University Timetabling in ECL^{i}PS^e

Marco Gavanelli

Abstract

This paper describes how the university timetabling problem is addressed in the Laurea course *Ingegneria dell'Informazione* (Information Engineering) for the University of Ferrara, Italy.

The university timetabling problem is modelled as a Constraint Optimisation Problem and addressed with ECL^{i}PS^e, one of the leading Constraint Logic Programming languages.

1 Introduction

A timetabling problem consists in defining a schedule for the meeting of students and lecturers, such that a set of constraints is satisfied. The kernel of the problem is fixed: in all universities constraints impose that a professor will not teach two different lessons at the same time, that a room can only contain a lesson at a time, that a group of students cannot attend two different lessons at the same time. This problem has been addressed in many works in the literature, with different techniques, including constraint programming [4, 18, 8, 16], local search [24, 14], or integer programming [26]. Nonetheless, due to peculiar organization issues in each university, in every faculty a different software tends to be developed [4].

Even though the mathematical formulation of the problem is clear and well assessed, it addresses typically only one of the issues of the problem, namely the generation of a timetable. Actually, the problem consists of a much wider problem, which is not well formalised, and includes interaction with the professors, the head of the department, the other developers of timetabling for the other faculties that might share courses, students, classrooms, and/or professors with the current one.

In this paper, we report how the timetabling problem has been solved in the course *Ingegneria dell'Informazione* (Information Engineering) in Ferrara University. We define how the problem was defined in the Constraint Logic Programming (CLP) [20] language ECL^{i}PS^e. The program was first used in 2004, and every year it was improved in order to address more issues. We do not only provide the CLP model that solves the problem, but we also report the practical problems that arise in the construction of the timetable, and how they are addressed (sometimes satisfactorily, sometimes not) with CLP(FD). We show strength and weakness of the present approach. We show how the data was represented in ECL^{i}PS^e, what are the phases to build a timetable, starting from scratch and arriving to a final timetable, after interaction with the various actors that will, in the end, use the timetable.

The rest of the paper is organised as follows. We first introduce some preliminaries, then state the requirements of the timetable in our Laurea course. We show how the data was organised in ECL^{i}PS^e in Section 4, and the constraint model in Section 5. We give some evaluation of the work in Section 6, review some of the related work in Section 7, and finally, we conclude.

2 Preliminaries

Definition 1 *A Constraint Satisfaction Problem (CSP) is formally a tuple (V, D, C) where V is a finite set of variables $X_1, \dots, X_n$ ranging on domains of objects of arbitrary type $D_1, \dots, D_n$. We call D_i the domain of X_i. C is a finite set of constraints on variables in V. A constraint c on variables X_i and X_j, i.e., $c(X_i, X_j)$, defines the subset of the Cartesian product of variable domains $D_i \times D_j$ of the consistent assignments of values to variables. A* solution *to a CSP is an assignment of values to variables which satisfies all the constraints. A constraint c on variables X_i, X_j is* satisfied *by a pair of values v_i, v_j if $(v_i, v_j) \in c(X_i, X_j)$.*

In many real-life problems, solutions are not all equally good, but some are preferable to others. One way to state which solutions are better is by means of an objective function.

Definition 2 *A* Constraint Optimization Problem (COP) *is a quadruple $(P, f, T, \leq)$ such that P is a CSP, $(T, \leq)$ is a totally ordered set and $f : D_1 \times \ldots \times D_n \mapsto T$ is a function that maps each assignment to a value. An* Optimal Solution *is a feasible solution of P that maximizes the function f.*

Constraint Logic Programming (CLP) [20] is a class of languages that embeds in Logic Programming the ability to solve constraints; in particular, CLP on Finite Domains (CLP(FD)), is particularly well-suited for solving combinatorial problems. Many logic programming systems include libraries for CLP(FD), e.g., CHIP [11], ECL^{i}PS^e [2], SICStus [1, 10], Oz [30].

3 User's Requirements

Besides the usual requirements in timetabling (two lessons cannot share a room, a professor cannot teach two lessons in parallel, students cannot attend two different lessons at the same time), and the professor's and student's preferences, that should be taken into account, we have some requirements given by how the lessons are organised in Italy, plus more specific requirements given by the coordinator of the Laurea Course.

