A Constraint Logic Programming Approach to the Course Timetabling Problem Using ECLiPSe

Grace S. Tacadao
Ateneo de Davao University
Computer Studies Division
E. Jacinto St. Davao City
gstacadao@addu.edu.ph

ABSTRACT
Before the start of a semester, schools suffer from the difficulty of scheduling a number of resources such as classrooms and professors to a set of students. This takes weeks and months for most schools, hence, taking a lot of the school staff's valuable time. Because of its difficulty and time-consuming characteristic, a lot of companies and studies have aimed to automate the timetabling process. However, most of those automated packages are too general to fit the distinct requirements of the institution.

The studies that were conducted, on the other hand, are directed toward developing a most efficient solution based on existing frameworks that include heuristics that are derived from Operations Research (OR) and Artificial Intelligence (AI). This study explored the use of an approach known as Constraint Logic Programming (CLP), which was derived from the area of AI, in a solution to the timetabling problem using a language called ECLiPSe. The timetabling problem studied is that of the Computer Studies Division of the Ateneo de Davao University. The timetable produced satisfied all the hard constraints and has a significantly low cost using an objective function.

Keywords
Timetabling, Course Timetabling, Constraint Programming, Constraint Logic Programming, ECLiPSe

1 INTRODUCTION
One of the most difficult problems that universities face when a new semester is about to start is the allocation of classrooms for periods of time to courses. However, some of these courses are attended by the same students or handled by the same professor. Surely, they must not overlap. Thus, timetable construction has always been a difficult task for universities to complete.

Some universities have already started to use automated timetabling solutions to address this problem. There appear two approaches in the literature: (1) Purchase an off-the-shelf school scheduling software or (2) Develop their own solution based on a study of an established solution framework. The first strategy seemed to be too general to fit the distinct requirements of an institution [13].

The other method, however, offers more guarantee as many solutions have been successful in producing timetables efficiently. These solutions were based on established frameworks that mostly come from two areas of study: Operations Research (OR) and Artificial Intelligence (AI).

In the works that were derived from OR, the timetabling problem was most generally formulated using mathematical programming and was solved using local search and evolutionary heuristics. These OR methods have been the natural choice since they were developed precisely for optimization problems, such as timetabling.

Most of these OR-inherited search methods are based on statistics of the neighborhood around a given point in the search space [19]. Thus, they require some initial feasible solutions to the problem at hand. However, local search methods tend to get trapped in local optima, although this is sometimes resolved through relaxation and restart methods. Moreover, they also lack flexibility [6] [15] as some problem-specific constraints cannot be introduced easily.

These weaknesses, however, are seem to have overcome by an approach called Constraint Programming (CP), which has now a growing number of researches, particularly for the problem of timetabling. It stemmed from the area of Constraint Satisfaction Problem (CSP), an approach from the field of AI.

Constraint Logic Programming (CLP), a derivative of CP, has had a rising number of researches in the timetabling and optimization community.

It is in this notion that the proponent explored the viability of the CLP approach in the development of a timetable generation system for the Ateneo de Davao University (AdDU) Computer Studies (CS) Division.

2 CONSTRAINT LOGIC PROGRAMMING
2.1 Constraint Programming
Constraint Programming is a programming paradigm where relations between variables can be stated in the form of constraints. Algorithms are used to find a solution that satisfies all the stated constraints. They are also internal to the tool being used, thus, the logical implications, the values and the constraints on the variables are carried out implicitly. This is contrary to the OR-inherited methods where the algorithms have to be rewritten to fit the problem.

The solving algorithms are also based on chronological backtracking; hence, this guarantees that a solution can be returned, if one indeed exists. Moreover, the paradigm lays claim to the following advantages: fast prototyping, adaptability and code maintainability [6].

The variety of CP systems are classified by their domains, which specify the kind of constraints that can be written, the relationship between these constraints and the set of values that
variables in the constraints can take. Although constraints were already seen to be suitable for formulating and solving a lot of problems, only a few systems were in practical use at first.\[14\]

For constraints to be used in real-world applications, they have to be interfaced with existing systems. More precisely, the use of constraints has to be integrated with another programming discipline. The foremost integration was done with logic programming, thus, called Constraint Logic Programming. Integration with other paradigms followed, hence, resulted in constraint functional programming and constraint imperative programming.\[25\]

Since the interpretation of constraints is declarative in nature, the extension of logic programming with constraints proves to be more elegant and more natural. Logic programming does not follow a definite sequence of operations as do the imperative programming and concentrates on what to solve and not how to solve the problem.\[14\]. In fact, it was noted that logic programming is just a particular kind of constraint programming.\[25\].

The combination of Constraint Solving and Logic Programming made CLP flexible and expressive and in most cases, more efficient than other kinds of programs.\[16\]. It has also been known to be a practical tool for solving combinatorial problems. In effect, CLP has been successfully attempted in a diversity of applications from the analysis and synthesis of analog circuits, options trading analysis, to DNA sequencing and chemical hypothetic reasoning.\[16\]. However, its success in the area of automated timetabling in the recent years is particularly remarkable.

2.2 Constrain and Generate Methodology

The typical form of a constraint program is as follows:\[18\]:

1) First, the variables and their initial domains are defined.
2) Then, the constraints modeling the program are stated.
3) Finally, a labeling predicate is used to invoke a complete solver.

This is thus called the constrain and generate methodology; first, constraints are applied, then a solution is generated by labeling.

2.3 Constraint Logic Programming

2.3.1 Constraints

A constraint is a well-formed expression that comprise of symbols. These symbols may be variables or functions, in the same sense as in a mathematical expression. The constraint indicates the relationship between the variables through the functions. One such example is the expression $X \geq Y + 1$.

For a constraint to be evaluated, a variable is defined to take a set of possible values. These values, the functions and their meaning, as well as the rules for forming constraints are specified by a constraint domain. Finite constraint domains are domains in which the possible values that a variable can take are restricted to a finite set.

