The constraint logic programming paradigm: declarativity, efficiency, and flexibility

U. Geske, H-J. Goltz, U. John, A. Wolf
GMD Berlin
Rudower Ch. 5, D-12489 Berlin, Germany
Phone: +49 30 6392 1862, Fax: +49 30 6392 1805
E-mail: {geske,goltz,john,armin}@first.gmd.de

Abstract

Real-world applications of theories for planning, scheduling, and design, call for more powerful and expressive theories than those provided by "classical" methods, which are based on a number of simplifying assumptions. Basic research on theories of constraint solving has made marked progress over the past few years. This paper sets out to show that the declarative nature of this new paradigm has significant advantages in software-engineering terms in the programming and maintenance phases. Moreover, the high efficiency of these methods allows the simulation of different strategies and supplies near-optimal solutions within a short time. However, complex problems require the application of heuristics. Research results in this area are presented.

Keywords

Constraint logic programming, optimization, planning, configuration

1 INTRODUCTION

Traditionally, the modeling and processing of combinatorial problems, like job-shop scheduling, material requirements planning, configuration, and time-tabling have been done by classical operations research methods. A number of algorithms have been developed to deal with problems of this kind. These include linear programming, branch-and-bound, constraint satisfaction, hill climbing, simulated annealing, tabu search, genetic algorithms, connectionist systems, and expert systems (see also [BT95]). The problem when using such algorithms to find solutions for industrial planning and configuration tasks, is, that they have an exponential complexity which normally prevents the fast generation of exact solutions.

Systems Implementation 2000 R. Nigel Horspool (Ed.)
© 1998 IFIP. Published by Chapman & Hall

New paradigms, such as *Constraint Logic Programming* (CLP) and *Hierarchical Constraint Logic Programming* (HCLP), have significant advantages in software-technological terms, in the programming and maintenance phases of classical methods. Moreover, the high efficiency of these methods allows the simulation of different strategies and supplies near-optimal solutions (i.e. solutions deviating no more than approx. 5% from the optimal solution), within a short time. Roughly speaking, (H)CLP is a combination of constraint satisfaction (CSP) and logic programming [Hen89, JL87, Jou94, MR95, Pod95, Tsa93]. While inheriting a strong theoretical background from logic programming, (H)CLP offers a declarative programming style, that is expressive and flexible enough for problem specification and heuristics programming, allowing rapid program development for complex problems and enabling programs to be easily modified and extended. This programming style is often considered a major contribution to software engineering, since it enables programs to be written with clear and transparent semantics, which is what distinguishes logic-based systems from mainstream programming languages.

An important extension of CLP is HCLP, which allows us to express soft constraints, whose satisfiability is uncertain and is controlled by error functions. Soft constraints are very useful for solving over-constrained problems and for modeling planning/design objectives.

The advantages and disadvantages of the above-mentioned methods are discussed in [BT95]. As show in this overview, constraint-satisfaction methods (CSP) are characterized by efficient processing if there are a large number of constraints. Moreover, CSP allows a problem-oriented problem description, dispensing with artificial parameters that have to be adjusted. Since it is also valid for other methods, the CSP method's order of execution is hard to understand, but it inspires greater confidence by confining attention to the actual defining quantities of the problem.

The CSP method consists essentially of two phases. In the first phase, the constraint system is simplified (constraint propagation and relaxation); in the second phase, backtrackable search is performed (labeling) to find a solution in the remaining search space. Constraint Logic Programming (CLP) uses the properties of logic programming for search. The CLP paradigm combines efficient methods known from Operations Research (OR) with flexible methods of logic programming for search and rule processing. The rules are suitable for formulating heuristics. In the context of CLP, heuristics are criteria that define additional constraints and the order in which they are considered during the process of problem solving. Their effect may be to find a better solution within a shorter execution time.

The CLP techniques may provide a necessary link between the declarative description of constraints, which arise in scheduling, planning, and configuration domains, and the demand for obtaining efficient usable programs.

