

Final Year Project Report  
School of Computer Science  
University of Manchester

# **A Scheduling Tool for PASS (using Constraint Satisfaction)**

**Justin Emery**

*Degree Programme:* BSc Computer Science  
*Supervisor:* Andrei Voronkov  
*Date:* May 2009

## **Abstract**

The PASS scheme involves groups of first year students being led by two or three students that are now in the second, third, or fourth year of their degree. I was asked to create a tool that would automatically schedule the PASS sessions in the Faculty of Life Sciences at the University of Manchester. This scheduling essentially involves the assignment of students and leaders into groups, and those groups into time slots, and is subject to constraints including the availability of leaders. In past years this schedule has been created manually using spreadsheets and has taken days of work. The result is a tool that, after data has been input, can create a good schedule in minutes.

The task was addressed by treating it as a Constraint Satisfaction Problem (CSP). This report shows the formulation of the problem as a CSP, and explains how it was implemented using Java and the JaCoP library. It shows the difficulty of finding a near-optimal solution for a CSP of considerable size (hundreds of variables).

Also described, how this constraint satisfaction program was integrated this into a working system in the form of a web application.

**Project Title** A Scheduling Tool for PASS (using Constraint Satisfaction)

**Author** Justin Emery

**Date** 6th May 2009

**Supervisor** Andrei Voronkov

## **Acknowledgements**

I would like to thank my supervisor, Andrei Voronkov, for his useful suggestions and feedback.

I would like to thank Emily Wiles, FLS Sabbatical Intern, for coming up with the real world PASS scheduling problem, without which this project would have been much less satisfying.

Finally, I would like to thank my family and friends, for their support and suggestions.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Background . . . . .	6
1.2	Project Proposal . . . . .	7
1.3	Objectives . . . . .	7
1.4	Tackling the Problem . . . . .	7
1.5	Report Overview . . . . .	8
<b>2</b>	<b>Constraint Satisfaction</b>	<b>9</b>
2.1	What is a Constraint Satisfaction Problem (CSP)? . . . . .	9
2.2	Solving CSPs . . . . .	10
2.3	Constraint Programming . . . . .	11
2.4	Optimisation . . . . .	12
2.5	Complexity . . . . .	13
<b>3</b>	<b>Design Overview</b>	<b>15</b>
3.1	System Architecture . . . . .	15
3.2	Scheduler Design . . . . .	16
3.3	Web Application Design . . . . .	18
3.3.1	Requirements for the PASS Leader Interface . . . . .	19
3.3.2	Requirements for the PASS Organiser Interface . . . . .	19
3.3.3	Non-functional requirements . . . . .	21
3.3.4	Database Design . . . . .	21
<b>4</b>	<b>PASS Scheduler Implementation</b>	<b>22</b>
4.1	A First CSP Model for PASS Scheduling . . . . .	22
4.2	Constraint Programming with JaCoP . . . . .	23
4.3	Mapping CSP Result to Java Objects . . . . .	25
4.4	Optimisation . . . . .	25
4.5	Redundant Model . . . . .	26
4.6	Symmetry . . . . .	27
4.7	The Final Formulation . . . . .	29

<b>5</b>	<b>Web Application Implementation</b>	<b>31</b>
5.1	Overview of Web Application Architecture . . . . .	31
5.2	Database Interaction . . . . .	32
5.3	Integrating the Scheduler into the Application . . . . .	34
5.4	User Interface . . . . .	34
5.5	Use of JavaScript . . . . .	35
5.6	Login . . . . .	36
<b>6</b>	<b>Testing and Evaluation</b>	<b>37</b>
6.1	Testing the Scheduler . . . . .	37
6.2	Measuring Schedule Quality . . . . .	38
6.3	Comparison to a Manual Schedule . . . . .	39
6.4	Testing of Web Application . . . . .	40
<b>7</b>	<b>Conclusions</b>	<b>41</b>
7.1	Achievement of Objectives . . . . .	41
7.2	Improving the Scheduler CSP . . . . .	41
7.3	Improving the Web Application . . . . .	42
7.4	Changes to Original Plan and Expectations . . . . .	42
7.5	Comments on the use of Constraint Satisfaction . . . . .	43
	<b>References</b>	<b>44</b>
<b>A</b>	<b>Application Demo</b>	<b>46</b>
A.1	Login . . . . .	46
A.2	Data Setup . . . . .	47
A.3	Creating and Viewing Schedules . . . . .	49

# List of Figures

3.1	The high level architecture of the system as a whole. . . . .	15
3.2	Domain class diagram for schedule model. . . . .	17
4.1	Lexographical ordering of rows in a matrix of leader to group assignments. . . . .	29
A.1	This login screen for the PASS scheduler. . . . .	46
A.2	The general setup page. . . . .	47
A.3	The time slot setup page. . . . .	47
A.4	The degree course setup page. . . . .	48
A.5	The students setup page. . . . .	48
A.6	The leaders setup page. . . . .	49
A.7	The schedule creation page. . . . .	50
A.8	A schedule processing. . . . .	50
A.9	Viewing a finished schedule. . . . .	51

# List of Tables

3.1	Input parameters for schedule() method of PassScheduler. . . . .	18
5.1	Java packages for web application . . . . .	32
6.1	Comparison of Generated and Manual schedule . . . . .	39

# Chapter 1

## Introduction

### 1.1 Background

Peer Assisted Study Sessions (PASS) involves groups of first year students being led by students that are now in the second, third, or fourth year of their degree.

In the Faculty of Life Sciences (FLS) at the University of Manchester, there are around 500 first year students. For each group of roughly 15 first year students there are either 2 or 3 leaders. Ideally, both first year students and PASS leaders should be on the same or a similar degree course. Otherwise, at least one PASS leader should share the same (or a very similar) degree program as the majority of the group. Occasionally it is not possible to match first year students and leaders with similar degree courses.

Each PASS group must have one session each week. It is not possible for all the PASS sessions to take place at the same time, so there are a number time slots (currently 5) in the first year timetable in which PASS sessions may take place. First year students would usually be able to attend for any of these slots. There is a limit to the number of PASS sessions that can take place in any time slot, based on the number of rooms available to hold PASS sessions. PASS leaders may only be able to attend certain time slots, based on their university timetables. Taking all of this into account, PASS groups need to be scheduled into time slots (each group attends the same timeslot every week).

In previous years a schedule has been designed manually using spreadsheets and has taken days to complete. The process has basically involved assigning students and leaders to groups (taking into account the availability of leaders), then moving them around until the schedule is reasonably good: meaning that most student and leader degree courses match up.

Throughout this report, first year students are referred to as *students*, and group leaders as *leaders* (despite the fact that they are students too). The other main entities referred to are *time slots*, *groups*, and *degree courses*.



## 1.2 Project Proposal

The proposal was to create a tool that would automatically schedule the PASS sessions in the Faculty of Life Sciences (FLS). The expectation was that due to the power of modern computers, it should be possible to do the scheduling automatically; therefore greatly reducing the time spent organising PASS sessions at the beginning of a year. Some time would still be needed for data setup, which includes inputting the details of students and leaders, specifying their degree courses and the time slots. However this would be a matter of hours rather than days.

It was requested that the tool be created as a website, so that PASS leaders can specify their availability for the time slots. The tool should automatically organise the first year students and leaders into groups of similar degree programs, and then assign these groups to time slots, based on the availability of PASS leaders. Finally it should assign groups to locations/rooms (this is just 1 group per location, so would be a trivial “1-to-1” matching). A PASS organiser should be able to override this automatic assignment by manually moving around leaders in anomalous cases.

## 1.3 Objectives

The primary objective was to create a tool that can automatically generate a schedule for the PASS scheme in FLS, as described above. This can be split into 2 milestones:

**Milestone 1** The tool can sort students and leaders into groups, and place groups into timeslots. In particular, leaders must be placed only into groups where they are available for the time slot of that group.

**Milestone 2** The generated schedule should be of high quality, where quality refers to how well the degree courses of students and leaders match. A created schedule should be of similar or better quality than a schedule created manually. There are numerous ways in which this could be measured, but the primarily used measure will be a count of the number of students who have at least one leader on the same or a similar degree course.

A secondary objective is summarised with the following milestone:

**Milestone 3** A web interface should be created to facilitate the scheduling tool to be used easily and intuitively as a working system.

The final web interface will have more detailed requirements, which are described in section 3.3.

## 1.4 Tackling the Problem

There would be many different approaches to tackling a scheduling or timetabling problem. This includes writing a custom matching algorithm or using genetic al-

gorithms. To properly explore which method would be best could be a final year project in itself, but my supervisor was able to point me towards the field of Constraint Satisfaction.

After doing some brief research, it was clear that Constraint Satisfaction would be a sensible approach to take, and that it would also be an academically interesting approach as constraint satisfaction is a field that is gaining popularity in recent years.

## **1.5 Report Overview**

Chapter 2 will describe my research into Constraint Satisfaction and how this is used in practice using Constraint Programming. Chapter 3 will describe the general design of the system. Chapter 4 describes the formulation of the problem as a Constraint Satisfaction Problem (CSP), and the implementation. Chapter 5 describes the implementation of the web interface, and how the CSP code is integrated into this. Chapter 6 shows evaluates the success of the system and shows how it was tested, while Chapter 7 concludes the report.

## Chapter 2

# Constraint Satisfaction

This chapter gives an overview of the relevant areas in the field of constraint satisfaction. For introductions on constraint satisfaction problems see [1, Chap. 1–3] or [3], on which much of this chapter is based. More advanced issues in constraint satisfaction that were encountered during the development of the project are left until Chapter 4.

### 2.1 What is a Constraint Satisfaction Problem (CSP)?

*“[Constraints] identify the impossible, narrow down the realm of possibilities, and thus permit us to focus more effectively on the possible”*

Rina Dechter, [1]

A constraint satisfaction problem can be seen non-mathematically. We encounter them frequently in everyday life. For example, how do we eat healthily but still enjoy food? For this problem a constraint could be that we must eat no more than our recommended daily amount of fat, or that all food consumed must taste better than Brussels sprouts. Such simple problems are (unfortunately not always!) easy for humans to solve. However, for a more complex problem we need to have a less vague definition in order to be able to solve it, so we define CSPs mathematically.

A CSP can be used to solve a wide range of problems. Typical examples include coloring maps so that adjacent countries do not use the same colour, solving cryptarithmic puzzles, and solving Sudoku puzzles.

