SDU
UNIVERSITY OF
SOUTHERN DENMARK

Department of Mathematics
and Computer Science (IMADA)

Master's Thesis

Modeling Timetabling Problems in Constraint Programming

Author:
Brian Alberg

Supervisors:
Marco Chiarandini
Jacopo Mauro

June 1, 2019

Project Description

Timetabling is the activity of scheduling events to happen at a particular time. It is an important and active area of research with a wide range of applications in both the academic world and the industry. Timetabling problems are however often non-trivial, and in Computer Science they are NP-hard. Large instances of timetabling problems, like those found at large universities, often contain hundreds of events involving courses, rooms and teachers, and thousands of students, all of which impose certain constraints on the events. This makes the problem of timetabling extremely difficult to solve by hand.

Constraint programming (CP) revolves around two central aspects, namely declarative modelling languages and solvers, including local search and exhaustive methods. Modelling languages are designed to abstract the problem from the solver, and MiniZinc (Nethercote et al., 2007) does this by providing translation to a low-level intermediate solver-input language called FlatZinc. This provides a natural approach to modelling of real-world problems and makes it easy to adapt the model to new situations, while the FlatZinc language makes it possible to solve the same model with a wide variety of FlatZinc-compatible solvers and compare the performance of different solvers.

The aim of this project is to investigate how educational timetabling problems can be modelled in constraint programming and how efficient different types of constraint programming solvers are at finding feasible or optimal solutions to these problems.

The focus is on two different timetabling problems. The IMADA timetabling problem is the timetabling problem of scheduling the elective courses offered by the Department for Mathematics and Computer Science at the University of Southern Denmark. The ITC2019 timetabling problem is a timetabling problem presented by the International Timetabling Competition 2019 (Müller, Rudová, and Müllerová, 2018) where timetabling data from several universities around the world is provided in a uniform format. The two models differ and we will discuss the differences.

In the course of the project I will:

- Review literature on timetabling,
- learn the MiniZinc and FlatZinc modelling languages,
- model the IMADA timetabling problem and the ITC 2019 timetabling problem and discuss their relationship,

- implement the models in MiniZinc or FlatZinc,
- carry out experimental tests on the models using a variety of different solvers and analyse the results,
- sum up the insight gained from the previous points,
- write a report using appropriate scientific language of the field.

Abstract

Timetabling is the activity to schedule events to happen at a particular time. It is an important problem that has to be resolved in both the academic world and industry. Previous efforts on the research of timetabling problems for universities have often focused on single-week timetabling problems. In this thesis, we argue that these do not accommodate the complexity of real-world university timetabling problems. Instead, we present two formulations, a *compact* and an *extended* formulation, for modeling of complex full-semester timetabling problems, along with models in the MiniZinc language and a method for generating FlatZinc models directly from instance files. By presenting how a problem in terms of the compact formulation can be encoded into a problem in the extended formulation, we show the important differences of the formulations, and argue that since timetabling problems differ, different types of models are needed depending on the problem. Finally, we found that solvers based on local search techniques might have an advantage in solving the presented flexible timetabling problems, but also that constraint programming solvers are generally not effective enough to solve these complex problems.

Acknowledgements

I would like to thank a number of people for supporting me during writing of this thesis. First I would like to acknowledge and thank Asc. Prof. Marco Chiarandini and Asc. Prof. Jacopo Mauro for always finding time in their busy schedules for discussions and guidance, and for pushing me forward. I would like to thank my wife Nikita, my daughter Aya and the rest of my family for bearing with me and my late work hours. Finally, a big thanks to the people of the SDU eScience Center, especially Dan Thrane, Jonas Hinchely and Henrik Schulz for their support and our daily lunches, Markus Lund for being there when I needed a day off, and the people of IMADA for making it such a nice place to learn.

Contents

1 Introduction

Timetabling is the activity of scheduling events to happen at a particular time. It is an important and active area of research with a wide range of applications in both the academic world and the industry. Timetabling problems are often nontrivial, and in Computer Science they are NP-hard (Burke et al., 2010). Large instances of timetabling problems, like those found at large universities, often contain hundreds of events involving courses, rooms and teachers, and thousands of students, all of which impose certain constraints on the events. This makes the problem of timetabling extremely difficult to solve by hand.

Bettinelli et al. (2015) gave an overview of *curriculum-based course timetabling* (CB-CTT) and defined the problem in mathematical terms, along with possible variations and extensions to the problem. A wide range of formulations and techniques published since the 90s were presented, reviewed and benchmarked on different timetabling problems, but ultimately it was not possible to determine any "best" method. It was argued that for timetabling problems in a university context, the problem differs slightly depending on the type of university, and these problems might deviate even more from timetabling problems in industry. As an example, some universities might be divided into multiple departments at different locations, in which case the travel distance between locations has to be included as a part of the problem formulation, while at single-building universities, this is usually not a problem. It was also argued that further research is needed on the topic of CB-CTT, and especially on which techniques excel depending on the type of the timetabling problem.

Müller, Rudová, and Müllerová (2018) gave an outline of existing research on university course timetabling and to encourage further research in the area of educational timetabling, the International Timetabling Competition 2019 (ITC2019) was introduced. A detailed definition of a generalized but flexible timetabling problem was given, along with a presentation of an XML format for specifying timetabling problems in a uniform format. Anonymized timetabling instances of various size and complexity, from universities around the world was made public to researchers who was invited to find solutions to the provided instances. Prizes will

be given to the groups with the best overall solutions, or lower bounds, found for the provided instances in different categories.

Constraint programming (CP) revolves around two central aspects, namely declarative modeling languages and solvers, including local search and exhaustive methods. Modeling languages are designed to abstract the problem from the solver, such that the same solver can be used for different models. Therefore, many solvers include their own modeling language, however this complicates things if multiple solvers are wanted for a project, as the researcher will have to learn a modeling language for each solver.

In Nethercote et al. (2007) MiniZinc was presented as a simple but expressive solver-independent modeling language. MiniZinc makes it possible to abstract the model from the solver by providing translation to a low-level intermediate solver-input language called FlatZinc. This provides a natural approach to modeling of real-world problems in MiniZinc and makes it easy to adapt the model to new situations, while the FlatZinc language makes it possible to solve the same model with a wide variety of FlatZinc-compatible solvers and compare their performance.

MiniZinc supports several features such as *functions*, *predicates*, where predicates are essentially functions with a *boolean* return type and *annotations* which can be used for defining search behavior for the solver and more. Functions and predicates in MiniZinc can be user-defined even though many build-in predicates and functions, named *global constraints* are included with MiniZinc. Furthermore, it is possible with MiniZinc to abstract data from the model, by the definition of MiniZinc Data (Dzn) files. Different Dzn files can be used on the same model if the data file contain data for a set of stated definitions in the MiniZinc model.

Earlier works on university timetabling, such as Bettinelli et al. (2015), have often revolved around the idea of scheduling classes for a single week and then replicating this schedule for all weeks of the semester. This way of scheduling is however not realistic for real-world timetabling problems where classes might not occur on a weekly basis, teachers might have planned vacation, or where external conditions affect days of only some weeks such as holidays or isolated events at the university. This way of formulating timetabling problems is simply too rigid to accommodate the complexity of timetabling problems at modern-day universities.

The aim of this thesis is to investigate how realistic educational timetabling problems can be modeled in constraint programming and how efficient different types of constraint programming solvers are at finding feasible or optimal solutions to these problems. For this, MiniZinc and FlatZinc will be utilized along with a range of different solvers.

Two different formulations will be presented. A *compact formulation*, which will

be presented in terms of the *IMADA Timetabling Problem*. This problem is the timetabling problem of scheduling the elective courses offered by the Department for Mathematics and Computer Science at the University of Southern Denmark and revolves around scheduling of classes into suitable times and rooms, with constraints imposed on the problem by students and teachers. In Chapter 2.1 the *compact formulation* in terms of the IMADA Timetabling Problem will be described and a solution in terms of a MiniZinc model will be presented.

An *extended formulation* will be presented in terms of the problem introduced for the *International Timetabling Competition 2019* (ITC2019). This problem revolves around choosing suitable *time patterns*, defined in terms of weeks, days, start time and duration for a set of classes. This problem also involves choosing which classes a student should attend, according to a set of courses that the student is signed up for. In Chapter 3.1 the extended formulation will be described in terms of the ITC2019 problem and two models will be presented in MiniZinc and FlatZinc respectively.

The compact and extended formulations are both *flexible* formulations and does not suffer from the inflexibility of earlier works on timetabling. This means that the load can change through-out the weeks and that they are both able to accommodate disruptions in a schedule at any time needed. However, the formulations do differ and this difference will be discussed and analyzed in Chapter 4 where a method for transforming a model in *compact* form into a model in *extended* form will also be presented.

In Chapter 5 results from a number of tests on the implemented models will be presented, along with a discussion of the results with respect to the two formulations, and a conclusion of the findings in this thesis will be given in Chapter 6.

2 A Compact Formulation: the IMADA case

In Alberg (2018), *Solving the IMADA Timetabling Problem using Constraint-based Local Search* we wrote about the IMADA Timetabling Problem and presented a mixed-integer linear programming (MILP) formulation of the problem. The IMADA Timetabling Problem was modeled and attempted solved using the general-purpose local search solver LocalSolver[1], along with a wide range of problems from the MIPLIB 2010 Problem Set presented in Koch et al. (2011). Relaxation techniques were implemented, but ultimately it was not possible to find any feasible solutions for the IMADA Timetabling problem using LocalSolver. The definition of the IMADA Timetabling Problem defined in this chapter reuse some components from Alberg (2018), but is mostly rewritten.

In this chapter a *Compact formulation* will be stated in terms of the IMADA Timetabling Problem which is the timetabling problem of scheduling the elective courses offered by the Department for Mathematics and Computer Science at the University of Southern Denmark. Real-world data for a semester was given, along with a mixed integer linear programming (MILP) implementation. The MILP implementation reads data from different sources and constructs a data file for use as input for the MILP model.

As mentioned, MiniZinc allows this separation of model and data as well by *MiniZinc Data*. This makes it possible to use the same model on different data sets. This feature was used by extending the MILP implementation such that the it was able to construct a Dzn data file for use with a MiniZinc model.

A constraint programming model was made for the IMADA Timetabling Problem in MiniZinc and solved using different solvers. The problem will be described in Section 2.1, and the implemented MiniZinc model as well as the problem in constraint programming terms will be presented in Section 2.2.

[1] http://localsolver.com

2.1 The IMADA Timetabling Problem

The IMADA Timetabling Problem is a problem of scheduling classes of elective courses into suitable rooms and times within a semester. In this chapter, a class, i.e. a lecture or exercise session, will be referred to as an *event*. Thus, an event is a single occurrence of a class of a course.

An event is *scheduled* when a suitable time and room has been chosen. A *time slot* for an event is represented by a *week*, *day* and *time* of the day, where *time* is a section of a day. For this case of the IMADA Timetabling Problem, a day is divided into sections or *times* of 1 hour each. Each event has a *duration* represented by the number of *times* needed for the event and is fixed to a specific week.

The problem involves a number of *hard* constraints that must be satisfied for a feasible solution to exist, and a number of *soft* constraints that does not prevent a feasible solution, but where a violation of the constraint adds a *penalty* to the objective function. The goal of solving the IMADA Timetabling Problem is to find a solution that satisfies all *hard* constraints, and minimizes the penalty imposed on the objective function.

The hard constraints can be defined as:

H1 All events must be scheduled exactly once,

H2 Multiple events may not occupy the same room at the same time,

H3 An event may not be assigned to a room if it is declared unavailable at the time where the event takes place, or if the room is unsuitable for the course which the event is a part of,

H4 An event should not be scheduled in a time slot where it would not have time to finish,

H5 An event should not be scheduled into time slots which are not allowed by calendar,

H6 A teacher may only teach one class at a time, and only if the teacher is available,

H7 For any course, only one event for that course must occur per day,

H8 For any course in each week, *introduction* events precede *exercise* events, and *exercise* events precede *laboratory* events,

The soft constraints can be defined as:

S1 Minimize the number of student conflicts, meaning the number of events a student has at any point in time,

S2 Events of the same type should preferably be scheduled to the same day of the week and the same time slot of the day,

S3 Events of the same type should preferably be scheduled in the same room every week,

S4 A teacher should not teach more than one event per day,

S5 Each student should preferably not have to attend more than 3 classes per day,

S6 Minimize the number of events occurring outside a normal work schedule (i.e. 9:00 to 17:00).

A feasible solution is found when all events have been scheduled such that none of the hard constraints are violated.

2.2 MiniZinc Model

In this section code samples of the MiniZinc model for the compact formulation will be presented. The full MiniZinc model can be found in the (*Project Repository* n.d.), namely the file `imada-mzn/models/imada-int.mzn`.

2.2.1 Notation

For the definition of the MiniZinc model for the IMADA Timetabling problem the following notation is used:

W Set of weeks,

D Set of days of a week,

h_s, h_e The starting time and ending time of a day, respectively,

C Set of courses,

α Set of teachers,

β Set of students,

R Set of rooms,

O_r Set of time slots where room $r \in R$ is occupied

E Set of events,

D_e Duration for event $e \in E$ given as a number of time slots,

W_e Defined week of each event $e \in E$.

C_e Defined course of each event $e \in E$.

A_s The set of events that student s is attending.

P Set of pairs of events $e_i, e_j \in E$, where e_i should preceed e_j for every pair.

ρ Set of pairs of events $e_i, e_j \in E$, where e_i and e_j is of the same type (i.e. lecture, exercise session, etc.), and $C_{e_i} = C_{e_j}$.

In the model, three definitions of a time slot is introduced. This is done by flattening the time definitions *week*, *day* and *time* into a single definition. A set of time slots T^w denote time slots within a week and T denote time slots over all weeks and days. They are defined as follows, where n is the number of sections per day $h_e - h_s$:

$$T^w = \Big\{0, 1, \ldots, |D|n - 1\Big\} \tag{2.1}$$

$$T = \Big\{0, 1, \ldots, |W| \cdot |D|n - 1\Big\} \tag{2.2}$$

A visual example can be seen in Table 2.1.

	Mon	Tue	$\cdots$	$\lvert D\rvert$
1	0	n	$\cdots$	$(\lvert D\rvert - 1)n$
2	1	$n+1$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$		
n	$n-1$	$2n-1$	$\cdots$	$\lvert D\rvert n - 1$

	Mon	$\cdots$
1	$\lvert D\rvert n$	$\cdots$
2	$\vdots$	
$\vdots$	$\vdots$	
n		

Table 2.1: Visual representation of *time slots* for n sections per day

Every event has to be scheduled into one or more time slots and a room. The time slots and room can be flattened into the definition of a *slot*. Every slot in the set of slots S will then correspond to a unique time on a given day in a given week in a given room.

The set of *slots* S can be defined as:

$$S = \Big\{1, 2, \ldots, |W| \cdot |D| \cdot (h_e - h_s) \cdot |R|\Big\} \tag{2.3}$$

A visual example of *slots* S can be seen in Table 2.2.

By using the definitions of *time slot* and *slot*, the problem of finding a feasible day, week, time and room for an event e, can be simplified to finding a feasible *slot* for event e. This is useful for defining constraints, as seen in this section.

	Mon		Tue		⋯	|D|	
	r_i	r_j	r_i	r_j	⋯	r_i	r_j
1	1	2	$2n+1$	$2n+2$	⋯	⋯	$2n(\|D\|-1)+2$
2	3	4	⋮	⋮			⋮
⋮	⋮	⋮					
n	$2n-1$	$2n$	⋯	⋯	⋯	⋯	$2n\|D\|$

	Mon		⋯
	r_i	r_j	⋯
1	$2n\|D\|+1$	$2n\|D\|+2$	⋯
2	⋮	⋮	
⋮	⋮	⋮	
n			

Table 2.2: Visual representation of *slots* for two rooms r_i, r_j for n sections per day

The variables for the model will now be introduced. Even though the definition of *slots* is able to simplify the problem to having one variable per event, multiple other arrays of size $|E|$ are kept to simplify the constraints. These are

- $\mathcal{W}_e \in W$ indicate the chosen week, for event e.
- $\mathcal{D}_e \in D$ indicate the chosen day, for event e.
- $\mathcal{H}_e \in H$ indicate the chosen time of the day for event e.
- $\mathcal{R}_e \in R$ indicate the chosen room, for event e.

Decision variables representing *time slot* and *slot* for an event e is defined as:

- $\mathcal{T}_e \in T$ indicate the chosen *time slot* for event e.
- $\mathcal{S}_e \in S$ indicate the chosen *slot* for event e.

These decision variables are defined using channelling constraints, such that:

$$\mathcal{T}_e = \mathcal{H}_e + (\mathcal{D}_e - 1) \cdot (h_e - h_s) + (\mathcal{W}_e - 1) \cdot |D| \cdot (h_e - h_s)$$
$$\mathcal{S}_e = \mathcal{T}_e \cdot |R| + \mathcal{R}_e$$

2.2.2 Hard Constraints

In this section the hard constraints from Section 2.1 will be described. By defining variables $\mathcal{W}$, $\mathcal{D}$, $\mathcal{H}$ and $\mathcal{R}$, the model already enforces that every event is scheduled exactly once (the hard constraint H1), and because of the domains of the variables, H5 is also enforced. However, the week of event e should be restricted to be within the week defined for the event, W_e.

This is done by retricting $\mathcal{W}_e$ to W_e as follows:

$$\mathcal{W}_e = w \qquad \forall e \in E,\ w \in W \text{ where } w = W_e$$

The corresponding MiniZinc constraint is

```
constraint forall(e in Events, w in Weeks where InWeek[e]==w)(
    eventWeek[e] = w
);
```

Constraint H2 states that multiple events may not occupy the same room at the same time. This is enforced by using one of MiniZincs build-in *global constraints*, `cumulative`. The `cumulative` constraint is used for describing cumulative resource usage, where in this case the resource is stated in terms of rooms, which are *used* by events. The `cumulative` constraint is defined as follows: (Peter J. Stuckey, 2018)

```
1 cumulative(array[int] of var int: s, array[int] of var int: d,
2            array[int] of var int: r, var int: b)
```

where a set of tasks given by start times s, durations d and resource requirements r never requires more than a global resource bound b at any one time.

However, before it is possible to use the `cumulative` constraint the start times should be defined. The duration D of an event does not work with the previously defined slots $\mathcal{S} \in S$, since two subsequent slots S_x and S_{x+1} points to two different rooms if $|R| > 1$. The set of time slots T does not include the notion of rooms, but can be used for an alternative representation that does.

If the starting time of an event in a room is defined as

$$\mu_e = \mathcal{T}_e \cdot |T| \cdot (\mathcal{R}_e - 1)$$

then subsequent time slots in the same room will give subsequent values of μ_e. Thus, μ_e can be used as starting times for the `cumulative` constraint, with D as the durations and since classes only occupy one room at a time, the resource requirements is 1 for every $e \in E$.

$$\text{cumulative}\Big(\big[\mu_e \mid e \in E\big],\ D,\ [1 \mid e \in E],\ 1\Big)$$

The MiniZinc constraint is defined as

```
1 constraint cumulative(
2     [Timeslot[e]+numTimeslots*(eventRoom[e]-1) | e in Events],
```

```
    Duration,
    RoomsRequired,
    1
);
```

Every room r have an associated set of time slots O_r where room r is occupied. A room should not be chosen for event e if any chosen time slot $\{\mathcal{T}_e, \ldots, \mathcal{T}_e + D_e\}$ is in the set of occupied slots for the room $O_{\mathcal{R}_e}$, as stated in H3. This constraint corresponds to:

$$\left|\left\{\mathcal{T}_e, \ldots, \mathcal{T}_e + D_e\right\} \cap O_{\mathcal{R}_e}\right| = 0 \qquad \forall e \in E \tag{2.4}$$

Since sets of variables such as $\{\mathcal{T}_e, \ldots, \mathcal{T}_e + D_e\}$ is not well-suppored in MiniZinc, this is avoided. Instead, another constraint is introduced for each time slot in $\{\mathcal{T}_e, \ldots, \mathcal{T}_e + D_e\}$ such that,

$$\mathcal{T}_e + d \notin O_{\mathcal{R}_e} \qquad \forall e \in E,\, d \in \{0, \ldots, D_e - 1\} \tag{2.5}$$

This prevents events from being scheduled into rooms occupied at the time of the events, and the corresponding MiniZinc constraint is defined as:

```
constraint forall(r in Rooms, e in Events where eventRoom[e]==r, d in
 ↪  0..Duration[e]-1)(
    not(member(RoomOccupied[r], Timeslot[e]+d))
);
```

Constraint H4 states that every event should have time to finish within the current day. For this, $\mathcal{H}_e$ denotes the section of the day for event e, which means that the constraint can be enforced by

$$\mathcal{H}_e + D_e \leq h_e \qquad \forall e \in E \tag{2.6}$$

which in MiniZinc corresponds to

```
constraint forall(e in Events)(
    eventHour[e]+Duration[e] <= hourEnd
);
```

The constraint H6 does in fact state two constraints, namely that *a teacher may only teach one class at a time* and *a teacher may only teach if the teacher is available*, thus these will be added seperately.

