INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians
Universität
München

University Course Timetabling using Constraint Handling Rules

Slim Abdennadher and Michael Marte

Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-2000-2, Januar 2000

University Course Timetabling Using Constraint Handling Rules

Slim Abdennadher, Michael Marte
Computer Science Department, University of Munich
Oettingenstr. 67, 80538 Munich, Germany
{Slim.Abdennadher, Michael.Marte}@informatik.uni-muenchen.de

Abstract

Timetabling the courses offered at the Computer Science Department of the University of Munich requires the processing of hard and soft constraints. Hard constraints are conditions that must be satisfied, soft constraints, however, may be violated, but should be satisfied as much as possible. This paper shows how to model our timetabling problem as a partial constraint satisfaction problem and gives a concise finite domain solver implemented with Constraint Handling Rules that, by performing soft constraint propagation, allows for making soft constraints an active part of the problem solving process. Furthermore, we improve efficiency by reusing parts of the timetable of the previous year. Our prototype needs only a few minutes to create a timetable while manual timetabling usually takes a few days. It was presented at the Systems'98 computer fair in Munich and several universities have enquired for it.

1 Introduction

University course timetabling problems are combinatorial problems which consist in scheduling a set of courses within a given number of rooms and time periods. Solving a real world timetabling problem manually often requires a significant amount of time, sometimes several days or even weeks. Therefore, a lot of research has been invested in order to provide automated support for human timetablers. Contributions come from the fields of operations research (e.g. graph colouring, network flow techniques) and artificial intelligence (e.g. simulated annealing, tabu search, genetic algorithms, constraint satisfaction) [Sch95]. This paper refers to terms and methods from constraint satisfaction [Mac92, Kum92]. The methods presented were developed using Constraint Logic Programming (CLP) [JM94, FA97, MS98]. CLP combines the declarativity of logic programming with the efficiency of methods from operations research and artificial intelligence. It has recently become a promising approach for solving timetabling problems.

Applying classical methods from constraint satisfaction requires to model the problem as a *constraint satisfaction problem* (CSP), i.e. a set of variables (representing the points in time courses must begin, for example), each associated with a domain of values it can take on, and a set of constraints among the variables. Constraints are relations which specify the space of solutions by forbidding combinations of values.

Methods include search, heuristics and constraint propagation. Typically, systematic search (e.g. chronological backtracking) assigns values to variables sequentially following some search order. If the procedure fails to extend a partial solution, decisions are undone and alternatives explored. Systematic search often relies on heuristics which define the order in which variables and values are chosen. Constraint propagation is complementary; it simplifies a problem by identifying values which cannot participate in a solution. This way the search space gets pruned and search becomes easier.

The classical CSP framework is of particular interest because many problems from design, resource allocation and decision support (among others) can be cast as CSPs naturally [JM94, Wal96]. However, it is not sufficiently expressive for the application under study here. In particular, it does not allow for a distinction between *hard constraints*, which are mandatory, and *soft constraints*, which should get satisfied but may get violated in case this is unavoidable. This limitation forces to treat soft constraints as if they were hard, which frequently leads to over-constrained CSPs without solutions.

Several CSP based frameworks have been introduced which facilitate the formal treatment of soft constraints. For example, hierarchical constraint logic programming [BFBW92] allows for constraint hierarchies (a constraint on some level is more important than any set of constraints from lower levels but constraints of the same level are equally important) while in partial constraint satisfaction [FW92] each constraint is associated with the cost of its violation; see [BMR97] for a more powerful framework, which subsumes other frameworks.

In practice, most constraint-based timetabling systems either do not support soft constraints [AB94] or use a branch & bound search instead of chronological backtracking [HW95, FHS95]. Branch & bound starts out from a solution and requires the next solution to be better. Quality is measured by a suitable cost function that depends on the set of violated soft constraints. With this approach, however, soft constraints play no role in selecting variables and values, i.e. they do not guide search.

Another approach is to adopt techniques developed to propagate hard constraints; soft constraint propagation is intended to associate values with an estimate of how selecting a value will influence solution quality, i.e. which value is known (or expected) to violate soft constraints, or the other way round, which value is known (or expected to) satisfy soft constraints. By considering estimates in value selection, one hopes that the first solution will satisfy a lot of soft constraints. For example, [Mey97] presents a commercial C++ library providing black-box constraint solvers and search methods for the nurse scheduling problem.