Definitions vary slightly, but for the purposes of this report, a CSP consists of:

- A set of variables;  $x_1, \dots, x_n$ .
- Domains  $D_1, \dots, D_n$  for those variables, where  $D_i$  gives a set of possible values for  $x_i$ . Throughout this project this has been restricted to finite domain CSPs, where the possible values of variables are always positive integers.
- A set of constraints  $C_1, \dots, C_m$  defined on a subset of the variables. These relations are sometimes expressed as a set of tuples defining all the allowed

combinations on a subset of variables. However, throughout this project numeric constraints are used, where the constraints are expressed by arithmetic expressions. These constraints could act between pairs of variables, or a variable and a value, eg.  $x_1 \neq 5$  or  $x_1 > x_2$ . They could also be *global constraints* acting on a number of values, such as the *allDifferent* constraint, which states that every variable must have a different value.

To solve a CSP we need to assign a value to each variable, from its domain, so that the constraints are satisfied.

It is easier to understand CSPs after looking at examples. Each of the books on constraints listed in the Bibliography for this report contains a selection of examples, such as n-queens, map colouring, and the knapsack problem.

**Example 1** *Here we define a basic (and meaningless) example of a CSP to be referred to throughout this chapter. It has three variables:  $\{a, b, c\}$ . Those variables have corresponding domains:  $D_a = D_b = \{1, 2, 3\}$ ,  $D_c = \{1, 2\}$ . There are two constraints:  $\text{allDifferent}(\{a, b, c\})$ <sup>1</sup> and  $a > c$ . There are three solutions to this CSP:  $\{a = 2, b = 3, c = 1\}$ ,  $\{a = 3, b = 2, c = 1\}$ ,  $\{a = 3, b = 1, c = 2\}$ .*

## 2.2 Solving CSPs

CSPs can be solved using general algorithms. These are split into two categories, *inference* algorithms and *search* algorithms. Inference algorithms tend to work by *propagating constraints*, that is, deducing new constraints from existing constraints. This kind of algorithm may solve a CSP entirely in simple cases; otherwise it should reduce the search space.

**Example 2** *Looking at Example 1, we can see constraint propagation looking at the  $a > c$  constraint. As  $c$  must have a value of at least 1, we can infer from this constraint that  $a$  cannot be equal to 1. So we could add in an extra constraint  $a \neq 1$ . As we add constraints like this, we are effectively removing values from the domains of values: this constraint removes 1 from the domain of  $a$ , so that  $D_a = \{2, 3\}$ .*

Search algorithms explore the possible solutions until a solution is found, or until the search space is exhausted, in which case there is no solution. Backtracking is one such search algorithm, which explores a search tree in a depth first manner. Search algorithms tend to be depth first, as we only require one solution.

**Example 3** *The constraint propagation shown in Example 2 would not reduce the problem to a single solution, so some searching would be required. One possible search could take the following steps:*

---

<sup>1</sup>Note that  $\text{allDifferent}(\{a, b, c\})$  is equivalent to  $a \neq b$ ,  $b \neq c$ ,  $c \neq a$ . Global constraints have two advantages: they provide an easier way to express numerous simpler constraints, and specific algorithms can be written so that they may be processed more efficiently.

1. Assign  $a$  an arbitrary value from its remaining domain  $(\{2,3\})$ . We assign  $a = 2$ .
2. The domains of  $b$  and  $c$  are now reduced due to this assignment. Their domains can no longer include 2, so now  $D_b = \{1,3\}$ , and  $D_c = \{1\}$ . Now  $c$  only has a single possible value, 1, so we could set  $c = 1$ . However, for the purpose of illustration let us continue with the variables in order, and set  $b$  with an arbitrary value, so  $b = 1$ .
3. This reduces the domain of  $c$  to  $\{\}$ , so we must backtrack. We return to  $b$ , this time assigning it with the other value in its domain, so  $b = 3$ .
4. The domain of  $c$  remains  $\{1\}$  so we assign must set  $c = 1$ . Each variable has now been assigned a value so we now have reached a solution, and can end the search.

*This search was very simple. There are different methods to decide how to proceed in a search. We could pick variables in a fixed order, as above, or as an alternative we could pick them based on the size of their domain. If we had always picked variables with a domain of a single value first, we could have avoided the backtracking above by picking  $c$  instead of  $b$  at step 2. Also we can choose how to assign values to variable using different methods. We could pick the lowest value in the domain first, or the highest values first, or we could pick values randomly.*

When searching a tree in this manner, the tree would represent the allocation of values to variables. We would expect the depth of a tree to correspond the number of variables. At each node we are choosing which value to assign to a certain variable. We would expect the number of children to be equal to the size of the domain of the variable at that node.

While knowing the details of how CSPs can be solved should be beneficial to anyone working with them, it is actually not necessary! This is because the general algorithms that are used to solve CSPs can apply to a wide range of problems. It is, however, necessary to be able to formulate a problem as a CSP, which is not always as easy as might be hoped.

## 2.3 Constraint Programming

*“Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.”*

E.C.Freuder, Constraints, 1997

The above quote expresses the concept behind solving CSPs with constraint programming. Rather than getting involved in the details of how to solve a problem, it should simply be enough to describe the problem precisely and accurately, and

then have a computer solve the problem using general algorithms as was described in section 2.2.

There are many ways to use constraint programming in practice. The Prolog language has built in support for constraint programming [9]. There are also libraries that allow constraint programming to be used available for many programming languages. Some of these are commercial products while others are available for free, with many being open source libraries. During work on this project, two free libraries for the Java language have been used. These were Choco (see [4]), and JaCoP (see [5]). There are many differences in the details of how these libraries are used, but essentially they both allow a CSP to be specified and can attempt to solve it automatically.

## 2.4 Optimisation

So far in this chapter CSPs have been described as being solved when just one solution has been found. There will often be more than one solution to a CSP, and it may be that the number of solutions is large. Sometimes it is necessary to find more than just any single solution; we need to find the best solution, as specified by the value of an *optimisation function*. The best solution can also be called an *optimal* solution, and is a solution for which the optimisation function gives a value that is greater than or equal to that of any other solution (there may be more than one optimal solution, having the same value for the optimisation function). Often it would not be necessary to find the optimal solution, just a good or near-optimal solution.

The term *possible solutions* will be used to refer to the number of solutions a CSP would have if we do not consider the constraints. So the number of possible solutions for a CSP is  $\sum_{i=1}^n |D_i|$ , the product of the variable domain sizes. As constraints are added, the number of possible solutions is reduced until we reach the number of solutions for the CSP.

We can see with Example 1 there are 3 solutions out of 18 possible solutions. This means that  $\frac{1}{6}$  of possible solutions are also solutions. If this was the case with a larger problem, a problem with a large number of variables and large domains, then the number of solutions will also be large. So for optimisation, where we want to search through all the solutions, it may be infeasible to do so. Instead we can look for a near-optimal solution, by searching for a set period of time, and using the most optimal solution that is found within that time-period.

It is fairly easy to see how optimisation can be an extension of standard CSP solving, where the value of the optimisation function will be stored in one of the variables called  $x_{\text{opt}}$ :

1. Search as normal until the first solution is found; if no solution found then stop.
2. Add in a new constraint stating that  $x_{\text{opt}}$  must be greater than the value of

$x_{\text{opt}}$  in the current solution. So if the current value of  $x_{\text{opt}}$  is 85, add a new constraint  $x_{\text{opt}} > 85$ .

3. Return to step 1, if time limit is reached at any point then stop.

However, many constraint programming libraries include this optimisation functionality built in. They allow an optimisation variable to be specified, and then find a solution where the value of this variable is either maximized or minimized. The optimisation function should be specified as a constraint in the CSP, so that the optimisation variable is assigned a value. Choco allows optimisation variable to be either maximized or minimized, whereas JaCoP only allows it to be minimized, referring to it as a cost variable.

## 2.5 Complexity

Constraint satisfaction problems tend to be NP-Complete. Many typical NP-Complete problems such as the knapsack problem, traveling salesman, and bin packing can be solved as CSPs [4]. The main concern in the complexity of solving CSPs is time complexity.

The precise complexity of a CSP is difficult to calculate, as the algorithms used to solve them are general, and the focus is on defining the problem rather than a specific implementation. However, it is possible to make some general observations concerning complexity.

There is a trade off between ease of finding a single solution, and ease of finding an optimal solution. A CSP can be described as loosely constrained, where the constraints are minimal so as to describe all solutions, or highly constrained, in which case some solutions have been eliminated by further constraints. The reason for highly constraining a CSP would be for the purposes of optimisation, to eliminate solutions that can be considered identical, therefore reducing the number of solutions to search through. This is usually done by eliminating ‘symmetric’ solutions, which is described in further detail in 4.6.

The most extreme case of a loosely constrained CSP is one without any constraints. Every combination of assignments of values from the variable domain to variables would be a valid solution. For such a CSP with  $n$  variables, each with a domain of size  $m$ , there would be  $m^n$  solutions. So finding the optimal solution, thereby searching through all possible solutions, would be of order  $O(m^n)$ . Finding a single solution to this CSP would be much easier as we could select any value from the domain of each variable and have a valid solution. So this would be of linear time complexity  $O(n)$ .

At the other extreme, we have a highly constrained CSP where there is only one solution. In this case the complexity would depend on how much the domains of the variable can be reduced by inference algorithms. If each variables domain is reduced to one value, then we have  $O(n)$  for both finding a single solution and finding the optimal solution. On the other hand, the inference algorithms may not reduce the

size of the variables domains at all, so for both finding a single solution and finding the optimal solution we have  $O(m^n)$ .

The extremes described above are unlikely situations, but they illustrate the factors that affect the complexity of solving a CSP. A typical CSP would usually be between these extremes. If, for example, as in section 2.4,  $\frac{1}{6}$  of possible solutions are solutions, we could search randomly through 10 possible solutions and have a good chance of finding a solution. Even if we have 0.1% of possible solutions being solutions, it should still be feasible to solve this CSP quickly, given the speed of modern computers.



## Chapter 3

# Design Overview

This chapter describes the high-level design aspects of the project, and describes some of the key implementation decisions, while leaving details of implementation to the following chapters.

### 3.1 System Architecture

The system is quite clearly split into two major parts. The scheduler or constraint satisfaction part would solve the scheduling problem described in Chapter 1 using constraint programming, while the web application would provide an interface to allow this to be easily used, making it a working system.

The original plan was to divide the work on the project roughly into two halves, reflecting the two sides to this system. However, it soon became apparent that the constraint satisfaction side of the project was far more challenging, so this report reflects this.

