Constraint Patterns

Toby Walsh

Cork Constraint Computation Center, University College Cork, Ireland.
tw@4c.ucc.ie

Abstract. Constraint models contain a number of common patterns. For example, many constraint models involve an array of decision variables with symmetric rows and/or columns. By documenting such constraint patterns, we can share modelling expertise. Constraint solvers can also be extended to exploit such patterns. For example, we can develop specialized methods like the global lexicographical ordering constraint for breaking such row and column symmetry.

1 Introduction

Many patterns occur in constraint programs. In this paper, I argue that we need to identify, formalize and document these patterns in a similar way to the design patterns identified by the software engineering community. The result will be a systematic and comprehensive methodology for modelling an informal problem. Such a methodology will permit us to tackle the modelling "bottleneck" that hinders the uptake of constraint programming. This is an ambitious project - modelling is not a task which lends itself to a piecemeal approach as even the smallest modelling decision can have far reaching consequences. However, we have made some progress and I describe some of the more interesting constraint patterns which have already been identified. I also discuss the different ways that we can exploit such patterns. For example, one way to exploit common constraint patterns is to extend the constraint language.

2 Design patterns

Patterns are an approach to design that started in architecture [1], which has since spread to many other areas including software engineering [2]. A pattern describes not only the context of a problem and its solution, but also the rationale behind the solution. Patterns are a valuable mechanism for describing best practice and good design. Patterns therefore have a useful role to play in software engineering. Designing software is hard. Designing good software is even harder. Fortunately, well engineered code exhibits many common patterns that support extensibility, modularity, and performance. By documenting these patterns, we can support and encourage good software engineering.

I can illustrate this by means of an analogy. Consider how you could become an America's Cup match racing helmsman. First, you learn the sailing rules (e.g. a boat on starboard tack has right of way over a boat on port). Then you learn the basic principles of racing (e.g. when you have the lead, you cover tack to protect that lead). Finally, you study past match races to learn the winning patterns of others (e.g. on a downwind leg, an expert helmsan in a trailing boat often rides an approaching gust, blankets the leading boat's sail, and overtakes on the windward side). Becoming an expert software engineer is little different. First, you learn the rules (e.g. algorithms, and data structures). Then you learn the basic principles (e.g. data abstraction helps code be modular and extensible). Finally, you study expert software engineers to learn valuable patterns (e.g. a good software engineer will often construct iterator methods so that elements of an aggregate object can be accessed without exposing the underlying representation).

Patterns are, by their very nature, not formal objects. They are therefore usually documented in natural language. A pattern descriptions typically includes the following category headings (as well as others that may be more domain specific).

Pattern Name:	A meaningful name for the pattern.
Context:	The circumstances in which the problem the pattern solves occurs.
Problem:	The specific problem that is solved.
Forces:	The often opposing considerations that must be taken into account when choosing a solution.
Solution:	The proposed solution to the problem.
Example:	An example of the problem and its resolution using the proposed solution.

Fig. 1. The typical categories used in specifying a pattern.

3 Constraint patterns

Why should we identify and document patterns in constraint programming? Constraint programming is programming and so many of the usual software engineering issues arise. However, constraint programming is also about modelling. There are many recurring patterns in good constraint models. These patterns cannot usually be precisely specified as there are many conflicting interactions in a complex problem. Patterns therefore seem a good vehicle for explicitly capturing the knowledge of expert modellers.

What benefits do constraint patterns bring? First, they help tackle the modelling "bottleneck". A library of patterns would be a valuable resource for passing on modelling expertise to neophyte constraint programmers. Second, constraint toolkits can be extended to support commonly occurring patterns. As I argue in Section 4.2, a constraint pattern can motivate the development of a new global constraint or language feature. Third, a longer term goal would be pattern automation. For example, in Sections 4.4 and 4.5, I discuss how we are trying to automate some common constraint patterns.

