Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh

Chapter 11

Modelling

Barbara M. Smith

Constraint programming can be a successful technology for solving practical problems; however, there is abundant evidence that how the problem to be solved is modelled as a Constraint Satisfaction Problem (CSP) can have a dramatic effect on how easy it is to find a solution, or indeed whether it can realistically be solved at all. The importance of modelling in constraint programming has long been recognized e.g. in invited talks by Freuder [14] and Puget [34].

In this chapter, it will be assumed that the problem to be solved can be represented as a CSP whose domains are finite; infinite domains are discussed in Chapter 16, "Continuous and Interval Constraints". In most of the examples, the variable domains will be sets of integers; see Chapter 17, "Constraints over Structured Domains", for more on set variables and other variable types.

A complicating factor in modelling is the interaction between the model, the search algorithm and the search heuristics. To simplify matters, it will be assumed that, having modelled the problem of interest as a CSP, the CSP will be solved using a constraint solver such as ILOG Solver, ECL^{i}PS^e, Choco, SICStus Prolog, or the like. The default complete search algorithms provided by these solvers are sufficiently similar that they provide a common context for discussing modelling. Furthermore, they are designed to solve large problems of practical significance, and for such problems, it is worth the effort of developing the best model of the problem that we can find. Some of what follows will also apply to other search techniques such as local search (covered in Chapter 8) or to other complete search algorithms, but much will not, because the search algorithm has a profound influence on modelling decisions.

In this chapter, it will be assumed that the problem to be solved is well-defined; although eliciting a correct and full problem description can be a significant proportion of the problem-solving effort, it will be assumed here that that step has been done. It will also be assumed that the problem does not involve preferences or uncertainty, which are covered in Chapters 9 and 21.

11.1 Preliminaries

In this section, the concepts needed in the rest of the chapter are defined.

A *Constraint Satisfaction Problem* (CSP) is a triple $\langle X, D, C\rangle$ where: X is a set of *variables*, $\{x_1, ..., x_n\}$; D is a set of *domains*, $D_1, ..., D_n$ associated with $x_1, ..., x_n$ respectively; and C is a set of *constraints*. Each constraint $c \in C$ is a pair $c = \langle\sigma, \rho\rangle$ where σ, the constraint *scope*, is a list of variables, and ρ, the constraint *relation*, is a subset of the Cartesian product of their domains.

The domain of a variable is the set of possible values that can be assigned to it. In this chapter, it will be assumed that the domain of a variable is a finite set.

An *assignment* is a pair (x_i, a), which means that variable $x_i \in X$ is assigned the value $a \in D_i$. A *compound assignment* is a set of assignments to distinct variables in X. A *complete assignment* is a compound assignment to all variables in X.

The relation of a constraint $c = \langle\sigma_c, \rho_c\rangle$ specifies the acceptable assignments to the variables in its scope. That is, if the constraint scope σ_c is $\{x_{i_1}, x_{i_2}, ..., x_{i_k}\}$ and $\langle a_1, a_2, ..., a_k\rangle \in \rho_c$, the compound assignment assigning a_i to x_{i_k}, $1 \le i \le k$, is an acceptable assignment; we say that the assignment *satisfies* the constraint c. A solution to the CSP instance $\langle X, D, C\rangle$ is a complete assignment such that for every constraint $c \in C$, the restriction of the assignment to the scope σ_c satisfies the constraint.

The relation of a constraint may be specified *extensionally* by listing its acceptable (satisfying) tuples, or *intensionally* by giving an expression involving the variables in the constraint scope such as $x < y$ from which it can be determined whether or nor a given tuple satisfies the constraint.

The *arity* of a constraint is the size of its scope. A unary constraint is defined on a single variable, a binary constraint on two variables. There is no requirement that different constraints must have different scopes.

Given a constraint $c = \langle\sigma_c, \rho_c\rangle$, the *projection* of c onto $\tau \subset \sigma_c$ is a constraint c' whose scope is τ and whose relation is the set of tuples derived by taking each tuple in ρ_c and selecting only those components corresponding to the variables in τ.

Many forms of consistency have been defined for CSPs and individual constraints. Here, only those commonly used by constraint solvers are defined. Consistency and constraint propagation are covered fully in Chapter 3.

A binary constraint is *arc consistent* if for every value in the domain of either variable, there exists a value in the domain of the other such that the pair of values satisfies the constraint. A non-binary constraint is *generalized arc consistent* or *hyper-arc consistent* iff for any value for a variable in its scope, there exists a value for every other variable in the scope such that the tuple satisfies the constraint. *Domain propagation* on a constraint removes unsupported values (i.e. values which cannot be extended to a pair of tuple of values satisfying the constraints) from the domains of the variables in its scope until the constraint is (generalized) arc consistent.

A constraint c on variables with ordered domains (such as integers) is *bounds consistent* if for every variable x in its scope, there exists a value d_j for every other variable x_j $(1 \le j \le k)$ in the scope of c, with $\min_{D_j} \le d_j \le \max_{D_j}$, such that the compound assignment $\{(x, l), (x_1, d_1), ..., (x_k, d_k)\}$ satisfies c, where l is the minimum of the domain of x, *and* similarly, values d'_j can be found with $\min_{D_j} \le d'_j \le \max_{D_j}$, such that $\{(x, u), (x_1, d'_1), ..., (x_k, d'_k)\}$ satisfies c, where u is the maximum of the domain of x. (For arithmetic constraints, the values d_j, d'_j can be real values rather than integers.)

Bounds propagation on an arithmetic constraint reduces the bounds of the variables until the constraint is bounds consistent.

11.2 Representing a Problem

It is difficult to define precisely what we mean when we say that a CSP *represents* a problem P. A possible definition is that: a CSP $M = \langle X, D, C\rangle$ represents a problem P, or M is a *model* of P, if every solution of C corresponds to a solution of P and every solution of P can be derived from at least one solution to C.

The definition does not require that there is a one-to-one correspondence between the solutions of P and the solutions of M. This is because modelling a problem as a CSP often introduces symmetry, by representing entities that are indistinguishable in P by distinct variables or values in M. Hence, multiple solutions of M may correspond to the same solution to P.

Symmetry causes a further complication, because if there is symmetry in both P and M, one way to deal with it is to add constraints to M; the aim is to eliminate all but one solution in every symmetry equivalence class. The symmetry-breaking constraints exist only in M, not in P, so that multiple symmetrically-equivalent solutions to P can correspond to the same solution to M. Hence, the correspondence between the solutions to M and the solutions to P can be many-to-many. We might avoid this last complication by agreeing that symmetry-breaking constraints are a special case, intended to eliminate solutions to M and therefore also solutions to P, and that they can be ignored in considering whether M is a model of P.

A final difficulty with the definition is that it implies that any CSP models a problem that has no solutions. The definition of equivalence of CSPs given by Rossi, Petrie and Dhar [36] similarly makes any CSPs with no solutions equivalent.

In practice, in modelling a problem as a CSP, we do not rely on this definition, but choose variables and values to represent entities in P and write the constraints on these variables to represent the rules and restrictions defining the solutions to P. However, it must certainly be true that any solution to M yields exactly one solution to P, and that any solution to P corresponds to a solution to M or is symmetrically equivalent to such a solution, and that if M has no solutions, this is because P itself has no solutions.

The aim in choosing a model of a problem is to arrive at a CSP that can be solved quickly; we typically require good run-time behaviour over the range of instances to be solved. Note that the shortest run-time does not necessarily mean the least search (as measured by nodes visited or backtracks, say).

11.3 Propagation and Search

In this chapter, it will be assumed, unless stated otherwise, that the CSP will be solved by a complete search algorithm that interleaves search with constraint propagation. Such search algorithms are dealt with in Chapter 4, “Backtracking Search Algorithms for CSPs”, along with variable and value ordering heuristics. The search proceeds by constructing a series of choice points, at each of which a set of mutually exclusive and exhaustive choices is constructed, involving variables whose value is not yet assigned. Common sets of choices are $\{x_i = a, x_i \neq a\}$ (binary branching); $\{x_i = 1, x_i = 0\}$ (when the variables are Boolean); $\{x_i \leq a, x_i > a\}$ (domain splitting); $\{x_i = v_1, x_i = v_2, ..., x_i = v_k\}$, where $v_1, v_2, ..., v_k$ are the values currently available in D_i (k-way branching). Choices can involve more than one variable, e.g. $\{x_i \leq x_j, x_i > x_j\}$; this is common in scheduling, for instance, where the choices might represent the two possible orders for the starting times of two activities (see Chapter 22). Although all these types of choice, and more, are possible, in the examples quoted in this chapter binary branching has been used.

The search pursues each choice in turn, first adding the constraint defining the choice to the existing constraints and propagating it, until local consistency is restored in the resulting subproblem. Typically, each type of constraint in the problem has an associated propagation algorithm, achieving the level of consistency specified for that constraint. Constraint propagation continues until no further propagation can be done, and every constraint is again in its target state of consistency. Given a target level of consistency for each constraint in C, the CSP $\langle X, D, C \rangle$ is *locally consistent* if every constraint achieves its target consistency level. If, at any stage during the search, constraint propagation results in an empty domain for some future (not-yet-assigned) variable, the search backtracks, restoring the domains to their state before the last choice was made, and exploring another of the choices created at the last choice point; if no further choices remain, the search backtracks to a previous choice point, and so on, until either a solution is found or all possible choices have been explored.

This form of search is used by default in commercial constraint solvers. It has a profound influence on the modelling process, because in taking many modelling decisions, the user needs to consider their effect on constraint propagation.

Typically, constraint solvers will enforce arc consistency (AC) on some, but not all, binary constraints and bounds consistency (BC) on arithmetic constraints. They will not usually maintain generalized arc consistency (GAC) on non-binary constraints, except for global constraints for which an efficient propagation algorithm exists. For some global constraints, the user may be able to choose the level of consistency to be maintained. For some complex constraints, the default may be to do very little consistency checking; the propagation algorithm may take action only when all but one or two of the variables in its scope have been instantiated.

