An Overview of Finite Domain Constraint Programming

Martin Henz
School of Computing
National University of Singapore
henz@comp.nus.edu.sg

Tobias Müller
Programming Systems Lab
Universität des Saarlandes
tmueller@ps.uni-sb.de

Abstract

In recent years, the repertoire of available techniques for solving combinatorial problems has seen a significant addition: finite domain constraint programming. This technique is best seen as a framework for combining software components to achieve problem-specific tree search solvers. Its strength depends on the synergy that can be achieved between these components. In this paper, we give an overview of constraint programming for solving combinatorial problems. We focus on practical aspects and highlight connections to Operations Research techniques and their applications.

Keywords: Constraint programming, combinatorial search, tree search.

1 Introduction

Constraint programming is a heterogenous field of research, ranging from theoretical topics in mathematical logic to practical applications such as job-shop scheduling. Constraints under consideration can be of symbolic nature such as tree constraints used in natural language processing, or of numerical nature, operating on real or integer variables. In this overview, we focus on a branch of constraint programming that has recently been applied to combinatorial search and optimization problems, namely finite domain constraint programming (CP(FD)). CP(FD) developed in the 1980s out of constraint logic programming [JM94], an extension of the logic programming paradigm. Since then, two developments turned CP(FD) into a solving technique that can compete in some application areas with more established Operational Research techniques for combinatorial search. Firstly, techniques and algorithms from Operational Research such as application-specific filtering algorithms and branch-and-bound search and from Artificial Intelligence, such as consistency algorithms [Mac77] and limited discrepancy search [HG95] were integrated in the constraint programming framework. Secondly, high quality software systems became available that support the development of constraint-based solutions to combinatorial problems. Initially, these systems were extensions of programming systems for Prolog; a current example is the CHIP system [DVS$^+$88]. The C++ library Ilog Solver [ILO97] demonstrated that the solving paradigm of constraint programming is programming language independent. The aim of the constraint programming languages Oz [Smo95, Moz99] and Claire [CL96b] and the modeling language OPL [Hen99] was to combine an expressive symbolic language for problem modeling with extensive support for problem solving.

To appear in *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore, 2000.

This paper guides the reader through the process of solving combinatorial problems using constraint programming. Section 2 gives an overview of the paradigm, explaining CP(FD) as a framework, in which *propagation*, *branching* and *exploration* algorithms cooperate for problem solving. Sections 3, Section 4 and Section 5 describe propagation, branching and exploration algorithms in detail. This paper is intended as an overview of CP(FD). Stuckey and Marriott [MS98] give a thorough treatment of constraint programming in general, Van Hentenryck [Hen99] describes problem modeling and solving using CP(FD) and Wallace [Wal96] presents an overview of applications of constraint programming.

2 Problem Solving with Finite Domain Constraint Programming

We focus on discrete search and optimization problems. Thus, decision variables in the considered problem models represent integers. A constraint store stores information on such variables in the form of the set of possible values that the variable can take; this set is called the current domain of the variable. More formally, the constraint store is a conjunction of constraints of the form $x \in S$, where S is a set of integers. These constraints are called basic constraints. Computation starts with an initial domain for each variable as given in the model. Some constraints can be directly entered in the constraint store by strengthening the basic constraint on a variable. For example, the constraint $x \neq 5$ can be expressed in the constraint store by removing 5 from the domain of x.

Other more complex constraints are represented by computational agents called propagators. Each propagator observes the variables given by the corresponding constraint in the problem. Whenever possible, it strengthens the constraint store with respect to these variables by excluding values from their domain according to the corresponding constraint. For example, a propagator for the constraint $x \leq y$ observes the upper and lower bounds of the domains of x and y. A possible strengthening consists of removing all values from the domain of x that are greater than the upper bound of the domain of y.

The process of propagation continues until no propagator can further strengthen the constraint store. The constraint store is said to be stable. At this point, many problem variables typically have still non-singleton domains. Thus the constraint store does not represent a solution yet, and search becomes necessary.

Search for solutions is implemented by choice points. A choice point generates a branching constraint c. From the current stable constraint store cs, two new constraint stores are created by adding c and $\neg c$, respectively, to cs. Typically, the new constraint stores are not stable, in other words c and $\neg c$ trigger some propagators in the respective new store. After stability is reached again, this branching process is continued recursively on both sides until the resulting store is either inconsistent or represents a solution to the problem.

Finite domain constraint programming is best seen as a software framework for combining software components to achieve problem-specific tree search solvers. These software components can be organized into three families.

Propagation algorithms implement individual constraints by describing how the constraints can be employed to strengthen the constraint store.