What drawbacks do constraint patterns have? First, it is hard work to identify and document good patterns. It requires the efforts of a whole community, not just of one individual. Second, patterns do not eliminate all the art of constraint modelling. Many problems also contain an unique feature or an unique combination of features which necessitates a special solution technique that will not be not of a pattern library. Third, patterns are not executable. However, as I argued above, we can look to automate aspects of them.

4 Some examples

To illustrate what a constraint pattern is, and how it can be useful, I will look at four examples from my own and other people's research.

4.1 Matrix models

Before I describe the first pattern, I want to define the context for a number of common constraint patterns. A *matrix model* is a constraint program with one or more matrices of decision variables. For example, a natural model of a sports scheduling problem has a 2-d matrix of decision variables, each of which is assigned a value corresponding to the game played in a given week and period [3]. In this case, the matrix is obvious in the solution to the problem: we need a *table* of fixtures. However, as we demonstrate in [4, 5], many problems that are less obviously defined in terms of matrices can be efficiently represented and effectively solved using a matrix model. Sometimes, the matrix model contains multiple matrices of variables. Channelling constraints are then used to link the different matrices together [6–8].

As an example, consider the matrix model given in [4] for the steel mill slab design problem [9]. We have a number of orders, each with a particular weight and colour, to assign to slabs. Slabs come in a number of different sizes. We want to assign orders to slabs and sizes to slabs so that the total weight of orders assigned to a slab does not exceed the slab capacity, and so that each slab contains at most p colours (usually 2). A 2-d matrix of 0/1 decision variables represents which orders are assigned to which slabs. A second matrix of 0/1 decision variables is used to post the colour constraints. Channelling constraints are used to connect this to the order matrix.

In [5], we demonstrated the prevalence of matrix models by surveying the first 31 models in CSPLib. At least 27 of these had natural matrix models, most of them already published. Matrix models are a very natural way to represent relations (e.g. the relation between orders and slabs in the steel mill slab design problem). Matrix models are also a natural way to represent functions (e.g. the function mapping exams to times in an exam timetabling problem). Finally, matrix models are a natural way to represent partitions (e.g. the partitioning

Matrices:

$$\begin{array}{cccccccccc} & \multicolumn{9}{l}{\longrightarrow \text{slabs} \longrightarrow}\\ s_i & 4&3&3&3&0&0&0&0&0\end{array}$$

$$\begin{array}{rccccccccc} & \multicolumn{9}{l}{\longrightarrow \text{orders} \longrightarrow}\\ O_{ij} & 0&0&0&0&0&0&1&1&1\\ \downarrow & 0&0&0&1&1&1&0&0&0\\ \text{slabs} & 0&1&0&0&0&0&0&0&0\\ \downarrow & 1&0&1&0&0&0&0&0&0\end{array}$$

$$\begin{array}{rcccc} & \multicolumn{4}{l}{\longrightarrow \text{colours}}\\ C_{ij} & 0&0&0&1\\ \downarrow & 0&0&1&1\\ \text{slabs} & 0&1&0&0\\ \downarrow & 1&1&0&0\end{array}$$

Constraints:

$$\begin{array}{ll} \text{(row weighted sum)} & \forall j \sum_i Weight_i * O_{ij} \leq s_j\\ \text{(column sum)} & \forall i \sum_j O_{ij} = 1\\ \text{(row sum)} & \forall j \sum_i C_{ij} \leq p\\ \text{(channelling)} & \forall ij\ O_{ij} = 1 \rightarrow C_{Colour_i j} = 1\end{array}$$

Fig. 2. Matrix model for the steel mill slab design problem (taken from [4])

of nodes in a graph colouring problem into different colour classes). Another indication of the prevalence of matrix models is the common use of 2-dimension matrices of decision variables in integer linear programming.