A user-defined constraint is of the form $p(t_1, \ldots, t_n)$ where $p$ is a n-ary predicate and $t_1, \ldots, t_n$ are expressions from the constraint domain. A literal is either a primitive constraint or a user-defined constraint. A goal, $G$, is a sequence of literals. $G$ has the form $L_1, L_2, \ldots, L_m$, where $m \geq 0$ and each $L_i$ is a literal. An empty goal is denoted by $\square$. A rule, $R$, is of the form $A \dashv \vdash B$ where $A$ is a user-defined constraint and $B$ is a goal. A fact is a rule with the empty goal as a body, $A \dashv \square$. A (constraint logic) program is a sequence of rules.\[18\]

A predicate $p$ in a program $P$ is the sequence of rules appearing in $P$ which have a head involving predicate $p$. The definition of rules allows predicates to be defined in terms of other predicates. This allows a predicate to be defined in terms of itself. It also allows a predicate to be defined by one or more rules.\[18\]

An algorithm that determines whether a solution to the constraint exists is called a constraint solver. A complete constraint solver can determine whether a constraint can be satisfied or not. However, in general, a solver may not be complete because (1) a complete algorithm is considered too expensive and (2) a complete solver is unknown or cannot exist. For these reasons, a solver can be incomplete, that is, it may answer with unknown, which indicates that it cannot determine whether a constraint is satisfiable or unsatisfiable.\[18\]

A Constraint Satisfaction Problem (CSP) consists of a constraint $c$ over variables $X_1, \ldots, X_n$ and a domain $D$ that maps each variable to a finite set of values.\[18\]

One of the simplest techniques in solving a CSP is by means of chronological backtracking. A variable is chosen and then for each value in its domain, the satisfiability of the constraint is determined by replacing the variable with the value. Good heuristics such as choosing the most constrained variable first can lead to smaller search trees.\[18\]

Node and arc consistency solvers are incomplete solvers that transform the CSP into an equivalent CSP with smaller domains. They are consistency-based since they propagate information based on the permissible domain values from one variable to another while maintaining the consistency in the constraint.\[18\]

A constraint $c$ is node-consistent with domain $D$ if either $\mathrm{vars}(c) \neq \emptyset$ or if $\mathrm{vars}(c) = \{x\}$, then for each $d \in D(x)$, $\{x \rightarrow d\}$ is a solution of $c$.\[18\]

A constraint $c$, on the other hand, is considered arc-consistent with domain $D$ if either $\mathrm{vars}(c) \neq \emptyset$ or if $\mathrm{vars}(c) = \{x, y\}$, then for each $d_x \in D(x)$ such that $\{x \rightarrow d_x, y \rightarrow d_y\}$ is a solution of $c$.\[18\]

In brief, for a constraint to be node-consistent, for each variable, the solutions to the constraints involving only that variable are intersected with its original domain to give the new domain. For a constraint to be arc-consistent, for each variable $x$, the algorithm takes each binary constraint involving $x$ and uses the domain of the other variable, say $y$, which appears in the constraint to restrict the domain of $x$.\[18\]

2.3.2 Modelling

The first phase in writing a constraint logic program is the translation of the problem’s high-level constraints to the primitive constraints available to the solver.

Cheadle et al.\[7\] put forth the following criteria that must be fulfilled by a good constraint model: expressive power, clarity for
humans and solvability for computers. A good model, then, does not significantly depart from the problem's formal description.

The first step in modeling is to identify the variables that will represent the parameters of the problem. Then, constraints are formed using these variables. To define a function that takes \( n \) arguments, a predicate is written that takes \( n + 1 \) arguments where the last argument is the value of the function. \([18]\)

It is also wise to use complex constraints, which are conjunctions of primitive constraints that are available to constraint solvers. They come with specialized propagation rules, thus, it makes constraint programs more efficient during domain pruning. \([18]\)

Three of such constraints are the following: \([18]\)

- \( \text{alldifferent}([V_1, \ldots, V_n]), \) which constrains the variables \( V_1, \ldots, V_n \) to take different values
- \( \text{element}([I, [V_1, \ldots, V_m]], X), \) which takes off the behavior of an array, i.e., if \( I = i, \) then \( X = V_i \)
- \( \text{cumulative}([S_1, \ldots, S_m], [D_1, \ldots, D_m], [R_1, \ldots, R_m], L) \) constrains the variables to satisfy a simple scheduling problem, where there are \( m \) tasks to schedule whose start times are \( S_1, \ldots, S_m \) and durations \( D_1, \ldots, D_m \), and which require \( R_1, \ldots, R_m \) units of a single resource, \( L \) units of which is only available at any one time

**2.3.3 Labelling**

To give a variable \( X \) its initial domain, a literal of the form \( X::[1\ldots4] \) is used. If a variable has not been initialized with a domain, it is assumed with some large default value.

Labelling is the assignment of domain values to variables. This is provided for by the constraint solver but is not automatically called by the CLP system due to its relatively high performance cost. This procedure should then be called explicitly by the programmer. Most CLP systems provide a complete constraint solver for FD constraints system using the built-in predicate labelling. It backtracks through each variable's domain by setting the variable to each of the values in turn. \([18]\)

There are two choices to be made when labeling: \([18]\)

1) Variable ordering

When labeling a list of variables, the variables with the smallest domains should be labeled first. Once these variables are assigned, the domains of other variables might be reduced.

2) Value ordering

It might be possible to label a variable with a value that would limit the domains of other variables and hence, would lead to a most likely solution. However, the choice is sometimes not intuitive and is very problem-specific.

**2.4 Constraint Logic Programming Languages**

In practice, two approaches have emerged in the use of constraint programming, the black-box approach and the glass-box approach. In the black-box approach, the constraint solver, which is an algorithm that determines the satisfaction of constraints, is hidden from the user. On the other hand, the glass-box approach enables the user to tailor the underlying solver to his needs.

However, there is really no clear separation between the two approaches as black-box languages can define the propagation of constraints as well as provide “hooks” in the unification algorithm and the combination of constraints with logical operators \([21]\). Thus, only the black-box approach is considered and used in the current study.

Some of the languages that were used in the black-box approach are CHIP \([13]\)[15], ECLiPSe \([9]\)[11] and Sicstus Prolog \([21]\) while the widely used language in the glass-box approach is CHR \([1]\)[2].

All of the languages above except CHR were reviewed for the current study. Aside from CHR being a glass-box language, it is made available as a library in another language, particularly Sicstus Prolog and ECLiPSe.

Inferring from past researches \([9]\)[11][13][15][21], a CLP language must have the following general attributes:

- Support for CLP for Finite Domains
- Has built-in complex constraints such as \( \text{element} \) and \( \text{alldifferent} \) or their equivalents
- Provides facilities for scheduling
- Provides facilities for processing data items
- Has built-in search heuristics
- Support for optimization facilities
- Has a certain “repair” strategy
- Support for some OR-inherited methods
- Provides interfaces with third-party languages

The following table then shows the summary of attributes of the three languages. It further shows that ECLiPSe has the noticeable advantage as the language to use in the study.

