A Constraint Modelling Language for Timetables

Project Report

James Matthews
(jxm03@doc.ic.ac.uk)

Supervisor: Dr. Sebastian Uchitel
Second Marker: Dr. Alessandra Russo

June 2006
Abstract

Timetables are complex. For this reason, few organisations possess reliable automated timetable solvers, and fewer still possess solvers that require no manual intervention.

This project introduces a new modelling language which allows the user to specify the domain of, and the constraints on, a timetable to be created.

Such a modelling language allows the creation of a generic timetable evaluator, which provides the next best thing to a timetable solver, guidance and feedback during the manual timetable creation process.

In the course of this project I have implemented an interpreter for my language, the above mentioned generic evaluator, and a front end specific to Imperial College Department of Computing that demonstrates the power of the evaluator in such fields as the creation of timetables and the integration of existing timetables.
Acknowledgements

I would like to thank my supervisor, Dr. Sebastian Uchitel for all his support. Without his advice and feedback, this project would have taken any number of wrong turns.

I would also like to thank Dr. Alessandra Russo for agreeing to be my second marker.

Furthermore I would like to thank…

Vanessa Ho Von for giving up some of her valuable time to help me evaluate my project.

Lastly, my housemates for accepting graciously that I haven’t been able to leave my room for the past two months.
# 1 Contents

## 1 CONTENTS

### 2 INTRODUCTION

2.1 INSPIRATION ................................................................. 11
   2.1.1 A System for Integrating Timetables ........................................ 11
   2.1.2 Previous Department of Computing Projects ............................ 11
2.2 THE OBJECTIVE ............................................................ 12
2.3 THE PROBLEM – INFORMING THE EVALUATOR ...................... 12
2.4 THE SOLUTION – A CONSTRAINT MODELLING LANGUAGE .......... 13
   2.4.1 The Meta-Model .................................................................. 13
   2.4.2 Constraints ....................................................................... 13
2.5 SOFTWARE CREATED FOR THIS PROJECT ............................ 13

## 3 BACKGROUND

3.1 WHY DO TIMETABLES PRESENT AN INTERESTING PROBLEM? .... 15
3.2 WHY CREATE A TIMETABLE EVALUATOR? ............................ 16
3.3 FACILITATING TIMETABLE INTEGRATION ............................... 16
3.4 PREVIOUS RESEARCH ....................................................... 16
   3.4.1 RCPSP ........................................................................... 17
   3.4.2 TTL .............................................................................. 18
3.5 EXISTING METHODS OF BUILDING A TIMETABLE EVALUATOR .... 18
   3.5.1 Constraint (Logic) Programming ......................................... 18
   3.5.2 Alloy Analyzer .................................................................. 19
   3.5.3 Temporal Databases ......................................................... 19

## 4 LANGUAGE REQUIREMENTS

4.1 EASE OF USE .................................................................. 20
4.2 GENERAL APPLICATION .................................................. 20
4.3 POWER OF EXPRESSION .................................................. 20
4.4 EFFICIENCY OF REPRESENTATION .................................... 20
4.5 COMPATIBLE REPRESENTATION ...................................... 21
4.6 PURELY DECLARATIVE CONSTRAINTS ............................... 21
4.7 EFFICIENT IMPLEMENTATION ......................................... 21

## 5 LANGUAGE CONCEPTS

5.1 THE META-MODEL ......................................................... 22
   5.1.1 Time ........................................................................... 22
   5.1.2 Resources ................................................................. 22
   5.1.3 Event Types .............................................................. 23
5.2 CONSTRAINTS .................................................................. 23
   5.2.1 Hard vs. Soft Constraints .............................................. 23
   5.2.2 Event Type Constraints ............................................... 24
   5.2.3 Global Constraints ....................................................... 24
5.3 TIMETABLE ................................................................. 25
5.4 Fixed Secondary Timetables .............................................................................. 25

6 LANGUAGE SPECIFICATION AND SYNTAX .................................................. 26

6.1 Basic Language Syntax .................................................................................. 26
  6.1.1 White Space ......................................................................................... 26
  6.1.2 Comments .......................................................................................... 26
  6.1.3 Identifiers ......................................................................................... 26
  6.1.4 Property References .......................................................................... 26
  6.1.5 Keywords ......................................................................................... 26
  6.1.6 Constants ......................................................................................... 26

6.2 Types and Type Checking ............................................................................ 27

6.3 The Meta-Model ......................................................................................... 27
  6.3.1 Time ................................................................................................. 27
  6.3.2 Resources ......................................................................................... 29
  6.3.3 Event Types ..................................................................................... 36

6.4 Constraints .................................................................................................. 40
  6.4.1 Event Type Constraints ................................................................. 40
  6.4.2 Global Constraints ......................................................................... 41
  6.4.3 Constraint Expressions ................................................................. 43
  6.4.4 Constraint Functions ...................................................................... 45

6.5 Timetables .................................................................................................. 47

6.6 Fixed Secondary Timetables ......................................................................... 50

7 INTERPRETER DESIGN AND IMPLEMENTATION ........................................... 52

7.1 Implementation Language and Tools ......................................................... 52

7.2 Interpreter Architecture ............................................................................. 52
  7.2.1 Flexible Plug-in Architecture .......................................................... 53

7.3 Lexical Analysis .......................................................................................... 54

7.4 Parsing ......................................................................................................... 55

7.5 Intermediate Representation ....................................................................... 55
  7.5.1 Resources and Event Types IR ....................................................... 55
  7.5.2 Constraint Expression IR ............................................................... 56
  7.5.3 Timetable IR .................................................................................... 57

7.6 Resource / Event Type Set Reduction ......................................................... 58

8 EVALUATION IMPLEMENTATION ............................................................... 62

8.1 Assessing Timetable Quality – The Timetable Rating .................................. 62
  8.1.1 Calculating the Rating – Single Valued Utility Function .................. 62

8.2 Complete Evaluation .................................................................................. 63
  8.2.1 Providing Feedback ......................................................................... 64

8.3 Partial Evaluation ......................................................................................... 64

8.4 Constraint Function Evaluation .................................................................. 64
  8.4.1 Requires All ..................................................................................... 64
  8.4.2 Requires One ................................................................................... 68

9 DOC TIMETABLE BUILDER ............................................................................. 70

9.1 Navigating the Timetable ............................................................................ 70
9.1.1 Filters .............................................................................................................. 71
9.2 ADDING SCHEDULED EVENTS ........................................................................ 71
  9.2.1 Selecting the Event Type ............................................................................. 72
  9.2.2 Adding Resources that are always required .............................................. 72
  9.2.3 Choosing a Time Slot ................................................................................. 73
  9.2.4 Finalising Resources .................................................................................. 75
9.3 MODIFYING, MOVING AND DELETING SCHEDULED EVENTS ......................... 75
9.4 EVALUATING THE TIMETABLE ...................................................................... 76

10 PROJECT EVALUATION .................................................................................. 78
  10.1 EASE OF USE ................................................................................................. 78
  10.2 EXPRESSIVENESS (AND GENERAL APPLICATION) .................................... 79
  10.3 USEFULNESS ................................................................................................. 79
  10.4 EASE OF USE AND EFFICIENCY OF REPRESENTATION – A TRIAL ............. 80
  10.5 EVALUATION EFFICIENCY ............................................................................ 81
  10.5.1 DoC Spring Term 2006 .............................................................................. 82
  10.5.2 Increased number of Scheduled Events .................................................. 82
  10.6 CONCLUSION ................................................................................................. 82
    10.6.1 Further Work .......................................................................................... 83

11 BIBLIOGRAPHY ............................................................................................... 84

12 APPENDIX A - DOC TIMETABLE EXAMPLE .................................................. 88
  12.1 BACKGROUND .............................................................................................. 88
    12.1.1 Some Typical Constraints .................................................................... 88
    12.1.2 Automatic Code Generation ................................................................. 89
  12.2 RESOURCES .................................................................................................. 90
    12.2.1 Resource Organisation .......................................................................... 90
    12.2.2 Abridged Resource Listing .................................................................. 91
  12.3 EVENT TYPES AND EVENT TYPE CONSTRAINTS ...................................... 94
    12.3.1 Event Type Organisation ...................................................................... 94
    12.3.2 Constraint Formulation ....................................................................... 95
    12.3.3 Abridged Event Type Listing ............................................................... 97
  12.4 GLOBAL CONSTRAINTS ................................................................................ 99
    12.4.1 Constraint Formulation ....................................................................... 99
    12.4.2 Abridged Global Constraint Listing ...................................................... 99
  12.5 FIXED TIMETABLE ....................................................................................... 100
  12.6 FIXED TIMETABLE FORMULATION .......................................................... 100
  12.7 FIXED TIMETABLE LISTING ...................................................................... 102

13 APPENDIX B – HIGHLIGHTS OF WIMBLEDON EXAMPLE .............................. 103

14 APPENDIX C – AVAILABLE TYPES .............................................................. 105
  14.1.1 Boolean .................................................................................................... 105
  14.1.2 Integer / Float .......................................................................................... 105
  14.1.3 String ....................................................................................................... 105
  14.1.4 Resource Set / Event Type Set ............................................................... 105
  14.1.5 General Sets ............................................................................................ 105
14.1.6 Time Expression

15 APPENDIX D – IMPLEMENTED CONSTRAINT FUNCTIONS

16 APPENDIX E – ANTLR

16.1 WHAT IS ANTLR?

16.2 WHY USE ANTLR?

16.3 SPECIFYING THE LEXER

16.4 SPECIFYING THE PARSER
2 Introduction

It would not take a great leap of faith to suppose that there has never been an organisation that did not make some use of timetables. Individual meeting schedules and calendars, project planning, train timetables and educational timetables are just a few examples.

This project demonstrates the design and implementation of an interpreted language that specifies, for an arbitrary given situation, what makes a timetable satisfactory, and furthermore what makes a timetable better.

The language is used by a generic timetable evaluator, which can easily be built upon to develop complex custom applications. To demonstrate that this evaluator is indeed useful, I have developed a small application specific to Imperial College London Department of Computing that makes extensive use of many of its functions.

2.1 Inspiration

2.1.1 A System for Integrating Timetables

This project was inspired by Dr. Alessandra Russo’s need for a system to automatically integrate two timetables, to be used in her role as Director of the Joint Mathematics and Computing Degree.

Dr. Russo envisaged two individual projects to complete this system. The first project, suitable for a BEng student, would look at automatically gathering and interpreting the required data and determining when and why conflicts had occurred between the two timetables. The second project, for a MEng student, would focus more closely on providing suggested solutions to the conflicts that had been found.

Although I was not allocated to this project, it was a problem that interested me, so I decided to define my own project around this theme. However, as I was no longer constrained by some of the restrictions and intricacies of the system specification, I was able to broaden and generalise the problem that my project attempted to solve.

2.1.2 Previous Department of Computing Projects

There have been several timetable related projects previously completed in the Department of Computing, covering a variety of problems (e.g. [30], [31]). An aspect that many have in common is that they identify constraints applicable to the timetable they are attempting to evaluate or generate.
Typically however, these constraints are hard coded, or at best constraints have limited elements of configurability. Furthermore, concepts in these solutions were often fixed. For example, it may be prescribed that there is a concept of a “lecture” and a constraint may be that each lecture requires a “room”. In most cases, adapting these solutions to a different situation, or modifying them when the current situation changed, would involve considerable re-programming.

By looking at some of these projects I saw that to allow the user the freedom to specify both the concepts of and constraints on a timetable would present an interesting challenge.

### 2.2 The Objective

The broad objective that I set myself was to develop a generic timetable evaluator, which would allow the user in the simplest way possible to define both the situation that the timetable had been developed for, and the characteristics of a good timetable. Such an evaluator could have a number of possible uses:

- To automatically determine the quality of and to give feedback on timetables generated by an automatic solver or otherwise.
- To integrate any number of timetables.
- To develop applications for the manual building of timetables that can give feedback and make suggestions to the user as the timetable is built.
- To potentially be used as part of an automatic solver.

### 2.3 The Problem – Informing the Evaluator

In order for a generic timetable evaluator to perform its function, it must be given descriptions of two things.

1. The domain of the timetable to be evaluated. That is:
   a. What time period does the timetable cover and how is that time divided?
   b. What aspects of the timetabling situation can be included on the timetable?
   c. What are the properties of these aspects and what are their relationships to each other?
2. Given all the possible timetables that could be created from this domain, which ones are solutions, and of any two solutions, which is the preferred.
2.4 The Solution – A Constraint Modelling Language

My project proposes that the evaluator should be informed by means of a new text based modelling language. The language should allow the user, or intermediary application, to declaratively define both the domain in which the timetable exists, and the constraints which are applicable to this timetable. The language should also provide the facility to specify the timetables to be evaluated.

2.4.1 The Meta-Model

A meta-model defines the domain of the timetable. A brief introduction to the concepts of the meta-model can be found in Section 5.1, while a more detailed description and details of how it is specified can be found in Section 6.3.

2.4.2 Constraints

Secondly, the language allows the user to specify a set of constraints over the meta-model, which determine the validity of a proposed timetable, and its comparative quality. Outline details of these constraints can be found in Section 5.2, and again a more detailed discussion can be found in Section 6.4.

2.5 Software Created for this project

All the software created for this project has been developed with a modular, component based architecture in mind. Completing this project has required the design and implementation of several such components.

The most significant development was an interpreter for the modelling language, which takes as input code and builds an advanced data structure and API based on the user’s specifications. The interpreter is a complex piece of software which has a component architecture in its own right. For details on the development of the interpreter please see Chapter 7.

Another development is the evaluator itself. It takes the data structures provided by the interpreter and provides either the users, or another application with an interface to query them. Details of some of the principals of evaluation can be found in Chapter 8. Some of the graph based techniques I developed in order to provide an efficient implementation can be found in Sections 7.6 and 8.4.
Lastly, I designed and implemented an application specific to the Department of Computing, which uses many of the evaluator’s functions and will be the focus of my demonstration. For details of this, please see Chapter 9.
3 Background

3.1 Why do timetables present an interesting problem?

Some timetables, such as meeting schedules, are created as needed, and serve as little more than reminders to their users. Other timetables however are planned in advance, and serve to coordinate many groups of users and many classes of resource. The creation of such timetables is normally subject to many constraints, some common to most situations and many far more esoteric.

The principal problem that faces many of the creators of such “advanced planning” timetables is that the number of possible timetables for their situation is potentially vast, yet the number of acceptable solutions may be comparatively tiny.

Example 3-1

A timetable may schedule events over a 7 day period, and events are listed as occurring in one of the $24 \times 7 = 168$ hours of this period.

The timetable must schedule exactly 20 distinct events over this period, each of which could have attached to it any combination of 50 resources.

Each event could utilise $2^{50} = 10^{15}$ possible different combinations of resources.

Therefore, the total number of combinations of resources amongst the 20 events (assuming that it is sometimes ok to allocate the same resource to the same event at the same time, which it may be) is $2^{50 \times 20} = 10^{301}$

There are $168^{20} = 10^{44}$ different arrangements of these 20 events.

Therefore the total number of distinct timetables possible in this situation is $10^{44} \times 10^{301} = 10^{345}$

Of course if it is possible to schedule unlimited scheduled events, the number of possible distinct timetables is infinite.

Specifying which of the potentially vast number of possible timetables for a given situation are acceptable, without explicitly listing them, can be a challenging task. An even bigger problem is generating an acceptable solution from this specification.
3.2 Why create a timetable evaluator?

