Classroom Assignment using Constraint Logic Programming

Slim Abdennadher, Matthias Saft and Sebastian Will

Computer Science Department, University of Munich
Oettingenstr. 67, 80538 Munich, Germany
{abdennad, saft, wills}@informatik.uni-muenchen.de

Abstract

The *Classroom Assignment problem* consists of scheduling a set of courses into a fixed number of rooms, given a fixed timetable. At the University of Munich, a classroom plan has to be created each semester after collecting timetables of all departments and wishes of teachers. This planning is very complex since a lot of constraints have to be met, e.g. room size and equipment. Using constraint-based programming, we developed a prototype, called *RoomPlan*, that supports automatic creation of classroom plans. With *RoomPlan* a schedule can be created interactively within some minutes instead of some days. *RoomPlan* was presented at the Systems'99 Computer exhibition in Munich.

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. In most universities, the university course timetabling problem is solved in two phases. In the first phase timetables have to be created, one for each department. Since departments can share rooms, the availability of rooms is not taken into account in the first phase. In the second phase, rooms have to be assigned to courses. The assignment of rooms is done centrally for the whole university.

The classroom assignment problem is a difficult and time-consuming expert task since a lot of requirements have to be met. For example, courses must be assigned to rooms based on the number of students taking the courses and capacities of rooms. Furthermore, some courses may require special equipments such as beamer or internet access. While for the first phase of the university course timetabling several systems have been developed [4, 12, 7], to our knowledge mainly theoretical work has been done on the topic of the classroom assignment problem [6, 5].

Recently, Constraint Logic Programming [14, 20, 11, 18] (CLP) has become a promising approach for solving scheduling problems. CLP combines the advantages of logic programming and constraint solving techniques. The use of CLP has the advantage that the solving procedure can easily be adapted to changing scheduling characteristics. The classroom assignment problem can be elegantly formalized as a partial constraint satisfaction problem and implemented by means of specialized constraint solving techniques that are available in CLP languages.

In this paper, the generation of classroom plans for universities is tackled using the CLP framework. The system is called *RoomPlan* and is currently tested at the University of Munich.

Our prototype brought down the time necessary for creating a classroom plan from a few days by hand to a few minutes on a computer.

Usually not all specified requirements can be fulfilled. We distinguish two kinds of constraints. Hard constraints are conditions that must be satisfied, soft constraints may be violated, but should be satisfied as far as possible. The classical approach to deal with these requirements is based on a variant of branch-and-bound search. This approach starts with a solution and requires the next solution to be better. Quality is measured by a suitable cost function. The cost function depends on the set of satisfied soft constraints. Usually, the computation of the cost function is incorporated into the labeling process. In this paper, we propose another approach computing the cost function during the constraint solving process independent of the labeling procedure. This requires to modify the constraint solving part.

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. Since this 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 problem using the "glass-box" approach Constraint Handling Rules (CHR) [9, 10]. CHR is a powerful special-purpose declarative programming language for writing application-oriented constraint solvers.

For our need, we extended an existing finite domain solver written in CHR [11] in a way that the cost for a solution is computed during the propagation of soft constraints. CHR allows to express the calculation of the cost in a very declarative and straightforward manner.

In this paper, we describe the main features of the constraint solver that was used to generate a classroom plan for the University of Munich. Section 2 introduces our classroom assignment 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. Section 5 describes the user interface. Finally, we conclude with a summary.

2 Problem Description

In universities, where each department is responsible for its own timetable and where rooms can be shared by different departments, timetables are usually generated in two phases. In the first phase an assignment of courses within a given number of periods is done without taking into account the availability of rooms. This task has to be performed separately for each department. For the Computer Science Department at the University of Munich, the first phase is solved automatically by a system that generates a new timetable based on a timetable of the previous year [2, 3]. For other departments the generation of timetables is still done by hand. In the second phase an assignment of courses within a given number of rooms has to be performed. After collecting timetables of all departments and wishes of teachers a classroom plan is generated centrally.