Branching algorithms select branching constraints at each node of the search tree after all propagation has been done. Branching algorithms define the size and shape of the search tree.

Exploration algorithms describe which part of a given search tree is explored and in which order.

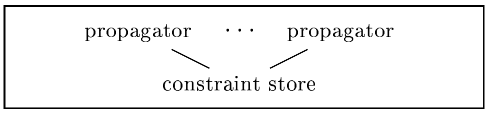
The task of CP(FD) programming systems is to provide two services. The first service is an environment in which these algorithms can interact. For example, after branching, the propagation algorithms corresponding to the constraints have to be revisited in order to achieve a stable constraint store. The necessary book-keeping is handled by the programming system. The second service is to provide libraries of commonly used instances of the algorithms. All practical systems for CP(FD) provides extensive libraries of propagation, branching and (more recently) exploration algorithms.

In this tutorial, we concentrate on CP(FD). For completeness, we mention two other constraint systems that are relevant for many applications; real intervals [BO97] and finite set [Ger97, MM97] constraints. Real interval constraints approximate a real number by an interval $[a, b]$ and provide the usual arithmetic propagators.

A set constraint S over sets of integers approximates a set value which is a finite set of integers s. It approximates a set value by a lower bound s_l ($s_l \subseteq S$) and upper bound s_u ($S \subseteq s_u$). Models based on set constraints often have significant advantages over CP(FD) models. The constraint programming systems Ilog Solver (and thus OPL) and Mozart provide real intervals and set constraints in addition to finite domain constraints.

3 Propagation Algorithms

The constraint store stores information on variables as basic constraints of the form $x \in S$. More complex *non-basic* constraints, as for example $x + y = z$, are represented by propagators, over a set of problem variables called propagator parameters. A propagator observes its parameters and as soon as a value is removed from one of their domains, it tries to remove further values from the domains of its parameters. The algorithm employed in this process is called propagation algorithm. By removing values it may trigger other propagators which in turn may remove value from basic constraints. Eventually, no further values can be removed and propagation stops at a fix-point. Since constraint propagation always removes values from finite domains, the process is guaranteed to terminate.



One run of a propagation algorithm can have three different outcomes:

- It may just remove values from its parameter's domains.
- The propagator may detect that it may never be able to remove any values from any domains in the future, no matter how the parameter domains shrink. In this case, we say the propagator is *entailed* by the constraint store and the propagator can be removed.
- The propagator may find out that it is inconsistent with the constraint store. It terminates and signals failure to the exploration algorithm.

Amount of propagation vs. computational effort. The effort taken by the propagation algorithm in combination with the branching and exploration algorithms is essential for the effectiveness and efficiency of the constraint solver. Consider the constraint $2x = y$ with domains $x \in \{1, \ldots, 5\}$ and $y \in \{0, \ldots, 8\}$. An often sufficient propagation technique is to inspect the bounds of the domains. That would narrow the bounds to $x \in \{1, \ldots, 4\}$ and $y \in \{2, \ldots, 8\}$. A different propagation algorithm considers all values in each domain and removes them, if there is no consistent assignment of the other variables. This technique is called arc-consistent propagation arc-consistent propagation [Mac77] and would result in $x \in \{1, \ldots, 4\}$ and $y \in \{2, 4, 6, 8\}$.

We use the "send+more=money"-problem to illustrate the trade-off between the degree of propagation and efficiency. In this puzzle, different digits need to be assigned to each occurring letter such that the "equation" holds. The following constraints must be satisfied:

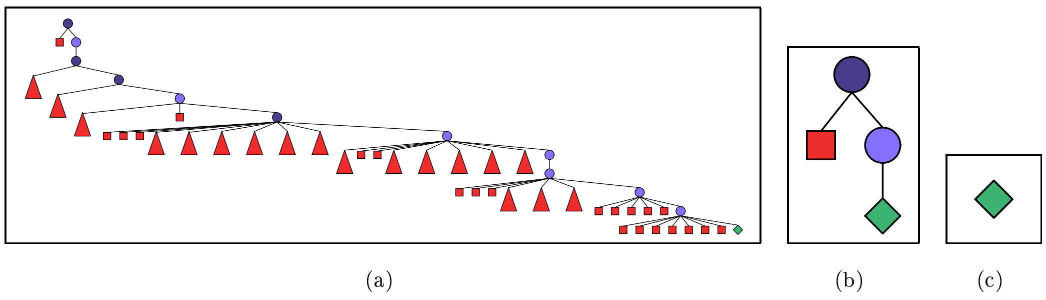


Figure 1: “send+more=money”-search trees for various degrees of propagation. Squares are failure nodes, diamonds solutions and triangles subtrees with no solution.

