First International Conference
Modelling and Development of Intelligent Systems
Sibiu - Romania, 22-25 October, 2009

A Constraint-Based Approach to the Timetabling Problem

Cristian Frăsinaru

Abstract

It is well known that timetabling problems are usually hard to solve and require a lot of computational effort. There are many theoretical models that address this type of problems and various algorithms have been developed in order to attempt to solve them efficiently. However, it is not easy at all to apply these models to real life situations. This paper presents a solution to create an universal constraint-based model for representing the timetabling problem that can be applied in universities, schools or any other similar domain. Once the model is created it can be effectively solved with any CSP solver. We have used our own CSP solver, called OmniCS[1] (Omni Constraint Solver), that allows an incremental, human-aided approach to the timetabling problem, which proved very useful in practical applications.

1 Introduction

A timetabling problem can be defined as the scheduling of some activities during a certain period of time. Each activity has a set of properties, like the participants who attend it or the resources it requires, and it is subject to certain restrictions regarding its possible planning. University timetables for instance must manage entities like courses, students, teachers and rooms in order to create a mapping between courses and the time-slots of the week. Usually, timetables cycle every week or every fortnight but this will not become a requisition of our model.

Traditionally, the problem is solved manually and it is a tedious job that requires days or even weeks. Automated building of the timetables is also very difficult because there are many types of restrictions that must be accounted for and it is not easy to express them in computational forms.

The timetabling problem has been studied intensively since the sixties ([11]) and different techniques for solving combinatorial problems have been used, such as graph coloring ([3]), integer programming ([13]), simulated annealing ([1]), tabu-search ([5]) or genetic algorithms ([2]). A survey can be found in ([16]).

Despite the fact that these methods have given good results, using them in real life applications was not easy and quite counterintuitive not only because the complexity of the restrictions could not be formalized properly but also because solving algorithms could not be modulated to follow human judgement, an aspect which is very important in the interactive creation of the timetble.

In recent years, many computationally difficult problems from areas like planning and scheduling have been proven to be easily modelled as constraint satisfaction problems (CSP) ([6], [18]) and a new programming paradigm emerged in the form of constraint programming, providing the opportunity of having declarative descriptions of CSP instances and also obtaining their solutions in reasonable computational time. As a result, constraint satisfaction techniques have been applied to the timetabling problem ([10], [15]).Because constraint programming received very much attention also from the industry, a lot of CSP

[1] *Omnics* is freely available at `http://omnics.sourceforge.net`

solvers emerged, i.e. applications that offer solutions to model a problem using constraints and also an engine able to solve it. To give only a few examples, we can mention Ilog ([12]), Minion ([14]), Choco ([4]) or our solver OmniCS ([7]).

This paper presents a solution to create an universal constraint-based model for representing the timetabling problem that can be applied in universities, schools or any other similar domain. Once the model is created it can be effectively solved with any CSP solver. We have used our solver OmniCS because it has distinctive features that make it appropriate for the timetabling problem, like the fact that it allows human interaction in the process of finding a solution ([8]).

2 The Timetabling Problem

We attempt to describe the timetabling problem in a very flexible way such that the model we create could be applied in an uniform manner in any domain that requires this specific type of planning.

The main entities that we deal with are: *events*, *actors*, *resources*, *restrictions* and a *temporal domain* that contains the available time-slots.

2.1 The Events

We call *events* the atomic activities that must be scheduled. The property of atomicity specifies the fact that we are developing a model in which the events are continuous, they cannot be interrupted and resumed later. This is the most common situation in timetabling problems. However, there may be situations, like planning the proceedings of a conference or workshop, when activities are not continuous but fragmented during one or several days. In these cases, because the length of the fragments are known, we will use multiple events to describe one complex activity and add appropriate constraints.

We have considered two categories of events:

- *Iterative* - the most common situation in schools or universities; here the temporal domain is usually the carthesian product of the available week's days and the possible starting hours, for instance:

$$\{Monday, \ldots, Friday\} \times \{08:00, \ldots, 19:00\}$$

  Most of the events will repeat by a specified number of times, usually the number of weeks in a semester, and the timetable will be complete once we know the starting date and the number of repetitions.