In the following, we want to investigate the classroom assignment problem of the University of Munich. Since timetables for departments change every semester, a new classroom plan has to be created each semester. The University of Munich dispose of different buildings. The biggest building consists of 40 rooms where about 1000 courses have to be held.

The generation of a classroom plan is a difficult and time-consuming task since different kinds of constraints have to be taken into account:

- The *no-occupation overlap constraint* tells that occupation time of a room by courses must not overlap.
- The *seat requirement constraint* tells how many seats a course requires.
- The *teacher's wishes*: We distinguish three kinds of wishes.
  - A *room constraint* binds a course date to a room.
  - A *building constraint* assigns a course date to a certain building.
  - An *equipment constraint* constrains a course date to be assigned to a room with certain technical equipment, e.g. beamer or video.

Usually not all specified requirements can be fulfilled since the number of (special) rooms is obviously limited. Therefore we distinguish two kinds of constraints. Hard constraints must always be satisfied, soft constraints may be violated. Roughly speaking, *no-occupation overlap constraints* and *seat requirement constraints* determine hard constraints, wishes may be hard or soft constraints.

3 A Constraint Model for Classroom Assignment

A *constraint satisfaction problem* (CSP) [16] (V, C) is a pair, where V is finite set of variables, each associated with a finite domain, and C is a finite set of constraints on these variables. A *solution* of a CSP maps each variable to a value of its domain such that all the constraints are satisfied. Since we have to address quality of a room plan and therefore, have to take into account wishes as well as exploitation of resources, CSP can not model our problem completely. Therefore, we use an extension of the CSP concept. A *partial constraint satisfaction problem* PCSP [8] is a triple (V, C, ω), where (V, C) is a CSP and ω is a total function $\omega : C \to \mathbb{R}$, i.e., ω maps constraints to weights. The weight of a constraint expresses its importance. Thus, one can describe *hard constraints*, which must be satisfied, as well as *soft constraints*, which should be satisfied. A hard constraint is given an infinite weight. Then, a *solution of the PCSP* is an assignment of the variables in V to their domains, such that the total weight of all violated constraints $c \in C$ is minimized.

Now, the classroom assignment problem is modeled as a PCSP. Note, that it does not suffice to assign a room to each course, but instead we have to assign a room to each date, where a course is held. Therefore, we use one variable for each *course date*. For example, if a course consists of two lectures, the course is represented by two different course date variables. The initial domain of each course date variable is the set of all rooms in the university. Thus, the solution is an assignment of course dates to rooms.

There are two constraints that occur only as hard constraints and thus have infinite weight: the *no-occupation overlap constraint* and the *seat requirement constraint*. Wishes may also be hard, i.e. have infinite weight.

To ensure a good exploitation of resources by a solution, we evaluate assignments of a room to a course date. For this reason, we modify the weight $\omega(\alpha)$ for the constraint α, that assigns a course date for a course c to a room r. This is done by adding a term to the user-defined evaluation $\omega'(\alpha)$, thus defining $\omega(\alpha)$ in the following way.

$$\omega(\alpha) = \omega'(\alpha) + a_1 \cdot \frac{\text{seats}_r - \text{requirement}_c^{\text{seat}}}{\text{seats}_r} + \min(0, a_2 \cdot \frac{\text{equipment}_r - \text{requirement}_c^{\text{equipment}}}{\text{equipment}_r}), \tag{1}$$

where seats_r is the number of seats in room r, $\text{requirement}_c^{\text{seat}}$ is the number of seats required by course c, equipment_r is a valuation of the technical equipment in r and $\text{requirement}_c^{\text{equipment}}$ a value for the technical requirements of c. a_1 and a_2 are negative constants weighting the exploitation of seats and equipment resources against each other and the violation of wishes. Since the *equipment constraint* can be soft, the value of the technical requirements can be greater than the value of the equipment. In this case, we have to avoid that the third term of the function $\omega(\alpha)$ is positive.