As demonstrated in the previous section, any timetabling problem of real world proportions is likely to have considerable complexity.

For this reason, creating a reliable automatic solver which requires no manual intervention is a very difficult problem, and most organisations do not have such a solution.

Instead, most timetables are created manually by expert administrators who have deep knowledge of the requirements of all parties involved.

I believe that a primary use of an evaluator for such timetable would be to give assistance to the manual creating and modifying of timetables. The human makes every decision, but can be guided as to what s/he can and cannot do, and can be given suggestions as to what is the better thing to do.

3.3 Facilitating Timetable integration

A further use of such an evaluator would be to integrate existing timetables which share the same domain, and therefore the same set of resources with finite availability. By evaluating two timetables as one, conflicts and violations that occur between the two timetables would be highlighted in a similar manner to violations detected on a single timetable.

3.4 Previous Research

Many hundreds of research papers have been published in the last fifteen years which address some aspect of “the timetabling problem”. Indeed, there is even an international conference on the “Practice and Theory of Automated Timetabling” (PATAT) which has taken place biannually since 1995. [2]

Most research has, understandably, concentrated on automatically solving difficult static timetabling problems. Common methods include genetic algorithms [3], linear programming [4], tabu search [5] and constraint programming [6].

Limited research has also been undertaken on solving dynamic timetabling problems, in which a change of circumstance would require the modification of an existing timetabling solution. Ideally of course the new solution would be as close as possible to the original solution.[7]
3.4.1 RCPSP

During my background reading, the related research area which most closely connected with the model I had in mind was the Resources Constraint Project Scheduling Problem (RCPSP). [8]

A RCPSP is a set of activities, and a set of resources. Each resource has a specified integer constant of availability. If this availability is greater than 1, the resource is said to be cumulative, otherwise it is said to be disjunctive.

Example 3-2

In a school, a teacher may be a disjunctive resource (a teacher cannot be in more than one lesson at a time) and portable whiteboards could be cumulative resources (i.e. several can be used at the same time, up to some availability limit).

Each activity has a duration, and for each available resource, a constant amount that is required of that resource.

Resource constraints dictate that for each time period, the total demand on each resource placed by activities occurring at that time must not exceed that resource’s constant of availability. Activities can be related using precedence constraints.

It has been shown that finding a feasible solution to these constraints is NP-Hard[32].

In [9] and [10] it has been shown that timetabling problems can successfully be solved as RCPSP. However many extensions are normally required to allow for more general resource and activity constraints.

I decided not to use RCPSP as a basis for my solution as there are a number of constraints that are very commonplace that are awkward to represent with resources having a fixed availability. For example, a class of students may be considered disjunctive at some times (i.e. when a compulsory course is happening) but cumulative at others (students can be scheduled to attend two optional courses at the same time).

Furthermore, as my project remit does not include generating a solution for a set of constraints, most of the research in the area would add little to my work.
3.4.2 TTL

TTL is a timetable specification language, which I have found a few examples of during the late stages of my research. However, I was unable to find an implementation of any form of tool using this language, and papers such as [29] state that extensions to the language must be used to specify a high school timetable, and further extensions still to represent the complexities of a university timetable.

It appears that the core focus of this language is to facilitate mathematical reasoning on timetable complexity and solvability.

3.5 Existing Methods of building a timetable evaluator

Although it is always possible to hand-code an evaluator for a specific timetable, there are a number of current methods that, to some extent or another, allow the declarative specification and evaluation of a good timetable.

3.5.1 Constraint (Logic) Programming

Constraint programming is an area of software research that allows the declarative description of combinatorial problems.

According to [11] “a constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain… it represents some partial information about the variables in interest.” Constraint satisfaction is the process of finding a value for each of these variables such that all constraints are satisfied.

Constraint logic programming is an extension of logic programming (such as Prolog) to include concepts of constraint programming and constraint satisfaction.

Possible Uses

Constraint programming has many applications outside of scheduling and resource planning. It has been successfully applied in the fields of natural language processing, operations research and molecular biology to name but a few.

ECL\textsuperscript{PS\textregistered}

ECL\textsuperscript{PS\textregistered} is a software system for the cost effective development of constraint programming applications [eclipse homepage]. ECL\textsuperscript{PS\textregistered} is based heavily on Prolog, but contains several solver libraries and a “high level modelling and control language”. [12]
Although ECL\textsuperscript{iPS}\textsuperscript{e} does go some way to achieving my goal of a declarative description of constraints, in reality, creating a timetable evaluator or solver with this or any other Constraint Logic Programming tool is still very heavily involved in terms of logic programming, and therefore requires specification to some extent of “how” the problem is solved. In most respects this is certainly a good thing, as it allows for complete generality, and application to any number of problems. However, for my language I wish to allow the user to escape from having to program the solution, preferring a style more similar to a query language such as SQL.

### 3.5.2 Alloy Analyzer

Alloy Analyzer is a tool developed by MIT for analysing models written in the language Alloy. Alloy is a structural modelling language based on first-order logic. [13]

I was able to take much inspiration from Alloy; it does for software modelling what I hope, in a more limited aspect, to do for timetable modelling. Indeed, were it not for a few limitations, I could have used or adapted Alloy and its analyzer for use within my project.

The key limitations that I identified with Alloy 2 were that I would not be able to deal satisfactorily with “hard” and “soft” scheduling constraints [see Section 5.2.1]. Alloy uses a set of logical constraints and reports that a model is incorrect if any of those constraints is not satisfied. It would be hard to develop software using Alloy that was able to attach priorities to constraints, and therefore I would not be able to decide that one acceptable but not perfect solution was better than another acceptable but not perfect solution.

### 3.5.3 Temporal Databases

Temporal databases are database systems which have built in time reasoning and an implementation of a temporal query language usually based on temporal logic. [14]

By storing the data of a given timetable in a pre-defined relational format, it would be possible to validate the timetable by running a series of queries which evaluate to true or false and inspect some aspect of the timetable.

When I first researched this area, I decided that although the theory was there, current implementation did not offer the facilities needed to address such problems. In fact, IC-Parc received funding in the early 90’s to develop such a system, but no such trace of this system can now be found.[33]
4 Language Requirements

In order to provide a useful function, my language must offer something over and above the current options for building a timetable evaluator.

4.1 Ease of Use

The number one requirement of my modelling language is that it should offer significant savings in development time over any other currently available method of creating a timetable evaluator. To set itself apart, the language must offer a shallow learning curve (a primary problem with many constraint based languages is that they take a great deal of expertise in order to formulate a reliable working solution). My language must allow abstractions that are closely related to the domain in which the timetable occurs.

4.2 General Application

The language should allow the user to represent environments and constraints from a wide range of situations. There should be no assumed or fixed concepts about the domain in which the timetable occurs. This includes the total period time in which the timetable operates and units in which time is measured.

4.3 Power of Expression

Although this is an area in which compromises will no doubt have to be made in order to achieve point 4.1, the modelling language must allow a wide range of constraints to be expressed. Without this the language would be of little use in all but the most simple situations.

4.4 Efficiency of Representation

The language should not only allow the user to represent a wide range of meta-models and constraints, it should allow the user to represent them in as concise a manner as possible. Although in many situations the language code may be generated by an intermediary application, it should allow for easy manual maintenance.
4.5 Compatible Representation

As this language may be generated automatically from existing data sources, commonly relational databases, it is important that the language uses syntax which has significant similarity to such data sources.

4.6 Purely Declarative Constraints

A more specific requirement which is developed from points 4.1 to 4.4 is that the user should be able to express constraints on the timetable in a purely declarative form. Much like a database query languages and software annotation systems, users should not have to specify any algorithmic details of what they require.

4.7 Efficient Implementation

The implementation of the interpreter and evaluator for my language must be able to process a problem of realistic proportions and complexity and provide evaluation quickly with only resources typically available on a personal computer.
5 Language Concepts

This chapter gives a brief outline of the concepts in my modelling language. For a more detailed discussion of each concept, complete with examples and language syntax, please see their corresponding section in chapter 6.

The timetable evaluator takes three inputs, a **meta-model**, a **set of constraints**, and the **timetable/s** to be evaluated, all of which can be specified by the modelling language.

### 5.1 The Meta-Model

The meta-model is specified by the user and describes the domain in which the timetable occurs. It describes the resources that are available and how they are organised, the type of events that can appear on a timetable and the period and granularity of the time that the timetable may cover.

#### 5.1.1 Time

- Time is discrete, and divided into a finite number of “time slots”.
- Time slots are organised by hierarchical levels.
- The user may specify the number of levels, and how many units of each level occur in the next. In doing this the user is specifying the total number of time slots available. (For example 4 sessions in a day, 5 days in a week, 10 weeks in a term = 3 levels, 200 time slots).
- Any two scheduled events which have the same time slot are said to occur concurrently.

#### 5.1.2 Resources

- Resources represent those elements of the environment in which the timetable occurs that need to be scheduled, that is their availability is in some way finite. (For example people, items, locations).
- A meta-model contains a finite set of resources.
- Resources are organised into categories, which together form a Directed Acyclic Graph (DAG), more specifically:
  - Each resource may be a member of zero or more resource categories.
  - Each resource category may in turn belong to zero or more resource categories.
  - No cycles are permitted in category membership.
• Once established, by referring to a resource category in constraints, the user is referring to all descendants of that category which are resources.

• Both resources and resource categories may have multiple named properties (for example rooms can have capacities and items can have costs).

• Properties of resource categories, both directly defined and inherited, are inherited by their members.

• As both resources and categories can belong to multiple categories, multiple inheritance is allowed, however inheriting two or more properties with the same name which were original defined in different categories creates a conflict and is not allowed.

5.1.3 Event Types

• Event types define the characteristics of particular events that may appear on a timetable.

• A meta-model contains a finite set of event types.

• Like resources, event types are organised by means of a DAG.

• Again like resources, event types may be assigned named properties, which are inherited through the graph structure from parent category to member.

• Event types may also be assigned event type constraints, which are inherited in the same fashion as properties, except constraints are not named so there is no possibility of multiple inheritance conflict.

5.2 Constraints

Constraints accompany the meta-model and describe how aspects of the meta-model must be arranged in a timetable for that timetable to be correct.

5.2.1 Hard vs. Soft Constraints

Constraints specified on a timetable fall in to one of two categories, hard and soft.

A hard constraint is a constraint that if broken for a timetable, that timetable is not a valid solution. For example, scheduling a person to be in two different locations at the same time would normally be considered a hard constraint.

A soft constraint is a constraint that it is desirable not to break, but if done so does not stop the timetable from being a valid solution; it merely harms the perceived quality of the timetable. An example soft constraint could be that workers should not be
scheduled to do anything during lunch time. A timetable that did so, all other things being equal, would not be as good as a timetable that avoided lunchtimes.

In my modelling language, all hard constraints are equally bad, that is, if a single hard constraint is broken for a given timetable, the timetable is considered to have no value.

Soft constraints however can vary in severity. Constraints must be tagged to inform the evaluator how important it is that they are not broken so that the evaluator can take this into account when assessing the quality of the timetable.

5.2.2 Event Type Constraints

- Each event type in the meta-model has zero or more associated event type constraints, as per section 5.1.3
- For every scheduled event in a timetable, all the constraints attached to its applicable event type are evaluated against it.
- Every event type constraint has a severity rating from 1 to 100.
- Those event type constraints with a rating of 100 are hard constraints.
- Those event type constraints with a rating of 1...99 are soft constraints with varying degrees of severity.

For more information on how constraint ratings are used in evaluating a timetable’s quality see Section 8.1.1.

5.2.3 Global Constraints

- Each global constraint is attached either to the timetable as a whole, in which case it is evaluated once, or to a time level, in which case it is evaluated once per unit of that time level.
- A global constraint is evaluated against the set of scheduled events that appear in the timetable during the time unit they are being evaluated for.
- A global constraint has access to all levels of the time hierarchy above the level it is attached to. For example, in a time hierarchy of weeks, days and hours, a global constraint may be attached to days. In that case it will be evaluated for each day, but may also query properties of the week that each day respectively belongs to.
5.3 Timetable

- A timetable is a set of scheduled events.
- A scheduled event has an event type, a time slot and a set of resources.

5.4 Fixed Secondary Timetables

- As an alternative to writing some time-based constraints, a fixed secondary timetable may be used in order to further constrain the primary timetable that is being evaluated.
- The fixed timetable consists of further Scheduled Events that are outside the control of the user creating the timetable and constrain the availability of some resources.
- If desirable, the secondary timetable can be assigned to a lower priority than the primary timetables, meaning that only the secondary timetable’s constraints can be broken by a clash between the two timetables.
6 Language Specification and Syntax

In this chapter I will cover two broad areas of design decisions that I have made, the design principals and semantics of the meta-model and constraints, and the syntactic design of the input language.

6.1 Basic Language Syntax

6.1.1 White Space

White space is used only to separate keywords and identifiers, extra white space is ignored.

6.1.2 Comments

Comments follow the age old C style, namely /* */ opens and closes multi-line comments, and // comments until the end of the current line. Comments have no impact on the semantics of the language and are removed before processing.

6.1.3 Identifiers

Identifiers are used to refer to previously named concepts in the model, such as Resources, Events Types and Constraint Function. They must always start with a lower case letter, and can contain alphanumeric characters and the '_' character.

6.1.4 Property References

To refer to a named property of a Resource or Event Type, a '$' character follower by an Identifier is used.

6.1.5 Keywords

All reserved keywords in the language are in all capitals.

6.1.6 Constants

Constants are also in all capitals, and cannot be a reserved keyword.
6.2 Types and Type Checking

Properties, parameters and return types for Constraint Functions are all typed, but types do not need to be declared and there is no strict type checking at interpretation time. Instead type inconsistencies will result in runtime errors during evaluation.

For a full list of available types see Appendix C – Available Types.

6.3 The Meta-Model

6.3.1 Time

Without delving into metaphysics, time in the real world has the following generally accepted properties:

- It is ordered. Given any two different instances in time, it is always possible to say whether one is before or after the other.
- Duration of time is usually measured by means of a hierarchical system of named periods. For example there are 365 days in a year, 24 hours in a day and 60 minutes in an hour.
- Time is continuous, that is for two given times it is always possible to find a third time in between by specifying ever smaller increments of measurement.

For the purposes of representing a timetable in my system, time will be considered as being discrete. **However, I have left the user the flexibility to decide the order and levels of hierarchy of time, as well as the ranges of their smaller composite units that they cover and how they relate back to “real” time.**

By specifying this at the start, a fixed finite number of named “Time Slots” are made available.

On a timetable, Scheduled Events [see Section 6.5] are assigned a single Time Slot. Any two scheduled events assigned the same Time Slot are said to happen concurrently, and therefore their use of resources may conflict, according to the necessary constraints.

**Example 6-1**

A user is creating a model for a conference timetable

- The conference takes place over two weeks
There are active conference sessions Monday to Thursday.

On each day there are four two hour conference sessions, with an hour for lunch, however the lunch time does not include any timetabled events.

The most sensible way for this user to define time would be to use three broad levels:

- **WEEK**, with a range of 2 (ie. WEEK 0 and WEEK 1).
- **DAY**, with a range of 4 (ie DAY 0 = Monday to DAY 3 = Thursday).
- **SESSION**, with a range of 4 (ie SESSION 0 to SESSION 3)

This would create $2 \times 4 \times 4 = 32$ distinct ordered time slots in which events can be scheduled.