2 CONSTRAINT LOGIC PROGRAMMING

Solver

Constraint Logic Programming (CLP) is an extension of the logic programming paradigm incorporating the efficient processing of constraints, which may be complex sets of equations or inequations. The processing of such constraints is performed by a constraint solver which combines methods from Operations Research and Artificial Intelligence.

Inherited from classical CSP methods, the CLP($\Re$) framework uses elements of the (infinite) domain of real numbers as possible values for variables. It uses the simplex method to solve systems of linear inequations, a well-known algorithm in OR. Where it is known that only a restricted set of integer values can be assigned to variables — as in many combinatorial problems — the more elaborate CLP(FD) framework can be applied. In this framework, (finite) domains — rather than single values — are assigned to variables. The goal of processing in CLP(FD) is to reduce the domains to single values. During this reduction process, a domain of a variable may be empty: this indicates an inconsistency (see also Figure 1).

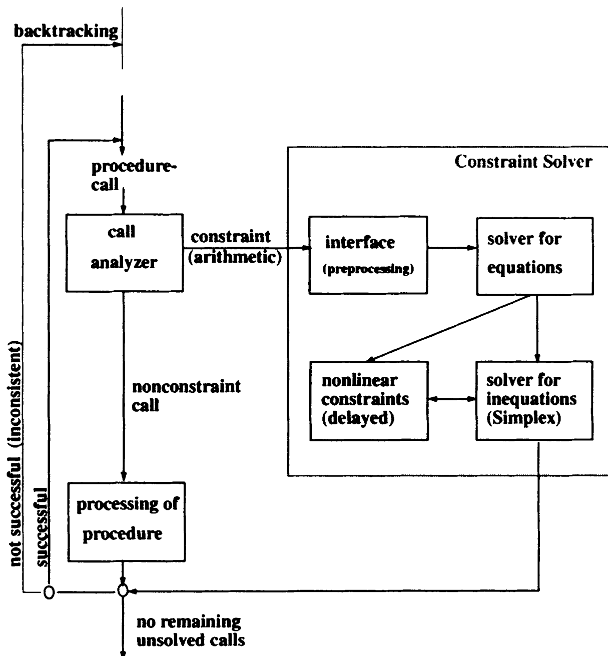

Figure 1 Execution scheme of constraint logic programs

One advantage of constraint-based methods is that, on the one hand, a consistent value assigned to a variable results in a search space reduction,

and on the other, an inconsistent value for a variable is immediately rejected. This behavior is achieved by arranging the conditions (constraints: equalities and inequalities) in a program in front of all other statements, even in cases whre the conditions are not computable because of uninstantiated variables. In this situation, within classical programming paradigms an error will occur during execution of such a program. Within the constraint logic programming paradigm, error-free processing is ensured by extending the classical call mechanism (e.g. "call by value" or, in Prolog, "unification"). The processing of constraints that are not computable is postponed. Constraints that are computable are processed immediately, and the effects on the postponed constraints will be investigated. If some inconsistency occurs, backtracking is initiated to find other values for variables. This procedure would appear too costly, but it is the reason for the significant increase in gain of efficiency as compared with other paradigms. The result of a constraint logic program may be a set of not further simplifiable constraints.

