Priority-driven Constraints Used for Scheduling at Universities

Michael Baumgart, Hans Peter Kunz, Sascha Meyer, and Klaus Quibeldey-Cirkel
Department of Electrical Engineering and Computer Science
University of Siegen, D-57068 Siegen, Germany

e-mail: quibeldey@ti.et-inf.uni-siegen.de

Abstract: This paper presents a timetabling application for the University of Siegen. The problem is the allocation of courses (activities) to a limited number of rooms (resources). The approach is based on an object-oriented taxonomy which splits constraints in hard and soft constraints. Hard constraints must be satisfied to achieve a consistent solution. Soft constraints are ordered by priority. The solver should satisfy as many soft constraints as possible. The object-oriented software method of Booch [3] is used giving a sense of the "social process" involved in developing a constraint-satisfaction application.

1 Introduction

At the University of Siegen – like at most other universities – scheduling is a complex problem. Only a few individual wishes of the lecturers as well as of the students can be considered in the planning phase, so dissatisfaction cannot be avoided. Due to the large number of courses (> 3,000) and the increasing number of students registered (> 12,500), lecture halls and seminar rooms are scarce resources (< 90). A computerised solution for this NP-hard problem has failed because of insufficient search algorithms. Although, timetabling is a well-known area in the field of Constraint Satisfaction [1], there is no commercial software available satisfying the complex requirements of university scheduling.

As a project group made up of 6 advanced students of computer science, we are attempting to solve the problem with the C++ library of ILOG Solver [2] (abbreviated to "Solver" in the following). During the whole software development process we are using the object-oriented method of Grady Booch [3].

2 Analysis of the Problem Domain

In this section, we give an overview of the results of our requirements analysis. When we started work, we had to fully familiarise ourselves with the planning situation. So we interviewed all persons at the university involved with creating timetables of any kind. As each of the twelve departments has its own timetable, we interviewed more than twenty people for information.

It turned out that these people were very interested in a software capable of doing the complex scheduling task. So the interviews were a great support for our work because we compiled a lot of domain-specific information. In the following we want to describe the scheduling process and examine the various restrictions which have to be taken into account. We will also have a look at our prototype objectives.

2.1 Scheduling Process

We found out in our interviews that the scheduling process varies from department to department. In spite of this, we can state 4 activities which can be found in every department:

1. **Data survey**: At first, a survey of the numerous data required for generating the new schedule has to be made. Concerning the resources, there is not much to do, because the assignment of rooms to the individual departments does not change very often.

   What is also needed is a lot of information about the courses which have to be placed in the new schedule. Therefore, every lecturer receives a survey form to fill out with the requirements of the courses he or she wants to hold and sends it back to the planning team of the department. Depending on the number of lecturers, this process usually takes some time. For example, in our department "Electrical Engineering and Computer Science", this process takes from 3 to 5 weeks. Only a few "smaller" departments do not have to send survey forms to their lecturers. Because of the small number of lecturers, they are able to do this verbally and, thus, save a lot of time.

2. **Generating the schedule**: After the survey, the main work of the planning team starts, i.e. the generation of a schedule. Because of the large number of changes from term to term, recycling the last schedule often is not possible. So the job has to be done each and every term. In most departments, the scheduling work takes several days, often weeks. Scheduling in our department, for example, is done by a group of 3 people and takes them up to 3 weeks. This work takes so much time because the planners have to consider an almost unlimited number of restrictions to the schedule, where the constraints of the administration and examination regulations often contradict the interests of the lecturers and students.

   When we interviewed the planners, they complained about the lack of any kind of software support. Unfortunately – until now – there is no software product available that is capable of solving the problem or supporting the planners at their work. So everything has to be done manually using several oversized sheets of paper, pencils, and erasers. The planners have to concentrate on the procedure over a long period of time, which makes it very difficult to avoid faults in the planning. It is also nearly impossible to generate a "really good" schedule for all involved.

3. **Reaching agreement with the lecturers**: Finally, the schedule has to be checked by the individual lecturer. Often lecturers are not satisfied with the schedule, because their time preferences have not been fulfilled or changed during scheduling. But due to the strong dependencies between courses and the fact that lecture halls and seminar rooms are fixed resources, moving a course dramatically affects the whole schedule by propagating multiple changes to other courses.

