CONSTRAINT PROGRAMMING

MICHELA MILANO

Dipartimento di Elettronica, Informatica e Sistemistica – Università degli Studi di Bologna

FRANCESCA ROSSI

Dipartimento di Matematica Pura ed Applicata – Università degli Studi di Padova

1 What is constraint programming?

Constraint programming [9, 1, 28] is a powerful paradigm for solving combinatorial search problems. Constraint programming is a multi-disciplinary research area, which combines techniques from artificial intelligence, operations research, databases, graph theory, and logic programming. The basic idea in constraint programming is that the user should state his problem by means of constraints, and a general purpose constraint solver should solve such constraints. A constraint solver can also be seen as a procedure that transforms a constraint problem into an equivalent (simpler) one. Constraints are relations, which specify the allowed ways of combining the values of some variables. A constraint satisfaction problem (CSP), being just a set of constraints, states which relations should hold among the given decision variables. For example, in scheduling exams at the university, the decision variables might be the times and locations of different exams, and the constraints might be on the capacity of each examination room (e.g. we cannot schedule more students to sit exams in a given room at any one time than the room's capacity) and on the exams scheduled at the same time (e.g. we cannot schedule two exams at the same time if they share students in common).

Given a constraint problem, a constraint solver finds an assignment of values to its variables that satisfies all the constraints. Sometimes finding any solution is not enough, but one should find optimal solutions according to one or more optimization criterion (e.g. minimizing the number of days over which exams need to be scheduled), or find all solutions, or replace (some or all) constraints with preferences, or also consider a distributed setting where constraints are distributed among several agents. Therefore the original constraint-based techniques, developed since the early 70's, have been extended and adapted to deal with such needs.

Constraint solvers find solutions by searching the solution space systematically, such as with backtracking or branch and bound algorithms, or use forms of local search which may be incomplete, i.e., it may not find the optimal solution or a feasible one even if it exists. Systematic methods usually use a mix of search and inference, where inference consists of the so-called constraint propagation, which propagating the information contained in one constraint to the neighboring constraints. Such kind of inference is useful since it can reduce parts of the search space.

Global constraints are often used to help modelling and solving a problem. In fact, global constraints model complex constraints that occur often in real life. Moreover, they come equipped with special propagation procedures which are very efficient to achieve a certain degree of inference over them.

Constraint problems on finite domains are in general NP-complete. However, there are several subclasses of constraint problems which can be solved polynomially. Ways of characterizing such classes involve the connectivity structure among the variables and the constraints, or the language to define the constraints. For example, constraint problems where the connectivity graph has the form of a tree are polynomial to solve.

When trying to model a real-world problem via a set of constraints, one may realize that the problem is overconstrained. Thus, no solution would be found if all constraints are considered. Actually, some constraints are not mandatory, but should rather be modelled as preferences. Soft constraints provide a formalism to do this, as well as techniques to find an optimal solution according to the specified preferences. Many of the constraint solving methods like search and constraint propagation can be adapted to be used with soft constraints.

Constraint programming has proven useful in important applications from industry, business, manufacturing, and science. Important extensions have been done encapsulating Operations Research techniques in CP solvers for solving real life applications. Example of application areas where constraint programming is now one of the most successful technologies are scheduling, resource alloca-

tion, timetabling, and conÞguration. However, constraint programming has also been successfully applied to other areas, such as timetabling, biology, and system design.

Parts of this paper are based on [37].

2 Modelling a constraint problem

Modelling a real-world problem as a CSP is not an easy task, since many problems are ill-speciÞed and/or over-constrained; this requires a long interaction with whomever speciÞes the problem. It is also a very crucial task, since the chosen model can greatly inßuence the efÞciency of the constraint solver.

Choosing a model means deciding the variables, their domains, and the constraints that apply to these variables.