$$
\begin{array}{lrcrcrcrcr}
\multicolumn{9}{l}{e, n, d, o, r, y \in \{0, \ldots, 9\}} & (1)\\
\multicolumn{9}{l}{s, m \in \{1, \ldots, 9\}} & (2)\\
\multicolumn{9}{l}{\mathit{alldiff}\,(s, e, n, d, m, o, r, y)} & (3)\\
 & & & 1000 \times s & + & 100 \times e & + & 10 \times n & + \;\; d & \\
 & & + & 1000 \times m & + & 100 \times o & + & 10 \times r & + \;\; e & \\
= & 10000 \times m & + & 1000 \times o & + & 100 \times n & + & 10 \times e & + \;\; y & (4)
\end{array}
$$

The constraints (1) and (2) initialize the problem variables with appropriate domains. Note that leading zeros are excluded. The symbolic constraint *alldiff* (3) enforces all problem variables to have pairwise distinct values. The core constraint of this puzzle is constraint (4) imposing the equation *send* + *more* = *money*. We search for the first solution.

Figure 1(a) shows the search tree where the propagation algorithm is just checking inconsistency but not performing filtering. The resulting search is thus generate-and-test search. Using a simple branching algorithms, the resulting search tree has 2488 choice nodes and 22325 failure nodes. At the other extreme, the tree in Figure 1(c) consists just of the solution node, because here, both propagation algorithms of constraint (4) and *alldiff*-constraint (3) implement arc-consistency. An efficient algorithm to achieve arc-consistency for the *alldiff*-constraint is given by [Rég94]. The computational effort of arc-consistent propagation is much higher than for consistency checking but avoids search entirely. Overall, the solution is found significantly faster with arc-consistent propagation. The most efficient search technique for this example, however, employs less powerful propagation algorithms. Here, propagation for constraint (4) reasons over the bounds of the domains and propagation for the *alldiff* constraint (3) removes a value v from the other domains as soon as a parameters domain has become $\{v\}$. The resulting solver, whose search tree is depicted in Figure 1(b), is faster than the other two and strikes the right balance between the computational effort of the propagation algorithms and the cost for traversing the search tree.

Complex symbolic constraints are often crucial for solving difficult problems. If such constraints have many parameters, they are often called global constraints. Resource constraints in scheduling are typically modeled using global constraints. For solving hard scheduling problems, a technique called edge-finding [AC91] is used, which has been integrated in the constraint programming framework in several variants [CL94a, CL96a, CL97, CL94b, BPN95, Wür96]. The basic idea of edge finding is to check whether a certain task t is to be placed before or after a set of other tasks T. In case this can be determined, the domains of the variable denoting the starting time of t can usually be significantly reduced. Constraint programming systems such as CHIP, Ilog Solver/OPL, Claire and Mozart provide libraries with several variants of these global constraints.

4 Branching Algorithms

Constraint propagation usually does not suffice to solve a combinatorial problem. We need to actively try out different alternative possibilities through choice points that generate branching constraints c. The choices of the constraints c at each node determines the size and shape of the tree and thus are crucial for the performance of the solver. Algorithms that generate branching constraints are called branching algorithms.

A popular class of branching algorithms that works well for small problems is variable enumeration. Here c always has the form $x = n$ for some integer n from the current domain of x. In variable enumeration, the degrees of freedom are the choice of variables x to enumerate (variable selection) and the choice of values n to try (value selection). The most naive variable selection is to assume a given fixed ordering and take the first variable in that ordering, which has a non-singleton domain. Other strategies such as taking the variable with the smallest domain (first-fail) or the variable that is parameter of the highest number of constraints often work better.

Apart from enumeration, another generally useful strategy is to successively split the domains of variables. That means for a selected variable x with currently lowest domain element x_l and highest domain element x_h, the branching constraint c has the form $x < x_l + (x_h - x_l)/2$.

Job-shop scheduling problems are solved using a class of branching algorithms called serialization algorithms. Tasks t in the scheduling problem are represented by their duration d_t and finite domain variables x_t denoting their starting time. Serialization algorithms pick successively critical resources according to various criteria, pick two critical tasks t_1 and t_2 that use the resource and generate branching constraints of the form $x_{t_1} + d_{t_1} \leq x_{t_2}$. Serialization techniques were originally developed in Operations Research [CP89], and deployed and extended in the context of constraint programming [BPN95, CL94b, CL94a]. Today, a variety of such serialization algorithms are available in scheduling libraries of CP(FD) systems.