- *Unique* - used for representing events that will occur only once, like exams for instance; in this case, the temporal domain contains explicitly specified pairs of dates and hours.

Regardless of the nature of the timetabling problem, the main features of an event are:

- *the length* - the required time for its completion, measured in an abstract manner;
- *the number of repetitions*, only for the events that are iterative;
- *the frequency*, once a week or once at two weeks, only for the events that are iterative;
- *the participants* that will attend to this event;
- *the resources* that are required, described at generic level;
- *the constraints* - specifications that will restrict the possible solutions for scheduling this event.

Scheduling an event involves the following operations:

- setting a value from the temporal domain: "the class x will be held at the moment t";
- assigning the resources: "the class x will take place in the room s";

- satisfying the constraints that apply to this event.

We say that an event is *resolved* if it has been scheduled.

Let e an event and T the temporal domain of the problem. If e is resolved, we note $start(e) \in T$ the moment when it begins and $end(e) \in T$ the moment when it is completed, otherwise $start(e)$ and $end(e)$ are undefined. We also note $week(e) \in \{0, 1, 2\}$ the week in which the event will take place (0 - all weeks, 1 - odds, 2 - even).

The fact that two events e_1 and e_2 are *concurrent* can be written as a disjunction $concurrent(e_1, e_2) = C1 \vee C2$, where

$$(C_1)\ start(e_1) < end(e_2) \wedge start(e_2) < end(e_1)$$

is the condition that the events will not overlap inside a week and

$$(C_2)\ week(e_1) = 0 \vee week(e_2) = 0 \vee week(e_1) = week(e_2)$$

is the condition that checks that the events take place at the same parity. We say that two events are *independent* either if at least one of them is not resolved or they are not concurrent.

2.2 The Temporal Domain

We call *temporal domain* or *temporal space* a set of integers $T = \{t_1, ..., t_n\}$ representing in an abstract manner time-slots of a specified calendar. The events that must be scheduled will be assigned values from this domain in order to be considered resolved. In this representation, the integer 1 will signify the temporal unit of the domain and the events will have their lengths specified as a number of temporal units, as oppossed to specifying them in minutes. Each temporal domain will be assigned a mapping function responsible with transforming its elements into real dates.

Let us consider a common representation of a temporal domain in the case of a timetable whose activities are iterative over a specified number of weeks. In this situation the structure of the week is the same throughout the whole period, an event beeing scheduled on a certain day and at a certain hour (each day has a start hour and an end hour). Let us note n the number of temporal units available every day and m the number of days in a week. The domain would contain all the integers in the interval $[0, nm - 1]$, as suggested in the following graphical representation:

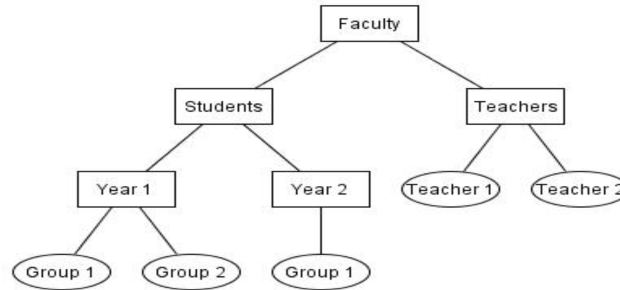
0	n	$\dots$	$n(m-1)$
1	$n+1$	$\dots$	$n(m-1)+1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	$2n-1$	$\dots$	$nm-1$

If we note h_0 the hour when the classes begin and d the length in minutes of a temporal unit, then for an element $t \in T$ the mapping function will be defined as $date(t) = (day(t), hrs(t), min(t))$, where $day(t) = 1 + [t/n]$, $hrs(t) = h_0 + (t - n[t/n])(d/60)$ and $min(t) = (t - n[t/n]) - 60(hrs(t) - h_0)$.