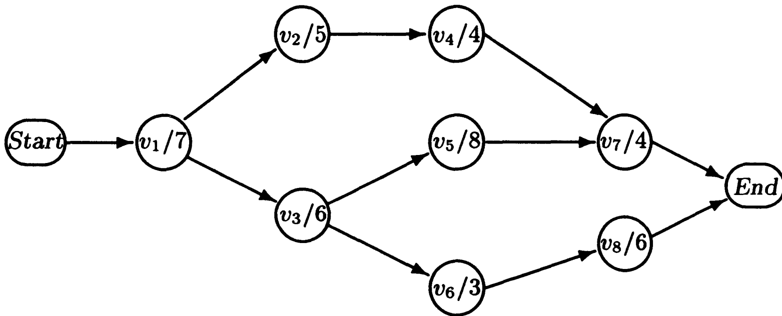

Figure 2 Example of a network plan

```
V1 + 7 =< V2,    V1 + 7 =< V3,    V2 + 5 =< V4,    V3 + 6 =< V5,
V3 + 6 =< V6,    V4 + 4 =< V7,    V5 + 8 =< V7,    V6 + 3 =< V8,
V7 + 4 =< End,   V8 + 6 =< End
```

Figure 3 Program for the network plan in Figure 2

The declarativity of the constraint logic methods can be nicely demonstrated by the network planning technique, which is well known in job-shop scheduling. An example of a network plan is given in Figure 2. The constraint logic program that computes this example consists essentially of the constraints shown in Figure 3.

Search: backtrackable domain reduction

The way of constraint solving results in large reductions of the search space. However, the remaining search space may be exponentially complex. Various heuristics, like backtrackable domain reduction, generation of redundant constraints, layered backtracking, and constraint hierarchies with a non-trivial error function ([Gol95, Wol97]), were investigated in order to restrict search to the meaningful parts of such search spaces. The basic algorithm for this search procedure, called 'labeling', is: select a variable, choose a value from the domain of this variable, and assign this value to the variable. We have developed a generalization of the labeling method: the assignment of a value to the selected variable is replaced by reduction of the domain of this variable. If backtracking occurs, the unused part of the domain is taken as the new domain for repeated application of this method. The effect of the domain reduction technique occurs if the size of the reduced domain differs significantly from the original domain and from a one-value domain — a reduced domain should be neither too large nor too small. Thus, the domain of a variable is reduced in such a way that the number of elements belonging to the reduced domain is not less than a given lower limit `Min`, and the number of elements of the reduced domain is not greater than an upper limit `Max`. As a heuristic we use for efficient computation the difference between the maximum and minimum value of a domain instead of the number of elements. In the case of backtracking, the domain reduction algorithm for a variable applies for the set difference of the former domain and the reduced domain of the corresponding variable.

The solution is narrowed by this reduction algorithm, however, it does normally not generate a solution immediately. Thus, after domain reduction, the assignment of variables has to be performed which may also include search. The main part of the search, however, should be carried out by domain reduction.

For 10×10-problems from the class of the above mentioned N×N problems, solutions can be computed quickly (execution time is about 500 ms for a SPARC20) applying CLP(*FD*) and backtrackable domain reduction, which deviate 3%—5% from the optimal solution. These values are in strong contrast to 30%-50% deviations obtained with commercially tools available.

3 CONSTRAINT HIERARCHIES

Principles

In practical applications, we are often confronted with over-constrained problem specifications that are unsolvable. However, a solution in the form of a layout, plan, schedule, etc. has to be generated. There are different theoretical approaches for specifying and solving over-constrained problems using nonstandard constraint processing techniques [HR96]. What the developed theories have in common is a kind of "softness" or "fuzziness" of constraints

and a measure for these properties. Our approach solving this conflict is to introduce Constraint Hierarchies [WB93]. These consist of required constraints, which have to be satisfied, and non-required constraints, which have only to be satisfied as well as possible.

A constraint hierarchy H consists of a finite sequence of hierarchy levels $H_0, H_1, ..., H_n$ (see also Figure 4). Each hierarchy level is a finite sequence of constraints $c_1, ..., c_k$. All constraints at level 0 are required constraints. They have to be satisfied. H_1 is the sequence of the strongest non-required constraints. After considering the constraints at level 0 the non-required constraints at level 1 have to be satisfied as well as possible. Then the non-required constraints at level 2 are considered, and so on.