<table>
<thead>
<tr>
<th>Table 1. Summary of Attributes of CLP Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLP Language</strong></td>
</tr>
<tr>
<td>CHIP</td>
</tr>
<tr>
<td>ECLiPSe</td>
</tr>
<tr>
<td>Sicstus Prolog</td>
</tr>
</tbody>
</table>
3 ECLiPSe

3.1 Background

ECLiPSe (ECLiPSe Common Logic Programming System) is a system based on Prolog that acts as a platform for the integration of logic programming extensions, particularly CLP. The kernel of ECLiPSe is an efficient implementation of Edinburgh-like Prolog. It is built around an incremental compiler, which compiles the ECLiPSe source into WAM-like code and an emulator of this abstract code. [3]

The ECLiPSe logic programming system was originally an integration of ECRC’s SEPIA, Mega-Log and parts of the CHIP systems. It was then further developed into a Constraint Logic Programming system with a focus on hybrid problem solving and solver integration. [3]

ECLiPSe has an efficient incremental compiler which compiles Prolog source code into instructions for an abstract machine. This is then executed by an emulator. The compiler compiles at about 1000 lines/sec on a Sun-4, which makes the usual debugging cycle acceptably short. The ECLiPSe compiler is also interactive and incremental, which means that Prolog programs are compiled during an ECLiPSe session directly into the Prolog database. [3]

3.2 Constraint Libraries

3.2.1 Interval solver

The standard constraint solver offered by most constraint programming systems is the finite domain solver, which applies constraint propagation techniques developed in the AI community. ECLiPSe supports finite domain constraints via the ic library, which implements finite domains of integers, together with a basic set of constraints. [7]

In addition, ic also allows continuous domains in the form of numeric intervals and constraints between expressions involving variables with continuous domains. Integrality is treated as a constraint, which may be composed of continuous and integral variables in the same constraint. Specialised search techniques support the solving of problems with continuous variables. [7]

3.2.2 Global Constraints

The ic_global library supports a variety of constraints, each of which takes a list of finite domain variables as an argument. Such constraints are called “global” constraints. A few of the constraints that are available from the ic_global library are alldifferent/1, maxlist/2, occurrences/3 and sorted/2. [7]

3.2.3 Scheduling Constraints

Several ECLiPSe libraries implement global constraints for scheduling applications. The constraints take a list of tasks, which are denoted by the start times, durations and resource needs, and a maximum resource level. They reduce the finite domains of the start times by reasoning on resource bottlenecks. The three ECLiPSe libraries that implement scheduling constraints are ic_cumulative, ic_edge_finder and ic_edge_finder3. They implement the same constraints declaratively, but with different time complexity and strength of propagation. [7]

3.3 Search Methods

ECLiPSe has built-in backtracking and is therefore well suited for performing depth-first tree search. With combinatorial problems, naive depth-first search is usually not good enough, even in the presence of constraint propagation.

According to Cheadle et al [7], search methods that are used in practice vary according to exploration, assignments and randomness. In terms of exploration, there are two basic categories, complete and incomplete. In terms of assignments, two categories emerged, constructive and move-based. Randomness, on the other hand, refers to some random element used in the search.

A complete search means that the search space is explored exhaustively so that all solutions are found. An incomplete search, however, suffices when only a relatively good solution is needed. [7]

On the other hand, a constructive search builds a solution by constructing assignments incrementally while a move-based search moves between total assignments, wherein all variables are already assigned a value [7]. Noticeably, OR methods fall under the latter category.

Cheadle et al also presented some of the search methods and their properties, with respect to the above-mentioned categories.

<table>
<thead>
<tr>
<th>Table 2. ECLiPSe search methods and their properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>Full tree search</td>
</tr>
<tr>
<td>Credit search</td>
</tr>
<tr>
<td>Bounded backtrack</td>
</tr>
<tr>
<td>Limited discrepancy</td>
</tr>
</tbody>
</table>

ECLiPSe implements several variants of the complete and incomplete tree search methods via ic library's search routine, search/6. The search predicate also provides a number of pre-defined variable selection methods and some pre-defined value assignment methods.

The following are some of the pre-defined variable selection methods that search/6 provides. [7]

- input_order - The first entry in the list is selected.
- first_fail - The variable with the smallest domain size is first selected.
- smallest/largest - The variable with the smallest or largest value in the domain is selected.
- occurrence - The variable with the largest number of attached constraints is selected.
- most_constrained - The variable with the smallest domain size is selected. If several entries have the same size, the entry with the largest number of attached constraints is selected.
The following are some of the pre-defined value choice methods that search/6 provides. On failure, the previously tested value is removed unless otherwise specified. [7]

- **indomain** – Values are tried in increasing order. On failure, the previously tested value is not removed.
- **indomain_min**/**indomain_max** – Values are tried in increasing or decreasing order.
- **indomain_middle** – Values are tried beginning from the middle of the domain.
- **indomain_median** – Values are tried beginning from the median value of the domain.
- **indomain_split** – Values are tried by successive domain splitting. This enumerates values in the same order as **indomain** and **indomain_min** but may fail earlier.
- **indomain_random** – Values are tried in random order. Using this routine may lead to irreproducible results. However, seed/1 can be used to force the same number generation sequence in another run.

### 4 RESEARCH METHODOLOGY

Like all CLP programs, this research also followed the *constrain and generate* methodology but it was incorporated with the Large-Scale Combinatorial Optimization Structure presented by Simonis [23].

![Operational Framework](image)

**Figure 1. Operational Framework**

### 5 THE AdDU TIMETABLING

A month and a half before the start of a semester, the CS program coordinators, with the help of the division secretary, prepare the list of subjects that will be taken up by students in each year level in a degree program. A course will be opened for each section (group of students). Some sections that are considered too small for a class are merged with other sections for subjects they have in common. But sections are only merged as long as the total class size does not exceed 40 and as long as the instructional perspective is the same for the degree programs.

After the course list has been completed, the courses are then assigned to professors. Professional subjects, also called major subjects, are assigned to professors from the division. On the other hand, professors for core or non-professional subjects, also called minor subjects, are requested to the servicing department.