Given a problem, there are many logically equivalent models for it. In fact, one can choose to use variables to model different items, and, consequently, to have different constraints. For example, when modelling a nurse scheduling problem in a hospital, we may assign time slots to nurses, or nurses to time slots. In many cases, such models should not be used as alternatives, but rather as different (redundant) components of a single model. Consistency among such components should then be maintained; this is usually done via so-called channelling constraints [6]. As redundant models can help, also redundant constraints can be helpful. Redundant constraints do not add anything to the problem semantics, but they are often used to help prune the search space.

Global constraints are an essential ingredient of any good model. Global constraints are complex constraints, usually involving several variables, providing a concise formulation of common subproblems. In fact, they represent patterns which often occur in real-life problems. Therefore, they help generating a compact model for a problem. Moreover, each global constraint is equipped with a speciÞc (propagation) algorithm, which exploits the semantics of the constraint to prune the domains of its variables as much as possible, and thus to make the search for a solution faster. Therefore, global constraints are also useful to solve a problem more efÞciently. The canonical example of a global constraint is the all-different constraint [34]. An all-different constraint over a set of variables states that the variables must be pairwise different. The all-different constraint is widely used in practice and is built-in in most, if not all, commercial and research-based constraint programming systems. Hundreds of global constraints have been proposed [2].

Symmetries occur naturally in many problems. Consider for example a university timetabling problem where several classrooms are identical [7]. Solving a problem whose model has a lot of symmetries may be very inefÞcient, since we may waste much time visiting parts leading to symmetric solutions, or (even worse) to symmetric non-solutions. Possible remedies involve adding constraints which eliminate symmetric solutions or modifying the search procedure to avoid visiting symmetric states.

Classical constraints have variables ranging over Þnite domains. However, real-world problems may need different domains, such as reals and sets. Constraint programming has therefore been extended to deal with many other domains, to provide faithful models of real-world problems.

3 Constraint propagation

Once we have chosen a model for the problem at hand, we must solve it via a constraint solver. Solvers usually solve a constraint problem via searching the set of solutions. However, no matter which search technique is used, one of the key ingredient of a constraint solver is constraint propagation. Constraint propagation is a form of inference that modiÞes the problem without changing its semantics (that is, the problem solutions). This inference eliminates some local inconsistencies. A local inconsistency is an instantiation of some of the variables that satisÞes the constraints among such variables but cannot be extended to one or more additional variables, and thus it cannot be part of any solution.

There are several kinds of constraint propagation, related to as many notions of local consistency. However, arc consistency (AC) [27] is the most used local consistency notion in practice. A constraint problem is arc consistent if all its constraint are arc consistent. A constraint c on two variables, say x and y, is AC if for all values in the domain of x there is at least a value in the domain of y that satisÞes C. If the constraint is not AC, it can be made AC by repeatedly removing unsupported values from the domains of its variables.

When the constraints are not binary, a generalized form of arc-consistency, called Generalized Arc-Consistency (GAC), is used. GAC ensures that, for each constraint, all elements in the domain of every variables of the constraint participate in some solution of the constraint.

Many algorithms for enforcing AC or GAC have been proposed. They all achieve the same result but use different data structures or support keeping techniques, thus having different space and time complexities. An optimal algorithm for an arbitrary constraint has $O(rd^r)$ worst case time complexity, where r is the arity of the constraints and d is the size of the domains of the variables [30].

Higher levels of local consistency can be considered. For example, path-consistency [31] is concerned with all the triangles made of three variables and the constraints among them. However, the higher the level of consistency, the more expensive it is to enforce such a level of consistency. So one must in general reach a tradeoff between the cost of the constraint propagation, and the amount of pruning achieved.

Global constraints have their own ad hoc propagation algorithm, which enforces a certain level of consistency among their variables. Such ad hoc algorithms are usu-

ally much more efficient than general-purpose propagation techniques such as GAC. For example, the all-different constraint can be made GAC with its ad hoc algorithm in $O(r^2 d)$ time in the worst case, using a maximum matching algorithm on a bipartite graph.