### Specifying Time Details

To inform the evaluator how time is formatted in a meta-model, the user can use the following syntax:

```
TIME_SETTINGS {
  LEVEL: [String] START: [Int] END: [Int];
  ...
  LEVEL: [String] START: [Int] END: [Int];
}
```

Where levels are specified from most significant to least, [String] is a name for the level and the [Int]s specify the range of units of this level in the level above. **Integer constants from level names to appropriate level numbers will automatically be defined.**

### The Time Expression

To specify a Time Slot (or Slots) in my language, a “time expression” is used, which takes the form of angled braces `< >` and comma separated integer values.

Read from left to right, the integers specify the values for each time level, from highest to lowest respectively.
Two additional operators are provided, the “range” \([a..b]\) and the “all” \(*\). The range represents all values of that level between \(a\) and \(b\) inclusive, and the all represents all possible values for that level.

Although principally a numerical system, the user may declare names constants to ease the readability of a given Time Slot representation.

**Example 6-2**

Using the time levels defined in example… above.

\(<0, 1, 3>\) would represent the final session on Tuesday of the first week.

For readability, from now on we shall assume that the user has defined the constants below:

\[
\begin{align*}
\text{FIRST WEEK} &= 0; \\
\text{SECOND WEEK} &= 1; \\
\text{MONDAY} &= 0; \\
\text{TUESDAY} &= 1; \\
\text{WEDNESDAY} &= 2; \\
\text{THURSDAY} &= 3;
\end{align*}
\]

So the above timeslot could now be represented as:

\(<\text{FIRST WEEK}, \text{TUESDAY}, 3>\)

Furthermore, we can specify ranges of Time Slots as follows:

\(<\text{SECOND WEEK}, [\text{TUESDAY}..\text{WEDNESDAY}], *>\) represent all sessions on Tuesday and Wednesday in the second week, which is a total of 8 Time Slots.

### 6.3.2 Resources

Resources represent all tangible elements of the domain in which we are creating a timetable. They represent the items, people or places that we wish to synchronise to be involved with particular events at particular times.

In each meta-model, there is a finite set of resources available that must be specified and organised before the rest of the model can be created.

**Example 6-3**

If the model I wished to create represented a maintenance schedule for a (very small) railway company, I might have the following real world elements at my disposal.

- A diesel train serial number 123
• An electric train serial number 124
• An electric train serial number 125
• A hybrid diesel/electric train serial number 126
• A train mechanic Bob
• A train mechanic and general maintenance person Suzy
• A platform sweeper Jane
• A platform sweeper and general maintenance person Richard

The set of resources for my model could thus be \{ \text{train123, train124, train125, train 126, bob, suzy, jane, richard} \}

Organising Resources

A meta-model may contain groups of Resources which have similar properties, or be instances of the same type of Resource. For efficiency of representation, it would be useful to be able to refer to each of these groups without having to name each member explicitly. Therefore, each Resource available in the meta-model can be a member of zero or more Categories. Furthermore each Category can in turn be a member of zero or more different Categories.

Example 6-4

The following observations could be made about the Resources available:

• train123 to train126 are trains
• train123 and train126 are diesel trains
• train124, train125 and train126 are electric trains
• All examples of electric trains and diesel trains are also examples of trains
• bob, suzy, jane and richard are all staff
• bob and suzy are train mechanics
• suzy and richard are general maintenance people
• jane and richard are platform sweepers
• All examples of train mechanics, general maintenance people and platforms sweepers are also examples of staff
The set of categories that could therefore be inferred are:

trains { electricTrains, dieselTrains }
edlectricTrains { train124, train125, train 126 }
dieselTrains { train123, train126 }
staff { trainMechanics, generalMaintenancePeople, platformSweepers }
trainMechanics { bob, suzy }
generalMaintenancePeople { suzy, richard }
platformSweepers { jane, richard }

**Mathematical Representation of Resources**

The system of categories that I have just described lends itself to the mathematical representation of the Directed Acyclic Graph (DAG).

As per its name, a DAG is a directed graph, with no cycles. That is, for any node n in a DAG g, there is no path through g that both starts and ends on n.

A DAG maintains a hierarchy as it is always possible to topologically sort the graph, that is produce an ordering such that each node comes before all nodes it has edges to.

To map my system of categories on to a DAG I did the following:

- Each “leaf” node of the DAG (that is a node which has no edges leaving it) represents a Resource
- Each “internal” node of the DAG (that is a node which has 1 or more edges leaving it) represents a Category
- There is a single implicit root category RESOURCES to which all other Categories and Resources belong.
- RESOURCES is not a member of any other Category.

**Example 6-5**
Defining Properties for Resources

In many cases it is desirable to assign properties to Resources, for purposes of description, or for reasoning about the attributes of some particular set of Resources.

As I have already established a formal structure for the set of Resources available for a particular model, I have at my disposal several convenient methods to allow efficiency in representing these properties.

Note that to avoid complications of syntax, I have implemented a global namespace for all properties in the meta-model. This means that for all Resources and Event Types the user will be notified of a conflict if two properties are defined with the same name in two locations.

Inheritance

All descendants of each node n of my Resource DAG will inherit all properties of n. This allows groups of Resources with the same property value to be defined in one place only.

Overriding of Inherited Properties

In some cases, the property that has been inherited from an ancestor may not be applicable, or is considered to be the default case only. In these cases, a node may assign a new value to an inherited property.

- 32 -
Defining “Abstract” Properties

When reasoning about a Category of Resources, it can be important to know beforehand that all Resources in that Category have a particular property defined, despite there not being a global default for that Category. In this case the Category may define an “abstract” property, that is one which all its descendant Resources must have a value for, be it defined directly in the Resource, or inherited from some intermediate Category.

Dealing with Multiple Inheritance

As in this data structure it is possible for any node to have more than one parent, it is therefore possible for a node to inherit two conflicting values for a single property (that is values for a property that were passed down by different parents, but not defined by the same ancestor node). There are three possible solutions to this problem:

- The simplest, and the solution currently implemented is to disallow it. That is, if two property definitions conflict at some node, the Resource collection is considered invalid.

- If two property definitions conflict at some node, then that node must explicitly assign a value to that property.

- If two property definitions conflict at some node, then the value from the property with the closest definition is taken. So if node n has property x inherited from two of its parents, if one parent explicitly defines that property, and the other has inherited it from one of its parents, the value from the first parent is taken. If all inherited properties are defined at the same distance, disallow as per methods one and two.

Example

If all staff have the following information stored about them:

- Name
- Age
- Sex
- Annual Bonus

The standard annual bonus for staff is £500, except for Suzy who has an annual bonus of
Also, train mechanics all have a Registered Train Mechanic Number

Specifying Resources

All Resources are identified with the keyword “RESOURCE”, and all Resource Categories with “CATEGORY”.

NB

As mentioned earlier, it is actually the position in the rooted DAG that determines whether a particular node is a Resource or a Category (a leaf node or internal node respectively). In an earlier implementation of this language, only the keyword RESOURCE was provided. However, to allow the user to make clear whether a declared node is intended to be a Resource or a Category, the alternative keywords were introduced. Nevertheless, even in the latest interpreter implementation, the keywords remain synonymous, and no run time checking of the user’s intent is carried out.

The most basic declarations take the following form:
RESOURCE [Resource Name Identifier];

In this case a named resource has been created that does not have any properties, and does not belong to any Categories, other than the default implicit RESOURCES category.

To assign properties to a Resource, the following syntax is used:

RESOURCE [Resource Name Identifier] (property1Identifier = property1Value, property2Identifier = property2Value……);

To indicate which Categories a Resource belongs to, the IMPLEMENTS keyword is used.

RESOURCE [Resource Name Identifier] (property1Identifier = property1Value, property2Identifier = property2Value……) IMPLEMENTS [Category Identifier 1], [Category Identifier 2];

The syntax for defining Categories is identical to that for Resources, with the additional item that Categories may define abstract properties, and the alternate keyword EXTENDS in place of IMPLEMENTS.

CATEGORY [Category Name Identifier] (property1Identifier = property1Value, property2Identifier = property2Value……abstractProperty1, abstractProperty2) EXTENDS [Category Identifier 1], [Category Identifier 2];

Example 6-6

The full code listing for the Resource structure that is shown in example…

CATEGORY staff(name, age, sex, annualBonus = 500);

CATEGORY platformSweepers EXTENDS staff;
CATEGORY gMP EXTENDS staff;
CATEGORY trainMechanics(registrationNumber) EXTENDS staff;

RESOURCE jane(name = “Jane Jones”, age = 41, sex = “F”) Implements platformSweepers;

RESOURCE richard(name = “Richard Smith”, age = 30, sex = “M”) Implements platformSweepers, gMP;

RESOURCE suzy(name = “Suzy Saunders”, age = 23, sex = “F”, annualBonus = 900, registrationNumber = 123) Implements gMP, trainMechanics;

RESOURCE bob(name = “Bob Bambridge”, age = 32, sex = “M”, registrationNumber = 456) Implements trainMechanics;
6.3.3 Event Types

An Event Type is an activity that you wish to have listed on your timetable. It can potentially be used unlimited times, and given different Resources and Time Slots to form different Scheduled Events.

In terms of the meta-model, an Event Type is a collection of properties and constraints that interact and dictate that when a scheduled event of this type occurs, certain resources should be assigned to it, and possibly that other scheduled events should now occur and when.

Example 6-7

Imagine a schedule for a tennis tournament.

An example Event Type may be “Tennis Match”, which will presumably occur many times throughout the schedule (that is, many Scheduled Events will have this Event Type in the timetable).

Organising Events

As with Resources, many Events Types may represent different instances of the same general type of event. Furthermore, Events Types may very commonly share Constraints and Properties.

Therefore, I decided to use the same data structure to organise Event Types as for Resources, namely the Directed Acyclic Graph.

This data structure follows identical rules to the Resource data structure.

- Each “leaf” node of the DAG (that is a node which has no edges leaving it) represents an Event
- Each “internal” node of the DAG (that is a node which has 1 or more edges leaving it) represents a Category of Events
- There is a single implicit root Category EVENTS to which all other Categories and all Events belong.
- EVENTS is not a member of any other Category.
Example 6-8

In our tennis tournament, “Tennis Match” may be a Category of Event Types, and Event Types “Doubles Match, “Mixed Doubles Match” and “Singles Match” may all be members of this category.

Defining Properties and Constraints for Events

Defining Properties for Event Types follow the same rules of representation, namely inheritance, overriding and abstract properties [See Section 6.3.2]

Constraints for Event Types are inherited but conflicts cannot occur as constraints are not named. An Event has imposed upon it the set of Constraints that are explicitly defined for it, as well as the union of the constraints defined in all of its ancestors.

In other words, if a Constraint is defined for a Category, all concrete Event Types that descend from that Category will have that Constraint imposed upon them.

Example 6-9

In our tennis tournament, we may have five Event Types:

- Men’s Singles Matches
- Men’s Doubles Matches
- Women’s Singles Matches
- Women’s Doubles Matches
- Mixed Doubles Matches

It would be pertinent to divide these in to Categories, namely:

- Singles Matches
- Doubles Matches
- Men’s Matches
- Women’s Matches

And in turn these are all members of the Category “Matches” (there may well be other events on the timetable, such as watering the grass!)
In diagrammatic form, this would give us the following:

Dealing with constraints in English for the time being, a few very basic common constraints applicable to the event types may be:

- For every match, a court is required
- For all Women’s Matches, a female changing room attendant is required.
- For all Men’s Matches, a male changing room attendance is required.
- For all Doubles Matches, 8 line judges are required.
- For all Singles Matches, 6 line judges are required.
- For all Doubles Matches, extra water is required.

Following the rules of inheritance, assigning the constraints and properties as per this diagram may be pertinent:
NB The constraints are numbered in order to keep the diagram minimal, constraints are not numbered in the model.

In reality all matches require some line judges, the question is how many? So by defining an abstract properly “nL” in Matches, and using this in constraint number 1, later Categories of Matches can define this property to be passed down to all of their descendants. When the constraint is queried for a Scheduled Event of the type of one of these descendants, the correct nL is substituted in to the constraint.

It is clearly best practice to keep Constraint and Property definitions as “high” up the DAG as possible, as this limits the number of changes that have to be made to a model if the situation changes. This system is analogous to good Object Oriented programming techniques. If all examples that extend from a common Class have some behaviour in common, it is best to push that behaviour up the class hierarchy to the common ancestor. [18]

**Specifying Event Types**

Due to similar natures of organisation, specifying Event Types is very similar to specifying Resources [See Section 6.3.2]
The only differences are that the RESOUCE keyword has been replaced by the EVENT keyword, and that Event Type Constraints are listed between braces before the final semicolon, giving:

EVENT [Resource Name Identifier] (property1Identifier = property1Value, property2Identifier = property2Value......) IMPLEMENTS [Category Identifier 1], [Category Identifier 2] {constraint1; constraint2;......};

Example 6-10

The full listing for the Event Types in example… would be:

CATEGORY matches(n1) {
    //Requires $n1 line judges
    //Requires a court
};
CATEGORY mensMatches EXTENDS matches {
    //Requires a male changing room attendant
};
CATEGORY womesMatches EXTENDS matches {
    //Requires a female changing room attendant
};
CATEGORY singlesMatches(n1 = 6) EXTENDS matches;
CATEGORY doublesMatches(n1 = 8) EXTENDS matches {
    //Requires extra water
};
EVENT mensSingles EXTENDS singlesMatches, mensMatches;
EVENT womensSingles EXTENDS singlesMatches, womensMatches;
EVENT mixedDoubles EXTENDS doublesMatches, mensMatches, womensMatches;
EVENT mensDoubles EXTENDS doublesMatches, mensMatches;
EVENT womensDoubles EXTENDS doublesMatches, womensMatches;

6.4 Constraints

6.4.1 Event Type Constraints

An Event Type Constraint (ETC) takes the form of a predicate Constraint Expression [See Section 6.4.3] and a rating and are applied to Event Types. An ETC can be read as:

For every Scheduled Event of this Event Type that occurs in the timetable, this expression must hold.

Furthermore, if you are assessing the quality of said timetable, and the expression does not hold:
Modify the timetables perceived quality according to this rating.

For details of how the rating affects the timetable’s overall rating, see Section 8.1

**Hard or Soft Constraints – the Constraint Rating**

The constraint rating is a flexible way of specifying if a constraint is “hard” or “soft”, and if it is soft, just how soft it is.

The rating is expressed as an integer in the range of 0 to 100 inclusive.

- To specify a hard constraint, a rating of 100 is given.
- If a rating of 0 is given, whether this constraint passes or fails, it has no impact on the perceived quality of the overall timetable.
- Ratings within the range 1 to 99 inclusive are soft constraints of varying degrees.

While it is not important for the user to know the mathematics of how different ratings affect a timetable, what is important is that relative to one another, the ratings are appropriate. For example, if a constraint would only be acceptable if no other alternatives were possible, this constraint should have a much higher rating than a constraint which although not ideal, is accepted that it must occur with some frequency.

**Specifying Event Type Constraints**

For convenience, Event Type Constraints are listed along with Event Types in the meta model. For details of this please see Section 6.4.1.

The syntax for each Event Type Constraint is as follows:

```
[Constraint Expression] RATING: [Rating Int];
```

For details on Constraint Expressions please see Section 6.4.3.