5 Exploration Algorithms

In the previous section, we saw that branching algorithms determine the search tree. It is the job of search algorithms to determine, which part of the search tree is explored and in which order. Exploration algorithms determine the following properties of the solver.

The exploration order determines the order in which the nodes are explored.

The interactivity determines the mode of interaction with other algorithms or the user. An exploration algorithm may return all solutions, compute solutions one-by-one, explore nodes one-by-one, etc.

The pruning behavior of an exploration algorithm may add additional constraints as exploration proceeds.

CP(FD) inherited the most basic exploration strategy, depth-first search, from Prolog. The observation that depth-first search often does not work well together with good branching algorithms led to the development of limited discrepancy search [HG95], where the nodes of the search tree are visited in an order of increasing deviations (discrepancies) from the branching strategy.

The most common implementation technique for exploration algorithms is trailing-based backtracking, also inherited from Prolog. This technique works well for sequential depth-first search and is employed by CHIP, Ilog Solver/OPL and Claire. Schulte [Sch99] showed that copying of constraint stores together with recomputation, which is employed by Mozart [Moz99] and has advantages for exploration algorithms other than sequential depth-first search, can be competitive with backtracking.

In order to achieve combinatorial optimization as in scheduling, exploration algorithms can be extended by a pruning behavior. Whenever a solution is encountered, the exploration algorithm generates an additional constraint that expresses that further solutions should be better with respect to optimization criteria than the solution found. This technique is called constraint-based branch-and-bound and can be seen as a generalization of branch-and-bound used in integer optimization. Branch-and-bound ensures that solutions are found in increasing quality. If the constraints that encode the quality of the solution are strong and if solutions of high quality are found early in the search, the tree can be pruned significantly. In job-shop scheduling, the model includes a variable f representing the overall duration of the schedule. After finding a solution with overall duration d, the additional constraint $f < d$ is introduced, which often—via interaction with propagation algorithms—leads to pruning of the search tree.

Another variant of the exploration algorithms is exemplified by the Oz Explorer [Sch97], which—in addition to depth-first search (with branch-and-bound)—visualizes the resulting search tree and allows interactive exploration, which is useful during the development and performance-tuning of constraint programs.

6 Modeling Techniques

This section presents techniques to model over-constraint problems and to improve search by improving the constraint model.

Handling over-constrained problems. Some problems contain conflicting constraints, and the task is find to find a solution that meets a maximal number of constraints. The concept of *reified* constraints allows handling of over-constrained problems by "soften" these constraints. The idea is to connect a constraint c with 0/1-variable B, i.e., $c(x_1, \dots, x_n) \leftrightarrow b$ and to reflect its validity into b. As long as $b \in \{0, 1\}$ the constraint does not remove any value from the domains of its parameters $x_1, \dots, x_n$. In case the c is entailed by the constraint store, $b = 1$. The 0/1-variable becomes 0 if the c is inconsistent with the store. On the other hand, if b is constrained to 1 (0) then c ($/negc$) is added to the computation space and removes values from the domains of its parameters. Usually the 0/1-variables of an over-constraint problem contribute a objective function which is then minimized or maximize using branch-and-bound search (see Section 5).

Excluding symmetries. Avoiding symmetries is essential for searching optimal solutions. This can be achieved by simply imposing an order on the solutions and thus significantly pruning the search tree. We demonstrate the benefit of this technique using the photo-alignment problem. A photo is to be taken of a group of people. Everybody has a preference whom she wants to stand next to. Not all of these preferences can be met, since they contradict each other, i.e., the problem is over-constrained. The objective is to meet as many preferences as possible by maximizing the number of fulfilled preferences using branch-and-bound search (see Section 5).

We model that two persons stand next to each other by stating that their distance is 1 and reify this constraint to be able to use it in the objective function of branch-and-bound search: $\forall$ two distinct persons p_i and p_j : $(|a_{p_i} - a_{p_j}| = 1) \leftrightarrow r_k$. The variable a_{p_i} is a finite domain variable and denotes the position of the person in the line. The objective function is $max\ \Sigma r_k$. Solving the problem for 7 persons takes 313 choice points and 313 failures to find the optimal solution. Adding a symmetry-breaking order constraint on two arbitrary person's positions $a_{p_1} < a_{p_2}$ reduces the search tree to 219 choice points and 219 failures.

Redundant constraints. Another technique to improve constraint propagation is to add (semantically) redundant constraints, i.e., these constraints are actually implied by the already present constraints but they add extra pruning since they, e.g., use a different filtering algorithms. We demonstrate this technique by the example of finding a so-called magic sequence $(x_0, \dots, x_n)$ of n elements. The "magic" is that x_i determines how often i occurs in the sequence. A solution for $n = 3$ is the sequence $(1, 2, 1, 0)$.