Once the courses have been assigned to professors, a timetable can now be manually produced. Courses are assigned a room for a period of time. At this point, since most of the rooms in the campus are administered by the Dean's Office, representatives from the divisions are convened to facilitate the room assignments. But some divisions, such as the Computer Studies, are privileged enough to have a roster of rooms under its supervision. These divisions have an amount of autonomy in the disposal of these rooms. As a result, a timetable can already be completed at their level.

#### 5.1 Schedules

The following two tables show the two set of schedules in a weekly timetable. Meetings of courses on the M schedule are held on Monday, Wednesday and Friday. The schedule is divided into ten 1-hour timeslots. The following are the start-times of these timeslots.

**Table 3. M Schedule Timeslots**

<table>
<thead>
<tr>
<th></th>
<th>MORNING</th>
<th>AFTERNOON</th>
<th>EVENING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7:40 AM</td>
<td>12:25 PM</td>
<td>5:50 PM</td>
</tr>
<tr>
<td>2</td>
<td>8:45 AM</td>
<td>1:30 PM</td>
<td>6:55 PM</td>
</tr>
<tr>
<td>3</td>
<td>10:00 AM</td>
<td>2:35 PM</td>
<td>8:00 PM</td>
</tr>
<tr>
<td>4</td>
<td>11:05 AM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Meetings on the T Schedule happen on Tuesday and Thursday. It is divided into eight 1.5-hour timeslots. The following are the start-times of these timeslots.

**Table 4. T Schedule Timeslots**

<table>
<thead>
<tr>
<th></th>
<th>MORNING</th>
<th>AFTERNOON</th>
<th>EVENING</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7:40 AM</td>
<td>1:00 PM</td>
<td>5:55 PM</td>
</tr>
<tr>
<td>2</td>
<td>9:15 AM</td>
<td>2:35 PM</td>
<td>7:30 PM</td>
</tr>
<tr>
<td>3</td>
<td>11:00 AM</td>
<td>4:10 PM</td>
<td></td>
</tr>
</tbody>
</table>

#### 5.2 Subjects

Subjects that are offered in the Computer Studies division can be categorized according to the number of units and the nature of instruction, specifically, whether a subject is held in a lecture room or in a laboratory room. The following are these categories:
(1) 3-unit lecture subjects, (2) 6-unit lecture subjects and (3) 5-unit lecture/laboratory subjects.

The number of units in the first two categories corresponds to the number of hours of contact per week. But for a 5-unit lecture/laboratory subject, 2 hours are given for lecture and the remaining 3 hours are given for the laboratory.

Each lecture subject is assigned a period in either the M schedule or the T schedule. A lecture/laboratory subject, on the other hand, may be assigned in both schedules.

Depending on the particular subject, the laboratory meeting may be assigned to 2-3 consecutive periods in a single day or to 1 period in a schedule. The second approach results to the 2-3 periods spread out in a week.

5.3 Rooms

The Computer Studies division has two types of rooms under its charge.

The first are the lecture rooms that are used for theoretical instruction. There are 6 such rooms and all have equal capacity. There are 5 that are provided with instructional tools such as a PC and an LCD projector.

The second are the laboratory rooms that are used for practical computer application. Like the lecture rooms, all rooms also have equal capacity. Aside from the computers and a printer that are provided for the students' use, some of the laboratories are also provided with an LCD projector and other equipments.

5.4 Constraints

A timetable is considered acceptable if it fully satisfies the following conditions, regarded as hard constraints.

1) Every section must only attend one lecture or laboratory session at any one time.
2) Every room must only hold one course at any one time.
3) Every professor must only teach one course at any one time.

The following are additional hard constraints that were identified for a CS division timetable:

4) A course that requires certain equipments must be assigned to a room that is provided with all the needed equipments.
5) If the laboratory component of a 5-unit lec/lab course is preferred to be scheduled as a straight 3-hour session, then the session should preferably be uninterrupted. This is avoided by restricting the lab session to be assigned on the following start times only.

6) If the lab component of a 5-unit lec/lab course, on the other hand, is preferred to be scheduled in the same way as a 3-unit lecture course, then the laboratory session must follow immediately after the lecture session.

Since a number of timetables can be produced that can satisfy the above-mentioned constraints, a most favorable timetable must also satisfy the soft or preferential constraints to a great, if not greatest, extent. The soft constraints that were identified for the CS division are the following:

1) Time preferences of courses and professors must be satisfied as much as possible.
2) It is preferred that schedules of sections have fewer gaps in between.
3) Faculty schedules must allow for breaks between periods for those teaching successively for more than three hours a day.
4) Daily section schedules should span for only 8 hours.

6 THE ECLiPSe MODEL

6.1 Model Design

The outline of the timetabling model is derived from the scheduling example presented in the book Programming with Constraints [18]. In this example, there is a list of tasks that need to be scheduled on a set of machines. Each task must be performed in a given period of time on a particular machine. Additionally, there might be other tasks that must be performed first before this task.

Each task in the example is presented as a structure of the form:

\[
task(name, duration, [names], machine)
\]

where name refers to the name of the task, duration is an integer indicating the number of minutes the task requires, names refer to the names of the other tasks that must precede this task and machine refers to the name of the machine where the task must be performed.

The problem instance is a list of task structures, as in the following:

\[
[task(j1, 3, [], m1), task(j2, 8, [], m1), task(j3, 8, [j4,j5], m1), task(j4, 6, [j1, m2], task(j5, 3 [j1, m2]),
\]

\[
task(j6, 4, [j1, m2])]
\]

The above problem instance is supplied as an argument to the query goal.

The course timetabling problem is fundamentally a scheduling problem. Each course is a task that requires a certain period of time to be completed. It may also need to be preceded by another course or courses.

However, although the scheduling model provides an invaluable starting point, the contexts of the two models pose key differences. While there is only one resource involved in the scheduling problem, there are three resources that need to be
considered in the timetabling problem. The said resources are the sections, professor and the classroom. The only resource (machine) in the above scheduling problem is also static since it was assigned at the outset while only two (professor, sections) of the three resources in the timetabling problem are static. The third resource (classroom) is dynamic and must be assigned by the program. Most importantly, the domains and constraints of the variables involved are not as straightforward as those in the scheduling problem.

The rest of the sections detail the phases of the model design.

6.2 Prepare Data

Data is formatted as valid Prolog terms in data files for the external data representation.

As mentioned, a list of all courses that will be offered is produced before the start of the semester. After the courses were assigned to professors, the timetabling process begins. This list of courses is the main input to the application. Other input files include the rooms and the professors.

