SOFTWARE ENGINEERING ASPECTS OF CONSTRAINT-BASED TIMETABLING – A CASE STUDY

A. Abbas[1]
Department of Computer Science,
University of Balamand,
Tripoli, Lebanon
email: abbas@balamand.edu.lb

E.P.K. Tsang
Department of Computer Science,
University of Essex,
Colchester, CO4 3SQ, England
email: edward@essex.ac.uk

ABSTRACT

This paper[2] details the stages of building a substantial, carefully specified, fully tested and fully operational university and school timetabling system. This is reported as a case study in applying Constraint Satisfaction techniques. The emphasis is on the software engineering aspects of the problem. That is, Constraint Satisfaction problems are expressed in a language more familiar to the formal software engineering community. Moreover, this language is used to formulate domain constraints and heuristic information. In addition to that, the user's needs are looked at more closely. For instance, the system supplies indications useful for relaxing or reformulating the constraints of the problem when a solution satisfying these constraints is impossible to produce. This has a value in bringing Constraint Satisfaction one-step closer to formal specification, program verification and transformation.

Keywords: Constraint Satisfaction, Timetabling, Program Specification, Software Engineering

1. INTRODUCTION

Timetabling is an instance of task scheduling. This is a well-known `NP`-complete problem. That is, no known algorithm is adequately efficient for all its instances. This problem is ubiquitous in all practical aspects of modern societies. In fact, timetables play an important and sensitive role specifically in people intensive institutions such as hospitals and universities. This naturally generates considerable interest in understanding the timetable generation process. This consequently forms a focus of attraction to researchers from a variety of backgrounds and disciplines, in the quest for adequately efficient and flexible timetable generators.

The degree of maturity of the field can perhaps be measured by the existence of regular conferences dedicated to the timetable generation process [10, 13]. This can also be estimated from the existence of numerous commercial timetabling companies ([38] and [47] are just examples). Such companies provide extensive timetable generation services in specialized domains. This is also an indication of the complexity of the timetabling task, which is directly related to the complexity of the conditions timetables have to satisfy.

[1] Contact author
[2] This is an extended version of a short paper presented at the AICCSA'01 (see [1]).

Timetables and the timetable generation process are looked at in the literature from many different angles (see [15] and [42] for good overviews and more references to the subject). Correspondingly, there have been many approaches to the realization of these views using a wide variety of problem-solving paradigms ([7], [11], [12], [33], [36], [39], [48] and [54] are sample instances). Compared with these, Constraint Satisfaction techniques figure prominently ([4], [27], [35] and [55] are just a few examples of Constraint-based approaches to university/school timetabling).

The view of the timetabling task adopted in this paper is similar to (but somewhat a simplification of) that reflected in [14]. However, the problem solving approach is closer to what is discussed in [53], whereas the software engineering element of this goes in the direction of [56]. More discussions and references are provided below that closer to the sources of the problem solving paradigm adopted in this paper.

This paper advances this view toward a formal setting. In fact, it provides explicit formulations of several concepts central to the generation of university and school timetables. This should be beneficial in making Constraint Satisfaction techniques (in general) and the timetabling task (in particular) more widely accessible to the computer science audience. This would especially be useful for those interested in formalizations, or in the construction of actual timetable generators.

2. THE CONTEXT

The case study reported in this paper is set in the context of a small university[3], where students come from a variety of different backgrounds. In this context, students taking the same kind of courses are not the total majority in any one class. Thus, without taking this variety into consideration, a timetable will very likely incur time conflicts preventing a meaningful number of students from taking their intended courses. This can result in a drop in semester enrollment and the closing down of classes because not enough students will be taking them.

This scenario does not arise that often in larger more-established universities because, in the majority of cases, enough students will take the courses on offer. Even when the odd case arises, the consequences will not be of the kind that the university can not afford. However, with less homogeneity and fewer students to go around, the impact will not be that pleasing.

This provides a particularly fitting test for Constraints Satisfaction techniques. Intuitively, the difficulty in constructing a timetable is directly proportional to the number of different cases it is taking into account. That is, the same space of possibilities would have fewer solutions resulting from tighter constraints.

[3] Faculty of Sciences and Engineering, The University of Balamand, Lebanon

3. THE PROBLEM

At the start of every academic semester, students go through a *pre-registration* period. This is when they can select the classes they would like to attend during the semester. On the list of information provided then, there would still be a noticeable number of empty slots. These are to do with which instructor would be teaching which class or what time (or times) of the week such a class would fall in. Besides, some of the existing information on this list is understood to be subject to change.

The administrators would then gather the enrollment information and try to fill in these empty slots. The aim would be a timetable that is suitable to all instructors, students and other university requirements, resources and facilities.

However, the task of producing by hand, a satisfactory, conflict-free timetable is a lengthy, tedious, time-consuming and error-prone process. The main difficulty is the absence of any dependable idea of doing that. This is essentially a trial-and-error process that relies solely on the basic knowledge of the extremely intricate dependencies between the students, instructor and course elements of the process.

Worse than that, in the absence of adequate planning, the inherently-random *pre-registration* process can result in irresolvable conflicts between various student and instructor choices. That is, very often, no timetable could possibly satisfy all choices, even with an exhaustive search of all available alternatives.

Thus, very frequently, administrators would resign themselves to accepting an approximate solution. This is a solution that, they know *a priori*, would not be pleasing to everyone. They simply hope that such a tentative solution will mend itself during the subsequent *drop-and-add* period. This usually means wasting the first two weeks of the semester with students reselecting their courses and/or instructors adjusting their course assignments. This way, the process may converge on a solution that everyone would be content to live with for the duration of the semester.

4. THE SOLUTION

The *Timetable Generator* project was initiated in response to this problem. Adopting the view that the solution resides in some intelligent search strategy, the first period of the project was spent investigating the possibility of benefiting from the best-first search algorithm (e.g. [16]). However, the size of the data involved was huge. Furthermore, no sufficiently powerful heuristics were available to cut down the amount of search involved. These two factors forced an early abandonment of this research direction.

Later implementations had more benefits from Constraint Satisfaction techniques, resulting in a much more usable program (see next section). Timetabling proved to be a rich domain, very much conducive to a Constraint-based solution. In fact, as it currently stands the system implements and reaps the benefits of many of the core techniques presented in [49]; all within a single framework.

5. A SOFTWARE ENGINEERING APPROACH TO CONSTRAINT SATISFACTION

Early research on Constraint Satisfaction focused on problem solving strategies and algorithms (see [49] for an account and section 5.2 below for more details and references). But now that Constraint Programming is mature enough for real life applications [52], the attention has gradually shifted toward its software engineering aspects. That is, with plenty of solving strategies around, the tendency nowadays is to relieve the programmer from the problem solving effort. The general approach is to hide the details in a declarative or high-level language and let the programming system do the interpretation (see [34] for an early reference on that).