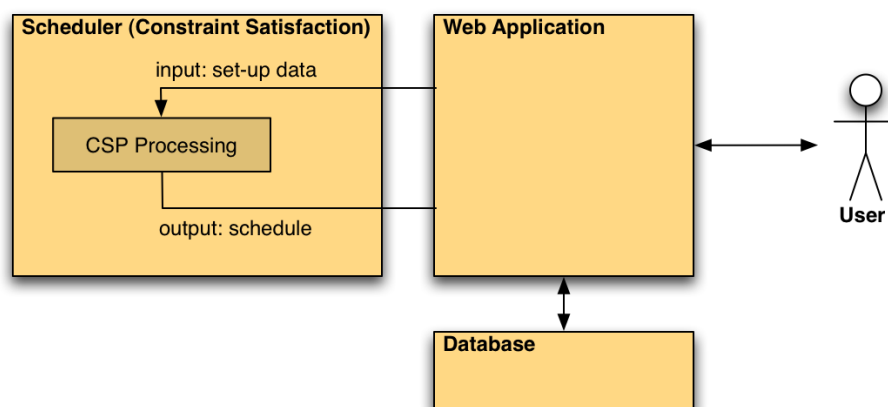


Figure 3.1: The high level architecture of the system as a whole.

The development process used during my project was an incremental one, using some aspects of the Universal Process (UP) . The most important aspect of the UP considered was that of addressing critical risks early on [17]. The formulation and implementation of the CSP, as described in Chapter 4, was the most challenging and highest risk part of the project. The web application side of the project was more straightforward (though still a considerable body of work) and therefore developed much later. However, the development of both sides did overlap, as it was useful to have a reliable impression of how the system would work as a whole as the details of the CSP implementation were still being finished off.

After some exploration of constraint satisfaction as detailed in Chapter 2, it was decided to use the Java programming language for my project. This was due to my familiarity with the language, the availability of constraint programming libraries in Java, and the fact that Java provides an effective way of creating web applications through Java Servlets. A benefit is that using one language for both the constraint satisfaction and the web application sides of the system eliminates any added difficulty in getting two languages to interact.

The system was developed using the Model-View-Controller (MVC) design pattern, which enables clear separation between business logic (Model) and presentation (View). The constraint satisfaction part of the project can be thought of as the Model. The web application side of the project can be viewed as the View and Controller, with the Controller being Java Servlets, and the View being Java Server Pages (JSPs). Keeping the Model completely separate from the rest of the system enables it to be reused. [3]

## 3.2 Scheduler Design

The scheduler part of the project involves formulating the problem as a CSP, and specifying this with a constraint programming library, and then using that library to attempt to solve the CSP. Formulating the CSP was an incremental process that involved starting with a basic formulation and then implementing it in Java, and then when subsequent changes were made to the formulation, the implementation was updated to match. Due to this, the description of formulating the CSP is left until Chapter 4.

Development originally started using Choco, as it was a free open-source constraint programming library that appeared to have all the functionality needed. Early versions of the system were working with Choco; however, the documentation for this library was somewhat lacking, so there was a switch during development to the using the JaCoP library, with which the project was completed. This was only a minor set-back as the two libraries are similar in concept.

The CSP needs to describe the problem described in Chapter 1. Figure 3.2 shows a domain class diagram of the PASS Scheduling Situation. This shows what is required by a finished schedule. In the implementation, corresponding Java classes represent this.

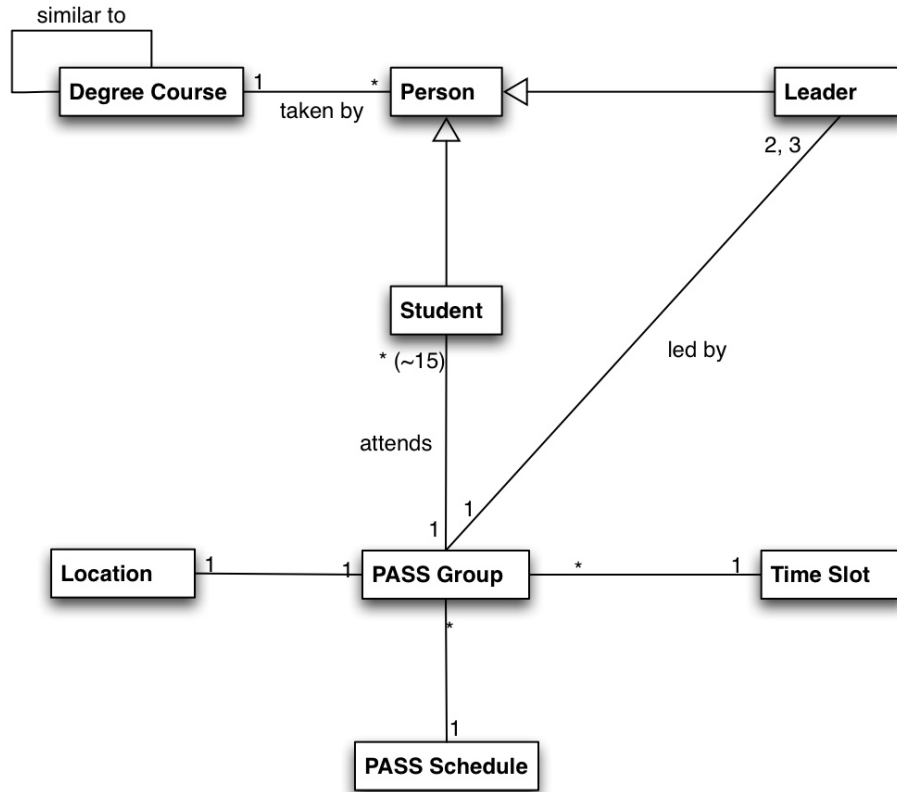


Figure 3.2: Domain class diagram for schedule model.

An abstract class, `PassScheduler`, defines the class that does the scheduling. This class can be subclassed; providing an implementation that can create a finished schedule that contains many PASS groups, as shown in 3.2. Using an abstract class in this way is a good example of flexible software design, as it separates the definition of a class (what it does) from the implementation (how it does it). In the final version of the application just one implementation is used, `JacopPassScheduler`, however, having this flexibility did aid development when the switch in implementation was made from the Choco constraint library to the JaCoP library. Because an abstract class was used here, it was easy to create a new subclass with a new implementation. It would even be possible to provide an implementation that deals with the problem using some other method rather than constraint satisfaction.

The scheduling is performed by the `schedule()` method of a `PassScheduler` implementation. The result of this method is a finished schedule as described above. There are many parameters for this method, they are shown in Table 3.1. Note that originally it was planned that the scheduler would assign a location to each group, but this feature was dropped due to a lack of time. It would be trivial to add this as it is a simple one-to-one mapping.

Table 3.1: Input parameters for `schedule()` method of `PassScheduler`.

Parameter	Description
<code>Student []</code>	Array of students to place into groups.
<code>Leader []</code>	Array of leaders to place into groups. Leaders should already have their availability specified; they should have references to the timeslots objects they are available for.
<code>TimeSlot []</code>	Array of timeslots in which groups may be placed.
<code>DegreeCourse []</code>	Array of degree courses that students and leaders may be taking. Each student and leader must contain a reference to one of the <code>DegreeCourse</code> objects in this array.
<code>PassScheduleParameters</code>	This is an object that contains a number of parameters to describe how a finished schedule should be organised. These parameters are the minimum and maximum number of leaders permitted in each group, the minimum and maximum number of students permitted in each group, and the minimum and maximum number of groups that may be placed in each timeslot.
<code>int timeInMinutes</code>	This specifies the maximum number of minutes that the scheduler should search for.

Creating an implementation using constraint satisfaction for this `schedule()` method was the most challenging part of this project, so warrants an entire chapter to itself, Chapter 4.

### 3.3 Web Application Design

As the motivation for this project was to create a working system that could be used in future years, the creation of a functional and usable web interface was of particular importance. In fact, two distinct interfaces would be needed: one for the organisers of PASS to input data and create a schedule, and another for leaders to specify their availability for time slots. This second interface, having many users, is the main reason for creating a web application rather than a desktop application. It would have been possible to create a desktop application for the PASS organisers interface, but in recent years web technology has advanced to a stage where a web application would have no significant disadvantage as compared to desktop application, at least in this case.

The planned functionality of the two interfaces will be described below. Some features were dropped due to a lack of time; these are indicated. The impact of the

absence of these features is discussed in Chapter 7. Otherwise the features listed should be considered as implemented successfully, although a discussion of the extent of success is left until Chapter 6.

### **3.3.1 Requirements for the PASS Leader Interface**

- Leaders can log in to the website with their emails and a password that they have been previously informed of.
- Leaders must select from a list of timeslots which they are available for, and submit this. They may enter special requirements into a separate field.
- After a schedule has been created and chosen by the organiser, leaders can view the organisation of the group. That is, the leaders and students in the group are listed along with their degree courses. The timeslot of the group is also shown.
- Leaders may enter and send an email to all students in their group using the web interface (dropped).

### **3.3.2 Requirements for the PASS Organiser Interface**

- Facilitate the set up of data required to generate a schedule. A key principle here is that the CRUD operations should be allowed for all data [16]. These are Create (or add), Read (or view), Update (or edit), and Delete. So CRUD operations are available on time slots, students, leaders, and degree courses. These operations all act on a single item of data, eg. create a student, or edit a leader. In addition, the following more advanced data setup features are supported:
  - Pairs of degree courses can be specified as being similar, and these similarities can be removed.
  - Rather than inputting one student at a time, a list of students can be entered by uploading a CSV (Comma Separated Value) file. This file contains one student per line, with a name, email and degree course for each. Uploading a CSV file in this way will greatly reduce effort in data input, as much of the information is already available in spreadsheet form (which can be easily saved as a CSV file). A factor to be cautious of here is that the degree courses must be spelt consistently, otherwise multiple degree courses will be created that in reality represent the same course. The use of codes to specify degree courses was considered, however this was decided against; it would have required extra preparation of data because codes are not currently used.
  - Similarly, A CSV file of leader data can be uploaded. For testing, this file may also contain leaders availability for timeslots. In real use, each

leader would specify their availability, by using the PASS Leader Interface described above. As each leader is created, a password for that leader is randomly generated, which can be given to the leader to allow him to log in.