All of the course, room and professors terms in a problem instance are stored in the respective .ecl file. The filename is passed as an argument to the query goal. The terms are then read using the read_data predicate, which is adapted from Simonis [23].

Each term is stored into a list and is then treated as an ECLiPSe structure in the constraint program.

6.2.1 Courses

The following are the attributes that appeared as arguments in the course term: Code, Name, Professor, Classes, Roomtype, Timeslots, Precedences, Equipments and Timepref. The course term is then written in the following form.

course(code, professor, classes, roomtype, timeslots, precedences, equipments, timepref).

The following are some samples of the course term:

course('1-365a', [ft8], [z1], lec, 1, [], [projector], [1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15, 16]).
course('4-82a', [ft3], [cs4, it4], lec, 1, [], [projector], []).

6.2.2 Rooms

A file containing all the available lecture and laboratory rooms is also needed in the application. This provides the rooms that can be assigned to courses.

Just like the courses, the rooms are also formatted as Prolog terms and then stored into data files. Each room term has the following arguments: Id, RoomNum, Purpose, EquipList. Just like the course term, the room term is also written in the following form.

room(id, roomnum, purpose, equiplist).

The following are samples of the room term.

room(6, f607, lecture, [projector, speakers]).

6.2.3 Professors

The list of professors can actually be extracted from the list of courses since the relevant professors are only those who have assignments for the semester. However, since the professor's time preferences must be noted, a separate file is needed.

Just like the courses and rooms, professor records are also written as Prolog terms. Each term has the following arguments: Id, Name and TimePref. Terms are written in the following form.

professor(id, name, timepref).

The following are samples of the professor term:

professor(ft3, "Teresa Quindoy", [1, 2, 3, 4, 5, 11, 12, 13]).

professor(ft4, "Rey Aliño", [1, 2, 3, 4, 5, 6, 7, 11, 12, 13, 14, 15]).

professor(ft5, "Sheila Nisperos", []).

6.2.4 Straight 3-hour Lab Courses

Before creating the variables for the solver, lab courses of 5-unit lecture/lab subjects that must be scheduled in a straight 3-hour period must be dealt with first. Note that such a course meets only once a week. Therefore, it cannot just be assigned a start time on the M or T schedule in the same manner as those courses with meetings that occur three times a week. Since the number of timeslots of these courses is 3, all lab courses of this nature alone would fill the two schedules. Such a scenario would not yield any solution. Thus, this must be treated with some consideration.

A section can have lab courses assigned to the same start time but on different days. For instance, section IT3 can have both IT 272 and IT 301 at 7:40 AM on an M schedule but IT 272 is scheduled on Monday while IT 301 is scheduled on Wednesday.

Using this notion, so-called special sections are identified. A section is regarded as special if it has three or more lab courses that must be scheduled in a straight 3-hour period. Those sections with less than three of these lab courses are not included since the said courses can be accommodated on their schedule without difficulty. The lab courses in question are identified as having a lab RoomType and 3 Timeslots.

Three lists are consequently built. These are: 1) Sections, which are extracted from the list of courses, 2) Special Sections, which in turn are derived from the list of sections, and 3) Special Courses, which were identified from the list of courses using the list of special sections and the roomtype/timeslot factor. These are accomplished by the predicates build_slist, build_spclist and build_spclist respectively. They are called as follows in the main predicate.

build_slist(Courses, [], Sections),
build_spclist(Courses, Sections, [], SpecialSections),
build_spclist(Courses, SpecialSections, [], SpecialCourses),

After identifying the straight 3-hour lab courses of the special sections, these courses are then deleted from the list of courses. Every two of them are merged into one course, regardless of whether the sections involved are the same or not. The deleted
courses are then replaced by merged courses in the list, reducing the number of these lab courses by a half. These are done through the predicates del_spcourses and handle_special. They are called in the main predicate as follows.

\[
deleted_spcourses(\text{SpecialCourses}, \text{Courses}, \text{Courses0}),
\]
\[
\text{handle_special}(\text{SpecialCourses}, \text{Courses0}, \text{NewCourses})
\]

### 6.3 Create Variables

Each course must be assigned a start time, a day and a room. The day assignment is particularly required for straight 3-hour lab courses as they only occur once a week.

Each course is transformed into a job for the solver. Each job is a term that contains the following arguments: \text{Code}, \text{Timeslots}, \text{StartTime}, \text{DayAssign}, and \text{RoomAssign}. Like the course term, a structure is also declared for the job term.

The variables that will be assigned by the solver are \text{StartTime}, \text{DayAssign} and \text{RoomAssign}. Each course has three such variables, each of which is given its initial domain.

The \text{StartTime} variable refers to the start time of a timeslot in the two schedules (see Tables 3 and 4). Its domain consists of the integer values from 1 to 18 with each denoting a start time in the two schedules. \text{StartTime} values from 1 to 10 refer to the ten start times on the M schedule while 11 to 18 refer to those on the T schedule. 3-unit lecture courses, 6-unit lecture courses and the lecture courses of 5-unit lecture/laboratory courses are initialized using the said domain.

The \text{DayAssign} variable is assigned only after the \text{StartTime} has been already assigned. Initially, each course is assigned either mwf if its \text{StartTime} is from 1 to 10 or tth if it is from 11 to 18.

For the straight 3-hour lab courses, however, the \text{DayAssign} variable is further assigned a day after the merged courses have been split.

Constraints are then enforced to ensure that courses that were previously merged do not get the same \text{DayAssign} value. The \text{DayAssign} is labeled afterward.

Finally, a course is assigned a room based on the type of room it needs and the equipments that it needs. To identify the rooms that match the requirements of a course, rooms are divided into those that are for lecture and those that are for laboratory. A list of all equipments that are provided in a room is built afterward. Finally, each room group is segregated based on each of the equipments.

The size of each sub-group is also determined. In particular interest is the size of the group of rooms without any equipment. If the size of this group is only one, then courses that do not need any equipment are not assigned to this only room since having only one room implies that the number of these courses should not exceed the total number of timeslots. Thus, these courses are instead initialized with the domain of all rooms. Such is the case for the lecture rooms.

The case is different for the laboratory rooms. Since the size of the group without equipments is two, the courses that do not need equipments are initialized with the rooms in this group.

For courses that need particular equipments, the groups of rooms that match the requirements are searched for and merged into one list. This list is consequently used to initialize the \text{RoomAssign} of the course.