**6.4.2 Global Constraints**

Global Constraints are not applicable to a specific Scheduled Event, but instead seek to analyze either the timetable as a whole, or certain subsections of it. In effect, a Global Constraint operates on sets of Scheduled Events, as opposed to single Scheduled Events.
**Organising Global Constraints**

As for Event Type Constraints, Global Constraints consist of a Constraint Expression and a Rating. Each Global Constraint is attached to one of the time levels that have been configured for this meta-model, or to the global level. The constraints attached to a time level L are then read as:

*For each time unit of level L, for the set of Scheduled Events that occur in that unit, this expression must hold.*

For the global level:

*For the set of all Scheduled Events, this expression must hold.*

Expressions attached to level L, may also refer to the set of Scheduled Events of any time unit in a level above L, as each unit of level L belongs to a unit of every level above that.

### Example 6-11

If we had the levels HOUR < DAY < WEEK

It is possible to say:

```
Global {

    // the number of occurrences of Scheduled Events of Event Type x
    // is greater than 20

}
```

This stipulates that some Event Type must be scheduled 20 times or more.

or

```
Level(DAY) {

    // the number of occurrences of Event Type x for that WEEK is zero
    // or the number of occurrences of Scheduled Events of Event Type x
    // for that DAY is strictly less than the number of occurrences of
    // Event Type x for that WEEK

}
```

This is equivalent to saying that for some Event Type, if it is scheduled in a particular week, it must be scheduled on at least two days.
Expressing Global Constraints

Global constraints are specified using the following syntax:

```
LEVEL({[Level Name or Number]}) {
    [Constraint Expression] RATING: [Rating Int];
    ...
    }
```

For details on Constraint Expressions please see Section 6.4.3.

### 6.4.3 Constraint Expressions

This section of the report introduces the single most important and most powerful element of the modelling language, Constraint Expressions.

The Constraint Expression’s purpose is to examine some aspect of the timetable that is being evaluated, detect if it meets the specification of the user, and accordingly return true or false.

#### Logical Expressions

Each constraint expression must have at its root a sentence of propositional logic. Standard logical connectives are available (with conventional precedence):

The possible operands for logical expressions are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Truth</td>
<td>TRUE</td>
</tr>
<tr>
<td>Falsity</td>
<td>FALSE</td>
</tr>
<tr>
<td>A predicate Constraint Function</td>
<td>functionName()</td>
</tr>
<tr>
<td>A Boolean Property</td>
<td>$propertyName</td>
</tr>
<tr>
<td>Predicate numerical expressions</td>
<td>23 &gt; ($property / 4)</td>
</tr>
</tbody>
</table>

- 43 -
Numerical Expressions

Numerical expressions are of a standard format, with standard operators available.

The following operands are available for numerical expressions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Example Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>float</td>
<td>6.2</td>
</tr>
<tr>
<td>A Constraint Function returning int or float</td>
<td>functionName()</td>
</tr>
<tr>
<td>A property with an int or float value</td>
<td>$propertyName</td>
</tr>
</tbody>
</table>

Set Logic Expressions

Set logic expressions are once again of standard form with standard operators (union, intersection, set difference etc), and can take as operands sets of Event Types and sets of Resources. N.B. Full set logic was a late addition to the project, and at the time of writing some elements have not been full implemented.

Precedence

For both logical and numerical expressions, precedence, when equal, is established on a left to right basis. Brackets can be used to explicitly determine precedence.

Example 6-12

```
aConstraintFunction() => ($someProperty >= 5)
```
6.4.4 Constraint Functions

Constraint Functions are the workhorse of the Constraint Expression; they are the means of accessing and processing data from the timetable being assessed.

Constraint functions fall into two distinct classes, those which operate on a single Scheduled Event and are therefore applicable as Event Type Constraints and those which operate on a set of Scheduled Events which are applicable to Global Constraints. However both classes can be used as arguments to other Constraint Functions depending on the requirements of that function.

Constraint Functions can be passed zero or more typed parameters, and return a typed output [see Section 14 for available types]. Only those Constraint Functions with Boolean outputs can be attached directly to an Event Type or time level.

Constraint Functions can also be used as input parameters for other Constraint Functions, when their return type matches the input type of another Constraint Function’s arguments.

A full list of the constraint function that I have implemented so far can be found in Appendix D – Implemented Constraint Functions but I will detail some of the more important ones here.

Commonly Used Constraint Functions

requiresOne

boolean requiresOne(ResourceSet r, boolean b) <- Scheduled Event s

This Constraint Function specifies that s must have allocated one of the Resources referred to by r that in turn passes the Constraint Expression b.

requiresOneExclusive

boolean requiresOneExclusive(ResourceSet r, boolean b) <- Scheduled Event s
Like requiresOne except this Constraint Function dictates that no other Scheduled Event that has the same Time Slot as \( s \) may have the chosen Resource allocated to it.

**Example 6-13**

\[
\text{requiresOneExclusive} \left\{ \text{rooms}, \text{$capacity > \text{$expectedNumberOfAtendees}$} \right\} \text{ RATING 100;}
\]

Specifies as a hard constraints that the Scheduled Event \( s \) it is applied to must have allocated exclusive use of a \( \text{rooms} \) Resource \( r \) such that when \( \text{$capacity > \text{$expectedNumberOfAtendees}$} \) is populated by the properties of \( r \) and \( s \), it holds.

*Note that due to the global property namespace mentioned in Section 6.3.2 I do not have to specify which properties in \( b \) belong to \( r \) and which properties belong to \( s \).*

**requiresAll**

\[
\text{boolean requiresAll} \left( \text{ResourceSet} \ r \right) \leftarrow \text{Scheduled Event} \ s
\]

Specifies that \( s \) must have allocated all Resources specified by \( r \).

**requiresAllExclusive**

\[
\text{boolean requiresAllExclusive} \left( \text{ResourceSet} \ r \right) \leftarrow \text{Scheduled Event} \ s
\]

Specifies that \( s \) must have allocated all Resources specified by \( r \) and that no other Scheduled Event with the same Time Slot as \( s \) may be allocated any of these Resources.

**countOccurrence**

\[
\text{int countOccurrence} \left( \text{int timeLevel, boolean} \ b \right) \leftarrow \text{Time Unit} \ u
\]

Returns the number of Scheduled Events that pass boolean \( b \) (which may be a Constraint Expression of type boolean) during the time level of \( \text{timeLevel} \) in which \( u \) occurs.

\( \text{countOccurrence} \) will apply \( b \) to each Scheduled Event it finds in the time level it is looking at, and each time \( b \) passes one will be added to the count.

**Example 6-14**
LEVEL(DAY) {
    countOccurrence( DAY, eventOfType({lectures}) ) <
    (countOccurrence( WEEK, eventOfType({lectures}) ) * 0.4)  
    RATING 20;
}

Specifies as a soft constraint that for every DAY, the count of those Scheduled Events that are of Event Type lectures must be strictly less than 40% of the Scheduled Events that are of Event Type lectures for that WEEK.

In other words, in a five day week, no day should have more than twice the average number of lectures for that week.

occurs

boolean occus(TimeExpression t) <- Scheduled Event s

Holds if s occurs at any of the Time Slots represented by t. The magic variable $TIME can be used in any location of the Time Expression to represent the value for that time level of s.

6.5 Timetables

A timetable is list of “Scheduled Events”.

A Scheduled Event is an instance of an Event Type [see Section 6.3.3], with an assigned set of Resources [see Section 6.3.2], and an assigned Time Slot.

The syntax for specifying a Scheduled Event is:

[Event Type Identifier] TIME: [Time Expression] RESOURCES: [Resource Identifier 1], [Resource Identifier 2]...;

Example 6-15

The event of type: “Welcome Speech”, assigned resources “all guests”, “conference director” and “main hall” happening at Time Slot <FIRST_WEEK, Monday, 0> might be expressed as:
As we have seen previously seen, we can represent more than one Time Slot with a single expression. In the case of specifying a timetable, this is just syntactic sugar for creating multiple Scheduled Events with a single expression. A single Scheduled Event can only be assigned one Time Slot.

**Example**

```plaintext
welcomeSpeech TIME: <FIRST_WEEK, [MONDAY..WEDNESDAY], 0> RESOURCES: guests, conferenceDirector, mainHall;
```

Creates three Scheduled Events, equivalent to:

```plaintext
welcomeSpeech TIME: <FIRST_WEEK, MONDAY, 0> RESOURCES: guests, conferenceDirector, mainHall;
welcomeSpeech TIME: <FIRST_WEEK, TUESDAY, 0> RESOURCES: guests, conferenceDirector, mainHall;
welcomeSpeech TIME: <FIRST_WEEK, WEDNESDAY, 0> RESOURCES: guests, conferenceDirector, mainHall;
```

**A Complete Timetable**

As all aspects of the timetable model, along with the complete suggested timetable can optionally be stored in the same text file, the user must indicate the timetable with the keyword **TIMETABLE** and enclose the list of semicolon separated Scheduled Events within braces i.e.

```plaintext
TIMETABLE {
    Scheduled Event 1;
    Scheduled Event 2;
    Scheduled Event 3;
}
```
Given the (somewhat sparse, yet very welcoming!) conference timetable for week one:

<table>
<thead>
<tr>
<th>Session</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
</tr>
</thead>
</table>
| 0       | Main Hall  
Conference Director  
Welcome Speech | Main Hall  
Conference Director  
Welcome Speech | Main Hall  
Conference Director  
Welcome Speech | Main Hall  
Conference Director  
Welcome Speech |
| 1       |         |         |           |          |
| 2       | Main Hall  
Christian Bär  
An invariant based on the Yamabe operator |         |           |          |
| 3       |         | Theatre B  
Eugene Ferapontov  
Differential geometry of nonlocal Hamiltonian operators of hydrodynamic type | | |

Depending on how the Resources and Event Types are organised the timetable may be specified as:

```java
TIMETABLE {
    // Timetable entries
}
```
6.6 Fixed Secondary Timetables

As an alternative to writing some time-based constraints, a fixed secondary timetable may be used in order to further constrain the primary timetable that is being evaluated. Such a timetable consists of Scheduled Events that are outside the control of the user creating the timetable and constrain the availability of some resources.

Example 6-16

If the coordinator of our conference knew that the Main Hall was unavailable every day for session three in the first week, they have two options to represent this.

1. Define a constraint such as

\[
\text{occurs}(<\text{FIRST\_WEEK, *, 3}>) \rightarrow (! \text{requires}([\text{mainHall}])) \text{ RATING 100;}
\]

and attach it to all Event Types

2. Create the Event Type mainHallBooked with constraint

\[
\text{requiresExclusive}([\text{mainHall}]) \text{ RATING 100;}
\]

and schedule this on a secondary timetable wherever the Main Hall is unavailable.

If desirable, the secondary timetable can be assigned to a lower priority than the primary timetables, meaning that only the secondary timetable’s constraints can be broken by a clash between the two timetables.
Example 6-17

If the coordinator was informed that a particular speaker (Prof. Smith) had preferences about which times of the day they presented papers, the coordinator would need to implement a soft constraint similar to the previous example.

To do this, they could create the Event Type lazySpeaker with the constraint

\[ \text{requiresExclusive}(\text{smith}) \text{ RATING: } 30; \]

And schedule this event accordingly on a secondary timetable

However, if a Scheduled Event of this type were placed on secondary timetables of the same level as the primary timetable, it may interfere with hard exclusivity constraints on the primary timetable. This is not the idea, therefore this secondary timetable should be assigned a lower priority than the primary timetable.
7 Interpreter Design and Implementation

The key piece of software created as part of this project is an interpreter for the modelling language.

The interpreter takes as input code which represents concepts discussed in Chapter 6 and outputs a data structure and API which can be used by a generic evaluator.

The API not only provides methods to aid evaluation, it also allows a user application to dynamically change all aspects of the meta-model, constraints or timetable.

7.1 Implementation Language and Tools

Implementation Language

All software created for this project is implemented in Java 5.0. I chose the latest edition of Java, even though I had not used it before, because I was interested in some of the new features of the language, most notably generics, which I made extensive use of [19].

Log4j

Throughout development I relied heavily on the Apache Project’s Log4j logging system [20].

Log4j offers many features which aid application logging and therefore help debugging. Firstly, it allows for the turning on and off of all logging statements at runtime, without need to modify binaries. Even in my final production implementation, if I were to spot a bug I can turn on logging via means of a XML configuration file and examine possible causes.

The Log4j API uses a hierarchical organization of loggers, typically one for each Java class, and output of each can be redirected to any number of appenders (which output the logging information in some way). Typical appenders include logging to the console, file or to a graphical user interface.

7.2 Interpreter Architecture

Writing a compiler or interpreter is one of the core practices of Computer Science, and has been repeated with such frequency that a widely accepted series of steps has been
established. My interpreter is no different, and therefore I was able to use tried and tested software design.

There are many very technical aspects to language and compiler theory [21], many of which I have had to develop an understanding of in order to write an implementation of my language. Although this is not the core focus of my project, in this chapter I shall give a brief outline of some of the work I undertook.

**Figure 7-1 – Interpreter Architecture**

### 7.2.1 Flexible Plug-in Architecture

One key aspect of my interpreter is that it can be modified easily at any point, without needing to modify or recompile the original source code.

Every key aspect of the interpreter and the meta-model representation it creates is specified by an interface, and the implementation of each interface is specified at run-time, not compile time, by an XML configuration file. Furthermore, mappings of constraint function identifiers in the language itself can be pointed to different implementations, and new Constraint Functions can be added.

Every Constraint Function that is available to the user has its own Java class, which implements the ConstraintExpression interface, typically by extending the AbstractConstraintFunction class. In the modelling language, each Constraint Function is identified by an identifier and in the interpreter’s XML configuration file, each identifier is linked to an implementing class.

**Example 7-1 – An example linking of identifier to implementing class**

```
<ConstraintFunction>
  <Name>requires</Name>
  <Class>doc.timetables.constraints.RequiresConstraint</Class>
</ConstraintFunction>
```
At runtime, the class “doc.timetables.constraints.RequiresConstraint” is loaded, and when the interpreter sees the identifier “requires” it instantiates an Object of this type and casts it to a ConstraintExpression. If the configuration specified a class which did not implement ConstraintExpression, a runtime error would result.

Therefore, if an extra Constraint Function was required to represent some new timetabling problem, the implementation of one interface and the addition of one new mapping in the configuration file would enable this new function to be fully integrated into the modelling language.

### 7.3 Lexical Analysis

The first step of my interpreter is to take the character based input, and break it down into a sequence of **lexical tokens**, which act as vocabulary symbols for a parser. The aspect of the interpreter that does this is commonly called the **lexer**.

Traditionally there are two common ways of writing a lexer, by hand, or using finite state machines, typically a deterministic finite automaton (DFA).

A DFA can recognise any language from the class of regular languages, which is any language that has input tokens that can be described by regular expressions. Therefore, DFAs are usually generated from a set of regular expressions, which makes their implementation quick and their representation efficient.

One the other hand when written by hand, although a lengthy process, a lexer can be highly efficient and specific to the input language. A hand built lexer can even use semantic analysis of the input, and is not restrained to the **regular class** of languages. Like any other human generated code, if written well, a lexer can be easy to understand, debug and modify.

The approach I took is to use ANTLR [see Appendix E – ANTLR] to interpret a grammar and build automatically a “hand written” lexer in Java. By using this approach, I felt I had the best of both worlds. The concise representation of declarative expressions, yet the Java code I was familiar with to be incorporated in my software.

To quote the ANTLR documentation, “ANTLR generates predicated-LL(k) lexers, which means that you can have semantic and syntactic predicates and use k>1 lookahead.” [22]
7.4 Parsing

Parsing is the process of analyzing a given input in order to determine its grammatical structure with respect to some formal grammar (in this case the input is a stream of lexical tokens).