4.2 Row and column symmetry

A common pattern in matrix models is row and column symmetry. For example, the order matrix in the steel mill problem has partial row symmetry since slabs of the same size are indistinguishable and partial column symmetry since orders of the same size and colour are also indistinguishable. We can swap any two slabs of the same size, and any two orders of the same size and colour and obtain an essentially equivalent solution. As a second example, consider generating **Balanced Incomplete Block Designs** or BIBDs (prob028 in CSPLib). A matrix model for this problem uses a 2-d matrix of 0/1 variables, with constraints on the sum of each row and each column, and on the scalar product between rows. This matrix model has complete row and column symmetry since we can permute the rows and columns freely without affecting any of the constraints.

Symmetry in constraint programs is problematic. It increases the search space dramatically. Row and column symmetry is especially problematic as it occurs often and there is a lot of it. For example, an n by m BIBD has $n!m!$ row and column symmetries. Even if are only interested in finding one solution, we may explore many failed and symmetrically equivalent branches. When proving optimality, this can be especially painful. An effective way to break such symmetry is by posting constraints that lexicographically order the rows and columns [10]. We call this **double lex** ordering the matrix. As long as we order the rows and columns in the same direction, (i.e. the rows and columns must both be lexicographically increasing, or must both be lexicographically decreasing), this will leave a solution if one exists. Unfortunately, it does not break all symmetry. Multiple, symmetric solutions can be left after double lex ordering the matrix. Indeed, a paper in this volume [11] proposes an additional ordering constraint, the **all perms** constraint, which can be effectively posted on the rows or columns to eliminate some (but still not all) of the remaining symmetry. To support double lex and all perms ordering constraints, we have developed efficient linear time constraint propagation algorithms [12, 11]. This illustrates how constraint solvers can be extended to support commonly occurring patterns like a global symmetry-breaking constraint. Techniques to eliminate all symmetries exist [13–15], but they appear to be more expensive than they are worth for dealing with row and column symmetries. The symmetries eliminated by double lex ordering and all perms ordering appear to offer a good compromise.

Pattern Name:	`MatrixSymmetry`
Context:	A matrix model containing an array of decision variables with (partial) row and column symmetry.
Problem:	Eliminating (much of) that symmetry.
Forces:	Eliminating all symmetry can be too expensive.
	Eliminating no symmetry can leave too much search.
Solution:	Post `lex` ordering constraints on rows and/or cols.
	If supported, post `all-perms` constraints on rows or cols.
Example:	Balanced Incomplete Block Design generation.

Fig. 3. Constraint pattern for `MatrixSymmetry`

An alternative to double lex ordering is to post lexicographical ordering constraints on the rows, and multiset ordering constraints on the columns (or vice versa). We have also developed an efficient constraint propagation algorithm for multiset ordering constraints [16]. This volume also contains a paper showing how we can effectively post during search just those symmetry breaking constraints that are not yet broken by the current assignment [17]. These examples illustrate how the solution proposed to a constraint pattern may depend on what is available in our particular solver. Constraint patterns also need to apply to a range of problems. For example, double lex ordering can be applied even to problems like the steel mill slab design problem with *partial* row and column symmetry. In this case, we just order lexicographically those subsets of rows or columns which are indistinguishable.

4.3 Dual models

The next pattern is documented in a number of papers (e.g. [18, 7, 19]) as well as ILOG's Solver 5.3 User's manual (Volume II). A constraint program defines a set of decision variables, each with an associated domain of values, and a set of constraints defining allowed values for subsets of these variables. The efficiency of a constraint program depends on a good choice for the decision variables, and a careful modelling of the constraints on these variables. Unfortunately, there is often considerable choice even in what to make the variables, and what to make the values. For example, in an exam timetabling problem, the variables could be the exams, and the values could be the times. However, we could take an alternative or dual viewpoint in which the variables are the times, and the values are the exams.

The choice of variables and values is especially evident in permutation problems. In a permutation problem, we have as many values as variables, and each variable takes an unique value. We can therefore easily exchange variables for values. Indeed, it is often beneficial to have both sets of variables with channelling constraints between the primal (or original) model and the dual [18, 7, 19]. Many assignment, scheduling and routing problems are permutation problems. For example, sports tournament scheduling can be modelled as finding a permutation of the games to fit into the time slots, or a permutation of the time slots to fit into the games.