### 6.4 Create Constraints

To ensure that the courses do not overlap, the cumulative complex constraint of the \text{ic_edge_finder3} library is used. The cumulative constraint constrains courses attended by a section or courses assigned to a professor to be assigned different start times.

Before using the constraint, all jobs are put together in a list. Their start times, durations and resource needs are subsequently extracted for the use of the constraint.

Each of the three hard constraints is enforced through the same scheme. Constraint 1 and 3 (see Section 5.4) are quite straightforward since the resource involved in each is static.

To enforce each constraint, a call to a predicate named \text{make_empty} is made. It builds a list of empty lists for the first argument. In constraint 1, this refers to the Sections list, while this refers to the Professors list in constraint 3. Such a call is made as in the following for the Sections list.

\[
\text{make_empty}(\text{Sections}, \text{Emptys1})
\]

After the list of empty lists is built, course codes are now grouped according to the sections that attend the course for constraint 1 and according to the professor assigned to the course for constraint 3. Note that since a course may be attended by several sections, the course must appear in the list of each of these sections. Both are done by two separate predicates named \text{group_courses1} for constraint 1 and \text{group_courses2} for constraint 3. A call to each predicate is made as follows.

\[
\text{group_courses1}(\text{NewCourses}, \text{Sections}, \text{Emptys1}, \text{CoursesPerSection}),
\]
\[
\text{group_courses2}(\text{NewCourses}, \text{ProfList}, \text{Emptys2}, \text{CoursesPerProf})
\]

The predicate \text{cumul_resource} is called afterward. It collects the start times, durations and resource needs of the course codes in the list specified in the argument. It is called in this manner for constraint 1.

\[
\text{cumul_resource}(\text{CoursesPerSection}, \text{Joblist})
\]

The \text{cumul_resource} predicate subsequently calls a predicate named \text{make_cumulative}, which then calls the cumulative constraint of the \text{ic_edge_finder3} library.

The cumulative constraint constrains courses attended by a section or courses assigned to a professor to be assigned different start times. Since each section only attends a course once and each professor only teaches a course once in the two schedules, each course takes 1 as its resource requirement. A section or a professor can only attend one course at a time. Thus, the resource limit is also 1 to denote that courses of a single resource (section or professor) must not overlap.

The ECLiPSe system provides the cumulative constraint in three of its libraries: \text{ic_cumulative}, \text{ic_edge_finder} and \text{ic_edge_finder3}. Among the three,

ic_edge_finder and ic_edge_finder3 implement stronger versions of the cumulative scheduling constraint. They implement a technique known as edge-finding to derive stronger bounds on the start times [4]. But between the two libraries, the ic_edge_finder3 provides the stronger propagation although it is also computationally more expensive [7].

Therefore, it is the ic_edge_finder3's version of the cumulative constraint that is used in the program. The call is made using the module qualifier ://2 to indicate that the cumulative constraint that is being referred to is that of the ic_edge_finder3.

\text{ic_edge_finder3}: \text{cumulative}(Ts, Ds, Rs, \text{N}).

where Ts refer to the start times of these courses, Ds refer to the respective duration of these courses, Rs to the resource requirement of each course (1 for professor and 1 for section) and N refers to the number of a particular resource that are available at any one time (1 for both section and professor).

However, a different approach is taken to impose constraint 2. The room, unlike the section and professor, is not static. It is in fact a variable, just like the start time, that is only labeled right after the constraints are set up. Thus, the above strategy used on the section and professor is not applicable to the constraint on rooms.

Nevertheless, the cumulative constraint is still used but on a reasonably different manner. Recall that when the RoomAssign variable is initialized, the courses and rooms are divided into two groups, lecture and laboratory. The rooms are further divided into groups based on equipments and the lack thereof. If the size of the group without any equipment is one, then courses that do not need equipments are initialized with the domain of all rooms. Courses that need particular equipments, on the other hand, are initialized with the domain of rooms that match the requirements. However, if the number of lecture courses exceeds 80% of the maximum number of courses that can be assigned to the lecture rooms, the search is halted as a solution may not be possible or may take a long time to be found. The 20% serves as a variance for the constraints that the courses are subjected to.

The call to the cumulative constraint also varies based on the above conditions. Specifically, if the size of the group of rooms without any equipment is one, the resource limit is the size of the lecture rooms less 1. This is the least number of courses that can be assigned the same timeslot. Declaratively, this call means that courses scheduled on a timeslot should not exceed the number of available rooms on a timeslot. The call is written as follows.

\text{make_cumulative}(\text{LecGroup}, \text{Joblist}, [], [], \text{NewNLec}) infers most

where NewNLec is NLec less 1.

On the other hand, if the size of the group of rooms without any equipment is more than one, then the group of courses is divided according to the domain of the RoomAssign variable. The resource limit of each group depends on the size of the domain of rooms that they correspond to. For instance, if the size of the group of rooms without any equipment is two, then the resource limit for the group of courses that does not need any equipment is also two. In the same manner, if the size of the group of rooms with an LCD projector is four, then the resource limit for the group of courses that need an LCD projector is also four. In general, the number of courses in a group that are assigned the same timeslot should not exceed the number of rooms for the group. This is performed through a call to cumul_rooms, which in turn calls make_cumulative. Grouping the courses, building the domains and calling the cumulative constraint via cumul_rooms are performed in the following calls for the laboratory rooms.

\text{get_rooms}(\text{Joblist, RAs}),
\text{build_dlist}(\text{RAs, [], DList}),
\text{sort}(\text{DList, SortedDList}),
\text{halve}(\text{SortedDList, LecDList, LabDList}),
\text{make_emptyys}(\text{LabDList, Emptys5}),
\text{group_jobs2}(\text{LabGroup, Joblist, LabDList, Emptys5, LabDGroup}),
\text{cumul_rooms}(\text{LabDGroup, LabDList, Joblist}) infers most

Calls to the cumulative constraint are annotated with \text{infers most} from the propia library. Propia is the ECLiPSe implementation of Generalized Propagation. Instead of evaluating the goal non-deterministically, Propia evaluates the constraint by extracting information from it deterministically. In this way, Propia reduces the number of choices that are to be explored and makes programs more efficient [4]. In this program, the use of the annotations considerably reduced the number of delayed goals before labeling takes place.

However, the above constraints on the room do not yet suffice to guarantee that constraint 2 is satisfied. The above constraints only ensure that the number of courses that are assigned on a timeslot does not exceed the number of rooms. An additional constraint is established to ensure that the number of courses that are assigned to the same room is no more than the number of timeslots, which is 18. This is done by calling the alldifferent constraint of the ic_global library. This is written in the following manner.