Since the black-box approach makes it hard to modify a solver or build a solver over a new domain, our aim was to implement a solver for our timetabling problem using the "glass-box" approach Constraint Handling Rules (CHR) [Frü95, Frü98]. CHR is a powerful special-purpose declarative programming language for writing application-oriented constraint solvers either from scratch or by modifying existing solvers. Inspired by an existing finite domain solver written in CHR we developed a solver which performs hard and soft constraint propagation. The core of the solver takes no more than 20 lines of code. Furthermore, our system, IfIPlan[1], brought down the time necessary for creating a timetable from a few days by hand to a few minutes on a computer.

In this paper we describe the main features of the constraint solver that was used to generate a timetable for the Computer Science Department of the University of Munich. Section 2 introduces our timetabling problem and the constraints that a solution of the problem had to satisfy. Section 3 shows how the problem can be modelled as a partial constraint satisfaction problem. Section 4 gives an overview of the implementation. We conclude with a summary. This paper is an extended and substantially revised version of [AM98].

2 The Timetabling Problem

2.1 The Process of Timetabling

The Computer Science Department at the University of Munich offers a five year program for a master degree in computer science consisting of undergraduate studies (two years) and graduate studies (three years). The problem of timetabling is to be solved every term on base of the timetable of the previous year, the teachers' personal preferences and a given set of courses, each associated with its teachers. The overall process of manual timetabling runs as follows.

After collecting wishes of teachers and information on new courses, a first proposal is developed with the timetable of the previous year as a starting point. This is done by using free slots in the timetable left by courses not taking place again for new courses offered by the same people, whereas wishes of teachers take precedence over the timetable of the previous year. After handing out the proposal to all teachers, evaluations and new wishes are collected.

With the current proposal as a starting point, a next proposal is developed incorporating the responses on the current proposal, again changing as little as possible, and so on. Creating a new timetable is thus a multi-stage, incremental process. Relying on the timetable of the previous year and changing as little as possible by incremental scheduling drastically reduces the amount of work necessary for creating a new timetable and ensures acceptance of the new timetable by keeping the weekly course of events people are accustomed to.

[1] IfIPlan is an acronym for the German „Planer für das Institut für Informatik".

Note that the assignment of rooms is done elsewhere. Nevertheless conflicting requirements for space or certain equipment may be a cause for changing the timetable.

2.2 Constraints

The general constraints are due to physical laws, academic reasons and personal preferences of teachers:

- A teacher cannot be at two places the same time, so avoid clashing the courses of a teacher. There should be at least a one hour break between two courses of a teacher.

- Some teachers prefer certain times or days for teaching.

- Monday afternoon is reserved for professors' meetings: Do not schedule professors' courses for Monday afternoon.

- The department consists of five units, each dedicated to a certain area of research. Most courses are held by members of a single unit while only a few courses are held by members of different units. Courses held by members of a certain unit must not clash with courses held by other members of the same unit.

- An offering typically consists of two lectures and a tutorial per week. There should be a day break between the lectures of an offering. The tutorial should not take place on a day, on which a lecture of the same offering takes place. All courses should be scheduled between 9am and 6pm. No lectures should be scheduled for Friday afternoon. No tutorials should be scheduled for late Friday afternoon.

- Only few of the courses are mandatory for and dedicated to students of a certain term while most courses are optional and open to all students. For each term of the undergraduate studies there is a set of mandatory courses, the attendance of which is highly recommended. Courses of the graduate studies only rely on the knowledge provided by courses of the undergraduate studies. There is no recommended order of attendance. Undergraduate courses of a term must not clash, while undergraduate courses of different terms are allowed to clash. Graduate courses should not clash.

2.3 Observations and Problems

First observations made clear that existing timetables do not meet the requirements stated, e.g. courses of a unit or graduate courses clash or a lecture of an offering and a tutorial of the same offering are scheduled for the same day. Furthermore, considering the number of graduate courses offered over the years, it became clear that there is too little space to schedule all graduate courses without clashes. This is due to the following reason. As mentioned before, undergraduate courses are mandatory and there is a recommended order of attendance. This way it is possible to distinguish students of the first term from students of the third term and students of the second term from students of the fourth term, which makes it possible to allow clashing of undergraduate courses of different terms. The graduate courses only rely on the knowledge provided by the undergraduate courses. There is no recommended order of attendance thus making it impossible to distinguish students of the fifth term from, e.g., students of the seventh term, which makes it necessary to disallow clashing of graduate courses in some way. So we faced two problems:

- The demand for incremental scheduling by basing the new timetable on the timetable of the previous year and changing as little as possible made it necessary to handle old timetables, which do not meet the requirements stated.