4. **Publishing the schedule**: Having completed scheduling, the schedule has to be published. Due to the duration of the procedure, many departments do not manage to finish their work before end of term. So, schedules are often published in a special booklet shortly before the next term starts. To the students, this late publishing date is quite annoying.

2.2 Restrictions Considered

What do the planners actually have to consider? There are a lot of restrictions to the schedule. Some of them have to be fulfilled, otherwise regular studies will be impossible. Others are *individual* requirements that should be satisfied to make the resulting schedule as acceptable as possible. Generally, these restrictions can be divided into the following 4 categories:

1. **Examination regulations**: The examination regulations specify the courses the student has to attend. Some courses are *compulsory*, others are *required elective*. The planners have to consider these requirements while working on the schedule. For example, they are not allowed to place compulsory courses at the same time, because students would not be able to visit all of the courses they have to in a given term. Furthermore, the planners should try to allow every student to choose freely among courses which are "required elective".

2. **Room distribution of the administration**: There is a contingent of lecture halls and seminar rooms assigned to each department. Some of these are at a department's sole disposal, others are only available at certain times, because other departments use these facilities as well. If a department requires more resources than it is given to, then it has to request them from the administration of the university. Regarding the room situation, getting additional rooms is often very difficult – if not hopeless.

3. **Requirements of the lecturers**: Of course, generating a schedule for the university concerns the personal schedule of the lecturers. They are allowed to exclude those time periods from the teaching schedule which they wish to reserve for their research and administrative work. Additionally, the lecturers state what room equipment they need for their courses and how many students they expect to come. This is needed to find a room which is large enough but not too large. For example, it does not seem to be a good idea to place a seminar for about 20 students in a lecture hall with 500 seats. But sometimes the planners have no other choice because of the lack of adequate rooms.

4. **Interests of the students**: The students are the only group that do not *dictate* any restrictions to the schedule. They only have *interests* which the planners should try to take into consideration. Unfortunately, these interests often contradict the requirements of the lecturers. It is up to the planners to achieve a result that possibly satisfies both groups. For example, the planners should avoid creating too much "free time" between courses, because this time often cannot be spent for useful activities. Also courses in the evening after 6 p.m. should be avoided, because concentration is bad at this time.

2.3 Prototype Objectives

We can state the following major objectives for our project: We want to implement a prototype for efficient and comfortable scheduling in our own department. In addition, we expect a significant improvement of the schedules' quality and an acceleration of the generation process. Further, the software will be designed to be adaptable to the requirements of other departments. The application will consist of a graphical user interface and an interface to a relational database where all data will be stored.

3 Design Results

In the following we distinguish between the terms *restriction* and *constraint*. We will use restriction in the context of our problem domain and constraint in the context of the Solver algorithm.

As a result of the problem analysis, we organised the various limitations in requirement classes which we call *restriction levels*. A restriction level is just a container for restrictions. Generally, a restriction level contains only those restrictions which are based on the same idea, so it should be possible to describe a restriction level with a half-sentence. For instance, all restrictions

of the type "no concurrence of courses of the same lecturer" are put on one restriction level. Each restriction of this level consists of all the courses of a lecturer, and the level contains as many restrictions as there are lecturers. Another restriction level comprises the time preferences of each lecturer, i.e. times at which a lecturer prefers to teach.

Restrictions can be divided into two groups which we call *hard* and *soft* restrictions. Hard restrictions are the minimum requirements which have to be fulfilled, otherwise it will be impossible to generate a *consistent* timetable. On the other hand, soft restrictions are *individual* requirements which should be satisfied to make the resulting schedule as *acceptable* as possible.

Examples of restriction levels with soft restrictions are: Considering lecturers' requested capacity (number of seats) and equipment of room, or paying attention to times excluded by lecturers who are not able to teach at these times, or taking their preferred times into consideration. This is only a selection of more than 20 soft restriction levels we have found. All hard restrictions are merged in one restriction level: They are of the types "no concurrence of courses of the same lecturer" and "no concurrence of compulsory courses".