These decisions in designing constraint solvers stem from a trade-off between the time and space required to maintain generalized arc consistency on all constraints and the reduction in search that could result. Puget has explained the decision to maintain only bounds consistency on arithmetic constraints in ILOG Solver by saying: "Solver is a compromise between efficiency and completeness...In the example... [of constraint propagation of arithmetic constraints] the incompleteness comes from the fact that arithmetic expressions only propagate bounds.. This is an example of the choice we made. Propagating holes in expressions would require much more memory and time than the current implementation. ¿From tests made on a very large set of examples, we found that the current compromise is by far better."

Even if we start from the assumption that the CSP will be solved using this general search algorithm, the form of the choices made at choice points, as well as the specific variable and value choices, will also affect the solution time.

Beacham, Chen, Sillito and van Beek [2] investigate the interaction between constraint models, search algorithms and search heuristics, using crossword puzzle problems. They compare three constraint models and two well-known search heuristics (minimum domain and domain/degree); the search algorithms are forward checking and a search algorithm that maintains generalized arc consistency, with three different ways of enforcing GAC. They conclude that the three choices of model, algorithm and heuristic interact, and that for the most efficient problem solving, none of the decisions can be made independently of the others.

It is a moot point whether the choice of search heuristics is part of modelling or not. It is certainly true that the performance of a model will be affected by the search heuristics, but for the purposes of this chapter, choosing the search heuristics will be excluded. However, for some types of model, there is a choice of which of the variables in the model should be used to drive the search, i.e. which variables should participate in choice points, and this will be considered as part of modelling.

11.4 Viewpoints

Different models of a problem P may result from viewing the problem P from different angles or perspectives. The term *viewpoint* was introduced informally by Geelen [19], in discussing permutation problems, and was subsequently adopted and formally defined by Law and Lee [29]. A viewpoint is a pair $\langle X, D\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, and D is a set of domains; for each $x_i \in X$, the associated domain D_i is the set of possible values for x. It must be possible to ascribe a meaning to the variables and values of the CSP in terms of the problem P, and so to say what an assignment in the viewpoint $\langle X, D\rangle$ is intended to represent in terms of P. The complete assignments defined by the viewpoint are intended to include all possible solutions of P. The constraints must then ensure that every solution to the CSP is a valid solution to P, and so are largely determined by that requirement. Hence, it is different viewpoints that give rise to fundamentally different models of a problem.

In principle, the values in the domain can be of any type. In practice, the types commonly supported by constraint solvers include integers, Booleans (perhaps only as a subtype of integers) and sets of integers. Other types have been proposed, e.g. multisets and tuples; constraint solvers may directly support these, or provide facilities to allow new types to be defined. Some of what follows may also apply to modelling using real-valued variables, and since the domains of integer variables are sometimes represented as intervals, the boundary can be blurred.

Except for some very small problems, the variables of a CSP are usually implemented using some data structure such as a list or an array. Flener, Frisch, Hnich, Kiziltan and Walsh [12] suggest that *matrix models*, based on matrices of variables, are a natural way to model many problems; indeed, almost all the examples given in this chapter use 1- or 2-dimensional matrices of variables. For some applications, other structures are important; for instance, models based on graphs are used in the network applications discussed in Chapter 25.

There are usually different viewpoints that could be chosen in modelling a problem. Although viewpoints can be combined, as will be described in section 11.9, it will be assumed for now that only one will be used. Having chosen a viewpoint, the next step is to express the constraints to ensure that the solutions to the CSP are correct, i.e. are solutions to P. However, although correctness is a minimum requirement, it is not sufficient if we are also concerned with how efficiently the CSP can be solved. A good rule of thumb in choosing a viewpoint is that it should allow the constraints to be easily and concisely expressed; we should prefer viewpoints that allow the problem to be described using as few constraints as possible, as long as those constraints have efficient, low-complexity propagation algorithms.

Nadel [30] was possibly the first to discuss different ways of modelling a problem. He lists nine different representations of the n-queens problem as a CSP (in fact, nine different viewpoints), although two of these are derived from another two simply by swapping the roles of rows and columns, and so result in identical CSPs. For instance, two of the viewpoints are:

1. the variables $r_1, .., r_n$ represent the rows of the board, and the domain of each variable is the set of integers $\{1, 2, ..., n\}$ representing the columns; an assignment (r_i, c) means that the queen in row i is in column c;

2. the variables $q_1, ..., q_n$ correspond to the n queens and the domain of each variable is the set of integers $\{1, 2, ..., n^2\}$, representing the squares; an assignment (q_i, a) means that the i^{th} queen is on square a.

In the first viewpoint, the rule that there is only one (in fact, exactly one) queen on each row is covered by the fact that any variable can only be assigned one value. The rule that there is only one queen in each column can be expressed by the constraints $r_i \neq r_j$ for $1 \leq i < j \leq n$ or by an allDifferent constraint on $r_1, ..., r_n$.

In the second viewpoint, the rules are more awkward to express. Constraints are needed to ensure that no two queens are in the same row; if the 'row' element of a value can be extracted, there could be a constraint between every pair of variables that their row elements are not equal; the column constraints could be dealt with similarly. The diagonal constraints are more difficult to write. One possibility is to state an extensional constraint between each pair of variables, listing for each of the n^2 values, the values representing squares that are not in the same row, column or diagonal, although domain propagation would then be expensive. Furthermore, such constraints would only express that fact that there is *at most* one queen in each row or column, not that there must be *exactly* one. Although only correct solutions would be found using these constraints, the model would allow partial solutions in which the queens already placed attack all the squares on a row or column, since there is nothing explicit in the constraints to forbid this. Hence, a model based on the second viewpoint would be less efficiently solved than the model based on the first viewpoint.

11.5 Expressing the Constraints

Once we have arrived at a viewpoint that allows the constraints to be easily and concisely expressed, there are often choices in exactly how to write the constraints; an example has already been seen in the first viewpoint for the n-queens problem, where there is a choice between binary $\neq$ constraints and an allDifferent constraint.

The way in which the constraints are written affects the efficiency of the resulting model, because it affects how the constraints will propagate during the search. Harvey and Stuckey [22] observe that, "An unnerving and not well studied property of propagation based solvers, is that the form of a constraint may change the amount of information that propagation discovers." They illustrate this with the constraints $c_1 \equiv (x = y)$, $c_2 \equiv (x + y = z)$ and $c_3 \equiv (2y = z)$, where x, y and z are integer variables. If $C = \{c_1, c_2\}$

and $C' = \{c_1, c_3\}$, C and C' are equivalent, in the sense that they have the same solutions. However, if C and C' are made locally consistent, then in C', the domain of z (using AC) or its upper and lower bounds (using BC) will be even integers, but this is not necessarily true of C.

Unfortunately, to arrive at a good model of P, it is essential to be aware of the range of constraints supported by the constraint solver and the level of consistency enforced on each and to have some idea of the complexity of the corresponding propagation algorithms. This is, of course, a long way from the declarative ideal. In this section, some of the choices available when writing constraints are discussed.

11.5.1 Combining Constraints

Combining constraints with the same scope can be a way of expressing them more concisely. The conjunction of two constraints with the same scope allows only the tuples that are allowed by both. Enforcing the same level of local consistency on a conjunction $c_1 \wedge c_2$ as on c_1 and c_2 separately will remove at least as many domain values. However, it may or may not reduce the run-time, depending on how time-consuming it is to enforce local consistency on the conjunction and on the separate constraints.

An example can be found in the n-queens problem. Using the first viewpoint listed earlier (the standard CSP model for this problem), the variables $x_1, x_2, ..., x_n$ representing rows 1 to n of the board, and the values are $\{1, 2, ..., n\}$, representing the columns. The rule that two queens cannot be on the same column or diagonal can most simply be written using more than one constraint between each pair of variables x_i and x_j, $i < j$. For instance:

- $x_i \neq x_j$
- $x_i - x_j \neq j - i$
- $x_j - x_i \neq j - i$

Figure 11.1 shows a state that might be arrived at during search, when $n = 6$. Two variables, x_1 and x_2, have already been assigned, and the crossed squares are no longer available, because queens placed there would conflict with the two already placed; the corresponding values will have been removed from the domains of the remaining variables x_3, x_4, x_5, x_6. A queen cannot now be placed in row 5, column 3, because it would conflict

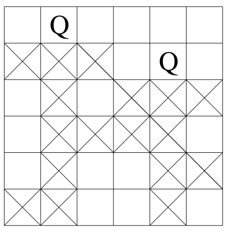
Figure 11.1: A search state in the 6-queens problem

with both remaining places for a queen in the 3rd row. However, the three constraints between x_3 and x_5 are arc consistent; the value 3 for x_5 is supported by the value 1 for x_3 as far as the first constraint is concerned, and by the value 3 for x_3 as far as the second constraint is concerned. If the conjunction of the three constraints were expressed as a single constraint, domain propagation would delete 3 from the domain of x_5. (However, since the conjunction is unlikely to be expressible as a single constraint using the standard constraints provided by constraint solvers, it might require writing a special constraint or forcing AC in some other way. Simply writing a single constraint as a conjunction of the separate constraints will not guarantee that the solver will enforce GAC on it, and it may in fact do *less* consistency checking than on the separate constraints.)

Katsirelos and Bacchus [28] discuss improving constraint propagation by enforcing GAC on conjunctions of constraints, rather than the individual constraints. If c_1 and c_2 are two constraints in a CSP, domain propagation on their conjunction $c_1 \wedge c_2$ removes at least as many domain values as domain propagation on c_1 and c_2 separately. If the scopes of c_1 and c_2 are disjoint, then domain propagation on the conjunction is equivalent to domain propagation on the separate constraints, but the larger the overlap in the scopes, the larger the potential domain pruning from conjoining the constraints. Katsirelos and Bacchus use Bessière and Régin's GAC-schema algorithm [4] in their experiments: for that algorithm, if the scope of c_1 is a subset of the scope of c_2, it is less time-consuming to enforce GAC on the conjunction than on the individual constraints. They propose, as a heuristic, to combine constraints which share all or most of their variables. They use the Golomb ruler problem, discussed in more detail in section 11.9, as an example. They model the problem as a CSP by using the positions of the m 'ticks' on the ruler as the variables $x_1, ..., x_m$. The constraints are that $|x_j - x_i| \neq |x_l - x_k|$, for $1 \leq i, j, k, l \leq m$. In this model, there are seven constraints of this kind over any set of four variables (four quaternary and three ternary). They show that combining the quaternary and ternary constraints on each set of four such variables reduces the number of backtracks slightly and the run-time a lot, compared to using the individual constraints; they maintain GAC on constraints in either case. (Note that this is not the model usually used for the Golomb ruler problem, so that their results are not comparable with others.)