University courses in Italy are of two types: a first degree (*Laurea* degree) is obtained after three years, a second degree after two more years (*Laurea*

	Monday	Tuesday	Wednesday	Thursday	Friday
8.30 - 9.30	A	B	A	A	D
9.30 - 10.30	A	B	A	A	D
10.30 - 11.30	A	B	C	B	G
11.30 - 12.30	B	C	C	B	G
12.30 - 13.30	B	C	C	D	F
14 - 15	C	F	G	D	F
15 - 16	C	F	G	G	E
16 - 17	D	F	G	G	E
17 - 18	D	E	E	F	E
18 - 19	D	E	E	F	

Table 1: Pre-defined patterns for the timetabling

Magistrale or *Laurea Specialistica*). In Ferrara, the Laurea Course *Ingegneria dell'Informazione* has four curricula: Automation, Electronics, Informatics, and Telecommunications. Students have many obligatory courses in the first three years, and some choices. After the first degree, they can continue with the second degree. The second degree has some obligatory courses, plus many choices. Students can continue with the Laurea Magistrale in a curriculum different from the one they took in the first level. In such a case, they will have less choices: they will have to choose the basic courses they did not take in the Laurea. For example, if a student moves from Electronics to Informatics, he/she will probably have not given some exams that are basic in computer science (like Databases, or Operative Systems), but not in Electronics; so in the Laurea Magistrale he/she will have to choose those exams.

Further requirements are given by the responsible of the Laurea Course. All the lessons should be scheduled in a pre-fixed pattern, shown in Table 1. The reason for having such patterns is that the students may choose the optional courses among a plethora; the possible choices are more than the maximum number of courses we can have in parallel. Since lessons are scheduled in 5 days, with 10 hours per day, and each course has typically 7 hours per week, the maximum number of non-overlapping courses in parallel is $\left\lfloor \frac{5\times10}{7} \right\rfloor = 7$. The set of courses that students can choose from (in each period) is wider, so overlapping is unavoidable.

The solution to such issue is to provide the timetable to the students *before* they choose the optional courses, so they can choose non overlapping courses. With such an organization, the fact that two courses either do not overlap or overlap completely can indeed help students in formulating their choice. Also, the *pattern* constraint reduces drastically the size of the search space, and provides a very convenient explanation in (the likely) case of complaints by the professors.

Moreover, since first and second year students have only basic (thus, compulsory) courses, lessons of the first year should be scheduled in patterns A, B, and C, which makes the timetable for first year students very compact (almost always in the morning, with Friday as a free-day); lessons of the second year are to be scheduled in patterns E, F and G, again very compact. This division of patterns makes the timetables of the first two years non-overlapping, which lets the students of the second year able to attend again the lessons of the first year (in Italian university, a student can enrol in second year even if he/she has not passed all the exams of the first year, and the number of students that fail to keep the pace with the exams is considerable). Also, in this way the students of both years use the same room, typically the biggest available.

On the other hand, such a constraint (besides being deprecated by the professors whose preferences are unsatisfied) is often infeasible (i.e., contradicted by hard constraints). For example, it is often violated by courses in common with other faculties. Some of the courses are held in a nearby town (Cento di Ferrara), and it is necessary to avoid students moving from one town to the other in a same day. Some of the professors have lessons in other faculties, or even other universities, and some have a professional job outside the university. Finally, in some periods students of the first two years have more than three courses in parallel.

Although stated as a hard constraint by the user, the *patterns* constraint cannot be implemented as a hard constraint for all courses. This is a common problem in software engineering: the user's requirements are often impossible to implement as they are stated, but must often be relaxed and defined formally.

The adopted solution consists in declaring some of the courses as *exceptional*: for exceptional courses the *patterns* constraint does not necessarily hold.

4 Data Representation

The timetabling program is divided into modules, that implement the search strategies, user defined constraints, etc. In particular, one of the modules represents the *instance* of the problem. We have an instance for each period. The instance is represented as a Prolog database, typically as a set of Prolog facts. The instance consists of the following predicates:

- course
- room
- professor

plus some further, ancillary predicates for defining other characteristics.

The predicate `course` contains, for each university course, the following information:

- course id
- professor
- number of students

- classes of students
- number of hours per week
- number of lessons per week
- minimal and maximal duration of a lesson (in hours)
- special requirements (laboratories, projectors, etc.)
- full name, URL of the web page of the course.

Some of the information is used to find a solution (i.e., to provide the COP model), while other (like, full name of the course, URL) is used for the visualisation of the final timetable on the web.

The `room` predicate provides information about the rooms that are available for the lessons in the university course. It includes information about the capacity, special resources available in the room (computers, laboratories, projector, etc.).