4 Solving the Problem using Constraint Handling Rules

In a PCSP, one has additionally to satisfying all hard constraints to take soft constraints into account. According to the PCSP model, we have to minimize the total weight of violated soft constraints. This is equivalent to maximize the total weight of satisfied constraints. We use a *branch-and-bound* approach to tackle this maximization problem. Branch-and-bound is a standard method to optimize a score that works by constraining the score during the search. Every time an assignment satisfying the hard constraints is found, the score is bound to be even better. Thus, the last assignment compatible to the hard constraints that is found will have an optimal score. Therefore, we incrementally compute a bound of the score, that an assignment compatible to the current hard constraints may have, during the enumeration. This way we prune the search tree every time the maximally achievable score is worse than the score of the previous solution.

To prune the search tree efficiently in our branch-and-bound algorithm, we have to keep track of the upper bound of the score. The upper bound of the score may be affected each time the constraint store changes. This change may be done either by a constraint which is directly inserted by the labeling process or by constraint propagation. If only changes of the first kind could affect the upper bound, the calculation of the score could be easily incorporated into the labeling process. However, since we also have to take care of the second kind of constraint store changes, it is much more natural and intuitive to do this calculation concurrently to the labeling process and triggered by the alteration of the constraint store. Constraint Handling Rules (CHR) allows to express this in a very declarative and straightforward manner, where the calculation is formulated independently of the labeling.

In the following, CHR is briefly introduced. Then, the core of the constraint solver, which performs hard and soft constraint propagation, is presented. Finally, a labeling strategy to find good solutions is described.

4.1 Constraint Handling Rules

CHR 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. The main implementations of CHR are currently available in Prolog: Besides the library from 1994, ECL^{i}PS^e 4.0 now includes a new experimental prototype of CHR. An advanced optimizing CHR compiler was released for Sicstus 3.7 in 1998 [13]. CHR has also been implemented in Common Lisp, OZ and Java. To implement the constraint solver for our classroom assignment problem we used the CHR library of Sicstus. Thus, code examples follow the syntax of CHR and Prolog.

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. Propagation rules add new constraints which are logically redundant but may cause further simplification. Repeatedly applying the rules incrementally solves constraints. 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 [10]. Detailed semantics results for CHR are available in [1]. We introduce CHR by example. Let =< and = be built-in constraint symbols. We implement a user-defined constraint for `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y`:

```
max(X,Y,Z) <=> X=<Y | Z=Y.
max(X,Y,Z) <=> Y=<X | Z=X.
max(X,Y,Z) ==> X=<Z, Y=<Z.
```

The first rule states that `max(X,Y,Z)` is logically equivalent to `Z=Y`, provided it is the case that `X=<Y`. This test forms the guard of a rule, a precondition of the applicability of the rule. Hence, whenever we see the constraint `max(X,Y,Z)` in any goal where it holds that `X=<Y` we can simplify it to `Z=Y`. Analogously for the second rule.

The first and second rules are *simplification rules*. The third rule propagates constraints. It states that `max(X,Y,Z)` unconditionally implies `X=<Z`, `Y=<Z`. Operationally, we add these logical consequences as redundant constraints, the `max` constraint is kept. This kind of rule is called *propagation rule*.

To the goal `max(1,2,M)` the first rule is applicable: `max(1,2,M)` will be simplified to `M=2`.

Redundancy from the propagation rule is useful, as the goal `max(A,3,3)` shows: To this goal only the propagation rule is applicable, but then the first rule: `max(A,3,3)` cause the propagation rule to fire and adds `A=<3, 3=3` to the goal. Now, the first rule is applicable and simplifies the constraint `max(A,3,3)` to `3=3`. The constraint `3=3` is simplified to true by the built-in constraint solver. The constraint solver for `max` solved `max(A,3,3)` and produced the answer `A=<3`.

4.2 The Core of the Constraint Solver

4.2.1 Handling Hard Constraints

Regarding just the hard constraints, our solver is essentially a finite domain solver, i.e. a course date variable is bound to a list of rooms and constraints may eliminate rooms from the domain list of the constrained variable.

