Complex project scheduling

van der Linden, I.

Award date:
2012

Complex Project Scheduling

Master Thesis

April 16, 2012

Student: I. van der Linden

Supervisor: prof. dr. ir. W.P.M. Nuijten

TU/e

Preface

This master thesis is the result of my master project concluding the master Computer Science & Engineering at the Eindhoven University of Technology (TU/e). The project has been commissioned by IBM, specifically by the ILOG department. I would like to take the opportunity to thank the people that contributed to the results of this project.

First of all, I would like to express my sincere gratitude to Wim Nuijten. The course Constraint Programming, the opportunity to do my master project for IBM, the pressure to perform, the valuable council in general: thanks a lot! I especially enjoyed the sessions where we would brainstorm about the various solutions.

As for other employees of IBM, I would specifically like to thank Philippe Laborie and Stéphane Michel. They have been very valuable throughout the project.

Furthermore, I would like to thank all my friends, especially my fellow members of GEWIS fraternity *B.O.O.M.*, for their continued support. Without them I would not have had such a great time during my studies at the TU/e. Last but not least I would like to thank my parents and sister for their support (and patience) during my education at the TU/e.

Ivo van der Linden
Eindhoven, April 2012

Abstract

Many large businesses involved with the construction of aircraft, buildings, tanks, ships, and similar objects are faced with complex project scheduling problems. IBM ILOG builds business-specific applications for solving those scheduling problems using their *IBM ILOG CPLEX Optimization Studio* and *IBM ILOG Optimization Decision Manager Enterprise* products. The assignment IBM defined, upon which the work in this master thesis is based, is to develop an optimization model to solve the Complex Project Scheduling Problem (CPSP) in a generic way using CP Optimizer, the engine underneath CPLEX Optimization Studio. The definition of the generic CPSP is based on the set of examples they encounter in their day to day practice.

The development of this optimization model involves clearly defining the requirements, designing a mathematical model, and implementing this mathematical model for CPSP using the aforementioned IBM ILOG tools. The resulting implementation is shown to be generic by using it in various use cases. Some of these use cases originate from researched literature, others were practical cases encountered by IBM.

It is shown that the performance of the model is within acceptable distance of domain-specific solutions, proving its usefulness as a basis for domain-specific applications (with domain-specific GUI and data-sources). On two problems originating in literature we have shown that our model performs within 1.9% and 0.6% respectively compared to domain specific models. For a problem concerning aircraft assembly coming from IBM's practice we have shown that the generic solution performs as well as the problem-specific solution. The generic solution is also shown to be suitable for rail maintenance scheduling.

Contents

Chapter 0

Introduction

Many large businesses involved with the construction of aircraft, buildings, tanks, ships and similar objects are faced with complex project scheduling problems. [Bak74] defines scheduling as the problem of allocating scarce resources to activities over time. A scheduling problem is thus the challenge of creating a *schedule* describing when which activity should be done by what resources. In project scheduling, resources can be employees, pieces of equipment, consumables, et cetera. An activity is elementary part of a project; it can require resources and has a certain duration, making time a key factor in scheduling.

In this thesis we specifically discuss *Complex* Project Scheduling (CPS[1]). These projects are called complex because of their size (hundreds or thousands of tasks) and the constraints and the objectives defined on them. This makes it difficult and very time-consuming for humans to construct a good schedule without violating the constraints while optimizing the objectives.

The objects to be constructed in such projects (aircraft, buildings, et cetera) are typically expensive: millions to hundreds of millions of dollars. Because of this, creating a good project schedule is an important and valuable task. The gain of a good project schedule compared to an inefficient schedule in practice is proven to be easily measured in the millions to ten of millions of dollars.

IBM ILOG builds business-specific applications with optimization models for solving those scheduling problems with Mixed Integer Programming (MIP) and Constraint Programming (CP). CP is a paradigm to solve combinatorial optimization problems [RVBW06]. One states the problem in terms of variables, constraints, and an objective function after which a search procedure is used to find an optimal solution. A crucial difference with MIP is that in CP all along the search procedure an explicit representation is available of the set of values a variable can still take. This set of feasible values is called the domain of a variable. An important component of CP is the process of constraint propagation which removes infeasible values from domains and deduces new constraints. We refer to [RVBW06] for an extensive description of the field of CP. It has been shown that CP is a very succesful paradigm for solving scheduling problems [BPN01].

IBM ILOG has implemented the CP paradigm in IBM ILOG CP Optimizer (CPO); imagine it as a black box for solving optimization problems. To develop optimization models, IBM ILOG uses IBM ILOG CPLEX Optimization Studio (COS), which is an IDE around CPO. Integration of an optimization model into an application is done with IBM ILOG Optimization Decision Manager Enterprise (ODME). ODME is a framework for devel-

[1]Note that page 61 contains a list of abbrevations used throughout this thesis.

oping an application with a GUI and data connections around an optimization model developed with COS.

The assignment IBM defined, upon which the work in this master thesis is based, is to develop an optimization model to solve the complex project scheduling problem in a generic way using CPO. The definition of the generic CPSP is based on the set of examples IBM ILOG encounters in its day to day practice.

The organization of this thesis is as follows. Chapter 1 discusses the available literature in the subject area. Chapter 2 formulates the goal of the project in more detail and states the requirements of the model. It then formally describes Complex Project Scheduling (CPS).

Chapter 3 discusses the implementation of an optimization model matching the formal definition in the Optimization Programming Language (OPL), which is a declarative programming language for expressing constraints in COS. In Chapter 4, we discuss the development of various conversion tools that were developed to integrate the optimization model in different environments, especially with ODME.

The performance of the generic optimization model relative to domain-specific models has been measured and analyzed. Also, various tests have been performed to study the performance of the model in practical cases. The results of these studies can be found in Chapter 5.

Finally, in Chapter 6 we review the results of the project and identify areas for future research.

Chapter 1

Literature

Complex Project Scheduling (CPS) is an extension of Resource-Constrained Project Scheduling (RCPS). RCPS has been around since the 1950s ([BK12] gives a good introduction).

An RCPS instance consists of a set of tasks, and a set of finite capacity resources. Each task puts some demand on the resources. For example: within the context of building a house, a task 'install front door in house' might require one employee and various tools. A partial ordering on the tasks is also given specifying that some tasks must precede others (for example, the walls of the house have to be constructed before the door can be installed). In RCPS, the goal is to minimize the duration of the entire project (called makespan) without violating the precedence constraints or over-utilizing the resources.

This chapter is organized as follows. Section Section 1.0 discusses various approaches to solving the RCPSP. The engine within COS, CP Optimizer (CPO), is based on the SA-LNS algorithm, which is discussed in Section 1.1. Section 1.2 deals with various extensions to RCPS which are of interest to CPS. Section 1.3 shortly mentions the discovery of a new application domain for CPS. Finally, in Section 1.4, other Constraint Programming toolits are studied to validate the choice for COS.

1.0 RCPS Approaches

RCPS is studied extensively and a wide range of solution methods are reported in the literature. In 2006, Kolisch and Hartmann published a paper, [KH06], comparing 37 RCPS algorithms of different types, which they categorized as follows: X-Pass Methods, classical meta-heuristics (genetic algorithms, tabu search, simulated annealing, ant systems), non-standard meta-heuristics (local search-oriented approaches, population-based approaches), and other methods (forward-backward improvement (FBI), further heuristics).

The 37 algorithms were compared on computational speed on problems of various sizes (again, the details will not be mentioned here). However, the algorithm within CPO is not included, because the paper introducing the CPO algorithm, SA-LNS [LG07], was released in 2007 (it would be "local search-oriented"). However, [LG07] tested its performance on the datasets used in [KH06], which will be shortly discussed in Section 1.1.

1.1 Self Adapting Large Neighbourhood Search

In [LG07], a robust scheduling algorithm is presented based on Self Adapting Large Neighbourhood Search (SA-LNS). This algorithm is the base for CPO. The paper focuses on single-mode RCPS, with the property of non-preemptiveness (also see Section 1.2). It mentions that the algorithm can be easily extended to handle multi-mode RCPS, which has been incorporated in CPO.

Large Neighborhood Search (LNS) is based upon a process of continual relaxation and re-optimization: a first solution is computed (this is assumed to be easy) and iteratively improved. Each iteration consists of a relaxation step followed by a re-optimization of the relaxed solution. This process continues until some condition is satisfied, typically, when a time limit is reached. What makes SA-LNS "self-adapting" is the integration of machine learning techniques to converge on the most efficient neighborhood (as opposed to always selecting the same neighborhood in the same order).

The machine learning part keeps track of two probability distributions, one for selecting neighborhoods, one for selecting the completion strategy. After a cycle of LNS, the selections are rewarded based on their contribution to the optimization of the solution. Currently, only one completion strategy (*SetJustInTime*) is used. This completion strategy uses a linear relaxation of the problem and, doing so, has a global vision of the ideal position of activities in time would there be no resource limitation. Because of the machine learning techniques, SA-LNS requires no directions as to the direction of the search. It can therefore be considered to be a black box and we will do so in the remainder of this thesis.

Laborie and Godard tested the SA-LNS algorithm according to the procedure described in [KH06], and concluded: "The average deviation from the path-based lower bound is 32.4%. We estimate the average number of LNS cycles to be slightly less than 50000 which would position SA-LNS in the top 7 best approaches for RCPSP among the 37 approaches reviewed.". For 17 out of 21 benchmarks, SA-LNS is better or less than 4% worse than the competing algorithm. One interesting comparison is that for the "aircraft assembly" benchmark, SA-LNS performs 8.7% worse. Aircraft assembly is one of the domains that should fit in the generic design of the CPS model. In 2011, in an e-mail conversation, Philippe Laborie, one of the authors of [LG07], has mentioned that the performance of CPO is now within 5% of the aforementioned benchmark. With a little tweak (MultiPoint search instead of Restart), it performs within 2%. As a conclusion, it is assumed that the SA-LNS algorithm is suitable for solving RCPS-related problems.

1.2 RCPS Extensions

In the literature various extensions to RCPS are discussed are studied. The extensions that are relevant to CPS are discussed in this section.

Single-mode vs multi-mode RCPS. Multi-mode scheduling problems have a "resource allocation" dimension, which means it there are various modes in which each activity can be executed. This allows one to model optional activities, alternative resources, alternative recipes or routes, et cetera. CPS will feature alternative resources and is therefore multi-mode. Papers on this subject include [DRH99], [JMR+01], [SD98], [Har01], [LG07], and [Lab09].

Multi-skills RCPS. In traditional RCPS, employees are modeled as a pool of employees (a

resource with a capacity). It is not possible to model that an employee has more than one skill. Extending RCPS to allow that is termed "Multi-skills RCPS" ([PBMN07], [HK10]). CPS will feature multi-skills. As a consequence, resources with skills (employees) can no longer be modeled as a pool: each employee becomes a separate entity with its own skillset, which is either participating in a task or not.

Preemptive activities. Using preemptive activities, it is allowed to split an activity into a variable number of parts of variable length which means activities can be interupted and resumed. The CPS model will feature preemptive activities, i.e. it features preemption. Papers on this subject include [BVQ08], [ZLT06], [PV10], and [ZLS11].

Objective functions. Typically, in literature, the objective function (the value that will be optimized) is the 'makespan' (finish the project as fast as possible). In practice, money is usually a big factor. The cost of a project is based on a lot of factors, so CPS supports a weighted function, involving makespan and various costs like: fixed costs, costs for not executing a task, tardiness of tasks, costs for employees and costs for equipment.

1.3 Application domains

The graduation project has been initiated by IBM ILOG with the goal to design a generic solution for the CPSP. Application domains include spacecraft, the military industry (tanks, artillery, aircraft carriers, et cetera), the naval industry, and construction (buildings, bridges, et cetera).

In [ST11] and [AFC07], software projects are discussed as application domain for scheduling problems. The construction of software (using the traditional waterfall model) is not that different from the construction of objects mentioned above. The individual activities include the design, implementation and testing of individual components and the integration of these components. However, not all styles of software design will match with CPS: for example, Agile programming does not.

1.4 Validation of the choice for COS

The problem to be solved in this graduation project has been designed by IBM ILOG. It states that the model for CPS should be developed using COS and CPO. In Section 1.1 we concluded that SA-LNS (and therefore CPO) are suitable for solving RCPS-related problems.

Of course, there are other Constraint Programming toolkits available (some proprietary, some open source). Investigating all of them is infeasible in the allotted time. Therefore, some alternatives were studied superficially. Concerning proprietary software: Rossi [RVBW06] (2006) mentions (on page 157): "ILOG is the market leader in commercial constraint programming software."[1]

From studying several constraint programming libraries/toolkits (Minion, Gecode, Google CP), we can observe the following:

- There are toolkits designed as C++ or Java libraries. Others introduce their own constraint programming syntax, similar to OPL. None of them introduce constructs (allDifferent, et cetera) that are not present in OPL.

[1] Unfortunately, the page containing the references is missing in the Google Book preview

- Almost all of the studied libraries claim that they are the 'best', or the 'fastest'. For example:
  - Minion claims on their website[2] that it is faster than competing commercial solvers, including CPO. However, the Minion framework does not seem to contain the concept of interval variables. So either the claim is based on non-interval-related CP-problems or modeling scheduling problems would be an issue.
  - Gecode[3] has won all the MiniZinc Challenges[4] so far (in 2011, 2010, 2009 and 2008). However, CPO has not participated in these contests. It has been used as a benchmark to compete against, but there is no information on who modeled the problems. This is significant, because in the field of Constraint Programming, the performance of a model still depends significantly on the skills of the developer.

Another key feature of COS is the integration with ODME, a framework to incorporate OPL models in an environment with a graphical user interface (including data input and analysis of output with for example Gantt charts). Other toolkits do not feature such a framework and for example GUIs would have to be built from scratch.

From this we can conclude that there is no reason to doubt the performance of COS when compared to other Constraint Programming toolkits.

[2] http://minion.sourceforge.net/
[3] http://www.gecode.org/
[4] http://www.g12.cs.mu.oz.au/minizinc/challenge2011/challenge.html

Chapter 2

The Complex Project Scheduling Problem

The goal of this theis is to develop an optimization model for the Complex Project Scheduling Problem, which will be defined in this chapter. Section 2.0 informally describes CPS and lists the requirements CPS adheres to. A formal definition of CPS is given in Section 2.1. We use this mathematical model to prove that the CPSP is NP-hard in Section 2.2.

2.0 Informal Description

CPS is an extension of RCPS. Therefore we are already familiar with a couple of concepts that CPS will have to cover: activities, resources and precedence relations. Research of the literature revealed other relevant concepts: skills, non-renewables (consumables), preemption, et cetera. Other extensions include availability calendars and transition matrices.

Figure 2.1 displays the various concepts in CPS and which of them are related. The figure does not cover all the concepts. For example, it does not demonstrate preemption of activities, or the hierarchical design of equipment and activities. The details are listed in the requirements section below and furthermore in the formal definition in Section 2.1 and in Chapter 3.

An optimization model describes the constraints the solution has to adhere to. Usually, there are many solutions for a problem and it is not hard to find one. But, we want to find a good solution, or even an optimal solution. To express how the solution should be optimized, an objective function is specified. CPS supports the typical 'makespan' objective function, also found in RCPS. As an extension, CPS features a weighted function (for optimization) of various costs: employee costs (salary), tardiness costs (penalty for finishing a task late), et cetera. The formal definition can be found in Section 2.1.4.