Pattern Name:	`DualModelling`
Context:	An informal problem specification.
Problem:	Choosing between a primal and an alternative dual viewpoint.
Forces:	Certain constraints can be easier to post on primal.
	Certain constraints can propagate better in primal.
	Certain constraints can be easier to post on dual.
	Certain constraints can propagate better in dual.
Solution:	Consider having a combined model with channelling between the primal and dual variables.
Example:	Balanced Academic Curriculum Problem.

Fig. 4. Constraint pattern for `DualModelling`

An alternative or dual viewpoint can be beneficial for a number of reasons. First, we can get different amounts of propagation in a primal, dual or a combined model [7, 19]. Second, certain constraints may be more easily stated (and propagated) on a dual model. If others are more easily stated (and propagated) on the primal, we can decide to use a combined model. Third, branching on the dual variables can be useful. For example, in a permutation problem, dual variables correspond to primal values. Therefore a variable ordering heuristic on dual variables is essentially a value ordering heuristic on the primal model.

Variable ordering heuristics like fail first tend to be cheap and effective. On the other hand, value ordering heuristics tend to be neither. Variable ordering on the dual therefore can be an effective means to get value ordering on the primal.

As a concrete example, consider the Balanced Academic Curriculum Problem (prob030 in CSPLib) [20]. The objective is to assign time slots to courses. A natural matrix model is a 2-d array of 0/1 decision variables indicating if a course is given in a particular time slot. Most of the constraints are easy to post using this array of variables (e.g. the constraint that a limited number of courses occur in any time slot is simply a row sum constraint). However, one type of constraint is not easy to post on this array of variables. This is the course prerequisite constraint: every course must occur after all its prerequisites. An easy way to post this constraint is to take an alternative dual viewpoint with a 1-d array of finite-domain variables. Each variable here takes as its value the time slot for a particular course. A prerequisite constraint is now simply ordering constraints between the course variable and each of the course variables associated with its prerequisites. The other constraints are not as easily specified on this 1-d array. It is therefore beneficial to have both arrays and to channel between them [20].

4.4 Auxiliary variables

The next pattern is documented in [21, 22]. A common method for improving a basic constraint model is to introduce auxiliary variables. Such variables permit propagation to occur between constraints with structure in common. Consider, for example, the problem of finding optimal Golomb rulers (prob006 in CSPLib). A Golomb ruler is a set of m integer valued ticks, such that the distance between any pair of ticks is different from the distance between any other pair. The objective is to find the optimal or shortest such ruler. Such rulers have practical applications in radio astronomy.

A natural model for the Golomb ruler problem has a finite-domain variable, X_i for each tick, and this is assigned the position of the tick on the ruler. To break symmetry between the ticks, we can post constraints of the form $X_i < X_j$ for $i < j$. To ensure that all inter-tick distances are distinct, we can post quaternary constraints of the form: $|X_i - X_j| \neq |X_k - X_l|$. Each such constraint computes two inter-tick differences, and each of these differences appears in a quadratic number of other constraints. A better model introduces auxiliary variables for these differences, D_{ij} and ternary constraints of the form: $D_{ij} = |X_i - X_j|$

Introducing auxiliary variables can be advantageous for several reasons. For example, we can get more propagation through the domains of these auxiliary variables. As a second example, we may be able to post simpler constraints on the auxiliary variables. In the case of the Golomb ruler problem, we can replace the large number of quaternary constraints by a single all-different constraint on the auxiliary variables. We can then use an efficient algorithm for enforcing GAC or BC on such a constraint [23, 24]. In [22], we show that modelling the Golomb ruler with such auxiliary variables increases the amount of constraint propagation and reduces runtimes significantly. More recently, we have developed methods for automatically introducing such auxiliary variables into a constraint model [25, 26].