11.5.2 Eliminating Variables

Harvey and Stuckey [22] give a number of theorems on rewriting linear constraints and how bounds propagation or domain propagation will be affected. For instance, one theorem concerns using a two-variable linear equation to substitute for one of these variables in a linear constraint: suppose $c_1 \equiv (\sum_{i=1}^{n} a_i x_i \text{ op } d)$, where op $\in \{=, \leq, \neq\}$ and $c_2 \equiv (b_j x_j + b_k x_k = e), j \neq k, b_j \neq 0, b_k \neq 0$. Let c_3 be the constraint resulting from using c_2 to remove x_j in c_1. Then bounds propagation on $\{c_3, c_2\}$ is stronger than bounds propagation on $\{c_1, c_2\}$. (i.e. each variable domain in the first case is a subset of its domain in the second case). The same is true for domain propagation.

11.5.3 Global Constraints

Constraint solvers provide a range of *global constraints*, developed to replace particular sets of constraints that occur frequently. Global constraints are the subject of Chapter 7. They allow a single constraint on any number of variables to replace a set of constraints, and provide a propagation algorithm that typically enforces GAC on the constraint.

There is sometimes a choice as to what level of consistency will be maintained on the global constraint. A frequently occurring global constraint is the allDifferent constraint, and it does provide such a choice. A constraint allDifferent($x_1, x_2, ..., x_n$) can either be treated as if it had been written as $n(n-1)/2$ binary $\neq$ constraints on which AC is maintained; or bounds consistency (BC) can be maintained on the global constraint; or generalized arc consistency (GAC) can be maintained. Maintaining a higher level of consistency takes more time; on the other hand, if more values can be removed from the domains of the variables, the search effort will be reduced and this will save time. Whether or not the time saved outweighs the time spent depends on the problem. In the case of the allDifferent constraint, experience suggests that if the number of values in the union of the domains of $x_1, x_2, ..., x_n$ is n or not much greater, maintaining GAC is likely to be worthwhile; but if the number of values is much greater than n, so that the allDifferent constraint is looser, it is less likely that domain propagation will remove more values than the $\neq$ constraints, and so it may not be cost-effective (see for instance [31]).

11.5.4 Extensional Constraints

Some constraint solvers give the user the option to enforce GAC on any constraint. CHIP, for instance, had the facility to enforce arc consistency on arbitrary constraints defined by Prolog predicates, and this was used in solving a microcode labelling problem, described in [47]. ILOG Solver provides a table constraint, in which the set of allowed (or not allowed) tuples can be explicitly listed. SICStus Prolog similarly has a `case` constraint that allows the solutions to the constraint to be specified as a directed acyclic graph.

Cheng and Yap [7] demonstrate the usefulness of the SICStus Prolog `case` constraint in Maximum Density Still Life, a problem derived from the Game of Life. The game is played on a squared board and in the problem considered, each cell of the board is either alive or dead according to the state of its eight neighbouring cells. The original model has a Boolean variable for each cell, with the value 1 representing 'alive' and 0 representing 'dead'. The constraint between a variable and the variables representing the neighbouring cells is complex: the value of the cell is 1 if the sum of the neighbouring variables is exactly 3, or 0 if their sum is < 2 or > 4. The aim is to find a configuration of live and dead cells on an $n \times n$ board that satisfies the constraints and maximizes the number of live cells. Cheng and Yap use the `case` constraint to represent the constraint between the cells in a $3 \times n$ 'super-row'. They use the fact that the variables in the problem are Boolean to construct a Binary Decision Diagram of the constraint and convert the BDD to a DAG. For a good ordering of the variables, the size of the resulting BDD increases only linearly with n, so that maintaining consistency of the `case` constraint remains efficient.

It can be useful to be able to express even binary constraints extensionally and ensure that arc consistency is maintained. For instance, in the Black Hole patience game [20], a pack of playing cards has to be arranged in sequence, in such a way that successive cards in the sequence have consecutive values, so that for instance a five can only be followed by a four or a six (of any suit). An ace can be either a high or a low value, and so can be followed by a two or a king. (There are other conditions on the sequence that are not relevant here.) The viewpoint used in solving Black Hole games using CP in [20] has a variable x_i for each position i in the sequence, $1 \leq i \leq 52$; the domain of each variable is $\{1, .., 52\}$, representing the cards, where the values 1 to 13 represent the ace to king of spades respectively, 14 to 26 represent the ace to king of hearts, and so on. To ensure a correct sequence, there must be a binary constraint between x_i and x_{i+1} for $1 \leq i \leq 51$; for instance, if x_i is assigned the value 15 (representing the two of hearts), the possible values for x_{i+1} are 1, 3, 14, 16, 27, 29, 40, 42, representing the aces and threes. The constraint is expressed extensionally by listing the possible values for x_{i+1} for each possible value of x_i, using the table constraint in ILOG Solver, which maintains AC on the constraint.

11.5.5 Reified Constraints and Meta-Constraints

A reified constraint associates a 0/1 variable x with a constraint c, so that x takes the value 1 if the constraint c is satisfied and 0 otherwise. More or less equivalently, in terms of expressivity, a meta-constraint is a constraint over constraints. Fernandez and Hill [11] discuss representing a *self-referential puzzle* introduced by Henz [23] in a variety of constraint programming languages, using reified constraints and meta-constraints.

More significantly, they can be used to express disjunctions of constraints. For instance, the condition that constraint c_1 or constraint c_2 (or both) must be satisfied can be expressed by associating the constraints with the variables x_1 and x_2 respectively and adding the constraint that $x_1 + x_2 \geq 1$.

Van Hentenryck and Deville [48] introduced the cardinality operator to express such disjunctive conditions; it allows upper and lower bounds to be stated on the number of constraints in a set that must be satisfied. Of course, it is not sufficient simply to allow disjunctive conditions to be expressed; changes to the domains of the variables involved must also be propagated efficiently. The implementation of reified constraints in constraint logic programming is discussed in Chapter 12.

11.6 Auxiliary Variables

In the last section, different ways of writing constraints on the variables in the chosen viewpoint were discussed. However, more choices are available, and the potential for more efficient models, if other variables can be introduced.

Auxiliary variables are variables introduced into a model, either because it is difficult to express the constraints at all in terms of the existing variables, or to allow the constraints to be expressed in a form that would propagate better, i.e. lead to more domain reductions.

An early example appears in a paper on the car sequencing problem (problem 1 in CSPLib) by Dincbas, Simonis and van Hentenryck [9]. A number of cars are to be made on a production line: each of them may require one or more options which are installed at different stations on the line. The option stations have lower capacity than the rest of the production line, e.g. a station may be able to cope with at most one car out of every two. The cars are to be arranged in a production sequence so that these capacities are not exceeded.

In [9], the initial viewpoint has variables s_i, $1 \leq i \leq n$, where n is the number of cars to be produced, and therefore the length of the production sequence. The value of s_i represents the car to be produced in position i in the sequence, or more precisely the *class* of car, since cars requiring the same set of options can be considered as identical.

It is straightforward to express some of the constraints required to model the problem in this viewpoint, for instance, that the number of variables assigned a specific value is equal to the number of cars in the corresponding class. However, the option capacities are difficult to express using these variables alone.

Dincbas *et al.* introduce auxiliary Boolean variables o_{ij}, $1 \leq i \leq n, 1 \leq j \leq m$, such that $o_{ij} = 1$ iff the car in the ith slot in the sequence requires option j. The constraints expressing the option capacities are expressed in terms of these variables; suppose that the capacity of option 1 is one car in every two. Then the capacity of the option can be enforced using the constraints:

$$o_{i,1} + o_{i+1,1} \leq 1 \text{ for } 1 \leq i < n$$

Constraints are also needed to express the relationship between the auxiliary variables and the original variables. In this case, this could be done by the constraints $o_{ij} = \lambda_{s_i,j}, 1 \leq i \leq n, 1 \leq j \leq m$, where the constant $\lambda_{kj} = 1$ iff car class k requires option j.

Usually, auxiliary variables are not sufficient to define a viewpoint, i.e. it would not be possible to build a model of the problem using only the auxiliary variables. However, the auxiliary variables in the car sequencing problem could constitute a viewpoint; every valid production sequence can be specified as a complete assignment to these variables.

It is sometimes worthwhile to use auxiliary variables as search variables, alongside the original variables. An example occurs in a network design problem arising from the deployment of synchronous optical networks (SONET) [43]. The network contains a number of client nodes and a number of SONET rings. A SONET ring joins a number of nodes; a node is installed on a ring using an add-drop multiplexer (ADM). There are known demands (in terms of numbers of channels) between pairs of nodes; in a simplified version of the problem, the level of demand is ignored, but if there is a traffic demand between two nodes, there must be a ring that they are both installed on. Each node can be installed on more than one ring, and there is a maximum number of nodes that can be installed on each ring. The objective is to minimise the total number of ADMs required, while satisfying all the demands.

The viewpoint used in [43] has variables x_{ik}, $1 \leq i \leq n, 1 \leq k \leq m$, where n is the number of nodes and m is the number of available rings. $x_{ik} = 1$ if node i is assigned to ring k, 0 otherwise.

A number of auxiliary variables are introduced, representing for instance the number of rings that each node is on. It was found to be a successful search strategy to assign this last set of variables first, before assigning the variables x_{ik}. In terms of the underlying problem, although deciding *how many* rings each node is not sufficient to specify the network, it greatly simplifies the remaining problem of deciding *which* rings each node is on.

Note that if the auxiliary variables would constitute a viewpoint in their own right, and we assign values to these variables as well as the viewpoint variables, the resulting model might be more appropriately considered as combining two viewpoints, as in section 11.9.