2.3 The Actors

Each event involves a number of entities, some of them beeing assigned as preordained properties of the event, while others are alloted dynamically depending on various conditions. We call *actor* or *participant* an entity that is assigned to some event *prior to* the process of creating the timetable, for instance a teacher or a group of students. Usually, actors are shared among events so they will be subject to constraints that prevent, for example, an actor beeing in two places at the same time. The set of actors may contain elements that are not independent regarding the inclusion, so representing actors in a unified structure is very important for determing the relationships between them. From the point of view of the *event-actor* association, we shall consider that an event e may have any number of actors, usually at least one, noted as $actors(e)$. In case of university timetables there are many situations when more than one group of students attend a lecture or more teachers have to participate to a certain event, such as a scientific meeting.

We define the *inclusion tree* as a structure that both enumerates all the actors that will participate to the timetable's events and surprises the relation of inclusion between them. The leaves of the tree will represent *atomic actors* that is persons or groups of persons that are regarded as a whole and the internal nodes will represent *composite actors*, precisely the union of all their leaves. An example of such a tree is given below:



An event may have actors assigned to it from any level of the inclusion tree, either atomic or composite. We say that two actors a_1, a_2 are *independent* if $a_1 \cap a_2 = \emptyset$. It is easy to see that if a_1 and a_2 are not independent then there exists a path from the root of the inclusion tree to a leaf, that contains both a_1 and a_2. Of course, when we schedule two events at the same time their actors must be independent.

2.4 The Resources

We call *resource* or *specific resource* an entity that is assigned to some event dynamically *during* the process of creating the timetable, for instance a room, a projector or some other didactic equipment. *The capacity* of a resource is the number of events that have this resource assigned to them and can be held at the same time. Usually, the capacity is 1 and this makes sense especially for rooms, where obviously only one event can be placed at a specific time. When it is not 1, the capacity of a resource represents a number of concrete items handled in an uniform manner; for instance the resource may be "Projector" with capacity 3, meaning that there are actally 3 different projectors available in the storehouse. If we place 4 courses on Monday at 08:00 and all require projectors, then we have a problem. We call a resource with capacity 1 *simple*, otherwise we say that it is *cumulative*.

A *generic* resource is either a single specific resource or a set of specific resources. For instance, the resource "Laboratories" may be the set of all available laboratories, say $\{L_1, L_2, L_3\}$.

An event may require some generic resources in different quantities and this is established at design time, before we begin the process of creating the timetable. We note $res(r)$ the resources required by an event e. The event e must be held in a laboratory, so it has assigned the generic resource "Laboratories". A resolved event must be assigned specific resources, in the same quantities as requested, so when we place the event e on the timetable we have to choose one of $\{L_1, L_2, L_3\}$ and assign it to e.

3 Constraint Satisfaction Problems

Once we have an informal description of the timetabling problem, we must represent it in a CSP specific manner.

A *constraint network* ([6]) is a triplet $\mathcal{R} = (X, \mathcal{D}, \mathcal{C})$ where:

- $X = \{x_1, ..., x_n\}$ is a finite set of variables;
- $\mathcal{D} = \{D_1, ..., D_n\}$ represents finite domains that are associated to variables of X;
- $\mathcal{C} = \{C_1, ..., C_t\}$ is a finite set of constraints.

A *constraint* C_i is a relation R_i on a subset of variables S_i, $S_i \subseteq X$, which denotes their simultaneous legal value assignments. In the above definition there is no restriction on the types of variables, their domains may be integers, strings or anything else. There are also no specifications on how constraints are defined. The purpose of a constraint is to restrict the possible values a variable x_i can be assigned from D_i.

The *instantiation* of a variable is the process of assigning it to a value from its domain.

If $\mathcal{R}$ has at least one solution, we say that it is *satisfiable* or *consistent*.

A partial instantiation of a set of variables S is *consistent* if and only if it satisfies all the constraints defined only over variables already instantiated.

In order to represent the timetablig problem as a network of constraints we have to identify the variables, the domains and the constraints. As in most cases, there are more than one formal models of the real life problem and, even if they are equivalent from a theoretical point of view, they may have an important impact in the process of obtaining the solution. Two main directions emerge:

a. the variables represent *events* and their values will be elements from the temporal domain;

b. the variables represent *time-slots* and their domains will be the set of events.