For the first part of H6, the build-in MiniZinc *global constraint* `disjunctive` is used, which restricts a set of tasks given by a set of start times s and a set of durations d to not overlap:

```
predicate disjunctive(array[int] of var int: s,
                      array[int] of var int: d)
```

This global constraint can be used to model that all classes taught by a teacher should be disjunctive.

$$\text{disjunctive}\Big(\big\{\mathcal{T}_e \mid e \in \theta_t\big\}, \big\{D_e \mid e \in \theta_t\big\}\Big) \qquad \forall t \in \alpha \tag{2.7}$$

The corresponding MiniZinc constraint is:

```
constraint forall(t in Teachers)(
    % Can't we just remove the "if" ?
    if length(e in TeacherEvents[t])(1)>0 then
        disjunctive(
            [Timeslot[e] | e in TeacherEvents[t]],
            [Duration[e] | e in TeacherEvents[t]]
        )
    endif
);
```

For the second part of the constraint, the availability of the teacher is modelled the same way as for the availabilities of rooms (see H3). That is, where the set of time slots where a room r is unavailable O_r is exchanged with the set of time slots where a teacher t is busy B_t. If θ_t is the set of events for teacher t, then

$$\mathcal{T}_e + d \notin B_t \qquad \forall t \in \alpha,\, e \in \theta_t,\, d \in \{0, \ldots, D_e - 1\} \tag{2.8}$$

enforces that every event taught by a teacher t is only scheduled at times where teacher t is not marked as busy. The corresponding MiniZinc constraint is defined as:

```
constraint forall(t in Teachers, e in TeacherEvents[t], d in 0..Duration[e]-1)(
    not(member(TeacherBusy[t], Timeslot[e]+d))
);
```

As stated in H7 for any course, only one event for that course must occur per day. Thus, this can be archieved by using MiniZinc's build-in global constraint `alldifferent`. The `alldifferent` constraint takes an array of variables, and restrict them to take different values. Thus, H7 can be enforced by restricting the days of each event of each course to be all different.

$$\text{alldifferent}\Big(\big\{\mathcal{D}_e \mid W_e == w \wedge C_e == c\big\}\Big) \qquad \forall w \in W,\, c \in C$$

The corresponding MiniZinc constraint is defined:

```
constraint forall(w in Weeks, c in Courses)(
    alldifferent(e in Events where InWeek[e]==w /\ PartOf[e]==c)(
        eventDay[e]
    )
);
```

Constraint H8 states that some events should precede other events in time. These events are given as a set of pairs $p_i, p_j \in P$, where p_i and p_j are events, and where p_i should precede p_j. The constraint can be enforced by:

$$\mathcal{S}_{p_i} < \mathcal{S}_{p_j} \qquad \forall p \in \{1, \cdots, |P|\} \text{ where } \mathcal{W}_{p_i} = \mathcal{W}_{p_j}$$

The corresponding MiniZinc constraint is:

```
constraint forall(p in 1..numPrecedences where
    InWeek[Precedences[p,1]] == InWeek[Precedences[p,2]])(
        Scheduled[Precedences[p,1]] < Scheduled[Precedences[p,2]]
);
```

2.2.3 Soft Constraints

In this section, the soft constraints will be described as implemented in the MiniZinc model. The soft constraints have variable values that are penalized in the objective function if the conditions on the variables aer not satisfied.

For the model of the soft constraint S1 one of MiniZinc's *global constraints* is used. The function `global_cardinality` is in MiniZinc clared as:

```
global_cardinality(array [int] of var int: x,
                   array [int] of int: c)
```

and returns an array of the number of occurrences of c_i in x $\forall i \in \{1, 2, \ldots, |c|\}$.

This is equivalent to the following, where $(a \rightarrow b) \wedge (\neg a \rightarrow c)$ means that if a is *true*, then b, else c:

$$\text{global\_cardinality}\Big(X, C\Big): \quad \Big[\sum_{x \in X}(x = c \rightarrow 1) \wedge (x \neq c \rightarrow 0) \mid c \in C\Big] \tag{2.9}$$

The `global_cardinality` function is used to count the number of events a student s has at any given time slot. That is for each time slot occupied by an event student s is attending (including its duration), the cardinality for the element of that timeslot is added by 1. For each element returned by the `global_cardinality` c, the absolute value of $c - 1$ denotes a violation at the given time slot for a student s. That is, for 0 or 1 events at a given time slot, the violation is 0, while if student s have 2 or 3 events at a given time slot, the violation is 1 or 2 respectively. This is given by:

$$\Big[|c-1| \mid c \in \Big[\sum_{x \in X}(x = t \rightarrow 1) \wedge (x \neq t \rightarrow 0) \mid t \in T\Big]\Big] \quad \forall s \in \beta \tag{P1}$$
$$\text{where } X = \Big[\mathcal{T}_e + d \mid e \in A_s,\, d \in \{0, \ldots, D_e - 1\}\Big]$$

where the sum of this is the total violations for student s. The MiniZinc constraint is defined as follows:

```
constraint forall(s in Students)(
    studentConflicts[s] = sum(c in global_cardinality(
        [Timeslot[e]+d | e in Attending[s], d in 0..Duration[e]-1],
        Timeslots
    ))(abs(c-1))
);
```

The constraint S2 states that events of the same type should preferably be schedule to the same day of the week, and the same time slot of the day. For this, ρ is used, which is the set of pairs of events where the events of each pair has the same course and is of the same type. Thus, for each pair in ρ, the constraint should penalize the objective function according to the distance between the pair of events' time slots within the week. Since T_e^w is defined as the time slot of event e within a week, the distance between two events $e_i, e_j \in \rho$ can be found by:

$$|\mathcal{T}_{e_i}^w - \mathcal{T}_{e_j}^w|$$

By this definition, the penalty to be imposed on the objective function can be defined as:

$$\sum_{e_i,e_j \in \rho} |\mathcal{T}^w_{e_i} - \mathcal{T}^w_{e_j}| \tag{P2}$$

For the MiniZinc constraint, the penalty is defined for each pair of events in ρ and summed later. The constraint is defined as follows:

```
constraint forall(p in 1..numPairings)(
    abs(TimeslotInWeek[Pairings[p,1]] - TimeslotInWeek[Pairings[p,2]]) =
     ↪  timeDiscrepancies[p]
);
```

The soft constraint S3 is defined the same way as constraint S2, just for the rooms of a pair of events in ρ. That is:

$$\sum_{e_i,e_j \in \rho} |\mathcal{R}_{e_i} - \mathcal{R}_{e_j}| \tag{P3}$$

This is actually incorrect. A short discussion about this is added to Chapter 7.

The constraints S4 and S5 states that the number of events per day for teachers and students respectively should be minimized. For a teacher a penalty is imposed on the objective function for each event that exceeds one event for each day. The number of events per day is counted using the MiniZinc build-in function `global_cardinality`, as previously defined in this section. For the input parameters, X is an array of unique identifiers of the day of each event, such that each element $x_e \in X$ is the unique identifer of a day, for event e and C is an array of unique identifiers for each day.

The identifier of a day for an event e, where W_1 denotes the number of the first week, is defined as:

$$(\mathcal{W}_e - W_1)|D| + \mathcal{D}_e$$

That is, given a problem with weeks of 5 days, if an event is scheduled on Monday of the first week of the semester the unique identifier of the day will be 1, while if an event is scheduled on Tuesday of the second week of the semester the unique identifier of the day will be 7, etc.

By defining:

$$X_t = \Big[(\mathcal{W}_e - W_1)|D| + \mathcal{D}_e \mid e \in \theta_t\Big] \quad \text{and}$$
$$C = \Big[(w - W_1)|D| + d \mid w \in W,\, d \in D\Big]$$

The number of violations of this constraint for a teacher t can be defined as:

$$\Big[\sum_{x \in X_t} (x = c \rightarrow 1) \wedge (x \neq c \rightarrow 0) \mid c \in C\Big] \qquad \forall t \in \alpha \tag{P4}$$

In MiniZinc terms the constraint is defined like above, but the penalty imposed on the objective function is defined as the number of events on the day with the maximum number of events for each teacher, where the minimum penalty is 1. This tells the solver to minimize the number of events on the day with the maximum number of events for each teacher. The constraint is defined as follows:

```
constraint forall(t in Teachers)(
    max(global_cardinality(
        [(eventWeek[e]-firstWeek)*numDays + eventDay[e] | e in
        ↪  TeacherEvents[t]],
        [(w-firstWeek)*numDays + d | w in Weeks, d in Days]
    )) <= maxEventsxDayxTeacher[t]
);
```

For S5, the same technique is used for the minimization of the number of events per day for each student. However, since this constraint should only impose a penalty on the days where a student have more than 3 events, the minimum penalty is set to 3, that is, the lower bound of the variable representing the penalty imposed on the objective function. If the imposed penalty of S5 is 3, it means that the student does not have any days where the student should attend more than 3 events. The equation is kept for reference, where θ_s of X_s is the set of events for student s.

$$\Big[\sum_{x \in X_s} (x = c \rightarrow 1) \wedge (x \neq c \rightarrow 0) \mid c \in C\Big] \qquad \forall s \in \beta \tag{P5}$$

The constraint S6 states that the number of events occuring outside a normal work schedule, such as 9:00 to 17:00, should be minimized.

This is modelled by imposing a penalty on the objective function, if the time section of the day where an event e starts $\mathcal{H}_e$ is less than a defined *good* start time g_s, or

if the time section of the day where e ends, $\mathcal{H}_e + D_e$ is greater than a defined *good* ending time g_n. The penalty is defined according to the number of time slots from g_s and g_n, such that a greater penalty is imposed the more the the start or end of e violate g_s and g_n respectively.

$$(\mathcal{H}_e + D_e > g_n \rightarrow \mathcal{H}_e + D_e - g_n) \wedge (\mathcal{H}_e < g_s \rightarrow \mathcal{H}_e - g_s) \qquad \forall e \in E \qquad \text{(P6)}$$

For the MiniZinc constraint, this penalty is stored in an array for each event e which is summed and added as a penalty to the objective function. The constraint in MiniZinc is defined as:

```
constraint forall(e in Events)(
    if eventHour[e]+Duration[e] > goodDayEnd then
        eventHour[e]+Duration[e]-goodDayEnd = badSlots[e]
    else if eventHour[e] < goodDayStart then
        goodDayStart-eventHour[e] = badSlots[e]
    endif
);
```

2.2.4 The Objective Function

The penalties are now defined in Equations P1 through P6. For minimization in minizinc these are passed as the variables to be minimized by the solver.

$$\text{minimize } P1 + P2 + P3 + P4 + P5 + P6$$

In MiniZinc the objective function is then defined as follows:

```
solve :: int_search(Scheduled, smallest, indomain_min, complete)
     minimize sum(maxEventsxDayxTeacher) +
              sum(timeDiscrepancies) +
              sum(roomDiscrepancies) +
              sum(maxStudentOverlaps) +
              sum(maxEventsxDayxStudent) +
              sum(badSlots);
```

3 An Extended Formulation: the ITC2019 case

In this chapter a *extended* formulation will be presented in terms of the problem presented for the International Timetabling Competition 2019 (ITC2019).

The International Timetabling Competition is a timetabling competition aimed at motivating further research on complex university course timetabling problems and is supported by the international series of conferences on Practice and Theory of Automated Timetabling (PATAT). Data sets containing real-world instances of timetabling problems, contributed by universities around the world, is provided and participants of the competition compete to find the best feasible solution for each instance. In 2018 the fourth International Timetabling Competition 2019 (ITC2019) was announced in Müller, Rudová, and Müllerová (2018).

A walk-through and definition of the ITC2019 problem will be described in Section 3.1. During the course of this thesis, two models were made in an attempt to solve the ITC2019 problem, in MiniZinc and FlatZinc respectively. A shared dependency for sectioning students of the ITC instances will be presented in Section 3.2.1, and a section dedicated to *schedules* and *classes* can be found in Section 3.2.2. The MiniZinc model is written in MiniZinc, and a program was developed in Python to generate data for use with the model. This will be introduced in Section 3.3. For the implementation in FlatZinc, instead of having one model for all instances, a program was developed which generates a model in FlatZinc from an instance file. This will be introduced and described in detail in Section 3.4.

3.1 The ITC2019 Problem

The ITC2019 problem is a generalized CB-CTT problem where classes have to be chosen for students among courses they have signed up for, and rooms and times have to be chosen for the set of classes. The problem involves many variables and constraints which will be described here.

A *course* is defined as a number of *configurations* where each *configuration* have one or more *subparts*, which in turn have one or more *classes*. A limit is imposed on each class, defining the maximum number of students that can attend that class, and a class may have a *parent class* defined which means that a student which is assigned to attend the class must also attend its parent class. Ultimately, a course structure can be seen as a tree structure. An example of this can be seen in figure 3.1, for a course $\mathcal{C}_1$ with two configurations $\mathcal{F}_1, \mathcal{F}_2$, three subparts P_1, P_2, P_3 and six classes $C_1, C_2, \ldots, C_6$.

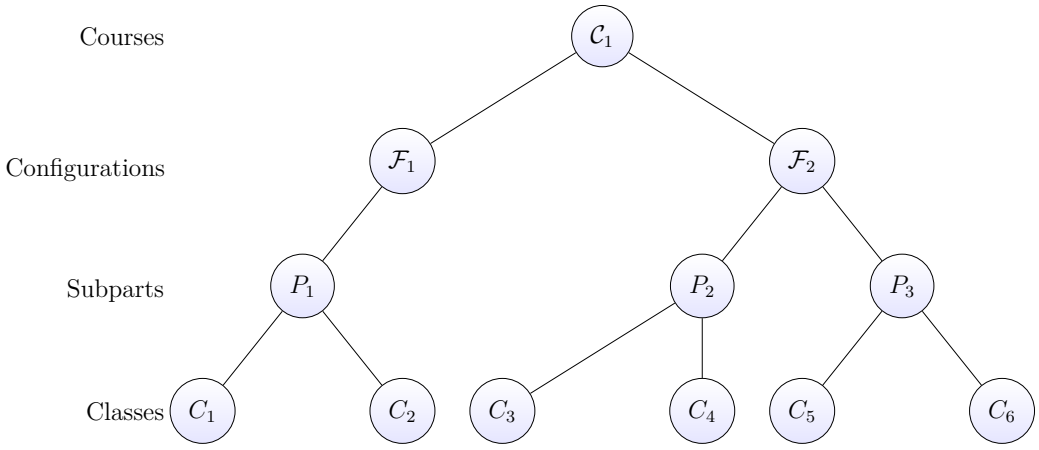

Figure 3.1: Example of course structure for a course $\mathcal{C}_1$

This structure allows for complex course structures to be build, with multiple types of classes, i.e. lectures, exercise and laboratory sessions. It also makes it possible for a teacher to define multiple parallel sections or classes for the same course, or courses where some classes are for all students attending the course, while other classes are just for a subset of the students.

A student have a set of courses that he/she is signed up. For every course a student is signed up for, exactly one class of each subpart of a single configuration of the course have to be chosen for that student. This selection of a class for a student will be referred to as *sectioning* of the student. Cases where a student is sectioned into classes that overlap in time should be minimized, but does not cause a solution to be infeasible.

Each class has a number of times and a number of rooms available for that class. A time for a class is defined as a pattern of weeks, days, a start time and a duration of the class. An example for a possible time pattern for a class could be:

Week 1,3 and 5 on Mondays and Thursdays at 8:00am for 2 hours

One of these time patterns will be referenced to as a *pattern* or a *schedule* in this report, and the *duration* of a pattern will also be referred to as the *length*.

A room is defined by a number of time patterns as well, defining times when the room is unavailable. These will be referred to as the *unavailability patterns* of the room.

For every class exactly one *room* and one *schedule* have to be chosen among the available schedules and rooms for that class. If two or more classes are sharing a room, their schedules should not overlap at any given point in time, and a chosen pattern for a class should not overlap with any *unavailability patterns* of the chosen room.

Every feasible schedule and room for a class is associated with a penalty which makes it possible to prefer some rooms or schedules over others. This penalty should be minimized.

In shorter terms the following constraints can be defined:

(CS) Class Schedule A class can only be scheduled into one of the schedules defined for that class,

(CR) Class Room a class can only be scheduled into one of the rooms defined for that class,

(SO) Schedule Overlap only one class per room at any given time,

(RU) Room Unavailable a class can not be scheduled into a room where the schedule of the class overlaps with a time where the room is marked as being occupied,

(SP) Schedule Penalty for each class the penalty of the chosen schedule should be minimized,

(RP) Room Penalty for each class the penalty of the chosen room should be minimized,

(SC) Student Conflicts for each student, the number of overlapping classes that the student is attending should be minimized.

A number of additional constraints may be imposed on the problem. These constraints will be referred to as *distribution constraints*. Distribution constraints are always defined for a set of classes and in some cases take additional parameters. Furthermore, distribution constraints can be either *hard*, meaning that the constraint should be enforced for a solution to be feasible, or *soft* in which case a *penalty* is defined for the constraint. In the latter case, a solution can be feasible if the constraint is violated, but the penalty should be minimized.

All distribution constraints are optional, thus an instance of the ITC problem does not have to include any distribution constraints, but can also include an unlimited number of any of the distribution constraints.

The types of distribution constraints are defined in Müller, Rudová, and Müllerová (2018), but for reference they are included in this thesis. The remaining of this section is a 1:1 of Müller, Rudová, and Müllerová (2018):p. 13–19, with added abbreviations, unless otherwise stated such as corrections.

(SS) **SameStart** Given classes must start at the same time slot, regardless of their days of week or weeks. This means that $C_i.start = C_j.start$ for any two classes C_i and C_j from the constraint; $C_i.start$ is the assigned start time slot of a class C_i.

(ST) **SameTime** Given classes must be taught at the same time of day, regardless of their days of week or weeks. For the classes of the same length, this is the same constraint as SameStart (classes must start at the same time slot). For the classes of different lengths, the shorter class can start after the longer class but must end before or at the same time as the longer class. This means that

$$\begin{aligned}(C_i.start \leq C_j.start \wedge C_j.end \leq C_i.end) \\ \vee (C_j.start \leq C_i.start \wedge C_i.end \leq C_j.end)\end{aligned} \tag{3.1}$$

for any two classes C_i and C_j from the constraint; $C_i.end = C_i.start + C_i.length$ is the assigned end time slot of a class C_i.

(DT) **DifferentTime** Given classes must be taught during different times of day, regardless of their days of week or weeks. This means that no two classes of this constraint can overlap at a time of the day. This means that

$$(C_i.end \leq C_j.start) \vee (C_j.end \leq C_i.start) \tag{3.2}$$

for any two classes C_i and C_j from the constraint.

(SD) **SameDays** Given classes must be taught on the same days, regardless of their start time slots and weeks. In case of classes of different days of the week, a class with fewer meetings must meet on a subset of the days used by the class with more meetings. For example, if the class with the most meetings meets on Monday–Tuesday–Wednesday, all others classes in the constraint can only be taught on Monday, Wednesday, and/or Friday (correction: *"other classes in the constraint can only be taught on Monday, Tuesday, and/or Wednesday"*). This means that

$$((C_i.days \text{ or } C_j.days) = C_i.days) \vee ((C_i.days \text{ or } C_j.days) = C_j.days) \tag{3.3}$$

for any two classes C_i and C_j from the constraint; $C_i.days$ are the assigned days of the week of a class C_i, doing binary "or" between the bit strings.

(DD) **DifferentDays** Given classes must be taught on different days of the week, regardless of their start time slots and weeks. This means that

$$(C_i.days \text{ and } C_j.days) = 0 \tag{3.4}$$

for any two classes C_i and C_j from the constraint; doing binary "and" between the bit strings representing the assigned days.

(SW) **SameWeeks** Given classes must be taught in the same weeks, regardless of their time slots or days of the week. In case of classes of different weeks, a class with fewer weeks must meet on a subset of the weeks used by the class with more weeks. This means that

$$(C_i.weeks \text{ or } C_j.weeks) = C_i.weeks) \vee (C_i.weeks \text{ or } C_j.weeks) = C_j.weeks) \tag{3.5}$$

for any two classes C_i and C_j from the constraint; doing binary "or" between the bit strings representing the assigned weeks.

(DW) **DifferentWeeks** Given classes must be taught on different weeks, regardless of their time slots or days of the week. This means that

$$(C_i.weeks \text{ and } C_j.weeks) = 0 \tag{3.6}$$

for any two classes C_i and C_j from the constraint; doing binary "and" between the bit strings representing the assigned weeks.

(O) **Overlap** Given classes overlap in time. Two classes overlap in time when they overlap in time of day, days of the week, as well as weeks. This means that

$$\begin{aligned}&(C_j.start < C_i.end) \wedge (C_i.start < C_j.end) \\ &\wedge ((C_i.days \text{ and } C_j.days) \neq 0) \wedge ((C_i.weeks \text{ and } C_j.weeks) \neq 0)\end{aligned} \tag{3.7}$$

for any two classes C_i and C_j from the constraint, doing binary "and" between days and weeks of C_i and C_j.

(NO) **NotOverlap** Given classes do not overlap in time. Two classes do not overlap in time when they do not overlap in time of day, or in days of the week, or in weeks. This means that

$$\begin{aligned}&(C_i.end \leq C_j.start) \vee (C_j.end \leq C_i.start) \vee \\ &((C_i.days \text{ and } C_j.days) = 0) \vee ((C_i.weeks \text{ and } C_j.weeks) = 0)\end{aligned} \tag{3.8}$$

for any two classes C_i and C_j from the constraint, doing binary "and" between days and weeks of C_i and C_j.

(SR) **SameRoom** Given classes should be placed in the same room. This means that $(C_i.room = C_j.room)$ for any two classes C_i and C_j from the constraint; $C_i.room$ is the assigned room of C_i.

(DR) **DifferentRoom** Given classes should be placed in different rooms. This means that $(C_i.room \neq C_j.room)$ for any two classes C_i and C_j from the constraint.

