $\mathit{\Pi}[1,1,\overline{\mathbb{IS}}[2]]$ or $\mathit{\Pi}[1,2,\overline{\mathbb{IS}}[2]]$ are eliminated due to $\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$. As a result, the implementation leaves in *Likely* the following patterns, which form all the actual symmetries of the problem:

$\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$: the groups of players are interchangeable within each week (from symmetries **D, E**).
$\overline{\mathbb{IS}}[3]$: the players are interchangeable (from **A,B,C**).
$\overline{\mathbb{IS}}[1]$: the weeks are interchangeable (from **F** and GAP consultation for Golf$[3,2,2]$ and Golf$[4,2,2]$).

The two $\overline{\mathbb{IS}}$ likely candidates can be proved by the approach of [24], while the other cannot. However, it can prove $\overline{\mathbb{IS}}[2]$, which is a subgroup of the symmetries represented by the detected $\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$ element.

This problem is an example of how a combination of patterns can capture a wreath product. In this case, it is the wreath product of the symmetry within each each week ($\overline{\mathit{\Pi}}[1,\overline{\mathbb{IS}}[2]]$) and the symmetry on the weeks themselves ($\overline{\mathbb{IS}}[1]$).

6.3 Golomb ruler

This problem and its model Golomb$[N]$ are described in Example 9, and has $Dims(N) = \langle m(N,N^2+1), d(\frac{N(N-1)}{2},N^2)\rangle$, with each literal $m_i = j$ and $d_i = j$ being uniquely identified as $\mathbf{m}_{ij}$ and $\mathbf{d}_{ij}$, respectively. The generating set found by Saucy for graph Golomb[3] is:

A $\langle \mathbf{d}_{11},\mathbf{d}_{12},\mathbf{d}_{13},\mathbf{d}_{14},\mathbf{d}_{15},\mathbf{d}_{16},\mathbf{d}_{17},\mathbf{d}_{18},\mathbf{d}_{19}\rangle \leftrightarrow$
$\langle \mathbf{d}_{31},\mathbf{d}_{32},\mathbf{d}_{33},\mathbf{d}_{34},\mathbf{d}_{35},\mathbf{d}_{36},\mathbf{d}_{37},\mathbf{d}_{38},\mathbf{d}_{39}\rangle$ plus
$\langle \mathbf{m}_{10},\mathbf{m}_{11},\mathbf{m}_{12},\mathbf{m}_{13},\mathbf{m}_{14},\mathbf{m}_{15},\mathbf{m}_{16},\mathbf{m}_{17},\ldots,\mathbf{m}_{24}\rangle \leftrightarrow$
$\langle \mathbf{m}_{39},\mathbf{m}_{38},\mathbf{m}_{37},\mathbf{m}_{36},\mathbf{m}_{35},\mathbf{m}_{34},\mathbf{m}_{33},\mathbf{m}_{32},\ldots,\mathbf{m}_{25}\rangle$

which reflects the lengths of the spaces between the marks, i.e. turns the ruler back-to-front. This symmetry involves variables from two separate matrices, **d** and **m**, and our simple implementation cannot yet handle this. But even if we only consider the literals in the **m** matrix, the implementation would need to match the symmetry to the parameterised permutation $\mathbf{m}_{iv} \mapsto \mathbf{m}_{(N-i+1)(N^2-v)}$ with $i \in 1..N, v \in 0..N^2$, which performs two different reflections in two different dimensions, something our implementation does not currently detect (unless each of the two reflections is itself a symmetry of the problem, as it will then be obtained by composing the two). Therefore, this model symmetry is not detected. As shown in Example 20, one could provide a different model where the symmetry can indeed be detected by our implementation. However, we believe such model is less natural and, thus, less common.

7 Results

This section evaluates the accuracy and practicality of our implementation of the symmetry detection method described in Section 4. The evaluation is performed over a set of problems that includes those discussed earlier, plus the following problems (some of which are described in CSPLib [15]). The MiniZinc models used for these problems will be available on the *Constraints* journal's editor's page (currently `http://www.crt.umontreal.ca/~pesant/Constraints/constraints.html`).