Because of the number, not all restrictions can be taken into account. They are ordered by their (subjective) importance. In order to get a representative scaling, we have asked the lecturers and the board of

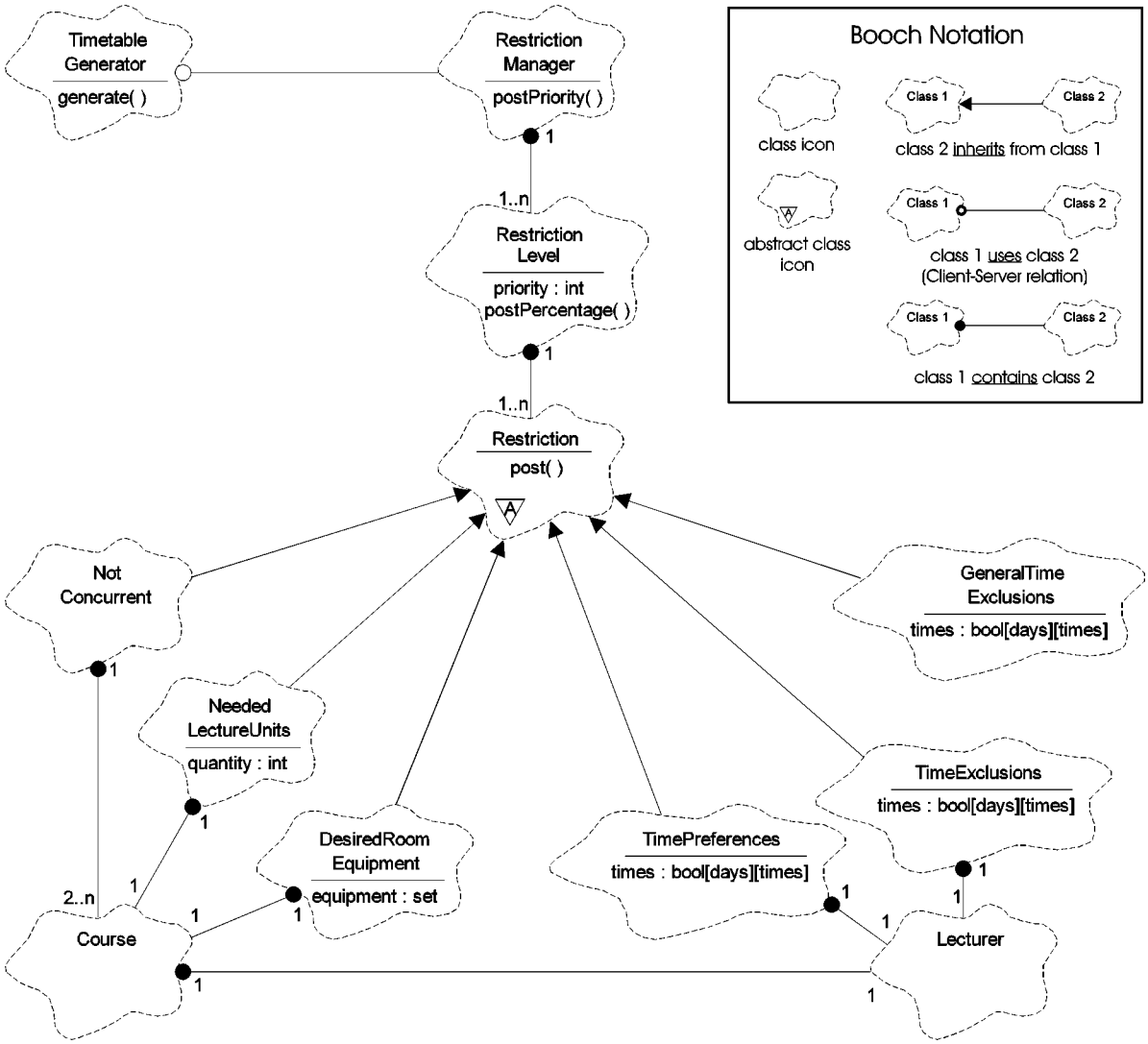

Figure 1: Class Diagram of restriction hierarchy

students to state their individual priorities in a list of restrictions. As a result, we have arranged the restrictions into levels ordered by priority. The priority 0 is assigned to the level of hard restrictions. Thus, the objective of the scheduling algorithm is to satisfy as many restriction levels as possible.

Restrictions of a similar meaning can be abstracted to a restriction *class*. For example, consider the hard restrictions: Both possible types can be abstracted to "no concurrence of courses". So we created the class "NotConcurrent" which is actually the most important restriction class. It allows to mark several courses, which must not take place at the same time. This class can be used for various reasons, e.g. to ensure that a student can visit his or her compulsory courses, or – to state a second example – to prevent a lecture and its associated exercises from taking place at the same time.