(SA) **SameAttendees** Given classes cannot overlap in time, and if they are placed on overlapping days of week and weeks, they must be placed close enough so that the attendees can travel between the two classes. This means that

$$\begin{aligned}(C_i.end + C_i.room.travel[C_j.room] \leq C_j.start) \vee \\ (C_j.end + C_j.room.travel[C_i.room] \leq C_i.start) \vee \\ ((C_i.days \text{ and } C_j.days) = 0) \vee ((C_i.weeks \text{ and } C_j.weeks) = 0)\end{aligned} \tag{3.9}$$

for any two classes C_i and C_j from the constraint; $C_i.room.travel[C_j.room]$ is the travel time between the assigned rooms of C_i and C_j.

(P) **Precedence** Given classes must be one after the other in the order provided in the constraint definition. For classes that have multiple meetings in a week or that are on different weeks, the constraint only cares about the first meeting of the class. That is,

- the first class starts on an earlier week or
- they start on the same week and the first class starts on an earlier day of the week or
- they start on the same week and day of the week and the first class is earlier in the day.

This means that

$$
\begin{aligned}
&(\text{first}(C_i.weeks) < \text{first}(C_j.weeks)) \vee \\
&\quad [(\text{first}(C_i.weeks) = \text{first}(C_j.weeks)) \wedge \\
&\qquad [(\text{first}(C_i.days) < \text{first}(C_j.days)) \vee \\
&\qquad\quad ((\text{first}(C_i.days) = \text{first}(C_j.days)) \wedge (C_i.end \leq C_j.start)) \\
&\qquad] \\
&\quad]
\end{aligned}
\tag{3.10}
$$

for any two classes C_i and C_j from the constraint where $i < j$ and first(x) is the index of the first non-zero bit in the binary string x.

(WD_S) **WorkDay(S)** There should not be more than S time slots between the start of the first class and the end of the last class on any given day. This means that classes that are placed on the overlapping days and weeks that have more than S time slots between the start of the earlier class and the end of the later class are violating the constraint. That is

$$
\begin{aligned}
((C_i.days \text{ and } C_j.days) = 0) \quad &\vee \\
((C_i.weeks \text{ and } C_j.weeks) = 0) \quad &\vee \\
(\max(C_i.end, C_j.end) - \min(C_i.start, C_j.start) \leq S)&
\end{aligned}
\tag{3.11}
$$

for any two classes C_i and C_j from the constraint.

(MG_G) **MinGap(G)** Any two classes that are taught on the same day (they are placed on overlapping days and weeks) must be at least G slots apart. This means that there must be at least G slots between the end of the earlier class and the start of the later class. That is

$$
\begin{aligned}
((C_i.day \text{ and } C_j.days) = 0) \quad &\vee \\
((C_i.weeks \text{ and } C_j.weeks) = 0) \quad &\vee \\
(C_i.end + G \leq C_j.start) \quad &\vee \\
(C_j.end + G \leq C_i.start)&
\end{aligned}
\tag{3.12}
$$

for any two classes C_i and C_j from the constraint.

(MD_D) **MaxDays(D)** Given classes cannot spread over more than D days of the week, regardless whether they are in the same week of semester or not. This means that the total number of days of the week that have at least one class of this distribution constraint $C_1, \ldots, C_n$ is not greater than D,

$$\text{countNonzeroBits}(C_1.days \text{ or } C_2.days \text{ or } \cdots C_n.days) \leq D \tag{3.13}$$

where countNonzeroBits(x) returns the number of non-zero bits in the bit string x. When the constraint is soft, the penalty is multiplied by the number of days that exceed the given constant D.

(MDL_S) **MaxDayLoad(S)** Given classes must be spread over the days of the week (and weeks) in a way that there is no more than a given number of S time slots on every day. This means that for each week $w \in \{0, 1, \ldots, \text{nrWeeks} - 1\}$ of the semester and each day of the week $d \in \{0, 1, \ldots, \text{nrDays} - 1\}$, the total number of slots assigned to classes C that overlap with the selected day d and week w is not more than S,

$$\begin{aligned} &DayLoad(d, w) \leq S \\ &DayLoad(d, w) = \\ &\quad \sum_i \Big\{ C_i.length \mid (C_i.days \text{ and } 2^d) \neq 0 \wedge (C_i.weeks \text{ and } 2^w) \neq 0) \Big\} \end{aligned} \tag{3.14}$$

where 2^d is a bit string with the only non-zero bit on position d and 2^w is a bit string with the only non-zero bit on position w. When the constraint is soft (it is not required and there is a penalty), its penalty is multiplied by the number of slots that exceed the given constant S over all days of the semester and divided by the number of weeks of the semester (using integer division). Importantly the integer division is computed at the very end. That is

$$\left(\text{penalty} \times \sum_{w,d} \max(DayLoad(d, w) - S, 0) \right) / \text{nrWeeks} \tag{3.15}$$

($\mathsf{MBR}_{R,S}$) **MaxBreaks(R,S)** This constraint limits the number of breaks during a day between a given set of classes (not more than R breaks during a day). For each day of week and week, there is a break between classes if there is more than S empty time slots in between.

Two consecutive classes are considered to be in the same block if the gap between them is not more than S time slots. This means that for each week $w \in 0, 1, \ldots, \text{nrWeeks} - 1$ of the semester and each day of the week $d \in 0, 1, \ldots, \text{nrDays} - 1$, the number of blocks is not greater than $R + 1$,

$$|\text{MergeBlocks}\{(C.start, C.end) \mid (C.days \text{ and } 2^d) \neq 0 \wedge (C.weeks \text{ and } 2^w) \neq 0\})| \leq R + 1 \tag{3.16}$$

where 2^d is a bit string with the only non-zero bit on position d and 2^w is a bit string with the only non-zero bit on position w.

The MergeBlocks function recursively merges to the block B any two blocks B_a and B_b that are identified by their start and end slots that overlap or are not more than S slots apart, until there are no more blocks that could be merged.

$$(B_a.end + S \geq B_b.start) \wedge (B_b.end + S \geq B_a.start) \implies (B.start = \min(B_a.start, B_b.start)) \wedge (B.end = \max(B_a.end, B_b.end)) \tag{3.17}$$

When the constraint is soft, the penalty is multiplied by the total number of additional breaks computed over each day of the week and week of the semester and divided by the number of weeks of the semester at the end (using integer division, just like for the MaxDayLoad constraint).

($\mathsf{MBL}_{M,S}$) **MaxBlock(M,S)** This constraint limits the length of a block of two or more consecutive classes during a day (not more than M slots in a block). For each day of week and week, two consecutive classes are considered to be in the same block if the gap between them is not more than S time slots. For each block, the number of time slots from the start of the first class in a block till the end of the last class in a block must not be more than M time slots. This means that for each week $w \in \{0, 1, \ldots, \text{nrWeeks} - 1\}$ of the semester and each day of the week $d \in \{0, 1, \ldots, \text{nrDays} - 1\}$, the maximal length of a block does not exceed M slots

$$\max(\{B.end - B.start \mid B \in \text{MergeBlocks}(\{C.start, C.end) \mid (C.days \text{ and } 2^d) \neq 0 \wedge (C.weeks \text{ and } 2^w) \neq 0\}) \}) \leq M \tag{3.18}$$

When the constraint is soft, the penalty is multiplied by the total number of blocks that are over the M time slots, computed over each day of the week and week of the semester and divided by the number of weeks of the semester at the end (using integer division, just like for the MaxDayLoad constraint).

— Müller, Rudová, and Müllerová (2018)

In this chapter, the following notation will be used:

- set of classes C,
- set of students B,
- set of rooms R,
- chosen schedule for class c is denoted $\mathcal{S}_c$,
- chosen room for class c is denoted $\mathcal{R}_c$,
- set of feasible schedules for class c is denoted S_c and
- set of feasible rooms for class c is denoted R_c.

3.2 Processing of Instances

Both implementations made for modeling of the ITC2019 problem utilize a shared dependency used mainly for parsing of ITC2019 instances and defining data structures used by the implementations. Since the implementation for use by the MiniZinc model was implemented at a different time than the implementation for the FlatZinc model, some features are only used by the latter and vice versa. This section will describe the most important features of the parser, and point out features which is only used by one of the implementations.

Besides from reading and parsing an ITC2019 instance file, this module also sections the students of the problem into classes. This is done prior to modeling of the problem, to avoid making the model too complex, and is possible because the students only impose a single *soft* constraint on the model, namely the minimization of conflicting classes of a student.

3.2.1 Sectioning of Students

As described in Section 3.1 a course consist of one or more configurations, each with one or more subparts with one or more classes. This course structure can be defined as a tree structure with the course at the root node, and the classes as leaf nodes.

Students have to be sectioned into exactly one class of each subpart of a single configuration of each course that the student is signed up for. A possible sectioning of a student who has to attend $\mathcal{C}_1$ from Figure 3.1 can be seen in figure 3.2, where the chosen paths for the student is marked with a bold red line. In this example, the student is sectioned into classes C_4 and C_6.

By defining the course structure as a tree, it is possible to section students using a depth-first tree traversal algorithm. However, before traversal of the tree, the

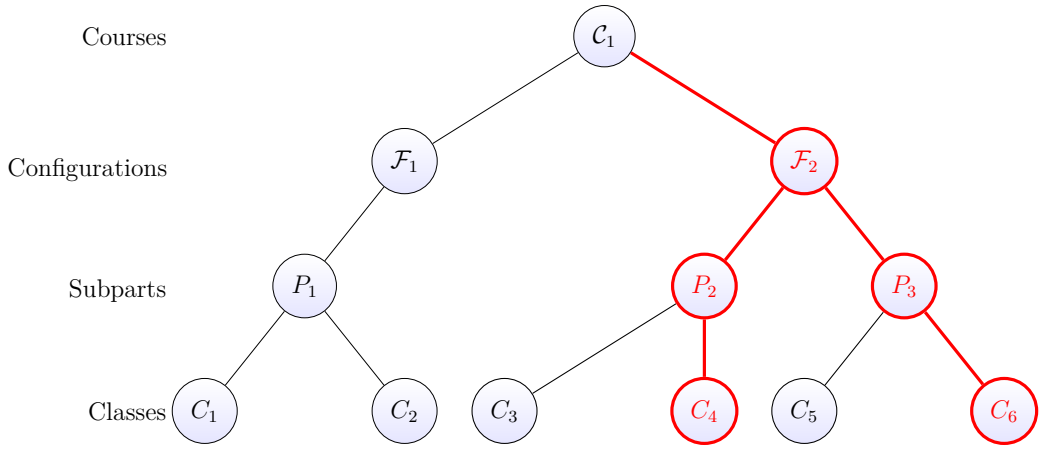

Figure 3.2: Example of student sectioning for course $\mathcal{C}_1$, into classes C_4 and C_6.

structure is altered slightly, to take parent classes into account.

Assuming that C_3 is the parent class of C_5, and C_4 is the parent class of C_6, C_5 and C_6 will be added as the child of C_3 and C_4 respectively. The old nodes will be pruned, and the new course structure will be as pictured in figure 3.3.

Every student who is signed up for $\mathcal{C}_1$ will now be sectioned into classes by performing tree traversal on the tree of figure 3.3. The algorithm will iterate through every configuration and every subpart of the course, and then perform a recursive traversal of the classes of each subpart. When a leaf node is found, the path from the subpart-node to the leaf node, will be returned as the classes of the student, given that there was available spots for all of the classes. In this case, the iteration over the configurations will also stop, since a student should only be sectioned into classes of a single configuration of a course. The implementation of the algorithm can be seen in Listing 3.1, which shows the iteration over students, courses, configurations and subparts, and Listing 3.2 which show the recursive traversal through classes.

```
1  def get_students(self, child):
2      for student in child:
3          student_id = "S" + student.attrib['id']
4          stud = {}
5          stud['classes'] = []
6          chosen_classes = []
7
8          # Needed courses
9          for course in student:
10             course_id = "C" + course.attrib['id']
```

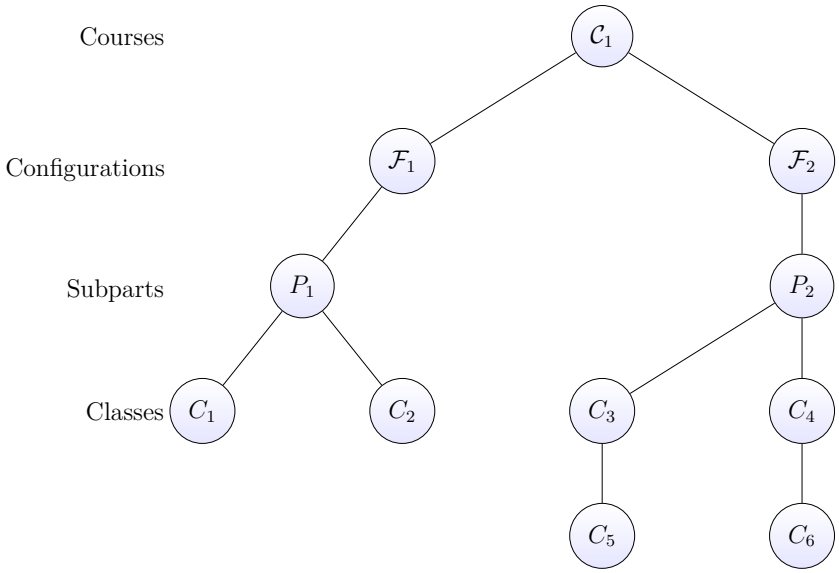

Figure 3.3: Altered course tree which takes parent classes into account.

```
            crs = self.courses[course_id] if course_id in self.courses else
             ↪  None

            for conf in self.course_tree[course_id]:
                for subp in self.course_tree[course_id][conf]:
                    if len(self.course_tree[course_id][conf][subp]) > 0:
                        chosen_classes = self.choose_student_classes(
                            self.course_tree[course_id][conf][subp],
                            chosen_classes
                        )

                    if len(chosen_classes) > 0:
                        stud['classes'] = chosen_classes
                        break

        self.students[student_id] = stud
```

Listing 3.1: Implementation of class assignments for each student by traversal of courses, configurations and subparts.

```
def choose_student_classes(self, subpart, chosen = []):
    for clas in subpart:
        if self.classes[int(clas)].attending < self.classes[int(clas)].limit:
            chosen.append(int(clas))
            self.classes[int(clas)].attending += 1
            if len(subpart[clas]) > 0:
                chosen = self.choose_student_classes(subpart[clas], chosen)
```

```
8                 break
9         return chosen
```

Listing 3.2: Choose classes for a subpart

3.2.2 Schedules and Classes

Schedules is used to indicate *when* a class take place, an is defined by a set of weeks, a set of days, a start time and a duration (or *length*).

As mentioned in the description of the ITC2019 Problem, a single schedule and a single room should be chosen for each class, from a set of rooms and schedules available for that class. An example of a class and its available rooms and schedules in the ITC2019 instance format can be seen in Listing 3.3.

```
1  <class id="40" limit="34" parent="39">
2      <room id="16" penalty="4"/>
3      <room id="21" penalty="0"/>
4      <room id="22" penalty="4"/>
5      <room id="3" penalty="0"/>
6      <room id="13" penalty="0"/>
7      <room id="25" penalty="4"/>
8      <room id="27" penalty="0"/>
9      <room id="7" penalty="0"/>
10     <room id="17" penalty="0"/>
11     <time days="0001000" start="96" length="22" weeks="011111111111110" penalty="0"/>
12     <time days="0000100" start="96" length="22" weeks="011110111111110" penalty="0"/>
13     <time days="0001000" start="120" length="22" weeks="011111111111110" penalty="6"/>
14     <time days="0000100" start="120" length="22" weeks="011110111111110" penalty="0"/>
15     <time days="0001000" start="144" length="22" weeks="011111111111110" penalty="6"/>
16     <time days="0000100" start="144" length="22" weeks="011110111111110" penalty="0"/>
17     <time days="0001000" start="168" length="22" weeks="011111111111110" penalty="6"/>
18     <time days="0000100" start="168" length="22" weeks="011110111111110" penalty="2"/>
19     <time days="0001000" start="192" length="22" weeks="011111111111110" penalty="0"/>
20     <time days="0000100" start="192" length="22" weeks="011110111111110" penalty="8"/>
21     <time days="0001000" start="216" length="22" weeks="011111111111110" penalty="2"/>
22     <time days="0000100" start="216" length="22" weeks="011110111111110" penalty="8"/>
23 </class>
```

Listing 3.3: Example of class in the ITC2019 Instance format, with feasible schedules.

The format declares the starting time and length of the schedule as the number of 5-minute slots, and the days and weeks in a binary representation where a 1 specifies an occurance of the class during the that day or week. The penalty attribute defines how much the objective function should be penalized if the schedule is chosen for the class. As an example, if the following schedule was chosen for a class,

```
<time days="0001000" start="96" length="22" weeks="011111111111110" penalty="0"/>
```

the class would be scheduled at 8:00am to 9:50am on Thursdays on weeks $2, 3, \ldots, 14$ and the objective function would be penalized by 0.

Upon parsing of ITC2019 instance, data related to classes, schedules and rooms is stored in multiple data structures. These differ between the implementations of the MiniZinc model and the FlatZinc model generator.

In the FlatZinc model, a few objects was defined to ease the development of constraints for the ITC2019 problem. This was a lesson learned from the implementation of the MiniZinc model, where data is stored in Python dictionaries. The definition of the `Schedule` object used for storing data about a schedule is seen in Listing 3.4.

```
class Schedule:
    def __init__(self, sid, weeks, days, start, length):
        self.id = sid
        self.name = "T" + str(sid)
        self.weeks = weeks
        self.days = days
        self.start = start
        self.length = length
        self.end = self.start + self.length

    def key(self):
        return self.weeks.to01() + "_" + self.days.to01() + "_" +
         ↪  str(self.start) + "_" + str(self.length)

    def overlap_with(self, s):
        # Time overlap
        if (self.end > s.start and self.start <= s.start) \
           or (s.end > self.start and s.start <= self.start) \
           or (s.start <= self.start and s.end > self.end) \
           or (self.start <= s.start and self.end > s.end):

                # Day overlap
                if (self.days & s.days).any():

                    # Week overlap
                    if (self.weeks & s.weeks).any():
                        return True

        return False

    def first_week(self):
        i = 1
        for w in self.weeks:
            if w:
                return i
            i += 1

    def first_day(self):
```

```
38          i = 1
39          for d in self.days:
40              if d:
41                  return i
42              i += 1
```

Listing 3.4: The `Schedule` class

Furtermore, a class object named `Clas` (to avoid conflicts with Python's built-in keyword *class*) was made to hold information about a class. The `Clas` object holds a set of feasible schedules, a set of feasible rooms, sets of penalties, number of attending students and limit, etc. Furthermore, methods is provided for checking if a class is *fixed* or not. A *fixed* class is defined as a class with only one feasble schedule, and 1 or less feasible rooms. The complete definition of the `Clas` object can be seen in Listing 3.5

```
1   class Clas:
2       def __init__(self, cid):
3           self.cid = cid
4           self.schedule_var = "C" + str(cid) + "_Schedule"
5           self.room_var = "C" + str(cid) + "_Room"
6           self.schedules = set()
7           self.rooms = set()
8           self.schedule_penalty = {}
9           self.room_penalty = {}
10          self.attending = 0
11          self.limit = 0
12
13      def add_schedule(self, schedule):
14          self.schedules.add(schedule)
15
16      def add_room(self, room):
17          self.rooms.add(room)
18
19      def add_room_penalty(self, room, penalty):
20          self.room_penalty[room] = penalty
21
22      def is_fixed(self):
23          if len(self.rooms) <= 1 and len(self.schedules) <= 1:
24              return True
25          return False
26
27      def has_fixed_room(self):
28          if len(self.rooms) <= 1:
29              return True
30          return False
31
32      def has_fixed_schedule(self):
```

```
33          if len(self.rooms) <= 1:
34              return True
35          return False
```

Listing 3.5: The *Clas* class

Another improvement was made during the implementation of the FlatZinc model generator when it was discovered that instances of the ITC2019 competition, contained a high number of identical schedules. This means that the number of schedules in an instance was reduced to an average of 4-5% of the original number of schedules when using FlatZinc generation, as opposed to data generation for the MiniZinc model. The exact number of total and unique schedules for each instance can be seen in Table 3.1.

Instance	Total Schedules	Unique Schedules	Unique Schedules %
wbg-fal10	4.617	154	3.3
lums-sum17	340	93	27.4
bet-sum18	210	50	23.8
pu-cs-fal07	2.958	182	6.2
pu-c8-spr07	30.538	896	2.9
agh-fis-spr17	145.868	9.655	6.6
agh-ggis-spr17	46.671	2.836	6.1
bet-fal17	23.366	595	2.5
iku-fal17	93.386	588	0.6
pu-llr-spr07	8.654	508	5.9
mary-spr17	12.331	620	5.0
muni-fi-spr16	9.556	789	8.3
muni-pdf-spr16c	150.575	2.696	1.8
pu-llr-spr17	9.290	993	10.7
muni-fsps-spr17	11.355	1.953	17.2
tg-fal17	18.384	1.645	8.9
Total	568.099	24.253	4.3

Table 3.1: Number of total schedules, number of unique schedules and the percentage of unique schedules for each ITC2019 Instance.

The unique schedules are found by defining an id for every schedule, composed of the binary strings of the weeks and days of the schedule, and the start time and length. Only unique ids will be added, thus schedule with the same days, weeks, start and length will only be saved once.

3.3 The MiniZinc Model

In this section the data creation for the MiniZinc model will be described, as well as the model for the ITC2019 problem. It is important to note that for this section, *schedules* can be identical and each schedule belongs to exactly one class.

The full MiniZinc model described in this section can be found in the (*Project Repository* n.d.) file `itc2019/mzn/base.mzn`.