4 Search

Usually a CSP is solved by searching its solution space. A search algorithm for solving a CSP can be either complete or incomplete. Complete, or systematic, algorithms, such as backtracking search, guarantee that a solution will be found if one exists. Moreover, if the search is for an optimal solution, then they always find one of the optimal ones. On the other hand, incomplete, or non-systematic, algorithms, such as local search, may fail to find a solution, or an optimal one. However, such algorithms are often effective at finding a solution if one exists, and can be used to find an approximation to an optimal solution.

A search tree is just a tree where the root is the given problem, to be solved, and where each internal node is obtained by the addition of a branching constraint to the father node (for instance by an instantiation of one of the variables).

Backtracking search usually performs a depth-first traversal of the search tree. At each node, an uninstantiated variable is selected, and one of its values is assigned to it. The constraints of the problem are used to check whether the new instantiation is possible. If the selected variable cannot be instantiated, backtracking occurs.

Since constraint problems are NP-complete, backtracking search has an exponential worst case time complexity. However, the efficiency efficiency of backtracking search can be improved by using heuristics for variable and value selection, intelligent techniques for non-chronological backtracking, or also by including some form of constraint propagation at each node of the search tree.

In fact, as noticed above, constraint propagation can eliminate inconsistencies, thus reducing variable domains. This can reduce the branching factor of the search tree and, if all values are eliminated from a variable domain, it can allow a backtracking phase to initiate earlier.

When we have to find an optimal solution, branch and bound search can be used. Depth First Branch and bound performs, as backtracking search, a depth-first traversal of the search tree. Recently, also Best-first branch and bound has been integrated into CP solvers. At each node it keeps an over-estimation of the quality of the complete assignments below the current node, which is compared to the quality of the best solution found so far. When the over-estimation is worse than the current best, the subtree can be pruned because it contains nothing better than what we already have. The efficiency of this algorithm depends largely on its pruning capacity, that relies on the quality of its over-estimation. Thus many efforts have been made to improve it.

Tree search is not the only possibility. Local search is fundamentally different than backtracking search: search is performed by passing from one complete instantiation to another one. Each complete instantiation is evaluated by a cost function, which measures how far we are from a solution, or from an optimal solution. For satisfaction problems, a standard cost function is the number of constraints that are not satisfied. For optimization problems, the cost function is a measure of the solution quality.

Choices of the starting instantiation, of the neighborhood, of where to move in the neighborhood, and of when to stop are crucial for the quality of a local search algorithm. For this purpose multi-starts, simulated annealing, and tabu search have been proposed.

CP systems such as COMET [42] support local search through facilities to propagate the consequences of change. COMET has the same modelling power of Constraint Programming languages, since it uses exactly the same constraints, but it differs in how constraints are built from a software engineering viewpoint. In CP, constraints have an embedded filtering algorithm which works on domain variables and removes inconsistent values. Every time an event happens (like domain value removal or the variable fixing and the changing in a domain bound) constraints are awaked and propagate. In COMET, constraints are again software components that are maintained in a constraint store and awaked each time one or more of their variables change one value (i.e., the local search is moving from one solution to another). Constraints are called *differentiable constraints* and they maintain propertied such as the satisfiability, or the violation degree, and how much the involved variables contribute to it. Constraints can be queried to evaluate the effect of local moves on the properties they preserve.

5 Soft constraints

It is often the case that, after having listed the desired constraints among the decision variables, there is no way to satisfy them all. That is, the problem is *over-constrained*. Even when all the constraints can be satisfied, and there are several solutions, such solutions appear equally good, and there is no way to discriminate among them. These scenarios often occur when constraints are used to formalize desired properties rather than requirements that cannot be violated. Such desired properties should rather be considered as *preferences*, whose violation should be avoided as far as possible. *Soft constraints* provide one way to model such preferences.