The model of the problem represents every element of the sequence $s = (x_0, \dots, x_n)$ by a finite domain variable with an initial domain $x_{i \in \{0,\dots,n\}} \in \{0, \dots, n+1\}$. We state for each position i that there are exactly x_i elements i in the sequence s: $\forall i \in \{0, \dots, n\} : exactly(s[i], s, i)$ where $s[i]$ denotes the ith element of s. Search uses the first-fail branching algorithm (see Section 4). Finding the first solution takes 164 choice points and 160 failures. We can improve the model by adding more constraints, e.g., it is straightforward to see that the sum of all elements of the sequence is $n + 1$: $\Sigma_{i=0}^{n} s[i] = n$ (5). Adding this constraint reduces the number of choice points to 29 and the number of failures to 25. But we can do better by adding the constraint $\Sigma_{i=1}^{n} (i - 1) \times x_i = 0$ (6). This constraint is perhaps not as straightforward as the first one: it is easy to see that $\Sigma_{i=0}^{n} i \times x_i = n$. The second redundant constraint (6) equates this sum with constraint (5). This leads to a further reduction of the size of the search tree to 9 choice points and 6 failures. Note that the pruning of the search tree due to redundant constraints has to outweigh the computational effort for the extra constraints as it happens in the magic sequence example.

7 Conclusion

We introduced finite domain constraint programming as a software framework for combining propagation, branching and exploration algorithms. The integration of algorithms and techniques from Operational Reseach and Artificial Intelligence allow the solving of hard combinatorial search problems. We hightlighted the importance of global symbolic constraints and application-specific branching algorithms. Systems that provide extensive support for finite domain constraint programming include CHIP [DVS+88], Ilog Solver/OPL [ILO97, Hen99], Claire [CL96b] and Mozart [Moz99]. Areas of application where constraint programming has been shown to be superior to Operational Research techniques include job-shop scheduling [CL94a, CL96a, CL97, CL94b, BPN95, Wür96] and sport scheduling [Hen00]. The success of constraint programming relies on the following properties of these applications:

- fruitful interaction of propagation and branching algorithms,
- existence of efficient and powerful propagation algorithms for symbolic constraints,
- tightness of the constraints, allowing for substantial pruning of the search tree.

In situations where search tree cannot be pruned effectively, for example in typical time tabling applications, constraint programming can still be used to guide heuristic incomplete tree search techniques [HW96].

References

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[BO97] Frederic Benhamou and William Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 32(1), 1997.

[BPN95] P. Baptiste, C. Le Pape, , and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *Proceedings of the First International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[CL94a] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the International Conference on Logic Programming*, pages 369–383, 1994.

[CL94b] Yves Caseau and François Laburthe. Improved clp scheduling with task intervals. In *Proceedings of the In International Conference on Logic Programming*, pages 369–383, 1994.

[CL96a] Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Joint International Conference and Symposium on Logic Programming*, 1996.

[CL96b] Yves Caseau and François Laburthe. CLAIRE: Combining objects and rules for problem solving. In *Proceedings of the JICSLP'96 workshop on multi-paradigm logic programming.* TU Berlin, 1996.

[CL97] Yves Caseau and Fran cois Laburthe. Solving various weighted matching problems with constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 17–31, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[DVS$^+$88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. The constraint logic programming language CHIP. In *Proceedings International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, December 1988. Springer-Verlag.

[Ger97] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[Hen99] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, MA, 1999.

[Hen00] Martin Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 2000. to appear.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers, San Mateo, CA.

[HW96] Martin Henz and Jörg Würtz. Using Oz for college time tabling. In E.K.Burke and P.Ross, editors, *The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Time Tabling, Edinburgh 1995*, Lecture Notes in Computer Science 1153, pages 162–177. Springer-Verlag, Berlin, 1996.

[ILO97] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 4.0, Reference Manual*, 1997.

[JM94] Joxan Jaffar and Michael Maher. Constraint logic programming—a survey. *Journal of Logic Programming*, 19/20:503–582, 1994.

[Mac77] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[MM97] T. Müller and M. Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 17–19 September 1997.

[Moz99] Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 1999.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints.* The MIT Press, Cambridge, MA, 1998.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence.* AAAI Press, 1994.

[Sch97] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press, Cambridge, MA.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 275–289, Las Cruces, New Mexico, August 1999. The MIT Press, Cambridge, MA.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1&2):139–168, 1996.

[Wür96] Jörg Würtz. Oz Scheduler: A workbench for scheduling problems. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 132–139, Toulouse, France, November16–19 1996. IEEE Computer Society Press.