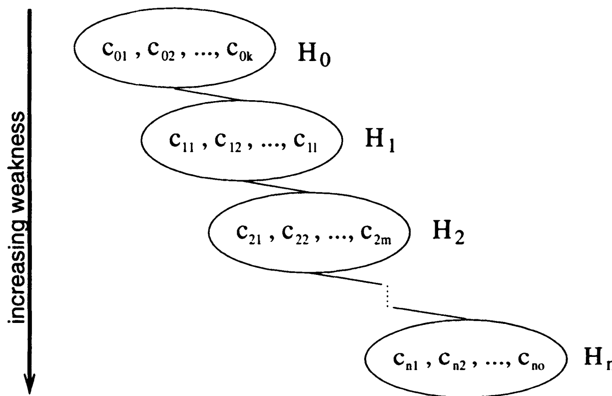

Figure 4 Constraint hierarchy

Error processing

Weak constraints can be used to formulate planning strategies and optimality criteria. They may also be used to express conditions reflecting wishes, which are considered even in cases where they contradict other conditions. Error processing must be used to obtain a solution. For this purpose, each constraint of a hierarchy is connected to an error function, which assigns an error value to a potential solution. The error value equals 0 if the constraint is satisfied. Examples of error functions are:

- The trivial error function: the error is 1 if the constraint is not satisfied.
- A nontrivial error function: the error is the absolute of the deviation.

A *comparator* defines a relation of all error values of all hierarchy levels. Various definitions are given in Table 1. The kind of this function is essential for acceptance of a solution, as the following example demonstrates.

The capacity needed for an action is to be distributed over, for example, two agents P and Q at a ratio of $3 : 7$. Additionally, there are two con-

Comparator	weighted-sum-metric-better	worst-case-metric-better	least-squares-metric-better
Description	Overall error of a hierarchy level equals the sum of single errors of constraints	Overall error of the hierarchy level equals the maximum single error	Overall error equals the sum of the quadratic single errors
Advantage	Fair solution in case of inequalities	Fair error minimization	Fair solution in the case of inequalities **and** fair error minimization

Table 1 Classification of comparators

straints (with the same priority) on the capacity for the action, one being 14, the other 18 units. This problem, which is inconsistent and unsolvable with hard constraints only, can be solved as a constraint-hierarchical problem. Both constraints $3P+7Q = 14$ and $3P+7Q = 18$ are members of the same (soft) hierarchy level. If the comparator *weighted-sum-metric-better* (Table 1) is used, there are two solutions that minimize the overall error: $P = 1$, $Q = 2$ and $P = 3$, and $Q = 1$. The overall error computes to $3 + 1 = 2 + 2 = 4$ for the corresponding solutions. Using the comparator *worse-case-metric-better*, the only and fairer solution is $P = 3$ and $Q = 1$. The overall error computes to $2 + 2 = 4$.
Our approach rewriting such programs with inconsistent constraint systems (Figure 5(left)) in the form of programs with constraint hierarchies is presented in Fig. 5 (right). Very often in real-life planning and design situations,

```
capacity(X, Y, Z) :-
   [X,Y,Z] :: 0..20,
   3*P+7*Q #= 14,
   3*P+7*Q #= 18.
```

```
capacity(X, Y, Z) :-
   [X,Y,Z] :: 0..20,
   hierarchy([ c(1, 3*P+7*Q #= 14),
               c(1, 3*P+7*Q #= 18)
             ]).
```

Figure 5 Program with inconsistent constraints (left) and the same program using constraint hierarchies (right).

the initial problem changes in such a manner that replanning or redesign is needed. Only in cases where constraints are deleted or additional constraints are consistent with the given solution is the previously evaluated plan still valid. In all other cases, especially if constraints are replaced by new ones, replanning can be performed only by starting the planning task from scratch. In these cases, the new solution should be as close as possible to the old solution. A modeling strategy based on constraint hierarchies [Wol96] guarantees this kind of solution stability.