11.7 Implied Constraints

Implied constraints, also called redundant constraints, are constraints which are implied by the constraints defining the problem. They do not change the set of solutions, and hence are logically redundant. The aim in adding implied constraints to the CSP is to reduce the search effort to solve the problem.

A necessary condition for an implied constraint to be useful in reducing search is that it forbids one or more compound assignments that the existing constraints will allow (given the level of propagation that will be maintained on the individual constraints during search).

A compound assignment forbidden by an implied constraint cannot lead to a solution, since it does not change the set of solutions. Without the implied constraint, such an assignment may occur during the search, and determining that it cannot be completed may take a very long time.

Dincbas, Simonis and van Hentenryck [9] used implied constraints in solving the car sequencing problem described earlier. In section 11.6, the constraints on the variables o_{ij} enforcing the option capacities are given. These constraints only express that fact that the option capacities cannot be exceeded; there is nothing to prevent a partial sequence of cars from using a particular option *below* capacity. However, a certain number of cars requiring each option have to be fitted into the sequence, so that going below capacity in one part of the sequence may make it impossible to avoid exceeding the capacity elsewhere. Hence, there are implied constraints which have not yet been expressed.

For instance, suppose there are 30 cars, and 12 of them require option 1, with capacity 1 car in any 2. Then at least one of the cars in slots 1 to 8 of the production sequence must require option 1; otherwise 12 of cars 9 to 30 will require option 1, which violates the capacity constraint. Similarly, cars 1 to 10 must include at least two option 1 cars, ... , and cars 1 to 28 must include at least 11 of the option 1 cars. Dincbas *et al.* added implied constraints of this kind for each option and for all sub-sequences starting with slot 1. Without these constraints, partial sequences in which one or more option stations are under-utilized can be formed, and eventually the search will have to backtrack when it is found that the sequence cannot be completed without exceeding the option capacity. The implied constraints prevent wasted search of unsatisfiable subproblems.

11.7.1 Implied Constraints and Search Order

Ensuring that each implied constraint forbids an assignment that would be allowed otherwise is not sufficient to guarantee that the added constraints will reduce the search effort. It may be that the assignments forbidden by a proposed implied constraint would never occur during the search anyway, given the search order. Hence, in backtracking search, the order in which the variables are assigned can affect whether it will be beneficial to add an implied constraint or not.

For instance, Borrett & Tsang [5] discuss adding an implied constraint between variables q and r when binary constraints between p and q and between p and r already exist in the CSP. The constraint c_{qr} could be derived by composing the constraints c_{pq} and c_{pr} - effectively, making this triple of variables path consistent. Borrett & Tsang show that using a simple backtracking algorithm (i.e. one doing no constraint propagation), if the three variables p, q and r are assigned in that order, the implied constraint c_{qr} will have no effect on the number of nodes visited. On the other hand, if the CSP already contains the constraints c_{pr} and c_{qr}, then adding the constraint c_{pq} can reduce the number of nodes visited, given the same search order.

Similarly, in the car sequencing problem, the usefulness of the implied constraints used by Dincbas *et al.* depends on the search order [39]. In the example given earlier, at least one car in slots i to $i+7$ of the sequence must require option 1, for any value of i from 1 to 23; hence, as well as the constraint added by Dincbas *et al.*, there are many other equally valid constraints. Overall, there are potentially very many implied constraints imposing a lower limit on the number of cars requiring a particular option in *any* sub-sequence of length k. However, if the search builds up the sequence of cars consecutively from slot

1, only the implied constraints on the first k cars affect the search. The other possible implied constraints would always be consistent, but checking this whenever one of the variables involved is assigned a value would slow down the search. On the other hand, if the variables were assigned in a different order, a different set of implied constraints would be useful.

11.7.2 Implied Constraints v. Global Constraints

Following the work of Dincbas *et al.* on the car sequencing problem, Régin and Puget [35] later developed a global constraint specifically for sequence problems, using the car sequencing problem as a test case. They noted that "our filtering algorithm subsumes all the implied constraints" used by Dincbas *et al.* The global constraint makes the effort of devising and implementing implied constraints redundant, in this case. It may often be true that implied constraints are only useful because a suitable global constraint does not (yet) exist. On the other hand, many implied constraints are simple and cheap to propagate, whereas global constraints are often time-consuming to propagate. Moreover, it is only worth the effort of implementing a global constraint if it can be used for a significant class of problems; for a one-off problem, where good implied constraints can be found, the implied constraints are likely to be more cost-effective.

11.7.3 Implied Constraints from Subproblems

Van Beek and Wilken [46] use implied constraints in finding minimum length instruction schedules for the object code produced by a compiler. The implied constraints are lower bounds on the number of steps between a pair of instructions, found by considering subproblems; if a consistency check in the subproblem shows that the current lower bound on the distance between two instructions cannot be achieved, a constraint increasing the bound can be added. Van Beek and Wilken comment that generating powerful implied constraints in this way was the key to being able to solve very large real instruction scheduling problems. In the SONET problem, described in section 11.6, implied constraints were also derived (in that case by hand) from considering subproblems; the SONET constraints are lower bounds on the auxiliary variables that represent the number of times that each node is installed on a ring. These examples suggest that subproblems might also be a useful source of tighter variable bounds in other cases.

11.7.4 Finding Implied Constraints

Implied constraints can often be explained as projections of a conjunction of a few of the problem constraints onto a subset of the variables in the union of their scopes. These constraints can be seen as partially enforcing some higher level of consistency in the problem. Although the search algorithm only enforces consistency on single constraints, there are forms of consistency that take all the constraints on a subset of the variables and find inconsistent tuples. Enforcing consistency on subsets of the constraints is computationally expensive, even if only done before search; if it generated the equivalent of useful implied constraints, it would likely also generate a much larger number that would not be useful during the search. Furthermore, consistency enforcing generates sets of forbidden tuples; these would be presented to the constraint solver as extensional non-binary constraints, which are time-consuming to propagate. This does not at present appear a promising route to generating implied constraints automatically; it is not sufficiently selective, and implied constraints need to be expressed in form that can propagate efficiently, like other problem constraints.

Alternatively, adding implied constraints to a CSP is often inspired by a search taking an unacceptably long time to solve a problem, and discovering on examining the search tree in detail that assignments that are obviously incorrect are being considered; implied constraints are generalizations that state explicitly what is incorrect about these assignments and other potential failed assignments of the same kind. On this view, implied constraints are akin to nogoods (inconsistent compound assignments) that are uncovered during search. However, individual nogoods have little effect on the search, and if there are enough of them to be useful, checking them will hinder the constraint solver. An advantage is that they do take account of the search heuristics. Again, automatically generating implied constraints from nogoods identified during the search would require some means of expressing the constraints in a form that can propagate efficiently.

Some attempts have been made to generate implied constraints automatically, by looking for logical consequences of the existing constraints. Hnich, Richardson and Flener [26] classify implied constraints, and discuss automatically generating implied constraints of each type. Some of the types that they identify have been discussed separately here; for instance, one of the types is a global constraint (such as an allDifferent constraint) used to replace a set of constraints (a clique of $\neq$ constraints). Other types require introducing new variables. However, two of their types fit closely the implied constraints discussed in this section: variable elimination (using one constraint to eliminate a variable in its scope from other constraints involving that variable) and constraints over a new scope (using a set of constraints to derive a new constraint over a subset of the union of their scopes). Hnich *et al.* describe using PRESS (PRolog Equation Solving System) to try to derive implied constraints from linear and nonlinear arithmetic constraints; in their test cases, it can find some implied constraints of the variable elimination type, and also implied linear constraints derived from nonlinear constraints, but not the other types.

Frisch, Miguel and Walsh [18] also make some initial steps towards automating the generation of implied constraints by developing *methods* (analogous to methods in proof planning) that can be applied to the set of constraints in a CSP to derive new constraints. One is the `eliminate` method, which attempts to eliminate variables or terms from a non-linear constraint, to give a constraint of lower arity that may propagate better. For example:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \text{ with } \frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI} \text{ yield: } 3\frac{A}{BC} \leq 1$$

Neither of these approaches addresses the interaction of the search heuristics and the implied constraints, but if a class of implied constraints can be identified for a type of problem, such as the car sequencing problem, it would be possible to identify the constraints that are useful during search, and discard the rest. Simonis *et al.* [38] discuss using visualization tools in a constraint solver to assess the value of implied constraints, by examining the progress of the search in detail. This makes it possible to check that the implied constraints work well with the search heuristics or to find out which of the implied constraints are effective.

11.8 Reformulations of CSPs

In the last sections, different ways of improving a model were discussed; the changes to the model keep the same viewpoint but change or add to the constraints, or expand the viewpoint by adding auxiliary variables. The alternative way to change the model is to change the viewpoint. This may require literally looking at the problem from a different perspective and developing some insight into the problem. However, some transformations from one viewpoint to another are standard or are useful in specific problem classes.

There is an established and continuing body of work on transforming CSPs into satisfiability problems (e.g. [49]). This work will not be discussed here, because its aim is fundamentally different; rather than developing a model that can be solved more efficiently as a CSP, using a constraint solver, it aims to solve the underlying problem more efficiently as a SAT problem, using a SAT solver.

11.8.1 Non-Binary to Binary translations

Early search algorithms for CSPs only dealt with binary constraints; as a result, there are some standard transformations of a CSP with non-binary constraints into a binary CSP [1]. The *hidden variable* transformation adds a new variable h_i to the CSP for each non-binary constraint, c_i; the values of h_i correspond to tuples of variables in the scope of c_i. The original constraint c_i is replaced by binary constraints between h_i and the variables in the scope of c_i; each value of h_i implies a value for each variable in the scope of c_i, and the binary constraints enforce this correspondence. In the terminology of this chapter, the hidden variables would be classed as auxiliary variables, rather than a change of viewpoint.

The *dual graph* translation of a non-binary CSP replaces the original constraints by new variables, and so produces a new CSP based on a different viewpoint. The dual variable d_i represents the constraint c_i, and its values represent the tuples satisfying c_i. There is a binary constraint between two dual variables d_i and d_j if the scopes of c_i and c_j have a non-empty intersection; the binary constraint forbids pairs of values which would assign different values to any of the shared variables.