In our approach we have chosen the first representation, not only because it allows us to include in the model events that are not iterative but because we wanted to keep the size of the variable domains smaller than the number of variables. Empirically, this seems to help the solving process.

3.1 Variabiles and Domains

For each activity e we consider a single variable noted also with e. Normally, the domains of the variables would be sets of elements of the temporal domain of the problem. But, because each event requires a set of resources, such as rooms or didactic equipment, it proved to be better to include these specifications into the domain of the variables.

Let us consider a variable e, $T = \{t_1, t_2, ..., t_k\}$ the temporal space of the problem and $\{R_i | i = 1, p\}$ the required resources. Then, we represent the domain of e as the carthesian product:

$$D_e = T \times R_1 \times R_2 \times ...R_p$$

For example, let us consider the following partial description of an event: *"The class will be held in one of the laboratories $\{L_1, L_2\}$, it requires a projector P and one of the special equipments $\{E_1, E_2\}$"*. If the temporal space would be $T = \{t_1 = Monday(10-12), t_2 = Tuesday(12-14)\}$ then the domain of the event's variable will contain eight elements, respectively:
$\{(t_1, R_1, P, E_1), (t_1, R_1, P, E_2), (t_1, R_2, P, E_1), (t_1, R_2, P, E_2),$
$(t_2, R_1, P, E_1), (t_2, R_1, P, E_2), (t_2, R_2, P, E_1), (t_2, R_2, P, E_2), \}$

Most of the times, the number of resources required by an event is small and this prevents the domains of the variables to become oversized.

If the event takes place once at two weeks then its domain will have to reflect this also. In this case, the domain we consider for the event is:

$$D_e = W \times T \times R_1 \times R_2 \times ...R_p$$

where $W = \{1, 2\}$ represents the week (1-odd, 2-even).

3.2 The Constraints

The constraints must represent in the first place the restrictions of the timetable that should not be violated at any time, like the fact that an actor cannot participate at two events at the same time or a simple resource cannot be assigned simultaneously to different activities. These are the *hard constraints*.

In order to create a timetable that is to be accepted by all the participants it is not enough to satisfy only the hard constraints. Nobody will be happy if they have 12 hours of classes in a row or a day that is

very fragmented. Of course, teachers or special study groups might have preferences regarding the time-slots that are acceptable for them in order to attend their classes. However, these type of restrictions could be violated if they lead to the imposibility of creating the timetable (suppose for instance that two teachers require the same room and the same time-slot for their classes). Because of that, we call them *soft constraints* ([17]).

The classical model of constraint satisfaction is defined over the premises that identifying a solution means satisfying all the constraints of the problem. But there are many real life situations that cannot be solved this way, either because they are *over-constrained* ([9]) and thus not consistent or because their restrictions cannot be imposed in a *yes-or-no* manner. A good example of such a problem is the timetabling problem. In a "perfect" solution all these preferences would be satisfied but in most cases this is not feasible and we are concerned in creating a timetable that is "as good as possible", in other words minimizing somehow the number or the magnitude of the constraints that are not satisfied. Considerring the soft constraints, the timetabling problem is usually represented as an optimization problem that uses some *valuation structure* in order to determine when a solution is better than another. This structure must specify levels of preferences that are assigned to constraints, meaning how bad is it if a constraint is violated, and an operator that "combines" levels of preferences, specifying how good is an instantiation with respect to satisfying all the constraints of the network. The result is called the *degree of satisfaction* offered by an instantiation.

From the practical point of view we have designed an XML representation of the restrictions that apply to the timetabling problem. Each actor will have a *descriptor file* that formalizes its preferences and also the resources required by his events. A special parser will transform these descriptions into constraints specific to the OmniCS solver, which are actually Java classes. Each restriction might have a penalty assigned. The presence of the penalty produces a soft constraint, otherwise the constraint is considered to be hard.

Some of the constraints we have implemented are listed below.