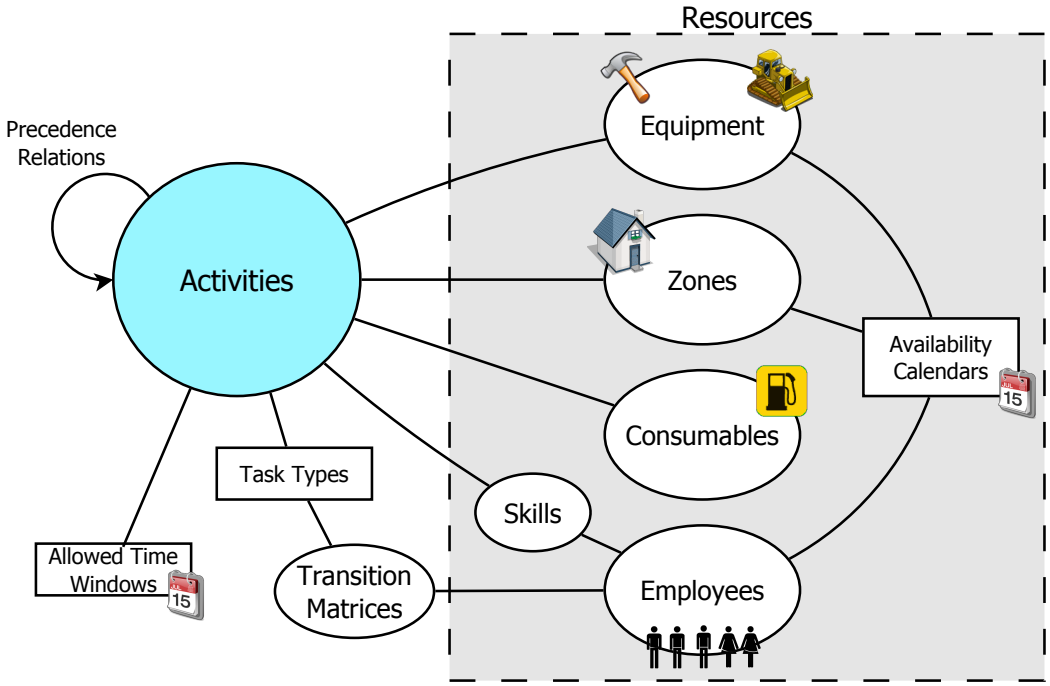

Figure 2.1: *Overview of CPS concepts.*

2.0.0 Requirements

This section formulates the requirements used to define the CPS model. The requirements are formulated in a traditional software engineering fashion, which makes it easy to verify whether a requirement has been met or not.

The requirements dealing with "zones" specifically originate from the desire to be able to fit aircraft construction scheduling into our model. So it is possible, for example, to model that a maximum of 3 people can work in a cockpit at the same time (the cockpit would be a zone with capacity 3). Also, because of safety, we need to be able to model that while tasks are executed on top of a wing, no employees are allowed to be under that wing (because of safety). This is refered to as "disabling a zone".

Rail maintenance scheduling is the cause of the requirements dealing with the possibility to split tasks into parts.

The requirements below are grouped in the categories *Resources*, *Activities*, *Activities using Resources*, and *Objectives*.

Read "CPS" below as "The model for CPS".

Resources

These requirements deal with the various types of resources in CPS: employees, equipment, zones, and consumables.

R0	CPS can model employees.
R0.a	An employee has a set of skills.
R0.b	An employee has an availability calendar.
R1	The output of CPS supports an overview of activities per employee.
R2	CPS can model equipment.
R2.a	Equipment can be modeled hierarchically (equipment can contain other equipment).
R2.b	Equipment has an availability calendar.
R3	CPS can model zones.
R3.a	Zones can be modeled hierarchically (zones can contain other zones).
R3.b	A zone has an availability calendar.
R4	CPS can model consumables (materials that can be produced and consumed).

Activities

The requirements below describe the concepts for activities in CPS.

R5	CPS can model activities.
R5.a	Activities can be modeled hierarchically (an activity can consists of one or more sub-activities). An activity which does not consist of sub-activities is called a task.
R6	It is possible to specify precedence relations on activities.
R7	Tasks have a duration.
R8	Tasks can be split into parts.
R8.a	A minimal and maximal duration of parts can be specified.
R8.b	A minimal and maximal distance in time can be specified between parts belonging to the same task.
R9	A set of time windows in which an activity can be executed can be specified.
R10	It is possible to set a due date for an activity.
R11	It is possible to specify alternative tasks for a task.

Activities using Resources

These requirements describe how activities are coupled with the resources.

R12	A task can require employees with skill sets.
R13	A task can require equipment.
R13.a	Equipment contained by the required equipment is also required.
R14	A task can require capacity in a zone.
R14.a	When capacity is requested in a zones, that capacity is also requested in the 'ancestor' zones (the zones above the requested zone in the hierarchy of zones).
R15	A task can disable a zone (a task using that zone cannot take place during the disabling task).
R15.a	It can be specified that contained zones are also disabled if a zone is disabled.
R16	A task can be assigned a task type.
R17	It is possible to specify transition matrices of minimal distances in time between task types.
R17.a	An employee can be assigned a transition matrix.
R17.b	When an employee is assigned two parts of (possible different) tasks in sequence, the minimal distance specified in the transition matrix of that employee is enforced.
R18	An activity can produce and consume consumables.

Objectives

The requirements below describe the objective functions that are available in CPS.

R19	CPS supports the concept 'makespan' (finish activities as soon as possible).
R20	CPS supports the minimization of the sum of costs (see below).
R20.a	Employees can be assigned a salary, based on skill.
R20.b	Equipment can be assigned a usage cost.
R20.c	Zones can be assigned a usage cost.
R20.d	Activities can be assigned a fixed cost.
R20.e	Activities can be assigned a cost incurred when they are not executed.
R20.f	Activities can be assigned a tardiness cost, which is incurred when the activity finishes after the specified due date of that activity.
R20.g	The transition matrices include a transition cost.
R20.h	A precedence relation between two projects can have a distance (in time) cost.

Non-functional Requirements

This graduation project is not just about developing an optimization model. There are several other important aspects:

- *Data-format of the input.* It has to be generic enough to suit the shapes of the different problems (more on this in Section 1.3).
- *Format of the output.* The output format can range from one 'simple' Gantt chart giving an overview of which activities will be executed when, by what resource. This output is usually formatted in some form of 'abstract time-units' and should be converted to actual calendars in order to be useful in practice.
- *Performance.* If it takes CPO weeks to find a proper solution, its usefulness in practice will be limited. At the moment, we are aiming for a maximum of 12 hours of calculation time to find satisfactory solutions, so the schedule can be made overnight.
- *GUI.* CPO and matching data-formats are practically useless without a way to present it to the end-user. Besides COS, IBM has software built as a layer on top of it: IBM ILOG Optimization Decision Manager Enterprise (ODME). ODME allows us to generate an application including a GUI, the underlying model, the CPO algorithm and the output.

2.1 Formal definition

Using the requirements which were defined in the previous section, a formal definition of the CPSP is constructed in this section. They define what the CPS model takes as input, which constraints apply and what its output is.

Definition An instance of the Complex Project Scheduling Problem (CPSP) consists of the sets and functions as described in Section 2.1.1. The problem is to find a solution, as described in Section 2.1.2, respecting the constraints specified in Section 2.1.5, optimizing the objective function specified in Section 2.1.4.

2.1.1 Input Data

The input data for the CPS model consists of a set of skills (S), a set of employees (E), a set of equipment (Q), a set of consumables (C), a set of projects (P), and a set of precedences (R). After some initial notes and clarifications, each topic is discussed separately. Transition matrices link employees and projects and do not form a set themselves; the concept is discussed last.

Notes up front:

- $\mathbb{B}$ are the Booleans `true` and `false`.
- The natural numbers $\mathbb{N}$ include 0. $\mathbb{N}^*$ excludes 0.
- The non-negative real numbers (so including 0) are denoted $\mathbb{R}_{\geq 0}$.

Description of the format for functions:

- $function : A \times B \to C$
  Description of the function with arguments A and B and result C (which are all sets) possibly constraining the domains of A, B and C.

Functions are used like this: $function(a,\ b)$

The model deals with moments in time and durations in a certain time unit (imagine minutes). A moment in time is a value in $\mathbb{N}$. A duration is also a value in $\mathbb{N}$.

Two moments in time describing the start and end of an activity form a time window. $\mathbb{W}$ (for time **w**indows) is the set of all possible intervals $[a, b)$ with $a, b \in \mathbb{N}$ and $a <= b$. Functions:

- $start : \mathbb{W} \to \mathbb{N}$ gives for a time window $[a, b) \in \mathbb{W}$, $start([a, b)) = a$.
- $end : \mathbb{W} \to \mathbb{N}$ gives for a time window $[a, b) \in \mathbb{W}$, $start([a, b)) = b$.

Frequently, sets of time windows are used. In a set of time windows, no two time windows overlap (which in enforced by the assertions, see Appendix A).

Skills

S is the set of skills. Employees (next section) have skills and tasks (subset of projects, see below) can request employees with sets of skills). Unlike the other sets, skills do not have any attributes relevant to a mathematical description. In the implementation, they obviously carry a identifier and a name.

Employees

E is the set of employees. An employee has skills, a cost when set to work (salary), and a calendar defining its availability.

- $\mathit{skills} : E \to \mathcal{P}(S)$ gives the set of skills the employee $e \in E$ has.
- $\mathit{cost} : E \times S \to \mathbb{R}_{\geq 0}$ gives the cost for putting employee $e \in E$ to work with skill $s \in S$ for one unit of time. $\mathit{cost}(e, s)$ is only defined for $s \in \mathit{skills}(e)$.
- $\mathit{availability} : E \to \mathcal{P}(\mathbb{W})$ gives the set of time windows that determine when an employee $e \in E$ is available for work.

Equipment

Q is the set of available equipment. Equipment has function over time describing the number of available units, and a cost when used in tasks (subset of projects). Note that a zone with capacity (mentioned in the requirements, Section 2.0.0) can modeled as equipment, for example: "Zone A has capacity X" is very different from "There are X hammers".

- $\mathit{amount} : Q \times \mathbb{N} \to \mathbb{N}$ describes the number of instances available of equipment $q \in Q$ at time $t \in \mathbb{N}$.
- $\mathit{cost} : Q \to \mathbb{R}_{\geq 0}$ gives the cost for using an instance of this equipment $q \in Q$ for one unit of time.

Note that this formal definition does not satisfy the requirements that states equipment can be modeled hierarchically. This concept was kept out of the mathematical model to avoid recursion.

Consumables

C is the set of consumables. Consumables can be produced and consumed by activities. A consumable has an initial level, a lowerbound and an upperbound.

- $\mathit{initialAmount} : C \to \mathbb{N}$ gives the initial 'level' of a consumable $c \in C$.
- $\mathit{lowerbound} : C \to \mathbb{N}$ is the level that consumable $c \in C$ may not drop below.
- $\mathit{upperbound} : C \to \mathbb{N}$ is the level that consumable $c \in C$ may not exceed.

Projects

P is the set of projects, which is designed hierarchically. The hierarchy allows us to logically group tasks together. This is for example convenient when defining precedence relations (see below). A project which is a leaf in the hierarchy is called a task; a project which has 'child-projects' is called a module.

- $\mathit{isRoot} : P \to \mathbb{B}$ returns `true` if and only if $p \in P$ has no parent (it is a root element).
- $\mathit{parent} : P \to P$ returns the parent project $p' \in P$ of project $p \in P$. $\mathit{parent}(p)$ is not defined if $\mathit{isRoot}(p)$.

- *isAlternativeNode* : $P \to \mathbb{B}$ returns `true` if the children of $p \in P$ form an alternative construction. It returns `false` if p should span its children. *isAlternativeNode*(p) is not defined if *isTask*(p) (see Section 2.1.3).
- *forcePresence* : $P \to \mathbb{B}$ is set to `true` if root project $p \in P$ and all of its children have to be present. It returns `false` if root project $p \in P$ and its children do not have to be present. *forcePresence*(p) is defined if and only if *isRoot*(p).

Functions about restricting a project in time:

- *allowedWindows* : $P \to \mathcal{P}(\mathbb{W})$ is the set of time windows in which project $p \in P$ is allowed to be scheduled.
- *dueDate* : $P \to \mathbb{N}$ gives for $p \in P$ the moment in time after which *tardinessCost*(p) is incurred.

Functions dealing with costs:

- *fixedCost* : $P \to \mathbb{R}_{\geq 0}$ is the fixed cost for inclusion of project $p \in P$ in the schedule.
- *unperformedCost* : $P \to \mathbb{R}_{\geq 0}$ is the cost incurred for not scheduling project $p \in P$.
- *tardinessCost* : $P \to \mathbb{R}_{\geq 0}$ is the cost incurred per time unit for project $p \in P$ finishing after *dueDate*(p).

T is the set of tasks: $\{p \in P \mid \mathit{isTask}(p)\}$. $\Theta \subset \mathbb{N}$ is the set of task types (used for transitions, see below). Tasks are split into one or more parts (preemption). The number of parts, their length, et cetera, are constrained by the functions below.

To prevent a split into an infinite number of 0-duration parts, parts are at least duration 1. So *minFirstPartDuration*$(t) > 0$, *minLastPartDuration*$(t) > 0$,
minMiddlePartsDurations$(t) > 0$ if *duration*$(t) > 0$. A 0-duration task has exactly one 0-duration part and
minFirstPartDuration$(t) = 0$, *minLastPartDuration*$(t) = 0$,
and *minMiddlePartsDurations*$(t) = 0$.

Functions for tasks and preemption:

- *duration* : $T \to \mathbb{N}$ gives the duration of task $t \in T$.
- *taskType* : $T \to \Theta$ gives the type of task $t \in T$, used for transitions between parts.
- *maxPartDuration* : $T \to \mathbb{N}^*$ is the maximal duration of the parts that form task $t \in T$.
- *minFirstPartDuration* : $T \to \mathbb{N}$ is the minimal duration of the first part of task $t \in T$.
- *minLastPartDuration* : $T \to \mathbb{N}$ is the minimal duration of the last part of task $t \in T$.
- *minMiddlePartsDurations* : $T \to \mathbb{N}$ is the minimal duration of each of the middle parts (neither first nor last) of task $t \in T$.
- *minPartSeparation* : $T \to \mathbb{N}$ gives the minimal distance between each pair of parts of task $t \in T$. 0 can be considered as a default.
- *maxPartSeparation* : $T \to \mathbb{N}$ gives the maximal distance between each pair of parts of task $t \in T$. ∞ can be considered as a default.

Tasks can require employees and equipment. Both tasks and modules can consume and produce consumables.