Balanced Incomplete Block Design (CSPLib problem 28): with parameters (v,b,k,r,λ), where the task is to arrange v objects into b blocks such that each block has exactly k objects, each object is in exactly r blocks, and every pair of objects occurs together in λ blocks. The model has a $v \times b$ matrix of Boolean variables, where variable x_{ij} is true if and only if object i is included in block j. Thus, $Dims((v,b,k,r,\lambda)) = \langle \mathbf{x}(v,b,2)\rangle$. Note that parameters b and r can be derived from the other three and so we specify only (v,k,λ) in Table 1. The objects are interchangeable [$\mathbb{IS}[1]$], as shown in the table, and the blocks are interchangeable [$\mathbb{IS}[2]$].

Graceful Graph (general): with a graph $G = (V,E)$ as a parameter, where the task is to label each vertex in V with a different value between 0 and $|E|$. Also, each edge (u,v) has an induced label $|u-v|$, and all edges must have different labels. The model has a variable for each vertex. Thus, $Dims((V,E)) = \langle \mathbf{x}(|V|,|E|+1)\rangle$. The only model symmetry is that the values are reversible, which matches $\mathbb{DI}[2]$.

Graceful Graph ($\mathbf{K_n} \times \mathbf{P_m}$): the same problem as the previous one, but for a particular kind of graph: the Cartesian product of K_n (complete graph of n vertices) and P_m (a path of m vertices). The model uses an $n \times m$ matrix of variables for the vertices, such that each column corresponds to a single K_n. Thus, $Dims((n,m)) = \langle \mathbf{x}(n,m,\frac{nm(n-1)}{2}+n(m-1)+1)\rangle$. In this version of the problem, there are additional symmetries thanks to the structure of the graph: the corresponding vertices in each clique (dimension 1) are simultaneously interchangeable [$\mathbb{IS}[1]$] and the order of the cliques (dimension 2) is reversible [$\mathbb{DI}[2]$].

(N $\times$ N) queens (not to be confused with N-queens): where an $N \times N$ chessboard is coloured with N colours, so that a pair of queens placed on any two squares of the same colour would not attack each other. The model uses an $N \times N$ matrix of integer variables, where the values (colours) range from $1..N$. Thus, $Dims(N) = \langle \mathbf{x}(N,N,N) \rangle$. The symmetries are those of the chessboard (variable rotations and reflections) [$\mathbb{DI}[1]$,$\mathbb{DS}[1,2]$], and the colours (values) are interchangeable [$\overline{\mathbb{IS}}[3]$].

N-queens (bool): the same problem as described in Section 6, but using an $N \times N$ matrix of Boolean variables for the model. Each variable x_{ij} is true if there is a queen in the square at row i, column j. Thus, $Dims(N) = \langle \mathbf{x}(N,N,2) \rangle$. The symmetries are those of the chessboard (variable rotations and reflections) [$\mathbb{DI}[1]$,$\mathbb{DS}[1,2]$].

Steiner Triples (CSPLib problem 44): where the task is to find $m = \frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, such that any pair of triples has at most one element in common. The model uses an array of m set variables, where the elements of each set are drawn from $1..n$. Thus, $Dims(n) = \langle \mathbf{x}(\frac{n(n-1)}{6}, n) \rangle$. The symmetries are that the triples (variables) are interchangeable [$\overline{\mathbb{IS}}[1]$] and the values are interchangeable [$\overline{\mathbb{IS}}[2]$].

Steel Mill Slab Design (CSPLib problem 38): where a set of orders, each having a weight and a colour, are to be assigned to slabs. The orders assigned to a single slab may have up to two different colours. Each slab has a size chosen from a fixed set; the total weight of the orders must not exceed the slab's size. Any unused slab capacity is to be minimised. The model uses an array of n integer variables, where n is the number of orders. The value of each variable is the slab to which that order is assigned. While in this case an element d of *Data* is not a simple integer tuple, we can still define $Dims(d) = \langle \mathbf{x}(n,s) \rangle$, where n is the number of orders and s is the number of slabs. The slabs (values) are interchangeable [$\overline{\mathbb{IS}}[2]$].