Compatibility constraints

We say thet two resolved events e_1 and e_2 are *compatible* if they do not share actors or generic resources (decided at design time) or they are not concurrent (decided at runtime):

$$((actors(e_1) \cap actors(e_2) = \emptyset) \wedge (res(e_1) \cap res(e_2) = \emptyset)) \vee independent(e_1, e_2)$$

Thus, for every pair of events that share actors or resources, we must create such a hard constraint. Compatibility constraints represent the most numerous set of constraints so their implementation is critical for the performance of the solving process.

Resource Capacity Constraints

This type of constraint imposes that the number of concurrent events that share a cumulative resource does not exceed the capacity of that resource. Let r be a resource with capacity $c(r)$. Let us note $\{e_1, ..., e_k\}$ a set of concurrent events each using $p(e_i)$ units of resource r. The constraint states that:

$$\sum_{i=1,k} p(e_i) \leq c(r)$$

For every resource we have to create such a global constraint defined over the set of all variables representing events that require that resource.

Resource Disponibility Constraints

Another type of hard constraints is related to the fact that certain resources are not available throughout the whole temporal space of the problem. This is a situation common for faculties that have different timetables but share some rooms, each faculty having alloted only some days or time intervals to hold classes in that rooms.

For every restricted resource we have to create a global disponibility constraint over the events that require the resource, specifing also the *temporal subspace* that is available. This is a subset of the temporal space and can be represented as an array a having hd elements where h is the number of temporal units of a day and d is the number of days in a week, containing the following elements:

- 0 - allowed, with no penalty

- $+\infty$ - strictly forbidden
- $0 < c < \infty$ - allowed, with penalty c; in that case the constraint is soft, otherwise it is hard.

Actor Disponibility Constraints
The majority of participants have preferences regarding the time-slots that are to be assigned to their events. These types of preferences might be of common sense, like not having more than six hours in a day, or of personal nature, like not beeing able to hold classes too late in the evening. These restrictions can apply to all events of an actor or only to subsets of them. As in the case of resources, for each actor and each of his preferences we have to create a constraint that accepts as argument the temporal subspace that is available. We have designed a descriptive, easy to use method to specify this constraint:

```
<prefs actor="A">
    <event id="E1, E2">
        <res id="Laboratories"/>
        <include day="1,2" hour="8,10,12"/>
        <include day="5"/>
    </event>
    <event id="E3">
        <exclude day="1"/>
        <exclude day="2" penalty="10"/>
    </event>
</prefs>
```

If no penalties are specified, this unary constraints will have the immediate effect of reducing the domain of the variables and they will be removed from the network of constraints.

Sequence and Ordering Constraints
This type of constraints imposes that a set of events should take place in a compact block. We use the directive `linked` to specify that a set of events should form a sequence and we can add special directives like `order`, `first`, `last` to define a partial relation of ordering over these events. The exemple below states that all three events E_1, E_2, E_3 should take place in the same day, but E_1 should precede E_2.

```
<prefs actor="A">
    <event id="E1,E2,E3">
        <include day="1,2,3"/>
        <linked/>
    </event>
    <order events="E1,E2" first="E1" last="E2"/>
</prefs>
```

Limitation Constraints
These are constraints that limit the number of days or hours in which a set of events should take place. They are usually applied globally over the set of events of an actor, like in the following example that states that actor A should have classes at most two days of the week and the number of hours should not exceed six per day.

```
<prefs actor="A">
    <days max="2"/>
    <hours max="6"/>
</prefs>
```