- From a scheduler's point of view the graduate studies lack structure taking freedom and leading to over-constrained timetable specifications.

Tackling the second problem by removing selected no-clash constraints turned out to be laborious and time-consuming and therefore impractical. Classifying graduate courses by contents and expected number of students and allowing clashing of courses of different categories won back some freedom, but it was not possible to identify enough categories in such a way that courses spread evenly over categories, which would have been necessary to prevent conflicts. It became clear that we were in need of some kind of weighted constraints able to express weak and strong constraints that are not mandatory.

3 Modeling the Problem as a Partial Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) consists of a finite set of variables, each associated with a finite domain, and a finite set of constraints. A *solution of a CSP* maps each variable to a value of its domain such that all the constraints are satisfied. A *partial constraint satisfaction problem* (PCSP) [FW92] is a CSP where each constraint is associated with a weight. A weight of a constraint expresses the importance of its fulfillment, allowing to distinguish hard from soft constraints. Hard constraints stand out due to infinite weights. The finite weights of soft constraints allow for the specification of preferences among constraints. A *solution of a PCSP* maps each variable to a value of its domain such that all hard constraints are satisfied and the total weight of the violated soft constraints is minimal.

Clearly we only need one variable for each course holding the period, i.e. the starting time point, it has been scheduled for. Each variable's domain consists of the whole week, the periods being numbered from 0 to 167, e.g. 9 denotes 9am on Monday, and so on. Requirements, wishes and recommendations can be expressed with a small set of specialized constraints.

- *No-clash constraints* demand that a course must not clash with another one.
- *Preassignment constraints* and *availability constraints* are used to express teachers' preferences and that a course must (not) take place at a certain time.
- *Distribution constraints* make sure that there is at least one day (hour) between a course and another one or that two courses are scheduled for different days.
- *Compactness constraints* make sure that one course will be scheduled directly after another one.

With respect to soft constraints, we chose to distinguish three grades of preferences: weakly preferred, preferred and strongly preferred, which get translated to the integer weights 1, 3 and 9.

4 Solving the Problem with CHR

Constraint Handling Rules (CHR) [Frü95] is a declarative high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce *user-defined* constraints into a given host language, be it Prolog, Lisp or any other language. To implement our timetabling problem we used the CHR library of ECL^{i}PS^e (ECRC Constraint Logic Programming System [ACD$^+$94]).

CHR is essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. There are basically two kinds of CHR rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence (e.g. `X > Y, Y > X <=> false`). Propagation rules add new constraints which are logically redundant but may cause further simplification (e.g. `X > Y, Y > Z ==> X > Z`). Repeatedly applying the rules incrementally solves constraints (e.g. `A > B, B > C, C > A` leads to `false`). With multiple heads and propagation rules, CHR provides two features which are essential for non-trivial constraint handling. Due to space limitations, we cannot give a formal account of syntax and semantics of CHR in this paper. An overview on CHR can be found in [Frü95]. Detailed semantics results for CHR are available in [Abd97].

4.1 Domains

Constraint solving for finite domains constraints is based on consistency techniques [Mac92, Kum92]. For example, the constraints `X :: [2, 3, 4]`, i.e. `X` must take a value from the list `[2, 3, 4]`, and `X :: [3, 4, 5]` may be replaced by the new constraint `X :: [3, 4]`. Implementing this technique with CHR is straightforward [FA97], but this scheme is not sufficient for our needs: Since soft constraints may be violated, the values to be constrained must not be removed from the domain of the variable. Moreover, when we have to choose a value for the variable during search, we must be able to decide whether a certain value is a good choice or not. Therefore, each value must be associated with an assessment. We chose to represent a domain as a list of value-assessment pairs. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]`. Then `X` may take one of values `3`, `4` and `5`, whereas `4` is encouraged with assessment `1` and `5` is discouraged with assessment `-1`.

4.2 Low-level Constraints

The solver is based on three types of constraints.

- `domain(X, D)` means that `X` must get assigned a value occuring in the list of value-assessment pairs `D`.
- `in(X, L, W)`: Its meaning depends on the weight `W`. If `W = inf`, i.e. if the constraint is hard, it means that `X` must get assigned a value occuring in the list `L`. If `W` is a number, i.e. if the constraint is soft, it means that the assessment for the values occuring in `L` should be increased by `W`.
- `notin(X, L, W)`, if hard, means that `X` must not get assigned any of the values occuring in the list `L`. If it is soft, it means that the assessment for the values occuring in `L` should be decreased by `W`.