3.3.1 Data Creation

For the MiniZinc implementation, after parsing of an instance file for ITC2019 the data is written to a Dzn file for use with a MiniZinc model. However, while the parsed data corresponds to the data from the instance, the data is first transformed into something more easily handled by a constraint programming solver. Furthermore it is possible to prune certain data to reduce the size of the input. Since the details of how the data is created is quite trivial, a *case study* will instead give an example of how data is generated for a single constraint, and give the full list of declarations in terms of MiniZinc to give an overview of what data is available to the model.

Case Study: Pruning Data during Data Creation for a Constraint

While the distribution constraints are defined for only a subset of the classes, some constraints need data for all possible combinations of classes. One example is the data needed for the SO constraint which enforce that

Two classes assigned to the same room, cannot have overlapping schedules.

Thus, the solver needs to know every overlapping schedule for each pair of classes. This data could be represented by an array of pairs, which correspond to every 2-combination of schedules which is overlapping for every pair of classes. This will be of size $\binom{n}{2}$, in the worst case. A way to reduce the size of the data file, is as follows.

Consider Δ to be an array of sets indexed by schedules s_i, with every set defined as the set of schedules $\{s_1, s_2, \ldots, s_n\}$ that overlap with schedule s_i. By this definition Δ can be defined as

$$\Delta_{s_i} = \Big\{ s_j \mid s_j \in S,\ s_i \text{ and } s_j \text{ are overlapping} \Big\} \qquad \forall s_i \in S$$

where s_i and s_j belongs to different classes.

However, this can be improved. While s_i and s_j have to belong to different classes because a class cannot overlap with itself, two classes with disjoint sets of feasible rooms will never be able to overlap either. Thus, while creating the data, it is possible to prune (or exclude) any s_j from Δ_{s_i} if

$$R_{s_i} \cap R_{s_j} = \emptyset.$$

Ultimately, given a schedule s_i, Δ_{s_i} is the set of schedules that overlap with s_i where it is possible for any of the schedules to be scheduled into the same room as s_i and none of the schedules belong to the same class as s_i.

Declarations

The following declarations correspond to data parsed from an ITC2019 instance file, and output to a Dzn file. Thus, these declarations are a part of the model, and defines what data is available to the model.

The `Classes`, `Schedules` and `Rooms` contains all the classes, schedules and rooms of the instance, respectively.

`ScheduleStarts` and `ScheduleLengths` are arrays containing the `start` time and length (or duration) of each schedule, respectively.

`ClassSchedules` and `ClassRooms` are arrays of sets, indexed by `Classes`, such that each class has a set of feasible `Schedules` in `ClassSchedules`, and a set of feasible `Rooms` in `ClassRooms`.

The `ClassRoomPenalties` and `ClassSchedulePenalties` are 2-dimensional arrays, containing the penalty of each combination of *class* and *room*, and *class* and *schedule* respectively. As an example, if a penalty of 4 is defined if room `R1` is chosen for class `C1` then `ClassRoomPenalties[C1, R1] = 4`.

```
1  enum Classes;
2  enum Schedules;
3  enum Rooms;
4  enum Weeks;
5  enum Days;
6
7  array[Schedules] of int: ScheduleStarts;
8  array[Schedules] of int: ScheduleLengths;
9  array[Schedules] of set of Weeks: ScheduleWeeks;
10 array[Schedules] of set of Days: ScheduleDays;
11 array[Classes] of set of Schedules: ClassSchedules;
12 array[Classes] of set of Rooms: ClassRooms;
13 array[Classes,Rooms] of int: ClassRoomPenalties;
14 array[Classes,Schedules] of int: ClassSchedulePenalties;
```

The most central part of the model is the two arrays of decision variables `ScheduledTime` and `ScheduledRoom`. These denote the chosen schedule and chosen room for each class respectively.

```
1 array[Classes] of var Schedules: ScheduledTime;
2 array[Classes] of var Rooms: ScheduledRoom;
```

Both arrays are indexed by the *classes* of the instance. This tells the solver that for the set of classes C, set of schedules S and set of rooms R, class c is scheduled into room $\mathcal{R}_c$ with schedule $\mathcal{S}_c$:

$$\mathcal{S}_c \in S \quad \forall c \in C \qquad \text{and} \qquad \mathcal{R}_c \in R \quad \forall c \in C.$$

A number of arrays are used for the distribution constraints. These arrays are all declared in a common way, namely as one-dimensional arrays of sets, indexed by `Schedules` and where each set contains a number of `Schedules`. As an example, `ScheduleOverlaps`$[s_i]$ is the set of schedules where each schedule $s_j \in$ `ScheduleOverlaps`$[s_i]$ is overlapping with s_i.

```
1 array[Schedules] of set of Schedules: ScheduleOverlaps;
2 array[Schedules] of set of Schedules: Precedences;
3 array[Schedules] of set of Schedules: SameStarts;
4 array[Schedules] of set of Schedules: SameDays;
5 array[Schedules] of set of Schedules: SameTime;
6 array[Schedules] of set of Schedules: SameWeeks;
```

As described in Section 3.1 distribution constraints defined for an instance can be either *hard* or *soft*. If the distribution constraint is *hard*, the constraint has to be enforced for a solution to be feasible, while if the distribution constraint is *soft*, the solver do not have to enforce the constraint, but should minimize a given penalty. This is modelled as follows.

Each type of distribution constraint defined in the instance-file is split into *soft* constraints and *hard* constraints, and then parsed to MiniZinc in the following form:

```
1 array[int] of set of Classes: OverlapHard;
2 array[int] of set of Classes: OverlapSoft;
3 array[int] of int: OverlapPenalties;
4
5 array[int] of set of Classes: NotOverlapHard;
6 array[int] of set of Classes: NotOverlapSoft;
```

```
7  array[int] of int: NotOverlapPenalties;
8
9  array[int] of set of Classes: PrecedenceHard;
10 array[int] of set of Classes: PrecedenceSoft;
11 array[int] of int: PrecedencePenalties;
12
13 array[int] of set of Classes: SameAttendeesHard;
14 array[int] of set of Classes: SameAttendeesSoft;
15 array[int] of int: SameAttendeesPenalties;
16
17 array[int] of set of Classes: SameStartHard;
18 array[int] of set of Classes: SameStartSoft;
19 array[int] of int: SameStartPenalties;
20
21 array[int] of set of Classes: SameTimeHard;
22 array[int] of set of Classes: SameTimeSoft;
23 array[int] of int: SameTimePenalties;
24
25 array[int] of set of Classes: DiffTimeHard;
26 array[int] of set of Classes: DiffTimeSoft;
27 array[int] of int: DiffTimePenalties;
28
29 array[int] of set of Classes: SameDaysHard;
30 array[int] of set of Classes: SameDaysSoft;
31 array[int] of int: SameDaysPenalties;
32
33 array[int] of set of Classes: DiffDaysHard;
34 array[int] of set of Classes: DiffDaysSoft;
35 array[int] of int: DiffDaysPenalties;
36
37 array[int] of set of Classes: SameWeeksHard;
38 array[int] of set of Classes: SameWeeksSoft;
39 array[int] of int: SameWeeksPenalties;
40
41 array[int] of set of Classes: DiffWeeksHard;
42 array[int] of set of Classes: DiffWeeksSoft;
43 array[int] of int: DiffWeeksPenalties;
44
45 array[int] of set of Classes: SameRoomHard;
46 array[int] of set of Classes: SameRoomSoft;
47 array[int] of int: SameRoomPenalties;
48
49 array[int] of set of Classes: DiffRoomHard;
50 array[int] of set of Classes: DiffRoomSoft;
51 array[int] of int: DiffRoomPenalties;
```

Thus, three arrays is defined for each type of distribution constraint *DistType*.

- *DistType*`Hard` constains the sets of classes where the condition of the distri-

bution constraint *DistType* have to be true for any solution to be feasible.

- *DistType*`Soft` constains the sets of classes where the condition of the distribution constraint is penalized if the condition of the distribution constraint is not fulfilled.
- *DistType*`Penalties` constains a penalty for each set of classes in the *DistType*Soft array.

The maximum possible violation of each variable is calculated when creating the data file, and is included to define the upper-bounds of the variables.

```
int: maxSameStartViol;
int: maxSameTimeViol;
int: maxDiffTimeViol;
int: maxSameDaysViol;
int: maxDiffDaysViol;
int: maxSameWeeksViol;
int: maxDiffWeeksViol;
int: maxSameRoomViol;
int: maxDiffRoomViol;
int: maxOverlapViol;
int: maxNotOverlapViol;
int: maxSameAttendeesViol;
int: maxPrecedenceViol;
int: maxWorkDayViol;
int: maxMinGapViol;
int: maxMaxDaysViol;
int: maxMaxDayLoadViol;
int: maxMaxBreaksViol;
int: maxMaxBlockViol;
```

Thus, the following variable declarations are used for the *soft* distribution constraints, and uses the previously declared upper-bounds for each element of the corresponding array.

```
array[1..length(SameRoomSoft)] of var 0..maxSameRoomViol: SameRoomViol;
array[1..length(DiffRoomSoft)] of var 0..maxDiffRoomViol: DiffRoomViol;
array[1..length(SameAttendeesSoft)] of var 0..maxSameAttendeesViol:
  ↪ SameAttendeesViol;
array[1..length(NotOverlapSoft)] of var 0..maxNotOverlapViol: NotOverlapViol;
array[1..length(OverlapSoft)] of var 0..maxOverlapViol: OverlapViol;
array[1..length(SameTimeSoft)] of var 0..maxSameTimeViol: SameTimeViol;
array[1..length(DiffTimeSoft)] of var 0..maxDiffTimeViol: DiffTimeViol;
array[1..length(SameWeeksSoft)] of var 0..maxSameWeeksViol: SameWeeksViol;
array[1..length(DiffWeeksSoft)] of var 0..maxDiffWeeksViol: DiffWeeksViol;
array[1..length(SameDaysSoft)] of var 0..maxSameDaysViol: SameDaysViol;
```

```
11 array[1..length(DiffDaysSoft)] of var 0..maxDiffDaysViol: DiffDaysViol;
12 array[1..length(SameStartSoft)] of var 0..maxSameStartViol: SameStartViol;
13 array[1..length(PrecedenceSoft)] of var 0..maxPrecedenceViol: PrecedenceViol;
```

This minimizes the size of the domains, and in case a distribution constraint is not used, the size of the domain will be 0, which allows MiniZinc to prune the corresponding array and any constraints dependent on that array during transformation to FlatZinc (or *flattening*).

3.3.2 The Model

As mentioned, the solve stage is handled by MiniZinc which is run on data from the data creation stage using a MiniZinc model. This section will cover the description of the MiniZinc model for the ITC2019 Problem.

Predicates and Constraints

As mentioned in Chapter 1, MiniZinc provides a feature named *predicates*. Predicates are functions with a boolean return type. The model makes use of these predicates to simplify common constraints, and makes it easy to adjust how a constraint is defined without considering the rest of the model. These predicates and their uses will be defined here.

The constraint CS enforces that for every class c, the chosen schedule $\mathcal{S}_c$ must be a member of the set of feasible schedules S_c for class c. This constraint does not make use of any predicates, and is defined as:

$$\mathcal{S}_c \in S_c \qquad \forall c \in C$$

In MiniZinc terms that is

```
1 constraint forall(c in Classes)(
2     ScheduledTime[c] in ClassSchedules[c]
3 );
```

The constraint CR enforces that for every class c, the chosen room $\mathcal{R}_c$ must be a member fo the set of feasible rooms R_c for class c.

That is

$$\mathcal{R}_c \in R_c \qquad \forall c \in C$$

and in MiniZinc terms

```
constraint forall(c in Classes)(
    ScheduledRoom[c] in ClassRooms[c]
);
```

The constraint SO enforces that for every pair of classes c_i, c_j where $R_{c_i} = R_{c_j}$, meaning that c_i and c_j are scheduled into the same room, c_i and c_j may not overlap. For this, the predicate *overlap* is defined. The predicate takes two classes, and check if their chosen schedules overlap by using the `ScheduleOverlaps` array. If O_{s_i} is defined as the set of schedules that overlap with schedule s_i, and $\mathcal{S}_c$ is the chosen schedule for class c then the `overlap` predicate can be defined as:

$$\texttt{overlap}(c_i, c_j) : \mathcal{S}_{c_j} \in O_{\mathcal{S}_{c_i}}$$

which in MiniZinc is:

```
predicate overlap(Classes: c1, Classes: c2) =
    ScheduledTime[c2] in ScheduleOverlaps[ScheduledTime[c1]]
;
```

For the constraint $R_{c_i} \neq None$ is added to this constraint to exclude this constraint for classes which does not need a room, in which case c_i and c_j is allowed to overlap. The MiniZinc constraint loops through every pair of classes $c_i, c_j \in C$, and if they are scheduled into the same room, and the room is not *None* the constraint states that the two classes should not overlap.

```
constraint forall(c1,c2 in Classes where c1<c2 /\ ScheduledRoom[c1] ==
 ↪  ScheduledRoom[c2] /\ ScheduledRoom[c1] != None)(
    not(overlap(c1,c2))
);
```

A minor but critical addition to the constraint is that of `c1<c2`. This adds symmetry breaking to the constraint, such that two classes are only constrained once. Considering a pair of classes c_i, c_j which are both scheduled into the same room, without limiting the constraint to $c_i < c_j$ two constraints would be added:

$$\neg\texttt{overlap}(c_i, c_j)$$
$$\neg\texttt{overlap}(c_j, c_i)$$

However, these constraints are identical. By adding the condition $c_i < c_j$, only one of the above constraints will enforced.

The constraint RU enforces that for any class c, $\mathcal{R}_c$ may not be in the set of occupied rooms for $\mathcal{S}_c$. The set of occupied rooms for a schedule $\mathcal{S}_i$ is the set of rooms where the times where the room is occupied or unavailable, overlap with the schedule $\mathcal{S}_i$.

```
constraint forall(c in Classes)(
    not(ScheduledRoom[c] in RoomUnavailable[ScheduledTime[c]])
);
```

The distribution constraint SR defines that for a set of classes, each pair of classes must be scheduled into the same room. The `same_room` predicate operates on a pair of classes, $ci, cj \in C$, and is defined as follows:

$$\texttt{same\_room}(c_i, c_j) : \mathcal{R}_{c_i} = \mathcal{R}_{c_2}$$

and in MiniZinc terms:

```
predicate same_room(Classes: c1, Classes: c2) =
    ScheduledRoom[c1] == ScheduledRoom[c2]
;
```

To enforce this constraint on each set of classes in `SameRoomHard`, that is, the sets of classes where this has to be enforced for a solution to be feasible, the following constraint is defined:

```
constraint forall(sameRoom in SameRoomHard, c1,c2 in sameRoom where c1<c2)(
    same_room(c1,c2)
);
```

Since a predicate was defined for determining if two classes are scheduled into the same room, using the same predicate, the opposite distribution constraint `DifferentRoom` can be defined as well:

```
constraint forall(diffRoom in DiffRoomHard, c1,c2 in diffRoom where c1<c2)(
    not(same_room(c1,c2))
);
```

The soft variant of the `SameRoom` constraint is defined a bit differently. In this case, the `SameRoom` constraint should only penalize the solution according the the

penalties defined in the `SameRoomPenalties` array. Thus, the previously defined `SameRoomViol` array is used for the violations, such that each element is the number of violation of the set of one soft definition of the `SameRoom` constraint.

```
constraint forall(s in 1..length(SameRoomSoft))(
    SameRoomViol[s] == sum(c1,c2 in SameRoomSoft[s] where c1<c2 /\
     ↪  not(same_room(c1,c2)))(1)
);
```

That is, for every set of classes C in `SameRoomSoft`, the element of `SameRoomViol` is equal to the sum of violations where a violation is 1 for every pair of classes $c_i, c_j \in C$, where c_i and c_j is not in the same room.

Since the predicates and constraints for the distribution constraints P, O, NO, SD, DD, SW, DW, ST, DT and SS are defined in a similar manner, these will not be described in detail, but can be found in the (*Project Repository* n.d.) in file `itc2019/mzn/base.mzn`.

One distribution constraint predicate is defined differently, namely the `SameAttendees` (SA) constraint. Since this constraint is dependent of both details of the chosen *schedule*, and the chosen *room* and distances between rooms, it is necessary to model it more closely to the official definition of the constraint.

The SA constraint states that for a pair of classes c_i, c_j, where c_i and c_j are scheduled to the same days and weeks, there should be enough time to reach the room $\mathcal{R}_{c_i}$ from the room $\mathcal{R}_{c_j}$ *or* vice versa. The travel-time (or distance) is given in the 2-dimensional array `RoomDistances`.

In mathematical terms, for a pair of classes $c_i, c_j \in C$, if $\delta(r_i, r_j)$ denotes the distance d between two rooms $r_i, r_j \in R$, i.e. $\delta : R_i, R_j \rightarrow d$, then the constraint can be defined as follows:

$$\begin{aligned}\neg\texttt{same\_weeks}(c_i, c_j) \ \vee \ \neg\texttt{same\_days}(c_i, c_j) \vee \\ (\mathcal{S}_{c_i}^{start} + \mathcal{S}_{c_i}^{duration} + \delta(\mathcal{R}_{c_i}, \mathcal{R}_{c_j}) \leq \mathcal{S}_{c_j}^{start}) \vee \\ (\mathcal{S}_{c_j}^{start} + \mathcal{S}_{c_j}^{duration} + \delta(\mathcal{R}_{c_j}, \mathcal{R}_{c_i}) \leq \mathcal{S}_{c_i}^{start})\end{aligned}$$

The corresponding predicate is defined in MiniZinc as:

```
predicate same_attendees(Classes: c1, Classes: c2) =
    not(same_weeks(c1,c2)) \/ not(same_days(c1,c2)) \/
```

```
3         ((ScheduleStarts[ScheduledTime[c1]] +
          ↪  ScheduleLengths[ScheduledTime[c1]] +
          ↪  RoomDistances[ScheduledRoom[c1],ScheduledRoom[c2]]) <=
          ↪  ScheduleStarts[ScheduledTime[c2]]) \/
4         ((ScheduleStarts[ScheduledTime[c2]] +
          ↪  ScheduleLengths[ScheduledTime[c2]] +
          ↪  RoomDistances[ScheduledRoom[c2],ScheduledRoom[c1]]) <=
          ↪  ScheduleStarts[ScheduledTime[c1]])
5 ;
```

Objective Function

The objective of the solver is to minimize the objective function which consist of

- Penalties for the chosen room of each class
- Penalties for the chosen schedule of each class
- Penalties for each soft distribution constraint

The objective function, along with the objective, is stated as

```
1  solve :: int_search(ScheduledTime, smallest, indomain_min, complete)
2      minimize sum(c in Classes)(ClassRoomPenalties[c,ScheduledRoom[c]])
3         + sum(c in Classes)(ClassSchedulePenalties[c,ScheduledTime[c]])
4         + sum(s in
           ↪  1..length(SameStartSoft))(SameStartViol[s]*SameStartPenalties[s])
5         + sum(s in
           ↪  1..length(SameTimeSoft))(SameTimeViol[s]*SameTimePenalties[s])
6         + sum(s in
           ↪  1..length(DiffTimeSoft))(DiffTimeViol[s]*DiffTimePenalties[s])
7         + sum(s in
           ↪  1..length(SameDaysSoft))(SameDaysViol[s]*SameDaysPenalties[s])
8         + sum(s in
           ↪  1..length(DiffDaysSoft))(DiffDaysViol[s]*DiffDaysPenalties[s])
9         + sum(s in
           ↪  1..length(SameWeeksSoft))(SameWeeksViol[s]*SameWeeksPenalties[s])
10        + sum(s in
           ↪  1..length(DiffWeeksSoft))(DiffWeeksViol[s]*DiffWeeksPenalties[s])
11        + sum(s in
           ↪  1..length(SameRoomSoft))(SameRoomViol[s]*SameRoomPenalties[s])
12        + sum(s in
           ↪  1..length(DiffRoomSoft))(DiffRoomViol[s]*DiffRoomPenalties[s])
13        + sum(s in 1..length(OverlapSoft))(OverlapViol[s]*OverlapPenalties[s])
14        + sum(s in
           ↪  1..length(NotOverlapSoft))(NotOverlapViol[s]*NotOverlapPenalties[s])
15        + sum(s in
           ↪  1..length(SameAttendeesSoft))(SameAttendeesViol[s]*SameAttendeesPenalties[s])
```

```
16        + sum(s in
           ↪  1..length(PrecedenceSoft))(PrecedenceViol[s]*PrecedencePenalties[s])
```

Mathematically it can be described as follows. Let D_t be the set of distribution types, P_d, the array of penalties for distribution type d and V_d the array of the number of violations of the constraints for distribution type d, then the objective function can be defined as

$$\sum_{c \in C} P_{R_c} + \sum_{c \in C} P_{S_c} + \left(\sum_{i \in d} v_i \cdot p_i \mid d \in D_t,\, p \in P_d,\, v \in V_d \right).$$

Some constraints were not implemented as it was not discovered how these constraints could be modeled in MiniZinc, within the time limit of this thesis. These are the MD, MBR and MBL constraints.

3.4 The FlatZinc Model

Although a MiniZinc model was implemented, FlatZinc gives more flexibility with the drawback of being more complicated to work with. Since it was not possible to create a model that was flexible enough for the purpose of ITC2019 in MiniZinc, it was decided that a model in FlatZinc should be made. This would provide more flexibility in the sense that a model could be created directly from the instance file without relying on data files for a unified MiniZinc model.