Of course, information is not static, and new items might be added from one year to the following. For this reason, a simple encoding as the previous might prove insufficient: a concept of *record* or *structure* is necessary for practical applications also in Logic Programming, not only in imperative and object-oriented programming. A first step stands in accessing the elements by means of the `arg/3` predicate: the n-th element of the term T can be accessed with $\texttt{arg}(n, T, X)$. But one wants also to add a name to fields. The solution proposed in ECL^{i}PS^e is based on preprocessing and macro expansion. The user can define a so-called *structure* with its fields:

$$struct(course(id, professor, n\_students, classes, hours, lessons,\\ min\_dur, max\_dur, req, full\_name, url)).$$

After such declaration, one can create a term with the right number of arguments with the macro `with`; e.g., the macro

$$\texttt{course with } [\texttt{professor}:\texttt{P}, \texttt{hours}:7]$$

is expanded to

$$\texttt{course}(\_, \texttt{P}, \_, \_, \_, 7, \_, \_, \_, \_, \_).$$

In this way, adding new fields is much easier, as it amounts to minor changes.

The `professor` predicate is defined by a set of rules, each containing the constraints of the professor. The `professor` predicate is invoked to impose the constraints. Moreover, the information it contains can also be accessed and reasoned about; this ability of Logic Programming of reasoning about programs will be useful in Section 5.2.

	Monday	Tuesday	Wednesday	Thursday	Friday
8.30 - 9.30	1	13	25	37	49
9.30 - 10.30	2	14	26	38	50
10.30 - 11.30	3	15	27	39	51
11.30 - 12.30	4	16	28	40	52
12.30 - 13.30	5	17	29	41	53
13.30-14	6	18	30	42	54
14 - 15	7	19	31	43	55
15 - 16	8	20	32	44	56
16 - 17	9	21	33	45	57
17 - 18	10	22	34	46	58
18 - 19	11	23	35	47	59

Table 2: Possible values of domains

5 Constraint Model

The problem can be modelled as follows. Variables represent the start times of all the lessons and the room assignment. The basic constraints are no-overlapping amongst all the lessons that are taught by the same professor, that are attended by a same group of students, or that share the same room. Values in the domains of the start times are integer, as represented in Table 2. Notice that there are values reserved for the lunch break (values $\{6+12\cdot X|X \in 0...4\}$), and for the break between each day and the following (values $\{12\cdot X|X \in 0...4\}$). This simplifies stating constraints such that *no lesson should start one day and finish the following*: it is enough to define a virtual lesson that is taken every day in the *gap* between days and impose that no lesson can overlap with such virtual lesson.

Beside these basic constraint, and the preferences and constraints of the professors, redundant constraints can be used to improve the pruning. A simple constraint, widely used in scheduling applications, is the `cumulative` constraint. Such constraint involves a set of activities, described with their start times, duration and resource needs; the total set of resources R is given as input, and the `cumulative` constraint imposes that the activities that overlap in any instant of time do not consume more than R resources. A simple model considers the available rooms as resources: in this way the number of rooms required in any time point is never exceeded. This constraint, however, does not ensure that the rooms will be appropriate for the lessons: the rooms could lack some of the features required by some of the lessons, or they could be too small for the students to fit in. The first problem can be simply solved by imposing a `cumulative` constraint for each of the features required for the lessons. For example, all the lessons requiring a computer lab are collected in a list L_{lab}; let S_{lab} be the list of the corresponding start times and D_{lab} that of the durations. Suppose the faculty owns N_{lab} computer laboratories, we can impose the constraint

$$cumulative(S_{lab}, D_{lab}, 1, N_{lab}).$$

Concerning the capacity of the rooms, one can impose, for each of the available room's capacity, the cumulative constraint for all the lessons that can fit in the room. Let $R_{\leq C}$ the set of available rooms with capacity up to C; let $L_{\leq C}$ the set of lessons that can fit in a room of capacity C, we can impose, for each available C the constraint

$$\mathit{cumulative}(S_{\leq C}, D_{\leq C}, 1, |R_{\leq C}|). \tag{1}$$

The number of such constraints is equal to the number of available rooms, at most.