Constraint solving for finite domains constraints is based on consistency techniques [17, 15]. For example, the constraints `X::[2,3,4]` and `X::[3,4,5]` may be replaced by the new constraint `X::[3,4]` (The constraint `X::[2,3,4]` means that `X` has to take a value from the list `[2,3,4]`). Implementing this technique with CHR is straightforward [11]. The first rule ensures that the domain for `X` is non-empty, the second rule intersects two domains for the same variable:

```
X::[] <=> false.
X::L1, X::L2  <=> intersection(L1,L2,L), X::L.
```

To solve our classroom assignment problem, the *no-occupation overlap constraint* can be expressed using the global constraint `all_distinct`. The constraint `all_distinct(Xs)` tells that all variables in the list `Xs` must be bound to different values. The *no-occupation overlap constraint* is propagated to constraints `all_distinct(Xs)`, where the `Xs` are lists of the course date variables for all course dates that overlap in time. The hard *seat requirement constraint* propagates by filtering the domains of the course date variables.

4.2.2 Handling Soft Constraints

In the following, the calculation of the score done by a CHR program is described. It is most intuitive to calculate the total score from three sub-scores, which can easily be done by a rule. One sub-score `ScoreWish` is the total weight of satisfied wishes, the second `ScoreSeatRes` is the sum of the second terms in equation (1) over all assignments of course dates to rooms, i.e. a measure of the exploitation of seats. Analogously, the last `ScoreEquipmentRes` is the sum of the third terms in equation (1), i.e. a measure of the exploitation of equipment. The total score is computed as a weighted sum of the sub-scores. Instead of minimizing the total weight of violated constraints, we equivalently maximize the total weight of the satisfied soft constraints. Since we maximize the score we need to compute an upper bound of the score, that an assignment which satisfies the hard constraints of the current constraint store may have.

We start with describing the computation of the sub-score `ScoreWish`. The different types of wishes are expressed by CHR constraints of the form `wish(Type, CourseDate, Wished, Weight)`, where the first argument gives the type of the wish, namely `room`, building or equipment. CourseDate holds a course date identifier, the variable Wished further specifies the instance of the wish and Weight holds the weight of the wish. We use further the constraint assignment(CDate,CDateVar) which tells that CDateVar is the course date variable corresponding to the course date CDate. The constraint scoreWish(Up) tells that Up is the upper bound of the sub-score ScoreWish.

In the following, we introduce simplification rules to update the sub-score ScoreWish, whenever a wish is satisfied or violated. In the following, we will discuss the rules for handling *room constraints*. The rules handling *building* and *equipment constraints* are analogous.

```
assignment(CDate, CDateVar), CDateVar::Dom,
wish(room, CDate, RoomWish, infinite) <=>
       assignment(CDate, CDateVar), CDateVar::Dom,
       CDateVar::[RoomWish].

assignment(CDate, CDateVar), CDateVar::[RoomWish],
wish(room, CDate, RoomWish, Weight) <=>
                Weight \== infinite |
       assignment(CDate, CDateVar), CDateVar::[RoomWish].

assignment(CDate, CDateVar), CDateVar::Dom,
wish(room, CDate, RoomWish, Weight), scoreWish(Up) <=>
                Weight \== infinite, \+ member(RoomWish, Dom) |
       assignment(CDate, CDateVar), CDateVar::Dom,
       scoreWish(Up - Weight).
```

The first rule propagates a hard wish, i.e. a wish with infinite weight, in such a way that an assignment of the course date variable to the wished room is performed. Therefore, the simplification rule tests whether constraints of the form assignment(CDate, CDateVar), CDateVar::Dom and wish(room, CDate, RoomWish, infinite) can be found in the constraint store. In this case, the rule fires and the constraint CDateVar::[RoomWish], that binds the room to the wished room, is added to the store. Furthermore, the wish constraint is removed from the constraint store. The application of this rule leads to the occurrence of two domains for the same variable. These constraints can be simplified by the intersection rule presented above.