- *needsNEmployeesWithSkillset* : $T \times \mathcal{P}(S) \to \mathbb{N}$ gives the number of employees with skillset $ss \subseteq S$ that task $t \in T$ needs.
- *needsNEquipment* : $T \times Q \to \mathbb{N}$ gives the number of pieces of equipment $q \in Q$ that task $t \in T$ needs.
- *disablesEquipment* : $T \times Q \to \mathbb{B}$ is `true` if and only if the execution of task $t \in T$ prevents the use of equipment $q \in Q$. See Section 3.1.5 for a detailed explanation.
- *producesXOfConsumable* : $P \times C \times \mathbb{B} \to \mathbb{Z}$ is the quantity of consumable $c \in C$ that project $p \in P$ produces (result is positive) or consumes (result is negative). The moment of production/consumption is determined by the boolean $b \in \mathbb{B}$. $b = true$ means 'at the start'. $b = false$ means 'at the end'.

Precedences

Precedences can be defined on projects, to determine an ordering among them. There are several types of precedences: Φ is the set of precedence types:

$$\{\texttt{endBeforeEnd}, \texttt{startBeforeStart}, \texttt{startBeforeEnd}, \texttt{endBeforeStart}\}.$$

R is the set of precedences. The placement of projects in the schedule can be further constrained by setting a minimum or maximum delay between them. Also, a cost can be associated with the distance between the two projects.

- *project*1 : $R \to P$ gives the first project involved in precedence $r \in R$.
- *project*2 : $R \to P$ gives the second project involved in precedence $r \in R$.
- *precendenceType* : $R \to \Phi$ is the type of precedence between the two projects.
- *minDelay* : $R \to \mathbb{N}$ sets the minimum delay between the edges of the projects based on *precedenceType*(r).
- *maxDelay* : $R \to \mathbb{N}$ sets the maximum delay between the edges of the projects based on *precedenceType*(r).
- *distanceCost* : $R \to \mathbb{R}_{\geq 0}$ is the cost incurred per time unit by the distance (in time) between the edges of the two projects, based on *precedenceType*(r).

Transitions

To model transition time and cost between parts of tasks, transitions matrices are introduced. Transition matrices are based on task types Θ (see *taskType*(t)) and can be different for every employee. For example, if you consider task types to be locations, it allows you to express, for example, that task t_1 is of type 'Location A' and task t_2 is of type 'Location B' and it takes an employee 10 minutes (and it costs 10 euros) to travel between the two location. This time and cost is incurred between every **part** of the tasks.

- *transitionTime* : $E \times \Theta \times \Theta \to \mathbb{N}$ gives the transition time between task types $\theta_1 \in \Theta$ and $\theta_2 \in \Theta$ for employee $e \in E$.
- *transitionCost* : $E \times \Theta \times \Theta \to \mathbb{R}_{\geq 0}$ gives the transition cost between task types $\theta_1 \in \Theta$ and $\theta_2 \in \Theta$ for employee $e \in E$.

2.1.2 Output Data

A solution to an instance of the CPSP is:

- For every $p \in P$, an assignment to $isPresent(p)$ and $window(p)$.
- For every $tp \in TP$ (task part, see below), an assignment to $window(tp)$, $employees(tp)$ and $skillsetOf(tp, e)$ for each $e \in employees(tp)$.

The definitions of the functions are given below. The solution has to adhere to the constraints specified in Section 2.1.5.

Part of the solution is an assignment of each project $p \in P$ to an interval in time (possibly void) respecting the constraints. As tasks (leaf elements of the project hierarchy) can be split into parts (preemption), each part has to be assigned an interval in time as well. The placement of modules (non-task projects) can be deduced from the placement of the tasks, but is part of the output for clarity. TP is the set of task parts that are 'created' to perform the tasks.

Because we need to know which employees are responsible for executing each part of each task, that information is part of the solution as well: each $tp \in TP$ has a set of tuples $\{\langle e, ss\rangle\}$, with $e \in E$ and $ss \subseteq S$, describing which employees are involved (via which skill set).

Projects P have the following functions defined:

- $isPresent : P \to \mathbb{B}$ returns whether or not project $p \in P$ is present in the solution.
- $window : P \to \mathbb{W}$ returns the assigned time window of the project.

Task parts TP have the following functions:

- $window : TP \to \mathbb{W}$ returns the assigned time window of the task part.
- $task : TP \to T$ gives task $t \in T$ the task part tp belongs to.
- $indexOf : TP \to \mathbb{N}$ returns the index in $[0, maxNrOfParts(task(tp)) - 1]$ identifying the part within the scope of the task.
- $employees : TP \to \mathcal{P}(E)$
  Employees responsible of the task part $tp \in TP$.
- $skillsetof : TP \times E \to \mathcal{P}(S)$
  Skill set that shows the responsibilities of employee $e \in E$ in task part $tp \in TP$. Defined if and only if $e \in employees(tp)$.

2.1.3 Helper Functions

This section lists functions that are neither part of the input nor of the output of the model. Their result values are deduced from core functions. The helper functions have been defined because their functionality is reused often or to make the constraints and objective functions in the next sections easier to understand.

Derived from Input

Skillset Cost $cost : E \times \mathcal{P}(S) \to \mathbb{R}_{\geq 0}$ gives the cost for putting employee $e \in E$ to work with skill set $ss \in \mathcal{P}(S)$. It is calculated by taking the maximum[1] of the $cost(s)$ for the single skills s in the set ss.

Project Children $children : P \to \mathcal{P}(P)$ gives the child projects of project $p \in P$. This function is derived from the *parent* function.

Project is Task $isTask : P \to \mathbb{B}$ returns `true` if and only if $p \in P$ has no children (it is a leaf element), that is, $children(p) = \varnothing$.

Max Number of Parts $maxNrOfParts : T \to \mathbb{N}$ gives the maximal number of parts that task $t \in T$ can consist of. This function is not part of the input, it is deduced using $minFirstPartDuration(t)$, $minLastPartDuration(t)$ and $minMiddlePartsDurations(t)$. Returns 1 if $duration(t) = 0$.

Based on Output

Task Part $isPresent : TP \to$ `true` returns true. A task part is always present. If it is not necessary, it does not exist. Some functions below work on both projects and task parts using *isPresent* on projects. To avoid cloning the functions for task parts, *isPresent* is defined for task parts here.

Interval Start $startOf : (P \cup TP) \to \mathbb{T}$ for $x \in P \cup TP$, short for $start(window(x))$. Defined if and only if $isPresent(x)$.

Interval End $endOf : (P \cup TP) \to \mathbb{T}$ for $x \in P \cup TP$, short for $end(window(x))$. Defined if and only if $isPresent(x)$.

Interval Duration $length : (P \cup TP) \to \mathbb{N}$ for $x \in P \cup TP$, short for $endOf(x) - startOf(x)$. Defined if and only if $isPresent(x)$.

Intervals overlap $overlap : (P \cup TP) \times (P \cup TP) \to \mathbb{B}$ determines whether two $x_1, x_2 \in P \cup TP$ overlap in time.
Returns $isPresent(x_1) \ \wedge \ isPresent(x_2) \ \wedge$
$(\ endOf(x_1) \leq startOf(x_2) \ \vee \ endOf(x_2) \leq startOf(x_1) \ )$

Intervals coincide $coincide : (P \cup TP) \times (P \cup TP) \to \mathbb{B}$
Determines whether two intervals $x_1, x_2 \in P \cup TP$ coincide in time (start and end match).
Returns $isPresent(x_1) \ \wedge \ isPresent(x_2) \ \wedge$
$startOf(x_1) = startOf(x_2) \ \wedge \ endOf(x_1) = endOf(x_2)$.

[1] In the implementation, the configuration parameter 'costs.skillsetCost.useSumInsteadOfMax' can be used to instruct the model to use 'sum' instead of 'max'.

Task Parts $parts : T \to \mathcal{P}(TP)$ gives the task part belonging to the task $t \in T$.

Task type shorthand $type : TP \to \Phi$ returns the task type of the task this part $tp \in TP$ belongs to. Short for $taskType(task(tp))$.

Task Part sequencing 1 $isAfter : TP \times TP \to \mathbb{B}$
Determines if tp_2 is scheduled after tp_1.
Returns $startOf(tp_2) \geq endOf(tp_1)$

Task Part sequencing 2 $isSequencedRightAfter : E \times TP \times TP \to \mathbb{B}$
Determines if tp_2 is right after tp_1 in the sequence of tasks $e \in E$ is involved in. No other task parts are in between.
Returns $isAfter(tp_1, tp_2) \ \wedge \ e \in employees(tp_1) \ \wedge \ e \in employees(tp_2) \ \wedge$
$\neg \exists\, tp_3 \in TP \mid e \in employees(tp_3) : isAfter(tp_1, tp_3) \ \wedge \ isAfter(tp_3, tp_2)$.

Precedence distance $distance : R \to \mathbb{N}$
For precedence $r \in R, p_1 = project1(r), p_2 = project2(r)$:
If $precedenceType(r) = endBeforeEnd$, returns $endOf(p_2) - endOf(p_1)$
If $precedenceType(r) = startBeforeStart$, returns $startOf(p_2) - startOf(p_1)$
If $precedenceType(r) = startBeforeEnd$, returns $endOf(p_2) - startOf(p_1)$
If $precedenceType(r) = endBeforeStart$, returns $startOf(p_2) - endOf(p_1)$
Defined if and only if $isPresent(p_1) \wedge isPresent(p_2)$.

Linking Input and Output

Employee is available $isAvailable : E \times TP \to \mathbb{B}$ determines if employee $e \in E$ is available for the duration of task part $tp \in TP$ (according to her availability calendar). So, it returns
$\exists\, w \in availability(e) : start(w) \leq startOf(tp) \ \wedge \ end(w) \geq endOf(tp)$

Project in allowed window $inAllowedWindow : P \to \mathbb{B}$ determines whether project $p \in P$ lies within one of its allowed windows. So, it returns
$\exists\, w \in allowedWindows(p) : start(w) \leq startOf(p) \ \wedge \ end(w) \geq endfOf(p)$

Other Functions

Boolean to natural number $b2n : \mathbb{B} \to \mathbb{N}$ returns for $b \in \mathbb{B}$, 1 if $b =$ `true`, 0 if $b =$ `false`.

2.1.4 Objective Function

The objective funtion of CPSP, the value to be minimized by the optimization process, is defined as follows (W_0 through W_7 (all $\in \mathbb{R}_{\geq 0}$) are configurable weights):

$$
\begin{aligned}
& W_0 \cdot \mathit{makespanCosts} \\
& + W_1 \cdot \mathit{fixedCosts} + \\
& + W_2 \cdot \mathit{unperformedCosts} \\
& + W_3 \cdot \mathit{tardinessCosts} \\
& + W_4 \cdot \mathit{employeeCosts} \\
& + W_5 \cdot \mathit{equipmentCosts} \\
& + W_6 \cdot \mathit{distanceCosts} \\
& + W_7 \cdot \mathit{transitionCosts}
\end{aligned}
$$

makespanCosts is defined as:

$$MAX\ p \in P \mid \mathit{isPresent}(p) : \mathit{endOf}(p)$$

fixedCosts is defined as:

$$\sum p \in P \mid \mathit{isPresent}(p) : \mathit{fixedCost}(p)$$

unperformedCosts is defined as:

$$\sum p \in P \mid !\mathit{isPresent}(p) : \mathit{unperformedCost}(p)$$

tardinessCosts is defined as:

$$
\begin{aligned}
\sum p \in P \mid & \mathit{isPresent}(p) \wedge \mathit{endOf}(p) > \mathit{dueDate}(p) \\
& : \mathit{tardinessCost}(p)\ \cdot\ (\mathit{endOf}(p) - \mathit{dueDate}(p))
\end{aligned}
$$

employeeCosts is defined as:

$$\sum \mathit{tp} \in \mathit{TP} : \sum e \in \mathit{employees}(\mathit{tp}) : \mathit{length}(\mathit{tp}) \cdot \mathit{cost}(e, \mathit{skillsetof}(\mathit{tp}, e))$$

equipmentCosts is defined as:

$$
\begin{aligned}
\sum t \in T \mid & \mathit{isPresent}(t) \\
& : \sum q \in Q : \mathit{duration}(t) \cdot \mathit{needsNEquipment}(t, q) \cdot \mathit{cost}(q)
\end{aligned}
$$

distanceCosts is defined as:

$$
\begin{aligned}
\sum r \in R \mid & \mathit{isPresent}(\mathit{project1}(r)) \wedge \mathit{isPresent}(\mathit{project2}(r)) \\
& : \mathit{distanceCost}(r) \cdot \mathit{distance}(r)
\end{aligned}
$$

transitionCosts is defined as:

$$
\begin{aligned}
\sum e \in E : \sum \mathit{tp}_1, \mathit{tp}_2 \in \mathit{TP} \mid & e \in \mathit{employees}(\mathit{tp}_1)\ \wedge\ e \in \mathit{employees}(\mathit{tp}_2) \\
& \wedge \mathit{isSequencedRightAfter}(e, \mathit{tp}_1, \mathit{tp}_2) \\
& : \mathit{transitionCost}(e, \mathit{type}(\mathit{tp}_1), \mathit{type}(\mathit{tp}_2))
\end{aligned}
$$

2.1.5 Constraints

The constraints to which the solutions of the CPSP must adhere are listed in this section.

Employees

C.E.0: Employees are only assigned to a task part when they are available (calendar):

$$\forall\, tp \in TP : \forall\, e \in employees(tp) : isAvailable(e, tp)$$

C.E.1: Employees can only participate in one task part at a time:

$$\begin{aligned}\forall\, tp_1, tp_2 \in TP \mid\ & tp_1 \neq tp_2 \wedge overlap(tp_1, tp_2) \\ & : employees(tp_1) \cap employees(tp_2) = \varnothing\end{aligned}$$

C.E.2: Tasks are assigned the requested employees:

$$\begin{aligned}\forall\, tp \in TP : \forall\, & ss \in \mathcal{P}(S) \\ & needsNEmployeesWithSkillset(task(tp), ss) \\ & = |\{e \in employees(tp) \mid skillsetof(tp, e) = ss \wedge ss \subseteq skills(e)\}|\end{aligned}$$

C.E.3: Enforce transition times between task parts (executed by employees):

$$\begin{aligned}\forall\, e \in E : \forall\, tp_1, tp_2 \in TP \mid\ & e \in employees(tp_1)\ \wedge\ e \in employees(tp_2) \\ & \wedge\, isSequencedRightAfter(e, tp_1, tp_2) \\ & : startOf(tp_2) \geq \\ & \quad endOf(tp_1) + transitionTime(e, type(tp_1), type(tp_2))\end{aligned}$$

Equipment

C.Q.0: Do not use more equipment than exists (for every point in time):

$$\begin{aligned}\forall\, q \in Q, t \in \mathbb{N} : & amount(q, t) \\ & \geq \sum tp \in TP \mid isPresent(tp) \wedge startOf(tp) \leq t < endOf(tp) \\ & \qquad : needsNEquipment(task(tp), q)\end{aligned}$$

C.Q.1: Equipment is not used at a time it is disabled by a task:

$$\begin{aligned}\forall\, q \in Q : \forall\, & tp_1 \in TP \mid needsNEquipment(task(tp_1), q) > 0 \\ & : \forall\, tp_2 \in TP \mid disablesEquipment(task(tp_2), q) : !overlap(tp_1, tp_2)\end{aligned}$$