4.3 The Core of the Solver

Propagating a soft constraint is intended to modify the assessment of the values to be constrained. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]` and assume the existence of the constraint `in(X, [3], 2)` stating that `3` should be assigned to `X` with preference `2`. Then we have to increase the assessment for value `3` in the domain of `X` by adding `2` to the current assessment of `3` obtaining the new domain `[(3, 2), (4, 1), (5, -1)]` for `X`. However, applying a hard constraint will still mean to remove values from the variable's domain. Consequently, an `in` constraint is processed by either pruning the domain or increasing the assessment for the given values.

```
domain(X, D), in(X, L, W) <=> W = inf |
    domain_intersection(D, L, D1),
    domain(X, D1).                              (fd_in_hard)

domain(X, D), in(X, L, W) <=> W \= inf |
    increase_assessment(W, L, D, D1),
    domain(X, D1).                              (fd_in_soft)
```

In case a hard `in` constraint has arrived, rule `fd_in_hard` looks for the corresponding `domain` constraint, which contains the current domain `D`, and replaces both by a new `domain` constraint, which contains the new domain `D1`. The domain `D1` results from intersecting `D` with the list of values `L`. Rule `fd_in_soft` works quite similiar except for `D1` results from `D` by increasing the assessments for the values occuring in `L`. Note that the guards exclude each other. Therefore, whichever constraint arrives, only one of the rules will be applicable. The rules for `notin` are similar.

```
domain(X, D), notin(X, L, W) <=> W = inf |
    domain_subtraction(D, L, D1),
    domain(X, D1).                          (fd_notin_hard)
```

```
domain(X, D), notin(X, L, W) <=> W \= inf |
    decrease_assessment(W, L, D, D1),
    domain(X, D1).                          (fd_notin_soft)
```

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a domain of a variable has been reduced to the empty list, the variable cannot get assigned a value without violating hard constraints. This case is dealt with by the following simplification rule.

```
domain(_, []) <=> false.                          (fd_empty)
```

With only one value left in a domain of a variable we can assign the remaining value to the variable immediately.

```
domain(X, [(A, _)]) ==> X = A.                (fd_singleton)
```

We use a propagation rule instead of a simplification rule because the `domain` constraint must not be removed. Without it the processing of `in` and `notin` constraints imposed on the domain of a variable would not be guaranteed and thus an inconsistency might be overlooked.

4.4 Treatment of Global Constraints

Up to now we only dealt with the low-level constraints of our finite domain solver. Now we exemplify how to express global (n-ary) application-level constraints in terms of `in` and `notin` constraints.

`no_clash(W, Xs)` means that, depending on the weight `W`, the variables from `Xs` must or should get assigned distinct values. It gets translated to `notin` constraints. This translation is data-driven: whenever one of the variables from `Xs` gets assigned a value, this value gets discouraged or forbidden for the other variables by the following rule.

```
no_clash(W, Xs) <=>
    Xs \= [_],
    select_ground_var(Xs, X, XsRest)
    |
    post_notin_constraints(W, X, XsRest),
    no_clash(W, XsRest).                     (fd_no_clash)
```

The guard first makes sure that Xs contains at least two elements. Then it selects a ground variable X from Xs remembering the other variables in XsRest. With no ground variable in Xs, the Prolog predicate select_ground_var fails. If the guard holds, no_clash(W, Xs) gets replaced by

- notin constraints produced by the Prolog predicate post_notin_constraints, one for each member of XsRest, discouraging or forbidding the value X and
- a no_clash constraint stating that the variables in XsRest should or must get assigned distinct values.

Note that the predicate post_notin_constraints fails in case XsRest contains the value X.

A singleton list of variables means that there is nothing more to do. This case is handled by the following rule.

```
no_clash(_, [_]) <=> true.        (fd_no_clash_singleton)
```

The translation of the other application-level constraints either follows this scheme or is a one-to-one translation.