The `pattern` constraint shown in Table 1 was implemented with the Propia library of ECL^{i}PS^e [32]. Propia is a very high level way of implementing constraints: any predicate (even recursive ones) can be transformed into a constraint. So, in our case, we only needed to implement `pattern` as a predicate:

```
pattern(a,25,37,1).
pattern(b,4,39,13).
pattern(c,7,16,27).
pattern(d,41,49,9).
pattern(e,22,34,56).
pattern(f,46,53,19).
pattern(g,44,51,31).
```

Invoking the goal `pattern(P,T1,T2,T3) infers fd` inserts a corresponding constraint in the constraint store; the domains of the variables are taken as the Least General Generalisation of the allowed values. For instance, if P has $\{c, d\}$ as domain, then the domain of $T1$ is restricted to $\{7, 41\}$ (in this case, the LGG is simply the union of the values selected by the clauses).

5.1 Objective function

Since many professors have hard constraints (like the need to teach courses in other faculties), the objective function tries to improve the timetable with respect to students' needs. Another possible viewpoint would be to create an objective function that tries to improve the professor's acceptance of the timetable, e.g., by maximising the preferences of the professors. The two objective functions could then be combined with one of the aggregation functions proposed in the literature of multi-criteria optimisation (like weighted sum [31], minimisation of the maximum, lexicographic orderings, Choquet/Vitali integral [19], etc.). These solutions have currently been avoided due to the need to arbitrarily state coefficients, and normalisation factors, that are prone to criticisms from the professors whose preferences are not satisfied. They will be nevertheless considered in future extensions of the framework.

The current implementation tries to make the timetable as compact as possible for the various groups of students. It is a weighted sum: not all groups have the same priority, as students of the first years are many and only a few reach the fifth year. Moreover, optimising hard the compactness for a group of students that have many choices could be meaningless, as non-chosen exams would inevitably introduce holes in the timetable.

For a given group of students, the objective function is computed as follows. For each day k, the quantity $H_{ijk} = min(|S_i - (S_j + D_j)|, |S_j - (S_i + D_i)|)$ represents the *displacement* between lessons i and j: it is roughly the distance between the end of the first lesson and the beginning of the second. H is a candidate for being a 'hole', i.e., a time slot between two lessons that is not a lesson as well. Holes should be minimised, as students typically prefer to come to faculty as late as possible, and go back home as early as possible. Summing the contributes H_{ijk} does not give exactly the number of 'holes', but is experimentally a good compromise. In fact, if we simply minimised the number of holes, the optimal solution would often contain very full days: a day in which every possible slot is occupied by a lesson does not have holes, so it is optimal; however it is very tiring for students, so it would be a bad choice. Since $\sum_{ij} H_{ijk}$ is not null for a full day, it avoids days with too many lessons, and provides a compact timetable.

Moreover, a *prize* is given for each free day: a day without lessons is highly appreciated, in particular by students living far from the campus. We empirically found that value -5 for free-day prise provides a good trade-off between very high prises (that tend to distribute the lessons in few very intensive days) and low prises (in which lessons tend to be distributed in the whole week).

Finally, since most of the lessons should respect the *pattern* constraint, it is easy to pre-compute bounds of the quantity $\sum_{ij} H_{ijk}$, that will be useful in the Branch-and-Bound search. For instance, if in a given period some group of students can attend 3 different courses, and none of these courses is *exceptional*, then we can pre-compute the best possible values of $\sum_{ij} H_{ijk}$ and use it as a bound.

5.2 Symmetries

It is well-known in constraint logic programming that symmetries introduce inefficiencies in the search; symmetries enlarge (often, exponentially) the search space and can make simple problems incredibly hard. Many techniques have been proposed to improve symmetry handling in constraint programming; typically those techniques can be divided in two main branches: symmetry breaking during search [15, 12, 13, 25], and symmetry breaking constraints [23].

The basic timetabling problem is highly symmetric: it is based on the `cumulative` constraint that is intrinsically symmetric. For example, given a feasible solution of `cumulative`, the solution obtained by exchanging two activities is still feasible.

Although the basic problem is symmetric, many symmetries are removed by the so-called side constraints: e.g., professors have often different preferences, rooms are not equivalent (different size, different resources), etc.

Nevertheless, some of the symmetries still remain, and breaking them can make the program faster. Even if only one symmetry is broken the computation time can be halved; this does not significantly change the size of the problems that can be addressed (so it is not a striking result in research: does not allow you to find solutions for open problems), but it is a remarkable improvement in practical problem solving (waiting one minute or two *is* practically different).

The idea is then to find the symmetries that are hidden in the definition of the problem instance. There are works [7] proposing to find symmetries in the problem model (disregarding the instance), by using a theorem prover. In this work, instead, we consider as given the fact that the problem contains symmetries, and try to discover if some of the symmetries are not broken by the side constraints in the particular instance.