- The setup of locations (using CRUD operations) was originally planned, but as the assignment of Locations to groups was dropped, this was too.
  - A number of general settings for the scheduler can be edited. These correspond to the `PassScheduleParameters` object described in section 3.2.
  - Once data has been set up, a schedule can be generated. The user can start the generation of a new schedule, giving it a name, and setting a time limit for the processing. The scheduler runs on the server, and for this period of time the page for that schedule will show a processing notice, and inform the user of the time left until the schedule will be complete. This generation of a schedule can be cancelled before the processing finishes.
- When the schedule is completed, it can be viewed. A list of groups is shown, each group with a list of the leaders and students that are in that group, and showing the time slot that group has been placed in.
  - A schedule can be edited: leaders and students can be moved from one group to another.
  - A schedule can be deleted.
  - A copy of a schedule can be created. A copy might be useful if a PASS organiser wants to try moving leaders or students around, while retaining the original schedule incase the result is no better.
  - Once a schedule has been created and decided upon, it can be selected as the final schedule. The PASS organisers may have experimented by generated multiple schedules, using different settings, and may have edited the schedules using the manual editing features. They would select the best of these schedules for final use. This then updates the PASS Leader Interface to show the group information for each leader.
  - Emails can be sent to all leaders. This feature can be used to inform leaders of their passwords and request that they log in to specify their availability. It could also be used when a schedule has been selected for use to inform the leader of their PASS group. (dropped)
  - In order to ensure this part of the web application is only accessible to PASS Organisers, it requires login with a username and password.

### 3.3.3 Non-functional requirements

The above features are functional requirements. There are also non-functional requirements. Using the FURPS model [17] (Functionality, Usability, Reliability, Performance, Supportability), here are some other requirements that were identified:

- To ensure usability, the web application is designed with intuitive use in mind, providing basic instructions to the user as part of the application where necessary.
- Again for usability, the look and feel of the user interface has been focused on (As will be shown in Chapter 5, CSS styling and JavaScript effects have been used).
- The system should be reliable, providing clear error messages to the user when necessary.
- For a user with a fast broadband internet connection, pages should have load without considerable delay, say, within 5 seconds.
- Major Internet browsers should be supported. Although it is infeasible to test the web application with the large number of minority browsers, it is expected that it should work with modern standard-compliant browsers.
- The application should be able to work without JavaScript enabled in a browser. (This was achieved in all but one place: the editing of schedules. A non-JavaScript version was not implemented here due to lack of time)

### 3.3.4 Database Design

The classes of the schedule model are kept to a minimum; they contain no persistence functionality. This is instead left to the web application. It stores the data for scheduling, and the generated schedules in a relational database. The MySQL Database Management System was chosen for this as it is available for free and there is much documentation available online.

## Chapter 4

# PASS Scheduler Implementation

This chapter explains the most challenging part of the project: the formulation of the schedule generation as a CSP, and its implementation in JaCoP. The development of this scheduler was incremental, so this chapter reflects this by beginning with a minimal formulation and building on it as the chapter progresses.

### 4.1 A First CSP Model for PASS Scheduling

As described in Chapter 2, to define a CSP, we have to define the sets of variables, domains and constraints. We denote the number of leaders by  $l$ , students by  $s$ , groups by  $g$ , and time slots by  $t$ .

Let us first define variables and their domains. We will use the following variables:

- Leader placement variables  $lp_1, \dots, lp_l$  with domain  $\{1, \dots, g\}$ , where assigning  $j$  to  $lp_i$  places leader  $i$  in group  $j$ .
- Student placement variables  $sp_1, \dots, sp_s$  with domain  $\{1, \dots, g\}$ , where assigning  $j$  to  $sp_i$  places student  $i$  in group  $j$ .
- Group placement variables  $gp_1, \dots, gp_g$  with domain  $\{1, \dots, t\}$ , where assigning  $j$  to  $gp_i$  places group  $i$  in time slot  $j$ .

These variables are sufficient to allow the representation of a basic schedule where leaders and students are placed in groups, and groups are placed in timeslots.

Next we define constraints over the variables. We will use the following constraints:

- $globalCardinality(\{lp_1, \dots, lp_g\}, 2, 3)$ ; there must be between 2 or 3 leaders per group.
- $globalCardinality(\{sp_1, \dots, sp_g\}, 15, 16)$ ; there must be between 15 and 16 students per group.



- *globalCardinality*( $\{gp_1, \dots, gp_t\}, minGroupsPerTimeslot, maxGroupsPerTimeslot$ );  
there must be between *minGroupsPerTimeslot* and *maxGroupsPerTimeslot* groups per timeslot.

The *globalCardinality*( $X, i, j$ ) constraint states that number of occurrences of a value in a set of variables  $X$  is bounded between  $i$  and  $j$ .

These constraints complete the definition of a basic CSP to find a schedule. However, we have not yet considered availability of leaders, or similarity of degree courses.

Let us first deal with availability of leaders. We will already know which time slots each leader is available for. So we can add in constraints disallowing the placement of leaders into groups for which the timeslot of the group is not available to the leader.

For each leader  $i$ , and time slot  $j$  for which leader  $i$  is unavailable, we place a constraint for every group  $k$ :

$$(gp_k = j) \Rightarrow (lp_i \neq k);$$

if group  $k$  is placed in time slot  $i$ , leader  $j$  may not be placed in group  $k$ . Hundreds of constraints of this form are likely to be required based on the expected numbers of leaders and time slots.

Now let us consider how to ensure similar degree courses within a group. We *could* add the following constraint for every leader  $i$  and student  $j$  where the leader and student have dissimilar courses:

$$lp_i \neq sp_j;$$

leader  $i$  and student  $j$  cannot be placed in the same group. However, it is very unlikely that all these constraints could be satisfied, so rather than add these constraints, this is dealt with through optimisation, described in section 4.4.

## 4.2 Constraint Programming with JaCoP

The above CSP was implemented in Java using the JaCoP library. Here is a brief explanation of how this is done, for a full guide on JaCoP see [6].

JaCoP allows finite domain variables to be created, and their domains specified. Here is an example, showing how the variables for the leader placements described above were created:

```
// Create variables each representing the placement of a leader in a group
Variable[] leaderInGroupPlacements = new Variable[noOfLeaders];
for(int i = 0; i < noOfLeaders; i++)
    leaderInGroupPlacements[i] = new Variable(store, "ligp"+i, 1, noOfGroups);
```

An array of variables is created, each variable having a domain  $\{1, \dots, \text{noOfGroups}\}$ .

A CSP is created using a `Store` object. `Variable` and `Constraint` objects must be added to a `Store`. Constraints are added to the `Store` using `store.impose()`, as shown in this example for the global cardinality constraint on leader placements (described above):

```
// *** CONSTRAINT: There must be between minLeadersPerGroup
//      and maxLeadersPerGroup leader placements to each group ***
Variable[] leaderInGroupCounts = new Variable[noOfGroups];
for(int i = 0; i < noOfGroups; i++)
    leaderInGroupCounts[i] = new Variable(store, "ligc"+i,
                                          minLeadersPerGroup,
                                          maxLeadersPerGroup);
store.impose(new GCC(leaderInGroupPlacements, leaderInGroupCounts));
```

The defining of the global cardinality constraint using JaCoP is considerably more complicated than necessary in this situation, requiring the creation of a variable for each value in the domain, although this does potentially make it more powerful. The object `GCC` is the JaCoP implementation of the global cardinality constraint, and is a subclass of the `Constraint` object, from which all constraints in JaCoP are subclassed.

Once the `Variable` and `Constraint` objects defining a CSP have been created and added to the `Store`, an attempt can be made to solve it:

```
Variable[][] vars;
// ... set up variable array ...
Search label = new DepthFirstSearch();
SelectChoicePoint select = new SimpleMatrixSelect(vars,
new MostConstrainedDynamic(), new IndomainMax());
boolean result = label.labeling(store, select);
```

This searches for a single solution. If a solution is found, `result` will be true. The `SelectChoicePoint` object that is used for the search defines some options concerning how the search is to be performed. Here, `MostConstrainedDynamic` causes the search to choose variables that are the most constrained first, and `IndomainMax` causes the search to try assigning values from the top end of the domain of a variable first. The `vars` array will contain all the variables for which we want to find a value.

If a result was found, the assigned values of the variables can be retrieved by using `value()` function, for example, to get the group number that the 5<sup>th</sup> leader was placed in:

```
leaderInGroupPlacements[5].value()
```

### 4.3 Mapping CSP Result to Java Objects

The result produced by solving the CSP with JaCoP will be an assignment for each variable of an integer value from its domain. The `PassScheduler.schedule()` method requires that the output is a `PassSchedule` Java object containing `PassGroup` objects. Therefore a mapping from the variables and values of the CSP to POJOs (Plain Old Java Objects) is necessary. The variables of the CSP have minimal structure, but the schedule as a collection of objects has references between the objects, as shown in Figure 3.2. A short section of code near the end of the implementation of `schedule()` method in `JacopPassScheduler` constructs this schedule based on the result from JaCoP.

### 4.4 Optimisation

We can consider the constraints applied so far to be *hard constraints*: they must be satisfied or no result will be found. As an alternative we have *soft constraints*, which do not have to be satisfied. With soft constraints we want to apply them if possible, but we still want a result if it is not possible [7]. Soft constraints can be handled using an optimisation function, with the aid of the *reified constraint*. A reified constraint acts on a constraint and a boolean variable (with domain  $\{0, 1\}$ ), and ensures that the variable is true (1) if and only if the constraint is satisfied. By having each soft constraint as the constraint in a reified constraint, we can have as our optimisation function a count of the reification variables that are true. We wish to maximize this count to satisfy as many of the soft constraints as possible. In the case of JaCoP, we must inverse this count as it only supports minimization.

The soft constraints added for the scheduling model relate to ensuring that where possible, the leaders and students in each group are on the same or similar degree courses. There would be many possible ways of measure this, but it was decided that the best way of doing it would be to count the number of students for whom at least one leader is on the same degree course. While it would be of benefit to have 2 or 3 leaders on the same degree course, having at least one leader on the same degree course is the most important thing; so just measuring this keeps things simple (see section 6.2 for further discussion).

Let us define a soft constraint for each student  $i$ , as follows:

$$sp_i = slp_1 \vee sp_i = slp_2 \vee \dots \vee sp_i = slp_n$$

where  $slp_1, \dots, slp_n$  are leader placements for leaders on the same degree course as student  $i$ .

Each of these constraints is reified into a variable, and another constraint binds the inversed sum of these variables into an objective variable, which can then be minimized.

With JaCoP, rather than only searching for a single solution as shown in section 4.2, we can search for a minimal solution by selecting a cost variable:

```
label.setTimeout(timeInMinutes*60);
boolean result = label.labeling(store, select, cost);
```

The cost variable here is the objective variable described above. A time limit is set, so JaCoP will stop searching if that is reached before all solutions have been found, returning the best solution found so far.