There are many classes of soft constraints. Examples are: fuzzy, possibilistic, probabilistic, weighted. In fuzzy constraints, each assignment of the variables involved in a constraint has a preference between 0 an 1, the preference of a complete instantiation is the minimum preference given by the constraints, and a complete instantiation

is optimal if it has the highest preferences. In weighted constraints, each constraint has a weight, the cost of an instantiation is the sum of all weights of the violated constraints, and an optimal solution is a complete instantiation with minimal cost.

The literature contains also at least two general formalisms to model soft constraints, of which all the classes above are instances: *semiring-based constraints* [4] and *valued constraints* [39]. Both formalisms rely on similar algebraic structures, and have the same expressive power if preferences are totally ordered. However, partially-ordered preferences, which can be expressed in the semiring-based formalism, can be useful in multi-criteria optimization, since in this case there could be situations which are naturally not comparable.

Soft constraint problems are as expressive, and as difficult to solve, as constraint optimization problems, which are just constraint problems plus an objective function. In fact, given any soft constraint problem, we can always build a constraint optimization problem with the same solution ordering, and viceversa. Therefore, branch and bound is a natural choice to solve such problems. As backtracking search can be improved by including constraint propagation, the same can be done, in some cases, also for branch and bound for soft constraints. Soft constraint propagation is just an adaptation of the usual notions of constraint propagation to soft constraints. However, while constraint propagation in classical constraints prunes the domains or the constraints while maintaining the same set of solutions, in soft constraints it modifies (worsens) the preferences (or the costs).

Unfortunately, in general this modification can change the semantics of the soft constraint problem. There are however cases where the semantics is maintained: when preference combination is idempotent. This is for example the case of the fuzzy constraint class, where combination is via the min operator.

Many real problems do not rely on idempotent operators because such operators provide insufficient discrimination, and rather rely on frameworks such as weighted or lexicographic constraints, which are not idempotent. For these classes of soft constraints, equivalence can still be maintained, compensating the addition of new constraints by the ÓsubtractionÓ of others. This can be done in all *fair* classes of soft constraints [8], where it is possible to define the notion of ÓsubtractionÓ Soft constraint propagation, used within a branch and bound algorithm, can help obtaining a tighter over-estimation, and thus achieving more pruning.

Soft constraints are a way to represent preferences. There are also other ways to do this, such as CP-nets. CP-nets and soft constraints have complementary advantages and drawbacks. Attempts to merge them into a unique framework have been done [12]. Preferences are also the subject of classical voting theory, which has been recently extended and adapted to account for multi-agent preference aggregation based on soft constraints [32].

Soft constraints have been applied to many fields, such as bioinformatics and computer security [5].

6 Hybrid CP/OR techniques

In the last decade, a very exciting research line has been explored to improve the performances of Constraint Programming solvers for a number of applications. A number of operations research methods have found their way into constraint programming, see [29]. This development is very natural, since these two techniques present many similarities. They both describe problems via a declarative model, they both are based on constraints, and tree search is the way both techniques explore the search space. However, while the view of constraints in CP is local, i.e., constraints are software components embedding a filtering algorithm that works locally, Operations Research techniques consider the constraint set as a whole cloth, considering them all in the linear relaxation. Constraint Programming is more focussed on the feasibility part, while OR concentrates more on the optimality part. Therefore, they show similarities and differences that suggest that their merging could lead to advantages in both sides.

There are mainly two directions for integrating CP and OR. The first concerns the use of OR based relaxations in CP, while the second explores problem decomposition and faces each subproblem with the most suited solver.

6.1 Embedding relaxations into CP.

When we face a combinatorial optimization problem, an additional filtering w.r.t. traditional propagation can be done. The one based on *optimality reasoning*, aimed at removing those values that are proven sub-optimal. To do so, Constraint Programming, more and more often, embeds algorithms and techniques from Operations Research. The use of relaxations in CP is becoming an essential component for the efficient solution of hard combinatorial problems.

Relaxations can be encapsulated in different ways, the most straightforward being the use of a linear solver as if it was a global constraint. This techniques has been proposed first in [36] and [3] and extended by [33].