Scene Allocation: where movie scenes are to be scheduled to minimise actors' fees [35]. Each scene requires a subset of the actors and at most five scenes may be shot per day. Each actor has a fixed daily fee payable for every day in which one of their scenes is scheduled, but there is no penalty for having an actor work non-consecutive days. The model uses an array of n integer variables, where n is the number of scenes, and the value of each variable is the day on which that scene is scheduled. Again, while an element d of *Data* is a complex data tuple, we can still define $Dims(d) = \langle \mathbf{x}(n,m) \rangle$, where n is the number of scenes and m is the number of days. The days (values) are interchangeable [$\overline{\mathbb{IS}}[2]$].

HP 2D-Protein Folding [18, 16]: where a sequence of amino acids is to be laid out on a 2D grid that minimises the energy of the resulting structure. Each amino acid is labeled either H or P, and the aim is to maximise the number of pairs of H amino acids that are adjacent. The sequence must not overlap itself. We use a model similar to that of Mancini and Cadoli [18], where the value of decision variable x_i (corresponding to amino acid i in the sequence) is the direction north, south, east or west w.r.t. x_{i-1}. Since an element d of *Data* is a sequence of H or P elements, we can define $Dims(d) = \langle \mathbf{x}(n,4) \rangle$, where n is the number of amino acids. The symmetries are those of the 2D grid, that is, value symmetries that reflect the plane [$\mathbb{IS}[2,1,3]$, $\mathbb{IS}[2,2,4]$] and a value symmetry that rotates the plane (and which does not match any pattern).

Table 1 shows the results of the evaluation, where the columns indicate the problem name, the base tuple (if any) used for generating the instances and the maximal amount by which each component in the base tuple is increased, the symmetries of a known generating set with their associated patterns in brackets (if any matching pattern exists), whether the pattern appears in *Likely* ("✓") or not ("-") , whether it is proven by the implementation of [24] ("P") or not ("-"), the total analysis time in seconds, and the percentage of the total

Table 1 Symmetry detection results.

Problem	Base + Inc.	Symmetries		Time (s)	Inst. Det.	Graph V/E
BIBD	(2,2,2)+2	objects [$\overline{\mathbb{IS}}[1]$]	✓P	1.0	86	5760/56448
		blocks [$\overline{\mathbb{IS}}[2]$]	✓P			
Social Golfers	(2,2,2)+2	weeks [$\overline{\mathbb{IS}}[1]$]	✓P	72.8	100	542768/3829496
		groups [$\overline{\Pi}[1,\overline{\mathbb{IS}}[2]]$]	✓-			
		players [$\overline{\mathbb{IS}}[3]$]	✓P			
Golomb Ruler	(3)+3	flip	- -	3.2	96	40293/88467
Graceful Graph ($K_n \times P_m$)	(2,2)+3	intra-clique [$\overline{\mathbb{IS}}[1]$]	✓P	5.8	57	72580/157690
		path-reverse [$\mathbb{DI}[2]$]	✓-			
		value [$\mathbb{DI}[3]$]	✓P			
Latin Square	(2)+3	rows/columns [$\mathbb{DS}[1,2]$]	✓P	0.1	27	875/1500
		columns/values [$\mathbb{DS}[2,3]$]	✓-			
		value [$\overline{\mathbb{IS}}[3]$]	✓P			
N × N queens	(4)+3	chessboard [$\mathbb{DI}[1]$,$\mathbb{DS}[1,2]$]	✓P	0.3	57	4704/8722
		colours [$\overline{\mathbb{IS}}[3]$]	✓P			
N-queens (int)	(8)+3	chessboard [$\mathbb{DI}[1]$,$\mathbb{DS}[1,2]$]	✓P	0.2	17	2101/3960
N-queens (bool)	(8)+3	chessboard [$\mathbb{DI}[1]$,$\mathbb{DS}[1,2]$]	✓P	0.7	54	977/5400
Steiner Triples	(4)+3	triples [$\overline{\mathbb{IS}}[1]$]	✓P	0.4	54	2786/8281
		value [$\overline{\mathbb{IS}}[2]$]	✓P			
Scene Allocation	(1)+1	days [$\overline{\mathbb{IS}}[2]$]	✓-	0.3	19	1386/8664
Steel Mill	(1)+1	slabs [$\overline{\mathbb{IS}}[2]$]	✓P	4.6	99	26255/114062
Graceful Graph (general)	(1)+1	value [$\mathbb{DI}[2]$]	✓-	14.3	87	80355/172040
Protein Folding	(1)+1	horiz. reflection [$\mathbb{IS}[2,1,3]$]	✓-	2.6	95	35448/81033
		vert. reflection [$\mathbb{IS}[2,2,4]$]	✓-			
		rotation	- -			