In the context of Constraint Satisfaction, modeling lies nowadays at the heart of this approach [19, 21]. The quest for ever-more high-level modeling constructs [28] has tremendously increased the modeling capabilities given to the user. This is interesting to software practitioners, because problem models can this way be clearly separated from the operational aspects of the solver, and from the surrounding software environment. Hence, the programmer can devote more time to more basic aspects such as modeling, modularity, extensibility, etc. The following are some of the advances on Constraint-based development that took place during the past decade:

- Modeling tools for constraint programming such as the modeling language `OPL` [51], the `CHIC-2` Project [17] and `ECLiPSe` [32].
- The `Ilog` Solver [29] library, which provides software components and abstractions for Constraint-based combinatorial search.
- Ongoing research on constraint modeling and programming ([20, 50]).

This same approach is followed in this paper. That is, timetabling is treated as a Constraint Satisfaction problem. As such, a formal specification language with high-level constructs (called `DEPICT 0.1`) is used to formally specify the timetabling problem ([2] and [3] have more details on `DEPICT 0.1`). Subsequently, an algorithm satisfying these specifications is presented.

5.1. Specifying Constraint Satisfaction Problems

A Constraint Satisfaction Problem (`CSP`) can be defined by the following three components (definition modified from [49]):

- a finite set of variables $A = (a_i)_{i=1..n}$
- a corresponding set of domains $B = (B_a)_{a \in A}$. Each B_a is a finite domain of values that the corresponding variable a can take its value from.
- a set S of subsets of A over which a set of constraints $C = (C_s)_{s \in S}$ is specified. Each C_s is a constraint that ties together the variables of the subset s by restricting the values they may simultaneously take.

A CSP solution consists of finding, for each of the variables $a \in A$, a value $b \in B_a$ satisfying all relevant constraints. That is, for each $s \in S$, the set $\{<a, b>; a \in s\}$ must satisfy the constraint C_s. An initial attempt at specifying this problem could be:

$$(\forall a \in A)(\exists b \in B_a)C(<a, b>)$$

However, this does not quite express the simultaneity of the application of the condition C over all pairs $<a, b> \in A \times B_a$. A more accurate description of this fact goes as follows: find a function f associating every element $a \in A$ with an element $b \in B_a$ so that $C(f)$ is satisfied. Here, given that f_s is the restriction of f over the subset s of A, $C(f)$ is specified by:

$$(\forall s \in S) C_s(f_s)$$

In the specification language of Martin-Löf's Theory of Types [40], having dependent types and the type constructors Π and Σ, the specification manifests itself rather succinctly as the type expression:

$$\Sigma(\Pi(A, B), C)$$

However, restricting ourselves to set theory and 1st order predicate calculus as a specification language, if B is taken as the set $(\cup B_a)_{a \in A}$, then this specification can still be written as:

$$(\exists f \in A \rightarrow B) C(f)$$

Note that the latter specification seemingly loses sight of the fact that $f(a) \in B_a$ for all $a \in A$, as the initial definition of the problem suggests. However, seen as a constraint, this fact can always be made a part of the constraint expression C itself.

5.2. Solving A Constraint Satisfaction Problem

Details of how to generate the solution obviously depend on the structure of the variables A, the values B and also on the way the constraints C are specified. But, whatever the nature of this solution, it can be safely said that it relies in a fundamental way on the fact that A and B are discrete and finite and also that C is not computationally too expensive to decide.

Intuitively, one possible solution can be reached through searching the sets of values corresponding to each variable. Here, backtracking will need to be called in whenever the current path the search is following reaches a dead end.

Heuristic information can sometimes be extracted from the way the sets A and B sets are structured. This can also be done from the way the set C is specified. Heuristic information can result in substantial savings on the number of alternatives explored on the way to the solution.

Many algorithms have been developed to exploit such information. Some propagate constraints in order to reduce the size of the search space [37] and detect failure early. Others learn from failure [23], as sibling branches are similar. Efficiency can also be gained by ordering the choice points in the search [26]. Although these algorithms have the same worst-case complexity as traditional state-based search algorithms, they can be much faster in practice. This last statement applies especially well to the algorithm adopted in this paper (see next section).

The majority of complete CSP algorithms conduct depth first search; they keep only one active state together with a set of pointers. Each pointer is associated with one of the variables of `A`. This keeps track of where the search is currently at in the corresponding value domain.

5.3. A General CSP Algorithm

The following is an algorithm that can solve this class of problems. It starts from the `CSP` specification above and returns a result that is a `solution` or a `failure`:

```
Result ≡ Solution ∨ Failure

Result SCHEDULE (A, B, C) { //A: variables, B: values, C: constraints

timetable = []; // timetable initially empty
scheduling = true; // {x∈A; ¬scheduled(x)} ≠∅ initially
a = highestpriority({x∈A; ¬scheduled(x)}); // select variable

while (scheduling)&&({x∈A; ¬scheduled(x)}≠∅) {
// select value
b = mostsuitable({y∈B_a;¬considered(y)∧C(timetable+<a,y>)});
if found(b) { // Successful selection
             add(<a, b>, timetable);
             a = highestpriority({x∈A; ¬scheduled(x)});}
else { // Unsuccessful selection, dead end, select a backtracking point
       t = mostsensible({x∈A; scheduled(x)});
       if found(t) then // backtracking point found
          {erase(a .. t, timetable); a = t;} // Bookkeeping
       else scheduling = false;}} // general failure
return({x∈A;¬scheduled(x)} = ∅)? timetable : failure);
}
```

A General CSP Algorithm

The `solution` to the `CSP` problem is the function `f` mentioned in the specification above. However, since `A` is assumed to be finite, this function may also be viewed as a list of pairs $<a,b>\in A\times B_a$. The elements of this list are chronologically ordered according to the time they are generated (earliest first). In fact, this is the view of `f` that will be taken by the algorithm above. On the other hand, a `failure` arises when no solution can be generated because the constraints are too tight or contradictory.

The pseudo code above basically outlines (in a C-like notation) Gaschnig's BackJumping algorithm [24] (also see [18] for an interesting survey paper). Some remarks are due to clarify assumptions that may not be sufficiently explicit there:

1. For any given $a \in A$ and during the process of locating `b`, constraints are checked with respect to already scheduled elements of `A` only. When `b` is found, it is marked as `considered` to leave it out of the search. This is crucial in case of backtracking for an alternative value in B_a. Moreover, unsuccessful attempts at locating `b` are recorded on a `failure table` as it is used in Gaschnig's Backmarking algorithm. This table is very useful for determining the most sensible point `t` to backtrack to, whenever backtracking is required.