Obviously, two lessons of the same duration in the same course are interchangeable: exchanging, during search, the two start times of the lessons would be a waste of computation time. So, we can impose that one should precede the other. But, we cannot do it if the two lessons are not indistinguishable: for example, if they have different room requirements (many professors do some of the lessons in the lab and some in a normal room). In the end, if the specifications are *identical* for two lessons of the same course, they are symmetrical and can be ordered.

In the same way, if two courses have *identical* specifications they are symmetrical. Identical specifications means that they are attended by the same students, have identical requirements, are taught by the same professor. Of course, this does not happen very frequently (actually, it never happened in the instances studied in this work). But, specifications need not to be exactly identical: for example, two courses might have different professors (all rest being identical). If the the professors have identical preferences and do not teach in other courses in the same period, we can identify the two courses as symmetrical. Although this seems quite an uncommon situation, it happened in all three periods, and allowed for not negligible time save. Moreover, the symmetries found in this way were not spotted in analysing by hand the specifications: they were actually discovered by the program.

In this case, the Logic Programming implementation is very helpful: in LP data and programs are of the same nature, so it is very easy to analyse the specifications of two professors (which are given as clauses, i.e., programs) and find out they are identical.

Of course, this symmetry discovery technique is very basic, but, as stated earlier, it was practically useful. In future extensions, we plan to improve this discovery to find other types of symmetries. For example, there are courses that have a same purpose in different groups of students. The *English* course is taught in first year to all students, but in groups depending on the Laurea course. The same professor gives (separately) lessons to the first year students of *Informatics and Automation* and *Electronics and Telecommunications*. These two groups of students have all the other courses in common. So, the two *English* courses are indeed symmetrical, but this symmetry was not discovered.

5.3 Search strategies

Different search strategies are necessary for different steps in the creation of the timetable.

A first step is the creation of the first timetable, from scratch. The timetable is published on the web and submitted to the professors for approval.

The professors suggest modifications, and the second step stands in creating an improved versions after the previous. Each modification must be resubmitted for approval, and many professors do not pay attention to the new releases after the third-fourth (and tend to complain after the deadline for publishing the final timetable has passed). So, a minimization of this interaction is necessary.

When the system reaches quiescence, and the deadline for publishing the timetable is reached, the timetable becomes official. In this third phase, some small modifications may still be necessary, due to wrong estimation of the number of students (that may not fit in the assigned room), professors that got bored of looking at the web page after the first two versions and complain for their timetable after the final deadline, and other unforeseen events. So, in the final version there might be small modifications, but they must be kept as small as possible: optimising the objective function given in Section 5.1 is no longer a requirement.

For these reasons, in the first phase, we optimise the proposed objective function (Section 5.1). In the second we still try to optimise the same function, but possibly introducing as small modifications as possible: in this case, reaching the global optimum is no longer necessary. In the third phase, the function to be optimised is the distance from previous solution: we try to minimise the modifications.

In all phases, the search is decomposed in two parts: labelling of the start times, and labelling of the rooms. The first is the only one that involves the objective function (currently, the objective function does not involve room assignment). This means that in the Branch-and-Bound procedure, there is no point in finding alternative room assignments: after the first room assignment is found, we can backtrack directly to the time assignment. Moreover, given the constraints 1, the room assignment is backtrack-free. Thus, we use a simple heuristics for the room assignment: for each time slot, we select all lessons scheduled in that time, we sort them according to the number of students that will attend that lesson, and assign the biggest room to the biggest course, and the others following. In this way, the best resources are always employed, while small rooms are typically empty and can be used by students as studying rooms.

Phase 1: Finding a solution for the first time In order to find a suitable search strategy, it is useful to picture the constraint graph. The full constraint graph is obviously fully connected: this can be easily seen by considering that all the lessons can be given in the biggest room, so any pair of variables in the room assignment is connected by an edge. However, if we relax the room constraint we get a loosely connected graph; one of the instances is given in Figure 1, where arrows represent constraints of non-overlapping between lessons.