analysis time that is spent in detecting instance symmetries. The final column lists the size (vertices/edges) of the full assignments graph – which is used to find instance symmetries – of the largest instance for each model. Note that for the last four benchmarks in the table, where the parameters cannot be automatically generated, we created data that resulted in several small instances. For these four benchmarks, the base and increment values refer to the instance identifiers rather than the actual instance data. The experiments were run using MiniZinc version 1.5 on an Intel Core 2 3.33GHz computer with 4GB of memory. No effort has been made to optimise detection time; the times are included simply to show the practicality of the approach.

The results show that the approach is capable of finding almost all symmetries in this set of problems. The constraints used in each problem vary, from many simple constraints to global constraints. In particular, all symmetries are found in the two quite different representations of the N-queens problem. Note that the system correctly added to *Likely* all but two members of a generating set, even though the instances used were very small. In the Social Golfers and BIBD problems, where the detection time is largest, most of the time is spent in detecting instance symmetries. For the Latin square problem, the symmetry between columns and values cannot be proved for this model, although it can be proved for a different model with Boolean variables instead of integer variables.

8 Detecting Other Properties

Our method for detecting symmetries can be generalised to other properties as follows:

1. For each $d \in V$, detect the set of properties $Properties_d$ of instance $P[d]$.
2. For each $d \in V$, lift each $p \in Properties_d$ to the model, obtaining the set of candidate model properties $Candidates_d$.
3. Create a subset *Likely* of $\bigcup_{d \in V} Candidates_d$ containing only those elements that are *likely* to be model properties.
4. Prove whether the elements of *Likely* are indeed model properties or not.

In the next subsection we show how the above generalised method can be used to detect conditions under which two subproblems are known to be equivalent. This information can be used by a caching search to significantly speed up the execution [31, 12]. See [22] for an actual implementation of this application of the generalised method. Note that determining whether such conditions occur in a model is a difficult problem for which there is no other automatic method, although a method does exist for instances [4].

8.1 Applying the Generalised Method to Caching

Caching modifies a search algorithm by storing the results of exploring some subtrees and reusing them wherever possible. A subtree q does not need to be explored if a previously explored subtree p is known to contain all the information that might be found in q; that is, any solution found in q is equivalent to a solution in p (and vice versa). If this is the case, we say that p and q are equivalent.

Example 21 Consider the CSP (X,D,C) where $X = \{a,b,c,d\}$, $D(a) = 1..3$, $D(b) = 1..4$, $D(c) = 1..5, D(d) = 1..6$ and the only constraint is $a+b+c+d = 10$ (see Figure 6). The search may proceed by asserting $a = 1$ and $b = 4$, leading to a subtree T_1. In this subtree there is the subproblem induced by the earlier assignments where $c+d$ must be made to equal 5. Later in the search, the initial decisions may be undone and instead the search may try $a = 2$ and $b = 3$, leading to a subtree T_2. The two subtrees T_1 and T_2 are equivalent, because every assignment over the variables $\{c,d\}$ that leads to a solution in T_1 also leads to a solution in T_2 (and vice versa). Therefore, we can cache the solutions for T_1 and reuse them for T_2.

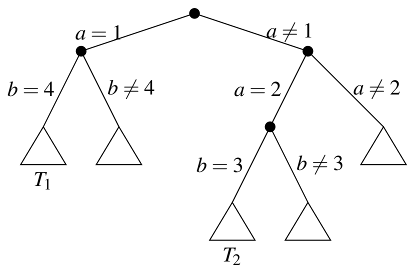

Fig. 6 Part of a search tree showing equivalent subtrees T_1 and T_2.