This section will present a program for converting instance files of ITC2019 into FlatZinc models, namely `itc2fzn`. `itc2fzn` is written in the programming language Python, and takes an input file, i.e. an instance file in the ITC2019 XML format, and a destination to an output file. `itc2fzn` will start by extracting all data from the input file, then generate a FlatZinc model (and store this in memory), and finally write the model to the output destination file.

In this section, a Python module `fzn`, along with the program `itc2fzn` will be presented along with source code samples and mathematical formulation of the FlatZinc model.

Note that in this section, every *schedule* $\mathcal{S} \in S$ is unique and that each schedule can be a feasible schedule for one or more classes. A description of why this is can be found in Section 3.2.2.

3.4.1 The `fzn` Module

A Python package (or library) named `fzn` was written as a part of `itc2fzn`. This module implements objects which represent FlatZinc variables and constraints and provides capabilities to construct a complete FlatZinc model in Python and write it into a FlatZinc file.

Boolean and integer variables, along with arrays of integers and arrays of boolean values are supported, and a subset of the constraints defined in the FlatZinc definition. The base class defines 4 internal structures:

`introduced_vars` A set of names of variables introduced to the model,

`defined_vars` a set of names of variables defined by constraints,

`variables` a list of variable objects,

`constraints` a list of constraint objects.

The two sets, `introduced_vars` and `defined_vars` are used only to check if a variable is already introduced or defined respectively. The `set` data structure

in Python provides fast search capabilities and does not allow multiple identical elements.

The lists `variables` and `constraints` contain the variable and constraint objects when introduced to the model. These are used for writing to the FlatZinc file, and to store additional information about each variable and constraint.

For every variable object and constraint object a `write` method is implemented, used for writing to a FlatZinc file, and the object stores information for use in this case. The implementation of the integer variable object `IntVar` can be seen in Listing 3.6, and the implementation of the element constraint of the type $r \leftrightarrow x \in S$ (`SetInReif`) is defined in Listing 3.7

Listing 3.6: Implementation of the integer variables (`IntVar`) object

```
class IntVar:
    def __init__(self, name, domain=None, output = False):
        self.name = name
        self.domain = domain
        self.output = output

    def write(self, f):
        if self.output:
            f.write("var {}: {} :: var_is_introduced :: output_var ::
             ↪  is_defined_var;\n".format(self.domain, self.name))
        else:
            f.write("var {}: {} :: var_is_introduced ::
             ↪  is_defined_var;\n".format(self.domain, self.name))
```

Listing 3.7: Implementation of the element constraint (`SetInReif`) object

```
class SetInReif:
    def __init__(self, x, s, r, defines = None):
        self.x = x
        self.s = s
        self.r = r
        self.defines = defines

        if len(s) < 1:
            if type(s) == list:
                s.append(0)
            elif type(s) == set:
                s.add(0)

    def write(self, f):
        if self.defines is not None:
            f.write("constraint set_in_reif({}, {{{}}}, {}) ::
             ↪  defines_var({});\n".format(self.x, ','.join([str(e) for e in
             ↪  self.s]), self.r, self.defines))
```

```
17          else:
18              f.write("constraint set_in_reif({}, {{{}}}, {});\n".format(self.x,
                ↪  ','.join([str(e) for e in self.s]), self.r))
```

The base class `fzn` implements access methods for each variable and constraint, to make it easier to add new constraints or variables to the model. Since it is not valid FlatZinc to have multiple declarations or definitions of the same variable, these access methods check `introduced_vars` and `defined_vars` for duplicates before adding the variables or the constraint to the model.

The implementation of the access methods `fzn.int_var` and `fzn.set_in_reif` is defined in Listing 3.8 and 3.9 respectively.

Listing 3.8: Implementation of the access method for the integer variable object

```
def int_var(self, name, domain = 'int', output = False):
    if domain == 'int':
        print("WARNING: IntVar {} has unbound domain".format(name))
    if name not in self.introduced_vars:
        self.introduced_vars.add(name)
        self.variables.append(IntVar(name, domain, output))
    return name
```

Listing 3.9: Implementation of the access method for the element constraint object

```
def set_in_reif(self, var, arr, r, defines = None):
    if defines is None:
        self.constraints.append(SetInReif(var, arr, r, defines))
    elif defines not in self.defined_vars:
        self.defined_vars.add(defines)
        self.constraints.append(SetInReif(var, arr, r, defines))
    return defines
```

Finally `fzn.write` can be invoked to write the model to a file. This will loop through all variables in `variables` and constraints in `constraints` in that order, executing the `write` method of the object. The order is important since FlatZinc requires variables to be introduced before being defined.

3.4.2 The `itc2fzn` Program

The `itc2fzn` program utilize functionality from the `fzn` module to write a FlatZinc model from a ITC2019 instance file. When run, the input file will be read and processed as described in Section 3.2, and the access methods from the `fzn` module will be used to create the objects that represent the FlatZinc model.

Schedule Penalties (SP)

Each class c has a set of feasible schedules S_c, which is represented by integers. The variables $\mathcal{S}_c$ are introduced to the model for each class as follows:

$$\mathcal{S}_c \qquad \forall c \in C,\, \mathcal{S}_c \in S_c \tag{3.19}$$

Each feasible schedule $s \in S_c$ for a given class c have an associated penalty p, denoted p_{cs}, meaning that the objective function is penalized by p_{cs} if schedule s is chosen for class c. The total penalty for all classes and the schedules chosen for these classes is modeled by a linear equation, where $\mathcal{S}_{cs}$ is 1 if schedule s is chosen for class c, and 0 otherwise.

$$\text{SchedulePenalty} = \sum_{c \in C} \sum_{s \in S_c} p_{cs} \cdot \mathcal{S}_{cs} \tag{3.20}$$

The writing of these variables and the `SchedulePenalty` is handled by the `class_schedules` function as seen in Listing 3.10.

```
1  def class_schedules(self):
2      ...
3      for c in self.dat.classes:
4          self.fzn.int_var(self.dat.classes[c].schedule_var,
5                           output=True,
6                           domain="{{{}}}".format(','.join([
7                               str(self.dat.unique_schedules[s].id)
8                               for s in self.dat.classes[c].schedules
9                           ])))
10
11     if self.include_soft_vars:
12         penalties = []
13         penalties.append(-1)
14         svars = []
15         svars.append("SchedulePenalty")
16         ub = 0
17         for c in self.dat.classes:
18             for s in self.dat.classes[c].schedules:
19                 sched1 = self.dat.unique_schedules[s]
20                 c1schedule1 = self.fzn.bool_var("C{}Schedule{}".format(
21                     c, sched1.id))
22                 self.fzn.int_eq_reif("C{0}_Schedule".format(c),
23                                      sched1.id,
24                                      c1schedule1,
```

```
25                                      defines=c1schedule1,
26                                      name="SchedulePenalty")
27                  c1schedule1int = self.fzn.int_var(
28                      "C{}Schedule{}int".format(c, sched1.id), domain='0..1')
29                  self.fzn.bool2int(c1schedule1,
30                                    c1schedule1int,
31                                    defines=c1schedule1int,
32                                    name="SchedulePenalty")
33                  penalties.append(self.dat.classes[c].schedule_penalty[s])
34                  svars.append(c1schedule1int)
35                  ub += self.dat.classes[c].schedule_penalty[s]
36
37          self.fzn.int_var("SchedulePenalty",
38                           domain="0..{}".format(ub),
39                           output=True)
40          self.fzn.int_lin_eq(penalties, # Linear Equation for SchedulePenalty
41                              svars,
42                              0,
43                              defines="SchedulePenalty",
44                              name="SchedulePenalty")
45      ...
```

Listing 3.10: Implementation of the FlatZinc generation of Schedule Penalties (SP)

As seen in line 40 the linear equation constraint object is created in the model for calculating the `SchedulePenalty`. However, the linear equation is defined as:

$$0 = \Big(\sum_{c \in C} \sum_{s \in S_c} p_{cs} \cdot \mathcal{S}_{cs} \Big) - 1 \cdot \text{SchedulePenalty}$$

rather than the definition in Equation 3.20. The reason for this is the definition of `int_lin_eq` in FlatZinc:

```
predicate int_lin_eq(array [int] of int: a,
                     array [int] of var int: b,
                     int c)
```

which constrains $c = \sum_i a_i \cdot b_i$. However, c is not allowed to be a variable, thus c is set to 0, and the variable `SchedulePenalty` is instead added to the array of variables b.

Since RP is defined in a similar way, this will not be described any further.

Room Unavailabilities (RU)

The RU constraint enforces that a class cannot be scheduled into a room which is declared unavailable at the time where the class takes place. During parsing of

an ITC2019 instance, every room is associated with a set of schedules U where the room is *unavailable*. The constraint can be enforced in the FlatZinc model by looping through every schedule $s \in S$, and every schedule $u \in U$ and check if these schedules overlap. Two schedules s, u overlap if and only if the set of weeks, days and time of the day overlap. That is

$$
\begin{aligned}
&(s.\text{weeks } \texttt{AND } u.\text{weeks}) > 0 \land \\
&(s.\text{days } \texttt{AND } u.\text{days}) > 0 \land \\
&(s.\text{end} > u.\text{start} \land s.\text{start} \leq u.\text{start} \ \lor \ u.\text{end} > s.\text{start} \land u.\text{start} \leq s.\text{start} \lor \\
&\ u.\text{start} \leq s.\text{start} \land u.\text{end} > s.\text{end} \ \lor \ s.\text{start} \leq u.\text{start} \land s.\text{end} > u.\text{end})
\end{aligned}
$$

where `AND` is the binary "and".

In case this condition is true, a constraint is introduced to the model for every class c where room r is in the set of feasible rooms for class c, and schedule s is in the set of feasible schedules for class c, that is $r \in R_c$ and $s \in S_c$. The constraint is defined as follows, where schedule $u \in U_r$ for room r overlap with schedule s:

$$
\mathcal{S}_c \neq s \lor \mathcal{R}_c \neq r \qquad \forall c \in C \text{ where } r \in R_c,\ s \in S_c,\ u \in U_r \text{ where } u \text{ and } s \text{ overlap}
$$

The generation of FlatZinc can be seen in Listing 3.11.

```
1  def room_unavailable(self):
2      ...
3      for s in self.dat.unique_schedules:
4          sched = self.dat.unique_schedules[s]
5          for room in self.dat.rooms:
6
7              # Find out if room is available for schedule
8              available = True
9              for room_unavail in self.dat.rooms[room]['unavailable']:
10                 ru_end = room_unavail['start'] + room_unavail['length']
11
12                 # Week overlap
13                 if (sched.weeks & room_unavail['weeks']).any():
14                     # Day overlap
15                     if (sched.days & room_unavail['days']).any():
16                         # Time overlap
17                         if (sched.end > room_unavail['start'] and sched.start
                               ↪  <= room_unavail['start']) \
18                                 or (ru_end > sched.start and
                                       ↪  room_unavail['start'] <= sched.start) \
```

```
19                                  or (room_unavail['start'] <= sched.start and
                                     ↪  ru_end > sched.end) \
20                                  or (sched.start <= room_unavail['start'] and
                                     ↪  sched.end > ru_end):
21                              available = False
22                              break
23
24              # Prevent every class (with that room and schedule) from using that
                 ↪  combination
25              if not available:
26                  for c in self.dat.classes:
27                      if s in self.dat.classes[
28                              c].schedules and room in self.dat.classes[
29                                  c].rooms:
30                          cnotroomr = self.fzn.bool_var(
31                              "C{}NotRoom{}".format(c, room))
32                          cnotscheds = self.fzn.bool_var(
33                              "C{}NotSchedule{}".format(c, sched.id))
34                          self.fzn.int_ne_reif("C{}_Room".format(c),
35                                               room,
36                                               cnotroomr,
37                                               defines=cnotroomr,
38                                               name="RoomUnavailabilities")
39                          self.fzn.int_ne_reif("C{}_Schedule".format(c),
40                                               sched.id,
41                                               cnotscheds,
42                                               defines=cnotscheds,
43                                               name="RoomUnavailabilities")
44
45                          self.fzn.array_bool_or([cnotscheds, cnotroomr],
46                                                 True,
                                                    ↪  name="RoomUnavailabilities")
47                  i += 1
48      ...
```

Listing 3.11: Implementation of the FlatZinc generation for the RU constraint.

Schedule Overlaps (SO)

The SO constraint enforce that if two classes c_i, c_j are scheduled into the same room, their schedules $\mathcal{S}_i, \mathcal{S}_j$ cannot overlap at any point. For the generation of FlatZinc to enforce this constraint, it is checked if c_i, c_j have any feasible *rooms* in common, and that at least one of the classes is not *fixed*. As previously mentioned a *fixed* class is a class with only one feasible schedule, and none or one feasible rooms. If both classes c_i, c_j are fixed, there is no possible way for the solver to prevent an overlap, thus they should not overlap per the input data.

In case at least one of the classes c_i, c_j is not fixed, and the classes share at least one feasible room the following constraint will be written for each schedules s_i, s_j where s_i, s_j overlap and $s_i \in S_{c_i}$ and $s_j \in S_{c_j}$:

$$\mathcal{S}_{c_i} \neq s_i \vee \mathcal{S}_{c_j} \neq s_j \vee \mathcal{R}_{c_i} \neq \mathcal{R}_{c_j} \qquad \forall s_i \in S_{c_i},\ s_j \in S_{c_j},\ c_i, c_j \in C$$

The implementation of the generation of these constraints can be seen in Listing 3.12.

```
def schedule_overlaps(self):
    ...
    for c1, c2 in itertools.combinations(self.dat.classes, 2):
        if len(self.dat.classes[c1].rooms
               & self.dat.classes[c2].rooms) > 0:
            if not self.dat.classes[c1].is_fixed(
            ) and not self.dat.classes[c2].is_fixed():
                for s1, s2 in itertools.product(
                        self.dat.classes[c1].schedules,
                        self.dat.classes[c2].schedules):
                    if self.dat.schedule_overlaps[s1][s2]:
                        sched1 = self.dat.unique_schedules[s1]
                        sched2 = self.dat.unique_schedules[s2]
                        c1c2diffroom = self.fzn.bool_var(
                            "C{}C{}DiffRoom".format(c1, c2))
                        self.fzn.int_ne_reif("C{}_Room".format(c1),
                                             "C{}_Room".format(c2),
                                             c1c2diffroom,
                                             defines=c1c2diffroom,
                                             name="ScheduleOverlaps")
                        c1notsched1 = self.fzn.bool_var(
                            "C{}NotSchedule{}".format(c1, sched1.id))
                        c2notsched2 = self.fzn.bool_var(
                            "C{}NotSchedule{}".format(c2, sched2.id))

                        self.fzn.int_ne_reif("C{}_Schedule".format(c1),
                                             sched1.id,
                                             c1notsched1,
                                             defines=c1notsched1,
                                             name="ScheduleOverlaps")
                        self.fzn.int_ne_reif("C{}_Schedule".format(c2),
                                             sched2.id,
                                             c2notsched2,
                                             defines=c2notsched2,
                                             name="ScheduleOverlaps")

                        self.fzn.array_bool_or(
```

```
38                              [c1notsched1, c2notsched2, c1c2diffroom],
39                              True,
40                              name="ScheduleOverlaps")
```

Listing 3.12: Implementation of the FlatZinc generation for the SO constraint.

Student Conflicts (SC)

A student conflict occurs when a student have to attend two classes at the same time, or if there is not enough time to travel between rooms of the classes that the student have to attend. This should not prevent a feasible solution, but should be minimized. As described in Section 3.2.1 students are sectioned when parsing the instance data, so the only job of the solver is to minimize student conflicts.

A student conflict can be described mathematically for two classes $c_i, c_j \in C$ that a student attends, where $s_i \in S_{c_i}$, $s_j \in S_{c_j}$, $r_i \in R_{c_i}$, $r_j \in R_{c_j}$ and where $d_{r_i r_j}$ is the distance between room r_i and r_j. Considering a condition $\mathscr{C}$,

$$\begin{aligned} \mathscr{C}_{s_i s_j r_i r_j} \leftrightarrow\ & (s_i.\text{end} + d_{r_i r_j} \leq s_j.\text{start}) \vee \\ & (s_j.\text{end} + d_{r_i r_j} \leq s_i.\text{start}) \vee \\ & (s_i.\text{weeks}\ \texttt{AND}\ s_j.\text{weeks}) = 0\ \vee\ (s_i.\text{days}\ \texttt{AND}\ s_j.\text{days}) = 0 \end{aligned} \tag{3.21}$$

if $\mathscr{C}_{s_i s_j r_i r_j}$ does not hold, the classes c_i and c_j cause a student conflict if $\mathcal{S}_{c_i} = s_i$, $\mathcal{S}_{c_j} = s_j$, $\mathcal{R}_{c_i} = r_i$ and $\mathcal{R}_{c_j} = r_j$. This means that if $\mathscr{C}_{s_i s_j r_i r_j}$ does not hold, if the combinations of schedules and rooms are chosen for classes c_i, c_j, a penalty should be added to the objective function. Thus, for each n cases where the condition $\mathscr{C}_{s_i s_j r_i r_j}$ does not hold, a reified constraint is introduced to the FlatZinc model:

$$\text{conflict}_{c_i c_j k} \leftrightarrow \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} = s_j \wedge \mathcal{R}_{c_i} = r_i \wedge \mathcal{R}_{c_j} = r_j \qquad \forall k \in 1, \ldots, n$$

for every student $\mathscr{S}$ and every combination of classes $c_i, c_j \in C_{\mathscr{S}}$ that $\mathscr{S}$ attends, and for every combination of s_i, s_j, r_i and r_j.

The total penalty to be imposed on the objective function can then be calculated using a linear equation, where p is the defined penalty for student conflicts for the instance:

$$\sum_{c_i, c_j \in C_{\mathscr{S}}, k \in \{1,\ldots,n\}} p \cdot \text{conflict}_{c_i c_j k} \tag{3.22}$$

summed over all students $\mathscr{S}$.

The number of constraints to be created in the worst case, namely the number of reifications of $\text{conflict}_{c_i c_j k}$ can be described, where $\mathfrak{S}$ is the set of students, C is the set of classes, S is the set of schedules and R is the set of rooms, as:

$$|\mathfrak{S}| \times \binom{C}{2} \times |S|^2 \times |R|^2$$

As the generation of FlatZinc constraints for minimization of student conflicts in the way described here is very costly in terms of memory and performance, the generation of constraints for student conflicts was not included for any of the tests described later in this thesis.

SameStart (SS)

The SS distribution constraint enforces every class in a set of classes C to start at the same time, independent of the day and week of each class.

For every pair of classes c_i,c_j in C and for the cartesian product of the set of feasible schedule of c_i and c_j respectively, if $\mathcal{S}_i.start = \mathcal{S}_j.start$, $\mathcal{S}_i, \mathcal{S}_j$ is a feasible combination of schedules for c_i and c_j respectively. Thus the constraint can be expressed as

$$\bigvee_{s_i,s_j \in S, c_i,c_j \in C} \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} = s_j \tag{3.23}$$

$$\text{where} \quad S = \Big\{ (s_i, s_j) \mid s_i \in S_{c_i},\ s_j \in S_{c_j},\ s_i.start = s_j.start \Big\}.$$

The code for enforcing this constraint can be seen in Listing 3.13.

```
1  def same_start_hard(self, classes, fzn, dat):
2      for c1,c2 in itertools.combinations(classes, 2):
3          pos_var = []
4          i = 1
5          for s1,s2 in itertools.product(dat.classes[c1].schedules,
           ↪  dat.classes[c2].schedules):
6              sched1 = dat.unique_schedules[s1]
7              sched2 = dat.unique_schedules[s2]
8
9              if sched1.start == sched2.start:
10                 c1c2_samestart =
                   ↪  fzn.bool_var("C{0}C{1}SameStart{2}".format(c1,c2,i))
```

```
                fzn.array_bool_and(
                    [
                        "C{}Schedule{}".format(c1,sched1.id),
                        "C{}Schedule{}".format(c2,sched2.id)
                    ],
                    c1c2_samestart
                )
                pos_var.append(c1c2_samestart)
                i += 1

        if len(pos_var)>0:
            fzn.array_bool_or(pos_var, True)
```

Listing 3.13: Implementation of the SS (hard) constraint.

The soft variant of the SS constraint adds a penalty p to the objective function for every pair of classes in a set of classes C that does not start at the same time. Thus, this can be modelled by reification of the negation of the hard variant of the SS constraint, and a linear equation with the penalties p as the coefficients, and the reified values as the variables. Thus, the reified values are introduced as follows:

$$\text{DifferentStart}_n \leftrightarrow \bigvee_{s_i,s_j \in S, c_i,c_j \in C} \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} = s_j \tag{3.24}$$
$$\text{where} \quad S = \left\{ (s_i, s_j) \mid s_i \in \mathbb{S}_{c_i},\ s_j \in \mathbb{S}_{c_j},\ s_i.start \neq s_j.start \right\}$$

where n is a unique identifier.

Every N soft SS constraint is defined like this, and the DifferentStart_n variable is saved along with the penalty p_n of the constraint.