Figure 1 shows an extract of the Booch class diagram as a result of the analysis and design process. The diagram was reduced to classes related to restrictions and thus relevant to the Solver. As you can see easily, all restriction classes are derived from the *abstract* base class "Restriction". This base class only consists of the *virtual* operation "post" that has to be overwritten in every class derived from it. It makes sure that an object of a restriction class can convert its data members (describing a restriction) to constraints, so that the Solver understands them.

As mentioned above, an object of the class "RestrictionLevel" is a container of restrictions, i.e. objects of classes derived from the class "Restriction". The attribute "priority" will be initialised with the value assigned to that restriction level in the ordering process. The operation "postPercentage" with its integer parameter "percent" posts the given percentage of all restrictions contained in the restriction level. The selection of the restrictions does not occur by chance. Restrictions are internally ordered in a pseudo-random manner with the effect that each restriction selected by a prior percentage is forced to be re-selected by the next increased percentage. If all restrictions of a level should be activated in Solver then the operation "postPercentage(100)" has to be called.

At a whole, the restriction levels are collected in a single object of the class "RestrictionManager" which can be accessed by an object of the class "TimetableGenerator".

The first class derived from the class "Restriction" is "GeneralTimeExclusions". It is extended by the attribute "times" being a container, e.g. an array of 5 (work days per week) x 6 (time slots per day) Boolean values. Each value set to "true" represents a *time slot* which is not available for the planning process. Objects of this class can be used for example to avoid "late-night" courses beginning at 6 p.m. in the early evening, a time when most students' concentration has decreased rapidly.

The next class derived from the base class is very similar to the previous one. In contrast, however, "TimeExclusions" refers to a lecturer. This means that only the courses taught by the person must not take place at the given times. At the University of Siegen any lecturer is allowed to exclude a whole day or the equivalent number of time slots from the planning process. If a lecturer has to attend a committee on a regular basis, for example, objects of this class can be used to prevent the lecturer's courses from taking place at the time stated.

In contrast to "TimeExclusions", the class "TimePreferences" marks times to be *preferred* in the planning process for the referenced lecturer. Like any other person, a lecturer prefers certain times of a day, e.g. he or she dislikes getting up too early in the morning or wants to be back home by 2 p.m. at the latest. Objects of this class can be used to mark certain times so that the Solver tries to place the given lecturer's courses at the marked times first.

The following class "NeededRoomEquipment" does not refer directly to a lec-

turer like the previous classes. Instead, it refers to a course and, thus, indirectly to the respective lecturer. The attribute is a set of desired room equipment like, for example, overhead projector, video-recorder, or microphone.

The class "NeededLectureUnits" is used in case a lecturer needs a minimum number of time slots (represented by the attribute "quantity") for his or her course. Then, an object of the class can be used to ensure that the course will be put on a day that occurs at least the given number of times in the current term. Let us have a look at a (German) calendar to get the idea: You will see that there are quite a lot of public holidays and no one will go to university on these days (even the lecturers won't). A German "summer term" normally lasts 13 weeks and on Thursdays there can be up to three public holidays.

There are more than a dozen constraint classes not mentioned in Figure 1. They are not described here because some of them are specific to timetable generation at the University of Siegen, or they are not so important, e.g. a class considering the distance between buildings on the university campus (so that one does not have to walk far between different courses), or others enforce a given sequence of courses (lecture before exercises). And finally, there are classes which just hold initialisation values for the Solver, for example, parts of last term's timetable to keep some continuity in scheduling.

Let us have a closer look at the design of the class "TimetableGenerator" (see Figure 2). When the timetable generator gets the message to *generate*, it changes its state from "Idle" to "Satisfying hard restrictions". In this state, the generator tries to satisfy *all* hard restrictions. If a solution of these restrictions could not be found, the generation has failed because all hard restrictions have to be satisfied as they are the minimum requirement of any *consistent* timetable.

With all hard restrictions satisfied, the generator proceeds to the state "Satisfying soft restrictions". There, the objective is to satisfy as many restrictions as possible of each restriction level. With this method, we receive the most acceptable timetable when all restrictions of all levels are satisfied. However, these considerations are rather theoretical. The generator will leave this state always successfully meaning that a consistent timetable could be generated as the hard restrictions could be satisfied at least.