Bacchus and van Beek [1] investigate these transformations empirically, using a forward checking algorithm: when applied to the original non-binary model, the algorithm checks a k-ary constraint whenever all but one variable in its scope has been assigned. They show that both the hidden variable and dual graph transformation can outperform the original model; however, given constraint solvers that have better ways of dealing with many types of non-binary constraint, these transformations have been little used in practice.

An exception is the use of dual variables to replace 9-ary constraints in the Maximum Density Still Life problem, described earlier in section 11.5.4. In [42], the 9-ary constraints between a cell and its eight neighbours are replaced by dual variables, exactly as in the dual graph transformation. Unlike the dual graph transformation, the original variables are also kept, although only in order to express the objective, that the number of live cells should be maximized. The dual variables represent 3×3 'supercells'; one advantage of the dual graph translation, as well as replacing the cumbersome 9-ary constraints, is that it allows the supercells rather than the cells to be the search variables. Hence, the dual graph translation in this case corresponds to a genuinely different perspective on the problem.

A similar transformation has been used by Hnich, Prestwich and Selensky [25] in modelling the covering test problem (problem 45 in CSPLib), arising in software testing. The covering test problem is: for a given tuple (t, k, g, b) find a *covering array* $CA(t, k, g)$ of size b or show that none exists. The covering array has k columns and b rows, and in every subset of t columns every possible t-tuple over the alphabet $Z_g = \{0, 1, 2, ..., g-1\}$ must occur in at least one row. A solution for $t = 3$, $k = 5$, $g = 2$, $b = 10$ is shown in Figure 11.2. Every triple of values from $\{0, 1\}$, from $(0, 0, 0)$ to $(1, 1, 1)$, occurs in the first three columns of the array, and this is true of every other subset of three columns as required.

1	2	3	4	5
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Figure 11.2: A covering array $CA(3, 5, 2)$ of size 10.

A natural way to model the problem is to introduce a $b \times k$ matrix of integer variables, x_{ri}, for $1 \leq r \leq b$ and $1 \leq i \leq k$, such that $x_{ri} = m$ if the value in column i and row r of the array is m. However, it is hard to express the constraints that in every subset of t columns, every possible t-tuple must occur.

To make these constraints easier to express, Hnich *et al.* introduced *compound variables*, analogous to the variables of the dual graph transformation, to represent every t-tuple of columns in each row. In the case of a binary alphabet, each compound variable has domain $\{0, ..., 2^t\}$. There are still non-binary constraints on these variables: there is a global cardinality constraint on the compound variables corresponding to a given t-tuple in each row, to ensure that every value between 0 and $2^t - 1$ is assigned at least once. In addition, just as in the dual graph translation, there are binary constraints between the compound variables corresponding to a row that if they have columns in common, in terms of the original variables, they must agree on the values that they give to their shared variables.

These examples show that the dual variables of the dual graph translation can be practically useful in rewriting non-binary constraints, even without eliminating the non-binary constraints completely.

11.8.2 Permutation Problems

A well-studied class of problem with two standard viewpoints is the class of permutation problems. A CSP is a permutation problem if the union of the domains has the same number of elements as there are variables and each variable must be assigned a different value. Any solution assigns a permutation of the values to the variables. Other constraints in the problem determine which permutations are acceptable solutions.

Each possible value is assigned to exactly one variable and each variable is assigned exactly one value. The *dual* viewpoint was identified by Geelen [19]; it switches the roles of the variables and values. For example, the usual CSP model of the n-queens problem in which the variables represent the rows and the values represent the columns is a permutation problem; the dual model has the variables representing the columns and the values representing the rows. In this instance, the two viewpoints give the same CSP, so that one is not better than the other. In many permutation problems, however, the constraints are easier to express and propagate better in one viewpoint than the other. For example, the problem of finding an $n \times n$ magic square, containing the numbers 1 to n^2 arranged so that the sum of every row and column is the same, can be expressed as a permutation problem; we can either find the number to go in each cell of square, or decide which cell to put each number in. However, the constraints on the row and column sums are much easier to express in the first viewpoint than the second.

As described in the next section, rather than choosing one viewpoint or its dual, we can combine the two; much recent work on permutation problems has investigated this possibility.

11.8.3 Boolean Models

Another possible viewpoint for a permutation problem has a Boolean variable x_{ij} for every possible variable-value combination (or value-variable combination in the dual viewpoint). For instance, in the n-queens problem, the variables x_{ij}, $1 \leq i, j \leq n$ correspond to the squares of the board. The assignment $(x_{ij}, 1)$ means that there is a queen on the square in row i and column j, and $(x_{ij}, 0)$ means that there is not.

Similarly, a Boolean viewpoint can be derived from and CSP viewpoint with integer or set variables. For any assignment (x_i, j) in an integer viewpoint, there is a Boolean variable b_{ij} in the Boolean viewpoint; the assignment $(b_{ij}, 1)$ corresponds to the assignment (x_i, j), whereas any other assignment to x_i corresponds to $(b_{ij}, 0)$. For any assignment (X_i, S) in a viewpoint with set variables, and for any value $j \in S$, the Boolean variable b_{ij} is assigned the value 1.

The variables of the Boolean viewpoint are closely similar to the variables of the direct encoding of a CSP into SAT [49]. However, the Boolean viewpoint usually gives a less efficient CSP than the integer or set model. The transformation to a Boolean viewpoint is described here to emphasize that there is always a choice of models in representing a problem as a CSP; in practice, it is often more useful to try to convert an initial Boolean model into one with integer or set variables.

11.8.4 Different Perspectives

So far in this section, the examples of changing viewpoint have involved reformulating an existing viewpoint. However, for some problems, it may be possible to find a new viewpoint by viewing the problem from a different angle; this is potentially valuable, because the constraints expressed in a radically different viewpoint may express different insights into the problem and so show different ways of solving it.

A problem where many different viewpoints have been devised is the 'open stacks' problem, set for the first Constraint Modelling Challenge, in connection with the Modelling and Solving Problems with Constraints workshop at IJCAI'05. The submissions to the Challenge can be found at `www.dcs.st-and.ac.uk/~ipg/challenge`. The problem, as stated for the Challenge, is: "A manufacturer has a number of orders from customers to satisfy; each order is for a number of different products, and only one product can be made at a time. Once a customer's order is started (i.e. the first product in the order has been made) a stack is created for that customer. When all the products that a customer requires have been made, the order is sent to the customer, so that the stack is closed. Because of limited space in the production area, the number of stacks that are in use simultaneously i.e. the number of customer orders that are in simultaneous production, should be minimized."

A wide variety of viewpoints were represented amongst the Challenge entries. Perhaps the most obvious viewpoint has variables representing positions in the production sequence and values representing the products; this creates a permutation problem, so that this viewpoint also has a dual. One insight into the problem is that although ostensibly requiring a sequence of the products, it can in fact be solved by sequencing the customers; this gives a viewpoint where the variables are the positions in a sequence of customers; the value of the ith variable is the ith customer to have their order completed. Other viewpoints focus on the stacks: one has variables representing the customers, and the value assigned to a variable is the stack area that customer will use. Also focussing on the stacks, another viewpoint has a Boolean variable for each pair of customers: the value 0 means that they share a stack location, and 1 means that they do not. This last viewpoint relates very directly to the objective, since minimizing the maximum number of open stacks is equivalent to maximizing the number of customers that can share a stack location. Several other viewpoints also feature in the entries.

Different viewpoints can be used individually as the basis of a model of the problem. However, a more interesting approach is to combine different viewpoints; this will be discussed in the next section. When the viewpoints being combined are based on different insights into the problem, this potentially allows all these insights to contribute to solving the problem, rather than forcing the modeller to choose just one.

11.9 Combining Viewpoints

If two viewpoints $V_1 = \langle X_1, D_1 \rangle, V_2 = \langle X_2, D_2 \rangle$ for the same problem have been identified, a complete model of the problem can be constructed from each viewpoint, say $M_1 = \langle X_1, D_1, C_1 \rangle$, $M_2 = \langle X_2, D_2, C_2 \rangle$. Hence, the models are *mutually redundant*. It can be beneficial to combine the two models rather than to choose between them. The combined model has variables $X_1 \cup X_2$ and (in the simplest form of combination) constraints $C_1 \cup C_2 \cup C_c$, where C_c is a set of *channelling constraints*. The channelling constraints express the relationship between the two sets of variables, X_1, X_2, in such a way that assignments in either viewpoint can be translated into assignments in the other. This idea was introduced by Cheng, Choi, Lee and Wu [6].

The potential advantage of combining viewpoints in this way comes from propagating the constraints of the two models during the search for a solution. The search variables can be the variables of one of the viewpoints, say X_1 (this is discussed further below). As search proceeds, propagating the constraints C_1 removes values from the domains of the variables in X_1. The channelling constraints may then allow values to be removed from the domains of the variables in X_2. Propagating these value deletions using the constraints of the second model, C_2, may remove further values from these variables, and again these removals can be translated back into the first viewpoint by the channelling constraints. The net result can be that more values are removed within viewpoint V_1 than by the constraints C_1 alone, leading to reduced search. Cheng *et al.* give a detailed account of how the propagation in a combined model works, using the n-queens problems as a case study.

Law and Lee [29] discuss a process they term *model induction*; this uses two viewpoints, without combining them, and provides an insight into why multiple viewpoints can be useful. Given two viewpoints $\langle X, D\rangle$ and $\langle X', D'\rangle$, the constraints of the second viewpoint are translated into constraints in the first viewpoint, using the channelling constraints. The new constraints can be merged into the existing constraints with the same scope in the first viewpoint. Law and Lee showed that this brings new information into the first viewpoint and can speed up search.

In section 11.8, permutation problems were defined and the dual viewpoint described. In solving a permutation problem, it is often beneficial to combine the two viewpoints. In a permutation problem with k variables $x_1, x_2, ..., x_k$, the domain of each variable is $\{1, 2, ..., k\}$. The dual variables are $d_1, d_2, ..., d_k$, also with domains $\{1, 2, ..., k\}$. The channelling constraints defining the relationship between the variables of the two viewpoints are: $(x_i = j) \equiv (d_j = i)$, $\forall i, j$, $1 \leq i \leq k, 1 \leq j \leq k$. (Note that these can be more efficiently represented by a global *inverse* constraint [3] rather than n^2 binary constraints, although the binary constraints give the same propagation.)