When every soft SS constraint have been parsed, the total penalty of the constraint can be defined as:

$$\text{SameStartPenalty} = \sum_{n=1}^{N} p_n \, \text{DifferentStart}_n \tag{3.25}$$

The source code for handling a soft SS constraint, can be seen in Listing 3.14.

```
def same_start_soft(self, classes, fzn, dat, penalty):
    for c1, c2 in itertools.combinations(classes, 2):
        pos_var = []
```

```
        i = 1
        for s1, s2 in itertools.product(dat.classes[c1].schedules,
                                        dat.classes[c2].schedules):
            sched1 = dat.unique_schedules[s1]
            sched2 = dat.unique_schedules[s2]

            if sched1.start != sched2.start:
                c1c2_diffstarti = fzn.bool_var(
                    "C{0}C{1}DiffStart{2}".format(c1, c2, i))
                fzn.array_bool_and([
                    "C{}Schedule{}".format(c1, sched1.id),
                    "C{}Schedule{}".format(c2, sched2.id)
                ],
                                   c1c2_diffstarti,
                                   defines=c1c2_diffstarti,
                                   name="SameStart_S")
                pos_var.append(c1c2_diffstarti)
                i += 1

        if len(pos_var) > 0:
            c1c2_diffstart = fzn.bool_var("C{}C{}DiffStart".format(c1, c2))
            fzn.array_bool_or(pos_var, c1c2_diffstart, name="SameStart_S")

            c1c2_diffstart_int = fzn.int_var("C{}C{}DiffStart_int".format(
                c1, c2),
                                             domain="0..1")
            fzn.bool2int(c1c2_diffstart,
                         c1c2_diffstart_int,
                         defines=c1c2_diffstart_int,
                         name="SameStart_S")
            self.soft_vars.append(c1c2_diffstart_int)
            self.penalties.append(penalty)
```

Listing 3.14: Implementation of FlatZinc generation for the SS (soft) constraint.

Since FlatZinc generation for multiple of the distribution constraints are defined using this technique, they will not be described in detail here. For those constraints it is a matter of redefining S to the definition of the distribution constraint as defined in Section 3.1. These constraints are, ST, DT, O and NO and the source code the generation of these constraints can be found in the (*Project Repository* n.d.), file `itc2019/src/fzn/constraint.py`.

SameDays (SD)

The SD distribution constraint is defined such that for a set of given classes, these classes must be taught on the same days. For a pair of classes c_i, c_j where class c_i is taught more days per week than class c_j, the days per week of c_j should be a

subset of c_j and vice versa.

For every pair of classes $c_i, c_j \in C$ for a SD constraint, and for the cartesian product of the set of feasible schedules $\mathcal{S}_{c_i}, \mathcal{S}_{c_j}$ of class c_i and c_j respectively, if:

$$\Big(\mathcal{S}_{c_i}.\text{days} \cap \mathcal{S}_{c_j}.\text{days} \neq \mathcal{S}_{c_i}.\text{days}\Big) \wedge \Big(\mathcal{S}_{c_i}.\text{days} \cap \mathcal{S}_{c_j}.\text{days} \neq \mathcal{S}_{c_j}.\text{days}\Big) \tag{3.26}$$

the SD constraint does not hold for this combination of schedules $\mathcal{S}_{c_i}, \mathcal{S}_{c_j}$. If this is the case, the following constraint is introduced in the model:

$$\neg\mathcal{S}_{c_i} \vee \neg\mathcal{S}_{c_j} \tag{3.27}$$

This prevents any pair of classes in the distribution constraint from being scheduled such that the constraint does not hold. The implementation of this can be seen in Listing 3.15.

```
def same_days_hard(self, classes, fzn, dat):
    for c1, c2 in itertools.combinations(classes, 2):
        for s1, s2 in itertools.product(dat.classes[c1].schedules,
                                        dat.classes[c2].schedules):
            sched1 = dat.unique_schedules[s1]
            sched2 = dat.unique_schedules[s2]

            if ((sched1.days | sched2.days) != sched1.days) and (
                (sched1.days | sched2.days) != sched2.days):
                c1notsched1 = fzn.bool_var("C{}NotSchedule{}".format(
                    c1, sched1.id))
                c2notsched2 = fzn.bool_var("C{}NotSchedule{}".format(
                    c2, sched2.id))
                fzn.int_ne_reif("C{}_Schedule".format(c1),
                                sched1.id,
                                c1notsched1,
                                defines=c1notsched1,
                                name="SameDays_H")
                fzn.int_ne_reif("C{}_Schedule".format(c2),
                                sched2.id,
                                c2notsched2,
                                defines=c2notsched2,
                                name="SameDays_H")

                fzn.array_bool_or([c1notsched1, c2notsched2],
                                  True,
                                  name="SameDays_H")
```

Listing 3.15: Implementation of the generation of FlatZinc for the SD hard distribution constraint

The soft variant of the SD constraint can be modeled by using the same condition as defined in Equation (3.26) to find the number of violations of the SD constraint.

For each pair $c_i, c_j \in C$ of classes in a soft SD constraint with penalty p, a set $\text{DifferentDays}_{c_i c_j s_i}$ is introduced for each schedule $s_i \in S_{c_i}$, that contains all schedules $s_j \in S_{c_j}$ where Equation 3.26 holds for s_i, s_j. That is:

$$\begin{aligned}\text{DifferentDays}_{c_i c_j s_i} &= \left\{ s_j \mid c_j \in C, D_{c_i s_i c_j s_j} \right\} \quad \forall c_i \in C,\ s_i \in S_{c_i} \\ \text{where } D_{c_i s_i c_j s_j} &= \Big(s_i.\text{days} \cap s_j.\text{days} \neq s_i.\text{days}\Big) \wedge \Big(s_i.\text{days} \cap s_j.\text{days} \neq s_j.\text{days}\Big)\end{aligned}$$

A violation of the soft variant of the SD occurs for the pair of classes, if the chosen schedule for c_j, $\mathcal{S}_{c_j}$ is in the set of schedules $\text{DifferentDays}_{c_i c_j \mathcal{S}_i}$, that is:

$$\mathcal{S}_{c_i} \wedge \mathcal{S}_{c_j} \in \text{DifferentDays}_{c_i c_j \mathcal{S}_{c_i}}$$

Therefore, the total penalty for all n soft SD constraints where C_k is the set of classes for SD constraint k, can be defined as:

$$\text{SameDaysPenalty} = \sum_{k=1}^{n} p_k \cdot \Big(\sum_{c_i, c_j \in C_k,\, s_i \in S_{c_i}} \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} \in \text{DifferentDays}_{c_i c_j s_i} \Big)$$

This linear equation impose a penalty on the objective function. The FlatZinc generation of intermediate constraints that defines the number of violations can be seen in Listing 3.16.

```
1 def same_days_soft(self, classes, fzn, dat, penalty):
2     for c1, c2 in itertools.combinations(classes, 2):
3         for s1 in dat.classes[c1].schedules:
4             diffdays_schedules = []
5             sched1 = dat.unique_schedules[s1]
6             for s2 in dat.classes[c2].schedules:
7                 sched2 = dat.unique_schedules[s2]
8                 if not (((sched1.days | sched2.days) == sched1.days) or
9                         ((sched1.days | sched2.days) == sched2.days)):
```

```
                diffdays_schedules.append(sched2.id)

        c1s1c2_diffdays = fzn.bool_var("C{}DiffDays{}S{}".format(
            c2, c1, sched1.id))
        c2sched_diffdaysc1s1 = fzn.bool_var(
            "C{}ScheduleDiffDaysC{}S{}".format(c2, c1, sched1.id))
        c1schedule1 = fzn.bool_var("C{}Schedule{}".format(
            c1, sched1.id))
        fzn.set_in_reif("C{}_Schedule".format(c2),
                        diffdays_schedules,
                        c2sched_diffdaysc1s1,
                        defines=c2sched_diffdaysc1s1,
                        name="SameDays_S")
        fzn.array_bool_and([c1schedule1, c2sched_diffdaysc1s1],
                           c1s1c2_diffdays,
                           defines=c1s1c2_diffdays,
                           name="SameDays_S")

        self.soft_vars.append(c1s1c2_diffdays)
        self.penalties.append(penalty)
```

Listing 3.16: Implementation of the FlatZinc generation for the soft SD constraint.

The generation of FlatZinc for DD, SW and DW is implemented in a similar manner, and will not be described here. The source code for the generation of FlatZinc for these constraints can be found in (*Project Repository* n.d.) in file `itc2019/src/fzn/constraint.py`.

SameRoom (SR)

The FlatZinc generation for the SR constraint is the simplest of the constraints. For every pair of classes in the constraint, the chosen room should be the same. That is:

$$\mathcal{R}_{c_i} = \mathcal{R}_{c_j} \qquad \forall c_i, c_j \in C$$

The code for the generation of the FlatZinc is defined as follows, where `classes` is the set of classes of a SR constraint:

```
def same_room_hard(self, classes, fzn):
    for c1,c2 in itertools.combinations(classes, 2):
        fzn.int_eq("C{}_Room".format(c1), "C{}_Room".format(c2))
```

For the soft variant of the SR constraint, the value of the objective function should be penalized if two classes c_i, c_j of the constraint is not scheduled into the same

room. The number of pairs of classes that violates the condition for a set of classes C_k can be defined as the number of cases where c_i and c_j are scheduled into different rooms. That is:

$$\sum_{c_i,c_j \in C_k} \mathcal{R}_{c_i} \neq \mathcal{R}_{c_j}$$

Thus, if p_k is the penalty of a soft SR constraint k for n soft SR constraints, then the penalty to be imposed on the objective function can be defined as:

$$\text{SameRoomPenalty} = \sum_{k=1}^{n} p_k \cdot \sum_{c_i,c_j \in C_k} \mathcal{R}_{c_i} \neq \mathcal{R}_{c_j}$$

For the generation of the FlatZinc for the violations of soft variants of the SR constraint, the source code is as follows:

```
def same_room_soft(self, classes, fzn, penalty):
    for c1,c2 in itertools.combinations(classes, 2):
        diffroom = fzn.bool_var("DiffRoomC{0}C{1}".format(c1,c2))
        fzn.int_ne_reif("C{}_Room".format(c1), "C{}_Room".format(c2), diffroom,
         ↪  defines=diffroom)

        diffroom_int = fzn.int_var("DiffRoomC{0}C{1}_int".format(c1,c2),
         ↪  domain="0..1")
        fzn.bool2int(diffroom, diffroom_int, defines = diffroom_int)

        self.soft_vars.append(diffroom_int)
        self.penalties.append(penalty)
```

Since the DR constraint is defined similarly to the SR constraint, this will not be described any further. The source code for the DR constraint can be found in the (*Project Repository* n.d.) in file `itc2019/src/fzn/constraint.py`.

SameAttendees (SA)

The SA constraint enforce that for a set of classes C_k, every pair of classes $c_i, c_j \in C_k$ is scheduled at times and in rooms such that it is possible to attend both classes, taking into account the distance between the rooms chosen for the classes.

For a pair of feasible rooms for the classes c_i, c_j, $r_i \in R_{c_i}$ and $r_j \in R_{c_j}$ and for a pair of feasible schedules $s_i \in S_{c_i}$ and $s_j \in S_{c_j}$, if the distance between r_i and r_j is denoted $d_{r_i r_j}$, the constraint holds for s_i, s_j, r_i and r_j if:

$$\begin{aligned}\mathscr{C}_{s_i s_j r_i r_j} \leftrightarrow\ & (s_i.\text{end} + d_{r_i r_j} \leq s_j.\text{start}) \vee \\ & (s_j.\text{end} + d_{r_i r_j} \leq s_i.\text{start}) \vee \\ & (s_i.\text{weeks}\ \texttt{AND}\ s_j.\text{weeks}) = 0\ \vee\ (s_i.\text{days}\ \texttt{AND}\ s_j.\text{days}) = 0\end{aligned}$$

To avoid including the notion of weeks and days in the model, the condition $\mathscr{C}$ can be used to model cases for classes c_i, c_j when the condition does *not* hold for a combination of s_i, s_j, r_i and r_j. Thus for every case where $\mathscr{C}_{s_i s_j r_i r_j}$ does not hold, the following reified constraint is added to the model:

$$\begin{aligned}&\text{NotSameAttendees}_{c_i c_j k} \leftrightarrow \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} = s_j \wedge \mathcal{R}_{c_i} = r_i \wedge \mathcal{R}_{c_j} = r_j \\ &\forall s_i \in S_{c_i},\ s_j \in S_{c_j},\ r_i \in R_{c_i},\ r_j \in R_{c_j} \text{ where } \neg\mathscr{C}_{s_i s_j r_i r_j} \quad c_i, c_j \in C_k\end{aligned}$$

For n hard SA distribution constraints, the SA constraint can then be enforced by preventing all found cases where $\mathscr{C}$ does not hold. That is:

$$\neg \bigvee \text{NotSameAttendees}_{c_i c_j k} \qquad \forall c_i, c_j \in C_k,\ k \in \{1, 2, \ldots, n\}$$

Another way this could be modeled would be to instead define a reified constraint $\text{SameAttendees}_{c_i c_j k}$, and then enforce

$$\bigvee \text{SameAttendees}_{c_i c_j k} \qquad \forall c_i, c_j \in C_k,\ k \in \{1, 2, \ldots, n\}$$

However, initial tests showed that by using the negation, less constraints were introduced in the FlatZinc model in most cases, meaning that the number of cases for two classes where the constraint does not hold is less than cases where the constraint holds. The source code for the generation of FlatZinc for the hard variant of the SameAttendees constraint can be seen in Listing 3.17.

```
1 def same_attendees_hard(self, classes, fzn, dat):
2     for c1, c2 in itertools.combinations(classes, 2):
3         pos_var = []
4         i = 1
5
6         for r1, r2 in itertools.product(dat.classes[c1].rooms,
7                                         dat.classes[c2].rooms):
8             c1room1 = fzn.bool_var("C{0}Room{1}".format(c1, r1))
```

```
            c2room2 = fzn.bool_var("C{0}Room{1}".format(c2, r2))
            fzn.int_eq_reif("C{}_Room".format(c1),
                            r1,
                            c1room1,
                            defines=c1room1,
                            name="SameAttendees_H")
            fzn.int_eq_reif("C{}_Room".format(c2),
                            r2,
                            c2room2,
                            defines=c2room2,
                            name="SameAttendees_H")

            for s1, s2 in itertools.product(dat.classes[c1].schedules,
                                            dat.classes[c2].schedules):
                sched1 = dat.unique_schedules[s1]
                sched2 = dat.unique_schedules[s2]

                if (sched1.end + dat.rooms[r1]['travel'][r2] <=
                        sched2.start
                    ) or (sched2.end + dat.rooms[r2]['travel'][r1] <=
                          sched1.start) or (
                              not (sched1.weeks & sched2.weeks).any()) or (
                                  not (sched1.days & sched2.days).any()):
                    pass
                else:
                    fzn.bool_var("C{0}C{1}SameAttendees{2}".format(
                        c1, c2, i))
                    fzn.array_bool_and(
                        [
                            "C{}Schedule{}".format(
                                c1, sched1.id), "C{}Schedule{}".format(
                                    c2, sched2.id), c1room1, c2room2
                        ],
                        "C{}C{}SameAttendees{}".format(c1, c2, i),
                        defines="C{}C{}SameAttendees{}".format(c1, c2, i),
                        name="SameAttendees_H")
                    pos_var.append("C{0}C{1}SameAttendees{2}".format(
                        c1, c2, i))
                    i += 1

        if len(pos_var) > 0:
            fzn.array_bool_or(pos_var, False, name="SameAttendees_H")
```

Listing 3.17: Implementation of FlatZinc generation for the hard SA constraint.

For the soft variant of the SA constraint, the same principles as described in this section was used. Thus this will not be described any further. The source code for the soft variant of the SA constraint can be found in the (*Project Repository* n.d.)

file `itc2019/src/fzn/constraint.py`.

Precedence (P)

The distribution constraint P states that for a ordered set of classes C_k, every ordered pair $c_i, c_j \in C_k$, the class c_i must be scheduled *before* class c_j. In a case where c_i is scheduled before c_j, c_i is said to *precede* c_j, which is equivalent of stating that c_j *succeeds* c_i.

A class c_j with schedule s_j succeeds c_i with schedule s_i if the first week of s_j is after the first week of s_i, or in case of the first week of s_i and s_j is the same week, the first day of s_j is after the first day of s_i or if the first week and first day of s_i and s_j is the same, s_j starts after s_i ends. That is:

$$
\begin{aligned}
&(\text{first}(s_i.weeks) < \text{first}(s_j.weeks)) \vee \\
&\quad \Big[(\text{first}(s_i.weeks) = \text{first}(s_j.weeks)) \wedge \\
&\qquad \Big[(\text{first}(s_i.days) < \text{first}(s_j.days)) \vee \\
&\qquad\quad ((\text{first}(s_i.days) = \text{first}(s_j.days)) \wedge (s_i.end \leq s_j.start)) \\
&\qquad \Big] \\
&\quad \Big]
\end{aligned}
$$

For the generation of the FlatZinc for constraint P for a set of classes C_k, for every pair of classes $c_i, c_j \in C_k$, for a schedule $s_i \in S_{c_i}$ a set of schedules can be defined as a subset of $s_j \in S_{c_j}$ where all schedules in this set is schedules that succeed s_i. That is:

$$
\text{succeeding}_{s_i} = \Big\{ s_j \mid s_j \in S_{c_j} \text{ where } s_j \text{ succeeds } s_i \Big\} \quad \forall s_i \in S_{c_i},\ c_i, c_j \in C_k
$$

This results in an array of sets, generated as follows:

```
succeeding = defaultdict(set)
for c1, c2 in itertools.combinations(dist['classes'], 2):
    for s1 in self.dat.classes[c1].schedules:
        sched1 = self.dat.unique_schedules[s1]
        for s2 in self.dat.classes[c2].schedules:
            sched2 = self.dat.unique_schedules[s2]

            # succeed by week
```

```
            if sched1.first_week() < sched2.first_week():
                succeeding[sched1.id].add(sched2.id)
            elif sched1.first_week() == sched2.first_week():
                # succeed by day
                if sched1.first_day() < sched2.first_day():
                    succeeding[sched1.id].add(sched2.id)
                elif sched1.first_day() == sched2.first_day():
                    # succeed by time
                    if sched1.end <= sched2.start:
                        succeeding[sched1.id].add(sched2.id)
```

This array of sets can be used to generate the constraints for the FlatZinc model by the use of element constraints. For every pair of classes c_i, c_j, if both classes are not fixed, then for every schedule $s_i \in S_{c_i}$, the schedule of c_j, $\mathcal{S}_{c_j}$ should be in the set of succeeding schedules of s_i. This gives a total of $|S_{c_i}|$ constraints for each pair of classes $c_i, c_j \in C_k$, where at least one of them has to be true for the constraint P to hold. That is:

$$\bigvee_{s_i \in S_{c_i}} \mathcal{S}_{c_i} = s_i \wedge \mathcal{S}_{c_j} \in \text{succeeds}_{s_i} \qquad \forall c_i, c_j \in C_k$$

This is implemented as follows in `itc2fzn`:

```
def precedence_hard(self, classes, fzn, dat, succeeding):
    for c1, c2 in itertools.combinations(classes, 2):
        pos = []
        i = 1
        if len(dat.classes[c1].schedules) > 1 or len(
                dat.classes[c2].schedules) > 1:
            for s1 in dat.classes[c1].schedules:
                sched1 = dat.unique_schedules[s1]

                c1schedule1 = fzn.bool_var("C{}Schedule{}".format(
                    c1, sched1.id))
                fzn.int_eq_reif("C{0}_Schedule".format(c1),
                                sched1.id,
                                c1schedule1,
                                defines=c1schedule1,
                                name="Precedence_H")
                c2succeedsc1 = fzn.bool_var("C{}SucceedsC{}S{}".format(
                    c2, c1, sched1.id))
                fzn.set_in_reif("C{}_Schedule".format(c2),
                                succeeding[sched1.id],
                                c2succeedsc1,
                                defines=c2succeedsc1,
                                name="Precedence_H")
```

```
24                  c1c2precedence = fzn.bool_var("C{}C{}Precedence_{}".format(
25                      c1, c2, i))
26                  fzn.array_bool_and([c2succeedsc1, c1schedule1],
27                                     c1c2precedence,
28                                     defines=c1c2precedence,
29                                     name="Precedence_H")
30                  pos.append(c1c2precedence)
31                  i += 1
32
33              fzn.array_bool_or(pos, True, name="Precedence_H")
```

Listing 3.18: Implementation of the integer variables (`IntVar`) object

The generation of FlatZinc is implemented for the remaining constraints as well, that is WD_S, MG_G, MD_D, MDL_S, $\mathsf{MBR}_{R,S}$ and $\mathsf{MBL}_{M,S}$. However, because of their complexity and to keep implementational details to a minimum these are not described in this thesis.

4 Formulation Encoding

Two formulations were presented in this thesis, namely the *compact* formulation and the *extended* formulation. The two formulations are different and in this chapter the key differences between the formulations will be presented, as well as a method for encoding of a problem in compact form into a problem in extended form.

4.1 Differences

The compact and extended formulations presented in this thesis shows two different ways of modeling timetabling problems. While some concepts and constraints are present in both formulations, there are differences between them. These differences will be summarized here to give a detailed but non-comprehensive overview.

Events and Classes The major difference between the compact and the extended formulation is the difference between *events* and *classes*. In the compact formulation, events have to be scheduled, where an event is a single occurrence of a class of a course. In the extended formulation *classes* have to be schedules where the notion of a *class* is a set of possibly recurring events of a course.

Sectioning of Students The sectioning of students is not a part of the problem for the compact formulation. For the compact formulation, the list of classes that a student is attending, is predetermined, but still imposes constraints on the classes. For the extended formulation, the sectioning of the student *is* a part of the problem, while imposing constraints on the classes.

Teachers The extended formulation does not include the notion of Teachers as included in compact formulation.

Feasible Rooms In the compact formulation, it is not possible to limit the feasible rooms for a specific event. The extended formulation defines a set of feasible rooms for each class.

Room and Time Penalty In the compact formulation, it is not possible to prefer a certain time or room for a given event. In the extended formulation this

can be done by adding penalties to the feasible schedule and feasible rooms of a class.