Unfortunately, as described in section 2.5, optimisation problems are extremely hard. With the initial CSP model shown in section 4.1, the result of leaving it overnight produced a result not much better than an arbitrary solution. To get an idea of the scale of the search, we can see that with approximately 40 groups and 500 students, we would have  $40^{500}$  possible solutions. After applying constraints, we can discount many possible solutions, but even if the numbers of solutions are reduced by a factor of say, a billion, searching through all solutions is still implausible.

## 4.5 Redundant Model

A *redundant model*, or *dual model*, can be used to strengthen constraint propagation. This involves defining a second CSP model, which is not strictly necessary to solve the problem. This second model defines the CSP using a different representation, different to the first representation. The models are linked together by extra constraints. The reason for adding this redundant model is that it can aid the constraint propagation algorithms, so that more can be inferred from the initial constraints, and the search space will be reduced. [4]

This method of using a redundant model was added to the PASS scheduler. The second model essentially represents the problem in reverse. However, it is a slightly more complicated representation.

Rather than have a variable for the placement of each leader into a group, as in the initial model, the second model creates variables for the placement of a leader in a each particular position of each group. Let us add the following variables and domains for each group  $i$ :

- Leader in group position placement variables  $ltga_{i,1}, \dots, ltga_{i,\maxLeadersPerGroup}$  where assigning  $k$  to  $ltga_{i,j}$  places leader  $k$  in group  $i$ , position  $j$ . A variable  $ltga_{i,j}$  has domain:
  - $\{1, \dots, l\}$  when  $j \leq \minLeadersPerGroup$
  - $\{1, \dots, l + 1\}$  when  $j > \minLeadersPerGroup$

It is to be expected that not every group will be full, so not every variable can be assigned a leader number. This explains giving some of the variable domains of  $\{1, \dots, l + 1\}$  so that they may instead be assigned with a value representing “no leader”, for which  $l + 1$  is used. There will be  $g \times \maxLeadersPerGroup$  of these variables for dealing with leaders in this second leader model.

Let us define constraints on these new leader placement variables as follows:

- A global cardinality constraint is defined over all the variables, ensuring each leader number in  $\{1, \dots, l\}$  is assigned to exactly 1 of the variables, and that all the other variables are assigned the “no leader” value,  $l + 1$ .
- The notion of a “position” is necessary for this representation but has no meaning for the application, that is, all leaders are of the same importance regardless of their position; all positions are equivalent. For example, if we have a solution where  $ltga_{i,1} = 34$  and  $ltga_{i,2} = 6$ , and another solution which differs only in that these assignments are reversed so that  $ltga_{i,1} = 6$  and  $ltga_{i,2} = 34$ , then both solutions place leaders 6 and 34 in group  $i$ , only in different positions, so the solutions are equivalent. Therefore constraints are added to ensure that for a number of leaders that may be assigned to a group, those leaders can only be assigned to that group in one possible way. This is done using lexicographical ordering, described further in section 4.6.

Also, constraints are defined to map this second model to the first model.

- For each group  $i$ , group position  $j$ , and leader  $k$ , we define a constraint:

$$(ltga_{i,j} = k) \Rightarrow (lp_k = i)$$

For those variables representing positions that may represent no leader (where  $j > \text{minNumberOfLeaders}$ ), a slightly more complicated constraint must be defined:

$$((ltga_{i,j} = k) \wedge (ltga_{i,j} \neq l + 1)) \Rightarrow (lp_k = i)$$

As this second model is more complex than the first, only the leader placement variables were discussed. Similar variables and constraints were added as part of the second model for the placement of students in the positions of groups, and also for the placement of groups in positions of time slots.

This redundant model, while tricky to get working correctly, succeeded in improving the results of running the solver (more solutions, and therefore a more optimal solution, could be found in the same period of time).

## 4.6 Symmetry

After adding a redundant model, a better schedule could be found, but it was clearly still much worse than one found manually. The concept of symmetry, or rather, symmetry breaking, offered some hope.

Sometimes a number of the solutions found for a CSP may be considered equivalent to each other. The equivalence of the positions in a group as described in section 4.5 is one such example. Equivalent solutions such as these are known as symmetric solutions. They are a consequence of the representation chosen to model the problem, so it may sometimes be possible to reformulate the problem to avoid these symmetries. Otherwise, it may be possible to add symmetry breaking constraints, often based on lexicographical ordering.[8]

Lexicographical ordering defines a total order on solutions, breaking symmetry. Taking the position of the leader in a group situation from section 4.5, we can define an order by adding a constraint on each pair of adjacent group positions. For each group  $i$  and each position  $j$ , where  $j > 2$ :

$$ltga_{i,j-1} \leq ltga_{i,j}$$

If we have leaders 34, 6, 47 in a group, this lexicographical ordering allows the leaders to be placed only in the ascending order 6, 34, 47, eliminating symmetric solutions.

Other examples of symmetry are more complicated. There is also a symmetry in the assignment of leaders, students, and time slots to groups. Imagine in one solution we have assigned 2 leaders and 15 students to group number 17, and that group was placed in time slot 1. In another solution we may have the same 2 leaders and 15 students assigned to a different group number, say 18, also in time slot 2. There is nothing fundamentally different about group 18 compared with group 17, in fact they represent nothing more than numbers until assignments are made. This is a symmetry that can be broken.

This symmetry should be considered by looking at solutions as a whole, not just a pair of groups. A symmetry like this can be seen whenever we have a pair of solutions that contain the same group assignments, but in a different order.

We can restrict the order that assignments can be made using lexicographical ordering. The matrix in Figure 4.1 (a) shows assignment of a leaders to groups for a single solution. We can see that if we reorder the rows in this matrix, we have equivalent leader assignments but in a different order. There are many matrices possible by reordering the rows, each representing a different solution, but these solutions are equivalent for our purposes. We effectively need to order this matrix to restrict the permutations of the rows in the matrix to one possible order, reducing the possible permutations of ordering the  $n$  rows in the matrix from  $n!$  to 1. Figure 4.1 (b) shows those same rows, now ordered. By restricting permutations in this way, we can reduce the number of solutions by a factor of up to  $n!$ , greatly increasing the speed of the searching for an optimal solution.

Constraints were added to restrict symmetry in this way, and the result was found to be good. The scheduler was now able to find more optimal solutions in 5 minutes than were before found in a few hours. The precise formulation of these lex ordering constraints is a little tricky, so will not be explained here (see [7] for an example of how this can be done).

Various ways of applying these symmetry constraints were tried, for example, defining equivalence on leader assignments, or as an alternative on student assignments. Defining the symmetry constraints on leader assignments seemed to work better, as there are only 2 to 3 leaders per group so the number of constraints necessary was considerably less. Sometimes adding constraints can be a trade-off. The number of solutions to search may be reduced, but at the same time the amount of time processing constraints may be increased. If constraint propagation takes place during the search, then with many the search may be considerably slower.

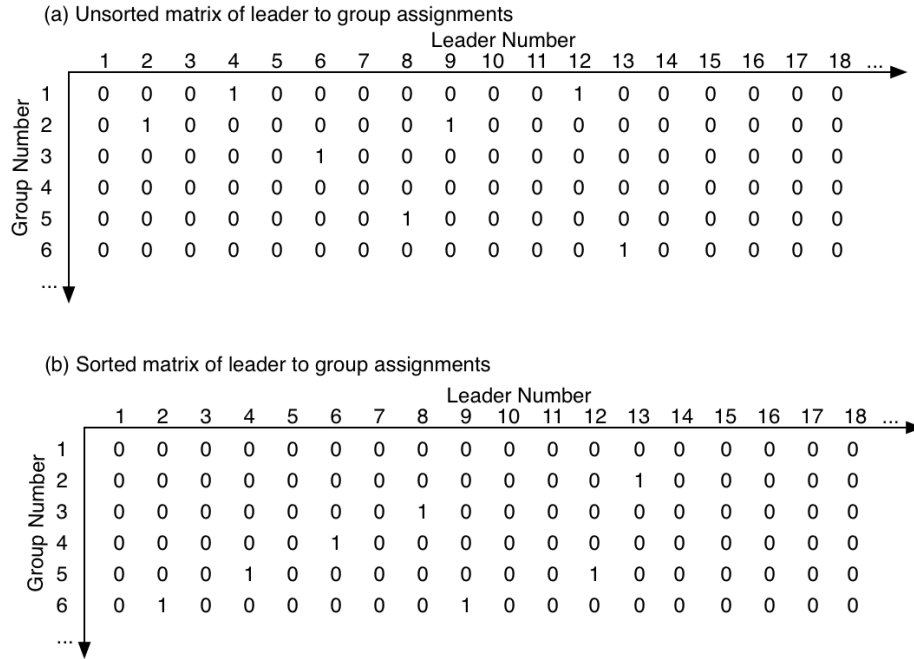


Figure 4.1: Lexographical ordering of rows in a matrix of leader to group assignments.

## 4.7 The Final Formulation

Even after adding symmetry breaking constraints, the scheduler was not producing good enough results. It is easy to see when looking at a schedule how students could be moved around (into groups with a leader on their degree course) to improve a bad schedule, as students do not have any time slot restrictions. So in this respect the CSP model I have for the situation is more than is necessary, as the CSP describes the problem as a whole, meaning that many different combinations of leader and time slot assignments will be searched through, when the easiest way to find a good solution is by looking at students. A very simple algorithm could be written to make some obvious assignments for students to groups that have leaders on the same degree course. For example, if we have enough 15 Biology students, then we can assign all those to a group with a Biology leader.

An simple sorting algorithm of this kind was integrated into scheduler. Here is how the final scheduler works:

1. The CSP is created and a solution searched for. There is an optimisation function that works to find a solution that groups leaders on the same degree course together, as this is a likely quality a good schedule would have.
2. After the solution is found, a simple sorting algorithm looks for whole groups

of students that can be matched to a leader, and adds constraints that require these students to be in the group that leader is in.

3. Constraints are added so that the leaders and time slots for groups will be unchanged from the solution already found (effectively “locking” them in place) . However, students are only assigned to a group if an obvious solution was found in step 2. A solution for the CSP is searched for again, this time with the original optimisation function as described earlier in the chapter.
4. After the time limit for searching is over, a reasonably solution will be available, but there may still be some obvious student to group assignments we could make to improve the schedule. A simple sorting algorithm does this.