   This `failure table` is reconstructed from scratch for every new $a \in A$. The size of this table is of the order of the size of B_a.

2. The existence of this `failure table` reduces the process of finding `t` to a simple linear scan. After finding `t`, all elements $a' \in A$ between `a` and `t` have to be marked as `¬scheduled`. Moreover, all `b`'s in the corresponding $B_{a'}$'s are marked as `¬considered`, because they may need to be searched again.

   This clearly has negative effects on the efficiency of the algorithm. The extent of that may be reduced by an intelligent choice of the next `a` to schedule. More elaborate backtracking schemes do exist (e.g. see [6], [25] and [31]) for minimizing these effects.

3. Although not explicitly stated above, the above algorithm employs some implicit forward checking, which is a form of constraint propagation[4] [41]. That is, depending on the current state of the search, the algorithm may skip without testing whatsoever many of the values in B_a of the current variable `a`. In fact, on the basis of the variables assigned so far, these values would be no-good values [30].

   But what the algorithm does not do is marking the no-good values ahead of time (see [44] and [45]). This is judged to take too much space, because of the extensive size of the domains involved. Additionally, the marking process will take too much time on its own right[5].

[4] The role of propagation in CSP solving is similar to that of heuristic information in an intelligent search procedure. That is, it serves to exclude from consideration any potential (that would turn out to be useless) solutions.

[5] Strong propagation defeats the purpose of its use in the timetabling context. This is because it is judged to take too much time to exclude possible solutions. In fact, in all but the most trivial timetabling applications, each variable tends to have many possible values. Such a variable is also tied in many intricate relationships with most (if not all) of the variables of the problem.

6. TIMETABLING AS A CONSTRAINT SATISFACTION PROBLEM

Given the courses on offer in any one semester, the task is to allocate times during the week for each one of those courses. The allocated times are subject to the following constraint: no two distinct courses can have time conflict, if they are taken by the same student or taught by the same instructor.

In so far as these requirements are concerned, the role of the student taking a course is no different from the role of the instructor teaching it. Both can not be in two different classes at the same time. Thus, in this respect, no loss of generality will occur if we can restrict ourselves to mentioning only students.

However, additional constraints may arise; such as the availability of adequate classrooms to accommodate these courses during the allocated times. However, to somewhat simplify the presentation; the only constraints we will tackle are to do with time conflict.

Note here, the constraint-based view directly naturally mirrors the structure of the timetabling problem. The advantage is that further constraints may be added to the system as supplementary data. These additions should not bring with them any changes to the flow of control of the main algorithm. However, this may obviously have a proportional effect on the overall efficiency of the program.

6.1. The Timetabling Problem Specification[6]

Given the discussion in the previous sections, the requirements of the problem can be captured by the following specification:

```
Timetabling ≡ (∃ f ∈ Courses → Times) Suitable(f)

Suitable(f) ≡ (∀c,c′∈Courses)(c ≠ c′)⇒ ¬Conflict(c, c′, f)

Conflict(c, c′, f) ≡ (∃ x ∈ Students) c∈s(x) ∧ c′∈s(x)
                              ∧ <c,b>∈f ∧ <c′,b′>∈f ∧ b∩b′≠∅
```

where (≡) is meant to be definitional equality `and s` is a function returning the set of courses taken by the student `x`. Once more, even though the solution is specified as a function, what is going to be generated is an association list, of course-time pairs, that plays the same role. This list will in fact be the desired `timetable`. Note also that the above is just an initial idea of the expression of `Suitable`. This will be further elaborated as soon as more information becomes available (see below).

[6] This specification is intended as an illustration of the approach adopted in this paper. As such, it is not meant to be complete. However, the underlying approach is hopefully clear enough to ease the task of filling in the details in any particular timetabling application.

6.2. The Variables

From the above specification, it is obvious that the variables are the courses on offer. In case of a multiple-sections course (i.e. when the same course is given to more than a single group of students), each section will be considered as a distinct variable. This makes sense because, in general, each section will have its own distinct timetabling requirements.

6.3. The Values

The space of values is the space of all available teaching times during the week. The unit value is a time `slot.` A time `slot` is defined as a time interval over one day of the week. This is specified by a triplet: `<d, t, h>`, where `d` is a number denoting one day of the week, `t` is a number denoting the starting time of the slot and `h` is the `length` of a slot; i.e. number of half-hours this slot consists of. For example, the triplet `<1, 1, 3>` denotes the time interval on `Monday`, starting at `8:00AM` and ending at `9:30AM`. Hence, the first constraint that a slot value `<d, t, h>` should respect is:

$$C_0 \equiv WD(d) \wedge ([t \; .. \; t+h] \subseteq WH(d))$$

Where `WD(d)` means that `d` is a working day and `WH(d)` denotes the set of working hours of the day `d`.

6.4. Composite Values

Each course is associated with a number of credits. This number is used to indicate the number of teaching hours associated with this course, the number of slots that those hours may be distributed over, and the constraints this distribution has to respect.

Credit No.	Teaching Hrs.	Slot No.	Slots length
1	3	1	6
2	2	1 or 2	4 or (2,2)
3	3	2 or 3	(3,3) or (2,4) or (2,2,2)
4	4	2 or 3	(4,4) or (2,2,4) or (2,3,3)

Course-Slot Distribution Table

Thus, depending on its number of credits, a course might be associated with a composite value (i.e. a set of slots). For example, a 1-credit course is given over a single 3-hour slot. A 2-credit course is given over one 2-hour slot or two 1-hour slots. A 3-credit course can be given over two 1.5-hour slots or three 1-hour slots, etc. A 4-credit course is given two 2-hour slots, etc (see table above).

The table above is used to make sure that only slots of the right size are ever considered for courses of a given type: i.e. number of credits. Hence the second constraint:

$C_1 \equiv \texttt{valid\_size}(c, n, s, T)$

Where `valid_size` is a function that makes sure that course `c` with a number of credit `n` has the correct size of the slot `s` in the Course-Slot Distribution Table `T`.

Multiple-slot courses are subject to an additional constraint. This requires that any two different slots `<d,t,h>` and `<d',t',h'>`, associated with the same course, should start at the same time during the day. Moreover, these slots should be separated by a gap of one day at least.

That is, the following two constraints should hold:

$C_2 \equiv (\langle d, t, h\rangle \neq \langle d', t', h'\rangle) \Rightarrow (t = t')$

$C_3 \equiv (\langle d, t, h\rangle \neq \langle d', t', h'\rangle) \Rightarrow (|d - d'| > 1)$

6.5. Value Ordering

Seeing the complexity of these values and in order to make their management easier, we chose to impose a total ordering on them. Given two slots `<d,t,h>` and `<d',t',h'>`, we define:

$(\langle d,t,h\rangle \leq \langle d',t',h'\rangle) \equiv (d < d') \vee ((d = d') \wedge (t \leq t'))$

Note that this is not the only ordering that we can impose over slot values. In fact, more useful ordering may be defined depending on the availability of more specific domain information.