Pattern Name:	`AuxiliaryVars`
Context:	A basic model in which two or more constraints repeat expressions.
Problem:	Insufficient propagation between these constraints.
Forces:	Overhead of introducing additional variables.
Solution:	Introduce auxiliary variables for the repeated expressions.
Example:	Golomb ruler problem.

Fig. 5. Constraint pattern for `AuxiliaryVars`

4.5 Implied constraints

The next pattern is also described in a number of papers (e.g. [27, 22]), as well as ILOG's Solver 5.3 User's manual (Volume II). A common method for improving a basic constraint model is to introduce implied constraints. These are constraints which are not logically necessary but which may reduce search. Consider again the problem of finding optimal Golomb rulers. In the last section, we argued for the introduction of auxiliary variables for the inter-tick differences, ternary constraints of the form $D_{ij} = |X_i - X_j|$ and an all-different constraint over D_{ij}. By transitivity, as $X_i < X_j$ for $i < j$, we can infer that $D_{ij} < D_{ik}$ for any $j < k$. This implied constraint is not logically necessary. We obtain the same set of solutions with or without it. However, as shown in [22], its inclusion in the model (along with other implied constraints) reduces search and saves runtime.

Not all implied constraints are useful. Constraint propagation on an implied constraint that is very immediate may do no more pruning than constraint propagation on a basic model. In addition, even if an implied constraint reduces search, it adds overhead to the constraint propagation. In [22], we outline two basic criteria for deciding which implied constraints to add. First, implied constraints either should have specialized, efficient and effective constraint propagation algorithms or should be of small arity. This limits the overheads of adding the implied constraint and helps ensure it will propagate. Second, circumstances in which an implied constraint leads to pruning should be obvious and frequent. The hope is that the implied constraint will reduce search sufficiently to justify the overhead.

One way to develop useful implied constraints is to study the search process. Suppose the constraint solver explores an "obviously" futile part of the search tree. The partial assignments considered by the solver cannot be extended to a complete solution, but they satisfy the constraints in the model. Our challenge then is to identify an implied constraint that would have pruned this branch immediately. We are currently developing methods for inferring useful implied constraints automatically [25, 26]. For example, one of our methods identifies a clique of not-equals constraints (e.g. the constraints $D_{ij} \neq D_{kl}$ in the Golomb ruler problem) and replaces them by an all-different constraint. Another method performs Gaussian-like elimination (e.g. if we introduce an auxiliary variable for a repeated expression, this method eliminates the repeated expression in favour of the auxiliary variable).

Pattern Name:	`ImpliedConstraints`
Context:	A basic constraint model.
Problem:	Search going down obviously futile branches.
Forces:	Overhead of introducing additional constraints.
	Applicability of the new implied constraints.
Solution:	Introduce implied constraints that prune such branches.
Example:	Golomb ruler problem.

Fig. 6. Constraint pattern for `ImpliedConstraints`

5 Related work

Unfortunately, constraint patterns are rarely described in a general way that permits their immediate use in other applications. One exception is a paper by Martin Green and David Cohen [28] that identifies a constraint pattern which is useful for modelling a range of assignment problems. The pattern occurs, for example, in the problem of assigning radio frequencies to pilots in a model aircraft tournament. In a straightforward model, in which the pilots are the variables and the values assigned to these variables are the radio frequencies, pilots assigned the same frequency are symmetric. We can therefore swap any two pilots assigned the same frequency and obtain a symmetric solution. To eliminate such symmetries, Green and Cohen propose an alternative viewpoint similar to the swapping of values for variables.

Hans Schlenker and Georg Ringwelski use a design pattern in POOC [29], a platform for object-oriented constraint programming. POOC provides Java wrappers around commercial and academic constraint solvers. Different constraint solvers can thus be easily compared. In addition, Java programmers can rapidly experiment with constraint solving. The wrappers are designed using the `object factory` design pattern. This defines an interface for creating an object, whilst allowing subclasses to decide which class to instantiate.