Hnich, Smith and Walsh [27] consider both permutation problems and injection problems (which are similar, but have more values than variables). Several possible combined models for injection problems are investigated, in some cases using dummy values for the dual variables, to allow for the values that are not assigned to the original variables.

Cheng *et al.* [6] also give an example of combining an integer variable viewpoint with a set variable viewpoint in a nurse rostering problem; the problem can be viewed as either allocating shifts to nurses or as allocating nurses to shifts. The first viewpoint has an integer variable n_{ij} for each nurse i and day j; its value represents the shift that nurse i works on day j. The second viewpoint has a set variable S_{kj} for each shift k and day j; its value represents the set of nurses that work shift k on day j. The channelling constraints to combine the viewpoints are $(n_{ij} = k) \equiv (i \in S_{kj})$.

As well as the *inverse* constraint already mentioned, a number of other global constraints such as the *element* constraint relate two sets of variables and so can often be seen as channelling constraints between the variables of two viewpoints. (See Chapter 7, "Global Constraints".)

However, although it is not necessary for channelling constraints to be binary, they must ensure that assignments in one viewpoint can trigger constraint propagation in the other when only a few variables have been assigned. If constraint propagation via the channelling constraints can only occur when a complete assignment has been made (i.e. therefore when a solution has already been found) there is no benefit from the combination.

11.9.1 Selecting Constraints

It is clearly safe to combine two or more models of a problem into a single combined model, containing the variables and the constraints of both models, together with the channelling constraints. The constraints of either model will ensure that the solutions to the CSP correspond to the solutions to the problem, so that this will also be true of the combined model.

However, it is often unnecessary to include all the constraints of both models, and the search will be speeded up if some of the constraints are dropped.

In many cases, a motivation for combining viewpoints is that some of the constraints are hard to express (and propagate weakly) in one viewpoint and some are hard to express in the other. The combined model allows the constraints to be expressed in the most convenient viewpoint. In this situation, it often happens that the two complete models, one for each viewpoint, only exist in theory; the only model actually constructed is the combined model, with a mixture of constraints expressed in each viewpoint.

In the Golomb ruler example, the requirement that the pairwise differences between the marks on the ruler are all distinct can be expressed in terms of either viewpoint: either as the 4-ary constraints $x_j - x_i \neq x_l - x_k$ or as a single global constraint allDifferent($d_{12}, d_{13}, ..., d_{m-1,m}$). These are equally correct in ensuring that the solutions meet the condition; however, they are not equivalent in terms of propagation. [44] shows empirically that the allDifferent constraint (or a clique of $\neq$ constraints) gives much better results than the 4-ary constraints (if GAC is not maintained on the 4-ary constraints).

For permutation problems, where two viewpoints with variables $x_1, x_2, ..., x_n$ and $d_1, d_2, ..., d_n$ can be combined as described earlier, with the channelling constraints $x_i = j) \equiv (d_j = i)$, these channelling constraints are sufficient to ensure that the values assigned to $x_1, x_2, ..., x_k$ (and so also those assigned to $d_1, d_2, ..., d_k$) are distinct [27]. Hence, the constraints $x_i \neq x_j, 1 \leq i < j \leq n$ or allDifferent($x_1, x_2, ..., x_n$), required in the original model, are no longer needed in the combined model to ensure correct solutions. Maintaining arc consistency on the binary channelling constraints can prune more values than binary $\neq$ constraints on these variables, though fewer than GAC on the allDifferent constraint. Enforcing AC on a set of binary $\neq$ constraints, representing an allDifferent constraint, removes a value from the domain of a variable if that value is the only one value in the domain of another variable (e.g. because it has been assigned that value). Enforcing AC on the channelling constraints does the same pruning as the $\neq$ constraints, and in addition removes all values but one from the domain of a variable (and thereby effectively assigns the remaining value to the variable) if the remaining value does not appear in the domain of any other variable in $\{x_1, x_2, ..., x_k\}$. Hence, in a combined model of a permutation problem, binary $\neq$ constraints between the variables of either viewpoint are a waste of effort; an allDifferent constraint on one set of variables is not needed for correctness but in some problems may do sufficient additional pruning to give a smaller run-time than the channelling constraints alone.

[40] introduced the idea of a *minimal dual model* of a permutation problem: this has both sets of variables, the constraints (excluding the allDifferent constraint) of only one model and the channelling constraints. For some permutation problems, the constraints of one model are strictly stronger than those of the other, so that including both sets of constraints gives no benefit in terms of reducing search, and incurs an overhead in run-time. In [40], it is demonstrated empirically that for Langford's problem (problem 24 in CSPLib),

the minimal dual model generates the same search as a model using all constraints of both models, but has a much shorter run-time.

Choi, Lee and Stuckey [8] investigate theoretically when some of the constraints in one viewpoint are *propagation redundant* in a model which also has the constraints of another viewpoint and the channelling constraints. A constraint is propagation redundant if the propagation that it would cause is subsumed by the propagation resulting from other constraints in the model. Propagation redundant constraints can clearly be removed from the model, and should be removed since they only add an unnecessary overhead. (Note that unlike many other changes to a model, removing propagation redundant constraints does not depend on the search heuristics.) Choi *et al.* suggest that their approach can be automated.

11.9.2 Choice of Search Variables

When combining two (or more) viewpoints of a problem, there is a choice of which set of variables to use to drive the search. Since each viewpoint could be the basis for a model of the problem, assigning values to either set of variables would be sufficient to solve the problem. This is obviously true if the combined model contains all the constraints of both individual models; the combined model could be treated as either of the original models, together with some extra baggage. It is still true if the combined model does not contain all the constraints of both models, provided that every condition defining the solutions to the underlying problem is expressed as a constraint in one or other viewpoint.

For instance, in Langford's problem the constraints expressed in one viewpoint propagate better than those in the other, but searching on the variables of the second viewpoint, in a combined model, leads to solutions with less search effort [27].

Another possibility is to use both sets of variables together as search variables. This makes most sense if the variables are of the same type and if the variable ordering is dynamic; the next variable can then be chosen from either set, according to the state of the search (although one could imagine a static ordering which chose alternately from each set of variables, say). When a variable from either set is assigned a value, the channelling constraints ensure that the corresponding dual variable is immediately assigned a value too. Hence, although the number of search variables may appear to be twice as large as it need be, only half of them will be specifically assigned during the search. This search strategy, choosing the variable with smallest domain, has been successfully used with problems that can be modelled as permutation problems, by Hnich, Smith and Walsh [27].

11.9.3 Multiple Viewpoints

Models in which more than two viewpoints are combined are possible. Given that combining mutually redundant models can lead to additional constraint propagation, Cheng *et al.* [6] suggested that "it seems reasonable to combine and implement as many mutually redundant models as one can dream of." Dotú, del Val and Cebrián [10] investigated this empirically in solving instances of the quasigroup completion problem, considered as a multiple permutation problem. A quasigroup completion instance requires completion of an $n \times n$ Latin square when some entries have already been filled.

The initial model has variables $x_{ij}, 1 \leq i, j \leq n$ representing the cell in row i, column j. The domain of every variable is the set $\{1, ..., n\}$. Since the values in every row and in every column of the Latin square must form a permutation of the values 1 to n, two other models that are duals of this are possible: in one the variables $r_{ik}, 1 \leq i, k \leq n$ represent the column in which the value k appears in row i; in the other, the variables $c_{jk}, 1 \leq j, k \leq n$ represent the row in which the value k appears in column j. There are three sets of channelling constraints that link each pair of models, for instance $(x_{ij} = k) \equiv (r_{ik} = j)$. Dotú *et al.* found that overall, a model combining three viewpoints linked by three sets of channelling constraints performed well.

11.10 Symmetry and Modelling

Symmetry in CSPs, and symmetry breaking, is a large topic in its own right and dealt with in Chapter 10, but some aspects of symmetry and its interaction with modelling are worth discussing here.

As already mentioned, modelling a problem P as a CSP may introduce symmetry, by using distinct variables and/or values to represent entities that are indistinguishable in P.

An example is the second viewpoint for the n-queens problem, given earlier, which has a variable for each queen. This introduces an unnecessary notion of the 1st queen, the 2nd queen and so on, so that different solutions to the CSP can correspond to exactly the same layout of the board, but with the queen labelled 1 swapped with the queen labelled 2. Neither of the other two viewpoints given has this symmetry (although the n-queens problem has inherent symmetry which does appear in the other viewpoints). This illustrates that introducing symmetry can sometimes easily be avoided by choosing another viewpoint.

The golfers problem (problem 10 in CSPLib) is another case in which some viewpoints introduce symmetry. One instance of the problem is stated as: *32 golfers want to play in 8 groups of 4 each week, in such a way that any two golfers play in the same group at most once. How many weeks can they do this for?* The problem can be generalised to different sizes and numbers of groups. To model the problem of finding a schedule for n weeks, using integer variables, a possible viewpoint has 0/1 variables x_{ijkl}, where $x_{ijkl} = 1$ if player i is the jth player in the kth group in week l, and 0 otherwise. However, the players within each group, the groups within each week, the weeks within the schedule and the players themselves could all be permuted in any solution to give an equivalent solution.

The first symmetry (the players within the group) can be eliminated by using set variables to represent the groups: the set variable G_{kl} represents the kth group in week l, and the value of this variable represents the set of players forming that group. The constraints on these variables are that:

- the cardinality of each set is 4;
- the sets in any week do not overlap, i.e. for all l, the sets G_{kl}, $k = 1, ..., 8$ have an empty intersection;
- any two sets in different weeks have at most one member in common.

Constraint solvers that support set variables provide cardinality constraints, and constraints on the intersection of set variables, to allow such constraints to be expressed. Using set variables rather than integer variables is a common way to avoid introducing symmetry in this way: where the order of objects within a group is immaterial, the group can be modelled as a set rather than as a sequence, which would introduce symmetrically equivalent sequences.