The second rule handles already satisfied soft constraints. If the constraints assignment(CDate, CDateVar), CDateVar::[RoomWish] and wish(room, CDate, RoomWish, Weight) are found in the constraint store, then the wish is obviously satisfied and the rule consequently removes the wish. The guard ensures that only soft constraints, i.e. wishes with finite weight, are handled by this rule, since hard wish constraints are already handled by the first rule. Note that the upper bound of the score is unaffected if a wish is satisfied. In contrast, if a wish is violated, the upper bound of the score has to be decreased.

The updating of the bound is done by the third rule. The guard ensures that only soft constraints which are violated lead to a change of the score. A *room constraint* is violated if the wished room is not contained in the domain of the course date variable. If the rule fires, the upper bound is recomputed as `Up - Weight` and the constraint `scoreWish(Up - Weight)` replaces the constraint `scoreWish(Up)`.

The evaluation of resource exploitation is handled by a single rule.

```
assignment(CDate, CDateVar), CDateVar::[RoomNr],
scoreSeatsRes(UpS), scoreEquipmentRes(UpT) <=>
       assignment(CDate, CDateVar), CDateVar::[RoomNr],
       scoreSeatsRes_diff(CDate, RoomNr, SSD),
       scoreEquipmentRes_diff(CDate, RoomNr, TSD),
       scoreSeatsRes(UpS + SSD),
       scoreEquipmentRes(UpT + TSD).
```

The rule updates the scores for seat resource and equipment resource exploitation, where the actual weight is calculated by the separate predicates `scoreSeatsRes_diff` and `scoreEquipmentRes_diff` analogously to equation (1). The updating of the sub-scores is done by replacing the constraints `scoreSeatsRes(UpS)` and `scoreEquipmentRes(UpT)` by the recomputed constraints `scoreSeatsRes(UpS + SSD)` and `scoreEquipmentRes(UpT + TSD)` each time an assignment of a course date variable to a room was newly created, either by labeling or by constraint propagation.

Each time a sub-score is recomputed, the total score has to be recalculated. This can be done by the following propagation rule.

```
scoreWish(UpW), scoreSeatsRes(UpS),
scoreEquipmentRes(UpT) ==>
       score(UpC + UpS + UpT).
```

The rule fires if a sub-score changes. Then, a constraint with the newly calculated upper bound of the total score is inserted. A further rule ensures that only the most restrictive `score` constraint remains in the constraint store.

```
score(A), score(B) <=> A=<B | score(A).
```

This rule removes the larger of two upper bounds of the total score.

Now, the upper bound can be used to prune the search tree, since we use a branch-and-bound algorithm. Every time an assignment of the course date variables that satisfies the hard constraints is found, we insert a constraint `last_score(Score)` with the score of this assignment. The following propagation rule ensures, that only better assignments can be found in consequence.

```
last_score(LastScore), score(Up) <=> LastScore >= Up | false.
```

This rule causes the constraint solver to fail whenever the score of an assignment in the current branch cannot be better than the last score. This is indicated by the upper bound of the score.

4.3 Labeling

After stating the problem constraints, normally there are still many feasible solutions, so it is necessary to label the domain variables, i.e. assign them with values that remain on their domains. For the labeling one needs to apply a heuristic strategy that tends to enumerate high scoring solutions early in the search. In a branch-and-bound search this helps to prune the search tree. In practice also suboptimal solutions may be appropriate, which also emphasizes the use of finding good solutions early.

We employed a *leftmost* variable, *leftmost* value strategy that selects for each assignment the leftmost course date variable and the leftmost value from the domain of the selected course date variable. Since we sort the list of variables as well as the values in the domains before the labeling as we describe below, this strategy prefers most constrained assignments. The sorting is done in the following way. We compute a weight for each course and a weight for each room with respect to a certain course, i.e. actually a weight for a certain assignment. The weights for courses respect seat and equipment requirements, such that courses with strong requirements get great weights. The weight of an assignment totals the weights of the soft constraints which are satisfied by the actual assignment. Then, the course date variables and the rooms in each domain are sorted in descending order by these weights. As a consequence the leftmost course date variables, which are selected first by our strategy, belong to the courses with the strongest requirements. Further, the leftmost values in each domain lead to assignments, which satisfy the ”best” soft constraints.