Consumables

C.C.0: There is always more of a consumable than *lowerbound* and less than *upperbound*:

$$\begin{aligned}\forall\, c \in C, t \in \mathbb{N} : & lowerbound(c) \\ & \leq initialAmount + \\ & \quad \sum p \in P, b \in \mathbb{B} \mid isPresent(p) \\ & \qquad \wedge ((b \wedge startOf(p) \leq t) \vee (!b \wedge endOf(p) \leq t)) \\ & \qquad : producesXOfConsumable(p, c, b) \\ & \leq upperbound(c)\end{aligned}$$

Projects

C.P.0: Projects are scheduled in their allowed time windows:

$$\forall\, p \in P \mid isPresent(p) : inAllowedWindow(p)$$

C.P.1: Check *forcePresence* for root projects:

$$\forall\, p \in P \mid isRoot(p) \wedge forcePresence(p) : isPresent(p)$$

C.P.2: Alternative constructions: 1 child is present if project is present and its interval coincides with the project interval:

$$\begin{aligned}\forall\, p \in P \mid\ & !isTask(p) \wedge isAlternativeNode(p) \\ & : b2n(isPresent(p)) = |\{c \in children(p) \mid isPresent(c)\}| \\ & \quad \wedge \\ & \quad isPresent(p) \Rightarrow \exists\, c \in children(p) \mid isPresent(c) : coincide(p, c)\end{aligned}$$

C.P.3: Span constructions: project spans its children:

$$\begin{aligned}\forall\, p \in P \mid\ & !isTask(p) \wedge\ !isAlternativeNode(p) \\ & : \forall\, c \in children(p) : isPresent(c) = isPresent(p) \\ & \quad \wedge \\ & \quad isPresent(p) \Rightarrow \\ & \quad\ startOf(p) = MIN\ \ c \in children(p) : startOf(c) \\ & \quad\quad \wedge \\ & \quad\ endOf(p) = MAX\ \ p \in children(c) : endOf(c)\end{aligned}$$

Preemption

C.P.4: Tasks span their parts:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \\ & : isPresent(p) \Rightarrow \\ & \quad startOf(p) = MIN\ \ tp \in parts(p) : startOf(tp) \\ & \quad\quad \wedge \\ & \quad endOf(p) = MAX\ \ tp \in parts(p) : endOf(tp)\end{aligned}$$

C.P.5: Task parts are sequenced:

$$\begin{aligned}\forall\, tp_1, tp_2 \in TP \mid\ & task(tp_1) = task(tp_2) \wedge isPresent(task(tp_1)) \\ & \quad \wedge indexOf(tp_1) = indexOf(tp_2) - 1 \\ & : endOf(tp_1) \leq startOf(tp_2)\end{aligned}$$

C.P.6: The sum of the length of the parts is equal to the duration of the task:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \wedge isPresent(p) \\ & : duration(p) = \sum tp \in parts(p) : length(tp)\end{aligned}$$

C.P.7: Check the minimal duration of the first part of a task:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \wedge isPresent(p) \\ & : \forall\, tp \in parts(p) \mid indexOf(tp) = 0 : length(tp) \geq minFirstPartDuration(p)\end{aligned}$$

C.P.8: Check the minimal duration of the middle parts of a task:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \wedge isPresent(p)\\ & : \forall\, tp \in parts(p) \mid indexOf(tp) > 0\\ & \qquad \wedge \exists\, tp' \in parts(p) : indexOf(tp') > indexOf(tp) : isPresent(tp')\\ & \qquad : length(tp) \geq minMiddlePartsDurations(p)\end{aligned}$$

C.P.9: Check the minimal duration of the last part of a task:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \wedge isPresent(p)\\ & : \forall\, tp \in parts(p)\\ & \qquad \mid !\exists\, tp' \in parts(p) : indexOf(tp') > indexOf(tp) : isPresent(tp')\\ & \qquad : length(tp) \geq minLastPartDuration(p)\end{aligned}$$

C.P.10: Check the maximal duration of the parts of a task:

$$\begin{aligned}\forall\, p \in P \mid\ & isTask(p) \wedge isPresent(p)\\ & : \forall\, tp \in parts(p) : length(tp) \leq maxPartDuration(p)\end{aligned}$$

C.P.11: Check *minPartSeparation* and *maxPartSeparation*:

$$\begin{aligned}\forall\, tp_1, tp_2 \in TP \mid\ & task(tp_1) = task(tp_2)\ \wedge\ indexOf(tp_1) = indexOf(tp_2) - 1\\ & : endOf(tp_1) + minPartSeparation(task(tp)) \leq startOf(tp_2)\\ & \qquad \wedge\\ & \quad endOf(tp_1) + maxPartSeparation(taks(tp)) \geq startOf(tp_2)\end{aligned}$$

Precedences

C.R.0: Precedences `endBeforeEnd`:

$$\begin{aligned}\forall\, r \in R \mid\ & precedenceType(r) = endBeforeEnd\\ & \wedge isPresent(project1(r) \wedge isPresent(project2(r)\\ & : endOf(project1(r)) \leq endOf(project2(r))\\ & \wedge minDelay(r) \leq endOf(project2(r)) - endOf(project1(r)) \leq maxDelay(r)\end{aligned}$$

C.R.1: Precedences `startBeforeStart`:

$$\begin{aligned}\forall\, r \in R \mid\ & precedenceType(r) = startBeforeStart\\ & \wedge isPresent(project1(r) \wedge isPresent(project2(r)\\ & : startOf(project1(r)) \leq startOf(project2(r))\\ & \wedge minDelay(r) \leq startOf(project2(r)) - startOf(project1(r)) \leq maxDelay(r)\end{aligned}$$

C.R.2: Precedences `startBeforeEnd`:

$$\begin{aligned}\forall\, r \in R \mid\ & precedenceType(r) = startBeforeEnd\\ & \wedge isPresent(project1(r) \wedge isPresent(project2(r)\\ & : startOf(project1(r)) \leq endOf(project2(r))\\ & \wedge minDelay(r) \leq endOf(project2(r)) - startOf(project1(r)) \leq maxDelay(r)\end{aligned}$$

C.R.3: Precedences `endBeforeStart`:

$$\begin{aligned}\forall\, r \in R \mid\ & precedenceType(r) = endBeforeStart\\ & \wedge isPresent(project1(r) \wedge isPresent(project2(r)\\ & : endOf(project1(r)) \leq startOf(project2(r))\\ & \wedge minDelay(r) \leq startOf(project2(r)) - endOf(project1(r)) \leq maxDelay(r)\end{aligned}$$

2.2 Computational Complexity

It is well-known that the RCPSP is proven to belong to the class of NP-hard problems. For example, [RBL07] states: "It has been shown by Blazewicz et al. [3] that the RCPSP, as a generalization of the classical job shop scheduling problem, belongs to the class of NP-hard optimization problems." ([3] in that paper is [BJ86])

For our problem at hand, we will prove its NP-hardness by viewing it as a generalization of RCPS, i.e., that all instances of the RCPSP can be mapped to an instance of the CPSP. In the previous section we have defined a mathematical model of CPS.

Formally, Resource Constrained Project Scheduling has the following input (taken from [Cra96]): a set of tasks T, a set of resources R, a capacity function $C : R \to \mathbb{N}$, a duration function $D : T \to \mathbb{N}$, a utilization function $U : T \times R \to \mathbb{N}$, and a partial order P on T. The problem is to find a solution that schedules all the tasks within the boundaries of the resources and minimizes the makespan (the end of the last task).

To clearly distinguish between the sets in RCPS and CPS, in the mapping below the sets are subscripted like this: T_{rcps} vs T_{cps}.

Tasks, Duration Create one Root Project $root \in P_{cps}$ and define $forcePresence(root) =$ `true` and $isAlternativeNode(root) =$ `false`. For every $t \in T_{rcps}$, create one $p \in P_{cps}$ and define $parent(p) = root$, and by doing so determine $children(root)$. Define $duration(p) = D_{rcps}(t)$. Define $allowedWindows(root) = [0, \infty)$.

Resources, Capacity For every $r \in R_{rcps}$, create one piece of Equipment $q \in Q_{cps}$. Define $amount(q, x) = C_{rcps}(r)$ for every x in time. Define $availability(q) = [0, \infty)$.

Utilization Map U_{rcps} directly to $needsNEquipment$. So for every $\langle t, r \rangle \in T_{rcps} \times R_{rcps}$ set, for the matching $\langle p, q \rangle \in P_{cps} \times Q_{cps}$, $needsNEquipment(p, q) = U_{rcps}(t, r)$.

Partial Order Map the partial order P_{rcps} to the precedences set R_{cps}. So for every $x, y \in T_{rcps}$ where x precedes y in the partial order P_{rcps}, create a precedence $pre_{x,y}$ and define $project1(pre_{x,y})$ and $project2(pre_{x,y})$ to be the matching projects in P_{cps} and set $precedenceType(pre_{x,y}) = endBeforeStart$.

Furthermore:

- Note that a resource in RCPS can also be a person. In CPS it not necessary to model employees however, as there are no skills involved.
- Disregard Skills (S_{cps}), Employees (E_{cps}), Consumables (C_{cps}).
- Disregard anything in CPS that has something to do with transition time, costs, or preemption.
- Some values of irrelevant functions are not mentioned explicitly above. Defining proper defaults for them is considered trivial.

Now, since RCPS can be modeled in CPS, CPS is at least as complex as RCPS and therefore NP-hard.

Chapter 3

OPL Model

In this chapter we will discuss the implementation of the CPS model in OPL. Section 3.0 describes the input format of the CPS model. Section 3.1 explains why the input format was designed that way. Those topics were split intentionally to keep the description of the input format concise. Section 3.2 discusses actual files containing the OPL code. Section 3.3 describes the output of the CPS model.

3.0 Input Data

Figure 3.1 defines the data structure of the input data for the OPL model. It was designed in ODME, because the OPL model will be integrated in ODME.

Contrary to normal use in ODME, the foreign keys are not marked as expected[1], as this has caused trouble (see Section 4.1 for an explanation). A visual indication of foreign keys (`FK`) has been added manually in a graphical editing program. Filled arrowheads represent a $0..*$ to 1 multiplicity. Non-filled arrowheads represent a $0..*$ to $0..1$ multiplicity.

The following sections each discuss a table. Attributes are only discussed if they are non-trivial. To keep the descriptions concise, we do not discuss *why* things are designed like this, Section 3.1 discusses the reasoning behind various design decisions.

[1] In ODME class diagrams with foreign keys, the attributes that are foreign keys have a different icon.

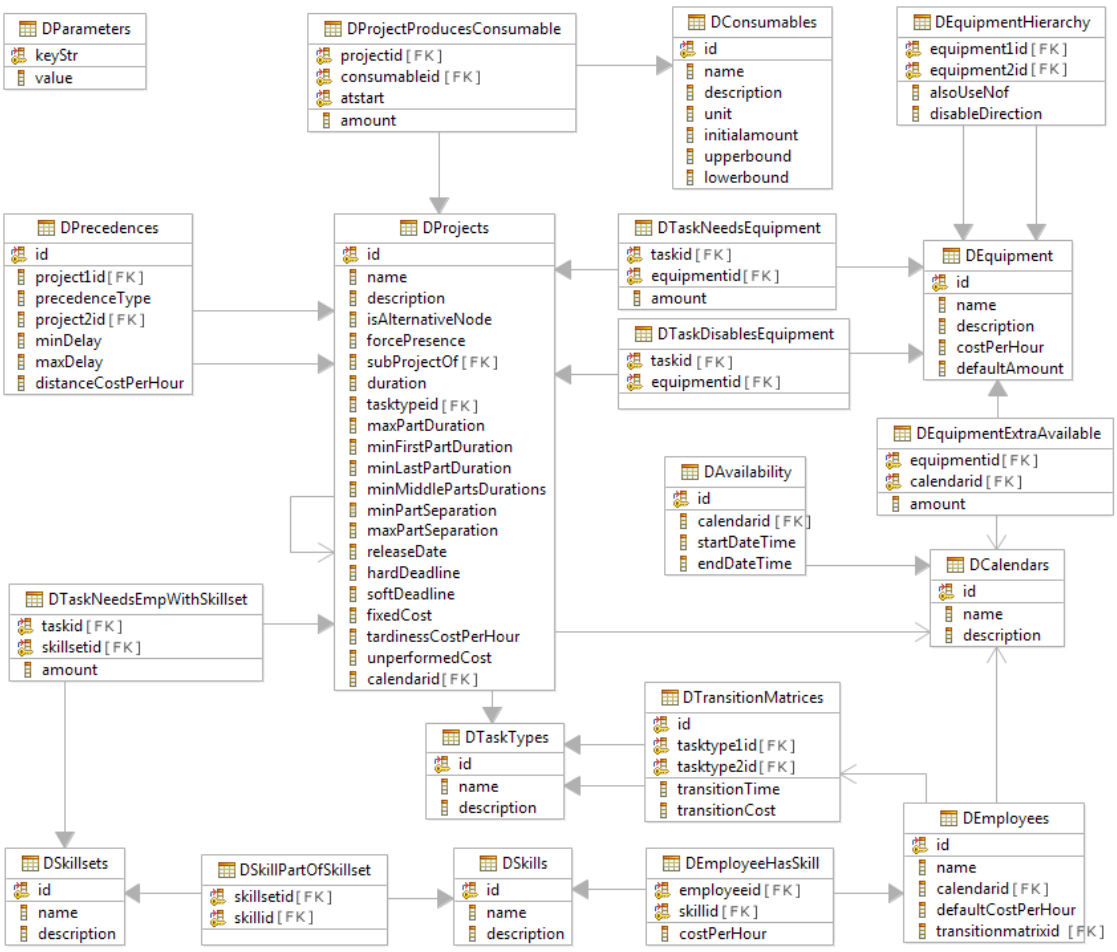

Figure 3.1: *Input data model as designed in ODME*

3.0.0 Parameters

The Parameters table contains key-value pairs of configurable parameters to the model. Because a column can only have one data type, the chosen data type is `float`, which also covers `integer` and `boolean` (0 and 1).

It supports the following keys:

optimize.timeLimit	(`int`) Time Limit (in seconds) of optimization
optimize.runMode	(`int`) 0 = nothing, 1 = makespan, 2 = total costs
optimize.maxTime	(`int`) Upper bound of TIME domain (in minutes)
costs.weights.makespan	(`float`) weight for makespan costs in total cost
costs.weights.fixed	(`float`) weight for fixed costs in total cost
costs.weights.unperformed	...
costs.weights.tardiness	...
costs.weights.employee	...
costs.weights.equipment	...
costs.weights.distance	...
costs.makespanCostPerHour	(`float`) cost associated with 1 hour of makespan
costs.skillset.useSumInsteadOfMax	(`boolean`) when determining employee cost when using a skillset, use sum instead of max over all skills involved.

optimize.maxTime is the upperbound of the domain in which intervals (projects, task parts) can be scheduled. Choosing a small upperbound (with the knowledge the solution will still fit) can benefit the performance of the model.

The option 'makespan' can also be configured by minimizing total costs and setting the weights appropriately. However, because makespan is often used in academic problems, which are usually not concerned with costs, it is offered as a separate option.

3.0.1 Calendars & Availability

Defines calendars, which are ranges of start- and end-times. These ranges are not allowed to overlap. It is possible to define a calendar without any associated availability ranges, which would mean "never available".