The model of the car sequencing problem described by Dincbas, Simonis and van Hentenryck [9], discussed in section 11.6, is also a reformulation to avoid symmetry. The statement of the problem asks for a sequence of the cars to be produced, so that one obvious way to model it would be as a permutation problem, in which the variables are the slots in the sequence and the values are the cars, or v.v. However, two cars requiring the same options are effectively identical, so that this model would allow symmetrically equivalent sequences in which identical cars are swapped. Dincbas *et al.* avoid this by introducing classes of identical cars. This requires additional constraints to ensure that the correct number of cars in each class appear in the sequence.

Both ideas can be useful in other contexts, such as staff rostering. Suppose a crew is required for each shift. Some or all of the crew can often be treated as a set, e.g. if staff are not allocated specific roles, and the only requirement is that a minimum number must be provided, they can be represented as a set. If staff with identical skills can be treated as interchangeable in constructing a roster, it may only be necessary to count how many staff within each skill-set have been allocated.

In [41], further models of the golfers problem are given which eliminate more of the symmetry. The first has an integer variable for each pair of players, i_1, i_2: the value assigned to the variable p_{i_1,i_2} represents the week in which this pair of players plays together, with a dummy value in case they never play together. This viewpoint does not distinguish between the players within a group, or between the groups within a week. To allow the constraints to be expressed concisely, auxiliary set variables were also introduced, for each player i and each week l, representing the set of players that player i plays with in week l.

A final model presented in [41] also eliminates the symmetry due to the fact that the weeks of the schedule are interchangeable, although it only deals with the special cases of the golfers problem in which every player plays every other player at some point during the schedule. For each pair of players i_1, i_2, it has a set variable representing the group of players that the pair plays with, and another representing the other pairs of players that play together in the week that i_1 and i_2 play together. Unfortunately, the model has a very large number of variables, but it proved better than the earlier models for solving small instances. Note that it still has some of the original symmetry, due to the interchangeability of the players. This work does demonstrate that designing models with the intention of reducing the symmetry can sometimes be successful, although the resulting model may become rather complex.

11.10.1 Symmetry-Breaking Constraints

When there is symmetry in the chosen model of a problem (either symmetry introduced in modelling, or inherent in the problem), one possible way to eliminate or reduce it is to add symmetry-breaking constraints. Devising such constraints is beyond the scope of this chapter, but it is worth pointing out here that as a side-effect, such constraints often allow implied constraints to be derived that would not otherwise be possible.

This was observed in the template design problem [33] (problem 2 in CSPLib). The problem is to design templates for printing large sheets of card with items such as cat-food boxes. An order quantity is specified for each product, such as different flavours of cat-food. The overall objective is to minimize the total number of sheets that have to be printed (and so minimize waste), while fulfilling the order quantities for each product.

The t templates to be used are numbered in the model, but in practice are interchangeable; constraints are added to the model to eliminate this symmetry. The variable r_i represents the number of sheets of card to be printed from template i, and the symmetry-breaking constraints specify that $r_i \leq r_{i+1}$, for $1 \leq i < t$.

The objective is to minimize $p = \sum_i r_i$, i.e. the total number of sheets of card to be printed, and the number of templates needed, t, is at most 4 in the instances studied. Implied constraints can be added, derived from the symmetry-breaking constraints. For instance, if there are two templates, at most half the sheets are printed from one template and at least half from the other. Because of the symmetry-breaking constraints, we can add: if $t = 2$, $r_1 \leq p/2$ and $r_2 \geq p/2$; if $t = 3$, $r_1 \leq p/3$; $r_2 \leq p/2$ and $r_3 \geq p/3$; and so on.

Deriving implied constraints from symmetry-breaking constraints has been discussed in more detail by Frisch, Jefferson and Miguel [15]. They show, for instance, that adding lexicographic ordering constraints on the rows and columns to reduce the symmetry in CSP representing the Balanced Incomplete Block Design problem (prob28 in CSPLib) allows powerful implied constraints and a considerable simplification of the other constraints, giving for some instances a huge reduction in the time to solve the problem. In many problems, there are several distinct ways of adding constraints to give the same reduction in the symmetry; Frisch *et al.* suggest that in some cases the choice could be guided by considering the implied constraints that can then be derived.

11.11 Optimization Problems

Tsang [45] defines a Constraint Satisfaction Optimization Problem (CSOP) as follows:

A CSOP $\langle X, D, C, f \rangle$ is defined as a CSP $\langle X, D, C \rangle$ together with an optimization function f which maps every solution to a numerical value. The task in a CSOP is to find the solution T such that the value of $f(T)$ is either maximized or minimized, depending on the requirements of the problem.

If P is an optimisation problem, and $M_O = \langle X, D, C, f \rangle$ is a CSOP that models P, then every solution of C can be translated into exactly one solution of P and at least one optimal solution of P can be derived from a solution to C. (There is no requirement in this case that every optimal solution to P should be found as a solution of C.)

Typically, a CSOP is solved in a branch-and-bound fashion, adding a constraint whenever a solution T is found that the value of the optimization function must be better than $f(T)$ in any future solution. This constraint provides an increasingly tight bound and can prune the search for future solutions; eventually, if it is proved that no solution satisfying the current bound exists, the last solution found has been proved optimal. The adaptation of the branch-and-bound principle from operational research was described by van Hentenryck [47].

Often, however, an optimization problem is represented and solved as a CSP or as a sequence of CSPs. This is especially appropriate when the optimization function measures some feature of the CSP structure, typically the number of variables. Hnich, Prestwich and Selensky [25], for instance, describe modelling a problem in software testing in which the objective is to construct a set of test vectors with specified coverage properties: the objective is to minimize the number of test vectors required. The CSP has a matrix of variables to represent the test vectors and hence the optimization function is the number of rows in the matrix. A sequence of CSPs is constructed, adding a row to the matrix each time, and the first CSP in the sequence that has a solution represents an optimal solution to the problem.

Even when the optimization function can easily be represented by an additional variable within the CSP, the problem may be represented as a CSP rather than a CSOP. For instance, in [43], the objective in the SONET problem described in 11.6 is represented as a variable, and assigned first during the search. The values of the objective variable are assigned in ascending order, and hence the first solution found has the smallest possible value of the objective variable, i.e. is optimal. For the problem described, this was found (empirically) to be more efficient than a branch-and-bound approach. However, it would only be feasible if there were only a few values between the smallest value in the domain of the objective variable, after initial constraint propagation, and the optimal value.

In an optimization problem, a compound assignment that satisfies the constraints can be forbidden if it can be shown that for any solution that this assignment would lead to, there must be another solution that is equally good or better. Dominance rules are constraints that forbid compound assignments that are dominated in this way; they are similar to implied constraints, in their effect, but are not logical consequences of the constraints C and do not necessarily preserve the set of optimal solutions. Prestwich and Beck [32], on the other hand, consider dominance rules as strongly related to conditional symmetry in satisfaction problems.

Getoor, Ottosson, Fromherz and Carlson [21] describe a scheduling application (optimal on-line scheduling of photo-copiers and similar machines) in which dominance rules play an important part. (Note that Getoor *et al.* use the term redundant constraint.) They classify the types of dominance rule that they found, including lower and upper bounds on the schedule length for a job, derived by relaxing some of the constraints to give a simpler problem.

Useful dominance rules can often be very simple and obvious. This can also be true of implied constraints, but in satisfaction problems, the search heuristics tend to guide the search away from obviously wrong compound assignments; in optimization problems, the search at some point has to prove that there is no solution, unless there is a good bound on the objective that makes the proof trivial. In proving that a problem has no solution by exhaustive search, every possibility allowed by the constraints has to be explored.

For instance, in the SONET problem, described earlier [43], it is obviously suboptimal to have a SONET ring with only one node on it, since installing a node on a ring contributes to the cost, but the only reason to install a node on a ring is to allow it to communicate with another node on that ring. A constraint that every ring must have at least two nodes on it, and that there must be traffic between them, rules out these solutions and makes a significant difference to the search.

Optimization problems arising in scheduling, and the importance of propagating the value of the objective to prune the search, are discussed in Chapter 22.

11.12 Supporting Modelling and Reformulation

As will be clear from this chapter, there can often be many different ways to model a problem. Ideally, an automatic modelling system should generate the best model; but given the interaction between the model, the search algorithm and the search heuristics, there is not likely to be a single best model. We could envisage a system that generates a number of different models of a problem, and can advise that one is better than another under certain circumstances. Flener, Pearson and Ågren [13] describe a system that refines a specification to a model that uses matrices of Boolean variables. Systems that generate alternative models from a specification of the problem are described by Hnich [24], and in two related papers by Frisch, Hnich, Miguel, Smith & Walsh [16] and Frisch, Jefferson, Martinez Hernandez and Miguel [17]. The system described in the last papers can generate models with multiple viewpoints, linked by channelling constraints. [17] presents empirical results based on a number of problems, comparing the models produced with those described in the literature. For instance, the system generated 27 models of the SONET problem, described earlier; even so, this did not include all of those described in [43]. Comparing the models generated, other than empirically, is still a gap.

A completely different route to formalizing modelling is by identifying common *patterns* that can be transferred from one problem to another. Flener *et al.* [12] advocated a need to "identify, formalise and document these patterns of formulation and solution". Walsh [50] relates the idea to design patterns in architecture and software engineering. This seems the most effective support available for modellers at present; for instance, since the paper by Cheng *et al.* [6], the use of multiple viewpoints linked by channelling constraints has become commonplace, and dual viewpoints of permutation problems in particular have been thoroughly studied and understood.

Although there is some progress towards identifying a range of possible ways of modelling a problem, there is less progress towards identifying good models, except by trying them empirically. In the early days of constraint programming, models were sometimes compared by estimating the sizes of their search spaces, i.e. the product of the domain sizes. This could be a good indication of the search effort if the search algorithm simply did generate and test, but it is too simplistic for any more sophisticated search algorithm. Since the choice of model interacts with the choices of search algorithm and search heuristics, models can only be compared in the context of the other choices. Simonis *et al.* [38] describe the use of visualization tools to examine the progress of the search in detail and to compare the performance of different models; in principle, such tools can also be used to identify inefficiencies in the search and to guide further improvements to the model.