5 User Interface

For a classroom assignment system to be complete, a flexible user interface should be provided, so that the specific requirements of the problem can be stated easily. *RoomPlan* provides such an interface.

The *RoomPlan* user interface was written in JAVA™ 1.1 using the Abstract Window Toolkit (AWT). It generates the data needed by the CHR solver in form of prolog facts and starts the CHR solver.

Figure 1 shows the input form for the courses that have to be planned. The user has to type in the courses with title, first and last day and start- and endtime. Furthermore, the interface allows the user to define the system parameters as preferred. All parameters like the needed amount of seats and needed equipment, e.g. overhead projector are handled as soft constraints. In addition, the user can specify a wish for a special room. These wishes are given in four categories: `imperative`, `strong preferred`, `preferred` and `weakly preferred`. Imperative wishes are hard constraints. If there is no wish for a special room, a building can be specified, which is considered by the classroom assignment.

It may happen that there exists not even one classroom plan that complies with all the given hard constraints. Then the problem is called *over-constrained*. In case of a direct clash in form of two conflicting imperative room wishes, *RoomPlan* may detect this and then gives the user a variety of alternative rooms which satisfy the specified equipment wishes.

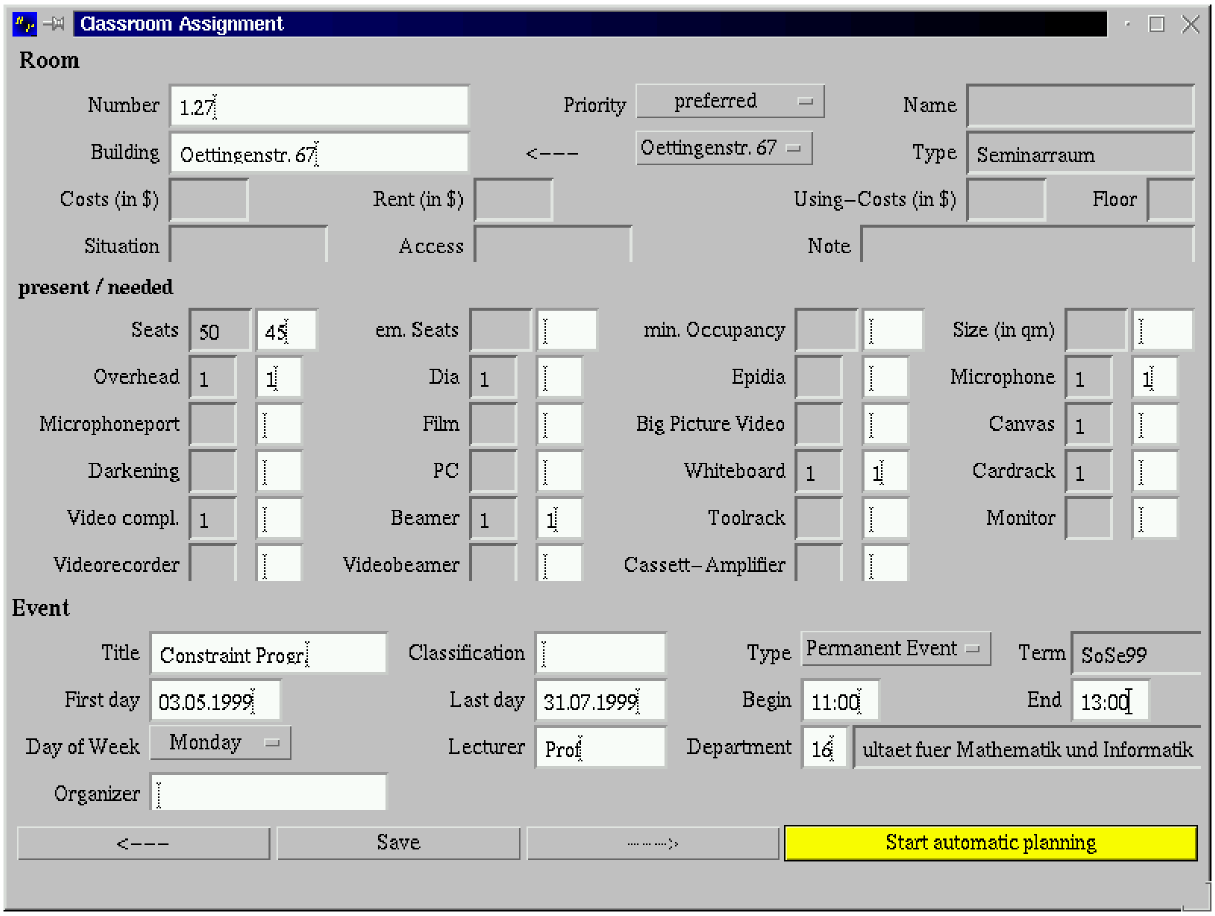

Figure 1: Classroom Assignment

Figure 2 shows a fragment of the generated classroom plan. In addition, the user can manually alter a completely generated classroom plan and let it check by *RoomPlan*. An advantage of the interactive system is that the expert might assist the program to produce even better results.

6 Conclusion

In this paper, we have argued that Constraint Handling Rules (CHR) is a good vehicle for implementing a PCSP solver. We extended an existing finite domain solver to deal with soft constraints. The idea was to compute the cost function during the propagation of constraints. This is much more natural and intuitive than the incorporation of the computation into the labeling process. Thus our approach is comparable in terms of declarativity to approaches embedding cost functions into a CLP(FD) system using reification. The solver is powerful enough to serve as the core of a classroom assignment system.

Our system, *RoomPlan*, has been implemented using Sicstus Prolog. The CLP code is just about 700 lines. *RoomPlan* assists a human planner in scheduling courses to rooms. *RoomPlan* was presented at the Systems'99 Computer exhibition in Munich. It is currently tested at the University of Munich. Typically, for 1000 courses and 40 rooms, *RoomPlan* generates a satisfying schedule within a few minutes.