4.5 Interaction of the no_clash Rules and the the Core of the Solver

In the following, we present two examples to show how the CHR rules interact with each other. In the first example, we deal only with hard constraints. Assume the current state of a computation consists of the constraints

```
    domain(X, [(1, 0), (2, 0)]),
    domain(Y, [(1, 0), (2, 0)])
and no_clash(inf, [X, Y]).
```

Since neither X nor Y are ground, no rule is applicable. After adding the constraint in(X, [1], inf) rule fd_in_hard becomes applicable and simplifies

```
    domain(X, [(1, 0), (2, 0)])
and in(X, [1], inf)
 to domain(X, [(1, 0)]).
```

Now rule fd_singleton becomes applicable and propagates the equality constraint X = 1. Then rule fd_no_clash becomes applicable and simplifies

```
    no_clash(inf, [1, Y])
 to notin(Y, [1], inf)
and no_clash(inf, [Y]).
```

Then rules fd_no_clash_singleton and fd_notin_hard become applicable: rule fd_no_clash_singleton removes no_clash(inf, [Y]) and rule fd_notin_hard simplifies

```
        domain(Y, [(1, 0), (2, 0)])
    and notin(Y, [1], inf)
    to  domain(Y, [(2, 0)]).
```

Finally, rule `fd_singleton` becomes applicable and propagates the equality constraint `Y = 2`. Thus, the final state of the computation consists of

```
        domain(X, [(1, 0)]),
        domain(Y, [(2, 0)]),
        X = 1
    and Y = 2.
```

In the second example, we want to show how the rules treat soft `no_clash` constraints. Assume the current state of a computation consists of the constraints

```
        domain(X, [(1, 0), (2, 0)]),
        domain(Y, [(1, 0), (2, 0)])
    and no_clash(1, [X, Y]).
```

Again we add `in(X, [1], inf)`. Until rule `fd_no_clash` becomes applicable, the computation proceeds as before. Then rule `fd_no_clash` simplifies

```
        no_clash(1, [1, Y])
    to  notin(Y, [1], 1)
    and no_clash(1, [Y]).
```

Finally, both rules `fd_no_clash_singleton` and `fd_notin_soft` become applicable: rule `fd_no_clash_singleton` removes `no_clash(1, [Y])` and rule `fd_notin_soft` simplifies

```
        domain(Y, [(1, 0), (2, 0)])
    and notin(Y, [1], 1)
    to  domain(Y, [(1, -1), (2, 0)]).
```

Thus, the final state of the computation consists of

```
        domain(X, [(1, 0)]),
        domain(Y, [(1, -1), (2, 0)])
    and X = 1.
```

4.6 Propagation Performance

The first rule for `no_clash` (`fd_no_clash`) acts as a constraint propagator that amplifies the constraint store by incrementally spanning a network of `notin` constraints. Since the propagator sleeps as long as none of the variables it surveys gets assigned a value, a `no_clash` constraint cannot contribute to a solution as long as none of its courses gets scheduled. This approach is similiar to the implementation of CHIP's `all_different` constraint [vH89].

Combining our solver with chronological backtracking results in a search procedure, which, with respect to propagation performance, is a little better than the forward checking algorithm [HE80] and much worse than the generalized arc-consistency algorithm [MM88].

Concerning the reuseability of our solver, we cannot give a definite answer. On the one hand, experience shows that, for a variety of problems, forward checking together with additional search is more efficient than applying more expensive consistency techniques [Kum92]. On the other hand, there is evidence that maintaining arc-consistency is necessary to solve the larger and the harder problems efficiently [BR96, SF97, Mar98].

Whether the performance of our solver is sufficient to solve a whole university timetabling problem depends on the structure of the problem. If departments share teachers, students, rooms or equipment and sharing has to be taken into account, the problem might be too hard. Otherwise, the university timetabling problem breaks down into several independent timetabling problems, one for each departement. This is the case with most German universities.

4.7 The Search Procedure

The search procedure employed integrates the solver given above with chronological backtracking and heuristics for variable and value selection. For variable selection, we chose the first fail principle [HE80] which dynamically orders variables by increasing cardinality of domains, i.e. the principle proposes to select one of the variables with the smallest domains with respect to the current state of computation. For value selection, we used a best-fit strategy choosing one of the best-rated periods. From an optimistic point of view, this will be one of the periods violating a set of soft constraints with minimal total weight, but the estimate may be too good due to the low propagation performance of the `no_clash` solver. Furthermore, the best assessment does not necessarily violate a minimum number of constraints: a strong personal preference may balance out ten weak no-clash constraints. This approach yielded a good first solution to our problem. It was not necessary to search for a better solution.