No Room For the compact formulation a room is needed for every event, whereas the extended formulation allows classes to not have a room. As an example this makes it possible to schedule online classes and similar events which does not need a room.

Room and Time Stability In the compact formulation events of the same type for the same course are allowed to take different rooms and different times, however, the occurrence of this should be minimized for events of the same type and course. In the extended formulation reoccuring classes are forced to have the same time, days and duration every week.

Room Distances Distances between rooms are only included in the extended formulation, but not in the compact formulation.

Room Restriction It is not possible to restrict or enforce a set of events to be scheduled into the same room in the compact formulation.

Time Restriction It is not possible to restrict or enforce a set of events to be scheduled into the same time of the day, nor to restrict or enforce a set of events to be scheduled into the same days of the week for the compact formulation. This can be enforced for a set of classes in the extended formulation by using the distribution constraints *SameTime*, *DifferentTime*, *SameDays*, *DifferentDays*, *SameWeeks* and *DifferentWeeks*.

Overlap It is not possible to enforce a set of events to overlap or not overlap in the compact formulation. This can be done by using the *Overlap* and *NotOverlap* constraints in the extended formulation.

Max One Event Per Course Per Day In the compact formulation, a constraint enforce that only one event can occur per course per day. This constraint does not exist in the extended formulation.

The two formulations may be used to model different kinds of timetabling problems. While the extended formulation offer more control with regards to the relationship between classes, the compact formulation might be better suited for timetabling problems where these features is not necessary.

4.2 Encoding

A problem expressed in *compact form* can be encoded into *extended form*. The process of this transformation will be described in this section, and we will refer

to Section 4.1 for an overview of the differences. Considering a model in compact form $\mathcal{M}_c$, the model of $\mathcal{M}_c$ in extended form will be referred to as $\mathcal{M}_x$.

Time slots

Both the compact formulation and extended formulation includes the notion of a *time slot*. Here it is assumed that the size of a time slot, and thus the number of time slots per day, is the same for $\mathcal{M}_c$ and $\mathcal{M}_x$.

Events ⇒ Classes

The events of $\mathcal{M}_c$ should be encoded into classes of $\mathcal{M}_x$. For each event $e \in E$ where E is the set of events of M^e, each event e corresponds to exactly one class $c \in C$ where C is the set of classes for $\mathcal{M}_x$. It is necessary to encode each event as individual classes, since $\mathcal{M}_x$ will enforce every event of a class in the model $\mathcal{M}_x$ to start at the same time of the day, have the same duration, and occur on the same days of the week, as defined by a *schedule*.

Since each event e has to be scheduled into any time slot for a specific week W_e, if d_e is the duration of event e, the set of schedules for class c is every combination of adjacent time slots of length d_e of a day for each day of week W_e.

As an example, and event e with duration 2 which have to be scheduled into the second weeks of the semester, if the number of time slots per day is 6, then the class can be defined in the ITC2019 instance format as follows:

```
...
<class id="1">
    <time weeks="01000000" days="1000000" start="0" length="2">
    <time weeks="01000000" days="1000000" start="1" length="2">
    <time weeks="01000000" days="1000000" start="2" length="2">
    <time weeks="01000000" days="1000000" start="3" length="2">
    <time weeks="01000000" days="1000000" start="4" length="2">
    <time weeks="01000000" days="0100000" start="0" length="2">
    <time weeks="01000000" days="0100000" start="1" length="2">
    ...
    <time weeks="01000000" days="0000001" start="4" length="2">
</class>
...
```

If H_d is the number of time slots per day, and D_w is the number of days per week, then the total number of schedules to be introduced for a class is:

$$D_w \times (H_d - (d_e - 1))$$

Rooms

Rooms are defined in an identical way for $\mathcal{M}_c$ and $\mathcal{M}_x$, thus these can be encoded directly. For the set of time slots T_r where a room r is occupied as specified for a model $\mathcal{M}_c$, the *room unavailabilities* of r in $\mathcal{M}_x$ can be defined using a bijective function f. Let,

$$f : W \times D \times S \to \mathbb{Z}^+$$

and

$$f^{-1} : \mathbb{Z}^+ \to W \times D \times H$$

such that every positive integer $z \in \mathbb{Z}^+$ maps to exactly one week W, day D and time slot of day H.

For a given week w, day d, and time slot of the day h, and where D_w is the number of days per week, H_d is the number of time slots per day,

$$f(w, d, h) = D_w \cdot H_d(w - 1) + H_d(d - 1) + h - 1$$

Then it is possible for each time slot $z \in T_r$ to define room r as unavailable in week w, on day d and in time slot of the day h by $f^{-1}(z)$.

As the set of feasible rooms for an event in $\mathcal{M}_c$ is equal to the set of all rooms, all rooms will also have to be added for each class c of $\mathcal{M}_x$.

Teachers

The notion of teachers is a part of the compact formulation, but not the extended formulation. However, constraints can be added to $\mathcal{M}_x$, such that it is possible to enforce the same constraints as teachers impose on $\mathcal{M}_c$. The two constraints imposed by teachers on $\mathcal{M}_c$ is:

- A teacher may only teach one class at a time and
- a teacher may only teach if the teacher is available

The first constraint can be enforced by adding a *NotOverlap* constraint for each teacher t, on the set of classes that t is teaching. The second constraint can be enforced by removing schedules from each class c in $\mathcal{M}_x$, that a teacher is teaching, such that no schedule of a class c overlap with a time slot where t is marked as busy.

Precedences

Precedence constraints exists for both $\mathcal{M}_c$ and $\mathcal{M}_x$, with the difference that the definition of precedences in $\mathcal{M}_c$ is a list of pairs of events e_i, e_j where e_i precede e_j where in $\mathcal{M}_x$ precedences are defined for ordered sets of classes, where the classes of each set should be scheduled to occur in that order.

This can be encoded directly from $\mathcal{M}_c$ to $\mathcal{M}_x$ where a precedence constraint on c_i, c_j is added to $\mathcal{M}_x$ for every pair of precedences of $\mathcal{M}_c$.

One Event per Course per Day

For the compact formulation, a constraint enforce that no more than one event for every course can occur per day. This is not enforced in the extended formulation, however the use of distribution constraints is able to enforce this. The *MaxDayLoad* (MDL) distribution constraint can be used *in some cases*.

For every course $\mathcal{C}$ of an instance in $\mathcal{M}_c$, introduce a MDL_S constraint on all corresponding classes $c_i \in C_\mathcal{C}$ of $\mathcal{M}_x$. The parameter S for each MDL constraint defines the number slots that may be occupied per day for each week, for the set of classes. For a feasible solution to be found, this should be equal to the size of the duration d_e of the event e with the highest duration. However, if more than one other class of the constraint is smaller than or equal to $\frac{d_e}{2}$, the constraint of $\mathcal{M}_c$ is not guaranteed to hold for $\mathcal{M}_x$. As an example, one day with one class of duration d_e, and another day with two classes each of duration $\leq \frac{d_e}{2}$ would not violate the constraint in $\mathcal{M}_x$, but would be violated for $\mathcal{M}_c$. Thus, the MDL constraint cannot be used to model this constraint in all cases.

Another distribution constraint, namely the $MaxBlock_{M,S}$ (MBL) constraint, limits the length of a block of two or more consecutive classes during a day, such that there is no more than M slots in a block, and where two classes are considered to belong to the same block if the gap between them is not more than S time slots.

The $\mathsf{MBL}_{M,S}$ constraint can be used to limit the number of classes per day of a course. For each course $\mathcal{C}$, add a $\mathsf{MBL}_{M,S}$ constraint to $\mathcal{M}_x$ on the set of all classes of $\mathcal{C}$. Let $M = 1$ and let S be the length of the day $S = H_d$. This will enforce that

at most one classes of $\mathcal{C}$ is scheduled on any day. Adding a second class to the day, would result in a *block* on the given day, and since $M = 1$, the length of a block cannot exceed 1 time slot, the assignment would not be valid. If only one class of $\mathcal{C}$ is scheduled on the given day, it is not a block, and thus this assignment is valid.

This shows that a problem expressed in terms of the *compact* formulation can be encoded into a problem expressed in terms of the *extended* formulation.

5 Computational Results

Tests were performed on all three models; the MiniZinc model for the compact formulation, and the MiniZinc model and FlatZinc model for the extended formulation. The nature of the instances used as input will be presented in Section 5.1, the methods used for testing and the results from tests performed on the model for the compact formulation will be presented in Section 5.2, and on the model for the extended formulation in Section 5.3. Finally the findings and the validity of the results will be discussed in Section 5.4.

The major advantage of using standardized modeling languages for modeling constraint programming problems, such as MiniZinc and FlatZinc is that the same model can be used for different solvers. A set of solvers was chosen to solve the IMADA Timetabling problem and the ITC2019 problem. Not all the chosen solvers were used for both problems, mainly because a subset of the solvers were only discovered after starting the development of the model for the extended formulation. The intersection between the set of solvers for each case should however still give a good overview. The solvers were chosen based on their performance in the MiniZinc Challenge[1] which is an annual competition of constraint programming solvers, with the exception of *Chuffed* which is included because it is included with the standard installation of MiniZinc. Only open source solvers were chosen and a brief description of each solver will be stated here.

Gecode (Gecode Team, 2006) The default solver of MiniZinc, based on Gecode which is an open source toolkit for developing constraint-based systems and applications. Gecode is developed in C++ and won gold medals in all categories at the MiniZinc Challenges from 2008 to 2012.

Chuffed (Chuffed Team, n.d.) A lazy clause generation solver. Chuffed utilize techniques to compute no-goods for reducing the search space.

Choco (Prud'homme, Fages, and Lorca, 2017) Solver for Constraint Programming written in Java. Choco have won a total of nine bronze medals, four silver

[1] https://www.minizinc.org/challenge.html

medals and one gold medal in the MiniZinc Challenges from 2013 to 2018.

OR-Tools (*Google's OR-Tools* n.d.) Constraint Programming Solver backed by Google. Won four gold medals in the MiniZinc Challenge 2018, and have won several medals every year since 2012 at the MiniZinc Challenge.

Yuck (Marte, n.d.) A constraint-based local-search solver written in Scala. Yuck won silver medals at the MiniZinc Challenge 2017 and 2018 in the Local Search category.

Oscar (OscaR Team, 2012) Another constraint-based local-search solver written in Scala. Oscar won Bronze medals in the Local Search category in 2017 and 2018, and a gold medal in 2016 at the MiniZinc Challenge.

All tests in this chapter were performed on a computer with an Intel Core i7-4790 Quad-Core CPU @ 3.60GHz and 16 GB Random-access Memory (RAM).

5.1 Instances

For the compact formulation a MiniZinc model was made for the IMADA Timetabling Problem. Real-world data from a semester at the Department of at Mathematics and Computer Science at the University of Southern Denmark was used as input. Details of the input data, named *IMADA Instance* can be seen in Table 5.1, which contains the number of *weeks*, *courses*, *students*, *events*, *teachers*, and the number of pairs of events of *pairings* and *precedences*. The size of the generated Dzn file is given as well.

	IMADA Instance
Weeks	15
Courses	10
Students	75
Rooms	2
Events	299
Pairings	255
Teachers	11
Precedences	1455
Dzn Size (KiB)	160.1

Table 5.1: Details of the IMADA Instance used as input data for the MiniZinc model for the IMADA Timetabling Problem

Instances from ITC2019 contains anonymized real-world data from universities around the world. These were used for the extended formulation and are split into two sets of data sets. The first set will be referred to as the *test sets* of ITC2019, and the second set will be referred to as the *competition sets*. Details about each data set for the test set and competition set can be seen in Table 5.2.

The table shows the size of the instance in the ITC2019 XML instance format and the number of courses, classes, rooms and students for each instance. Furthermore the number of *hard* and *soft* distribution constraints are given for each instance, along with the *total* number of schedules and the number of schedules that are unique.

						Distributions		Schedules	
Instance	Size (MiB)	Courses	Classes	Rooms	Students	Hard	Soft	Total	Unique
wbg-fal10	0.47	21	150	7	19	56	26	4617	154
lums-sum17	0.2	19	20	62	0	2	0	340	93
bet-sum18	0.13	48	127	46	0	110	34	210	50
pu-cs-fal07	0.44	44	174	13	2002	68	34	2958	182
pu-llr-spr07	4.5	602	802	56	27881	194	135	8656	508
pu-c8-spr07	8.54	1036	2418	213	29514	1288	716	30538	896
agh-fis-spr17	14.57	340	1239	80	1641	823	400	145870	9655
agh-ggis-spr17	5.82	272	1852	44	2116	2202	488	46667	2836
bet-fal17	3.88	353	983	62	3018	861	390	23369	595
iku-fal17	12.61	1206	2641	214	0	2238	665	93398	588
mary-spr17	2.95	544	882	90	3666	3155	796	12328	620
muni-fi-spr16	1.41	228	575	35	1543	645	95	9556	789
muni-fsps-spr17	1.48	226	561	44	865	331	69	11356	1953
muni-pdf-spr16c	15.93	1089	2526	70	2938	1458	570	150567	2696
pu-llr-spr17	4.7	687	1001	75	27018	427	218	9292	993
tg-fal17	1.94	36	711	23	0	461	42	18384	1645

Table 5.2: Overview of instances of ITC2019, split into *Test set* (top) and *Competition set* (bottom).

As seen in Table 5.2, each instance includes a number of hard and soft distribution constraints. A more detailed view of each instance is given in Table 5.3 for the test set, and Table 5.4 for the competition set, where the number of hard and soft constraints are given for each type of distribution constraint and instance.

5.2 The Compact Formulation

The MiniZinc model for the IMADA Timetabling problem was tested using Gecode, Chuffed, Choco and OR-Tools. Because of the relatively small problem size, a time limit of 900 seconds (15 minutes) was set for each of the solvers. The model

	wbg-fal10		lums-sum17		bet-sum18		pu-cs-fal07		pu-llr-spr07		pu-c8-spr07	
	H	S	H	S	H	S	H	S	H	S	H	S
SS	—	—	—	—	2	3	—	—	10	—	25	—
ST	—	—	—	—	—	—	8	—	—	—	27	—
DT	—	—	—	—	—	—	—	—	1	—	1	—
SD	—	—	—	—	10	9	—	—	22	9	84	129
DD	—	—	—	—	2	2	—	—	1	—	2	—
SW	—	—	—	—	—	—	—	—	—	—	—	—
DW	—	—	—	—	—	—	—	—	—	—	—	—
SR	—	—	—	—	28	10	—	8	24	4	70	106
DR	—	—	—	—	—	—	—	—	—	—	—	—
O	—	—	—	—	—	—	—	—	—	—	—	—
NO	—	26	—	—	1	1	8	26	4	110	173	346
SA	56	—	2	—	55	—	52	—	116	—	823	—
P	—	—	—	—	—	—	—	—	—	—	20	1
WD	—	—	—	—	12	7	—	—	11	6	56	114
MG	—	—	—	—	—	2	—	—	5	6	7	20
MD	—	—	—	—	—	—	—	—	—	—	—	—
MDL	—	—	—	—	—	—	—	—	—	—	—	—
MBR	—	—	—	—	—	—	—	—	—	—	—	—
MBL	—	—	—	—	—	—	—	—	—	—	—	—

Table 5.3: Number of hard (H) and soft (S) distribution constraints of each type for each ITC2019 test set.

was run 3 times for each solver on the IMADA Instance. Table 5.5 shows for each solver, the value of the objective function for the first feasible solution found, the value of the objective function for the best feasible solution found and the average time (in seconds) for both.

The results with all intermediate feasible solutions for each solver can be seen in Figure 5.1.

5.3 The Extended Formulation

Here the results from tests on the implementations of the extended formulation will be presented. For the ITC2019 competition an *online validator* was available for validating found solutions. This was used for finding the cost of each found solution presented in this section.

5.3.1 The MiniZinc Model

Plans were made to test this model for 1 hour for each solver and instance. However, limited time and unexpected circumstances meant that this was not done in time. Errors were reported by the online validator for all but the smallest instances, but it was not possible to locate the source of these problems in time.

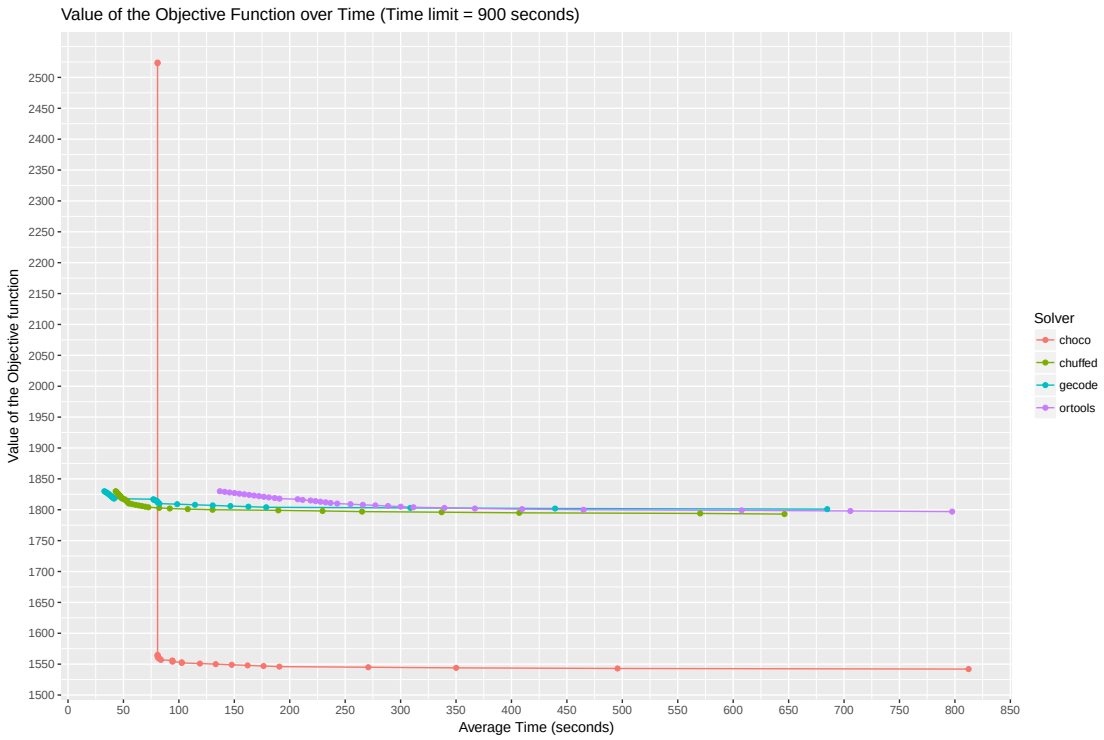

Figure 5.1: Results from running multiple solvers on the IMADA model

5.3.2 FlatZinc Generation

FlatZinc models were generated using the implemented `itc2fzn` software described in Section 3.4. The goal was to generate a FlatZinc model for each of the instances of ITC2019. No time limit was imposed on the FlatZinc generation and statistics about the number of generated constraints for each instance were gathered during generation of each FlatZinc model. Since the first attempt to generate a FlatZinc model for all instance of ITC2019 failed for some of the instances, two further attempts were made where some constraints were removed from the problem. These three attempts and the process and logic behind them will be described here. In all attempts, the minimization of *student conflicts* were disabled as the implementation of this is too ineffective. This will be described further in Section 5.4.

For reference, the number of constraints of each type can be seen in Table 5.3 and 5.4 which displays the number of constraints present in the instances.

First Attempt In the first attemt FlatZinc models were generated with the full set of constraints (except minimization of *student conflicts*) and the number of generated constraints introduced to each model can be seen in Table 5.6 and 5.7. As seen, generation of FlatZinc models for 5 of the instances failed, namely *agh-fis-spr17*, *agh-ggis-spr17*, *bet-fal17*, *iku-fal17* and *muni-pdf-spr16c*. In all cases this was due to lack of memory.

Second Attempt As the first attempt to generate FlatZinc models failed for some of the instance because of the memory restrictions, all soft constraints were disabled to reduce the memory usage by `itc2fzn`. This will still create valid models, since all hard constraints are still enforced. However, even after disabling all soft constraints, the memory usage was still too high to generate FlatZinc models for the 5 missing instances.

Third Attempt As seen in Table 5.6 and 5.7 the distribution constraint with the highest number of introduced constraints on average is the SA, or SameAttendees constraint. This constraint was disabled in this attempt, as another attempt to reduce to memory usage by `itc2fzn`. This obviously produce models where the SA constraints are allowed to be violated. However, this attempt also failed, and FlatZinc models were never generated for the 5 missing instances.

5.3.3 Solving the FlatZinc Models

Since three different FlatZinc models were generated for each instance, tests were made on each instance to try to understand the complexity of the problems. For this, the solvers Gecode, Chuffed, Yuck, OR-Tools and Oscar were chosen as these

represents a variety of some of the best open source FlatZinc solvers available.

Each FlatZinc solver was executed 3 times with a time limit of 1 hour on each of the generated FlatZinc models of each instance. For each instance and solver where a result was found, the instances were validated using the online validator of ITC2019. The following tables show the average of the reported *cost*, that is, the value of the objective function, for all three tests on each solver and instance.

- In Table 5.8 the results can be seen for each solver and instance on the models generated in the *first attempt*, that is, models where all constraints except for the minimization of student conflicts are enabled.
- In Table 5.9 results from can be seen for each solver and instance on the models generated in the *second attempt*, that is where all soft constraints were excluded during the generation of the model.
- In Table 5.10 results from can be seen for each solver and instance on the models generated in the *third attempt*, that is where the SA constraints were excluded during generation of the model.