As well as validating the input to confirm that it compiles with my language’s grammar, I used a parser to build an abstract syntax tree (AST), a representation of the input which mirrors the grammar’s structure in the structure of the tree.

Once again I used ANTLR to generate a parser, based on an input grammar in ANTLR’s meta-language. [23, 24]

Again, please see Appendix E – ANTLR for more details.

7.5 Intermediate Representation

After parsing, my interpreter differs from most compilers or interpreters. Although normally an intermediate representation (IR) is produced, this is normally optimized then either used for code generation, or simply executed once.

In my case, I needed to walk my abstract syntax tree with a tree parser and generate a Java data structure that could both be “executed” (once again interpreted) for a given input, and later modified by the user.

The tree parser I wrote by hand, and this tree parser performs many checks which go beyond the syntax checks performed by the parser.

A great deal of work went in to this stage of the implementation, far too much to cover in any detail here. Therefore, I shall just highlight a few details of some of the structures created by the interpreter.

7.5.1 Resources and Event Types IR

As detailed in section 6.3, Resource and Event types have very similar organisation; therefore I decided to use the same recursive data structure, namely one consisting of the Java interface DAGNode.

Each DAGNode is a uniquely named node that can have zero or more children, and can have one or more parents, with the exception of the root node which has no parents. A DAGNode contains zero or more named properties. The interface provides methods for querying and modifying all aspects of the DAGNode.
Checking for a valid structure

The tree parser must make the following validation checks over and above those performed by the parser, and report errors accordingly to the user.

- Each Resource, Event Type or equivalent Category (node) must have a unique name.
- Each node listed as a parent of another node must have been defined.
- There must not be a cycle in the DAG
- Inherited abstract properties may not go undefined in concrete Resources / Event Types (leaf nodes)
- No conflicts due to multiple property inheritance with the same name may exist.

Final Output

Once validation has been completed, the interpreter instantiates and configures all required DAGNodes, and outputs both the root node, and a HashMap of all nodes in the DAG hashed by name. This allows quick accesses to any particular point in the DAG if the name of the start node is known.

7.5.2 Constraint Expression IR

Constraint expressions are a complex part of the data structure outputted by the interpreter. Each constraint expression can be evaluated many different times during a single timetable evaluation and its variables / properties may be populated with different values for each valuation.

Therefore, Constraint Expressions are converted from an AST to another recursively defined tree, which each tree node being an instance of the ConstraintExpression interface.

External to this tree, the rest of the software is only aware of the root ConstraintExpression node. All method calls to this node must be recursively passed down to the children of each ConstraintExpression node.
Example 7-2

For the expression “2 > $aProperty”

Three ConstraintExpression nodes are created, the binary operator “greater than”, the integer literal “2” and the variable “aProperty”.

The root node of this ConstraintExpression tree is “>”, and this has the other two nodes as its children.

If the constraint expression were to be populated with the properties of a given Resource, this Resource would be passed only to “>” which in turn would be responsible for passing it on to its children.

If “>” is asked if it holds, it must first ask both of its children for their values, then use its own logic (i.e. is my left child greater than my right) to return a Boolean.

7.5.3 Timetable IR

Although essentially a timetable is a simple collection of Scheduled Events, I implemented various techniques to allow for efficient querying of the timetable.

Event Listings

The EventListing class is a collection of Scheduled Events by time level. The timetable has a reference to the global event listing, which contains all events in the timetable. Each event listing also has a hash of event listings by time unit for each time level below its own level. This continues recursively until the smallest time level is reached.
By using this extensive hashing, asking the timetable for all the events of a particular time range does not involve iterating through all Scheduled Events and seeing which match.

**Example 7-3**

If we have time levels of WEEK>DAY>HOUR and we wished to find all those Scheduled Events which occurred on during WEEK 3 at HOUR 10 on *any* DAY in that WEEK.

1. Obtain the global event listing from the timetable, which contains all Scheduled Events.
2. From this, obtain the event listing for WEEK=3, which contains all events of week 3.
3. From this, obtain the event listing HOUR=10, which contains the Scheduled Events for 10am on every day of that week.

**Used and Conflicting Resources**

When each Scheduled Event is added or removed from the timetable, the timetable keeps track of for each time slot those resources which are unused, used, or in conflict, that is are in use by more than one Scheduled Event.

**7.6 Resource / Event Type Set Reduction**

One interesting function which the interpreter performs is reduce to its minimal complete form any Resource or Event Type Sets which are listed as arguments to either Constraint Functions or as properties.

This reduction plays a crucial role in the efficient implementation of some evaluation algorithms, as detailed in Section 8.4.

Assuming we have a set of DAG nodes $s$, reduction follows two basic principals.

1. **Push Categories up the DAG** – If there is a Category $c$ in the DAG which has as concrete resource descendants *only* those nodes which are completely represented by some subset $t$ of $s$, replace $t$ with $c$ in $s$. 
2. **Add all Categories represented but not included**– If there is a Category c in the DAG which has as concrete resource descendants *only* those nodes which are completely represented by s, but c is not the descendant of any node in s, add c to s.

**Example 7-4 – DAG Node Set Reduction**

Given the DAG (without implicit root):

- The set \{f, g, h\} would be reduced to \{b\} courtesy of rule 1.
- The set \{c\} would be reduced to \{c, d, e\} courtesy of rule 2.
- The set \{e\} would remain unchanged
- The set \{a\} would be reduced to \{a, e\} courtesy of rule 2
- The set \{b, d\} would be reduced to \{a, e\} courtesy of rules 1 and 2.
Algorithm for Set Reduction

To perform set reduction, I essentially wound each node of the set down to its concrete descendants, unioned these and repeatedly pushed the resultant set back up the DAG until a fixed point is reached.

Pseudo-code algorithm:

```java
//set up data for recursive call by winding down DAG
DAGSet reduce(DAGSet s) {
    DAGSet s1;
    for each DAGNode n in s {
        s1 = s1 UNION n.getConcreteDescendants();
    }
    return reduceHelper(s1);
}

//recursive "push up" function
DAGSet reduceHelper(DAGSet s) {
    //Keep track of changes that need to be made at
    //end of inspection. Cannot be done until end to
    //respect rule 2
    DAGSet toAdd;
    DAGSet toRemove;
    boolean changeMade = false;

    //obtain a set of all ancestors of s
    DAGSet ancestors;
    for each DAGNode n in s {
        ancestors = ancestors UNION n.getAncestors();
    }

    for each DAGNode ancestor in ancestors {
        //get direct children of ancestor
        DAGSet children = ancestor.getChildren();

        //if all children of ancestor are in s
        if(children SUBSET s) {
            toAdd.add(ancestor);
            toRemove.add(children);
        }
    }
    if(changeMade) {
        //make changes
        s.add(toAdd);
        s.remove(toRemove);
        //could be further possible reductions so recurse
    }
}
```
return reduceHelper{s}
{
    
    
    //we've reached a fixed point so we're done,
    return s;
}
}
8 Evaluation Implementation

8.1 Assessing Timetable Quality – The Timetable Rating

As previously discussed one function of the evaluator is to assess the quality of a given timetable. To do this the evaluator outputs a rating, a number between 0 and 1.

- A rating of 1 identifies a perfect solution, that is, no constraints have been broken.
- A rating of 0 identifies a timetable which is not a solution, that is one or more hard constraints have been broken.
- Ratings between 0 and 1 are imperfect solutions, the closer to 1 the higher the perceived quality of the timetable.

8.1.1 Calculating the Rating – Single Valued Utility Function

Every constraint which is broken by a timetable acts as a modifier to that timetable’s overall rating. All modifiers are between 0 and 1. The overall timetable rating is achieved by multiplying all broken constraint modifiers together. Therefore, a hard constraint (rating 100) should have a modifier of 0, and a constraint with a rating of 0 should have no effect (a modifier of 1).

To calculate the modifier that a broken constraint has on a timetable from the constraint’s severity rating, a single value utility function is used:

\[
U(x) = \left(1 - \frac{x}{100}\right)^{\frac{1}{k}}
\]

Where \(x\) is the constraint’s rating and \(k\) is a constant.

The higher the value of \(k\), the greater the reduction of negative modifiers, i.e. the more the formula “ignores” minor infringements. However, whatever the value of \(k\), this formula obeys the rule that \(U(100) = 0\) and \(U(0) = 1\).

The value of \(k\) may be configured by the user. If the user is looking for a very high quality timetable with no or few soft constraints broken, then the linear equation of \(k=1\) would suffice. For example if just 10 constraints with a rate of 30 were broken, the timetable would have a rating of just 2%.
In a more realistic situation however, to avoid all possible solutions having a miniscule rating, a higher value of $k$ would be desirable.

**Figure 8-1 – Plots of $k = 1$ to $5$**

### 8.2 Complete Evaluation

The algorithm for complete evaluation of a timetable is straightforward and goes as follows.

```plaintext
rating = 1;

// Event Type Constraints
for each Scheduled Event s in Timetable {
    for each Constraint Expression e in the Event Type of s {
        if(e fails to hold for s) {
            rating = rating * getModifier(e.getRating);
        }
    }
}

// Global constraints
for each time level L in Time {
    for each Constraint Expression e in L {
        for each time unit u in L {
```
```java
if(e fails to hold for u) {
    rating = rating * getModifier(e.getRating);
}
return rating;
```

### 8.2.1 Providing Feedback

Each Constraint Expression can be asked for feedback as to why it failed. The default evaluator will just log these warnings (to be re-directed as needed by Log4j). [20] However when building a custom application, Constraint Expressions may be queried for this information as required.

### 8.3 Partial Evaluation

The evaluator also allows for partial evaluation of the timetable, which can be used, for instance, as an efficient method of comparing different modifications to a timetable. For an example of this function being used, please see Section 9.2.3.

The evaluator is capable of analysing constraint expressions to determine if their result may be affected by modification to a particular time slot. Frequently, Event Type Constraints to be evaluated outside of the time slot in question are usually not affected. Global constraints always have the potential to be affected.

### 8.4 Constraint Function Evaluation

The key reason for implementing the reduction of Resource sets as per Section 7.6, was to allow for the efficient implementation of some commonly used Constraint Functions. I will demonstrate why this is useful by detailing the algorithms used in “requires all” and “requires one”.

#### 8.4.1 Requires All

The requires all constraint takes as an argument a Resource Set we will call *required*, and is applied to a single Scheduled Event which in turn has a Resource Set of allocated resources we will refer to as *allocated*. 
The constraint’s task is to check that every concrete Resource represented by \textit{required} is contained within those concrete Resources represented by \textit{allocated}. That is to say, that the concrete Resource Set of \textit{required} is a subset of the Concrete Resource set of \textit{allocated}.

\textbf{Naïve Approach}

For now let us assume that DAG reduction has not taken place.

The only way of checking this constraint is to break each set down to the set of concrete Resources it represents, and check each Resource in \textit{required} is in \textit{allocated}.

```java
boolean requiresAll(DAGSet allocated, DAGSet required) {
    DAGSet concreteAllocated;
    DAGSet concreteRequired;
    
    //get concrete descendant set of allocated
    for each DAGNode a in allocated {
        concreteAllocated = concreteAllocated
        UNION a.getConcreteDescendants();
    }
    
    //get concrete descendant set of required
    for each DAGNode r in required {
        concreteRequired = concreteRequired
        UNION r.getConcreteDescendants();
    }
    
    //check each in concreteRequired is
    //in concreteAllocated
    for each DAGNode r in concreteRequired {
        boolean found = false;
        for each DAGNode a in concreteAllocated {
            if(a == r) {
                found = true;
                break;
            }
        }
        if(!found) {
            return false;
        }
    }
    return true;
}
```

This algorithm can prove to be very costly for all but the smallest resource DAGs.
Example 8-1

Given the resource DAG:

If:

- \( \text{required} == \{c\} \)
- \( \text{allocated} == \{b, d\} \)

Requires all should obviously hold in this case.

The resource sets would first be broken down to the concrete resource sets they represent, namely \( \{c_1\ldots c_{1000}\} \) and \( \{b_1\ldots b_{1000}, c_1\ldots c_{1000}\} \) respectively.

Assuming the average case that each concrete resource from \( \text{required} \) would be found half way through the concrete resources of \( \text{allocated} \), the algorithm would terminate after \( 1000 \times 1000 = 1 \times 10^6 \) comparison operations.

**Using Reduction**

If we now have it guaranteed that both \( \text{required} \) and \( \text{allocated} \) sets have been reduced, we can work directly on the nodes themselves and use the principal that
All concrete resources of a node \( n \) in \textit{required} are a subset of the concrete resources of \textit{allocated} if and only if either \( n \) or one of its ancestors are included in \textit{allocated}.

This allows the following complicated but efficient algorithm to be implemented:

```java
boolean requiresAll(DAGSet allocated, DAGSet required) {

    for each DAGNode r in required {

        //build set containing r and all its ancestors
        DAGSet nodeAndAncestors = r.getNodeAndAncestors();

        boolean found = false;

        //check to see if one of
        //nodeAndAncestors is in allocated
        for each DAGNode nAA in nodeAndAncestors {
            for each DAGNode a in allocated {
                if(nAA == a) {
                    found = true;
                    //found it, can stop looking
                    break;
                }
            }
            if(found == true) {
                //only needed to find one, so again
                //we can stop looking
                break;
            }
        }

        //one of required wasn’t covered by allocated
        //can terminate with false
        if(!found) {
            return false;
        }

    }

    //all of required must have been covered by
    //allocated, can return true
    return true;
}
```

This approach can lead to marked increase in efficiency.

\textbf{Example 8-2}
Assuming we have the same Resource DAG as per Example 8-1 and:

- \( \text{required} = \{c\} \)
- \( \text{allocated} = \{b1...b1000, c1...c1000\} \)

The two sets would have previously been reduced by the interpreter to:

- \( \text{required} = \{d\} \)
- \( \text{allocated} = \{a, d\} \)

The algorithm would now check if each of \( \text{required} \), namely \( d \), is either explicitly found in, or one of its ancestors is found in \( \text{allocated} \). It is, so the algorithm returns true. In total we have performed 1 comparison operation.

Why was reduction needed?

If reduction had not been performed, regardless of efficiency, the algorithm would not have returned the correct result. None of the original required set or its ancestors (\( \{c, d, a\} \)) are to be found in the original allocated set, so the algorithm would have incorrectly returned false.

### 8.4.2 Requires One

The “requires one” constraint can also make use of its Resource Set arguments being reduced before evaluation.

The principal is the same, so I will not labor the point with more examples, I will just present the naïve and reduction based algorithms.

**Naïve Approach**

```java
boolean requiresOne(DAGSet allocated, DAGSet required) {
    DAGSet concreteAllocated;
    DAGSet concreteRequired;

    //get concrete descendant set of allocated
    for each DAGNode a in allocated {
        concreteAllocated = concreteAllocated
                             UNION a.getConcreteDescendants();
    }
```
Using Reduction

boolean requiresOne(DAGSet allocated, DAGSet required) {

    for each DAGNode a in allocated {
        for each DAGNode r in required {
            if(a.getNodeAndAncestors().contains(r)) {
                return true;
            }
            if(r.getNodeAndAncestors().contains(a)) {
                return true;
            }
        }
    }
    return false;
}
9 DoC Timetable Builder

To demonstrate that the functions performed by my generic evaluator are useful, I have developed an application in the form of a Graphical User Interface (GUI), specifically designed for Imperial College Department of Computing, based on the complete example given in Appendix A - DoC Timetable Example.