3.0.2 Task Types & Transition Matrices

Defines possible task types. Transition matrices are matrices between task types. Employees are assigned a transition matrix which puts requirements on transition times and cost between task parts that they execute. See Section 3.1.7 for a more detailed explanation.

transitionTime (`int`) The minimal time (in minutes) between parts of tasks of types *tasktype1id* and *tasktype2id*.

transitionCost (`int`) The cost incurred for transitioning from type *tasktype1id* to *tasktype2id*.

3.0.3 Projects

The table Projects contains the tree-structure of projects, which can have multiple roots. Non-leaf projects are called modules (so a root is a module too). Leafs are called tasks, which are split into one or more parts in the optimizer (pre-emption).

isAlternativeNode (`boolean`) (Modules only) If set to `false`, a module spans its children. If set to `true`, a module spans exactly one child and the other children are not present; the children form an alternative construction.

forcePresence (`boolean`) (Roots only) If set to `false`, the optimizer can decide not to include this part of the tree in the solution. If set to `true`, it has to be present (necessary for makespan optimizations). Section 3.1.0 explains why this is only available for roots.

duration (`int`) (Tasks only) Tasks are split into one or more parts of which the sum equals this *duration* in minutes. *duration* is ignored for modules, because their duration is determined by the child projects they span.

tasktypeid (`int`) (Tasks only) The task type of this task, used for transition matrices.

... PartDuration/Separation (`int`) (Tasks only) Defines constraints on the size of the parts of a task and how far they can be apart. For a more detailed explanation, see Section 3.1.1.

releaseDate (`Date`) Point in time after which this project has to take place (may not start before it).

softDeadline (`Date`) Point in time after which *tardinessCostPerHour* takes effect. Usually before *hardDeadline*.

hardDeadline (`Date`) Point in time before which this project has to take place (may not end after it).

fixedCost (`float`) Fixed cost of this project when it is present. TotalFixedCost is a sum of all present projects in the tree. Care has to be taken when a project tree is constructed where parent projects and child projects both have *fixedCosts*, those values are summed.

unperformedCost (`float`) Cost incurred when the project is not executed.

calendarid (`int`) The project (and its children) can only be scheduled within the time windows defined in the calendar. Introduced to work with 'Possessions' for Rail Maintenance Scheduling. -1 means "can always be scheduled" (note *releaseDate* though). See Section 3.1.2 for an explanation why both constructs are present.

3.0.4 Precedences

Defines the precedence network of projects.

precedenceType (`string`) out of the set {*synchronize,*
endAtEnd, endAfterEnd, endBeforeEnd,
startAtStart, startAfterStart, startBeforeStart,

startAtEnd, startAfterEnd, startBeforeEnd,
endAtStart, endAfterStart, endBeforeStart}.
synchronize means both *startAtStart* and *endAtEnd*.

minDelay & maxDelay (`int`) time (in minutes) that is at least or at most between the two projects. Ignored when *precedenceType* is *synchronize* or *... At ...*.

distanceCostPerHour (`float`) Cost calculated for the distance between the two projects. The 'edges' of the projects for the calculation are determined by the *precedenceType*. For example "A endBeforeStart B" with cost 60: if A ends 2 hours before B starts, 120 cost is incurred.

3.0.5 Skills, Skillsets & SkillPartOfSkillset

Defines the skills that employees can have. Skills are part of skillsets, because tasks can request that an employee has all the skills in a skillset. For a lot of applications, skills and skillsets will have a 1:1 mapping. See also Section 3.1.4.

3.0.6 Employees & EmployeeHasSkill

Defines the available employees. EmployeeHasSkill links employees with skills.

calendarid (`int`) Calendar that describes the availability of the employee. -1 means "is always available".

defaultCostPerHour (`float`) Salary of the employee. For skill-dependent salary, it can be overwritten by *costPerHour* in EmployeeHasSkill.

transitionMatrix (`int`) Transition matrix for this employee. Incurs transition time and cost between task types in the sequence of task parts this employee handles. Specify -1 to model that an employee does not have a transition matrix (transition times and costs between any pair of task types is 0).

3.0.7 TaskNeedsEmpWithSkillset

TaskNeedsEmpWithSkillset allows tasks to request employees. The requested employee needs to have ALL the skills in the requested skillset. The *costPerHour* for selecting a suitable employee is based on the individual skills in the set, for each of which a *costPerHour* is defined (either the default for the employee or an overriden value). There are two options: the *max* or *sum* over the individual skills, determined by the parameter *costs.skillset.useSumInsteadOfMax*. See Section 3.1.4 for more details on skills and skillsets.

3.0.8 Equipment, EquipmentExtraAvailable

The table Equipment contains the pieces of equipment (tools, machinery, et cetera) that are available. EquipmentExtraAvailable allows you to define there is more equipment available than the *defaultAmount* for certain periods.

costPerHour (`float`) Cost of the usage of one piece of this equipment.

calendarid (`int`) Calendar that describes the availability of these extra pieces of equipment.

Note that using this construct, it is possible to define, for example: till Sunday there are 4 hammers, after that there are 5.

3.0.9 TaskNeedsEquipment & EquipmentHierarchy

TaskNeedsEquipment allows tasks to request equipment. Equipment can be defined hierarchically in the EquipmentHierarchy, in the abstract format "if a task request Equipment A, it also requests X of Equipment B". Tasks can not take place if their requested equipment is not available. See Section 3.1.5 for an explaination as to why it is defined this way.

3.0.10 TaskDisablesEquipment

TaskDisablesEquipment allows tasks to disable all available pieces of equipment (as if there were 0 available). It is easiest to imagine this in a *zones* concept, for example: task t prevents execution of any other task that requires zone a. This construct is different from a task requesting all available pieces of equipment, because two tasks disabling a certain zone are allowed to be executed in parallel, while requesting all capacity in the zone would prevent that. See Section 3.1.5 for more details.

3.0.11 Consumables & ProjectProducesConsumable

The table Consumables defines 'materials' that are consumed or produced by projects.

unit (`string`) A word that would be used to describe the quantity. For example: 5 "liters" or 300 "screws".

amount (`int`) Amount that is produced. To specify consumption, produce a negative value.

atStart (`boolean`) Determines whether the increase/decrease happens at the start or at the end of the project.

Contrary to requesting equipment and employees, which only tasks can do, modules can produce/consume consumables.

See Section 3.1.6 for more details.

3.1 Model Design

The previous section describes the input format, but to keep it concise, we did not discuss why it is designed like it is. This section explains various concepts in more detail.

It is assumed that it is clear why basic scheduling concepts such as *Projects*, *Employees*, *Equipment*, *Precedences* and *Consumables* are present in the model. These topics are therefore not discussed below.

3.1.0 The Project tree

In the model, projects are modeled in a tree structure, with possibly more than one root. Leaf nodes are called tasks, non-leaf nodes are called modules. Figure 3.2 gives an abstract example of a possible tree (with an example precedence).

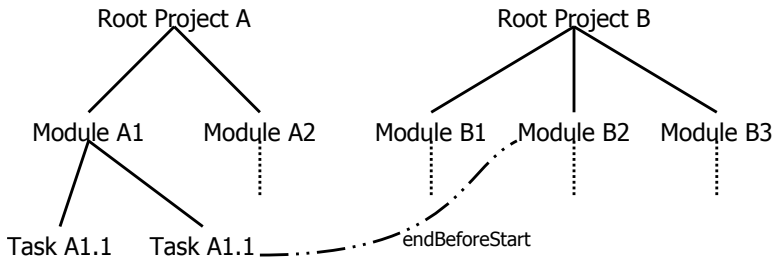

Figure 3.2: *Abstract example of a Project tree*

Modules have the boolean attribute *isAlternativeNode*. If it is `true`, the child projects of that module form an alternative construction (exactly one has to be done). This is useful for, for example, third party work: we could build the engine ourselves or we could buy one, the latter being more expensive but taking less time. Alternative constructions are also useful for executing a task with alternative equipment. For example: a task could take less time with a faster/newer piece of machinery.

Big Generic Project table vs Separate Task table

It can be argued that a separate Task table (besides a Project/Module table) is a good decision, as tasks are significantly different from modules:

- A lot of attributes only matter for tasks, such as *duration* and *minFirstPartDuration*.
- Tasks can be split into parts (preemption), modules cannot.
- Only tasks are allowed to request employees and equipment. Actually their parts require employees and equipment. Modules do not have parts.

However, tasks and modules share a lot of functionality:

- production/consumption of consumables
- Precedences
- Part of a tree (*subProjectOf*)

- *releaseDate*, *softDeadline*, *hardDeadline*, *calendarid*

With a separate Task table, there would be two 'copies' of each piece of functionality, which is considered to be a lot more troublesome than checking whether a project is a task or a module.

Optionality: forcePresence on root projects

Generally, within the scope of CPS, tasks or modules are not optional. An example: when building an airplane, attaching the left wing is not optional. Of course, this does not prove that is not desired to have the possibility of optional modules.

However, the possibility of optional projects harms the performance of the 'push down presence' in the implementation: a project is present if and only if its parent is present. So from a performance point of view, you would want to decide to not support optionality.

But in RMS optionality is required, for example when 5 sections of 100 meter of rails has to be replaced and we wonder how many sections can we do in one weekend. To solve this, noticing that it is allowed to have multiple roots in the project tree ('push down presence' does not link them), a new attribute *forcePresence* was introduced, which is only relevant to root projects. *forcePresence* can be set to `false` on a root, allowing the optimization to decide to not include the entire tree belonging to this root.

In the end, it is also possible to fake optionality by using an *isAlternativeNode*-construction between the original project and a 0-duration task.

Note that when a project is absent, it is no longer relevant in the precedence network. This can potentially harm transitive precedence relations.

3.1.1 Task Parts, duration and separation

Tasks can be split into parts, which is commonly known as "preemption" ([BVQ08], [ZLT06]). This is required because it is common that employees work in shifts (for example, available for 8 hours) and a certain task can take longer than 8 hours. Without preemption, it would not be possible to ever execute this task. Of course, the combined duration of the parts should equal the specified duration of a task. It might appear that the duration of a task is longer than specified, which is caused by gaps in time between the parts.

For performance reasons, the number of possibilities in which a task can be split into parts has to be limited. Also, in practice is it undesirable to have many small parts. To imagine the impact on performance without such restrictions, think of a task of 30 hours, which can be split into parts, with minutes as time unit. The task could be split in $30 * 60 = 1800$ parts of 1 minute, 60 parts of 30 minutes, 30 parts of 1 hour and every other possible combination. Parts are not even required to have the same duration! As such the number of possibilities explodes...

Figure 3.3 gives an example task split into 4 parts. The durations of the parts and the spacing between them (separation) can be bounded. Note that different parts can be executed by a different set of employees.

A reasonable way to limit the number of parts is by specifying the minimal duration of parts, by introducing an attribute *minPartDuration*. A decent choice would for example

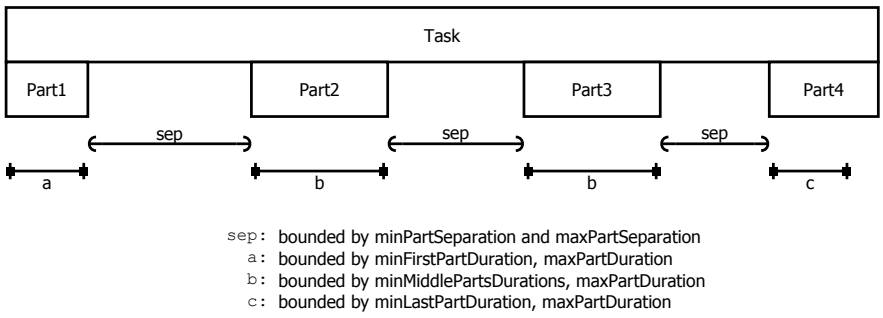

Figure 3.3: *Graphical representation of a task and its parts*

be 'half the length of a shift', lets say 4 hours. So a task of duration 13 hours would have a maximum of 3 parts (of duration 4, 4 and 5). If all tasks are defined this way, however, it could happen for example that employees have nothing to do for 3 hours, as that time is not long enough to start a part of another task.

To solve this problem, 3 minimal durations were introduced: *minFirstPartDuration*, *minMiddlePartsDurations* and *minLastPartDuration*. So the first and last part of the task are now special cases. This allows you to define that the first part of a task does not have to be of, for example, shift length (or half shift length).

Besides part durations, the distance in time (separation) between the parts can also be controlled. It is common, for example, that at the end of a shift of an employee, another employee has to immediately continue with that task ($maxPartSeparation = 0$).

To disable preemption for a task, set its minimal part durations equal to its duration.

3.1.2 Projects: *releaseDate*, *softDeadline*, *hardDeadline* and Allowed Time Windows

releaseDate is a common attribute for projects in scheduling. For example, when you want to state that a module can only be executed after next Sunday, because that is when necessary parts arrive.

Deadlines are also a common feature. For example: the contract with a customer states that the product is delivered before a set date. We distinguish between two different kinds of deadlines: *hard* and *soft*. In case of a hard deadline, the project has to end before the deadline, no exceptions. In case of a soft deadline, it is allowed to finish the project after it, but then *tardinessCost* is incurred. It expresses that it is preferred to finish a project before a set date, but it is ok to end it after, in which case a cost is incurred. Soft deadlines are commonly referred to as due dates, but in this model the term *softDeadline* was chosen to clarify its relation with *hardDeadline*.

Combined, these attributes allow you define one time window in which a project has to be executed. This is sometimes not enough, specifically in rail maintenance scheduling: maintenance usually takes place in the weekends (more than one time window). Some tasks, like preparation, can be executed before the weekend. Other tasks, like storing used equipment, can be executed after the weekend. But the tasks that actually prevent trains from using the railway have be executed in the specified time windows.

This is what the *calendarid* attribute for projects is for. The project has to be executed within the time windows specified in the calendar. Use -1 to specify that a project is not bounded by this kind of time windows.

The time window construct would make the attributes *releaseDate* and *hardDeadline* obsolete. It was decided to keep them for two reasons:

- More than one time window is not common (in our case it was introduced specifically for RMS). Creating a new calendar for every project which only needs one time window is an unnecesary hassle and using *releaseDate* and *hardDeadline* is easier in that case.

- The concepts can be combined. Many projects can share a calendar that describes which weekends are available for work. But each project can have a separate *releaseDate* and *hardDeadline*.

3.1.3 Precedences, *minDelay*, *maxDelay* and *distanceCost*

The existence of precedences is an extremely common concept within scheduling. The most common format is "Project A has to end before project B can start" (A *endBeforeStart* B). Figure 3.4 shows the four basic types of precedences.