□

The proposed generalised method is modified to detect equivalent subproblems as follows. In the first step, we construct the search tree resulting from solving each initial instance. This task corresponds to the generation of the full-assignment graph for detecting symmetries. Then, we transform the tree into a directed acyclic graph, where identical subtrees are merged and, thus, easily identified. This task corresponds to calling Saucy to detect automorphisms of the graph. In the second step, we match each of a set of patterns with the information contained in every merged node. Patterns include permutations of the values of already assigned variables, domain of unassigned variables, and subsets of remaining constraints. In the third step, we eliminate from the candidate patterns every one that applies to two non-merged nodes in the graph. The fourth step attempts to prove that each likely candidate applies to the model. This last step is currently done manually. As shown in [22], the method successfully detects all known candidates in a reasonably wide set of benchmarks.

8.2 Limitations of the Generalised Method

Our generalised method is suitable for the detection of properties for which *good* patterns can be defined. We say a pattern is good if it directly depends on the structure of the instance (i.e. the relationships established by the constraints on its variables and values) and can be easily checked. For example, the generalised method can be applied to finding implied global constraints. In this case the patterns are the global constraints themselves (represented, for example, as their associated full assignment graphs) and the structure of the instance is also the constraint graph. The pattern occurs in the instance if the global constraint appears as a sub-graph of the instance's graph.

Theoretically, the generalised method is not affected by the particular choice of constraints used in the model, since it can be implemented using a complete instance-based detection method that does not depend on this. In practice, however, this might be impractical. The generalised method is sensitive, however, to the way in which variables are specified in the model, since they determine the way in which the structure of the problem is represented. This structure is the basis to define the patterns, and some structures might make it easier to detect properties than others

9 Related Work

There are three areas of research related to this paper. Firstly, there is a body of work on static analysis to prove properties of programs (symmetry being just one such property). Secondly, there is a smaller body of literature on the discovery of properties of problem models from sets of their instances. And thirdly, there is a substantial amount of research literature on symmetry detection. We discuss each of these in turn.

9.1 Static Program Analysis

Static program analysis is the process of inferring information at compile-time about the run-time properties of the program. *Abstract interpretation* [7], one of the most popular static analysis approaches, works by abstracting the data to encode the property of interest, abstracting the operations on the data, and then running the program using these abstractions. Abstract interpretation has been successfully applied to a wide variety of programs and programming paradigms (see for example, [1, 37], and previous editions of the same events), including Constraint Logic Programs (e.g. [13, 28, 19]).

Since the program inputs are only supplied at runtime, static analysis is data-independent and, in this sense, it is similar to the inference of properties of CSP models. Unfortunately, for many properties the conjunction of constraints and the domain of the variables have a complex effect on the presence or absence of the properties. Therefore, the abstract conjunction operation will lose considerably accuracy. This is indeed the case when inferring symmetries (see Section 9.3 for an example of the rapid loss of accuracy when static analysis is applied to CSP models). Our method avoids this loss of accuracy by using concrete (rather than abstract) data to infer its information.

9.2 From Properties of Instances to Properties of Models

The machine learning community has for many years used information learnt about instances of a model (or class) to infer information likely to hold for new instances of the model. This type of inductive inference has been used for algorithm selection [32], among other purposes.

Inductive reasoning from a set of problem instances was also proposed in [3] where a theorem prover generates additional constraints for each instance, and it is conjectured that constraints that hold for all the instances in the set tested so far, hold for all instances of the problem model. A theorem prover is then used to attempt to prove that the additional constraint holds not just for specific problem instances, but for the problem model itself. The HR theorem prover used in [3] requires the problem model to be axiomatised as a "basic model" in first-order logic. Any first order formula has a fixed number of variables. Moreover any constraint has a fixed arity, of course. Consequently, the basic model cannot use a representation where the number of variables in the model and in the scope of a global constraint depends on a parameter. Such a representation is required for both the definition and the recognition of model symmetries. Moreover, the basic model for the HR theorem prover is independent of the size of any problem instance, so no property that involves the size of the instance (or one of its dimensions – as in symmetry pattern $\mathbb{DI}$) – could be derived as a possible theorem.