\text{ic_global: alldifferent}(\text{RAs, 18})

The said constraint is called from the constrain_rooms predicate, the call of which is written as follows.

\text{constrain_rooms}(\text{Joblist, LecGroup, LabGroup})

After the RoomAssign variables are labeled, a final constraint is imposed. It ensures that courses assigned to the same room do not overlap. All codes of courses that are assigned to the same room are collected. The make_cumulative predicate is called on each group of courses with a resource limit of 1. This is performed by the cumul_room predicate, the call of which is written as follows.

\text{cumul_room( Joblist )}

Apart from these constraints, another constraint is also defined to constrain a course that has precedences to be assigned a timeslot that immediately follows the precedences. The preceding course is also restricted from being assigned at the last timeslot on the M schedule. This entails the next course to be scheduled at the first timeslot on the T schedule, thereby violating the constraint in
6.5 Search and Labelling

Among the three variables, the DayAssign is the last to be labeled since it relies on the assignment of StartTime and thus, must be labeled after StartTime.

The variables with the smallest domain must be labeled first. Between StartTime and RoomAssign, it is RoomAssign that has the smaller domain. Thus, the RoomAssign variable is labeled first. In fact, this is already anticipated in the design of the constraints as the constraints on rooms would have been written differently if the StartTime is to be labeled first.

Since a complete search is too costly, a bounded incomplete search is used instead. In particular, the search used in the study is a depth bounded search that explores the first 2 level choices completely and then switches to bounded backtracking search that does not allow any backtracking steps for the next levels. But any incomplete search method is sufficient in practice. What matter more in reality are the choices for the variable selection and value choice methods.

In variable selection, variables with the smallest domains should be labeled first. This connotes that either first_fall or most_constrained methods can be used. Both methods choose the variable with the smallest domain size first. When several variables have the same domain size, the most_constrained method labels the variable with the most number of constraints first.

In value ordering, on the other hand, there is no intuitive choice that one can use for the present course timetabling problem. This indicates that no particular order for the value assignment can be perceived.

Each of the value choice methods are tried with the most_constrained variable selection method while using the depth bounded search described earlier. However, only the value choice method indomain_random was able to generate successful results. This was expected in light of the observation that there is no intuitive order for value selection that the search can follow to produce immediate results.

Since the choice for the values to assign first is entirely random, results may be irreproducible. Sometimes, the search might not produce a result at all. But this just indicates that the algorithm has chosen an arbitrary value that can not produce a solution on that particular run.

ECLiPSe has provided the seed/1 predicate to set the initial seed value that a random number generator can use. Seed values can run from 1 to 2^31 – 1. Each run takes up 60 seconds on average to produce a solution. Unfortunately, trying all possible seed values from 1 to 2^31 – 1 is clearly not viable. Nonetheless, the seed values from 1 up to an initially smaller value (100) were still tested.

Upon observation, certain solution costs actually recur. However, this does not denote that runs with the same solution cost produced identical timetables. But since the study is interested in getting a solution with a relatively smaller cost, the seed values were still explored to a certain degree.

In particular, seed values were tried one at a time for the program in a 24-hour period for the First Semester SY 2008-2009 problem instance.

6.6 Optimization

There are two approaches to optimization in CLP. The first refers to the use of an objective function to define the cost of a solution. The other approach distinguishes between constraints by specifying the degree to which they should be satisfied. Two models fall under the second approach, Constraint Hierarchies and Partial Constraint Satisfaction. Partial Constraint Satisfaction was used in the works [1][2] where the glass-box approach was followed.

Since an in-depth study of the internal solver is needed to model the soft constraints using either model, an objective function is used instead to produce the optimal solution.

The objective function is a compound metric that is composed of the soft constraints discussed in Section 5.4, all of which takes equal weights.

For the first soft constraint, the assignments that are not preferred for the professor and for the course are counted. The value is returned as part of a list named RawCounts. Additionally, the same value is used to get the percentage of violations in all the courses of professors with time preferences and courses with time preferences. These are done through the build_pplist and count_ppviol predicates for the professor and build_cplist and count_cpviol for the course. These are called as follows for the professor time preferences.

```prolog
build_pplist(PList, Professors, [], RPPrefList),
reverse(RPPrefList, PPrefList),
count_ppviol(CPProf, Joblist, PPrefList, 0, N, 0, PRaw)
```

For the second constraint where a "hole" or gap in-between classes should be minimized, the gap of every two consecutive courses is determined to find the largest gap. As a factor of the total cost, the closeness of this value to the maximum gap, 8, is obtained. All of these are completed through the following subgoals.

```prolog
gap_max_gap(CPSect, Joblist, 0, MaxGap),
GapRaw is MaxGap / 8
```

As for the third constraint, a professor should not be assigned to the timeslots 14, 15 and 16 as the professor would result to teaching for 4.5 hours with 5-minute intervals only. Start times of courses that will be taught by a single professor are checked whether they contain the start times 14, 15 & 16. The number of professors whose individual schedules contain these start times is counted. The percentage of the professors that are assigned these start times is obtained as a factor of the total cost. These are performed by the following subgoals.

```prolog
fine_nobreak(CPProf, Joblist, 0, TNoBreaks),
length(Professors, NProf),
NBRaw is TNoBreaks / NProf
```
The fourth constraint states that each section's daily schedule should span for no greater than 8 hours. The number of timeslots 8 hours after the first course attended by a section during the day, which is approximately 8 timeslots on the M schedule and 5 timeslots on the T schedule, is counted. The maximum number of timeslots that exceeded this timeframe is obtained. The closeness of this value to the worst case, which is 3, is obtained as a factor of the objective function. These are all completed through the following subgoals.

\[
\text{count\_extension}(\text{CPSect}, \text{Joblist}, 0, \text{MaxExt}), \text{ExtRaw is MaxExt} / 3.
\]

All the raw counts and costs that are mentioned above are incorporated with equal weights in the objective function. The raw counts are collected in the \text{RawCounts} list.