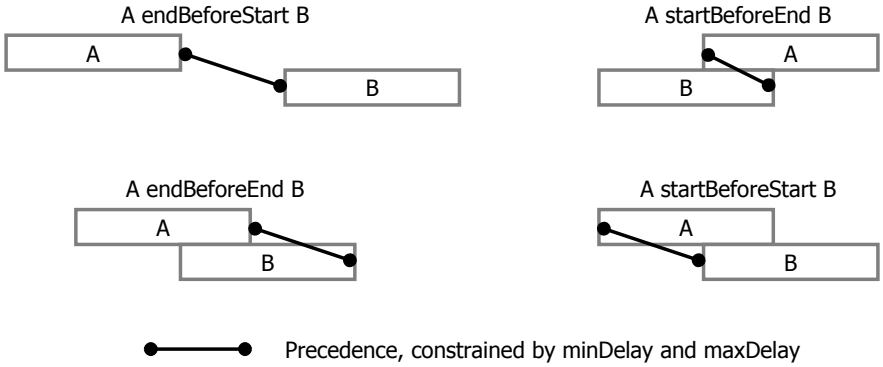

Figure 3.4: *Types of precedences and how* minDelay *and* maxDelay *would affect them.*

The input model supports more types, but those are just other ways to formulate the precedence, like described in the table below:

Precedence	**Mapping**
A *endAfterEnd* B	same as B *endBeforeEnd* A
A *startAfterStart* B	same as B *startBeforeStart* A
A *startAfterEnd* B	same as B *endBeforeStart* A
A *endAfterStart* B	same as B *startBeforeEnd* A
A *endAtEnd* B	A *endBeforeEnd* B with $maxDelay = 0$
A *startAtStart* B	A *startBeforeStart* B with $maxDelay = 0$
A *startAtEnd* B	A *startBeforeEnd* B with $maxDelay = 0$
A *endAtStart* B	A *endBeforeStart* B with $maxDelay = 0$
A *synchronize* B	A *startAtStart* B `and` A *endAtEnd* B

minDelay and *maxDelay* restrict the distance between the start/end of two projects. Whether the start or end is involved, is based on the precedence type and it can be concluded from the black dots in Figure 3.4. For example, with *A endBeforeStart B*,

minDelay = 30 and *maxDelay* = 90, *A* has to end at least 30 minutes and at most 90 minutes before *B* starts.

The model also allows you to specify a *distanceCostPerHour* between two projects. It is incurred when the distance between the relevant start/end of the two projects is bigger than 0. For example, with *A endBeforeStart B*, *distanceCostPerHour* = 300 and the decsion that *A* ends 2 hours before *B* starts, the incurred distance cost is 600.

3.1.4 Employee have Skills, Tasks request Skillsets

The concept of different skills is quite common in scheduling ([PBMN07], [HK10]). A task can only be executed by employees with the required skill. For example, when you want to specify that only a qualified mechanic can do a certain task.

In our model, skills are designed a bit differently. It is best demonstrated by an example:

> Skills: *Mechanic*, *Electrician*, *Dutch*, *English*.
> 2 employees: Anna and Bob are mechanics, but only Bob knows English.
> Task: Place engine in Car, requires 1 employee with skillset {*Mechanic*, *English*}
> In this case, only Bob can execute the task, as Anna does not know English.

To model this, give Anna and Bob the relevant skills in the table EmployeeHasSkill. Define a skillset that contains the skills "Mechanic" and "English" in the tables Skillsets and SkillPartOfSkillset. Finally, have the task "Place engine in car" request 1 employee with that skillset in the table TaskNeedsEmpWithSkillset.

This design is equivalent to a design with just skills. For example, the skill "Mechanic that knows English" would be linked to "Bob". However, the design with skillsets is easier to extend when new skills are introduced. For example, if there were 10 languages defined as skill and a 11th (Spanish) was added, you would just have to assign the skill to the relevant employees. In a design with just skills, you would have to investigate each employee carefully: each employee with the skill "Mechanic that knows English" would need to be assigned "Mechanic that knows Spanish" if they know Spanish.

Also, the Skillsets table only needs to contain sets that are actually used by tasks. For example, there could be 10 different languages defined as skills besides "Mechanic", "Electrician", et cetera. If the skillset "Dutch Electrician" is not used by any task, the set does not have to be defined. This design is especially advantageous when used properly in a GUI: a task would require a set of skills (in the example "English" and "Mechanic") and the skillset entity representing "English Mechanic" is created 'under the hood' and hidden from the user.

There is also a slight disadvantage: if a task requests a single skill, this also has to be defined as skillset (with one element). This, however, has no impact on performance on the optimization, because in preprocessing, employees are matched to the skillsets they cover with their skills.

In some situations, skill levels are desired. For example: "Mechanic" with levels 1 through 5, with a task requiring a Level 3 Mechanic. In the model, this is not directly supported, but can be done, by defining skills "Level 1 Mechanic" through "Level 5 Mechanic" explicitly.

3.1.5 Equipment and Zones

In CPS, "Equipment" is the name for discrete resources, an essential part of RCPSP ([BK12], [BLK83]). "Resource without Skills" would describe the concept well. It is called Equipment because within CPS, it is mostly used to model equipment. However, it can also model zones and even employees in a limited form, as described below.

When the problem to be solved with the CPS model does not include the concept of skills, it can suffice to model 'nameless' employees as equipment: a function in time of available employees and tasks requesting a number of them. A good example of this can be found in the proof that CPS is NP-hard (Section 2.2). However, this kind of use is limited in reality, because you most likely want to know *which employee* needs to do *what task* and *when*.

Besides pieces of equipment, the concept of equipment in CPS can also be used to model zones. For example, a task requiring 2 hammers is not that different from a task requiring 2 units of capacity in zone A. In aircraft construction, this concept is used to model for example that only 3 people can be in the cockpit at the same time. This similarity even continues when you consider equipment and zones to be modeled in a 'containment hierarchy'. See Figure 3.5. There are some important differences though, which are discussed below.

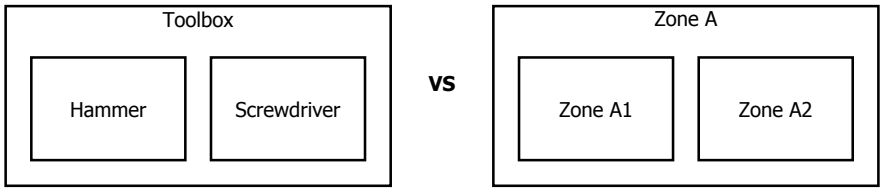

Figure 3.5: *Equipment* `vs` *Zones.*

Tasks requesting Equipment, hierarchically

Assuming the *Equipment* table contains various pieces of equipment, the table *Equipment-Hierarchy* can be used to model: if a task needs one of equipment A, it also needs X of 'Equipment B' (a 'child' piece of equipment A). For example, you can model that when an employee takes a toolbox with her, she also takes the tools in that toolbox (2 hammers and 3 screwdrivers) with her; other employees can not use those tools. Of course it is possible to instead model this explicitly for every task requesting equipment A, but imagine the inconvenience you would be confronted with if there are hundreds of those tasks. Note that in the mathematical model (Section 2.1) this is the case. It is modeled that way to keep recursion out of the mathematical model.

When you use the table Equipment to model zones, *EquipmentHierarchy* is arguably an even more useful concept. It allows you to, for example, model that if a task takes up one unit of capacity of zone $A1$, it also takes up one unit of capacity of zone A (the 'parent' zone). For example, a maximum of 3 people are allowed to be in a the cockpit, but 2 on the left and right side individually; occupying space on either side counts towards the maximum of 3 for the cockpit.

Tasks disabling Equipment, hierarchically

The concept of 'tasks disabling equipment' originates from aircraft construction, where zones are modeled as equipment. For example, because of safety regulations, it is required that when a task takes place on top of the right wing (puts one person there), no tasks can be executed *under* that wing. With *On Right Wing* and *Under Right Wing* being zones, and the relevant task t, this would be modeled as: "Task t needs one of *On Right Wing*. Task t disables *Under Right Wing*".

Now, lets say *Under Right Wing* consists of two sub-zones: *UR1* and *UR2*. So the model contains: "*UR1* 'also use 1 of' *Under Right Wing*" and "*UR2* 'also use 1 of' *Under Right Wing*". When *Under Right Wing* gets disabled, *UR1* and *UR2* need to be disabled too. The model takes care of that, assuming *disableDirection* is set to 'right to left' (more on that later).

The concept of TaskDisablesEquipment works for 'normal' equipment too. Imagine a maintenance task that prevents the use of pieces of equipment by other tasks. Note that maintenance on pieces of equipment might also be modeled with TaskNeedsEquipment instead of TaskDisablesEquipment. A very important difference, however, is that "Task t_1 needing all (the specified max) equipment for maintenance" (and same for task t_2) would mean that task t_1 and t_2 can not be executed at the same time. If expressed with disable-constructs, they can be executed in parallel. Also note that "(the specified max)" is not necessarily a constant (because of EquipmentExtraAvailable), another reason to use a disable-construct if you want to claim all pieces of an equipment entity.

Note that the direction of the hierarchy is very important when considering disabling equipment hierarchically:

normal Equipment "Toolbox 'also use X of' Hammer". When a task disables the toolbox and also disables the hammers, the '*disableDirection*' is **left to right** (same direction as 'reading the hierarchy').

Zones "Zone A1 'also use 1 of' Zone A". When a task disables zone A and also disables zone A1, the '*disableDirection*' is **right to left** (opposite direction as 'reading the hierarchy').

Note there are two more *disableDirections*: 'none' and 'both'. Use 'none' when disabling entity A does not affect entity B nor the other way around. For example, you might want to model that cars are disabled for maintenance, but the toolboxes that are in those cars are still usable. Use 'both' when you want to model that disabling works both ways: disabling zone A disables zone A1 too and disabling zone A1 disables zone A too.

3.1.6 Producing and consumption of Consumables

Consumption of 'non-renewables', a common term in scheduling ([DRH99], [JMR+01]), is also possible in the CPS model. The term "non-renewable" originates from the opposite of "renewable" resources, which are available again after a task is completed (they are not consumed). "Consumable" is a more practical word than "non-renewable". A consumable can be fuel, screws, water, et cetera. With ProjectProducesConsumable, you can model for example that certain tasks consume fuel; other tasks can replenish fuel. It is named "ProjectProducesConsumable" and not "TaskProducesConsumable" because the producing interval does not have to be a task, it can be a module too.

The *atStart* boolean

CPO does not allow a linear change in consumables over time, so we are limited to production/consumption at either the *start* or *end* of a task or module. Usually, a consumable is produced at the *end* and consumed at the *start*. But for example, if you want to model emissions (which are not allowed to reach a toxic level), it makes more sense to produce at the start of a task/module. This is probably more strict than necessary, but at least you are sure it is safe.

3.1.7 Transition Matrices

The CPS model supports the concept of transition time and cost (also called setup time and cost). It expresses that it takes time (and has a cost) to transition from one state to another. In the case of CPS, it can be used to model for example the *location* certain tasks have to be executed in. Define the different locations as task types and assign them to the tasks. Then define a matrix of 'travel times' between those locations. This is of course especially useful when travel times are significant.

There are more uses for transitions, although less likely to occur within the scope of CPS. By modeling an oven as an employee, and different temperatures as task types, you can model that it takes time for the oven to heat up and cool down.

Note that this concept makes it possible to model the Traveling Salesman Problem within CPS.

In the CPS model, there is no cost incurred for an employee 'doing nothing' (not scheduled to participate in a task part). Even when transitioning (non-zero transition time between two task parts), the employee is 'doing nothing'. The *defaultCostPerHour* (or overridden *costPerHour* in the EmployeeHasSkill table), is only applied when the employee is participating in a task part, and is therefore not included in the calculation of transition cost.

3.2 OPL files

Before the OPL model is described (per file), the way the files are structured is described.

3.2.0 OPL files structure

The main OPL model includes various smaller files (12 at the moment). Having just one (or two) file (typical for Constraint Programming) is just not convenient with a project of this size.

Including them as follows:

```
include "00_read_inputdata.mod";
include "01_assertions_inputdata.mod";
include "02_expand_inputdata.mod";
...
```

does not work too well, because when working in file `01`, COS does not understand that file `00` is also part of the project and will raise unnecessary errors. The model will run properly, but during development it is inconvenient.

So, the files are 'chained' as follows:

```
(in the main file)                      include "11_output_variables_to_console.mod";
(in 11_output_variables_to_console.mod) include "10_write_Gantt_XML.mod";
(in 10_write_Gantt_XML.mod)             include "09_constraints.mod";
(in 09_constraints.mod)                 include "08_objective_function.mod";
...
```

3.2.1 OPL Model files

00_read_inputdata

- Constants (like *TIME* and precedence types).
- Tuple-types for each table.
- Reads the input data.

01_assertions_inputdata

- Assertions on the input-data.
- Not all assertions are in this file, some assertions depend on operations later on. Because of the naming of the assertions (A1_ means the assertion is in file `01`) you can tell what file to look in if an assertion fails.

02_expand_inputdata

- Contains preprocessing on the input data, like:
- Splits the precedences into four sets which cover all possibilities: endBeforeEnd, startBeforeStart, startBeforeEnd and endBeforeStart. All other precedence types are a special case of these.

- Determines for each employee which skillsets she covers, based on her skills. The *costPerHour* of the skillset is the maximum of the *costPerHour* values for each skill included in the set. Or sum if the parameter *costs.skillset.useSumInsteadOfMax* is set.
- Determines project relationships, like parent and children.
- Determines equipment relationships, like parent and children.
- Splits tasks (leaf projects) from modules (non-leaf projects).
- Splits production/consumption of consumables
- Converts the input Date-elements to integers (based on minutes since the earliest date in the input data).
- Converts the calendar availability ranges to integers.

03_assertions_circularreferences

- Using `execute` blocks, determines if there are circular references in either the project hierarchy or the equipment hierarchy.
- Halts the model (using assertions) if circular references were found.

04_projectpreemption

- Responsible for allowing the preemption of tasks, by setting constraints on how many parts each task can have.
- Now that the tasks have been split from the modules (in file `02`), there are quite a few assertions listed that only hold for tasks.
- Calculating the maximum number of parts each task can consist of, by using the *minFirstPartDuration*, *minLastPartDuration* and *minMiddlePartsDurations* data.
- Constructs tuple set linking Tasks × Skillsets.
- Constructs tuple set linking Tasks × Parts × Skillsets × Employees.

05_treeoperations_on_equipment

- This file mostly deals with recursively generating true *TaskNeedsEquipment* requirements from the input *TaskNeedsEquipment* and *EquipmentHierarchy*. For example, "Task T needs 2 of equipment A" and "Equipment A 'also use 2 of' equipment B", results in "Task T needs 2 of equipment A and 4 of equipment B".
- It builds the required elements for *stepFunctions* for equipment availability.
- Similar to the recursive generation of *TaskNeedsEquipment*, it recursively generates true *TaskDisablesEquipment*.

06_decision_variables_and_expressions

- `stepFunctions` for calendars.
- `cumulFunctions` for equipment and consumables.
- `interval` variables for projects and parts and employee participation.
- `sequence` variables for employees.
- Decision expression dealing with makespan.
- Decision expressions dealing with calculation of total costs.

07_configure_search

- File containing `execute` block to set optimization parameters.
- At the moment it only sets the time limit of the optimization.

08_objective_function

- Contains the various possible objective functions.
- Currently there are three possibilities: minimize nothing, minimize makespan and minimize costs (which is flexible because of the configurable weights).

09_constraints

- Contains the actual constraints.
- Contains markers like `@mathC1.1`, which means the constraint is the implementation of the mathematical constraint in Section 2.1.5.

10_write_Gantt_XML

- Writes an XML-file representing a Gantt-chart of the solution.
- The file can be opened with the Java-program in the `output` folder.
- It converts integer time representations to proper dates.

11_output_variables_to_console

- Writes the most important output variables (which are not in the Gantt-chart) to console.
- This is very useful for solving instances in bulk (using batch-files and `oplrun`) because the relevant information is lost (can not be seen in the COS IDE).