The GUI was created using Java Swing[25, 26] but the implementation is outside the remit of this report, so I include no details.

All of the functionality in relation to timetables and constraints comes from my generic evaluator. The GUI is merely a custom presentation of the information and functions that the evaluator can provide.

Instead, I shall briefly walk though some of the features of this application.

9.1 Navigating the Timetable
The main display of the GUI shows the timetable, and separate section of the display details the properties, constraints and allocated Resources of any Scheduled Event that is clicked on.

### 9.1.1 Filters

At the top of the display, filters can be defined and applied by Resource or Event Type.

For example you could create a filter that shows only those Scheduled Event that utilise room 308, or you could display a timetable of just those courses attended by the first year.

### 9.2 Adding Scheduled Events

The primary purpose of the GUI is to provide assistance to someone who is manually building a timetable for the department.
If the user wishes to schedule a new event, the GUI will provide him or her with as much assistance as possible, based on the data from that part of the timetable which already exists, the meta-model and the constraints.

### 9.2.1 Selecting the Event Type

Firstly, the user must select the Event Type that s/he wishes to schedule. This is done by showing the user the Event Type DAG represented as a tree. In order to do this, some nodes will have to be displayed at several locations, once for each parent they have.

![Figure 9-2 – Selecting Event Type](image)

### 9.2.2 Adding Resources that are always required

Once the user has selected the Event Type they wish to add, they are presented with a new dialogue box which asks them to specify some more details for the event.

### Analysing Resource Constraints

At this point, on the user’s request, the evaluator will analyse the Constraints Expressions attached to this Scheduled Event and determine which Resources must always be allocated, regardless of when the event is scheduled.
Example 9-1

If an Event Type has the following Constraint Expressions attached:

//if su2 is allocated to an event, his assistant must be also requiresAll({su2}) -> requiresAllExclusive({su2_assistant});

//su2 must be allocated to this event requiresAll({su2});

//if this Event Type occurs at 9am on any day, frk is required occurs(<*,9>) -> requires({frk});

Then the evaluator would suggest that the resources \{su2, su2_assistant\} should be allocated to any Scheduled Event of this type.

Adding Additional Resources

As well as those resources the evaluator has determined compulsory, the user at this stage may allocate resources according to his preference. For example, s/he may only be interested in scheduling the event in room 308, in which case room308 would be added now. In the next stage, when the evaluator is making suggestions about time, it will only consider time slots in which room 308 is free.

9.2.3 Choosing a Time Slot

After the user has specified some of the resources s/he wants allocated to this Scheduled Event, the GUI returns to the main timetable display, and using color gradient, suggests where this event can be scheduled.

Those Time Slots during which if the event were scheduled, a hard constraint would be broken, are coloured red. Those which would break no constraints are coloured green. Those which would have some negative impact on the timetable but do not break hard constraints are coloured by severity, from red through yellow to green.
Figure 9-3 – Suggesting a Time Slot

In the example of Figure 9-3, the best times to schedule the event (in this case a new instance of Introduction to Graphics Lecture) are Tuesday, Wednesday and Friday 9am. Then in order of preference, Friday 10am and 2pm, Thursday 12pm and Monday 12pm. All other times would break some hard constraint.

Suggestion Calculations

Time Slot suggestions are calculated efficiently using the principals of partial evaluation, as detailed in Section 8.3. The evaluator “dips the toes” of the Scheduled Event in each available timeslot, and re-evaluates only those constraints which could be effected. It also orders the constraints to be evaluated in an efficient manner. Once a hard constraint has been broken for a particular Time Slot, there is no point in checking any more.
9.2.4 Finalising Resources

Once the user has chosen an appropriate time to schedule the event, s/he must now finalise the Resource allocation, allocating those Resources of which s/he has a choice (e.g. which room, only one is required).

The user may select Resources from the same tree format used to select the Event Type, but the GUI will helpfully highlight those Resources which have already been allocated for the Time Slot that has been chosen.

![Figure 9-4 Finalising Resources](image)

Once the user has selected the final resources, they can confirm the Scheduled Event and it will be added to the timetable.

9.3 Modifying, Moving and Deleting Scheduled Events

The GUI provides the facility to modify and delete existing Scheduled Events.

Modifying and moving events follows much the same pattern as adding an event, except that the initial dialog will be pre-populated with the chosen event’s details.
Deleting is the simplest operation, select an event on the timetable, click **Delete**, and it is removed.

### 9.4 Evaluating the Timetable

To evaluate all constraints on the timetable, an **Evaluate** button is provided at the bottom of the display. Once clicked, the evaluator is invoked for a full evaluation. Every Scheduled Event on the timetable is then coloured according to the combined effect the modifiers of its Event Type Constraints are having on the timetable, using the same green through yellow to red scheme mentioned previously.

By selecting an individual Scheduled Event, feedback can be obtained on a constraint by constraint basis.

![Figure 9-5 Evaluation Display](image-url)
In Figure 9-5 above, an event has been highlighted red, indicating that at least one of its hard constraints have been broken.

By clicking on the event, the “Event Info” panel displays the applicable Constraint Expressions, and highlights those which passed and failed. By selecting an individual constraint, the “Evaluation” panel at the bottom explains in English why that constraint was broken.
10 Project Evaluation

Evaluating this project presents many challenges as many of the key objectives of my language, such as ease of use, are highly subjective. Furthermore, meaningful qualitative analysis would involve many hours of third party evaluation. However, I feel I am able to present some personal insight into the effectiveness of most aspects of my work, and in Section 10.4 I have had the opportunity to conduct limited third party testing to substantiate my findings.

10.1 Ease of Use

One of the key objectives of this project was that it should be easier to create software to evaluate a timetable using my language, interpreter and generic evaluator than it would be to write such software from scratch or by adapting an existing tool.

By using third party testing in Section 10.4 I have attempted to demonstrate that at least in an absolute sense, without comparing it directly to other methods, my solution passes this test.

Possible alternatives available for writing a custom timetable evaluator include:

- Writing one in a high level imperative language such as Java / C#
- Writing one in a more declarative functional language such as Prolog [27] or a CLP language [28]
- Customising and using a different third party tool such as Alloy.

I believe that providing you can successfully represent the constraints on your timetable using my language, writing a custom timetable evaluator in a language such as Java would take considerably more effort. Furthermore, such software could potentially be quite large and therefore maintaining and modifying the code base could also be costly.

While using a methodology such as CLP offers many advantages over a high level imperative language such as Java, I suggest that it would still be more complex and time consuming than using my language.

Lastly, using an existing tool for a new purpose is always an option, and the “ease of use” factor depends largely on how familiar the user is with the original tool. I believe that some very interesting and powerful solutions could be developed with a tool like Alloy, an example of which, in a different problem domain, can be found in [13].
10.2 Expressiveness (and General Application)

It is obviously highly desirable that the modelling language should be able to represent any timetabling situation and its constraints.

To do this it is certainly necessary to have a wide range of reusable basic Constraint Functions. Currently, I have only implemented those Constraint Functions that I have found I have required for each example I have looked at. Therefore, at the moment, I cannot claim that my modelling language can represent any situation, or even a large proportion of all possible timetable situations.

I am confident however that given enough development time, my language can represent all but the most esoteric of constraints with its default implementation. In the situation that something cannot be represented, I have made it very easy to make additions or modifications to the available Constraint Functions [see Section 7.2.1].

Interestingly, I have discovered that my language can represent some things I did not even anticipate.

For example, I did not intend to be able to model timetable events of varying lengths which could potentially overlap by varying amounts. However, I have discovered that although the current implementation does not handle it particularly efficiently, it is possible to represent this with my language.

Assuming events have varying lengths in minutes, by making minutes your smallest time level and using the Constraint Function `consecutive(int n)` you can specify by Event Type how many minutes such an event should last. In the timetable itself the Scheduled Event can be specified using the `range` operator in the Time Expression.

During evaluation, each minute of the Scheduled Event will be examined for potential clashes with other Scheduled Events.

By using a combination of hard and soft constraints, you can even specify, for example, that a particular event should last 60 minutes, but has to last at least 40.

10.3 Usefulness

A secondary objective of this project was that the interpreter for my language should output data structures and an API which would facilitate the creation of custom applications.
I aimed to demonstrate that this was the case by creating the DoC timetable evaluator /
builder application [see Chapter 9] which makes extensive use of the modelling
language for all its core functionality.

In reality this application is little more than a Graphical User Interface, which chooses to
interpret and display certain elements of the model in a custom fashion. About 90% of the
application’s code relate purely to Java Swing objects, used to create the GUI. There is
almost no logic that relates to the meta-model, constraints or timetable.

I firmly believe that assuming my language can be used to model a timetabling problem,
the data structures and API that my interpreter provides would enable the easy
development of any tool that manipulates or displays this data.

10.4 Ease of Use and Efficiency of Representation – A Trial

I have always maintained that creating a timetable evaluator in my modelling language
would be both relatively simple and quick for someone with the correct background.

To help demonstrate this in a rather unscientific fashion, I devised a simple “challenge”
for some of my DoC compatriots to undertake in order to generate some feedback.

Unfortunately during this busy period it is quite hard to find Computing students who are
willing to give up several hours of their time to learn and use a new language, so I was
only able to conduct this experiment once, with my one unlucky volunteer, Vanessa Ho
Von (MEng 3).

I presented Vanessa with the problem of the DoC timetable and the English constraints as
per Section 12.1.1. I also gave her a guide to my language, which was very similar in
content to Chapter 6 of this report. Vanessa’s task was to develop a meta-model and set
of constraints to represent this problem.

To test the success of Vanessa’s work, I manually created a DoC timetable in which I
deliberately inserted several constraint violations of various subtleties. If Vanessa was
able to capture all of the constraint successfully in her meta-model, all of my deliberate
mistakes (and perhaps some more I did not do on purpose!) should be detected.

Although Vanessa was not allowed access to the test schedule while completing her work,
I did give her the unorganised lists of concrete Resources and Event Types that would be
used in the schedule. Vanessa was on her own to complete the task, her instructions were
that if she could not work out how to do something with only the documentation I
provided, that aspect was to be left incomplete.

Results

Overall, it took Vanessa just under four hours to complete the meta-model and constraints.

Her organisation of the Resources and Event Types was near identical to those I
demonstrated in Appendix A. It is most likely that the way in which I phrased the
problem and the set of Resources and Event Types I gave her lend themselves heavily to
one solution.

Vanessa managed to express most of the constraints successfully, although she did have
some difficulty with the Global Constraints, particularly the very general “course events
must be spread evenly across the timetable if possible”.

There was also some confusion with constraint ratings; Vanessa was very unsure what
ratings to attach to some of the soft constraints. In reality there is no answer to this, the
relative priorities of constraints can only be provided by the “expert” who is responsible
for creating the timetable.

Vanessa set of constraints picked up 90% of the violations I had inserted in to my
timetable. Those that were missed were due to some incorrectly formulated global
constraints.

10.5 Evaluation Efficiency

Efficiency of implementation was not a core objective of my project. However from the
outset I felt that for this project to be meaningful, I must demonstrate that a declarative
constraint based language can operate at scale in a reasonable time, even when written in
a high level interpreted language like Java.

In-depth analysis would be both highly complex and would serve little purpose, so I have
not gone much further than to demonstrate that, on a real world example, evaluation time
is perfectly acceptable.

Please note that the times quoted are for evaluation only, they do not include the loading
into memory and interpreting of the text files. The times are from the point after the meta-
model is constructed in memory, from the start of evaluation to the results being returned.
10.5.1 DoC Spring Term 2006

Firstly I conducted tests on the complete example of last term’s Department of Computing timetable. Although events were only scheduled from one term, the meta-model included all Resources and Event Types from the department of computing database.

This example has the following properties:

<table>
<thead>
<tr>
<th>Number of Resources</th>
<th>Number of Event Type</th>
<th>Number of Timeslots</th>
<th>Number of Scheduled Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>521</td>
<td>303</td>
<td>85</td>
<td>140</td>
</tr>
</tbody>
</table>

The evaluation mean time of 50 repeated evaluations was 16ms.

10.5.2 Increased number of Scheduled Events

I increased the number of Scheduled Events in the previous example by various factors, by adding all Scheduled Events on the previous timetable multiple times at random Time Slots.

<table>
<thead>
<tr>
<th>Number of Scheduled Events</th>
<th>Factor Increase in Scheduled Events</th>
<th>Average Evaluation Time</th>
<th>Factor increase in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>240</td>
<td>2x</td>
<td>102ms</td>
<td>6x</td>
</tr>
<tr>
<td>1400</td>
<td>10x</td>
<td>518ms</td>
<td>32x</td>
</tr>
<tr>
<td>7000</td>
<td>50x</td>
<td>3056ms</td>
<td>191x</td>
</tr>
</tbody>
</table>

Therefore, although there are many factors I have not considered in these tests, for a timetable with many tens of thousands of events, a considerable delay per evaluation would still be expected.

10.6 Conclusion

It is not an exaggeration to say that the wide ranging objectives for this project have proved to be one of the biggest challenges I have ever set myself, given the time available.
There are certainly aspects of my language design and implementation that I could improve upon given extra time.

A regret I have is that I have not had sufficient time to test my language thoroughly on a wide range of real world, or simulated problems. Because of this, I cannot say for sure that I have incorporated enough power in to my language to be of general use.

However, I believe that with sufficient refinement and development, the final product that I have created would offer significant value.

### 10.6.1 Further Work

I can divide potential further work in to two categories. Firstly, improvements, refinements and extensions to the work I have already attempted, and secondly a new direction for this work.

Given the time, there are many improvements I would make to the language design. For instance, ideally all types should be declared and the interpreter should do strict type checking from the outset. This is just one of the many corners I have cut from modern standard language practice in order to allow for quicker implementation. Furthermore, I would like to improve all aspects of error reporting to the user, be they for syntax errors or of a more serious nature. Identifying a user’s mistakes in an interpreter or compiler is a difficult challenge, yet given the time it is one I feel I could do much better at.

There are also many opportunities to improve the efficiency of evaluation in terms of processor time required. Although I feel I have implemented some clever algorithms that do bring efficiency from terrible to quite good, with more thought and time things could get better.

On a broader level, I think it would be interesting to see how far the assistance offered to the user in constructing a timetable can be taken. Writing a reliable automatic solver that requires no manual intervention for any given timetable problem is a task that borders on the impossible. However, I believe there is a lot that could be done to aid the human guided process.

I think for example it would be feasible to make suggestions about swapping events, perhaps even two to three levels deep.

Timetabling problems are going to continue to be a rich area of research, and hopefully, as the field advances, new possibilities for the use of languages such as the one I have detailed here will only continue to grow.
11 Bibliography


[4] JOAO, P. PEDROSO “A hybrid multi-objective/goal programming approach for timetabling” The Fifth Metaheuristics International Conference


[12] ECLiPSe, www.icpare.ic.ac.uk/eclipse


[23] PARR, T. QUONG, R W. ANTLR: A Predicated-LL(k) Parser Generator


[28] RUDOVA, H. MURRAY, K. University Course Timetabling with Soft Constraints

[29] COOPER, T B. KINGSTON J F. The Complexity of Timetable Construction Problems, Basser Department of Computer Science, University of Sydney

[30] MUNIM, M A. Genetic Algorithm For University Timetabling, Department of Computing, Imperial College London, June 2005

[31] HOWSON, P. Genetic Algorithm For Nursing Duty Rosters, Department of Computing, Imperial College London, June 2004

[33] IC-Parc, [www.icparc.ic.ac.uk/newweb/grants.htm](http://www.icparc.ic.ac.uk/newweb/grants.htm)
12 Appendix A - DoC Timetable Example

This example demonstrates my language being used to create a meta-model and set of constraints to specify the timetable for Imperial College London Department of Computing Spring term 2006.