The document type definition (DTD) of the XML model we have designed for representing the timetabling problem is described below:

```
<!ELEMENT prefs (event|days|hours|linked|order)*>
<!ATTLIST prefs actor CDATA #IMPLIED>
<!ELEMENT event (res|include|exclude|days|hours|linked|order)*>
```

```
<!ATTLIST event id CDATA #IMPLIED>
<!ELEMENT res EMPTY>
<!ATTLIST res id CDATA #IMPLIED>  <!-- Specifies the required resources -->
<!ELEMENT include EMPTY>
<!ATTLIST include                 <!-- Specifies a temporal subspace -->
    day CDATA #IMPLIED            <!-- One ore more days, comma separated -->
    hour CDATA #IMPLIED           <!-- One ore more hours, comma separated -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
<!ELEMENT exclude EMPTY>
<!ATTLIST exclude                 <!-- Excludes a temporal subspace from domain -->
    day CDATA #IMPLIED            <!-- One ore more days, comma separated -->
    hour CDATA #IMPLIED           <!-- One ore more hours, comma separated -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
<!ELEMENT linked EMPTY>
<!ATTLIST linked                  <!-- Specifies that some events form a sequence -->
    events CDATA #IMPLIED         <!-- One or more events, comma separated -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
<!ELEMENT order EMPTY>
<!ATTLIST order                   <!-- Specifies that some events are ordered -->
    events CDATA #IMPLIED         <!-- One or more events, comma separated -->
    first CDATA #IMPLIED          <!-- The events that should take place before -->
    last CDATA #IMPLIED           <!-- The events that should take place after -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
<!ELEMENT days EMPTY>
<!ATTLIST days                    <!-- Limitation of days within a week -->
    events CDATA #IMPLIED         <!-- One or more events, comma separated -->
    max CDATA #IMPLIED            <!-- Maximum number of days -->
    min CDATA #IMPLIED            <!-- Minimum number of days -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
<!ELEMENT hours EMPTY>
<!ATTLIST hours                   <!-- Limitation of hours within a day -->
    events CDATA #IMPLIED         <!-- One or more events, comma separated -->
    max CDATA #IMPLIED            <!-- Maximum number of hours -->
    min CDATA #IMPLIED            <!-- Minimum number of hours -->
    penalty CDATA #IMPLIED>       <!-- Penalty inflicted by not satisfying this -->
```

4 The Solving Process

Once a problem is represented as a CSP instance we have to use a CSP solver in order to obtain a solution or to discover that it is inconsistent. We have used our own CSP Solver, called OmniCS ([7], [8]), which has some distinctive features that make it appropriate for approaching effectively the timetabling problem, like the abilities to solve both classical and soft constraint satisfaction problems in an uniform manner and to provide a mechanism for controlling the whole process of systematically searching the solution. The basic structure of the algorithm we have used to develop our solver OmniCS is *backtracking* that uses a flexible filter-and-propagate mechanism in order to obtain efficiency. In order to solve both classical and soft CSP instances the solver also uses a *branch-and-bound* algorithm.

The systematic search algorithm must make a series of decisions in order to explore the search space. A *forward strategy* is responsible with selection of the next variable that will be instantiated, thus defining a relation of ordering over the whole set of variables. However, this order is not static and can be specified during execution depending on specific conditions that can be evaluated only at runtime.

For this specific problem we have used a strategy that selects first variables with the smallest domains, thus reducing the width of the search space. We considered more important the temporal nature of the domain and for each event a *tightness* factor was computed, representing the degree of temporal

restrictions that apply to it, according to the preferences of its actors. A value close to 0 means very restrictive, while a value near 1 means very relaxed ($tightness \in [0, 1]$). The solver will instantiate variables in an order based on the tightness values of their corresponding events, starting with the lowest value. The second ordering criteria is *maximal adjacency* (we say that two events are *adjacent* if they share at least one actor). The solver selects the variable that has as many adjacent resolved events as possible; instantiating such a variable is likely to determine a better behavior of filter-and-propagate algorithms.

An *assignment strategy* is responsible with defining a relation of ordering over the values of a variable's domain. As in the case of the forward strategy, this relation can be defined dynamically during execution. We have used an assignment strategy that ensures a low fragmentation of the timetabling, from the actors point of view.

A *backward strategy* defines how the solver will select the variable from which it will resume the search process, after a failure was detected. Here, we have attempted to use an heuristic that attempts to identify the real variable whose current instantiation is responsible for the current failure and return to it rather than to return to the last chronologically variable instantiated before the one that provoked the failure. Unfortunately, this was time consuming and did not improve the solving process very much.

Using these strategies, for a problem with around 300 events and more than 20.000 constraints, that was not over-constraint, the solver OmniCS was able to find a solution in a couple of minutes with a very small number of backtracks.