6.6. Value Generators

Such an ordering may be used as the basis of a value generator `(VG)`. In fact, since the values of a domain will generally be too numerous to store explicitly, this approach has a number of advantages. For instance, this value generator may be used to produce a next value of a slot every time this is needed. Accordingly, earlier values are `considered` first. Furthermore, when a current value is being `considered`, earlier ones are implicitly marked as `considered` and later ones are still as yet ¬`considered`.

In addition to that, each slot (of each course) can have its own generator `(SG)` producing values of the appropriate type. Obviously, to be useful, this generator should be able to generate all valid values without neither missing nor repeating any. That is, the time when a value is so generated must define a total order over the domain of these values.

6.7. A Miniature Scheduler

Following this line of reasoning, any single course (a slot set) constitutes a miniature-timetabling problem, having its own variables, values and constraints. Thus, we constructed a miniature program (SSG) that can generate a composite value for each such a set, respecting all relevant constraints. This program obviously makes use of the generators (SG) above.

It is worth noting here that the generator (SSG) adopts only a blind backtracking scheme, because a set of slots is never more than a few elements in size, and such sizes do not justify the amount of overhead required by a more intelligent backtracking scheme.

6.8. The Constraints

As we said earlier, the constraints that will be addressed are mainly those to do with timing various courses. This is elaborated in the subsections below.

6.8.1. Time Conflict

According to the specification of the problem given earlier, a timetable is considered unsuitable if there exists a student who is enrolled in two different courses whose times overlap. We add to that another constraint: the time of the course should not overlap with the times this student cannot attend the course. This additional constraint is particularly useful for part-time students.

Such information can come from a set Σ of records on every student of the institution. Each record is a pair <cs, ss> defined as follows:

- cs is the set of courses that will be taken by the student.
- ss is the set of slots during which the student cannot attend.

We will assume here a course generator (CG) that produces the highest-priority course to schedule. The ordering relation (<) over courses is defined with respect to the time at which each such course is so produced. The details of this generator will be explained in the next section.

Now, given a course $c \in \text{Courses}$ we are trying to find a time for, a slot s being considered for c will cause no conflict if the following conditions hold for every record $\langle cs, ss\rangle \in \Sigma$:

$C_4 \equiv (c \in cs) \Rightarrow (s \cap ss = \emptyset)$

$C_5 \equiv (c \in cs) \Rightarrow (\forall c' \in cs (c' < c) \Rightarrow (\forall s' \in time(c') (s \cap s') = \emptyset))$

For a student taking the cs courses, constraint C_4 says this student can not attend classes at his/her specified times, while C_5 says that the different classes of this student should not overlap. Here, the function time, applied to an already scheduled course, will return the set of slots allocated for this course, and also:

$$(s \cap ss) = \emptyset \equiv \forall\, s' \in ss\ (s \cap s') = \emptyset$$
$$<d,t,h> \cap <d',t',h'> = \emptyset \equiv (d=d') \Rightarrow \max(t,t') \geq \min(t+h,t'+h')$$

The simplicity of the latter relation is but one of the benefits of our adopted representation of a time slot. This turns out to be of a major importance for efficiency, considering the enormous number of times the algorithm is going to check this relation during any reasonable-sized scheduling task.

6.8.2. Load Conflict

Another constraint that should be satisfied by a slot is that no student should have too many attendance hours on any single day. That is, the total load per day should not exceed a certain given maximum M. That is, a slot <d, t, h> for a course c causes no conflict if the following condition holds for every record $<cs, ss> \in \Sigma$:

$$C_6 \equiv (c \in cs) \Rightarrow \mathrm{sum}(\{\mathrm{length}_d(c');\ c' \in cs \wedge c' \leq c\}) \leq M$$

Where $\mathrm{length}_d(c) \equiv (\exists <d, t, h> \in \mathrm{time}(c)?\ h : 0)$

7. DETAILS OF THE TIMETABELING ALGORITHM

We are now at a stage where we can supply more details of the behavior of the general CSP algorithm.

7.1. Course Generator

One major task the algorithm is doing every iteration is *dynamically* choosing the next course to schedule. A good choice here will obviously have a major impact on its global efficiency. This choice is usually based on the available heuristic information.

The general strategy is based on choosing the course judged *hardest* to schedule; i.e. the course with the tightest constraints. The general insight behind this strategy is the following: if such a course has the least room to maneuver then, after it has successfully been scheduled, other courses will hopefully still have enough room left to be scheduled. Even in the case where this course is not being successful, no time would have been wasted on scheduling easier ones. On this basis, we list below the factors that are judged to reduce most the ease with which a course c is scheduled.

All these quantities are made explicit in our formulation of the problem. They are all available in numerical form, and they are easy to access and calculate. The course that is chosen will have the *greatest combination* of these quantities. The meaning of the auxiliary functions used to specify that should be self-explanatory:

– the number of slots this course is currently taking:

```
NS(c) ≡ size(time(c))
```

– the length of time this course will occupy during the week:

```
LS(c) ≡ sum({h; <d,t,h> ∈ time(c)})
```

– the total number of student records this course belongs to:

```
NR(c) ≡ size({cs; <cs,ss>∈Σ ∧ c∈cs})
```

– the number of courses belonging to each one of these records:

```
NC(<cs,ss>,c) ≡ sum({size(cs);<cs,ss>∈Σ∧c∈cs})
```

– the length of time constraints associated with each one of these records:

```
LR(<cs,ss>,c) ≡ sum({h; <d,t,h>∈ss})
where <cs,ss>∈Σ ∧ c∈cs.
```

The particular combination that we have been using is the following[7]:

```
NS(c)×LS(c)×NR(c)×sum({NC(r,c)×LR(r,c); r∈Σ})
```

However, it should be clear that this is just a heuristic expression is application-dependent; that is, more thought may need to be invested in weighing and combining these factors to come up with the optimal combination for any particular application.

7.2. The Backtracking Scheme

While attempting to schedule the current course, the current path the algorithm is following may reach a dead end. This happens because slots fail to find a set of values suitable for all of them. Repairing this failure can be attempted if it can be traced back to a decision related to a previously scheduled course. This way, undoing that decision might make further progress possible.

[7] The point of whether heuristic information should be part of the timetabling problem is debatable. Whatever stance one takes from this point, the timetabling process is conducted here through some form of search. Moreover, it is well-known that heuristic information help in speeding up the search and therefore in improving the overall efficiency of the system.