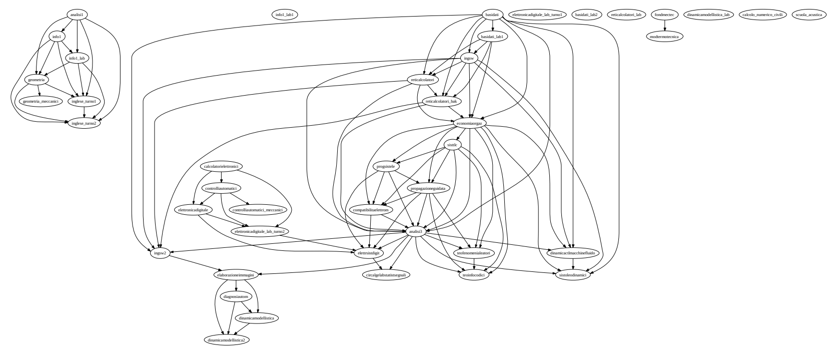

Figure 1: Time assignment graph

Although this structure might seem peculiar to a particular instance, a similar structure is replicated in the other periods. From the graph we notice that some of the lessons are isolated: some of these lessons are virtual, and were introduced to model room unavailability, some are indeed lessons that have no constraints with respect to other lessons. There is a clique on the left: these courses are first-year obligatory courses. Since they are obligatory, all of them must be taken, so they form a clique; since they must be taken in first year there is no need to avoid overlaps with courses of the following years. Finally, there is a big connected group that contains all the remaining courses; amongst these, we can see cliques that correspond to courses of similar subjects (students in Informatics should be allowed to attend all possible courses in computer science, and so on). These cliques are connected among them by some constraints.

Since the problem is structured, exploiting its structure may provide significant improvement. A simple idea would be to solve to optimality one connected subgraph, take the optimal assignment for this subset of the variables, then move to another connected sub-graph, taking as input the optimal assignment of the first, and so on. Of course, there is no insurance that the found solution is indeed optimal, but this solution is indeed very simple and fast, so it may be useful in practice, when optimality is not important.

This strategy could be transformed into a complete strategy imposing precedence constraints between lessons scheduled in parallel, as in probe backtracking [28, 22] or in scheduling [27].

Phase 2: Repairing a solution and optimising As explained earlier, after receiving feedbacks from the professors, new solutions should be proposed. The new solution should be nearly optimal, but should also introduce few modifications from the previous version.

We adopted an algorithm based on Limited Discrepancy Search [17]: we first search for solutions with discrepancy $d = 1$ from the previous solution. If a new solution S^* is found, this is the new candidate solution; we continue the search for better solutions with discrepancy 1 from S^*. Otherwise, if no solution is found with discrepancy d, we try with discrepancy $d+1$ from the reference solution.

This simple algorithm is complete, and very fast in finding good solutions. It also straightforward to implement in CLP.

The room assignment is again performed with the previous heuristics: each course is assigned the best possible room.

Phase 3: Repairing with minimal deviation Finally, in third phase we do not try any longer to optimize the objective function: we simply try to repair the last solution with minimal modifications; as soon a feasible solution is found, the search stops, even if there exist better solutions. The room assignment tries to follow as closely as possible the previous one, in order to minimise the probability of professor's complaints.

5.4 Explanations

One difficulty in using constraint programming stands in the fact that if there is no solution, the only answer the user gets is *no.* There is no explanation about *why* the search failed. This is a serious problem when dealing with people: the answer *no* is not enough for explaining a professor that he cannot get his favourite time schedule.

An explanation for a failure is a set of constraints that cannot be satisfied together; of course such set should be as small as possible, in order to be easily understandable. We adopted a simple implementation, that does not require to rewrite previous code, or rearrange the order in which constraints are imposed, although it does not always provide a good (minimal) explanation.

The explanations we are looking for typically involve some given set of variables (e.g., the lessons of a given professor, that is asking why he cannot teach on Monday mornings).

We simply implemented a predicate `signal_removal/1` that is awaken by the same scheduling engine of constraints and is considered by ECL^{i}PS^e as a constraint (although it does not perform propagation). Such a *"fake constraint"* has a very high priority, so it is activated as soon as the domain of the variable it contains is changed. Whenever activated, prints on screen the domain of the variable is involves, together with the cause that lead to such domain reduction. The cause is the current set of branching constraints, i.e., the set of constraints that identify the current branch of the search tree. So, in case of failure the user gets a sequence of domain reductions, together with their causes: the union of all causes can give a hint about the cause of the failure, and communicate to the professor.

Of course, this is a very simple explanation mechanism, but it was practically useful in the timetable generation and was easy to implement. In future releases, we plan to adopt a more complete explanation algorithm, such as QUICKXPLAIN [21].