Figure 3 shows a rough implementation of the timetable generator's operation "generate()". In this pseudo C++ source code, we give an insight into the realisation of our concept of priority-driven restrictions or, say, priority-driven constraints. To get an overview about the main idea, the code has not been optimised. As nobody knows how long it takes the Solver to terminate with a *non*-solution, the goal applied to ILCSOLVE

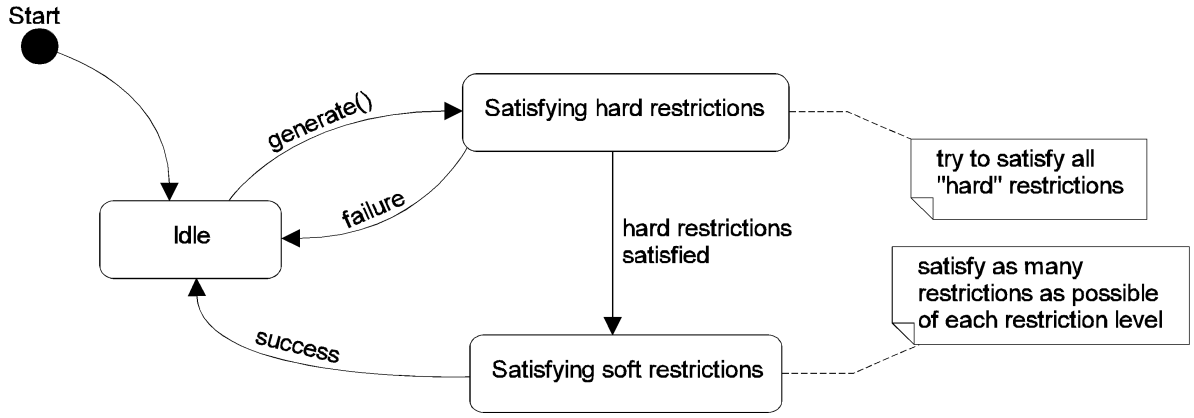

Figure 2: State Diagram of class "TimetableGenerator"

has to be extended considering a time-out based on empirical time measurements.

4 Solver Model

For timetabling we consider only the period of five work days and one term. At university, each course usually takes two hours. Therefore each work day is divided into *time slots* with a duration of two hours each. We further use the term *resource unit* meaning a certain time slot for a given room.

At first, we calculate the number of resource units, i.e. the number of free time slots in all given rooms. Then we initialise the constant integer arrays TIME and ROOM. TIME holds the time identifier and ROOM the room identifier of each resource unit. The first time slot, i.e. Monday 8 a.m. to 10 a.m., is associated with the integer 0, the second one, i.e. Monday 10 a.m. to 12 a.m., with 1, and so on until Friday 6 p.m. to 8 p.m. which is associated with 29.

A course can take place at one of these resource units. So all the courses to be planned are modelled as an array COURSE of constrained integer variables. The domain of each constrained variable ranges from 1 to the number of resource units. We post a constraint applying the Solver function ILCALLDIFF to the array COURSE with the result that at a single resource unit just one course can take place. In other words: two or more courses must not take place at the same time in the same room.