12.1 Background

Every term, the Department of Computing must create a timetable to organise the activities of students, staff and physical resources. This is a highly complex example of the timetabling problem as the timetable must take account of the preference of many different interested parties. In addition, many students are shared across departments (e.g. JMC, ISE and MSc) so there may already be pre-existing constraints on their time.

12.1.1 Some Typical Constraints

Hard Constraints

- A student cannot be scheduled to participate in more than one compulsory course at a time.
- A staff member cannot attend more than one course at the same time.
- Lectures require exclusive use of a lecture theatre of sufficient size for the course they are part of.
- Tutorials require exclusive use of any room of sufficient size for the course they are part of.
- Lab sessions require exclusive use of a lab of sufficient size for the course they are part of.
- All students are unavailable on Wednesday from 12pm onwards.
- All students are unavailable for 1pm to 2pm every day.
- Each course event must be scheduled a specified number of times.
- Some resources (staff, lectures, rooms) may be completely unavailable at set times.
- No events can be scheduled when the department is closed, namely before 9am and after 6pm.
Soft Constraints

- Ideally no student will be scheduled for two or more optional courses at the same time.
- Ideally students should not be used from 12pm – 1pm.
- Some staff may have preferences as to when they teach.
- Tutorials should ideally follow or precede a lecture of the same course.
- The timetable should be spread as evenly across the week as possible. To specify this in more detail, we shall say that ideally no day should contain more than 30% of the events for that week.
- Whenever possible, it is preferable to schedule an event after 10am.

12.1.2 Automatic Code Generation

I wrote a small customisable application which accessed the departmental database and generated my set of resources, event types and constraints automatically.

The departmental database contains a number of tables of use. As I have to some extent achieved the requirement of allowing my language to have distinct similarities to most existing data formats, in most cases there was a one to one relationship between rows of each table and lines of my code.

Staff

The staff table contains one row for each member of staff in the department. Each row records, amongst other things, name, login, and role of that staff member. Each row of the staff table could be transformed into one line of code.

Classes

The Classes table contains one row per class of students, and again was translated directly to the associated Resources.

Courses

The Courses table contains broad details about each course on offer. Each course will become an Event Type category not an Event Type, but will be populated with some of the details of the course such as the expected number of students. The concrete Event Types will be lectures, tutorials and labs of each course.
Options

The most complicated table I used, this contains one row per course per class group that can take that course. It also lists if this is a compulsory or optional course for that class group. The details in this table were used purely to formulate some of the constraints for each course Event Type category.

12.2 Resources

12.2.1 Resource Organisation
12.2.2 Abridged Resource Listing

```plaintext
RESOURCES {
    CATEGORY staff(firstName, lastName);
    CATEGORY academic EXTENDS staff;
    CATEGORY postGraduate EXTENDS staff;
    CATEGORY research EXTENDS staff;
}```
CATEGORY administrative EXTENDS staff;
CATEGORY researchAdmin EXTENDS staff;
CATEGORY ta EXTENDS staff;
CATEGORY students;
CATEGORY rooms(capacity);
CATEGORY lectureTheaters() EXTENDS rooms;
CATEGORY tutorialRooms() EXTENDS rooms;
CATEGORY labs() EXTENDS rooms;

//some example lecture theaters
RESOURCE room308 (capacity = 150) IMPLEMENTS lectureTheaters;
RESOURCE room311 (capacity = 150) IMPLEMENTS lectureTheaters;
RESOURCE cloreLectureTheatre (capacity = 50) IMPLEMENTS lectureTheaters;

//some example tutorial rooms
RESOURCE room344 (capacity = 150) IMPLEMENTS tutorialRooms;
RESOURCE room346 (capacity = 80) IMPLEMENTS tutorialRooms;

//some example labs
RESOURCE room219 (capacity = 200) IMPLEMENTS labs;
RESOURCE room206 (capacity = 200) IMPLEMENTS labs;

//some example staff
RESOURCE ob3 (firstName = "Olav", lastName = "Beckmann") IMPLEMENTS ta;
RESOURCE psc (firstName = "Peter", lastName = "Cutler") IMPLEMENTS ta;
RESOURCE xfl (firstName = "Xiang", lastName = "Feng") IMPLEMENTS ta;
RESOURCE ih (firstName = "Ian", lastName = "Harries") IMPLEMENTS ta;
RESOURCE iwm (firstName = "Ian", lastName = "Moor") IMPLEMENTS ta;
RESOURCE latallah (firstName = "Louis", lastName = "Atallah") IMPLEMENTS research;
RESOURCE kb (firstName = "Krysia", lastName = "Broda") IMPLEMENTS academic;
RESOURCE m.brookes (firstName = "D", lastName = "Brookes") IMPLEMENTS academic;
RESOURCE elb1 (firstName = "Elliott", lastName = "Brooks") IMPLEMENTS postGraduate;
RESOURCE b.v.brophy (firstName = "B", lastName = "Brophy") IMPLEMENTS academic;

//complete student listing
CATEGORY firstYear EXTENDS students;
CATEGORY secondYear EXTENDS students;
CATEGORY thirdYear EXTENDS students;
CATEGORY fourthYear EXTENDS students;
CATEGORY fifthYear EXTENDS students;
CATEGORY computing EXTENDS students;
CATEGORY ise EXTENDS students;
CATEGORY jmc EXTENDS students;
CATEGORY a EXTENDS students;
CATEGORY b EXTENDS students;
CATEGORY e EXTENDS students;
CATEGORY y EXTENDS students;
CATEGORY v EXTENDS students;

CATEGORY c1 EXTENDS firstYear, computing;
CATEGORY c2 EXTENDS secondYear, computing;
CATEGORY c3 EXTENDS thirdYear, computing;
CATEGORY c4 EXTENDS fourthYear, computing;

CATEGORY i2 EXTENDS secondYear, ise;
CATEGORY i3 EXTENDS thirdYear, ise;
CATEGORY i4 EXTENDS fourthYear, ise;

CATEGORY j1 EXTENDS firstYear, jmc;
CATEGORY j2 EXTENDS secondYear, jmc;
CATEGORY j3 EXTENDS thirdYear, jmc;
CATEGORY j4 EXTENDS fourthYear, jmc;

CATEGORY e3 EXTENDS thirdYear, e;
CATEGORY e4 EXTENDS fourthYear, e;

CATEGORY a5 EXTENDS fifthYear, a;

CATEGORY b5 EXTENDS fifthYear, b;

CATEGORY y5 EXTENDS fifthYear, y;

CATEGORY v5 EXTENDS fifthYear, v;

RESOURCE bc1 IMPLEMENTS c1;
RESOURCE bc2 IMPLEMENTS c2;
RESOURCE bc3 IMPLEMENTS c3;
RESOURCE bise2 IMPLEMENTS i2;
RESOURCE bise3 IMPLEMENTS i3;
RESOURCE bjmc1 IMPLEMENTS j1;
RESOURCE bjmc2 IMPLEMENTS j2;
RESOURCE bjmc3 IMPLEMENTS j3;
RESOURCE mc1 IMPLEMENTS c1;
RESOURCE mc2 IMPLEMENTS c2;
RESOURCE mc3 IMPLEMENTS c3;
RESOURCE mc4 IMPLEMENTS c4;
RESOURCE mcai1 IMPLEMENTS c1;
RESOURCE mcai2 IMPLEMENTS c2;
RESOURCE mcai3 IMPLEMENTS c3;
RESOURCE mcai4 IMPLEMENTS c4;
RESOURCE mccm1 IMPLEMENTS c1;
RESOURCE mccm2 IMPLEMENTS c2;
RESOURCE mccm3 IMPLEMENTS c3;
RESOURCE mccm4 IMPLEMENTS c4;
RESOURCE mcep1 IMPLEMENTS c1;
RESOURCE mcep2 IMPLEMENTS c2;
RESOURCE mcep3 IMPLEMENTS c3;
12.3 Event Types and Event Type Constraints

12.3.1 Event Type Organisation
12.3.2 Constraint Formulation

Each of the hard and soft constraints that translate to an Event Type Constraint were created as follows:

<table>
<thead>
<tr>
<th>English Constraint</th>
<th>Translation</th>
</tr>
</thead>
</table>
| A student cannot be scheduled to participate in more than one compulsory course at a time. | Every course Event Type that is compulsory for some set of students S will have the ETC: 

requiresAllExclusive({S}) RATING 100;

Which dictates as a hard constraint that S must be an allocated resource for this Event Type, and cannot be allocated to any other Scheduled Event at the same time. |
| Ideally no student will be scheduled for two or more optional courses at the same time. | Every course Event Type that is optional for some set of students S will have the ETCs: 

requiresAll({S}) RATING 100; |
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A staff member cannot attend more than one course at the same time.</td>
<td>Every course Event Type that requires staff members S1…Sn will be given the constraint: requiresAllExclusive({S1…Sn}) RATING: 100; If possible, this constraint may be merged with some of the other hard requiresAllExclusive constraints such as those for students.</td>
</tr>
<tr>
<td>Lectures require exclusive use of a lecture theatre of sufficient size for the course they are part of.</td>
<td>Every Event Type which is a lecture will have the constraint: requiresOneExclusive({lectureTheaters}), $capacity &gt;= $expectedNumberOfStudents) RATING: 100;</td>
</tr>
<tr>
<td>Tutorials require exclusive use of any room of sufficient size for the course they are part of.</td>
<td>Every Event Type which is a tutorial will have the constraint: requiresOneExclusive({rooms}), $capacity &gt;= $expectedNumberOfStudents) RATING: 100;</td>
</tr>
<tr>
<td>Lab sessions require exclusive use of a lab of sufficient size for the course they are part of.</td>
<td>Every Event Type which is a lab session will have the constraint: requiresOneExclusive({labs}), $capacity &gt;= $expectedNumberOfStudents) RATING: 100;</td>
</tr>
<tr>
<td>Tutorials should ideally follow or precede a lecture of the same course.</td>
<td>Each Event Type of type tutorial will have the constraint of the form: occurs(&lt;$TIME, $TIME+1 &gt;, eventOfType({&quot;lecture of same course&quot;}))</td>
</tr>
</tbody>
</table>
Which dictates as a soft constraint that every tutorial must either occur one hour after or one hour before the lecture of the same course.

12.3.3 Abridged Event Type Listing

```
EVENTS {

  CATEGORY courses(title, expectedNumberOfStudents, optionalRating = 40);

  CATEGORY lectures EXTENDS courses {
    requiresOneExclusive({lectureTheaters}, $capacity >=
    $expectedNumberOfStudents) RATING: 100;
  };

  CATEGORY tutorials EXTENDS courses {
    requiresOneExclusive({rooms}, $capacity >=
    $expectedNumberOfStudents) RATING: 100;
  };

  CATEGORY labSessions EXTENDS courses {
    requiresOneExclusive({labs}, $capacity >=
    $expectedNumberOfStudents) RATING: 100;
  };

  CATEGORY course355(expectedNumberOfStudents = 111, title = "Organisations and Management Processes") IMPLEMENTS courses {
    requiresAllExclusive({fjw, bc3, mc3, mcai3, mccm3, mcep3, mcse3})
    RATING: 100;
  };
  EVENT course355Tutorial IMPLEMENTS course355, tutorials{
  };
  EVENT course355Lab IMPLEMENTS course355, labSessions{
  };
  EVENT course355Lecture IMPLEMENTS course355, lectures{
  };