4.8 Generation of Timetables

The generation of a timetable runs as follows. Each course is associated with a `domain` constraint allowing for the whole week, the periods being numbered from 0 to 167. It is important to note that, for each course, the initial assessment for all periods is 0 indicating that no period is given preference initially. Then preassignment constraints and availibility constraints will be translated into `in` and `notin` constraints. Adding `in` and `notin` constraints may narrow the domains of the courses using the rules presented above. Propagation continues until a fixpoint is reached, that is to say, when further rewriting does not change the store. Usually, our consistency based finite domain solver is not powerful enough to determine that the constraints are satisfiable. In order to guarantee that a valid solution is found the search procedure is called. Addition of an `in` constraint may initiate propagation, and so on.

Now, that we have discussed the details of creating a timetable, how do we create a new timetable based on a timetable of the previous year with our system? Central to our solution is the notion of *fixing a timetable*. Fixing a timetable consists in adding a (strongly preferred) soft preassignment constraint for each course that has been scheduled ensuring that all courses offered again will be scheduled for the same time.

The time necessary to compute a timetable depends on whether a previous timetable is reused or not. Scheduling 89 courses within 42 time periods from scratch took about five minutes. Considering an "almost good" previous timetable saved about two and a half minutes.

5 Conclusion

Constraint Handling Rules (CHR) is a declarative high-level language extension especially designed for writing application-oriented constraint solvers. In this paper, we have argued that CHR is a good vehicle for implementing a finite domain solver, which performs hard and soft constraint propagation. The solver is powerful enough to serve as the core of a university timetabling system.

Our scheduler runs on ECL^{i}PS^e complemented by the CHR library. We rely on the internet, and more specifically the World-Wide-Web (WWW) to enable teachers to enter new wishes and offerings into the specification by themselves. The Web frontend is based on HTML, pages are also generated by a Prolog program. Developing the constraint solver and attaching it to a database of timetabling problems took about three weeks; developing the front end took another two weeks. The solver takes only a few lines of code.

The IfIPlan system has been in use at the Computer Science Department of the University of Munich for four terms. The good execution times achieved and the very reasonable timetables generated demonstrate that CHR is able to reconcile efficient execution and declarative implementation. Our very high-level approach also means that the program can be easily maintained and modified.

References

[AB94] Francisco Azevedo and Pedro Barahona. Timetabling in constraint logic programming. In *Proceedings of 2nd World Congress on Expert Systems*, 1994.

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Princi-*

ples and Practice of Constraint Programming, LNCS 1330. Springer, 1997.

[ACD+94] A. Aggoun, D. Chan, P. Dufrense, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Maier, D. Miller, B. Perez, E. van Rossum, J. Schimpf, P. Tsahageas, and D. de Villeneuve. *ECL^iPS^e 3.4 User Manual*. ECRC Munich Germany, July 1994.

[AM98] Slim Abdennadher and Michael Marte. University timetabling using constraint handling rules. In *Actes des Journées Francophones de Programmation en Logique et Programmation par Contraintes*, 1998.

[BFBW92] Alan Borning, Bjorn N. Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[BR96] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Second International Conference on Principles and Practice of Constraint Programming*, LNCS 1118, pages 61–75. Springer, 1996.

[FA97] Thom Frühwirth and Slim Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen*. Springer, 1997.

[FHS95] Harikleia Frangouli, Vassilis Harmandas, and Panagiotis Stamatopoulos. UTSE: Construction of optimum timetables for university courses — A CLP based approach. In *Proceedings of the Third International Conference on the Practical Applications of Prolog*, pages 225–243, 1995.

[Frü95] Thom Frühwirth. Constraint handling rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer, 1995.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 1998.

[FW92] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[HE80] Robert M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[HW95] Martin Henz and Jörg Würtz. Using Oz for college time tabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 283–296, 1995.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, 1994.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1), 1992.

[Mac92] Alan K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. Wiley, 1992. Volume 1, second edition.

[Mar98] Michael Marte. Constraint-based grammar school timetabling – A case study. Diplomarbeit, Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen, Institut für Informatik, Ludwig-Maximilians-Universität München, 1998.

[Mey97] Harald Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *Proceedings of the 3rd International Conference on the Practical Application of Constraint Technology*, pages 257–272, 1997.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656. Pitman Publishers, 1988.

[MS98] Kim Marriott and Peter Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

[Sch95] Andrea Schaerf. A survey of automated timetabling. Technical Report CS-R9567, CWI - Centrum voor Wiskunde en Informatica, 1995.

[SF97] Daniel Sabin and Eugene C. Freuder. Understanding and improving the mac algorithm. In *Third International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, pages 167–181. Springer, 1997.

[vH89] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

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