In all three tables, the 5 instances where the generation of a FlatZinc model failed are disregarded. The symbols used in the tables are:

"—" No feasible solution found within the time limit.

N (N is a number), solver returned N as a feasible solution.

N^* (N is a number), solver returned N as the optimal solution.

(N) (N is a number), solver returned N as the total cost, but there was unexpected errors during validation.

5.4 Discussion

In this chapter results from the MiniZinc model for the compact formulation and results from the MiniZinc model and the generated FlatZinc models were presented.

Tests on the MiniZinc model for the compact formulation looks promising, and all tested solvers were able to find feasible solutions for the IMADA case. Since this was the only instance available for this model at the time, and since this is quite small compared to the problems presented for the extended formulation, it is not possible to conclude anything from these results. An obvious next step would have been to model problems of ITC2019 in terms of the compact formulation. As seen in Figure 5.1, the Choco solver gave solutions that does not look consistent with

the solutions of the other solvers. It has not been possible to confirm within the time limit of this thesis why this is, or if these solutions are valid.

Test results on the MiniZinc model made of the extended formulation were not presented in this chapter. This was due to errors reported by the online validator for the ITC2019 instances. This is unfortunate and suggests that one or more constraints contain problems which were not uncovered within the time of writing. Since the techniques used for modeling are quite different from the techniques used for the generated FlatZinc models, the results would not have been directly comparable. However, one major drawback compared to the generated FlatZinc models can still be made.

For the MiniZinc model data will first have to be generated in the Dzn format. This process takes a long time, much longer than generation of a FlatZinc model as described in this thesis. This can partly be owed to the fact that the MiniZinc model operates on the full set of schedules for each problem, while the generated FlatZinc models only operate on the unique set of schedules. After the generation of the data file, MiniZinc will be executed with the model file and data file as input. This will generate a FlatZinc file, which will be passed to the chosen solver. For the generation of FlatZinc models, these two steps of generating the data and then generating the FlatZinc is done in one single step, before passing the generated FlatZinc file to the solver. Thus, for the implemented generation of FlatZinc models, the overhead introduced by MiniZinc is completely avoided.

During testing of the generated FlatZinc models of the extended formulation, results were gathered from three different generated models for each instance and each FlatZinc solver. While many of the solvers failed to find solutions for a large subset of the instances, they provide an interesting insight in why the solvers failed to find any feasible solutions in most cases.

As seen in Table 5.8, solutions are found for four instances by Oscar. These are, as seen in Table 5.2, the four smallest instances in terms of the number of *unique* schedules. Gecode, Chuffed and OR-Tools are only able to find solutions for the two smallest instances. This might suggest that solvers based on Local Search have an advantage in solving at least simple timetabling problems. However, this also suggest that the complexity of a timetabling problem in the extended form relies partly on the number of unique schedules present in a given problem.

This is not a surprising result, since many of the constraints in the extended formulation are defined such that the complexity, in terms of the number of introduced constraints, increases polynomially according to either the number of unique schedules or the number of rooms. That is where C is the number of classes of a distribution constraint, and S is the number of unique schedules, the number

of generated constraints for a distribution constraint can often be described as being in the order of

$$\binom{C}{2} \times |S|^2 \quad \text{or} \quad \binom{C}{2} \times |R|^2$$

in the worst case. In the average case however, this is often better since constraints for a pair of classes is not generated if the features of the classes, e.g. the set of feasible rooms for the classes, by themselves prevent the constraint from being violated. However, for the central constraints of a problem, such as the SO constraint, the number of constraints generated most likely approach

$$\binom{C}{2} \times |S|^2$$

where C and S is the total number of classes and unique schedules of the problem.

From looking at the Tables 5.6 and 5.7 another problem is evident, namely the number of constraints generated for the SA, or SameAttendees distribution constraint. This can be explained by its dependence on both *rooms* and *schedules*, instead of just one or the other. That is, the number of generated constraints of a SA distribution constraints, where C is the number of classes of the distribution constraint and S and R is the number of schedules and rooms for the classes C respectively, in the worst case the number of generated FlatZinc constraint will be in the order of:

$$\binom{C}{2} \times |S|^2 \times |R|^2$$

The same complexity applies for the minimization of student conflicts, since this is essentially implemented in the same way as a soft SA constraint for the classes of a student. In that case however, C is the total number of classes of the problem, making it obvious why models were difficult to generate without disabling this constraint.

By disabling all soft constraints, it is clear that it was easier for some solvers to find a feasible solution for some instances. In this case however, Oscar was only

able to find feasible solutions for two of the instances. Since the implementational details of the solvers are out of scope for this thesis, it is unclear why this is, but it could be suggested that for Oscar the soft constraints helps guide the solver to find a feasible solution. Where Chuffed and Yuck were used as the solvers, more feasible solutions were found when the soft constraints were not included in the problems. This could be because of the smaller number of variables and constraints.

In two cases, namely using Chuffed on the problems *muni-fi-spr16* and *muni-fsps-spr17*, feasible solutions were found where the online validator of ITC2019 found errors due to students not being sectioned correctly. This could be due to a problem which will be summarized in Section 7.

In Table 5.10, even more solutions are found, although the results from this case made it possible for the solvers to violate the SA constraint. Thus these solutions are not feasible solutions to the initial problems. Especially Yuck excels here by finding solutions for all but two of the problems. This clarifies that the SA constraint is implemented in an insufficient way, which makes it too complex to handle by the solvers. It also adds to the theory from this section, that local search solvers such as Yuck and Oscar, might have an advantage on timetabling problems expressed in the extended form.

In Section 4.2 it was described how a problem expressed in terms of the compact formulation could be encoded into a problem expressed in terms of the extended formulation, and showed that this conversion was theoretically possible. It was described how the notion of *events* in the compact form can be encoded into *classes* and *schedules* of the extended form. As presented each event is encoded into exactly one class, and a number of schedules which in the worst case will be in the order of:

$$D_w \times (H_d - (d_e - 1))$$

for each class, where D_w is the number of days per week, H_d is the number of time slots per day, and d_e is the duration of the event. As seen, the number of schedules for each class is highly dependent on the number of time slots per day, H_d, which is a combination of the length of the day and the size of each time slot for the problem. However, the results presented in this chapter argues that the number of schedules of a given problem in extended form affect the number of introduced constraints dramatically. Thus, it can be argued that while the encoding is possible, solvers might find some types of problems easier to solve when expressed in the compact form, as opposed to the same problem expressed in extended form.

The opposite of this can be argued as well. Given a problem stated in terms of the compact formulation on a large set of weeks and events, and a small set of courses, many of these events could be recurring events of one course. In this case, the number of corresponding classes and schedules in extended form, could be considerably smaller than the number of events given in the compact form, since one schedule is able to represent multiple events. This is however also largely dependent on the number of time restrictions of each event.

In conclusion, these arguments show that the two formulations are different beside what is stated in Chapter 4, and whether a given timetabling problem can be solved or not might depend heavily on whether the right model was chosen.

	agh-fis-spr17		agh-ggis-spr17		bet-fal17		iku-fal17		mary-spr17		muni-fi-spr16		muni-fsps-spr17		muni-pdf-spr16c		pu-llr-spr17		tg-fall17	
	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S
SS	—	—	—	—	4	—	—	1	—	—	—	—	—	—	1	—	45	6	—	—
ST	54	4	189	11	12	—	59	—	15	—	30	—	22	1	47	1	11	—	37	—
DT	—	—	—	—	2	—	—	—	—	—	—	—	—	—	—	—	6	2	—	—
SD	54	17	448	71	22	125	229	11	53	29	59	10	37	2	64	—	15	37	37	14
DD	11	3	3	1	195	132	2	—	—	—	8	7	2	—	—	—	4	1	—	12
SW	6	—	—	—	—	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—
DW	—	—	—	—	—	—	—	—	—	—	—	—	—	—	382	—	—	—	—	—
SR	24	6	155	17	11	4	124	10	14	5	25	1	15	1	5	—	1	—	—	—
DR	—	—	—	—	—	1	—	—	—	—	—	—	—	—	—	—	378	26	21	—
O	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
NO	1	6	2	4	3	—	17	—	—	111	9	42	1	64	—	126	23	115	325	—
SA	659	68	1124	—	419	5	1100	1	3025	628	404	—	191	—	570	—	292	—	62	4
P	4	273	31	324	—	—	47	631	10	—	73	26	52	—	383	1	1	5	—	—
WD	—	3	250	55	144	4	170	10	32	6	31	7	11	1	4	15	4	25	—	12
MG	—	—	—	—	—	119	—	—	6	17	—	—	—	—	—	19	—	4	—	—
MD	10	6	—	3	10	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
MDL	—	—	—	—	1	—	—	—	—	—	2	—	—	1	—	29	—	—	—	—
MBR	—	14	—	2	—	—	—	—	—	—	—	—	—	—	—	—	—	2	—	—
MBL	—	—	—	—	38	—	—	—	—	—	3	2	—	—	1	19	—	—	—	—

Table 5.4: Number of hard (H) and soft (S) distribution constraints of each type for each ITC2019 competition set.

	First feasible	Time	Best ub	Time
Gecode	1830	32.77	1801	684.82
Chuffed	1830	43.01	1793	646.27
Choco	2524	80.75	1542	812.36
OR-Tools	1830	136.95	1797	797.50

Table 5.5: Summarized results for each solver on the MiniZinc model on the IMADA Instance stating the *first* feasible solution found by the solver, and the *best* upper-bound along with the time in seconds for both.

	wbg-fal10		lums-sum17		bet-sum18		pu-cs-fal07		pu-llr-spr07		pu-c8-spr07	
	H	S	H	S	H	S	H	S	H	S	H	S
SP		9235		681		421		5917		17313		61063
RP		862		1624		3978		780		26364		51560
SO	261600		14143		6132		59541		1804893		3591370	
RU	0		10582		1117		0		893		20067	
SS	0	1	0	1	6	19	0	1	153	1	1170	1
ST	0	1	0	1	0	1	216	1	0	1	247	1
DT	0	1	0	1	0	1	0	1	181	1	181	1
SD	0	1	0	1	0	94	0	1	23840	907	33202	4729
DD	0	1	0	1	4	7	0	1	0	1	204	1
SW	0	1	0	1	0	1	0	1	0	1	0	1
DW	0	1	0	1	0	1	0	1	0	1	0	1
SR	0	1	0	1	54	53	0	17	24	9	92	441
DR	0	1	0	1	0	1	0	1	0	1	0	1
O	0	2	0	2	0	2	0	2	0	2	2350	10
NO	0	48235	0	1	21	76	1561	53935	126	8140	10637	251011
SA	84907	1	263686	1	49879	1	4771	1	1597109	1	4409434	1
P	0	1	0	1	60	87	0	1	413	289	1716	2819
WD	0	1	0	1	0	9	0	1	40	249	56	422
MG	0	1	0	1	0	1	0	1	0	1	0	1
MD	0	1	0	1	0	1	0	1	0	1	0	1
MDL	0	1	0	1	0	1	0	1	0	1	0	1
MBR	0	1	0	1	0	1	0	1	0	1	0	1
MBL	0	1	0	1	0	1	0	1	0	1	0	1

Table 5.6: Number of hard (H) and soft (S) constraints introduced by the FlatZinc generator `itc2fzn` for test sets of ITC2019, for each constraint.

	agh-fis-spr17		agh-ggis-spr17		bet-fal17		iku-fal17		mary-spr17		muni-fi-spr16		muni-fsps-spr17		muni-pdf-spr16c		pu-llr-spr17		tg-fal17	
	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S	H	S
SP										24657		19113		22713				18585		36769
RP										21636	4818		2698				29452		6144	
SO									2244030		703438		438395				1247520		666404	
RU									1587		5185		2995				0		9506	
SS									0	1	0	1	0	1			682	1443	0	1
ST									726	1	2539	1	11609	91695			1078	1	3798	1
DT									0	1	0	1	0	1			275	18	0	1
SD									7900	2548	37074	901	91019	9007			8240	1783	16918	22462
DD									0	1	532	694	854	1			449	154	0	994
SW									0	1	840	1	0	1			0	1	0	1
DW									0	1	0	1	0	1			0	1	0	1
SR									14	11	25	3	15	7			26	43	0	1
DR									0	1	0	1	0	1			0	1	0	1
O									112	2	1199	592	2517	2			0	213	0	2
NO									0	50401	9521	63055	3	23329			7069	18235	63740	1
SA									4733438	405974	746218	1	932093	1			2959517	1	3379783	876311
P									112	2	1199	592	2517	2			0	213	0	2
MG									114	851	0	1	0	1			0	57	0	1
WD									814	237	2584	385	285	67			42	1073	0	625
MD									0	1	0	1	0	1			0	1	0	1
MDL									0	1	0	1	0	1			0	1001	0	1
MBR									0	1	4818	1	0	1			0	1	0	1
MBL									0	1	659	3433	0	1			0	1	0	1

Table 5.7: Number of hard (H) and soft (S) constraints introduced by the FlatZinc generator `itc2fzn` for competition sets of ITC2019, for each constraint.

Instances	gecode	chuffed	yuck	ortools	oscar
wbg-fal10	—	—	—	—	546
lums-sum17	4*	4*	—	48	4
bet-sum18	3321	2229	—	2788	1786
pu-cs-fal07	—	—	—	—	1113
pu-llr-spr07	—	—	—	—	—
pu-c8-spr07	—	—	—	—	—
mary-spr17	—	—	—	—	—
muni-fi-spr16	—	—	—	—	—
muni-fsps-spr17	—	—	—	—	—
pu-llr-spr17	—	—	—	—	—
tg-fal17	—	—	—	—	—

Table 5.8: Average cost for each FlatZinc solver and ITC2019 Instance on generated FlatZinc models with all constraints enabled.

Instances	gecode	chuffed	yuck	ortools	oscar
wbg-fal10	—	—	877	—	714
lums-sum17	51	88	69	—	—
bet-sum18	—	3547	3389	3436	3421
pu-cs-fal07	—	2101	2058	—	—
pu-llr-spr07	—	—	—	—	—
pu-c8-spr07	—	—	—	—	—
mary-spr17	—	—	—	—	—
muni-fi-spr16	—	(23883)	—	—	—
muni-fsps-spr17	—	(163430)	—	—	—
pu-llr-spr17	—	—	—	—	—
tg-fal17	—	—	—	—	—

Table 5.9: Average cost for each FlatZinc solver and instance on generated FlatZinc models where all soft constraints were excluded.

Instances	gecode	chuffed	yuck	ortools	oscar
wbg-fal10	—	—	446	—	431
lums-sum17	4*	4*	4	48	4
bet-sum18	2368	1753	1727	2676	1628
pu-cs-fal07	—	—	620	—	823
pu-llr-spr07	—	—	63956	—	—
pu-c8-spr07	—	—	—	—	—
mary-spr17	—	—	50150	—	—
muni-fi-spr16	—	—	14917	—	—
muni-fsps-spr17	—	—	—	—	—
pu-llr-spr17	—	—	—	—	—
tg-fal17	—	—	4448	—	—

Table 5.10: Average cost for each FlatZinc solver and instance on generated FlatZinc models where the `SameAttendees` constraint were excluded.

6 Conclusion

In this thesis two formulations, a *compact* formulation and an *extended* formulation, for modeling flexible timetabling problems were presented. Three implementations were described, namely two models in the MiniZinc modeling language and one method for FlatZinc model generation, based on the presented formulations.

The work in MiniZinc and FlatZinc throughout this thesis have been extensive. The MiniZinc is heavily documented in Peter J. Stuckey (2018), and the language makes it simple to model problems and use different solvers to solve them. However, the cost of using MiniZinc is often in terms of transparency and flexibility. It is not obvious how MiniZinc constraints are converted into FlatZinc, without spending a lot of time analyzing the corresponding FlatZinc file. Furthermore, since this transformation changes depending on the used solver, the process of finding which constraints are too complex for a given solver can become very tedious, especially without knowing how the utilized solver works. This becomes even more difficult when a MiniZinc constraint in itself is complex. The FlatZinc language on the other hand is too low-level for hand-written models, but very useful when generated as described in this thesis. The biggest drawback have been in terms of the support of the solvers, since these do not always implement the full FlatZinc specification. When this is said the overall experience of using MiniZinc and FlatZinc for this project have been overwhelmingly positive, and we definitely think they serve their targeted purpose.

In Chapter 4 the differences between the compact and extended formulations were given along with a method for encoding a problem in *compact* form into a problem in *extended* form. This shows that the encoding is theoretically possible, but as argued in Section 5.4, the resulting problem might be too computationally difficult to solve in practice. Because of this, it is concluded that even though the formulations are both flexible and can be used as foundations for modeling real-world timetabling problems, they are very different, and able to accommodate different types of timetabling problems. This is similar to the findings in Bettinelli et al. (2015), and suggests that a universal *one-size-fits-all* solution is not possible considering the complexity and differences of real-world university timetabling

problems.

In Chapter 5 results were presented from each of the implementations, along with a discussion of the results with respect to the work of this thesis. Unfortunately it was only possible to test one instance on the implementation of the compact formulation within the time limit of this thesis. The size of this instance being relatively small, it was possible for all tested solvers to find a feasible solution. Larger problems would have been interesting for comparison with the results from the tests of the extended formulation.

For the extended formulation it was stated that some constraints of the models are more complex than others, and that the implementations might not be efficient enough to handle these. However, the implemented models still provide insight in how complex problems for timetabling of full semesters can be modeled and solved using constraint programming. Given that some of the most recognized open source solvers available were used in the making of this thesis, it is unfortunate that it was not possible to find feasible solutions for all the presented problems. Even when some of the complex constraints were removed from the model, the problems were still too difficult to solve in most cases. This could indicate a common problem in the implemented models for the extended formulation, or that constraint programming solvers are simply not mature enough to solve problems with the high complexity of timetabling problems.

For future research in the topic of timetabling problems, we suggest emphasizing on *flexible* timetabling problems as described in this thesis, as opposed to research on single-week, inflexible timetabling problems. These problems are often too simplified to accommodate the needs of modern-day universities where timetabling problems are often very complex. Further research on how the *compact* and the *extended* formulations differ and the ideal purposes of each formulation could be highly relevant, as well as an analysis of why constraint programming solvers fail to solve the problems of the extended formulation presented in this thesis. Since the solvers based on local search techniques generally performed better than the other constraint programming solvers, we think that research in this direction could be successful if the right precautions are taken.

7 Future Work & Known Issues

This section will list a few improvements that could be made to this work in the future, along with problems discovered for which there was not time enough to fix before the deadline.

Ideas for improvements:

- During the implementation of the MiniZinc model for the extended formulation, it was not discovered that many schedules were actually unique. This fact could be used to improve the implementation as done in the implementation of the generation of FlatZinc models.
- Adding search annotations to the FlatZinc model. MiniZinc and FlatZinc supports *search annotations* which can be used for some solvers as guidance. These might have a great impact on the solutions, if supported by the solver.

Known issues discovered during writing which should be fixed in the future:

Section 2.2.3 The soft constraint $S3$ is defined as

> *Events of the same type should preferably be scheduled in the same room every week*
>
> In the model, the constraint counts the discrepancy between rooms of a pair of events. This is not correctly implemented, since this would impose larger penalty on the objective function for two rooms which have a larger discrepancy in the set of rooms R. Instead, a *fixed* penalty should be imposed if two events does not share the same room.

Section 3.2.1 The sectioning of students for the International Timetabling Problem is described. However, while the algorithm for sectioning of students work in most cases there could be cases where it does not. In particular, there could be cases where a child class of a class have less available spots than its parent class. In this case, the algorithm should back-track to another class of the subsection.

Section 3.3 Not all constraints are implemented for the MiniZinc model of the extended formulation. These were non-trivial to implement in MiniZinc, and attempts were made, but it was not possible to find a working solution for these within the time limit.

References

Alberg, Brian (2018). "Solving the IMADA Timetabling Problem using Constraint-Based Local Search". Individual Study Activity.

Bettinelli, Andrea et al. (2015). "An overview of curriculum-based course timetabling". In: *Top* 23(2), pp. 313–349.

Burke, Edmund K et al. (2010). "A supernodal formulation of vertex colouring with applications in course timetabling". In: *Annals of Operations Research* 179(1), pp. 105–130.

Chuffed Team (n.d.). *Chuffed: The Chuffed CP solver*. URL: https://github.com/chuffed/chuffed.

Gecode Team (2006). *Gecode: Generic constraint development environment*. URL: http://www.gecode.org.

Google's OR-Tools (n.d.). Google. URL: https://developers.google.com/optimization/.

Koch, Thorsten et al. (2011). "MIPLIB 2010". In: *Mathematical Programming Computation* 3(2), p. 103.

Marte, Michael (n.d.). *Yuck: A constraint-based local-search solver with FlatZinc interface*. URL: https://github.com/informarte/yuck.

Müller, Tomáš, Hana Rudová, and Zuzana Müllerová (2018). "University course timetabling and International Timetabling Competition 2019". In: *PATAT 2018 – Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2018)*, pp. 5–31.

Nethercote, Nicholas et al. (2007). "MiniZinc: Towards a standard CP modelling language". In: *International Conference on Principles and Practice of Constraint Programming*. Springer, pp. 529–543.

OscaR Team (2012). *OscaR: Scala in OR*. URL: https://bitbucket.org/oscarlib/oscar.

Peter J. Stuckey Kim Marriott, Guido Tack (2018). *MiniZinc Handbook*.

Project Repository (n.d.). The repository is also attached with the delivery of this thesis. URL: https://git.imada.sdu.dk/march/GLS.

Prud'homme, Charles, Jean-Guillaume Fages, and Xavier Lorca (2017). *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S.