6 Conclusions and Future Work

I have argued that we need to identify, formalize and document patterns in constraint models in a similar way to the patterns identified by the software engineering community. A library of such patterns will help tackle the modelling "bottleneck" that hinders the uptake of constraint programming. I have described some of the more interesting constraint patterns which have already been identified. I also discussed the different ways that we can exploit such patterns. For example, one way to exploit common constraint patterns is to extend the constraint language.

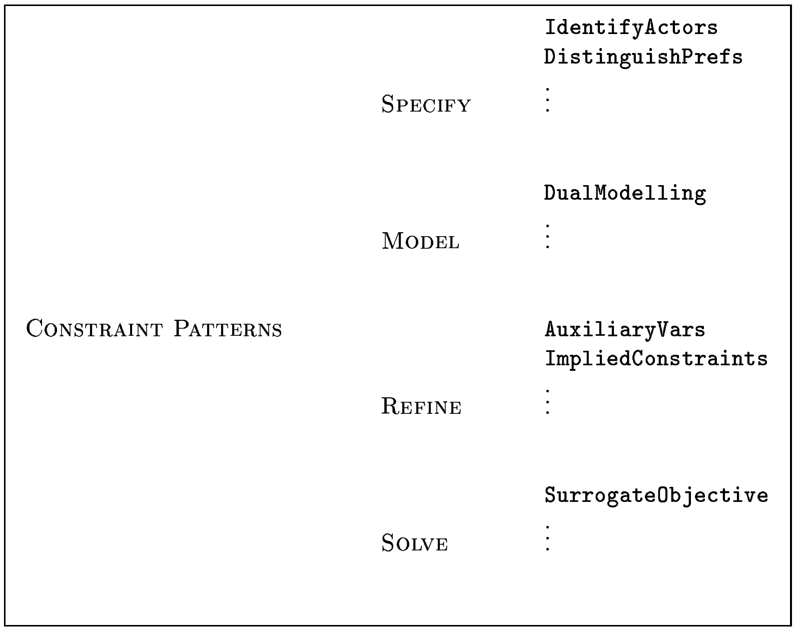

Fig. 7. A sketch of a possible constraint pattern taxonomy

There are a number of important directions to follow. First, more patterns need to be collected. As I argued before, this requires the efforts of the whole community, not just of one individual. Second, the patterns outlined here need more detail and generality. Third, the patterns need to be organized into a pattern taxonomy so that they can be accessed easily. See Figure 7 for a possible start to such a taxonomy. Fourth, we need to collect these patterns into a pattern library. I believe such a library would be a significant asset to the constraint programming community. A first attempt at such a library is taking shape at `4c.ucc.ie/patterns/`.

Acknowledgments

The author is supported by the Science Foundation Ireland. He wishes to thank the members of the 4C lab, the APES research group (especially Barbara Smith), the York AI Group (especially Alan Frisch), the Uppsala ASTRA group, and David Cohen.

References

1. Alexander, C.: A Pattern Language. Oxford University Press, New York (1977)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
3. Hentenryck, P.V., Michel, L., Perron, L., Regin., J.C.: Constraint programming in OPL. In Nadathur, G., ed.: Principles and Practice of Declarative Programming, Springer-Verlag (1999) 97–116 Lecture Notes in Computer Science 1702.
4. Fleiner, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. Technical Report APES-36-2001, APES group (2001) Available from http://www.dcs.st-and.ac.uk/ apes/reports/apes-36-2001.ps.gz. Presented at Formul'01 Workshop on Modelling and Problem Formulation, CP2001 post-conference workshop.
5. Fleiner, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling: Exploiting common patterns in constraint programming. In: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems. (2002) held alongside CP-2002.
6. Cheng, B., Choi, K., Lee, J., Wu, J.: Increasing constraint propagation by redundant modeling: an experience report. Constraints **4** (1999) 167–192
7. Smith, B.: Modelling a Permutation Problem. In: Proceedings of ECAI'2000 Workshop on Modelling and Solving Problems with Constraints. (2000) Also available as Research Report from http://www.comp.leeds.ac.uk/bms/papers.html.
8. Walsh, T.: Permutation problems and channelling constraints. Technical Report APES-26-2001, APES group (2001) Available from http://www.dcs.st-and.ac.uk/ apes/reports/apes-26-2001.ps.gz. Also in Proceedings of the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints, 2001.
9. Frisch, A., Miguel, I., Walsh, T.: Modelling a steel mill slab design problem. In: Proceedings of IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints, International Joint Conference on Artificial Intelligence (2001)
10. Fleiner, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetry in matrix models. In: 8th International Conference on Principles and Practices of Constraint Programming (CP-2002), Springer (2002)
11. Frisch, A., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In Rossi, F., ed.: Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP2003), Springer (2003)
12. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: 8th International Conference on Principles and Practices of Constraint Programming (CP-2002), Springer (2002)
13. Crawford, J.: A theoretical analysis of reasoning by symmetry in first-order logic (extended abstract). In: Proceedings of AAAI 1992 Workshop on Tractable Reasoning. (1992) 17–22
14. Gent, I., Smith, B.: Symmetry breaking in constraint programming. In Horn, W., ed.: Proceedings of the 14th European Conference on Artificial Intelligence. (2000) 599–603