Of course, if at all possible, we are interested in the most critical point to backtrack to. This choice is made for the sake of minimizing the amount of work left to be done, before the final solution is complete, or before hope in finding any solution is totally lost.

7.3. The Failure Table

When selecting a fresh course c to schedule, we create a fresh `failure table` along with it. The number of entries of this table is the number of all possible slot values associated with this course.

During the process of scheduling the course, the following bookkeeping routine takes place: we record in the corresponding entry on the table all courses whose interaction with c caused that entry to fail.

7.4. The Backtracking Point

If c fails to schedule, then we go through a minimization process that leaves, at each entry of the table, the *earliest* of all courses listed at that entry. Then, through another maximization process, we go through all entries determining the *latest* of all courses left on the table. If at all found; that course will be the one to backtrack to. In our notation, this can be expressed by the following:

$$\mathrm{max}\Big(\{\mathrm{min}(\{c';(c'<c)\wedge(v\cap\mathrm{time}(c')\neq\emptyset)\});v\in\cup\{\mathrm{time}_s;s\in\mathrm{time}(c)\}\}\Big)$$

Here, time_s is used to denote the set of all possible values of the slot s (see [49], for more details on this *min-max* process).

7.5. Remarks

There are few things to note about the above process:

- The table size is bounded by the size of the set: $\cup\{\mathrm{time}_s;\ s\in\mathrm{time}(c)\}$.
- The process of filling the table is done completely iteratively, during the process of checking the constraints related to each value of the above set.
- The table can be easily turned into a list, by doing minimization on each entry during the process of filling the table. This will reduce the storage requirements on the table without any extra burden on the execution time, because the process of minimization will have to be done anyway.
- Determining the backtracking point is done in linear time in the order of the size of the table itself.

7.6. Constraint Ordering

A closer look at the `CSP` algorithm reveals that almost all the work that it is doing is spent on Constraint Verification. For this reason, if efficiency is a weighing factor in how much this algorithm is acceptable; one should invest an equivalent amount of effort in optimizing Constraint Verification.

The organization of Constraint Verification in the program has been motivated by the following key idea: since local failure is unavoidable in general, then the best is to catch it early. In fact, in this program, we have seven key constraints: C_0, C_1, … C_6. They are being used, through a chain of processes, exactly to filter out values from the initial rough domain of all possible slot values, till these become valid course times at the end.

In order to achieve this task, some optimization is used. In fact, the application of these constraints is ordered along this chain as follows: C_0, C_1, C_3, C_2, C_6, C_4 and C_5. The guiding intuition behind this ordering is: a bad value, traveling along this chain, should be detected and eliminated from further consideration as soon as possible, and with the least amount of computation.

What helped the most in realizing this intuition is the strict modularity enforced on all stages of development of this program. This is especially true for the generators: `SG`, `SSG` and `CG`.

8. CONSTRAINT OPTIMIZATION

The preceding details seem to imply a lot of pointless preparations that should be made at the beginning of every new semester. This would be alarming for an administrator in a university that has (say) 20 000 students. At the face of it, the algorithm seems to suggests that it would need 20 000 sets of courses. Each such set represents what the corresponding student might like (but not necessarily has) to take!

In practice, however, the administrator will feed the system sets for typical students, plus the odd special-case student that may occur here or there. Still, these fine details may yield no equivalent return at the end. This is especially true when the list Σ of student records implies unsatisfiable constraints. In such a situation, the program will simply return `failure`.

Fortunately, the matter should not be so rigid in real life. In the face of a timetable conflict, students usually develop escape routes that can not easily be formally captured: e.g. take a different section of the same course, choose another available elective or delay a planned course till next semester and take another required one instead.

Clearly, this is a genuine problem. So, in situations where constraints could not all be satisfied, we looked into the idea of a timetable with a tolerable number of conflicts. The structure of the `CSP` algorithm proved essential to the realization of this idea.

Cost of Constraint Violation

One idea of addressing this problem is to attach a weight to each of the constraints [22]. A weight represents a cost that have to be paid for the violation of any instance of the corresponding constraint. For example, this cost can be the number of individual student conflicts a course can tolerate without having to close down. Accordingly, the goal of the algorithm is modified in the following way: instead of having to satisfy all constraints, a time `t` is assigned to a given course `c`, if the total cost of the conflicts that `t` cause is affordable. Reader are referred to [5] and [9] for more in-depth research in this direction.

The strategy of attaching weights to constraints varies depending on the problem at hand. We can treat some or all of the constraints as soft [43]. In the case of hard constraints, we can increase the weights so much as to make them impossible to be acceptably violated.

A version of this program, with a cost parameter, has already been implemented and is currently in use. It is able to come up a satisfactory solution when an ideal conflict-free timetable is impossible to generate.

9. EXPERIMENTATION

At present, a fully operational system is being used at the start of every semester for generating a timetable for almost a thousand students, distributed over more than a hundred different classes. The most remarkable feature of this program is its high degree of modularity. In fact, it is this feature that helped the backbone of this program to survive a sequence of implementations[8], modifications and upgrades.

9.1. System's Acceptance Tests

In order to gain the trust of the administration, the system went through many rigid tests, including data gathered from the years prior to the system's installation. Once accepted, the system is being used with remarkable success at the start of every semester for the past five years.

9.2. System's Role and Value

Thanks to the existence of this system, two weeks of hard disruptive work at start of every semester were replaced with a run-time on a small PC that can be measured in minutes. As such, this system is currently being used to the satisfaction of everyone. Some part-time students and instructors can now ask for their courses to be scheduled at certain times of the week. We can have some times of the week declared lecture-free hours without being afraid of disturbing anyone.

[8] The initial prototype of the system was implemented in `LISP`. The first working version was in `PASCAL`. The current version is in `C++`.

Its ease of use, flexibility and speed has been a factor in sometimes allowing several runs during a single semester. These runs were required to accommodate unanticipated circumstances of some students or instructors. Some instructors indulge themselves sometimes in changing the time set by the system for one particular course without disturbing the rest of the timetable[9]; something that could not even be envisaged before.

A sample run is presented in the Appendix to illustrate the input and output of the system. The reader will appreciate that the example presented there is a simplified one (used for the purpose of illustration). More elaborate real-life examples, demonstrating the power of the system, will involve hundreds of students, many tens of classes. As such, these need more space than can be spared in the context of this paper.

9.3. Generality of the System

From the very beginning, the system was tailored to fit the university special requirements. The flexibility of its design allowed its use for the scheduling needs of a local high school, with only some minor modification to its I/O module. That is, its core module remained untouched. The effort that was required for these modifications was minimal; hence the merits of the approach advocated in this paper.

The high-school timetabling problem is not specified here. This is because it would seem to be a virtual repetition of the same steps outlined in the university context in theoretically similar circumstances.