For being effective the interaction between the two solvers should last during the overall computation. In fact, the linear programming solver should achieve a tight integration with the standard constraint propagation.

Clearly this procedure is more effective if the bound is tight. Therefore, Refalo [33] has proposed to tighten the relaxation with cutting planes added during search and coming from CP global constraints. In particular, cutting planes can be derived from global constraints representing sub-problems and added to the Linear Programming model so as to tighten the overall relaxation. Refalo has provided

many examples and shown that it is very effective in practice.

An additional way to use relaxations in Constraint Programming is to integrate them within global constraints. One can optimally solve the problem represented by the global constraint plus an objective function, as it happens for instance for the all-different constraint with costs. In this case, we can achieve generalized arc consistency by using for instance a network ßow algorithm [35]. If instead the constraint represents an NP-hard problem, like in case of the global path constraint for expressing the Travelling Salesperson Problem, then we could embed a relaxation into the constraint and perform cost-based Þltering [16]. The only requirement is that the relaxation provides its optimal solution and a gradient function measuring the variable-value assignment cost. Again the relaxation can be tightened via cutting planes as proposed in [17].

6.2 Problem Decomposition

A second, very interesting, way to cope with complex hard combinatorial problem is to decompose them in easier subproblems, and identify the solver most suited for each part. Clearly the solvers should interact and exchange information. One of the most successful examples is the CP-based Benders Decomposition [23].

Benders decomposition has been studied in the 60õ and is an effective method for solving a variety of structured problems. It is particularly suited for those problems where Þxing a number of variables, called *hard* variables, makes the problem simpler.

Instead of blindly trying tentative values for the *hard* variables we can solve a master problem (which takes into account constraints on *hard* variables). After Þxing *hard* variables, a subproblem can be solved, taking into account the remaining variables. The process iterates and converges to the optimal solution. The iteration between the master and the subproblem is regulated by the so called Benders cuts.

In Operations Research, the subproblem should be a linear program while in [23] this restriction has been relaxed deÞning the Logic-Based Benders Decomposition framework. In this setting, the subproblem can be expressed as a Constraint Satisfaction Problem.

An interesting and successful application of Logic-based Benders Decomposition is scheduling with alternative resources [21], [22], [19]. The allocation is done via Integer Linear Programming, while scheduling is solved through a Constraint Programming solver. The two solvers interact via the generation of no-goods and cutting planes. The procedure is proved to converge to the optimal solution and performances improve up to three order of magnitude with respect to a single approach.

Another decomposition technique is CP-based column generation and branch and price [15]. It is a very useful approach for problems for which an integer-linear model would involve too many (sometimes an exponential number of) variables. The purpose is to enable an optimal solution to be found, and proven optimal, without ever considering more than a small proportion of these variables - certainly only a number that grows polynomially with the problem size. Therefore the problem is decomposed into two parts: the master problem working on a subset of the columns (variables), while the subproblem decides which column is convenient to add to the master at the next iteration.

More typically column generation is the main algorithm and the hybridisation comes through the choice of algorithms used to solve the subproblem. In the classic application of column generation to crew scheduling, the algorithm for the subproblem is often a shortest path algorithm on a network whose edge costs are dual prices inherited from the master problem. CP-based column generation has been used for solving the travelling tournament problems [14], crew rostering and scheduling [40] and employee timetabling [10].

7 Constraint Programming Languages

Constraints can be, and have been, embedded in many programming environments, but some are more suitable than others. The fact that constraints can be seen as relations or predicates, that their conjunction is a *logical and*, and that backtracking search is a basic methodology to solve them, makes them very compatible with logic programming [26]. The addition of constraints to logic programming has given the *constraint logic programming* paradigm [25, 28].