Colton and Sorge [6] investigate a method for distinguishing all finite algebras of a given size by eliciting a minimal set of *element-type* properties sufficient to separate each pair of instances. An element-type property is a property of an element, independent of the size of the algebra. Consequently, it is possible for the same set of element-type properties to distinguish instances of algebras of different sizes. An aim is to find minimal sets of element-properties sufficient to distinguish all finite algebras of size up to a given maximum size. Our research also focuses on *model* properties which are independent of the size of the instance, though our properties are not element-type and the class of models we address is wider.

9.3 Symmetry Detection

Van Hentenryck et al. [36] propose to define the symmetries of individual constraints, and then combine these symmetries to obtain the symmetries of the model. To achieve this, they first define the composition of two CSPs as follows:

Definition 3 Let $P_1 = (X_1, D_1, C_1)$ and $P_2 = (X_2, D_2, C_2)$ be two CSPs. Then, their composition $P_1 \wedge P_2$ is $(X_1 \cup X_2, D_3, C_1 \wedge C_2)$ where D_3 is the intersection of their associated domains.

The key observation that allows some symmetries to be derived by composition is the following (proposition 1 in [36]):

> A value-interchangeable CSP is a CSP where all values are interchangeable. Let $P_1 = (X, D, C_1)$ and $P_2 = (X, D, C_2)$ be two value-interchangeable CSPs. Then, $P_1 \wedge P_2$ is value-interchangeable.

Although the above proposition is defined only for the case in which all values are interchangeable, similar results apply for interchangeable variables, and also when only some values (or variables) of the problem are interchangeable [36]. This approach can detect sets of values (or variables) that are interchangeable, and can be extended to row- and column-interchangeability for matrices of variables. It is, however, unable to capture other kinds of symmetry, such as reflections of a matrix or variable-value symmetries.

Another serious limitation is that symmetries are not inherently compositional. Two subproblems P_1 and P_2 may have no symmetry when considered independently, yet $P_1 \wedge P_2$ may contain symmetries. This kind of symmetry cannot be captured by compositional derivation. For example, in the two subproblems (a) $x_1 \leq x_2$; and (b) $x_2 \leq x_1$, x_1 and x_2 are not interchangeable, yet in the composed problem x_1 and x_2 are interchangeable.

Furthermore, symmetries may exist in subproblems but be lost upon composition. For example, the three subproblems (a) $x_1 \neq x_2$; (b) $x_1 \neq x_3$; and (c) $x_2 \neq x_3$, each have a symmetry that interchanges its variables, and the problem formed by conjoining these subproblems also has those symmetries. However, the symmetries in each problem are different and, therefore, taking their intersection yields no symmetries.

Given the weakness of the proposition above (which requires the variables and domains of P_1 and P_2 to be the same), one could think that a formulation of the model in terms of *global* constraints would be advantageous. However, often this is not enough. For example, formulating the N-queens problem exclusively in terms of global *alldiff* constraints does not enable its symmetries to be detected using the method in [36] because the three *alldiff* constraints in N-queens do not share the same variables (and, thus, symmetry). Further, global constraints may not be available in the modelling language or constraint solver used, the problem may be more easily expressed without them, or the modeler may not have the necessary expertise to use them. Lastly, it is difficult to asses the practicality of their method since, as far as we know, there is no implementation.

The method of Roy and Pachet [30] uses the symmetries of global constraints to automatically detect symmetries in problem instances. Every constraint divides its variables into *intensional permutability* (or IP) classes, where two variables are intensionally permutable if they can be interchanged while leaving the constraints unchanged. Intuitively, two variables are intensionally permutable if they will undergo the same events (i.e. domain reductions) during search. The IP classes of the whole problem are computed by combining the IP classes of the problem's constraints. They also show how their method may be generalised to more complex symmetries that interchange collections of variables by grouping variables into particular structures. It is possible that this approach could be generalised to models. However, no algorithm is given for identifying these structures.