Figure 2: Fragment of a Classroom Plan

RoomPlan has been designed to meet the specific requirements of the University of Munich. However, it can be applied to other universities, since the classroom assignment problem is already handled in a general way. However, if necessary, adaption can easily be done due to the the declarativity of constraint logic programming.

In general problems a labeling strategy with previously fixed enumeration order is not suitable, since a good order of variables or domain values often cannot be fixed in advance. Only the special nature of our classroom assignment problem makes this strategy a suitable choice. Nevertheless, we are currently developing intelligent labeling strategies as quoted by Tsang [19].

Acknowledgements

We would like to thank Thom Frühwirth and the anonymous referees for useful comments on a preliminary version of this paper.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330. Springer-Verlag, 1997.

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

[3] S. Abdennadher and M. Marte. University course timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, to appear 2000.

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

[5] M. Carter and C. Tovey. When is the classroom assignment problem hard? *Operations Research*, 40(1):28–39, 1989.

[6] J. Ferland and S. Roy. Timetabling problem for university as assignment of activity to resources. *Computers and Operational Research*, 12(2):207–218, 1985.

[7] H. Frangouli, V. Harmandas, and P. 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.

[8] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, December 1992.

[9] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910. Springer-Verlag, 1995.

[10] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.

[11] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen*. Springer-Verlag, September 1997.

[12] M. Henz and J. 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.

[13] C. Holzbaur and T. Frühwirth. A prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, to appear 2000.

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

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

[16] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[17] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley, 1992. Volume 1, second edition.

[18] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

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

[20] M. Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1,2), September 1996.