Syntactically, constraints are added to logic programming by just considering a speciÞc constraint type (for example, linear equations over the reals) and then allowing constraints of this type to appear in the body of the clauses. Besides the usual resolution engine of logic programming, one has a (complete or incomplete) constraint solving system, which is able to check the consistency of constraints of the considered type. This simple change provides many improvements over logic programming. First, the concept of uniÞcation is generalized to constraint solving: the relationship between a goal and a clause (to be used in a resolution step) can be described not just via term equations but via more general statements, that is, constraints. This allows for a more general and ßexible way to control the ßow of the computation. Second, expressing constraints by some language (for example, linear equations and disequations) gives more compactness and structure. Finally, the presence of an underlying constraint solver, allows for the combination of backtracking search and constraint propagation, thus generating more efÞcient complete solvers.

CLP is not only a programming language, but a programming *paradigm*, which is parametric with respect to the class of constraints used in the language. Constraint logic programming over Þnite domains was Þrst implemented in the late 80õ by Pascal Van Hentenryck [20]

within the language CHIP [11]. Since then, newer constraint propagation algorithms have been developed and added to more recent CLP(FD) languages, like GNU Prolog and ECLiPSe [24]. Constraint logic programming has also been extended to deal with sets and multi-sets, to be able to handle both Þnite domain constraints and constraints over sets/multi-sets [13].

Constraint-based tools have also been provided for imperative languages in the form of libraries. The typical programming languages used to develop such solvers are C++ and Java. ILOG is one the most successful companies to produce such constraint-based libraries and tools.

Constraints have also been successfully embedded within concurrent constraint programming [38], where concurrent agents interact by posting and reading constraints in a shared store. Languages which follow this approach to programming are AKL and Oz.

High-level modelling languages exist for modelling constraint problems and specifying search strategies. For example, OPL [41] is a modelling language in which constraint problems can be naturally modelled and the desired search strategy easily speciÞed, while COMET is an OO programming language for constraint-based local search [42]. CHR (Constraint Handling Rules) is instead a rule-based language related to CLP where constraint solvers can be easily modelled [18].

8 Promising research directions

When using a constraint solver, often it is not easy to understand what went wrong, or why a certain solution is returned rather than another one. Explanation tools could greatly help in making constraint technology easier to use. system.

Preference elicitation allows users to intelligently interact with a constraint system without being forced to state all their constraints, or preferences, at the beginning of the interaction. This can also be useful in scenarios where the users want to avoid revealing all their preferences, for example for privacy reasons.

Even when the user is willing to state all the information at the beginning of the interaction, sometimes it may be difÞcult for him to actually state it in terms of constraints. For example, it could be easier to state examples of desirable or unacceptable solutions. In this cases, machine learning techniques can be helpful to learn the constraints from the partial and possibly imprecise user statements.

Uncertainty occurs in many real-life situations. involve stochastic constraint programming, possibilistic constraints, and quantiÞed constraints. More work is needed to handle uncertainty and thus make constraint programming more widely usable.

Fruitful cross-fertilizations with related areas of research, such as SAT, Operations Research, knowledge representation, multi-agent systems, and belief revision, will certainly produce many useful results for constraint programming and its applications.

REFERENCES

[1] K. R. Apt. *Principles of Constraint Programming*. CUP, 2003.

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

[3] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plumer, editors, *Logic Programming: formal Methods and Practical Applications*, pages 245 272. North Holland, 1995.

[4] S. Bistarelli, U. Montanari, and F. Rossi. Semiring based constraint solving and optimization. *Journal of the ACM*, 44(2):201 236, 1997.

[5] S. Bistarelli, S.N.Foley, and B. OÕSullivan. Detecting and eliminating the cascade vulnerability problem from multi-level security networks using soft constraints. In *Proc. IAAI-04*, 2004.

[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] D.A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M. Smith. Symmetry deÞnitions for constraint satisfaction problems. In P. van Beek, editor, *Proc. CP 2005*, pages 17 31. Springer, 2005.

[8] M. Cooper. High-order consistency in Valued Constraint Satisfaction. *Constraints*, 10:283 305, 2005.

[9] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[10] S. Demassey, Gilles Pesant, and L.-M. Rousseau. Constraint programming based column generation for employee timetabling. In *Proc. CPAIOR*, volume 3524 of *LNCS*, page 140, 2005.