Summarizing, constraint logic programming and hierarchical constraint logic programming, applied to real-world planning, provide an adequate theoretical framework for modeling practical planning and design problems declaratively and solving them efficiently.

4 APPLICATIONS

4.1 Flexible configuration of technical devices

Figure 6 Configuration space and solution for configuration of a technical device

Often when configuring products from components, it happens that certain parts of a derived solution have to be changed, while the remaining parts are to remain unchanged. Unfortunately, the required changes generally have global effects producing changes elsewhere in the product's configuration. The goal of reconfiguration is therefore to keep other parts of the configuration 'as stable as possible', i.e. for an error function minimizing unavoidable changes. Without this goal there would be a choice between either deriving a completely new configuration (allowing any changes) or getting an error message and no configuration (prohibiting any changes). The technique for deriving configurations that are 'as stable as possible' is to use 'soft' constraints.

Figure 6 shows the initial variety of configurations (configuration space)

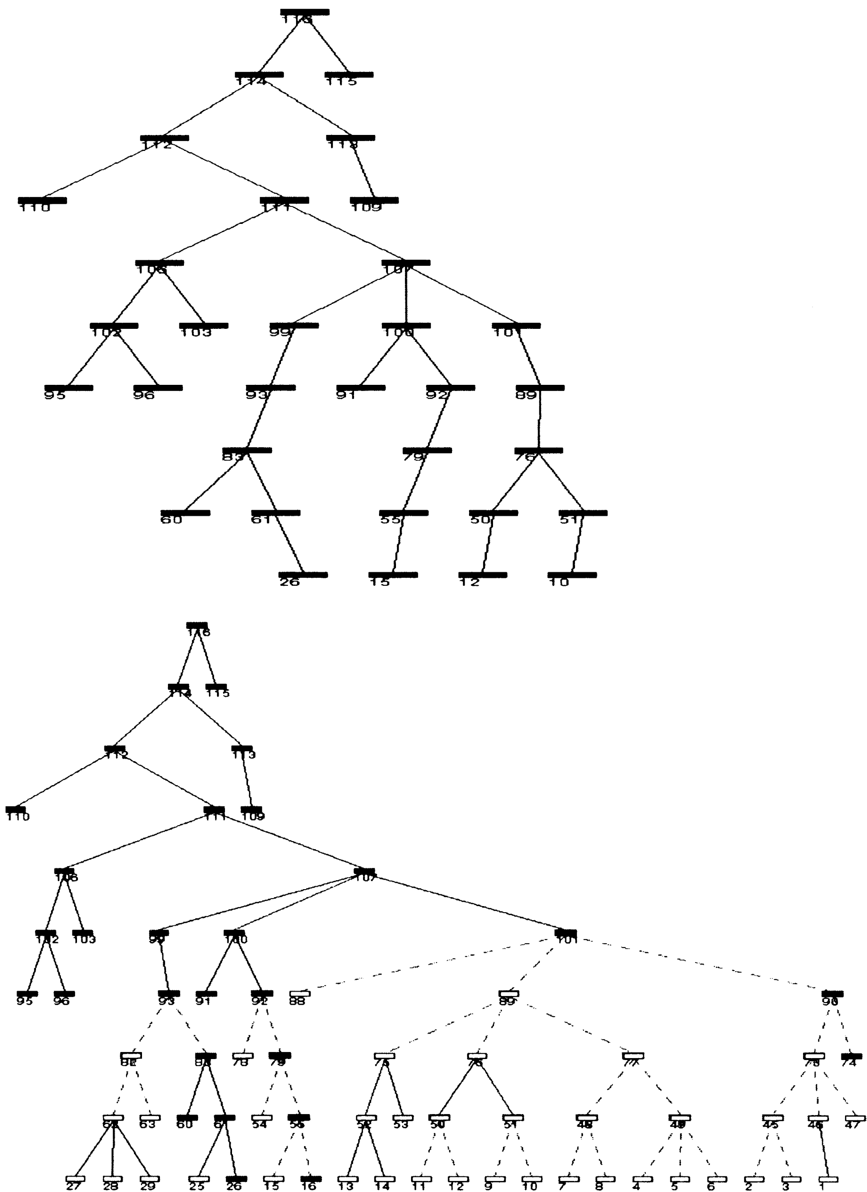

Figure 7 Reconfiguration by using soft constraints(top) and by using hard constraints (bottom)