Let us now take a look at the detailed design of a restriction class. The most important restric-

```
int TimetableGenerator::generate() {
    // array percentage holds the satisfaction rate of each restriction level
    int percentage[MAXPRIORITY+1];

    InitSolver();

    // post 100% of the restrictions of restriction level 0 (hard restrictions)
    RestrictionManager.post(0,100);

    // generate solution
    if( not IlcSolve(...) )      // no solution found
        return FAIL;             // hard restrictions must be satisfied

    // solution was found, all hard restrictions satisfied,
    // i.e. 100% of restrictions of restriction level 0 are completely satisfied
    percentage[0] = 100;

    BackupSolution(...);

    // satisfy as many soft restrictions as possible of each restriction level
    for(int priority = 1; priority <= MAXPRIORITY; priority++) {

        ResetSolver();

        // post restriction levels up to priority
        for(int level = 0; level < priority; level++)
            RestrictionManager.post(level, percentage[level]);

        // assume that 100% of current restriction level could be satisfied,
        // so post restrictions of whole level
        RestrictionManager.post(priority, 100);

        if( IlcSolve(...) ) {    // the assumption was right ...
            BackupSolution(...);
            percentage[priority] = 100;
        }
        else {                   // wrong assumption ...
            ResetSolver();

            // post all restrictions which already have been satisfied
            // i.e. all previous restriction levels with their individual
            // satisfaction rates
            for(int level = 0; level < priority; level++)
                RestrictionManager.post(level, percentage[level]);

            // try to satisfy as many restrictions as possible of the current
            // restriction level. 100% of the restrictions cannot be satisfied
            int percent = 0;

            do {                 // STEP is constant integer between 1 and 100
                percent += STEP;
                // post additional restrictions
                RestrictionManager.post(priority, percent);
                successful = IlcSolve(...);
                if(successful) BackupSolution(...);

            } while(successful);

            // save percentage of satisfied restrictions of the current
            // restriction level one step before Solver failed
            percentage[priority] = percent-STEP;
        }
        EndSolver();
    }

    Timetable = RestoreSolution();
    return SUCCESS;
}
```

Figure 3: Implementing priority-driven constraints

tion class "NotConcurrent" works as follows:

An object of this class contains n courses, pairs of which must not take place at the same time. The respective indices of the courses in the array COURSE are i_1 to i_n. We dynamically create an array NOTCONCURRENT of constrained integer variables and put the integers TIME[COURSE[i_1]] to TIME[COURSE[i_n]] in this array. Then we post the constraint ILCALLDIFF(NOTCONCURRENT) with the effect that two or more of the n courses do not take place at the same time. Note that at this position in the code the terms COURSE[i_j], $j = 1,..,n$ consist of a domain of resource units and not of a single one.

Unfortunately, the model is not sufficient for the real world. We simplified the model to explain the general idea. In real university life, there are courses with an average length of one hour per week over the whole term, i.e. these courses take place every two weeks for two hours.

A straightforward extension of the model considering fortnight courses has the consequence that same times in both weeks would be associated with different integers, e.g. Monday 8 a.m. to 10 a.m. would be associated with 0 in the first week and 30 in the second one. Then, the elegant implementation of the class "NotConcurrent" would fail. Take, for example, two courses A and B that should not take place concurrently. Assuming they are assigned to resource units with respective time slots of 0 and 30, this would result into both courses happening on Monday 8 a.m. to 10 a.m., and this clearly indicates concurrency!

Therefore, we decided to extend our model by setting a time slot to the length of one hour. Courses with a normal duration of two hours are split into two parts, where each one can take place at a single time slot. Constraints are posted so that the first part has to take place in an even time slot (8 a.m., 10 a.m., etc.) and that the second part has to follow the first one immediately. Courses with only two hours a fortnight are handled as one part. If such a course is placed in an even time slot, the course takes place in an even calendar week, an odd time slot indicates an odd calendar week.

5 Conclusions

We were very quickly acquainted with the Solver writing successfully some CSP examples. What is striking is the very readable and compact code due to the power of the function ILCPOST (see the design of the restriction class "NotConcurrent"). As there is a clear separation between the problem statement, i.e. decision variables and the constraints imposed, and the resolution of the problem, i.e. constraint propagation, tree traversal, and backtracking, an *initial* model can be extended easily. Thus, the Solver has proved to be very useful for *rapid prototyping*.

At the University of Siegen, we are not the first group attempting to solve the scheduling problem. Many have tried but failed. With the Solver as part of our prototype, we have already achieved the same results as the conventional planners of our department (mostly considering only hard restrictions), but in significantly less time. On an Intel Pentium (100 MHz) PC with 96 MB RAM, our prototype found a solution within some minutes compared to the manual scheduling work of three human planners taking at least two weeks.

The long-term aim of the project is a university-wide version which will be developed in a diploma thesis at the end of 1997. The final version will be a *client-server* application using an object-oriented database management system to satisfy the additional safety and integrity requirements for the stored data.

Acknowledgement: We would like to thank the anonymous reviewers for their helpful suggestions and comments.

References

[1] Hentenryck, van, Pascal, *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
[2] ILOG, *Ilog Solver Reference Manual, Version 3.1*, 1996.
[3] Booch, Grady, *Object-Oriented Analysis and Design*. Benjamin/Cummings, 2nd ed., 1994.