Some modelling advice has been devised; for instance, Simonis [37] gives '30 Golden Rules' for modelling. There are a few specific guidelines in the CP folk-lore, e.g. "Avoid Boolean models", and more generally, "Reduce the number of variables" or "Reduce the number of constraints". These guidelines are worth discussing, because although they have a grain of truth, they should not be taken too literally:

- *Reduce the number of variables.* Clearly, reducing the number of variables conflicts with using multiple viewpoints and/or auxiliary variables, which have been demonstrated to be a good approach to modelling. Furthermore, increasing the number of search variables, by assigning values to the extra variables, can reduce search. Even so, it is likely that a model which requires fewer variable assignments to describe the solutions to the problem will be a better model; hence, an integer model is likely to be better than a Boolean model of the same problem. However, this is only true if the variables chosen allow the constraints to be expressed in a way that propagates well; it would be easy, for instance, to artificially reduce the number of variables by making a single variable in the new model stand for a pair of variables in the old model, but in general, this will not result in a better model.

- *Reduce the number of constraints.* Again, this conflicts with introducing implied constraints, if taken literally. However, rewriting a set of constraints in a more compact form is likely to be beneficial, if the resulting constraints can propagate efficiently; this covers, for instance, combining constraints with the same scope or using a global constraints to replace a set of constraints. As before, however, simply conjoining constraints for the sake of reducing their number will not result in a better model if the new constraints cannot propagate efficiently.

One could equally well reverse this advice, to say "*Add* more variables and constraints". New variables (whether auxiliary variables or a complete new viewpoint), and constraints on these variables, that make explicit knowledge of the underlying problem that was not hitherto expressed, can allow the problem to be solved more easily.

However, with any changes to the model, whether the changes are adding variables and constraints or removing them, one caveat should be borne in mind: changes to the model that reduce search may not always reduce run-time. It may be necessary to test a model empirically in order to see whether a proposed change will in fact lead to solutions being found more quickly.

Bearing in mind this caveat (and also the interaction between the model, the search algorithm and the search heuristics), the best advice at present seems to be to aim for a rich model, using multiple viewpoints, auxiliary variables and implied constraints, incorporating as much insight into the problem as possible. The more we understand the problem and build that understanding into the model, the better we will be able to solve it.

Acknowledgements

I should like to thank Pascal van Hentenryck, Jimmy Lee, Jeff Choi and Yat Chiu Law for useful comments, and Ian Miguel for his help. This material is based on works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075.

Bibliography

[1] F. Bacchus and P. van Beek. On the Conversion Between Non-Binary and Binary Constraint Satisfaction Problems. In *Proceedings AAAI'98*, pages 311–318, 1998.

[2] A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*, pages 78–87, 2001.

[3] N. Beldiceanu. Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report Technical Report T2000/01, SICS, 2000.

[4] C. Bessière and J. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In *Proceedings CP'99*, pages 103–117, 1999.

[5] J. E. Borrett and E. P. Tsang. A Context for Constraint Satisfaction Problem Formulation Selection. *Constraints*, 6:299–327, 2001.

[6] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999.

[7] K. C. K. Cheng and R. H. C. Yap. Applying Ad-hoc Global Constraints with the case Constraint to Still-Life. *Constraints*, 11, 2006. (To appear).

[8] C. W. Choi, J. H. M. Lee, and P. J. Stuckey. Removing Propagation Redundant Constraints in Redundant Modeling. *ACM Transactions on Computational Logic*, 2006. (To appear).

[9] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Y. Kodratoff, editor, *Proceedings ECAI-88*, pages 290–295, 1988.

[10] I. Dotú, A. del Val, and M. Cebrián. Redundant Modeling for the QuasiGroup Completion Problem. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003*, LNCS 2833, pages 288–302. Springer, 2003.

[11] A. Fernández and P. M. Hill. A comparative study of eight constraint programming languages over the Boolean and finite domains. *Constraints*, 5:275–301, 2000.

[12] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix Modelling: Exploiting Common Patterns in Constraint Programming. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems - Towards Systematisation and Automation*, 2002.

[13] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In M. Bruynooghe, editor, *LOPSTR'03: Revised Selected Papers*, LNCS 3018, pages 214–232. Springer, 2004.

[14] E. C. Freuder. Modeling: The Final Frontier. In *Proceedings PACLP99, the 1st International Conference on the Practical Applications of Constraint Technologies and Logic Programming*, pages 15–21, 1999. Keynote address.

[15] A. Frisch, C. Jefferson, and I. Miguel. Symmetry-breaking as a Prelude to Implied Constraints: A Constraint Modelling Pattern. In *Proceedings of ECAI 2004*, pages 171–175, 2004.

[16] A. M. Frisch, B. Hnich, I. Miguel, B. M. Smith, and T. Walsh. Transforming and Refining Abstract Constraint Specifications. In J.-D. Zucker and L. Saitta, editors, *Abstraction, Reformulation and Approximation, 6th International Symposium, Proceedings SARA 2005*, LNCS 3607, pages 76–91. Springer, 2005.

[17] A. M. Frisch, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The Rules of Constraint Modelling. In *Proceedings IJCAI05*, pages 311–318, 2005.

[18] A. M. Frisch, I. Miguel, and T. Walsh. Extensions to proof planning for generating implied constraints. In *Proceedings of Calculemus-01*, pages 130–141, 2001.

[19] P. A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In B. Neumann, editor, *Proceedings ECAI'92*, pages 31–35, 1992.

[20] I. Gent, C. Jefferson, I. Lynce, I. Miguel, P. Nightingale, B. Smith, and A. Tarim. Search in the Patience Game 'Black Hole'. Technical Report CPPod-10-2005, CPPod Research Group, 2005. Available from http://www.dcs.st-and.ac.uk/s˜cppod/publications/reports/.

[21] L. Getoor, G. Ottosson, M. Fromherz, and B. Carlson. Effective Redundant Constraints for Online Scheduling. In *Proceedings of AAAI'97*, pages 302–307, 1997.

[22] W. Harvey and P. J. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 8:173 – 207, 2003.

[23] M. Henz. Don't Be Puzzled! In *Proceedings of Workshop on Constraint Programming Applications*, Aug. 1996.

[24] B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, University of Uppsala, 2003.

[25] B. Hnich, S. D. Prestwich, and E. Selensky. Constraint-Based Approaches to the Covering Test Problem. In B. Faltings, A. Petcu, F. Fages, and F. Rossi, editors, *Recent Advances in Constraints, Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2004, Revised Selected and Invited Papers*, LNCS 3419, pages 172–186. Springer, 2005.

[26] B. Hnich, J. Richardson, and P. Flener. Towards Automatic Generation and Evaluation of Implied Constraints. Technical Report Technical report 2003-014, Department of Information Technology, Uppsala University, Sweden, 2003.

[27] B. Hnich, B. M. Smith, and T. Walsh. Dual Models of Permutation and Injection Problems. *Journal of Artificial Intelligence Research*, 21:357–391, 2004.

[28] G. Katsirelos and F. Bacchus. GAC on conjunctions of constraints. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, LNCS 2239, pages 610–614. Springer, 2001.

[29] Y. C. Law and J. H. M. Lee. Model Induction: a New Source of CSP Model Redundancy. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, pages 54–60, 2002.

[30] B. A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. *IEEE Expert*, 5:16–23, June 1990.

[31] K. E. Petrie and B. M. Smith. Symmetry Breaking in Graceful Graphs. Technical Report APES-56-2003, APES Research Group, 2003. Available from http://www.dcs.st-and.ac.uk/~apes/apesreports.html.

[32] S. Prestwich and J. C. Beck. Exploiting dominance in three symmetric problems. In *Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, 2004.

[33] L. G. Proll and B. M. Smith. ILP and Constraint Programming Approaches to a Template Design Problem. *INFORMS Journal on Computing*, 10:265–275, 1998.

[34] J.-F. Puget. Constraint programming next challenge: Simplicity of use. In M. Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004*, LNCS 3258, pages 5 – 8. Springer, 2004. Invited talk.

[35] J.-C. Régin and J.-F. Puget. A Filtering Algorithm for Global Sequencing Constraints. In G. Smolka, editor, *Principles and Practice of Constraint Programming - CP97*, LNCS 1330, pages 32–46. Springer-Verlag, 1997.

[36] F. Rossi, C. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proceedings of ECAI-90*, pages 550–556, 1990.

[37] H. Simonis. Finite Domain Constraint Programming Methodology. Tutorial presented at the PACT 2000 conference. (Available as a Powerpoint presentation from the author.), 2000.

[38] H. Simonis, T. Cornelissens, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco. Using Constraint Visualisation Tools. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, LNCS 1870, pages 321–356. Springer, 2000.

[39] B. M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering Heuristics. In M. Wallace, editor, *Proceedings PACT97, 3rd International Conference*

on the Practical Application of Constraint Technology, pages 321–330. The Practical Application Company, 1997.

[40] B. M. Smith. Modelling a Permutation Problem. Research Report 2000.18, School of Computer Studies, University of Leeds, 2000.

[41] B. M. Smith. Reducing Symmetry in a Combinatorial Design Problem. In *Proceedings of CP-AI-OR'01, the International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2001.

[42] B. M. Smith. A Dual Graph Representation of a Problem in 'Life'. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, LNCS 2470, pages 402–414. Springer, 2002.

[43] B. M. Smith. Symmetry and Search in a Network Design Problem. In R. Bartak and M. Milano, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Proceedings of CPAIOR 2005 (2nd International Conference)*, LNCS 3524, pages 336–350. Springer, 2005.

[44] B. M. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings AAAI-2000 (Conference of the American Assocation for Artificial Intelligence)*, pages 182–187, 2000.

[45] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[46] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, LNCS 2239, pages 625–639. Springer, 2001.

[47] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[48] P. van Hentenryck and Y. Deville. The Cardinality Operator: A New Logical Connective and Its Application to Constraint Logic Programming. In *Proceedings of the 8th International Conference on Logic Programming (ICLP-91)*, pages 745–759, 1991.

[49] T. Walsh. SAT v CSP. In *Proceedings CP'2000*, pages 441–456, 2000.

[50] T. Walsh. Constraint patterns. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003*, LNCS 2833, pages 53–64. Springer, 2003. Invited talk.