10 Conclusions

We have proposed a new method for automatically detecting model properties. In particular, we have shown how the method can be applied to the task of finding symmetries in constraint satisfaction problem models. It takes advantage of existing, and even future, powerful detection techniques defined for problem instances by generalising their results to models. We have described an incomplete, ad hoc implementation that only considers a limited range of symmetry patterns. While limited, the method is nonetheless capable of detecting all of the symmetries in almost all of the benchmarks we have tested. Of course, more complete implementations of the method will be able to detect even more kinds of symmetries.

Despite the effectiveness of our approach on the benchmark problems, there remain some symmetries that it cannot find. The main source of incompleteness in the implementation is that only known symmetry patterns are found. However, it does find some of the most commonly occurring forms of symmetry, and it is these kinds of symmetry that we believe are the most profitable to exploit in symmetry breaking [20]. The approach is inherently limited to finding symmetries that appear in every instance of a CSP model. Any symmetry that occurs due to the data and therefore appears only in some instances (such as where the data part is a graph that has a symmetry) cannot be found.

We have also discussed the generalisation of the method to properties of constraint models other than symmetries. The approach behind the method is not specific to symmetry, and by replacing its components with others tailored to different properties, the resulting generalised method can be used to detect other useful features of models. One such example is subproblem equivalence, which is used by caching to avoid searching equivalent areas of the search space and reduce search time. Preliminary results from our prototype implementation suggest the generalised method has significant potential for accurately detecting properties of constraint models.

11 Acknowledgements

The authors thank the anonymous reviewers for their thorough and constructive comments. We also thank Todd Niven for helpful discussions.

References

1. María Alpuente, editor. *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*, volume 6564 of *Lecture Notes in Computer Science*. Springer, 2011.
2. Belaid Benhamou. Study of symmetry in constraint satisfaction problems. In *PPCP'94: Second International Workshop on Principles and Practice of Constraint Programming*, pages 246–254, 1994.
3. John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 73–77. IOS Press, 2006.
4. Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Automatically exploiting subproblem equivalence in constraint programming. In Andrea Lodi, Michela Milano,

and Paolo Toth, editors, *CPAIOR*, volume 6140 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2010.

5. David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.
6. S. Colton and V. Sorge. Automated parameterisation of finite algebras. In *Workshop on Empirical Successful Automated Reasoning in Mathematics*, 2008.
7. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.
8. James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *KR*, pages 148–159. Morgan Kaufmann, 1996.
9. Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004.
10. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.
11. Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In Thomas L. Dean and Kathleen McKeown, editors, *AAAI*, pages 227–233. AAAI Press / The MIT Press, 1991.
12. Maria Garcia de la Banda and Peter J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. *INFORMS Journal on Computing*, 19(4):607–617, 2007.
13. Maria J. Garcia de la Banda, Warwick Harvey, Kim Marriott, Peter J. Stuckey, and Bart Demoen. Checking modes of HAL programs. *TPLP*, 5(6):623–668, 2005.
14. Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *ECAI*, pages 599–603. IOS Press, 2000.
15. I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from `http://www.csplib.org/`.
16. Kit Fun Lau and Ken A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Macromolecules*, 22(10):3986–3997, 1989.
17. Y.C. Law, J.H.M. Lee, T. Walsh, and J.Y.K. Yip. Breaking symmetry of interchangeable variables and values. In *Principles and Practice of Constraint Programming - CP 2007*, 2007.
18. Toni Mancini and Marco Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In Zucker and Saitta [38], pages 165–181.
19. Kim Marriott and Peter J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In Mary S. Van Deusen and Bernard Lang, editors, *POPL*, pages 334–344. ACM Press, 1993.
20. C. Mears, M. Garcia de la Banda, B. Demoen, and M. Wallace. Lightweight dynamic symmetry breaking. In *Eighth International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'08*, 2008.
21. Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. On implementing symmetry detection. *Constraints*, 14(4):443–477, 2009.
22. Christopher Mears, Maria Garcia de la Banda, and Mark Wallace. Eliciting sub-problem equivalence candidates for CP models. Technical Report 2011/266, Caulfield School of