9.4. A Room-Allocation Program

It is perhaps worth mentioning here, that based on the general `CSP` algorithm a room-allocation program has recently been completed[10]. This program takes a timetable as input and finds suitable rooms in the university for each of courses mentioned in the timetable. This is obviously done with respect to the given times, the number of students and other possible needs of each of those courses. The specification of this problem and the corresponding details of the program will not be reported here.

10. DISCUSSION AND POSSIBLE EXTENSIONS

10.1. Program efficiency

The general `CSP` algorithm implements BackJumping [24], which is a complete algorithm. That is, this algorithm will guarantee a solution if one exists. However, it is

[9] Simple; since the system allows constraints in the form of courses with preset times, all courses whose generated times are judged to be satisfactory will be considered as preset in the next run.

[10] Room allocation is managed apart because tackling too many constraints simultaneously is judged at the time to affect the overall efficiency of the algorithm in the absence of enough computational power.

well known that scheduling is an NP-complete problem and, as such, no known algorithm is adequately efficient for all its instances. Thus, in theory, the algorithm has a worst-case situation. That is, in some instances, it may run for too long before it gives a solution or before it can decide that no solution exists. Two escape routes have been devised to avoid such situations:

- Plenty of care is paid to the design of the initial input data. This is crucial if we are to eliminate from consideration many of the intractable situations that may arise in practice.
- An escape option, that allows the program to exit without a solution (if it appears to be taking too long). In this case, the system supplies useful indications for easing the constraints for another successful run[11]. This information would have been accumulating in the failure table during the current run.

Moreover, as it currently stands, the algorithm will give the first solution it finds satisfying all constraints. However, given some measure of optimality, the algorithm does not always return the optimal solution.

Optimality can be turned into a Constraint Satisfaction Decision Problem by adding to the specification a conflict cost, as explained above. A future direction would be to modify the algorithm (into a form of branch and bound, for example) for solving optimization problems directly.

10.2. An Initial Preprocessor and an Interactive Editor

With a complete program, the burden of a solution now falls on the design of the initial input data. This is a particular burden when there are multiple sections of the same course. There is another difficult situation when there are many electives that the student can take, but does not have to be assigned any one of them a priori. In this case, any student-course assignment may unnecessarily make the constraints impossible to satisfy.

For this reason, initial data should go through an initial filtering phase that distributes students on multiple sections and/or multiple electives, so as to minimize interaction between various classes. This is an optimization problem that awaits a solution.

In any reasonable-sized university, preparing the initial data to the program is a lengthy process that is very much prone to error. Some automatic help will do a good job in reducing the potential of errors here.

10.3. Problem Modeling

The problem of assigning students to one of the multiple sections can be addressed by an adequate problem reformulation. For example, suppose that a student selects a course

[11] This feature has been incorporated only recently. It is still in its exploratory stages. But the key idea is that if failure is inevitable then the system should supply the user with the causes of this failure. These may be used to analyze the situation for a more successful run.

having five different sections. Then, instead of scheduling time slots for student-courses, we can schedule sections to the student choice (assign one of the five sections to this choice), and time slots for sections (assign a time to each section, taking into consideration their assignments to student choices). It is widely believed that problem modeling may have a significant impact on search efficiency [8, 46].

10.4. Scope of the Approach

Constraint Satisfaction techniques are by now well known and their usefulness already been demonstrated in many applications. These techniques naturally fit design problems, which is the creation of objects satisfying given criterions. This is very well a distinct characteristic of the timetabling problem.

These techniques have been applied with demonstrable success to the generation of timetables for a small university, to room allocation for classes with preset times and also to the generation of high-school timetables (which, if anything, is a simplification the former application). Thus, in theory, the way is clear for the application of these techniques to bigger institutions or even to other timetable generation contexts, without of course being oblivious to the negative effect low efficiency can, in practice, have on such an enterprise.

11. CONCLUDING SUMMARY

This paper reports a successful application of constraint technology. The software engineering aspect of Constraint Satisfaction is emphasized in this project. We have taken a formal approach to specify a timetabling problem. A university and a school have used the timetable generation program presented in this paper, which is well tested and fully operational. The problem that we have specified is general enough, and therefore our experience should be useful to other researchers with similar applications.

ACKNOWLEDGEMENT

The authors are very grateful to the reviewers for their generous, detailed and insightful comments. These comments helped in significantly raising the quality of the material presented in this paper.

The first author would like to thank staff and students at the Faculty of Sciences and Engineering at the University of Balamand. This community has shown plenty of patience in bearing the consequences of the system's output during its early years. After all, it is these real-life tests together with the feedback that came from them that made the system what it is today.

Research by the second author is partially sponsored by EPSRC funds GR/H75275, GR/L20122, GR/M46297 and a Research Promotion Fund from University of Essex.

BIBLIOGRAPHY

1. Abbas, A. and Tsang, E., *Constraint-Based Timetabling - a Case Study*, proceedings of AICCSA'2001, the ACS/IEEE International Conference on Computer Systems and Applications, June 2001.
2. Abbas, A. and Tsang, E., *Toward a General Language for the Specification of Constraint Satisfaction Problems*, Proceedings of the CP-AI-OR 2001 workshop, Imperial College, London, England, April 2001.
3. Abbas, A. and Tsang, E., *DEPICT 0.1: A Formal Specification Language for Constraint Satisfaction Software Engineering*, A journal submission, currently under review, January 2003.
4. Azevedo, F., and Barahona, P. M., *Timetabling in Constraint Logic Programming*, Proceedings of the 2nd World Congress on Expert Systems, 1994.
5. Bistarelli, S., Montanari, U. and Rossi, F. *Semiring-based Constraint Solving and Optimization*, ACM Journal, pp. 201-236, Volume 44, No. 2, 1997
6. Bliek, C., *Generalizing Partial Order and Dynamic Backtracking*, Proceedings of AAAI, 1998.
7. Blum, C., Correia, S., Dorigo, M., Paechter, B., Rossi-Doria, O., and Snoek, M., *A GA Evolving Instructions for a Timetable Builder*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.
8. Borrett, J.E., *Formulation Selection for Constraint Satisfaction Problems: a Heuristic Approach*, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, 1998.
9. Boizumault, P., Gueret, C. and Jussien, N., *Efficient Labelling and Constraint Relaxation for Solving*, Proceeding of the 1994 ILPS post-conference workshop on Constraint Languages/Systems and their Use in Problem Modelling, Lim and Jourdan (Eds.), Volume 1 (Application and Modelling), pp. 116-130, 1994.
10. Burke, E.K. & Carter, M. (Eds.), *The Practice and Theory of Automated Timetabling*, Volume 2: Selected Papers from the 2nd International Conference on the Practice and Theory of Automated Timetabling, University of Toronto, August 20th-22nd 1997, Springer Lecture Notes in Computer Science Series, Volume 1408, 1998.
11. Burke, E. K., Gustafson, S., and Kendall, G., *Is Genetic Programming a Sensible Research Direction for Timetabling?* Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.
12. Burke, E. K., MacCarthy, B. L., Petrovic, S., and Qu, R., *Knowledge Discovery in a Hyper-Heuristic for Course Timetabling Using Case-Based Reasoning*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.