  CATEGORY course222(expectedNumberOfStudents = 120, title = "Software Engineering - Design II") IMPLEMENTS courses {
    requiresAllExclusive({frk, bc2, mc2, mcai2, mccm2, mcep2, mcse2})
    RATING: 100;
    requires({bjmc2, mjmc2}) RATING: 100;
    requiresAllExclusive({bjmc2, mjmc2}) RATING: $optionalRating;
  };
  EVENT course222Tutorial IMPLEMENTS course222, tutorials{
```
EVENT course222Lab IMPLEMENTS course222, labSessions{
};
EVENT course222Lecture IMPLEMENTS course222, lectures{
};

CATEGORY course130(expectedNumberOfStudents = 150, title = "Databases I")
IMPLEMENTS courses {
    requiresAllExclusive({jj, bc1, mcl, mcai1, mccm1, mcep1, mcse1})
    RATING: 100;
};
EVENT course130Tutorial IMPLEMENTS course130, tutorials{
};
EVENT course130Lab IMPLEMENTS course130, labSessions{
};
EVENT course130Lecture IMPLEMENTS course130, lectures{
};

CATEGORY course352(expectedNumberOfStudents = 35, title = "Humanities")
IMPLEMENTS courses {
    requiresAllExclusive({mcep3}) RATING: 100;
    requires({bc3, bjmc3, mc3, mccm3, mcse3, mjmc3}) RATING: 100;
    requiresAllExclusive({bc3, bjmc3, mc3, mccm3, mcse3, mjmc3})
    RATING: $optionalRating;
};
EVENT course352Tutorial IMPLEMENTS course352, tutorials{
};
EVENT course352Lab IMPLEMENTS course352, labSessions{
};
EVENT course352Lecture IMPLEMENTS course352, lectures{
};

CATEGORY courseS4.17(expectedNumberOfStudents = 1, title = "Speech Processing")
IMPLEMENTS courses {
    requiresAllExclusive({m.brookes}) RATING: 100;
    requires(mise4) RATING: 100;
    requiresAllExclusive({mise4}) RATING: $optionalRating;
};
EVENT courseS4.17Tutorial IMPLEMENTS courseS4.17, tutorials{
};
EVENT courseS4.17Lab IMPLEMENTS courseS4.17, labSessions{
};
EVENT courseS4.17Lecture IMPLEMENTS courseS4.17, lectures{
};

CATEGORY course261(expectedNumberOfStudents = 114, title = "Laboratory 2")
IMPLEMENTS courses {
    requiresAllExclusive({bc2, mc2, mcai2, mccm2, mcep2, mcse2})
    RATING: 100;
};
EVENT course261Tutorial IMPLEMENTS course261, tutorials{
12.4 Global Constraints

12.4.1 Constraint Formulation

Each of the hard and soft constraints that translate to a Global Constraint were created as follows:

<table>
<thead>
<tr>
<th>English Constraint</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each course event must be scheduled a specified number of times.</td>
<td>Attached to the global level, for Event Type E that need to be scheduled n times:</td>
</tr>
<tr>
<td></td>
<td>countOccurrence (GLOBAL, eventOfType ({E})) == n RATING: 100;</td>
</tr>
<tr>
<td>The timetable should be spread as evenly across the week as possible. To specify</td>
<td>A single soft constraint will be attached to the time level.DAY:</td>
</tr>
<tr>
<td>this in more detail, we shall say that ideally no day should contain more than</td>
<td>(countOccurrence (DAY, eventOfType ({courses}))) / (countOccurrence (GLOBAL, eventOfType ({courses}))) &lt; 0.3 RATING: 80;</td>
</tr>
<tr>
<td>30% of the events for that week.</td>
<td></td>
</tr>
</tbody>
</table>

12.4.2 Abridged Global Constraint Listing

GLOBAL_CONSTRAINTS { LEVEL (GLOBAL) { countOccurrence (GLOBAL, eventOfType ({course261Lecture})) = 2 RATING: 100; countOccurrence (GLOBAL, eventOfType ({course261Tutorial})) = 1 RATING: 100; ... }
12.5 Fixed Timetable

Although the following constraints could all be expressed in some form of Event Type Constraint or Global Constraint, I instead decided to use a fixed timetable of a lower priority level as per Section 6.6 which provides a much neater representation of the time constraints attached to some resources.

12.6 Fixed Timetable Formulation

<table>
<thead>
<tr>
<th>English Constraint</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>All students are unavailable for 1pm to 2pm every day.</td>
<td>Event Type <em>wednesdayActivities</em> is declared with the ETC:</td>
</tr>
<tr>
<td></td>
<td>\texttt{requiresAllExclusive({students}) RATING: 100;}</td>
</tr>
<tr>
<td></td>
<td>and scheduled at appropriate times on the fixed timetable, i.e.</td>
</tr>
<tr>
<td></td>
<td>\texttt{wednesdayActivities TIME: \texttt{&lt;WEDNESDAY, [12..17]&gt;}}</td>
</tr>
<tr>
<td></td>
<td>\texttt{RESOURCES: students;}</td>
</tr>
<tr>
<td>All students are unavailable on Wednesday</td>
<td>Event Type <em>lunchTimeHard</em> is declared with the ETC:</td>
</tr>
<tr>
<td></td>
<td>\texttt{requiresAllExclusive({students}) RATING: 100;}</td>
</tr>
</tbody>
</table>
from 12pm onwards. and scheduled at appropriate times on the fixed timetable, i.e.

<table>
<thead>
<tr>
<th>lunchTimeHard</th>
<th>TIME: &lt;[MONDAY..TUESDAY], 13&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCES: students;</td>
</tr>
<tr>
<td>lunchTimeHard</td>
<td>TIME: &lt;[THURSDAY..FRIDAY], 13&gt;</td>
</tr>
<tr>
<td></td>
<td>RESOURCES: students;</td>
</tr>
</tbody>
</table>

Ideally students should not be used from 12pm – 1pm.

Event Type *lunchTimeSoft* is declared with the ETC:

<table>
<thead>
<tr>
<th>lunchTimeSoft</th>
<th>TIME: &lt;[MONDAY..TUESDAY], 12&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCES: students;</td>
</tr>
<tr>
<td>lunchTimeSoft</td>
<td>TIME: &lt;[THURSDAY..FRIDAY], 12&gt;</td>
</tr>
<tr>
<td></td>
<td>RESOURCES: students;</td>
</tr>
</tbody>
</table>

No events can be scheduled when the department is closed, namely before 9am and after 6pm.

Event Type *departmentClosed* is declared with the ETC:

<table>
<thead>
<tr>
<th>departmentClosed</th>
<th>TIME &lt;*, [0..8]&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCES: rooms;</td>
</tr>
</tbody>
</table>

Some staff may have preferences as to when they teach.

In a similar fashion, Event Types with appropriate ETCs should be created to reflect who is unavailable and to what extent (how hard or soft the constraint is). These can then be scheduled at appropriate times.

Whenever possible, it is preferable to schedule an event after 10pm.

Event Type *everyoneLikesALieIn* is declared with the ETC:

<table>
<thead>
<tr>
<th>everyoneLikesALieIn</th>
<th>RATING: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RESOURCES: students, staff;</td>
</tr>
</tbody>
</table>
**12.7 Fixed Timetable Listing**

EVENTS {

    CATEGORY fixedEvents(title);

    EVENT wednesdayActivities(title = "Wednesday Activities")
    EXTENDS fixedEvents {
        requiresExclusive({students}) RATING: 100;
    };

    EVENT lunchTimeSoft(title = "Lunch Time") EXTENDS fixedEvents {
        requiresExclusive({students}) RATING: 50;
    };

    EVENT lunchTimeHard(title = "Lunch Time") EXTENDS fixedEvents {
        requiresExclusive({students}) RATING: 100;
    };

    EVENT everyoneLikesALieIn (title = "Lie In") EXTENDS fixedEvents {
        requiresExclusive({staff, students}) RATING: 10;
    };
}

TIMETABLE {
    departmentClosed TIME <*, [0..8]> RESOURCES: rooms;
    everyoneLikesALieIn TIME: <*, 9> RESOURCES: staff;
    wednesdayActivities TIME: <2, [12..17]> RESOURCES: students;
    lunchTimeSoft TIME: <$[0..1]$, 12> RESOURCES: students;
    lunchTimeSoft TIME: <$[3..4]$, 12> RESOURCES: students;
    lunchTimeHard TIME: <$[0..1]$, 13> RESOURCES: students;
    lunchTimeHard TIME: <$[3..4]$, 13> RESOURCES: students;
}
In order to evaluate the expressiveness of my language, I specified an example schedule for the first week of the Wimbledon Tennis tournament.

The core example adds little more to the understanding of my language than the previous DoC example, so instead of listing it fully I shall instead highlight a few of the more interesting constraints and how I was able to express them.

The key Resources available for the schedule were courts, players, umpires, line judges and ground staff.

The schedule was set over a 2 WEEK, 7 DAY, 24 HOUR time setting.

Players have the properties “name” and “nationality” and are organised into the following categories.

- Male
- Female
- Seed
- tvPriority
- singles Participants
- doubles Participants
- mixedDoublesParticipants

Umpires also had the property of nationality.

The Event Types DAG contains suitably organised Event Types such as mensSinglesMatch, mixedDoublesMatch, groundMaintenence etc.

All core constraints for organising Resource availability are as expected, e.g. all Event Types in the “matches” category require the correct number of the correct type of players, a court, the appropriate number line judges etc.
### Sample Interesting Constraints

<table>
<thead>
<tr>
<th>English Constraint</th>
<th>Constraint Expression</th>
</tr>
</thead>
</table>
| If a seeded or tvPriority player is involved in a match, they must be scheduled on one centre court or court number one. | ETC for matches<br>
  \[
  \text{requires}((\text{seeded, tvPriority})) \rightarrow \text{requiresOne}((\text{court1, centreCourt}));
  \] |
| At every point of the day there must be at least three courts free, in case a primary court is put out of use. | Global Constraint for time level HOUR<br>
  \[
  \text{countOccurance}((\text{requires}((\text{court})))) < \text{TOTAL_COURTS} - 3.
  \] |
| Every match requires an umpire, but that umpire must not be of the same nationality as any of the players. | ETC for matches<br>
  \[
  \text{requiresOneExclusive}(
    \{\text{umpire}, \}
    ! \text{EMPTY} (\{$playerNationality$} \text{INTERSECT propertySet (}
      "\$umpireNationality",
      \text{resourcesSet}\{\text{player}\}
    ));
  \] |
14 Appendix C – Available Types

14.1.1 Boolean

The modelling language has an explicit Boolean type, which can be expressed with the constants TRUE and FALSE.

14.1.2 Integer / Float

As there is currently no integer arithmetic implemented in this language, there is little point in distinguishing between the two types. However, I have implemented it in the lexer / parser, so the facility is there to distinguish between the two.

A number expressed with no decimal point is interpreted as an integer, a number with a decimal point is interpreted as a float.

14.1.3 String

Strings are identified by being enclosed in quotes (“”). To escape a literal quote within a string, consecutive quotes can be used (“”).

14.1.4 Resource Set / Event Type Set

A Resource Set or Event Type Set is expressed as a set of nodes from the respective DAG, be they Categories or Concrete Resources / Event Types. The nodes identifiers are listed between braces “{“ and “}”.

Regardless of what nodes are selected, the semantics of a Resource or Event Type Set dictate that they represent the union of:

- All concrete nodes from the specified set.
- The concrete descendants of all Category nodes in the specified set.

Therefore, semantically the Resource Set {rooms} and the Resource Set which listed all rooms {room1, room2…. room n} are the same.

14.1.5 General Sets

String and ints can also be included in sets, which count as two new types in their own right. Mixed sets and comparisons between sets of different types are not allowed and will create a type error. Sets can be expressed, as {“string1”, “string2”….}
14.1.6 Time Expression

A time expression represents one or more Time Slots. It is expressed between angles braces ‘<’ and ‘>’ and contains a list of comma separated integer expressions that represent the unit number of each time level from most to least significant. Unit numbers may also be expressed as ranges [a..b] and “all units” “*”.

When used in the context of a Scheduled Event, the magic variable $TIME may be used, which is populated at run time with the unit value for the correct location it is used.

e.g. the Time Expression <$TIME, $TIME+1> for the time levels WEEK>DAY represents the same week, one hour later than the time of a scheduled event.
15 Appendix D – Implemented Constraint Functions

requiresOne

boolean requiresOne(ResourceSet r, boolean b) <- Scheduled Event s

This Constraint Function specifies that s must have allocated one of the Resources referred to by r that in turn passes the Constraint Expression b.

requiresOneExclusive

boolean requiresOneExclusive(ResourceSet r, boolean b) <- Scheduled Event s

Like requiresOne except this Constraint Function dictates that no other Scheduled Event that has the same Time Slot as s may have the chosen Resource allocated to it.

requiresN, requiresNExclusive

boolean requiresN (int count, ResourceSet r, boolean b) <- Scheduled Event s

boolean requiresNExclusive (int count, ResourceSet r, boolean b) <- Scheduled Event s

Like requiresOne and requiresOne exclusive except larger amounts of the required resources can be specified.

requiresAll

boolean requiresAll(ResourceSet r) <- Scheduled Event s

Specifies that s must have allocated all Resources specified by r.

requiresAllExclusive

boolean requiresAllExclusive(ResourceSet r) <- Scheduled Event s
Specifies that \( s \) must have allocated all Resources specified by \( r \) and that no other Scheduled Event with the same Time Slot as \( s \) may be allocated any of these Resources.

**countOccurrence**

\[
\text{int countOccurrence(int timeLevel, boolean b) <- Time Unit u}
\]

Returns the number of Scheduled Events that pass boolean \( b \) (which may be a Constraint Expression of type boolean) during the time level of \( timeLevel \) in which \( u \) occurs.

\( \text{countOccurrence} \) will apply \( b \) to each Scheduled Event it finds in the time level it is looking at, and each time \( b \) passes one will be added to the count.

**occurs**

\[
\text{boolean occurs(TimeExpression t) <- Scheduled Event s}
\]

Holds if \( s \) occurs at any of the Time Slots represented by \( t \). The magic variable $TIME can be used in any location of the Time Expression to represent the value for that time level of \( s \).

**eventOfType**

\[
\text{boolean eventOfType(EventTypeSet e) <- Scheduled Event s}
\]

Holds if \( s \) has as an Event Type any of the Event Types represented by \( e \).

**propertySet**

\[
\text{Set propertySet(String propertyName, ResourceSet or EventTypeSet s)}
\]

Returns the Set of appropriate type of all values of the specified property found within the set of Event Type or Resources \( s \). If applied to a Scheduled Event this has no impact on the result.
resourceSet

ResourceSet resourceSet(ResourceSet r) Scheduled Event s

When applied to a Scheduled Event s, resourceSet returns the set of resources allocated to that event that are represented by the ResourceSet r.
16 Appendix E – ANTLR

16.1 What is ANTLR?
ANTLR is a state of the art lexer/parser generator developed by Terence Parr at the University of San Francisco. [23]

In case you are curious, ANTLR stands for “ANother Tool for Language Recognition”.

16.2 Why use ANTLR?
ANTLR will generate both a lexer and a parser from grammatical descriptions of the language, and allows for easy maintenance and extension of these important components compared to writing them by hand.

However, that being said, ANTLR has a very steep learning curve and it took me a significant amount of time to get to grips with.

By using ANTLR for all but the simplest of languages, you are immediately faced with very complicated concepts of parsing theory, such as linear approximate look ahead [23]

16.3 Specifying the Lexer
A lexer in ANTLR is specified by means of a series of rules, which take the standard form of:

Rule : (regular expression) | (regular expression)

Where regular expressions separated by | are seen as alternatives.

Example 16-1 – A sample lexical rule
My definition of the token INT (unsurprisingly representing an integer) is:

```plaintext
INT : ('0'..'9')+;
```

This specifies that an INT is represented by a sequence of one or more characters from the range of ‘0’ through ‘9’.

In turn, ANTLR takes this rule and generates Java code, much more complicated, but
Some of the more complicated lexer rules I had to specify include:

**Whitespace**

The rule for whitespace must be designed to digest spaces, tabs, form feeds or “end of lines”. A complication is that there are three different formats for specifying the end of a line, Unix, with a “newline” character, DOS, with a “newline, carriage return” combination, and Macintosh, with a single “carriage return”.

In addition, for my whitespace rules I am able to communicate some extra information to the receiver of the lexical tokens (usually a parser), namely that the line number for error reporting should be incremented, and that all whitespace can be ignored (i.e. the token type is set to SKIP).

**Range or Integer – Syntactic Predicate Lookahead**

Because it is impossible to specify in a “context free” grammar whether a given numerical character forms the start of a RANGE (1..2) an INT (5) or a REAL (2.32), I needed to use syntactic predicates to look ahead of the current scanning position to determine the type of the token I was about to pass on.

**16.4 Specifying the Parser**

ANTLR’s meta-language allows a parser to be specified by a series of rules that correspond closely with EBNF grammar e.g.
Expression Parser Rules

The most complicated parts of the parser to specify are the rules for expression handling. My rules for numerical and logical expressions (without set logic) are as follows:

```
expression   : or_expr ( IMPLIES ^ or_expr )*;
or_expr   : and_expr ( OR ^ and_expr )* ;
and_expr     : not_expr ( AND ^ not_expr ) * ;
not_expr : NOT ^ not_expr
     | expr ( ( NOT_EQUALS ^ | LTE ^ | LT ^ | GTE ^ | GT ^ | EQUALS ^) expr )? ;
expr : MINUS ^ term
     | term (( PLUS ^ | MINUS ^ ) term )* ;
term : factor ( (TIMES ^ | DIV ^) factor)* ;
factor : STRING_LITERAL | INT | REAL | VARIABLE | function | "TRUE" | "FALSE"
     | LPAREN !  expression  RPAREN !;
```

As well as producing a data structure with all of the information required for interpretation, the parser which I generated does most of the work of disambiguating expressions, and outputs an AST which represents the correctly formed expression tree including precedence and ordering. The method of parsing used to do this is LL(2), which is a top down parser with a look ahead of two.

**Example 16-2**

Given the expression 2 + 5 * 6, my lexer would output to my parser the stream of tokens:

```
[(INT=2), (PLUS), (INT=5), (TIMES), (INT=6)]
```

And using the (simplified) parser rules:

```
expr : term (( PLUS ^ | MINUS ^ ) term )* ;
term : factor ( (TIMES ^ | DIV ^) factor)* ;
factor : INT | LPAREN !  expression  RPAREN !;
```

In which ^ indicates that the preceding token should be the root in the AST of that rule, and ! indicates that the preceding token should not be included in the AST.

The parser would output the AST:
Meaning the expression should be interpreted as the result of adding together the result of “2” and the result of “5 * 6”.

However, if the expression were (2 + 5) * 6, and therefore the lexical token stream was:

```
[(LPAREN), (INT=2), (PLUS), (INT=5), (RPAREN), (TIMES), (INT=6)]
```

The same rules would produce:

Meaning the expression should be interpreted as the result of multiplying together the result of “6” and the result of “2+5”.