15. Fahle, T., Schamberger, S., Sellman, M.: Symmetry breaking. In Walsh, T., ed.: Proceedings of 7th International Conference on Principles and Practice of Constraint Programming (CP2001), Springer (2001) 93–107
16. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Multiset ordering constraints. In: Proceedings of 18th IJCAI, International Joint Conference on Artificial Intelligence (2003)
17. Puget, J.F.: Symmetry breaking for matrix models using stabilizers. In Rossi, F., ed.: Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP2003), Springer (2003)
18. Cheng, B., Choi, K., Lee, J., Wu, J.: Increasing constraint propagation by redundant modeling: an experience report. Constraints **4** (1999) 167–192
19. Walsh, T.: Permtuation problems and channelling constraints. In: Proceedings of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001). (2001)
20. Hnich, B., Kiziltan, Z., Walsh, T.: Modelling a balanced academic curriculum problem. In: Proceedings of 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2002). (2002)
21. Smith, B., Stergiou, K., Walsh, T.: Modelling the golomb ruler problem. In: Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints, International Joint Conference on Artificial Intelligence (1999) Also available as APES report, APES-11-1999 from http://apes.cs.strath.ac.uk/reports/apes-11-1999.ps.gz.
22. Smith, B., Stergiou, K., Walsh, T.: Using auxiliary variables and implied constraints to model non-binary problems. In: Proceedings of the 16th National Conference on AI, American Association for Artificial Intelligence (2000) 182–187
23. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the 12th National Conference on AI, American Association for Artificial Intelligence (1994) 362–367
24. Lopez-Ortiz, A., Quimper, C., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: Proceedings of the 18th International Conference on AI, International Joint Conference on Artificial Intelligence (2003)
25. Frisch, A., Miguel, I., Walsh, T.: Extensions to proof planning for generating implied constraints. In: Proceedings of 9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus-01). (2001) 130–141
26. Frisch, A., Miguel, I., Walsh, T.: CGRASS: A system for transforming constraint satisfaction problems. In O'Sullivan, B., ed.: Recent Advances in Constraints. Volume 2627 of Lecture Notes in Artificial Intelligence., Springer (2003) Volume contains selected papers from the ERCIM/CologNet 2002 workshop. The paper is also available as APES-42-2002 Research Report.
27. Proll, L., Smith, B.: Integer linear programming and constraint programming approaches to a template design problem. INFORMS Journal on Computing **10** (1998) 265–275
28. Green, M., Cohen, D.: A CSP design pattern for hard problems. Technical report, Royal Holloway, University of London (2003)
29. Schlenker, H., Ringwelski, G.: Pooc - a platform for object-oriented constraint programming. In O'Sullivan, B., ed.: Recent Advances in Constraints. Volume 2627 of Lecture Notes in Artificial Intelligence., Springer (2003)