Information Technology, Monash University, 2011.
23. Christopher Mears, Maria Garcia de la Banda, Mark Wallace, and Bart Demoen. A novel approach for detecting symmetries in CSP models. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2008.
24. Christopher Mears, Todd Niven, Marcel Jackson, and Mark Wallace. Proving symmetries by model transformation. In Jimmy Ho-Man Lee, editor, *CP*, volume 6876 of *Lecture Notes in Computer Science*, pages 591–605. Springer, 2011.
25. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.
26. Jean-Francois Puget. Symmetry breaking revisited. In Pascal Van Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 446–461. Springer, 2002.
27. Jean-Francois Puget. Automatic detection of variable and value symmetries. In van Beek [34], pages 475–489.
28. Viswanath Ramachandran, Pascal Van Hentenryck, and Agostino Cortesi. Abstract domains for reordering CLP(R_{lin}) programs. *J. Log. Program.*, 42(3):217–256, 2000.
29. Arathi Ramani and Igor L. Markov. Automatically exploiting symmetries in constraint programming. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 3419 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 2004.
30. Pierre Roy and François Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 Workshop on Non-binary Constraints*, pages 27–33, 1998.
31. Barbara M. Smith. Caching search states in permutation problems. In van Beek [34], pages 637–651.
32. Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1):6:1–6:25, 2008.
33. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.9*, 2006.
34. Peter van Beek, editor. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*. Springer, 2005.
35. Pascal Van Hentenryck. Constraint and integer programming in OPL. *INFORMS Journal on Computing*, 14(4):345–372, 2002.
36. Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Compositional derivation of symmetries for constraint satisfaction. In Zucker and Saitta [38], pages 234–247.
37. Eran Yahav, editor. *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*. Springer, 2011.
38. Jean-Daniel Zucker and Lorenza Saitta, editors. *Abstraction, Reformulation and Approximation, 6th International Symposium, SARA 2005, Airth Castle, Scotland, UK, July 26-29, 2005, Proceedings*, volume 3607 of *Lecture Notes in Computer Science*. Springer, 2005.

A MiniZinc models of examples

Example 1 (Latin square).

```
int : N = 3 ;
array [1..N, 1..N] of var 1..N : x ;
constraint forall (i,j in 1..N, k in j+1..N) (x[i,j] != x[i,k]) ;
constraint forall (i,j in 1..N, k in j+1..N) (x[j,i] != x[k,i]) ;
```

Example 9 (Golomb ruler).

```
int : N ;
array [1..N] of var 0..N*N : m ;
array [1..N*(N-1) div 2] of var 1..N*N : d =
  [ m[j]-m[i] | i,j in 1..N where i<j ];
constraint forall (i,j in 1..N*(N-1) div 2 where i<j) (d[i] != d[j]);
```

Example 19 (Variant of Latin square).

```
int : N ;
array [1..N*N] of var 1..N : x ;
constraint forall (i,j in 1..N*N where i < j / (i-1) mod N = (j-1) mod N)
  (x[i] != x[j]) ;
constraint forall (i,j in 1..N*N where i < j / (i-1) div N = (j-1) div N)
  (x[i] != x[j]) ;
```

Example 20 (Variant of Golomb ruler).

```
int : N ;
array [1..N-1] of var 1..N*N : d;
constraint forall (i,j,k,l in 1..N where i<j / k<l / (i != k  j != l))
  (sum (a in i..j-1) (d[a]) != sum (a in k..l-1) (d[a])) ;

solve satisfy;
output [show(d)];
```

N-queens.

```
int : N ;
array [1..N] of var 1..N : q ;
constraint forall (i in 1..N, j in i+1..N) (q[i]   != q[j]);
constraint forall (i in 1..N, j in i+1..N) (q[i]+i != q[j]+j);
constraint forall (i in 1..N, j in i+1..N) (q[i]-i != q[j]-j);
```

Social Golfers.

```
int : N ;
int : W ;
int : G ;
array [1..W, 1..G] of var set of 1..N*G : p ;
constraint forall (w in 1..W, g in 1..G) (card(p[w,g]) = N) ;
constraint forall (w in 1..W, g1,g2 in 1..G where g1<g2)
  (card(p[w,g1] intersect p[w,g2]) = 0) ;
constraint forall (w1,w2 in 1..W, g1,g2 in 1..G where w1<w2)
  (card(p[w1,g1] intersect p[w2,g2]) <= 1) ;
```