3.3 Output

As output, the OPL model prints some values to console and generates an XML-file which represents a Gantt-chart. Both formats are explained here.

3.3.0 Console Output

The text below is an example of the console output printed by the OPL model.

```
1  makespan: 50
2
3     0 * 0      (MakespanCost)
4   + 1 * 80     (TotalProjectFixedCost)
5   + 1 * 20     (TotalProjectUnperformedCost)
6   + 1 * 0      (TotalProjectTardinessCost)
7   + 1 * 7.6    (TotalProjectEmployeeCost)
8   + 1 * 136    (TotalProjectEquipmentCost)
9   + 1 * 2      (TotalDistanceCost)
10  + 1 * 18     (TotalTransitionCost)
11  = 263.6      (Weighted Total)
```

3.3.1 Gantt Chart

The OPL model generates an XML-file. The structure of this XML-file is a common format which can be opened with Gantt-chart viewers used by IBM ILOG. A simple viewer is `sv.jar`, included with the OPL project. Examples of the Gantt-chart visualization can be found in Figure 3.6 (Project View) and Figure 3.7 (Resource View).

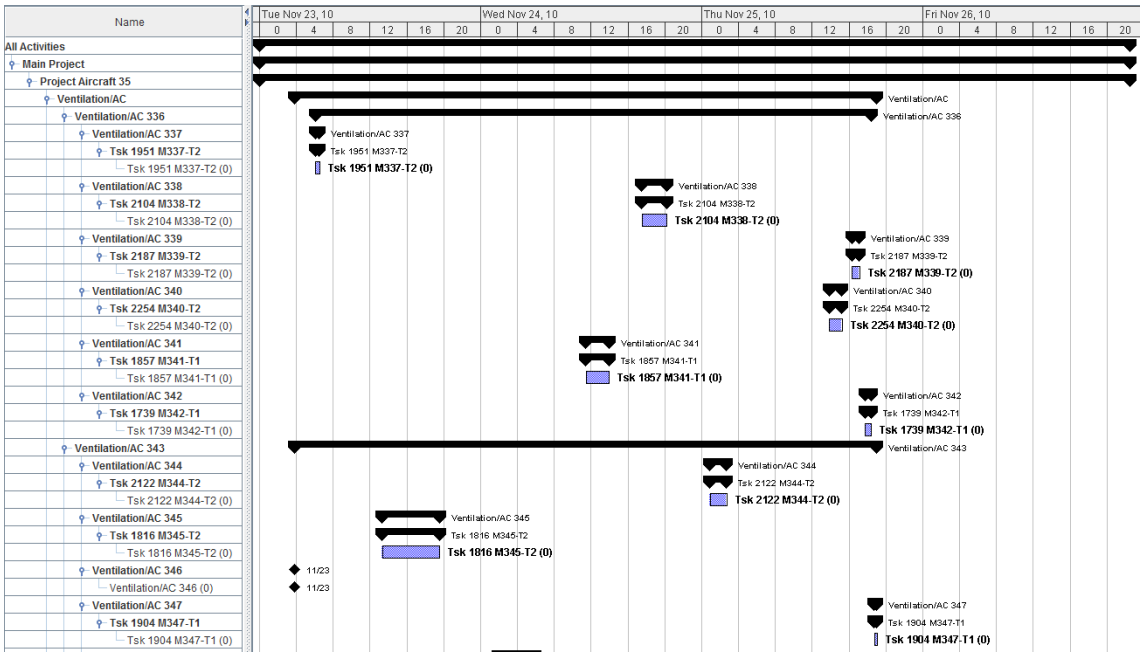

Figure 3.6: *Example Gantt Project View.*

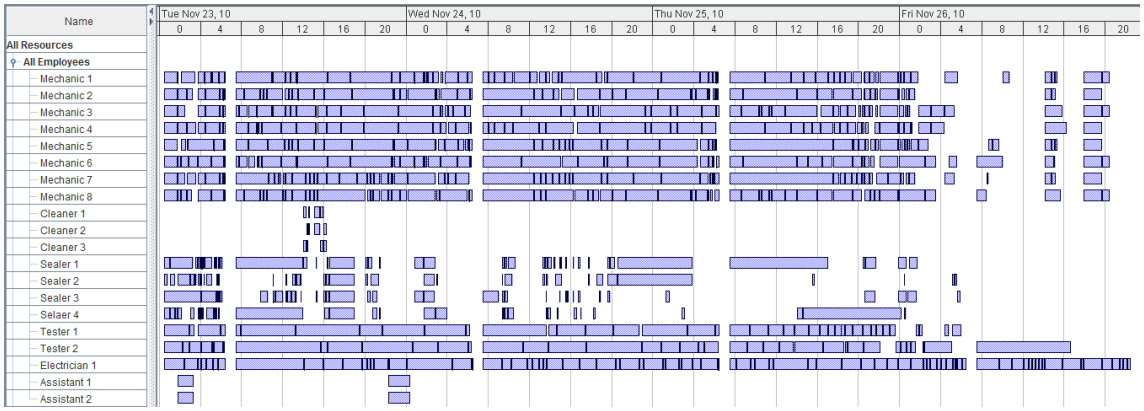

Figure 3.7: *Example Gantt Resources View.*

The resource view also contains a line for every equipment entity and consumable in the model. However, the XML format and the viewer were not designed to support entities being used by more than one task at the same time. The view can therefore tell you when equipment and consumables are used by a task, but it does not tell you how many of them.

Chapter 4

Integration

In this chapter we will discuss the integration of the OPL model in various contexts.

4.0 Rail Maintenance Scheduling

Confidential part of thesis.

4.1 ODME

ODME can be used to construct a GUI for an optimization model. A lot of the elements are generated automatically. This section describes the process of creation of an ODME Application for CPS.

4.1.0 Data Model

First, the *Data Model* has to be defined. This can be done from scratch, which would be typical when working with ODME-integration from the start of an optimization project. ODME also offers features to reverse engineer a data model from a relational database or OPL model. As the OPL model for CPS was already under construction, that last feature was used to reverse engineer its data model for use in ODME.

Using this feature, the foreign key relations are not included in the resulting data model. They can be added, of course, but in the CPS case, it was decided not to because it caused some issues. For example, values of *calendarid* in *Employees* can be -1, meaning "Employee is always available". But -1 is not in *Calendars* and is therefore not a valid foreign key. `null` could be replaced as having this semantic, but in OPL, a `null` value for an integer attribute is represented as 0, which could be confused for a valid *calendarid*. Because similar semantics for -1 are used throughout the model and foreign key relations are checked in the assertions in the OPL model anyway, it was decided to not redesign the model.

After the tables have been imported, some manual work has to be performed to create a visual representation of the data model. Because of the lack of proper foreign keys, 'plain arrows' were used to indicate the relations between the tables. This resulted in the model in Figure 3.1 (on page *26*).

4.1.1 Mapping

Next, views on the data have to be defined and initial data for each table has to be specified. For each table, a simple table view was generated. The initial data for each table is empty, except for the Parameters table, which contains the supported parameter keys and acceptable default.

The relational data model now has to be mapped to OPL input constructs (see Figure 4.1). ODME can then generate an OPL `mod` file containing the necessary data structures. However, in the CPS case, the generated `mod` file is simply ignored and `00_read_inputdata.mod` is used instead, because it uses `with ... in ...` OPL constructs to restrict the domains of input variables.

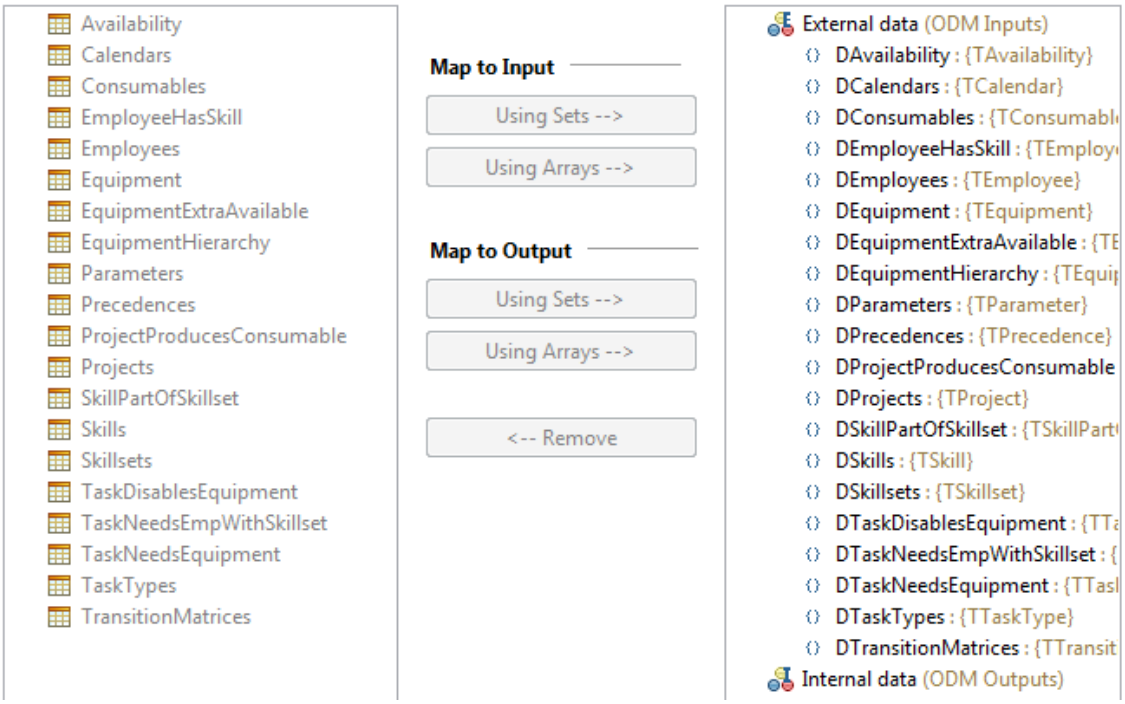

Figure 4.1: *Mapping of relational data model to OPL elements.*

4.1.2 ODME Application

It is now possible to generate the ODME Application shown in Figure 4.2. This GUI can still be improved a lot. For example, when entering the input data, you would want to hide the integer keys from the user where possible. However, constructing a user interface was not one of the goals of this project. Also, it is not that useful to have a generic user interface. Even though the CPS model can be used for a lot of different problems, the user interface will probably be case- or company-specific.

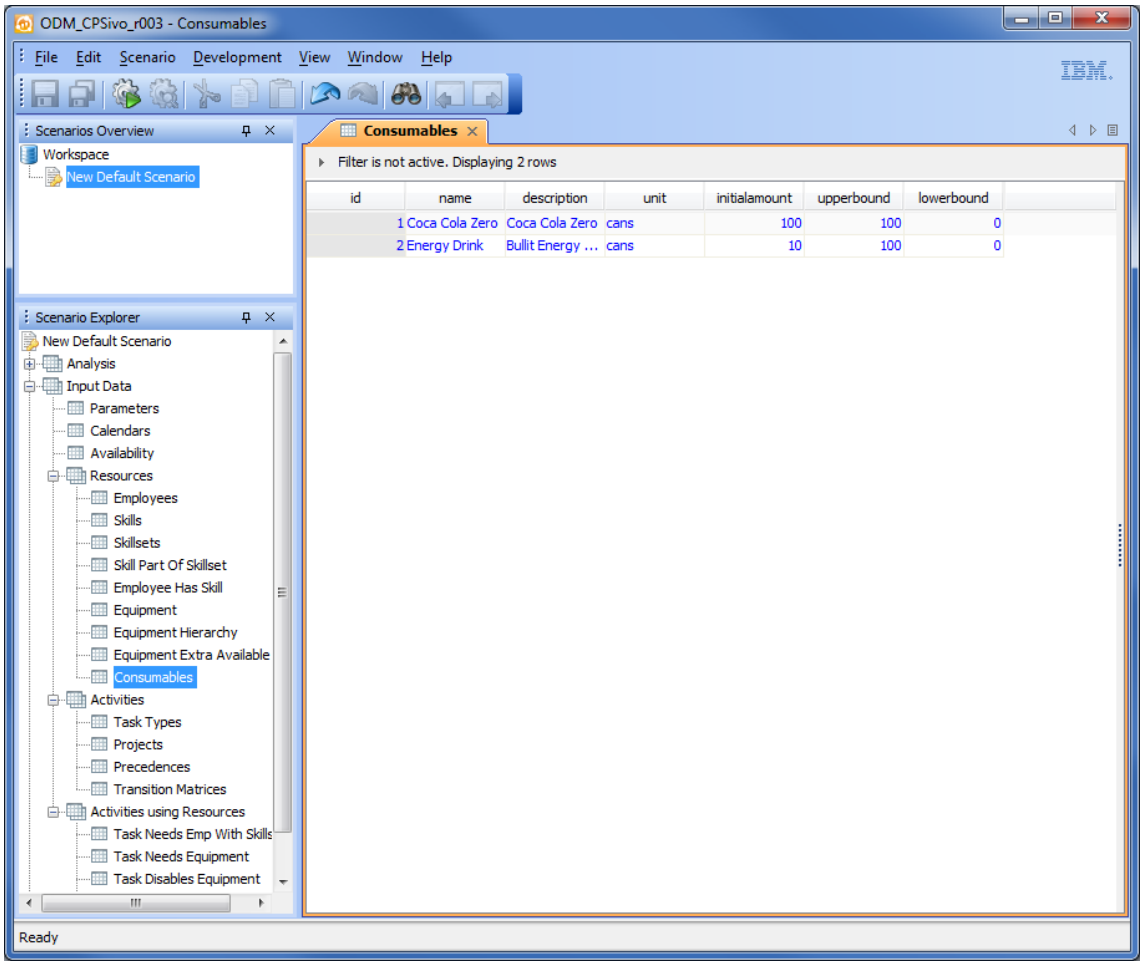

Figure 4.2: *Generated ODME Application for CPS.*

Chapter 5

Performance Analysis

In this chapter we will evaluate the performance of the CPS model on various datasets. For each dataset, we describe the process of acquiring the dataset, transforming it to the CPS input format, analysis of the output and comparison to benchmarks.

Philippe Laborie has been so kind to supply a library of 12 RCPS-related datasets originating in literature. The number of instances in each dataset varies greatly: one dataset has only one instance, another one has over 11 thousand. 8 out of 12 datasets would fit in the CPS model, indicating the worth of CPS being generic. Some datasets do not fit, for example because the dataset requires earliness costs or a different objective function. We analyze two that fit: "MultiSkills RCPS" ([PBMN07]) and "MultiMode RCPS" ([SD98]).

5.0 Common Elements

This section discusses some common elements in the processes of testing the various datasets.

5.0.0 Input data conversion

During the project a custom tool was developed to convert XML-data to an Excel data format. It was extended to support the conversion to data formats that CPS can take as input.

By reusing this code, several small tools were developed to parse various input formats so they could be converted to the CPS input format. Batch-scripts were designed to execute the conversion on multiple input files in sequence.

5.0.1 CPO from command-line (`oplrun`)

COS does not support queuing of several optimization jobs. However, it is possible to call `oplrun` from the command-line. We pass the path to the CPS model, the input data and the name of the output Gantt-file as arguments. Batch-scripts were designed to call `oplrun` for each test instance. The console output of each instance was saved to a separate file for each instance.