13. Burke, E.K. & Ross, P. (Eds.), *The Practice and Theory of Automated Timetabling*, Volume 1: Selected Papers from the 1st International Conference on the Practice and Theory of Automated Timetabling, Edinburgh August/September 1995, Lecture Notes in Computer Science Vol.1153, 1996.
14. Carter, M. W., *A Comprehensive Course Timetabling and Student Scheduling System at the University of Waterloo*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
15. Carter, M. W., and Laporte, G., *Recent Developments in Practical Course Timetabling*, Practice and Theory of Automated Timetabling, Burke and Carter (Eds.), Springer-Verlag LNCS 1408, pp. 4-19, 1996.
16. Charniak & McDermott, *Introduction to Artificial Intelligence*, Addison-Wesley, 1985.
17. *The CHIC-2 Project*, http://www-icparc.doc.ic.ac.uk/chic2/
18. Dechter, R. and Frost D., *Backjump-based Backtracking for Constraint Satisfaction Problems*, Artificial Intelligence Journal, pp. 147-188, Volume 136, No. 2, 2002.
19. El Sakkout, H., *Modelling Fleet Assignment in a Flexible Environment*, Proc., Practical Applications of Constraint Technology (PACT-96), London 1996.
20. Flener, P., *Towards relational modelling of combinatorial optimization problems*, in: Ch. Bessiere (ed.), Proceedings of the IJAI'01 Workshop on Modelling and Solving problems with Constraints, 2001.
21. Freuder, E.C., *Modeling: the final frontier*, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming (PACLP), pp.15-21, London 1999.
22. Freuder E. C. and Wallace R. J., *Partial Constraint Satisfaction*, Artificial Intelligence Journal, Volume 58, pp. 21-70, 1992.
23. Frost, D. and Dechter, R., *Dead-End Driven Learning*, Proc., 12th National Conference for Artificial Intelligence (AAAI), pp. 294-300, 1994.
24. Gaschnig, J., *Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying-assignment problems*, Proc. 2nd National Conference of the Canadian Society for Computational Studies of Intelligence, pp. 19-21, 1978.
25. Ginsberg, M. L., *Dynamic Backtracking*, Journal of Artificial Intelligence Research, volume I, pp. 25-46, 1993.
26. Haralick, R.M. & Elliott, G.L., *Increasing tree search efficiency for Constraint Satisfaction problems*, Artificial Intelligence, Volume14, pp. 263-313, 1980.
27. Henz, M. and Wuertz, J., *Using Oz for College Timetabling*, The Selected Proceedings of the 1st International Conference on the Practice and Theory of Automated Timetabling, LNCS 1153, Edinburgh 1995.
28. Hnich, B., *Function Variables for Constraint Programming*, Ph.D. Thesis, Computer Science Division, Department of Information Science, Uppsala University, Sweden 2003.
29. ILOG, Inc, *ILOG CPLEX 7.0 User's Manual and ILOG CPLEX 7.0 Reference Manual* (Gentilly, France, 2000); also see http://www.ilog.fr/.

30. Jiang, Y.J., Richards, T. & Richards, B., *No-good backmarking with min-conflict repair in constraint satisfaction and optimization*, Proc. of 2nd Workshop on the Principles and Practice of Constraint Programming Workshop, 1994
31. Jussien, N., Debruyne, R. and Boizurnault, P., *Maintaining Arc-Consistency within Dynamic Backtracking*, Principles and Practice of Constraint Programming (CP 2000) LNCS Series, no. 1894, pp.249-261, Springer-Verlag, Singapore 2000.
32. Kamarainen, O., *Using ECLiPSe to solve large-scale piecewise linear scheduling problems,* ECLiPSe User Group Newsletter, October 2002.
33. Kingston, J. H. and Yin-Sun Lynn, B. *A Software Architecture for Timetable Construction*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
34. Lauriere, J. L., *ALICE: A Language and a Program for Solving Combinatorial Problems*, Artificial Intelligence, Vol. 10, pp 29 – 127, 1978.
35. Legierski, W., *Search Strategy for Constraint-Based Class-Teacher Timetabling*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.
36. Löhnertz, M., *A timetabling system for the German Gymnasium*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.
37. Mackworth, A.K., *Consistency in networks or relations*, Artificial Intelligence, Vol.8, No.1, pp. 99-118, 1977.
38. McCollum, B. and Newall, J. *Introducing Optime: Examination Timetable Software*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
39. Müller, T., and Barták, R. *Interactive Timetabling: Concepts, Techniques, and Practical Results*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.
40. Nordstrom, B., Petersson, K. and Smith, J., M., *Programming in Martin-Lof's Type Theory - An Introduction*, volume 7 of International Series of Monographs on Computer Science, Oxford University Press, 1990.
41. Prosser, P., *Hybrid Algorithms for the Constraint Satisfaction Problem*, Computational Intelligence, Vol.9, No.3, pp. 268-299, 1993.
42. Reis, L. P. and Oliveira, E. *A Language For Specifying Complete Timetabling Problems*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
43. Rudová, H., and Murray, K., *University Course Timetabling with Soft Constraints*, Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling (PATAT '02), KaHo St.-Lieven, Gent, Belgium, Burke and De Causmaecker (Eds.), August 2002.