The user may also choose which among the five preference constraints are included for the objective function. This choice is represented as a list of five 1s and 0s, each representing the minimization of gaps, daily section schedule, professor time preferences, course time preferences and successive teaching respectively. This is passed as arguments to the main predicate. The \text{RawCounts} list and the \text{TotalCost} are returned to the main predicate. The computation of the \text{TotalCost} and the construction of the \text{RawCounts} list are done as follows.

\[
\begin{align*}
\text{GapCost} & = \text{GapRaw} \times \left(\left(\text{GapSel}/\text{Sum}\right) \times 100\right), \\
\text{ExtCost} & = \text{ExtRaw} \times \left(\left(\text{ExtSel}/\text{Sum}\right) \times 100\right), \\
\text{PCost} & = \text{PRaw} \times \left(\left(\text{PPSel}/\text{Sum}\right) \times 100\right), \\
\text{CCost} & = \text{CRaw} \times \left(\left(\text{CPSel}/\text{Sum}\right) \times 100\right), \\
\text{NBCost} & = \text{NBRaw} \times \left(\left(\text{NBSel}/\text{Sum}\right) \times 100\right), \\
\text{PRaw2} & = \text{PRaw} \times N1, \text{CRaw2} = \text{CRaw} \times N2, \\
\text{RawCounts} & = [\text{MaxGap}, \text{MaxExt}, \text{PRaw2}, \text{CRaw2}, \text{TNoBreaks}], \\
\text{TotalCost} & = \text{GapCost} + \text{ExtCost} + \text{PCost} + \text{CCost} + \text{NBCost}.
\end{align*}
\]

The built-in \text{minimize} predicate was initially used to search for the solution with the least cost for the First Semester SY 2008-2009 instance. The First Semester SY 2008-2009 instance is composed of 131 courses, 38 professors, 12 lecture and laboratory rooms. However, after a 24-hour period, the minimum solution that the search described in Section 6.5 was able to find has a cost of 42+. The said search did not use any seed value.

Nevertheless, even if the search used a seed value, the number of possible solutions that it has to traverse reaches at least twelve thousand. So it is not feasible to use the minimize method to search for the solution with the smallest possible cost.

As an alternative, seed values were tried one by one for the model in a 24-hour period resulting to seed values from 1 to 1635 only.

The following table shows the better solutions among those generated:

<table>
<thead>
<tr>
<th>Seed Value</th>
<th>Counts</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>427</td>
<td>[3,2,5.0,8.0,0]</td>
<td>29.4697</td>
</tr>
<tr>
<td>894</td>
<td>[3,2,6.0,7.0,0]</td>
<td>30.01515</td>
</tr>
</tbody>
</table>

The seed value that will be used for the said instance can either be 427, which produced a solution with a cost of 29.4697, or 894, which produced a solution with a cost of 30.01515. Between the two solutions, seed value 427 produced a solution with lesser professor time preference violations while seed value 894 produced a solution with lesser course time preference violations. In most cases, the time preferences of the professors precede those of the courses. Therefore, seed value 427 is used for the problem instance.

7 CSDPlanner

Just like the Parc Technologies’ AirPlanner and RiskWise applications, the current application called CSDPlanner also uses the following full application structure discussed by H. Simonis [23].

![Figure 2. Full Application Structure](image)

In this design, the main application poses queries for the ECLiPSe solver via a Java-ECLiPSe interface while passing data and arguments into ECLiPSe. The problem solver runs the query and returns results as variable bindings. The internals of the problem solver as well as how queries are resolved are hidden from the user. [23]

For the CSDPlanner, additional components are incorporated in the structure to display the timetable in a presentable format in MSExcel. As a result, the overall design that the application uses is shown below.

![Figure 3. CSDPlanner Application Structure](image)
After obtaining the results as variable bindings, the Java application formats the variables and displays them in an Excel workbook.

The CSDPlanner user interface is shown in the figure below. It contains three panels that refer to the data that the main query goal needs as well as some details that are needed for the Excel workbook. In particular, the first and the second pane must be completed before the query is performed.

![CSDPlanner User Interface](image)

The names of the three input files that the solver needs must be keyed in on the first pane. The Browse button may also be used to choose the input file from the Open dialog box. Soft constraints that will be included in the computation for the objective function are marked on the second panel. The third panel, on the other hand, consists of the Semester and School Year that the particular problem instance refers to. These are typed in for the Excel workbook. When all the panels have been completed, the Run Query button can now be clicked.

After the query has been sent to the ECLiPSe engine, the Java side awaits the response from the call. When the ECLiPSe engine sends back the output, the raw counts of the soft constraints and the Total Cost of the solution are displayed in a Message dialog box as shown below.

![CSDPlanner Message Dialog Output](image)

Aside from the message dialog box, the contents of the array of courses are also shown in the Standard Output stream as shown below.

![CSDPlanner Standard Output](image)

The application then creates a workbook that contains the worksheets for each schedule (M and T) and room type (Lec and Lab). The following figure is the M-Lec worksheet generated for the First Semester SY 2008-2009 problem instance.

![Sample Schedule Worksheet](image)

8 CONCLUSIONS

After the model has been created and the front-end application has been developed, the following can be inferred from the study.

It is feasible for the AdDU Computer Studies Division scheduling process to be modeled using the constraint logic programming approach.

A timetable can be produced that satisfies all the hard constraints and has a significantly low cost using the objective function.

One major aspect that makes the produced timetable less desirable is the fact that not all First Year subjects are assigned their preferred start times. This is reasonable in the study as the said constraint was not defined as a hard constraint. However, in reality, First Year subjects are laid in the timetable first, hence, all of them are assigned a desired start time. Thus, it appears that these subjects should be assigned their preferred start times and this should have been defined as a hard constraint. This was actually attempted in the model but it was not able to produce any solution at all whichever search strategy, variable selection method and value choice method are used. So this was not
considered in the present model as this also necessitates an entirely different search and labelling strategy.

The application returns only one solution since the other solutions are just symmetric equivalences of the first solution. Symmetric equivalences are classes of solutions that have identical cost and that differ only by exchanging a resource value [17]. This was not intentionally made and was only a result of the model design and search heuristics used. In particular, to produce a non-symmetrical solution, another seed value must be tried.

In light of the said conclusions, the proponent recommends the exploration of the following notions for future researches.

- Develop an interface for an existing Hierarchical CLP (HCLP) interpreter to a popular CLP system, such as ECLiPSe or to develop an HCLP interpreter in a popular CLP system, such as ECLiPSe.
- A repair strategy can be used wherein the search will work with an initial solution, which can perhaps be the timetable producible in the same semester of the previous school year.

9 REFERENCES


of Automated Timetabling IV, LNCS 2740 (pp. 310 – 328). Germany: Springer-Verlag.