[11] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems*. Tokyo, Japan, 1988.

[12] C. Domshlak, S. Prestwich, F. Rossi, K. B. Venable, and T. Walsh. Hard and soft constraints for reasoning about qualitative conditional preferences. *Journal of Heuristics*, 12(4/5), 2006.

[13] A. Dovier, C. Piazza, and G. Rossi. Multiset rewriting by multiset constraint solving. *Romanian Journal of Information Science and Technology*, 4(1 2):59 76, 2001.

[14] K. Easton, G. Nemhauser, and M. Trick. Solving the traveling tournament problem: A combined integer programming and constraint programming approach. In *Proc. PATAT*, 2002.

[15] T. Fahle, U. Junker, S. E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint programming based column generation for crew assignment. *Journal of Heuristics*, 8(1):59 81, 2002.

[16] F. Focacci, A. Lodi, and M. Milano. Cost-based domain Þltering. In *Proc. CP'99*, pages 189 203, 1999.

[17] F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming: an hybrid approach. In *Proc. CP 2000*, pages 187 201, 2000.

[18] T. Fruhwirth. Theory and practice of constraint handling rules. *Journal of Logic programming*, 37:95 138, 1998.

[19] I. E. Grossmann and V. Jain. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing*, 13:258 276, 2001.

[20] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[21] J. N. Hooker. A hybrid method for planning and scheduling. In *Proc. CP 2004*, pages 305 316, Toronto, Canada, Sept. 2004. Springer.

[22] J. N. Hooker. Planning and scheduling to minimize tardiness. In *Proc. CP 2005*, pages 314 327, Sites, Spain, Sept. 2005. Springer.

[23] J. N. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96:33 60, 2003.

[24] IC-PARC. *The ECLiPSe 4.2 Constraint Logic Programming System*. http://www.icparc.ic.ac.uk/eclipse/, 1999.

[25] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19 and 20, 1994.

[26] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1993.

[27] A. K. Mackworth. Consistency in networks of relations. *AI Journal*, 8:99 118, 1977.

[28] K. Marriott and P. J. Stuckey. *Programming with constraints: an introduction*. MIT Press, 1998.

[29] M. Milano. *Constraint and Integer Programming*. Kluwer, 2004.

[30] R. Mohr and G. Masini. Good old discrete relaxation. In *Proc. ECAI-88*, pages 651 656, Munchen, Germany, 1988.

[31] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inform. Sci.*, 7:95 132, 1974.

[32] M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh. Aggregating partially ordered preferences: possibility and impossibility results. In *Proc. TARK X, ACM Digital library*, 2005.

[33] P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Proc. CP 2000, LNCS 1894*. Springer-Verlag, Berlin Heidelberg, 2000.

[34] J.-C. Regin. A Þltering algorithm for constraints of difference in CSPs. In *Proc. AAAI-94*, pages 362 367, Seattle, 1994.

[35] J.C. Regin. Arc consistency for global cardinality constraints with costs. In *Proc. CP '99*, 1999.

[36] R. Rodosek, M. Wallace, and M.T.Hajian. A new approach to integrating Mixed Integer Programming and Constraint Logic Programming. *Annals of Operational Research*, 1997. Recent Advances in Combinatorial Optimization.

[37] F. Rossi, P. van Beek, and T. Walsh. Constraint programming. In *Hanbook of Knoweldge Representation, F. van Hermelen, V. Lifschitz, B. Porter eds.* Elsevier, to appear in 2006.

[38] V. Saraswat. *Concurrent constraint programming*. MIT Press, 1993.

[39] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: hard and easy problems. In *Proc. IJCAI 1995*, pages 631 637, 1995.

[40] A. V. Moura T. H. Yunes and C. C. de Souza. Hybrid column generation approaches for urban transit crew management problems. *Transportation Science*, 39(2):273 288, 2005.

[41] P. van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[42] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, Cambridge, 2005.