The scheduler now produces a highly optimised schedule comparable to the one created manual (in a matter of minutes rather than days).

The steps taken in this final stage “over-constrain” the model so that it is unlikely that the optimal solution could be found; the scheduler now settles for a good solution.



## Chapter 5

# Web Application Implementation

This chapter highlights the web application side of the project in greater detail, including the technologies that were used, and the design of the user interface. An overview of using the web application, illustrated using screenshots, can be found in Appendix A.

### 5.1 Overview of Web Application Architecture

As mentioned in section 3.1, the system follows the MVC design pattern. Java Servlets form the Controller. There is essentially one servlet for each group of related use cases. For example, there is a single servlet that deals with the set-up of degree courses. Sometimes this corresponds with a single page of the web application, but in other cases it does not, for example the `Schedule.do` servlet controls the page for creating new schedules, and also the pages for viewing existing schedules. Servlets focus on control of data, for example setting up and forwarding on data to be displayed by a JSP, or retrieving data taken a HTTP POST request and calling a method that inserts into a database.

JSPs form the view in the MVC design pattern. There tends to be a JSP for each page that the user can navigate to (the exception in the being the navigation bar, which is a separate JSP included by other JSPs). A JSP is basically a HTML page, but with the inclusion of some extra tags that are processed each time the page is requested. These extra tags may insert parameters that were set by a Servlet, or may provide control such as conditional content or loops. The extra JSP tags are minimal, with any bulky processing being done by a Servlet, therefore allowing the focus to be on presentation when writing JSPs.

The MVC Model is represented by the CSP scheduling model, described in detail in Chapter 4. However, the web application also requires a database, so the code that deals with database interaction can also be seen as a Model. This is reflected in the Java package structure created for the web application, as shown in Table 5.1.

Note that JSPs are not Java classes so are not contained within a package.

Table 5.1: Java packages for web application

Package	Function
<code>uk.ac.man.cs.emeryj5.pass.model</code>	The CSP scheduling model
<code>uk.ac.man.cs.emeryj5.pass.app</code>	Web application control (servlets)
<code>uk.ac.man.cs.emeryj5.pass.data</code>	PASS database interaction

## 5.2 Database Interaction

A class `PassDatabase` performs all the database interaction. This is done using JDBC for MySQL, querying the MySQL database with SQL queries as necessary. The class essentially provides an extra layer of abstraction to the web application code. Rather than having SQL queries directly in Servlets, the Servlets call methods such as `PassDatabase.getDegreeCourses()`, which queries the database to get the data on degree courses, constructs Java objects from this data and returns an array of those `DegreeCourse` objects to the Servlet.

The `PassDatabase` class also implements some caching. When large amounts of data are retrieved from the database, that data is often saved into memory in case the same data is needed in the future, therefore reducing database queries, which tend to be slower than accessing internal data structures. For example, the `getDegreeCourses()` method saves all the `DegreeCourse` objects that it constructs into an `ArrayList`, and if the method is called a second time the `DegreeCourse` objects in this `ArrayList` are returned, rather than retrieving the data by calling the database again. This caching reduces the loading time of pages. The caching in its current form is possible only because this is the only system accessing the database. If multiple systems were to access the database, a more advanced caching technique would be required, in order to update the system correctly when other systems change the database.

The `PassDatabase` class is one of the largest in the web application, however it is relatively straightforward if JDBC and SQL are understood. Here is a brief example, the `getDegreeCourses()` method, with `//` comments explaining:

```
public static DegreeCourse[] getDegreeCourses()
{
    if(degreeCourseList==null) // if degreeCourseList is null there is no cache
    {
        // so try to retrieve degree courses from database
        try // Create a new SQL statement and use it to retrieve
        { // data from degree_course table
            Statement statement = getConnection().createStatement();
            ResultSet resultSet = statement.executeQuery
                ("SELECT * FROM pass_scheduling.degree_course");
            // create a temporary ArrayList in which to place data
```

```

ArrayList<DegreeCourse> tempDegreeList = new ArrayList<DegreeCourse>();
while(resultSet.next()) // for each row in the degree_course table,
{
    //we construct a DegreeCourse object
    // from course_code and course_name
    tempDegreeList.add(new DegreeCourse(
        resultSet.getString("course_code"),
        resultSet.getString("course_name")));
}
// a second SQL query retrieves data from similar_course
resultSet = statement.executeQuery
    ("SELECT * FROM pass_scheduling.similar_course");
while(resultSet.next())
{
    String firstCode = resultSet.getString("first_course");
    String secondCode =resultSet.getString("second_course");
    DegreeCourse firstCourse = null;
    DegreeCourse secondCourse = null;
    // search through the DegreeCourses
    // find the two similar degree courses
    for(DegreeCourse dc : tempDegreeList)
    {
        if(firstCode.equals(dc.getCourseCode()))
            firstCourse = dc;
        if(secondCode.equals(dc.getCourseCode()))
            secondCourse = dc;
    }
    // call a method on each degree course to set it
    // as being similar to other degree course
    if(firstCourse!=null && secondCourse!=null)
    {
        firstCourse .addSimilarDegreeCourse(secondCourse);
        secondCourse.addSimilarDegreeCourse(firstCourse);
    }
}
// all degree courses are loaded,
// so make temporary list the permanent cache list
degreeCourseList = tempDegreeList;
statement.close();
}
catch(SQLException e) { System.err.println(e); }
}
// return the list of degree courses as an array
return degreeCourseList.toArray(new DegreeCourse[0]);
}

```

As can be seen from the example, this mapping of relational database structures to Java data structures can be a little tricky, but as it is all encapsulated within one Java class, it is easy to use elsewhere. With the `PassDatabase` class in place, all that has to be considered when programming the servlets, is that there is some

data store that must be accessed using the methods provided; allowing the details of mapping data structures and caching data to be hidden.

### 5.3 Integrating the Scheduler into the Application

The scheduler takes a minimum of about 5 minutes to complete. Web users expect pages to load without noticeable delay, so clearly the scheduler cannot be run while a page is loaded. Instead the scheduler must be run in the background on the server, allowing the user to access pages while it is in progress. This was implemented by creating a separate Java **Thread** in which to run the scheduler. Whenever the form to create a schedule is submitted, a new **Thread** is created. More than one scheduler may be processing at the same time in different threads. If the server is a multi-cored computer, then one thread can run per core, so multiple schedules could be generated in the same time span as a single schedule.

The processing of a schedule is well integrated into the web application, providing feedback to the user on the progress. For each schedule created, a new page is added to the navigation bar to view the schedule. If the schedule is still processing, that page will display an estimated time until it is complete. This updates regularly until the processing is complete, in which case, assuming success, the schedules structure will be displayed. If no schedule is found a notice will inform the user of possible ways to enable a schedule to be found next time.

### 5.4 User Interface

Screenshots showing the user interface can be seen in Appendix A. The user interface for PASS leaders was quite simple, corresponding to the basic functionality required for it. The user interface for the PASS organiser is more complicated. A navigation menu appears on every page, and the links on this are split into categories of data setup and schedules. The data setup pages are split up by the type of data, that is, a page each for students, degree courses, general setup, *etc.* Each data setup page allows all the CRUD and other operations on that data type, as described in section 3.3.

The scheduling pages consist of a page for creating a new schedule, and a page for each created schedule. After submitting the simple form on the schedule creation page, the schedule will process as described in section 5.3. This processing page provides important feedback to the user. Firstly, it shows rotating arrows, as a signal to the user that something is actually happening. Secondly, it shows the estimated time remaining, and updates this every 10 seconds, so the user can see that some measurable progress is occurring. For a finished schedule, a list of groups in the schedule is shown, with each group being shown as a table, showing the leaders, students, and timeslot for that group. Alternative representations of the schedule would have been possible, but this shows all the necessary information together on one page in a clear way, so was considered the best option.

CSS (Cascading Style Sheets) was used to apply styling to the pages: this keeps presentation separate from the content, allowing more focused development and accessible web pages.

## 5.5 Use of JavaScript

JavaScript was used throughout the application to improve the user interface. This was done with the aid of the script.aculo.us JavaScript effects library. Script.aculo.us can provide visual effects for hiding and showing content, and also drag and drop functionality for lists in web pages [14]. Many of the features such as adding new student, or removing a schedule, are presented to the user in boxes near the top of the page (I have named these *Action Boxes*), near the main content. To give more focus to the main page content, JavaScript was used to collapse (with a script.aculo.us effect) the Action Boxes allowing them to be expanded when clicked on (without reloading the page).

Another area where JavaScript was used is on the schedule processing page. This page uses Ajax requests (that is, a HTTP request in the background, without reloading the page). The Ajax requests return the current estimated time, and this is displayed on the page. The Ajax requests were dealt with using the Prototype JavaScript library, see [12] for more info.

The principle of unobtrusive JavaScript was used. JavaScript may not be available in all web browsers, or some users may have chosen to disable it, so unobtrusive JavaScript is a design principle that allows the page to function correctly without JavaScript in such environments. Unobtrusive JavaScript keeps JavaScript code out of the HTML, similar to the way CSS styling is treated [14]. So in this application, the Action Boxes are open by default when the page initially loads, and then are instantly collapsed by the JavaScript. If a browser does not run the JavaScript code then the Action Boxes will always remain open and therefore not impede functionality.

The editing of a schedule after it has been created is an important feature. It is quite likely that after the schedule is created, some changes may be necessary due to unforeseen circumstances. Perhaps a leader has changed the schedules that he is available for, meaning he will need to be moved to a different group. Or perhaps the automatic scheduler has placed a single Genetics student in a group without Genetics leaders, and there is another group full of Genetics leaders and students but it already has the maximum number of students in. The PASS organisers may decide it is worth making an exception, and move the lone Genetics student into the other group. This feature was implemented using the drag and drop functionality of the script.aculo.us library. This allows items in a HTML list (<li> tags within <ul> tags) to be moved around that list, and also dragged to entirely different lists of the same class (see [15] for example of drag and drop between lists). This is used to enable leaders to be dragged from one group to another, and the same for students. Each time a student or leader is moved, an Ajax request is sent to the server so that

the result of the move is stored by the application (otherwise the drag and drop action would be lost if the page were reloaded).

## 5.6 Login

The application requires users to log in, to access both the leaders interface and the organisers interface. Leaders use their email address as username and a randomly generated 6-character password, while organisers use hardcoded username and password (this could easily be changed to be stored in the database if changing of the password were required). The login is implemented using a Java Servlet *filter*, which is run before any other servlet in the application. If the user is already logged in, the servlet that was being accessed will continue to be processed. If the user is not logged in, that servlet is not processed and a login form is displayed to the user instead. Whether or not the user is logged in is recorded by a session variable on the server that will contain the user name if the user is logged in. This is an effective way of ensuring logging in is necessary for all the servlets in the web application, without changing the code for those servlets themselves.