However, creating a benchmark was not our interest here. In real life applications, there are a number of limitations concerning the process of creating the timetable as a monolith. Traditionally, the algorithms for solving this type of problems have been designed considering that the network created after the modelling phase is in its final form, that is assuming that it is static during the solving process. In a real life situation, it is not uncommon that in the middle of the solving process we receive additional information about the preferences of some participant that must be added to the existing ones. It would be very frustrating if we had to start the whole process again, wasting all the computational effort performed so far.

Fortunately, OmniCS offers the possibility to change the problem dynamically. Not only it permits to rewrite the problem at runtime, without having to restart the whole process again, but it allows human intervention in the search process, overwriting the default behavior of the solver. We have used this facility in order to stop/resume the solving process, to adjust the partial solution manually, to add new constraints or to remove deprecated ones.

Also, because the solver generates events as it searches for solutions and informs special objects called *observers* it was easy to integrate it into a graphical user interface, especially designed for the timetabling problem.

5 Conclusion

The timetabling problem is a typical planning problem that is very tedious to solve manually but also difficult to approach with an automated solving technique. This complexity comes mostly from the numerous types of restrictions that have to be formalized and taken into account in order to satisfy all the educational requirememts and the various preferences of the participants.

Our goal was to create a set of specifications that allows the modelling of the timetabling problem in a flexible, declarative manner that is suited not only for universities, but also for schools or other similar domains. Then we show how to transform this model into a CSP instance, more precisely into a network of constraints, and how to employ constraint programming techniques in order to obtain a solution or to find the inconsistency of the network. For that, we have used our own CSP solver, called OmniCS, that was invoked in a dynamical and interactive fashion. Eventually, the combination of automated solving and human judgement proved to be very effective from the practical point of view, shortening very much the process of creating the timetable.

The software system that has emerged from this study was used successfully at Faculty of Computer Science of Iaşi, România, solving timetabling problems with more than 300 variables and 20.000 constraints.

References

[1] D. Abramson. Constructing school timetables using simulated annealing: Sequential and parallel algorithms, 1991.

[2] Edmund Burke, David Elliman, and Rupert Weare. A genetic algorithm based university timetabling system. In *East-West Conference on Computer Technologies in Education, Crimea, Ukraine*, pages 35–40, 1994.

[3] E.K. Burke, D.G. Elliman, and R. Weare. A university timetabling system based on graph colouring and constraint manipulation, 1993.

[4] Choco. http://choco.sourceforge.net.

[5] Daniel Costa. A tabu search algorithm for computing an operational timetable. *European Journal of Operational Research*, 76(1):98–110, July 1994.

[6] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[7] Cristian Frasinaru. Omnics. http://omnics.sourceforge.net.

[8] Cristian Frasinaru. Basic techniques for creating an efficient csp solver. *Scientific Annals of Computer Science*, pages 83–112, 2007.

[9] Eugene C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*, pages 278–283, 1989.

[10] Hans-Joachim Goltz, Georg Kchler, and Dirk Matzke. Constraint-based timetabling for universities. In *In Proceedings INAP'98, 11th International Conference on Applications of Prolog*, pages 75–80, 1998.

[11] C.C Gotlieb. The construction of class-teacher timetables. In *In Proceedings of IFIP Congress, North-Holland Pub. Co., Amsterdam, 73-77*, 1962.

[12] Ilog. http://www.ilog.com.

[13] N.L. Lawrie. An integer programming model of a school timetabling problem. *The Computer Journal*, 12:307–316, 1969.

[14] Minion. http://minion.sourceforge.net/.

[15] Hana Rudova and Keith Murray. University course timetabling with soft constraints. In *Practice And Theory of Automated Timetabling IV. Springer-Verlag LNCS 2740*, pages 310–328. Springer, 2002.

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

[17] Thomas Schiex. Soft constraint processing. First International Summer School on Constraint Programming, July 2005.

[18] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

Cristian Frăsinaru
University A.I.Cuza of Iaşi
Faculty of Computer Science
General Berthelot 16, 700483 Iaşi
ROMANIA
E-mail: *acf@info.uaic.ro*