of a certain technical device. It results from the specification of the components and their dependencies and customer requirements concerning the properties of a special version of such a device. The boxes represent basic

components (leaves), compositions of components (boxes connected with several other boxes), and the final device (top of the tree). The box number is a code for the name of the corresponding component. Different 'colored' edges denote alternative (dotted), indispensable (black), and optional (blue in the original) components. Black boxes in Figure 6 belong to a derived configuration for user requirements (not given here). This configuration may be a solution that is partly accepted by the user, but in addition to the original specification of the device, component 108 should be replaced by component 109. The exchange should not influence the structure of the derived configuration and their components if this can be avoided. Figure 7 shows the result of this exchange using soft constraint. The configuration is similar to the former one, except that component 94 is replaced by component 93. This substitution results however, from the specification of the device, there being a rule stating that 'if component 109 is used, component 93 has to be used, too'.
The result of using hard constraints for reconfiguration is shown in Figure 7 (bottom). The replacement of component 94 by component 93 is performed also, but other components are replaced, too. The structure and types of components used do not correspond to the desired configuration.

4.2 Efficient optimization of workflow processes

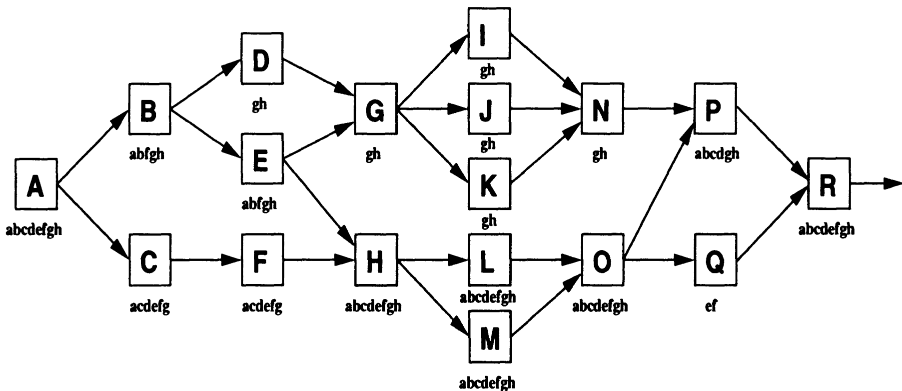

Figure 8 Workflow scenario

One of the tasks of workflow management is to find a distribution of actions to agents which ensures that the constraints describing the goals of a company are satisfied. Such constraints may be the fast processing of actions or comparable stress of the agents, or both. The method used to solve this problem with the additional optimality criteria to ensure that the agent's (e.g. an office

or a workshop) workload is below their maximum capacity, is demonstrated by the example [Bau97], shown in Figure 8.

Capital letter at the boxes denote agents (persons, desks, offices); small letters next to the boxes denote the types of actions (processes) that can be executed there. The distribution of actions must take into account the status of buffers in the current execution. It is assumed that each agent has a different buffer for each action. The content of the buffers is described by positive/negative numbers.

CLP($\Re$)		CLP(*FD*)		
without early projection	with early projection	without DR, without RC	with DR	with DR and RC
out of memory	220 ms	$> 24h$	150 ms	50 ms

Table 2 Efficiency of workflow planning using different techniques in CLP($\Re$) and CLP(*FD*) (DR = backtrackable domain reduction, RC = redundant constraints)

The efficiency in finding a solution with CLP(FD) is derived essentially from the use of the backtrackable domain reduction technique. If the technique of redundant constraints is applied additionally, the efficiency can be further improved (see also Table 2). Moreover, the presented results demonstrate the usefulness of our early projection technique [FGN96].