More could have been done to ensure access is secure. Primarily, adding some encryption such as SSL to protect against the interception of login details. However, the application does not give access to highly confidential information; security was not a major concern, so time was best spent elsewhere.

## Chapter 6

# Testing and Evaluation

Due to the general nature of CSP solving algorithms, and having concentrated on the formulation of the problem rather than algorithms that are used to solve it, analysis in any formal manner would be difficult. Therefore testing and evaluation of the scheduler was empirical; it is described in this chapter along with testing of the web interface.

### 6.1 Testing the Scheduler

As the data from the current year of the PASS scheme in the Faculty of Life Sciences was available, it was possible to use this for testing. This data included students and leaders both with their degree courses specified, and the available time slots for leaders.

In line with the incremental development of the scheduler, testing occurred frequently as changes were made. Early versions were tested simply using numbers to represent students, leaders, *etc* in the CSP, and later on when the real data was received, testing was based on that. The web interface was developed later on in the development process so earlier versions were tested using a Java class to import data and run the scheduler. When the web interface was finished testing became simpler, partly because a user interface was provided, and partly because setup data was saved a database, so any change in the scheduler parameters did not have to be hard-coded into a testing class. Much of the effort on testing was expended throughout the development, influencing changes in the design and implementation, but this chapter focuses on testing and evaluation of the final result.

The final schedule used for this year was available as a part of the real world data for PASS. This meant it would be possible to compare the schedules generated by this tool against this schedule, but first some measure for comparison had to be decided upon.

## 6.2 Measuring Schedule Quality

The quality of the schedule is solely based on the degree courses of the members of PASS groups. The ideal situation is that each student should have leaders that will have knowledge in his area of study, so his leaders will be on the same degree course, or if that is not possible, a very similar degree course. However, it is very unlikely (from looking at past experience with human-created schedules) that it will be possible to have all leaders of a group on the same or a similar degree course to a student. A more achievable aim is to have at least one of the leaders on the same or a similar degree course to the student, so this was chosen as the primary method of measuring the quality. This can be split into:

**Measure A** The number of students with at least one of their leaders on an identical degree course (ideal). For simplicity, this was chosen as optimisation function of the CSP, as it is the most important factor in scheduling.

**Measure B** The number of students with at least one of their leaders on a similar but not identical degree course, but no leaders on an identical course (less than ideal but still good). This was not optimised by the CSP, but the simple sorting algorithms described in section 4.7 will swap students from groups where they have dissimilar leaders to groups where they will have similar leaders.

With Measure A and Measure B in place for 2 schedules, how do we decide which schedule is of better quality?

- If Measure A and Measure B are both equal for the schedules, then it is clear they are of the same quality.
- If Measure A + Measure B is greater for one schedule, and Measure A for that schedule is also greater, then we can be sure that schedule is of better quality (as Measure A is more important than Measure B).
- If Measure A + Measure B is greater for one schedule, but Measure A for that schedule is less than for the other schedule, we can not be sure which schedule is better, as we have not defined how much more important Measure A is than Measure B.

This third possibility is a problem. For example, if we have Measure A at 40 for the first schedule, and at 39 for the second schedule, but we have Measure B at 0 for the first schedule and 300 for the second, then we have a much better Measure A + Measure B for the second schedule but it has a very slightly worse Measure A. The second schedule is likely to be much more desirable here, but so far we have no way of measuring that.

If two schedules in this category were being compared by a PASS organiser, they could decide between them quite easily based on their own knowledge of the situation, unless there is not much difference between them, in which case it would not matter which was picked. However, for the purposes of testing I will say that



Measure A is 1.2 times as important as Measure B, and that will at least allow decisions to be made on situations like the above example. So:

$$\text{Overall schedule quality} = 1.2 \times \text{Measure A} + \text{Measure B}$$

In practice a PASS organiser who has a good knowledge of the situation could find this too restricting, but it will do for the purposes of testing.

### 6.3 Comparison to a Manual Schedule

The data for this years PASS scheme was provided on spreadsheets, one with a list of students and their degree courses, and another with a list of leaders, their degree courses, and availability. These two spreadsheets were converted into CSV files and then imported using the import students and import leaders features of the web application. Also, the similarities between degree courses were identified and entered into the system (they had not been explicitly identified when the scheduling was done by a person). The general setup parameters were chosen according to the initial specification, with 2 to 3 leaders, and 15 to 16 students per group.

The comparison to the manual schedule was done by creating a schedule with the web interface, then moving leaders and students around until they matched the structure of the the manual solution (which I had on a spreadsheet). The web interface can display the statistics for Measure A and Measure B above. Table 6.3 shows these measures for both the manual and generated schedules (along with the count of students that did not have any similar leader.

Table 6.1: Comparison of Generated and Manual schedule

<b>Schedule</b>	<b>Measure A</b>	<b>Measure B</b>	<b>Measure A+B</b>	<b>None Similar</b>
Generated	373	25	398	72
Manual	368	32	399	71

The numbers are very close. In practicality the quality of each schedule is so close that the difference is inconsequential. It was noticed that the manual schedule did not keep to the specification given for the this automatic scheduler. The generated scheduler always places between 15 and 16 students in each group, but the manual schedule does not always keep to this rule, often placing more or less students in a group when convenient. So the automatic scheduler creates a schedule of similar quality even with stricter constraints. It may be that in practice these rules can be broken, for example if we have 18 Genetics students, allow them to be in the same group with a genetics leader. But adjustments like this should be left until after the schedule is created, then changes made using the drag and drop functionality provided.

## 6.4 Testing of Web Application

Testing of the web interface was mainly achieved alongside its development: as a new page or feature was added, it was also tested with reasonable thoroughness, and any incorrect behavior fixed.

Towards the end of development the web interface was tested on a selection of major web browsers (Firefox, Internet Explorer, Safari, Opera) to ensure that it worked on each of them. This was not as thorough as the incremental testing, instead focusing on client-side aspects such as layout of pages and behavior of javascript (server side behavior should not be affected by the browser).

# Chapter 7

## Conclusions

This conclusion summarises what I have been achieved and learnt, including some personal comments.

### 7.1 Achievement of Objectives

In section 1.3, three milestones were listed. Milestone 1, creation of a schedule (without considering its quality) was clearly met. Milestone 2, requiring a schedule of as good quality as one created manually, was difficult to measure because the requirements contradicted the manually created schedule I was given to measure against. I would argue that this milestone was met, as the results were very nearly as good as the manual schedule, even with stricter parameters. Milestone 3, creation of an easy to use web interface, was met, although some features were dropped.

### 7.2 Improving the Scheduler CSP

While the final scheduler was satisfactory, I was slightly disappointed that I had to add extra sorting code to force the constraint library to choose obvious assignments. If I had found a way to get the scheduler working using only constraint programming I would have been more satisfied<sup>1</sup>. It is unlikely that any further work will be done on the scheduler developed in this project, as it already produces good schedules, I would recommend two areas to look at:

**Search strategy** In this project I have focused on finding a good formulation of the CSP, only spending minimal time looking at search strategies. My exploration of this area was limited to experimenting with the available search options provided by the JaCoP library, and choosing those that found solutions quickest. A more intelligent search strategy could be developed, specific to the situation. This might have a similar effect to the simple assignment algorithms I added

---

<sup>1</sup>Perhaps it would be better to say I was satisfied with my success with CSPs, but it was not optimal!

to the scheduler, encouraging the constraint library to search the most likely solutions first. However, in one respect this should be better than the algorithms I added, as they force certain likely assignments, whereas this could try those likely assignments, but then go on to try other assignments, so it might be possible to find the most optimal solution.

**The formulation of the CSP** There are many possible ways of formulating a CSP. This is shown in [10], showing that even the relatively simple n-Queens problem has at least 9 possible formulations. So it is possible that a better formulation could have been found for the PASS scheduling situation, especially if the attempt were to be made by someone with a good deal of past experience in constraint programming.

### 7.3 Improving the Web Application

It is quite possible that the organizers of the PASS scheme would want to have the web application developed further, as it can be of great practical use to the scheme. As mentioned, some features of the web application were dropped during development, as mentioned in section 3.3. If any future work were to be done on the web application, these should be the first to be implemented. The emailing feature could be particularly important to ensure the system is useful, as without it, emails addresses and PASS group organization would have to be copied and pasted and then composed individually. If this were in the application, it could provide a kind of mail merge functionality, automatically inserting the name and PASS group information into each email.

In addition to those dropped features, I would recommend the consideration of adding the following features:

- Exporting of schedule into a spreadsheet. This would be useful if anything unexpected was required with the schedule, as the organisers of PASS should be quite capable of manipulating data on a spreadsheet.
- The system currently only handles one year of students. In its current form, each year all the data would have to be deleted (one data item at a time) before the next years is added in. This could be dealt with most easily by providing a reset feature, which would quickly delete everything ready for another year. A more complicated solution is to allow a different data set for each year.

### 7.4 Changes to Original Plan and Expectations

When I started this project, I underestimated the work required on the scheduler. I expected that the scheduler would be a similar amount of work to the web interface. I expected that I could write the scheduler in my first semester, and the interface in the second. But I soon began to appreciate how difficult the scheduling problem actually

was, so I adjusted my plan accordingly, and in the end around three quarters of my time were spent on it. The web interface was created in less than a month, alongside finishing off the scheduler and attending lectures, although this was probably only possible because I have some past experience in this area. As a result of this reduced time for the web interface, some of the “nice to have” features, such as emailing functionality, were left out. However I am generally pleased with this outcome as it allowed me to focus on the more challenging aspects of the project.

## **7.5 Comments on the use of Constraint Satisfaction**

I found Constraint Satisfaction to be very different to anything I had done before in programming. I found the formulating a CSP difficult, with most of the examples that I was able to find being simple “toy problems”, and more advanced material being excessively difficult to comprehend. However, as stated by K. Apt in [2], modeling using constraints is “more an art than science”, and seems to be an activity where years of experience would be greatly beneficial. Having said that, this is the most computationally difficult problem I have written any kind of program to solve, so I could now solve more simple problems using CSPs with ease. I expect I will find use for constraint programming in the future.