6 Results

This work was driven by the practical need to provide a university timetable; its scope was not to provide the fastest possible algorithm, but to provide an algorithm that suited the needs of the Laurea course. Our implementation has timing results between few seconds and some minutes (on a Pentium M 715 running at 1.5 GHz), which is a reasonable time to wait. To give an idea of the size of the problem, we have between 30 and 40 courses for each period, and a room availability of 6 general purpose rooms (with different capacities), three computer science laboratories and an electronics laboratory.

We compared the results of the algorithm with those of the previous year, in which the timetable was produced by hand. We were able to collect the information only for two periods (of three) of year 2003.

In the version produced by hand there are no overlaps between obligatory courses but there are 9 overlaps between an obligatory and an optional course. This means that students were not able to attend all the lessons of the optional course. There are 20 overlaps between an obligatory course in Laurea Specialistica (second level) and fundamental courses in Laurea Triennale (first level). This means that students moving from one Laurea Course to a different specialisation were unable to attend all the obligatory courses.

The improvement for students is striking: the number of overlaps since 2004 has fallen to zero. Notice that the comparison is unfair, as the data for 2003 does not cover the whole year (but only two of the three periods), so, for year 2003, the number of overlaps is a lower bound estimate (while it is the precise number for the following years).

Finally, the timetable used to be provided a couple of weeks before the lessons started (in each of the three periods). In the new version, the timetable is produced at the beginning of the year, so the students are able to choose the (optional) courses they want to attend knowing the timetable, thus they are able to avoid overlapping courses.

7 Related work

University timetabling has been the subject of many works in Constraint Programming, and Operations Research. Most of the works focus on the efficiency of the algorithm. In this work, instead, we reported the practical problems besides the responsible for the timetable has to address *besides* the computational complexity.

Schaerf [29] classifies timetabling problems into *School Timetabling*, in which there cannot be overlaps between courses having common students, *Course Timetabling*, in which the aim is to *minimize* the overlaps of courses with common students, and *Examination Timetabling*, in which there cannot be overlaps and the exams should be spread as much as possible. Our work falls in the category of School Timetabling, according to Shaerf. Paper [29] also surveys various solution approaches to the three types of problems.

One of the first works that uses CLP for university timetable is that by Azevedo and Barahona [4]. In this early work, they suggest that an integer programming formulation is not suitable for this type of problem, and model the problem in CLP(FD), with non-overlapping constraints stated as a disjunction of the type $Start_1 + Dur_1 \leq Start_2 \vee Start_2 + Dur_2 \leq Start_1$. Henz and Würtz [18] propose the use of constructive disjunction and reified constraints for college timetabling in Oz. Others define the constraints in CHR [3]. More recent works [8] exploit global constraints, line `alldifferent` or `gcc` (Global Cardinality Constraints). In our work, we used the `cumulative` constraint, as in [16], that is typical to scheduling applications, and can be implemented in various ways; our choice was to use the edge finder propagation [9, 5], that gives high pruning power, at the cost of time spent for constraint propagation.

Many works propose Local Search techniques to address university timetabling problems [24, 14]; local search is typically very fast, and able to solve problems of very big size, but it gives up the proof of optimality, so the user cannot know if the proposed solution is really the optimum.

A recent work by Qualizza and Serafini [26] proposes a Branch-and-Price (Column Generation) approach to the university timetabling problem. The integer programming model contains a column for each possible pattern of lessons a course can have; the columns are exponentially many, but only a few of them are actually generated during search. This approach could be useful also in our instance, in case in the future the patterns (shown in Table 1) will be computed, instead of being fixed and assigned by the head of department.

8 Conclusions

This work reports the use of Constraint Logic Programming in building the university timetable of the Laurea Course of Information Engineering of Ferrara University. CLP has been used for such task since 2003, and the software is currently in a mature stadium. The software provides the timetable in an easy to read, web friendly html format, that can be seen at `www.ing.unife.it/informazione/orario/`. The use of CLP has been extremely valuable for solving this type of problem, that was very tedious to perform by hand, and took several time (typically weeks) of the researcher or professor to whom the work was assigned.

Currently, the most time consuming operation is coordinating with the timetable of the other courses of the faculty and of the University: their timetables are still developed by hand. In the next years, we plan to extend the use of CLP also to the other courses. In this year (2006) the rector gave two deadlines for publishing the timetable: one for a temporary timetable and one for the final version. The course of *Ingegneria dell'Informazione* was the only one able to meet the first deadline. Since then, no modifications have been necessary.

Acknowledgments

This work has been partially supported by the MIUR PRIN 2005 project n. 2005-015491: *'Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems'*.

References

[1] SICStus prolog. http://www.sics.se/sicstus/.