5 RESULTS AND FURTHER RESEARCH

We have discussed the advantages of the constraint logic programming paradigm for the software engineering task of programming combinatorial problems and presented research results on extensions of this paradigm. These include modeling the problem using a problem-specific description; application of the method (in contrast to other methods) if the number of constraints is large; considerable flexibility in changing the problem description (adding or deleting constraints) or in deriving adequate solutions; fast program execution through the use of Operations Research methods; easy integration of special search strategies and heuristics (rules) like backtrackable domain reduction, or investigations of different comparators; facilities for using constraint hierarchies and early projections; simulation of different situations using the integrated backtracking facility.

Further research is planned to extend and combine recently developed theoretical results in (hierarchical) constraint logic programming, including partially ordered constraint hierarchies, different nontrivial error functions, composite constraints, incremental solution methods, and program analysis and

transformation methods. Moreover, solving real-life problems, efficient implementations are needed that make allowance for dynamically changing environments and yield usable solutions within a reasonable time.

REFERENCES

[BT95] BT Technl. J. Vol. 13, No. 1, Jan. 1995

[Bau97] H. Baumgärtel. *Constraint-based multi-criterial optimization for flow production planning.* Proc. IMACS'97. Aug. 1997.

[FGN96] A. Fordan, U. Geske, A. Nareyek. Optimizing Constraint-Intensive Problems Using Early Projection., *Proc. JICSLP'96*, The MIT Press, 1996.

[Gol95] H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. Proc. CP'95, Springer LNCS 976, pp. 549 - 562, 1995.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge (Mass.), London, 1989.

[HR96] Walter Hower and Zsófia Ruttkay, editors. *Non-Standard Constraint Processing*, ECAI-96 workshop W27, Budapest, 1996.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. 14th Principles of Programming Languages*, 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez (ed.), *Proc. 4th Int. Conf. Logic Programming*, pp. 196–218. MIT Press, 1987.

[Jou94] J.-P. Jouannaud, editor. *Constraints in Computational Logics*, Springer-Verlag, Volume 845 of *LNCS*, 1994.

[MR95] U. Montanari and F. Rossi, editors. *Principles and Practice of Constraint Programming - CP'95*, Springer-Verlag, Volume 976 of *LNCS*. 1995.

[Pod95] A. Podelski, editor. *Constraint Programming: Basics and Trends*, Springer-Verlag, Volume 910 of *LNCS*, 1995.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Wal96] M. Wallace. Practical applications of contraint programming. *Constraints, An International Journal*, 1:139–168, 1996.

[WB93] M.A. Wilson and A. Borning. Hierarchical constraint logic programming. *The Journal of Logic Programming*, 16(3 & 4) 1993.

[Wol96] A. Wolf. Transforming ordered constraint hierarchies into ordinary constraint systems. In *LNCS*, Vol. 1106, 1996.

[Wol97] A. Wolf. Solving hierarchies of finite-domain constraints. *Journal on Experimental and Theoretical Artificial Intelligence, Special issue on non-standard constraint processing*, 1997.

6 BIOGRAPHIES

Ulrich Geske is a professor and chair of the Department of Planning Techniques and Declarative Programming (*http://www.first.gmd.de/concorde/plan*) at the Research Institute for Computer Architecture and Software Technology (FIRST) in GMD — German National Research Center for Information Technology.

Hans-Joachim Goltz is senior researcher in the Constraint-based Planning group in GMD-FIRST and is group supervisor of a timetabling project.

Ulrich John is a scientist at GMD-FIRST. He works in the fields of constraint-based configuration, knowledge-based systems, and distributed AI.

Armin Wolf is project group supervisor of the Constraint-based Configuration group at GMD-FIRST and performs research in extensions of constrained-based methods.