# Bibliography

- [1] Rina Dechter, *Constraint Processing*, Morgan Kaufmann, 2003
- [2] Krzysztof R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003
- [3] Stuart Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach (Second Edition)*, Chapter 5, Prentice Hall, 2002
- [4] *Documentation for Choco Constraint Library*, 22nd April 2009  
[www.emn.fr/x-info/choco-solver/doku.php?id=documentation](http://www.emn.fr/x-info/choco-solver/doku.php?id=documentation)
- [5] *JaCoP Constraint Programming solver*, 22nd April 2009  
[jacop.osolpro.com](http://jacop.osolpro.com)
- [6] *JaCoP Library User Guide*, 22nd April 2009  
[JaCoPguide.osolpro.com/guideJaCoP.html](http://JaCoPguide.osolpro.com/guideJaCoP.html)
- [7] *Lecture Notes for a Lund University Course on Constraint Programming*, 3rd February 2009  
[www.cs.lth.se/EDA340/schedule.html](http://www.cs.lth.se/EDA340/schedule.html)
- [8] *Understanding and Managing Symmetry in CSPs*, 23rd February 2009  
[www.cs.rhul.ac.uk/home/green/mathscsp/slides/talks/linton/handout.pdf](http://www.cs.rhul.ac.uk/home/green/mathscsp/slides/talks/linton/handout.pdf)
- [9] Kim Marriott, Peter J Stuckey, *Programming with Constraints: An Introduction*, The MIT Press, 1998
- [10] Bernard A Nadel, *Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens*, IEEE Expert, Volume 5, Number 3, June 1990, 16-23
- [11] Bryan Basham, Kathy Sierra, Bert Bates, *Head First Servlets and JSP (Second Edition)*, OReilly, 2008
- [12] *Prototype Tutorials*, 23rd April 2009  
[www.prototypejs.org/learn](http://www.prototypejs.org/learn)
- [13] *Behavioral Separation*, 29th April 2009  
[www.alistapart.com/articles/behavioralseparation](http://www.alistapart.com/articles/behavioralseparation)

- [14] *script.aculo.us Documentation*, 23rd April 2009  
[wiki.github.com/madrobby/scriptaculous](https://github.com/madrobby/scriptaculous/wiki)
- [15] *Drag and Drop with Scriptaculous Lists*, 23rd April 2009  
[/www.gregphoto.net/sortable/advanced/](http://www.gregphoto.net/sortable/advanced/)
- [16] *Create, read, update and delete*, 4th May 2009  
[en.wikipedia.org/wiki/Create,\\_read,\\_update\\_and\\_delete](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)
- [17] Craig Larman, *Applying UML and Patterns*, Prentice Hall, 2004

## Appendix A

# Application Demo

This provides a walkthrough of using the application, along with screenshots that show the user interface.

### A.1 Login

The login screen provides the entry point for both the organisers of the PASS scheme and PASS leaders. The red background used throughout the application is the colour of the Faculty of Life Science.

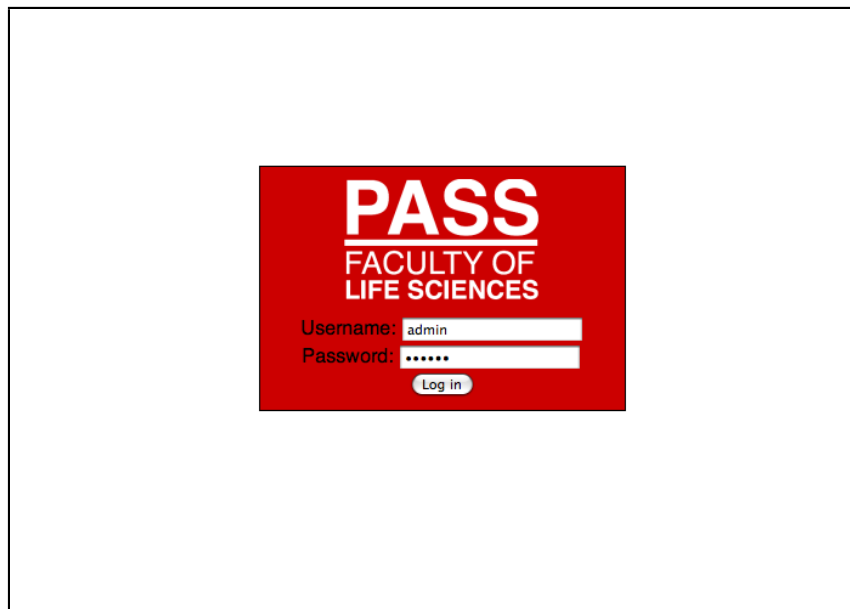


Figure A.1: This login screen for the PASS scheduler.



## A.2 Data Setup

Before a schedule can be generated, data must be input. This is aided with CSV import features for student and leader data.

The screenshot shows the 'General Set-up' page of the PASS SCHEDULER. On the left is a red sidebar with the logo 'PASS SCHEDULER FACULTY OF LIFE SCIENCES' and navigation links: 'Log out', 'Set-up' (with sub-links for 'General Set-up', 'Students', 'Leaders', 'Time Slots', 'Degree Courses', and 'Create New Schedule'), and 'View Schedule:' (with sub-links for 'Schedule 1', 'Schedule 2', and 'Schedule 3'). The main content area is titled 'General Set-up' and contains the text 'A schedule must have:'. Below this are three rows of input fields: 'Between 2 and 3 Leaders Per Group.', 'Between 15 and 16 Students Per Group.', and 'Between 6 and 7 Groups per Time Slot.'. A 'Save' button is located below these fields.

Figure A.2: The general setup page.

The screenshot shows the 'Time Slots' page of the PASS SCHEDULER. The left sidebar is identical to the one in Figure A.2. The main content area is titled 'Time Slots' and features a red 'Add Time Slot' button. Below this is a form for 'Description of Time Slot:' with a text input field and a placeholder '(eg. "Friday 1pm")', followed by a 'Create Slot' button. A table below lists existing time slots with columns for 'Time Slot Description', 'Delete', and 'Edit'.

Time Slot Description	Delete	Edit
Tuesday 11:00-12:00		
Tuesday 12:00-13:00		
Tuesday 14:00-15:00		
Thursday 14:00-15:00		
Friday 10:00-11:00		

Figure A.3: The time slot setup page.

**PASS**  
**SCHEDULER**  
**FACULTY OF**  
**LIFE SCIENCES**  
  
Log out  
**Set-up**  
General Set-up  
Students  
Leaders  
Time Slots  
Degree Courses  
Create New Schedule  
  
View Schedule:  
Schedule 1  
Schedule 2  
Schedule 3

### Degree Courses

Add Degree Course

Add/Remove Degree Course Similarity  
Anatomical Sciences is similar to Anatomical Sciences  
Add Similarity  
Remove Similarity  
Save

Course Code	Course Name	Similar Courses	Delete	Edit
2	Anatomical Sciences			
0	Biochemistry	Molecular Biology Medical Biochemistry		
19	Biological & Computing Science	Biology with Science & Society Biology with Business & Management Biology		
8	Biology	Biology with Business & Management Life Sciences Biology with Science & Society Zoology Biological & Computing Science		
11	Biology with Business & Management	Biology Biology with Science & Society Biological & Computing Science		
16	Biology with Science & Society	Biology with Business & Management Biological & Computing Science Biology		
1	Biomedical Sciences	Zoology		

Figure A.4: The degree course setup page.

**PASS**  
**SCHEDULER**  
**FACULTY OF**  
**LIFE SCIENCES**  
  
Log out  
**Set-up**  
General Set-up  
Students  
Leaders  
Time Slots  
Degree Courses  
Create New Schedule  
  
View Schedule:  
Schedule 1  
Schedule 2  
Schedule 3

### Students

Import data from CSV File

Add Student

Title	First Name	Surname	Degree Course	Email	Delete	Edit
-			Anatomical Sciences	-		
-			Biochemistry	-		
-			Biochemistry	-		
-			Biochemistry	-		
-			Biomedical Sciences	-		
Miss			Biomedical Sciences	-		
-			Biomedical Sciences	-		
Mr			Biomedical Sciences	-		
Mr			Biomedical Sciences	-		
-			Physiology	-		
-			Medical Biochemistry	-		
Miss			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		
Mr			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		
Mr			Anatomical Sciences	-		
Mr			Anatomical Sciences	-		
Mr			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		
Miss			Anatomical Sciences	-		

Figure A.5: The students setup page.

As the data used for testing is real, people's names and email addresses have been deleted from these screenshots for their privacy.

Title	First Name	Surname	Degree Course	Email	Available Time Slots	Confirmed Availability	Password	Delete	Edit
-			Biochemistry		Tuesday 11:00-12:00 Tuesday 12:00-13:00 Tuesday 13:00-14:00 Tuesday 14:00-15:00	X	293324	✖	👤
-			Biomedical Sciences		Friday 10:00-11:00	X	e75b27	✖	👤
-			Anatomical Sciences		Friday 10:00-11:00	X	e31a4f	✖	👤
-			Biomedical Sciences		Tuesday 11:00-12:00 Tuesday 12:00-13:00 Tuesday 13:00-14:00 Tuesday 14:00-15:00	X	f87d5c	✖	👤
-			Molecular Biology		Tuesday 12:00-13:00 Tuesday 13:00-14:00 Thursday 14:00-15:00 Friday 10:00-	X	92753f	✖	👤

Figure A.6: The leaders setup page.

## A.3 Creating and Viewing Schedules

Once data has been set-up, a schedule can be generated.

Students can be dragged and dropped from one group to another if a manual change is necessary.

# PASS

## SCHEDULER

### FACULTY OF LIFE SCIENCES

[Log out](#)

**Set-up**

[General Set-up](#)

[Students](#)

[Leaders](#)

[Time Slots](#)

[Degree Courses](#)

[Create New Schedule](#)

**View Schedule:**

[Schedule 1](#)

[Schedule 2](#)

[Schedule 3](#)

## Create Schedule

Name:

Time limit:  minutes (minimum 5 minutes) [Go](#)

Figure A.7: The schedule creation page.

# PASS

## SCHEDULER

### FACULTY OF LIFE SCIENCES

[Log out](#)

**Set-up**

[General Set-up](#)

[Students](#)

[Leaders](#)

[Time Slots](#)

[Degree Courses](#)

[Create New Schedule](#)

**View Schedule:**


[Schedule 1](#)

[Schedule 2](#)

[Schedule 3](#)

[Schedule 4](#)

Schedule 4 is still processing



Time remaining (estimated):  
**5 Minutes**

[Cancel Schedule Processing](#)

Figure A.8: A schedule processing.