[2] The ECLiPSe constraint programming system, 2006. http://eclipse.crosscoreop.com/eclipse/.

[3] Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.

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

[5] Philippe Baptiste and Claude Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *IJCAI (1)*, pages 600–606, 1995.

[6] Edmund K. Burke and Michael A. Trick, editors. *Practice and Theory of Automated Timetabling V, 5th International Conference, PATAT 2004, Pittsburgh, PA, USA, August 18-20, 2004, Revised Selected Papers*, volume 3616 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] Marco Cadoli and Toni Mancini. Using a theorem prover for reasoning on constraint problems. In Stefania Bandini and Sara Manzoni, editors, *AI*IA 2005: Advances in Artificial Intelligence, 9th Congress of the Italian Association for Artificial Intelligence, Milan, Italy, September 21-23, 2005*, number 3673 in Lecture Notes in Computer Science, pages 38–49. Springer-Verlag, 2005.

[8] Hadrien Cambazard, Fabien Demazeau, Narendra Jussien, and Philippe David. Interactively solving school timetabling problems using extensions of constraint programming. In Burke and Trick [6], pages 190–207.

[9] J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, 26:269–287, 1990.

[10] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and*

Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer Verlag, 1997.

[11] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, and Alexander Herold. The chip system: Constraint handling in prolog. In Ewing L. Lusk and Ross A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*, pages 774–775. Springer Verlag, 1988.

[12] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Walsh [33], pages 93–107.

[13] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Walsh [33], pages 77–92.

[14] Luca Di Gaspero and Andrea Schaerf. Multi-neighbourhood local search with application to course timetabling. In Edmund K. Burke and Patrick De Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002, Gent, Belgium, August 21-23, 2002, Selected Revised Papers*, volume 2740 of *Lecture Notes in Computer Science*, pages 262–275. Springer, 2002.

[15] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *ECAI2000, Proceedings of the 14th European Conference on Artificial Intelligence*, pages 599–603, Amsterdam, 2000. IOS Press.

[16] Christelle Gueret, Narendra Jussien, Patrice Boizumault, and Christian Prins. Building university timetables using constraint logic programming. In Burke E. and Ross P., editors, *The Practice and Theory of Automated Timetabling: Selected Papers (ICPTAT '95)*, volume 1153 of *Lecture Notes in Computer Science*, pages 130–145, Berlin Heidelberg, New York, 1996. Springer-Verlag.

[17] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montral, Quèbec, Canada, August 20-25 1995.*, pages 607–615. Morgan Kaufmann, 1995.

[18] Martin Henz and Jörg Würtz. Using Oz for college timetabling. In *Proceedings of the 1995 International Conference on the Practice and Theory of Automated Timetabling*, Edinburgh, Scotland, aug 1995.

[19] F. Le Huèdè, M. Grabisch, C. Labreuche, and P. Savèant. Integration and propagation of a multi-criteria decision making model in constraint programming. *Journal of Heuristics*, 12(4-5):329 – 346, sep 2006.

[20] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[21] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.

[22] Olli Kamarainen and Hani El Sakkout. Local probing applied to network routing. In Jean-Charles Régin and Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science*, pages 173–189. Springer, 2004.

[23] Zeynep Kiziltan. Symmetry breaking ordering constraints. *AI Communications*, 17(3):167–169, 2004.

[24] Philipp Kostuch. The university course timetabling problem with a three-phase approach. In Burke and Trick [6], pages 109–125.

[25] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2):133–163, 2001.

[26] Andrea Qualizza and Paolo Serafini. A column generation scheme for faculty timetabling. In Burke and Trick [6], pages 161–173.

[27] Riccardo Rasconi, Nicola Policella, and Amedeo Cesta. SEaM: Analyzing schedule executability through simulation. In Moonis Ali and Richard Dapoigny, editors, *IEA/AIE*, volume 4031 of *Lecture Notes in Computer Science*, pages 410–420. Springer, 2006.

[28] Hani El Sakkout and Mark Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5(4):359–388, 2000.

[29] Andrea Schaerf. A survey of automated timetabling. *Artificial Intelligence Review*, 13(2):87–127, 1999.

[30] Gert Smolka. Constraints in OZ. *ACM Computing Surveys*, 28(4es):75, 1996.

[31] Ralph E. Steuer. *Multiple Criteria Optimization: Theory, Computation, and Application.* Wiley, New York, 1986.

[32] Mark Wallace and Thierry Le Provost. CHIP and Propia. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 507–509. Springer, 1992.

[33] Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.