44. Schiex, T. and Verfaillie, G., *Nogood Recording for Static and Dynamic Constraint Satisfaction Problems*, International Journal of Artificial Intelligence Tools, vol. 3, no. 2, pp. 187-207, 1994.
45. Schiex, T. & Verfaillie, G., *Stubborness: a possible enhancement for backjumping and nogood recording*, in Cohn, A.G. (Ed.), Proc., 11th European Conference on Artificial Intelligence, John Wiley & Sons, Amsterdam, pp.165-169, 1994.
46. Smith, B.M., Brailsford, S.C., Hubbard, P.M. & Williams, H.P., *The progressive party problem: integer linear programming and Constraint Programming compared*, Constraints, Kluwer Academic Publishers, Boston, Vol.1, Nos.1&2, pp. 119-138, September 1996.
47. *Syllabus Plus*, Scientia Ltd, Cambridge, U.K. Web site: www.scientia.com
48. Trick, M. A., *A Schedule-then-Break Approach to Sports Timetabling*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
49. Tsang, E.P.K., *Foundations of Constraint Satisfaction*, Academic Press, 1993.
50. Tsang, E.P.K., Mills, P., Williams, R., Ford J. & Borrett, J., *A Computer-Aided Constraint Programming System*, The First International Conference on The Practical Application of Constraint Technologies and Logic Programming, pp. 81-93, London, 1999.
51. Van Hentenryck, P. *The OPL Optimization Programming Language*, The MIT Press, 1999.
52. Wallace, M., *Practical applications of Constraint Programming*, Journal of Constraints, Kluwer Academic Publishers, Boston, Vol.1, Nos.1&2, pp. 139-168 1996.
53. White, G. M., *Constraint Satisfaction, Not So Constraint Satisfaction and the Timetabling Problem*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
54. White, G. M. and Xie, B. S., *Examination Timetables and Tabu Search with Longer Term Memory*, in Proceedings of the 3rd International Conference on The Practice and Theory of Automated Timetabling, Constance, Germany, Burke and Erben (Eds.), August 2000.
55. Yoshikawa, M., Kaneko, K., Nomura, Y. and Watanabe, M., *A Constraint-Based Approach to High-School Timetabling Problems: A Case Study*, Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference, AAAI Press/ MIT Press, pp. 1111-1116, 1994.
56. Zervoudakis, K. and Stamatopoulos, P, *A Generic Object-Oriented Constraint Based Model for University Course Timetabling*". Proceedings of the 3rd International Conference on the Practice and Theory of Automated Timetabling PATAT 2000, pp. 128-147, Constance, Germany, Burke and Erben (Eds.), August 2000.

APPENDIX – A SAMPLE TIMETABLE

Since the problems addressed by the system are essentially design problems, there are generally multiple solutions that will satisfy any given set of constraints. From these, the system will give the first solution it encounters. Obviously if, for some reason, the user prefers some later solution, the given constraints will have to be manipulated appropriately so that solution comes out first.

The reader will appreciate that the example presented below is made deliberately simple for illustration purposes. Even from this modest example, the readers should still be able to form a good idea about the complexity of the task. While the timetable generation task is now the responsibility of the system, the administrator still has to prepare a voluminous amount of input data. But, compared with timetable generation, this is just a basic data-gathering task, which has to be done anyway and is not as problematic.

I. INPUT DATA

The system requires the following lists of input data (see illustration in table below):

	Mon	Tue	Wed	Thu	Fri	Sat
8:00	L206 L206		L206 L206			
9:00	L206	H204	L206	H204		
10:00		H204 H204		H204 H204		
11:00				▨		
12:00				▨		
1:00						
2:00						▨
3:00					▨	▨
4:00					▨	▨

Available Times & Preset Courses

a. Available Times

For the purposes of this example, the weekly times that are available for scheduling are Monday to Saturday, from 8:00AM to 5:00PM everyday.

b. Time Constraints

Within these times, the administration will require, for one reason or the other, that classes may not be scheduled at certain times during the week. We will take these to be Thursday 11:00-1:00, Friday 3:00-5:00 and Saturday 2:00-5:00.

c. Preset Courses

In addition to the above, the administration might require few courses to have preset times. That is, it does not want to leave it up to the generator to decide times for these courses. This is usually done for a few special courses. We will take these to be:

L206: Monday 8:00-9:30 and Wednesday 8:00-9:30
H204: Tuesday 9:30-11:00 and Thursday 9:30-11:00

d. Students Information

Finally, we need the list of students $(S_i)_{I=1..13}$ and the list of courses each is enrolled in:

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13
H204					✓	✓	✓	✓	✓	✓	✓	✓	✓
L206	✓	✓	✓	✓	✓	✓	✓	✓	✓				
C101	✓	✓	✓	✓									
C102	✓	✓	✓	✓									
C201					✓	✓	✓	✓	✓				
C204					✓	✓	✓	✓					
C302										✓	✓	✓	✓
C307										✓	✓	✓	✓
E102	✓	✓	✓	✓									
E204					✓	✓	✓	✓	✓				
M106	✓	✓	✓	✓									
M201					✓	✓	✓	✓	✓	✓	✓	✓	✓
P101	✓	✓	✓	✓									

Student Enrollment Information

e. Courses to Schedule

This is the list of all courses we need to determine a time for. Each such course should be listed with its associated number of credits. This latter number indicates the length of time this course takes and the number of slots this time may be split over (For an idea on that, see Course-Slot Distribution Table given earlier in the main body of the paper).

The list of course-credit pairs is:

C101	C102	C201	C204	C302	C307	E102	E204	M106	M201	P101
4	3	3	3	3	3	1	4	4	4	3

Course-credit list

f. Instructor Information

Instructor	Course Assignment	Availability
C1	C101 & C102	Mon, Wed & Fri
C2	C201 & C307	Tue & Thu
C3	C204 & C302	Tue & Thu
E1	E102 & E204	Sat only
M1	M106 & M201	Anytime
P1	P101	Tue & Thu Afternoons

The above is a table of instructor information. This includes the courses assigned to the instructor plus the times this instructor can be available for teaching at the university.

II. THE TIMETABLE OUTPUT

	day	from	to	day	from	to
E204	Sat	8:00	11:00			
E102	Sat	11:00	2:00			
C101	Mon	9:30	11:30	Wed	9:30	11:30
C102	Mon	11:30	1:00	Wed	11:30	1:00
P101	Tue	1:00	2:30	Thu	1:00	2:30
M106	Mon	1:00	3:00	Wed	1:00	3:00
M201	Mon	9:30	11:30	Wed	9:30	11:30
C201	Tue	8:00	9:30	Thu	8:00	9:30
C204	Tue	1:00	2:30	Thu	1:00	2:30
C302	Tue	8:00	9:30	Thu	8:00	9:30
C307	Tue	1:00	2:30	Thu	1:00	2:30

The table above is the timetable generated by the system from the above input data. Note here that, the hardest constraints to overcome are the times imposed by the instructors, especially where some instructors are teaching two different groups of students. However, this is not necessarily always the case.

Note the way these courses are ordered in the list. This is the order generated by the system: the most constrained course comes first. Moreover, the system is able to guarantee the same starting time in two different days for the same course.

III. OTHER VARIATIONS

Note that the E courses can be taught on Saturdays only, because the corresponding instructor is not available on any other day. For this reason, if we add the constraint that student S9 (for example) cannot be at the university on Saturdays between 8:00 and 11:00, then the same timetable as above will be generated, except that the times of the courses E102 and E204 will be swapped.

On the other hand, if the constraint is that this same student can not be at the university at all on Saturday, then a conflict-free timetable will be impossible to generate. In this case, the system will use its approximate reasoning feature and generate the same timetable as above. This excludes student S9 from attending the E204 course.