5.0.2 Aggregating CPO Output

The text below is an example of console output that is generated by CPO. In COS, it can be viewed in the Engine Log. A star (*) at the start of the line indicates a new optimal solution is found.

```
1  ! ----------------------------------------------------------------------
2  ! Minimization problem - 119 variables, 514 constraints
3  ! Preprocessing : 68 extractables eliminated
4  ! TimeLimit            = 18
5  ! Initial process time : 0,01s (0,00s extraction + 0,01s propagation)
6  !  . Log search space  : 549,9 (before), 515,6 (after)
7  !  . Memory usage      : 1,6 MB (before), 2,0 MB (after)
8  !  . Variables fixed   : 8
9  ! Using parallel search with 4 workers.
10 ! ----------------------------------------------------------------------
11 !          Best Branches  Non-fixed    W       Branch decision
12                   1.000           3    1     on ITaskParts({12001,0})
13                   1.000          22    3     on IProjects(12001)
14                   1.006           4    2     on IProjects(6003)
15 *               38      581 0,42s          3          -
16                 38    1.001           3    4     on IProjects(10001)
17 *               29    1.423 0,48s          1          -
18 ! ----------------------------------------------------------------------
19 ! Search terminated normally, 2 solutions found.
20 ! Best objective         : 29 (optimal - effective tol. is 0,0029)
21 ! Number of branches     : 5.984
22 ! Number of fails        : 2.601
23 ! Total memory usage     : 14,8 MB (11,6 MB CP Optimizer + 3,2 MB Concert)
24 ! Time spent in solve    : 0,49s (0,49s engine + 0,00s extraction)
25 ! Search speed (br. / s) : 11.987,1
26 ! ----------------------------------------------------------------------
```

Some datasets discussed below resulted in hundreds or thousands of such output files. In order to get an overview of the data, a custom tool was developed (in Java) and we named it "OPL2Overview". It takes all console output files in a folder and generates a CSV-file which contains one row for each instance. Before viewing the CSV-file in Excel, depending on the locale in which the optimization was performed and the locale of the Excel installation, decimal points might have to replaced by decimal commas. Description of the columns (together with the value in the example output):

`variables`	the number of decision variables	119
`constraints`	the number of constraints	514
`firstResult`	the objective value of the first solution that was found	38
`firstBranch`	the branch at which the first solution was found	581
`firstAt`	time at which the first solution was found	0,42
`lastResult`	the objective value of the last solution that was found	29
`lastBranch`	the branch at which the last solution was found	1423
`lastAt`	time at which the last solution was found	0,48
`endType`	the way the optimization was terminated, either "normally" (CPO returns a solution and proves optimality), "by limit", "infeasible", or "no solution within limit"	normally
`nSolutions`	the number of solutions that was found	2
`nBranches`	the number of branches that were searched	5984
`time`	'time spent in solve' (slightly larger than either the time at which the optimal solution was found or the time limit)	0,49

5.0.3 Test machine

The machine used to run the tests has the following characteristics:

- AMD Phenom II X4 965 Processor, quad-core, 3.40 GHz
- 8 GB of DDR3 memory
- Windows 7 Professional 64-bit Service Pack 1
- IBM ILOG CPLEX Optimization Studio 12.4 64-bits
- Java 6, Update 31.

5.1 MultiSkills RCPS

In RCPS, employees are modeled as discrete resources with a capacity function over time. For example, the resource "Mechanics" with a constant capacity function of 12 would model that there are 12 mechanics. Tasks can request a number of employees (possibly from different resources). From the perspective of an employee it has exactly one skill. In MultiSkills RCPS, employees can have more than one skill.

The data originates from [PBMN07].

5.1.0 Input Analysis

Figure 5.1 shows an example input file of a MultiSkills RCPS instance.

By taking a closer look at the input format, we can determine which elements of CPS we need (and we can conclude that the conversion is possible):

- *Skills:* The number of different skills is specified, but they do not have a name associated with them. For CPS, generate the required number of Skills, Skillsets and link the relevant entries using SkillPartOfSkillset.

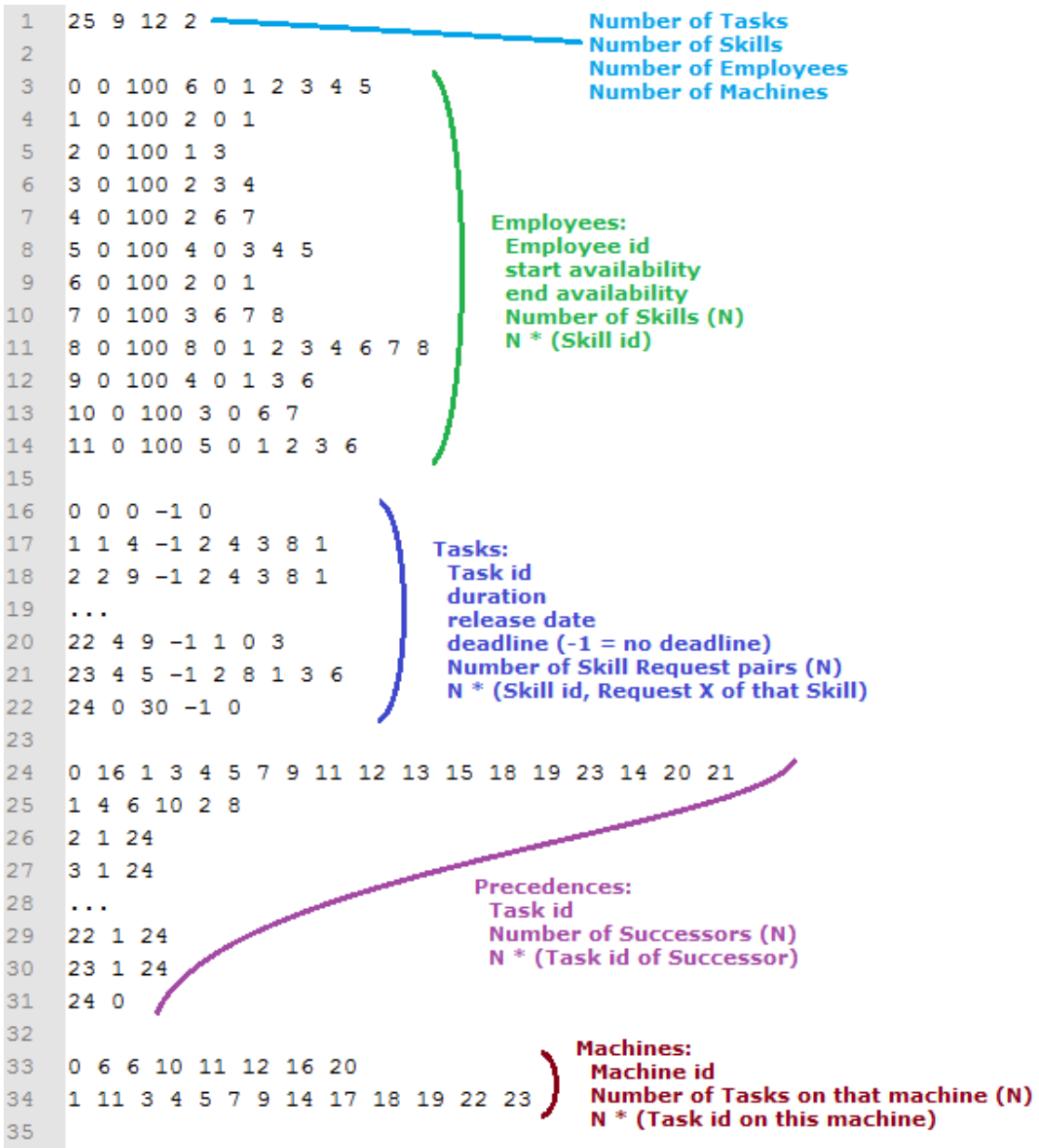

Figure 5.1: *Example input file of MultiSkillsRCPS.*

- *Employee:* Even though 'start availability' and 'end availability' are specified (always 0 and 100 respectively), they are not used in the model. It suffices to create Employees and their relevant skills in EmployeeHasSkill.
- *Tasks:* There is no project hierarchy in this dataset. There are no tasks in any of the instances that have a deadline. In CPS, create tasks in the Projects table, specify their release date and disable preemption. Use TaskNeedsEmpWithSkill to have the tasks request employees. One default TaskType is required.
- *Precedences:* In this dataset, precedences are defined as lists of successors. In CPS, create a new "endBeforeStart"-precedence for each successor of each task.
- *Machines:* Model the machines as Equipment with capacity 1. Define the relevant TaskNeedsEquipment entries.

The follwing CPS tables are empty: Calendars, Availability, TransitionMatrices, EquipmentHierarchy, EquipmentExtraAvailable, TaskDisablesEquipment, Consumables and ProjectProducesConsumables.

Also, a time limit for each instance was determined. The time limit should be large enough to give CPO an opportunity to find a solution and preferably it should be based on the expected complexity. After some experimentation, $NumberOfSkills * NumberOfTasks$ (in seconds) was chosen as time limit. The dataset is split into 5 subsets: instances with 25, 35, 40, 45 and 50 tasks. For this series of tests, only the last subset (160 instances) was used.

5.1.1 Output Analysis

The result of processing the 160 instances is an Excel sheet with 160 rows, which is too big to be displayed here. A short overview is given below:

CPS Output Summary MultiSkillsRCPS		
#instances	160	
#no solution within time limit	3	1.9%
#normal termination (optimal)	2	1.3%
#terminated by time limit	155	96.9%

As noted before, the overview contains data on the first solution and on the last solution. On average, the solution improves 9.7% between the first and last solution found, with a standard deviation of 6.8% and maximum of 29.9%. This indicates that in general the first solution found is already a very decent one.

To determine the performance of the model, we would like to make a comparison with benchmarks (known optimal solutions). However, there are no benchmarks publicly available for this dataset. Philippe Laborie made a dataset-specific CPO model. Using the same time limits, we ran that model (from now on referred to as the domain-specific model (DSM)) on all the instances, resulting in the overview below:

DSM Output Summary MultiSkillsRCPS		
#instances	160	
#no solution within time limit	0	0.0%
#normal termination (optimal)	7	4.4%
#terminated by time limit	153	95.5%

The instances that result in a proven optimal solution in CPS also result in (the same) proven optimal solution in DSM. The other way around, when DSM proves a solution is optimal but CPS does not, CPS finds that solution very quickly (within 1 second) but fails to prove optimality (within the time limit).

Out of the 160 instances, CPS performs better in 22 cases (13.8%), worse in 114 cases[1] (71.3%) and equal in 24 cases (15%). The table below gives an overview of the performance of CPS compared to DSM.

Instances	**CPS performs**	σ	`max`
all instances	1.9% worse	2.5%	10.2% worse
instances CPS does better	1.5% better	1.1%	4.8% better
instances CPS does worse	3.0% worse	2.1%	10.2% worse

A graphical representation of the comparison can be found in Figure 5.2. The X-axis has instances ordered by 'difficulty' (#tasks * #employees * #skills. The Y-axis is the percentage CPS performs worse than the domain-specific model (so negative values represent it does better). Because of the ordering of the instances on the X-axis and the lack of a pattern in the graph, it can be concluded that the performance of CPS compared to DSM is not related to the difficulty of the instance.

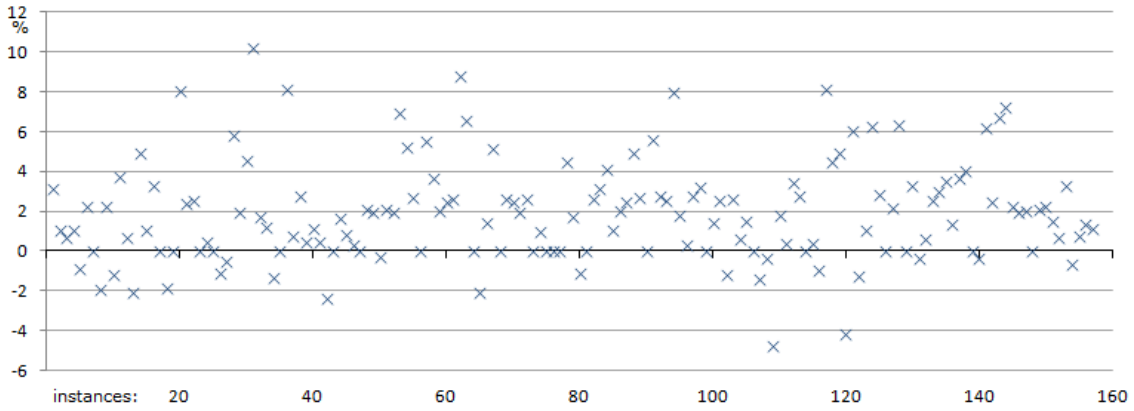

Figure 5.2: *Comparison of MultiSkillsRCPS results between CPS and DSM.*

It was determined that the generic CPS model performs 1.9% worse on average, compared to a domain-specific model, on the Multi Skills RCPS dataset. This is considered an acceptable loss of performance.

[1] In the 3 instances where CPS does not find a solution within the time limit, but DSM does, CPS is considered to perform 'worse', but the 3 entries are not included in the calculation of the averages and the graph.

5.2 Multi Mode RCPS

In RCPS, tasks can only be executed in one way. MultiMode RCPS introduces the concept of alternatives: multiple ways to execute a task, making different resource requests and/or having a different duration. In practice, this allows you to for example model that certain employees can do a task faster. Another example is having a third party alternative for a task.

The data used for this test case originates from [SD98]. How the instances were generated is described in the second part of the paper, titled "Computation". This dataset has also been used in [Har01] as a test set, among others, which strengthens the reliability of the dataset.

5.2.0 Input Analysis

The listing below is an example of the input format of a MultiMode RCPS instance.

```
1  ******************************************************************************
2  file with basedata            : mm2_.bas
3  initial value random generator: 1424959589
4  ******************************************************************************
5  projects                      :  1
6  jobs (incl. supersource/sink ):  12
7  horizon                       :  86
8  RESOURCES
9    - renewable                 :  2   R
10   - nonrenewable              :  2   N
11   - doubly constrained        :  0   D
12 ******************************************************************************
13 PROJECT INFORMATION:
14 pronr.  #jobs rel.date duedate tardcost  MPM-Time
15     1     10      0       13        3       13
16 ******************************************************************************
17 PRECEDENCE RELATIONS:
18 jobnr.    #modes  #successors   successors
19    1        1          3           2   3   4
20    2        3          2           5   6
21 ...
22   11        3          1          12
23   12        1          0
24 ******************************************************************************
25 REQUESTS/DURATIONS:
26 jobnr. mode duration  R 1  R 2  N 1  N 2
27 ------------------------------------------------------------------------------
28   1      1      0       0    0    0    0
29   2      1      3       6    0    9    0
30          2      9       5    0    0    8
31          3     10       0    6    0    6
32 ...
33  11      1      6       0    2    0   10
34          2      9       0    1    0    9
35          3     10       0    1    0    7
36  12      1      0       0    0    0    0
37 ******************************************************************************
38 RESOURCEAVAILABILITIES:
39   R 1  R 2  N 1  N 2
40     9    4   29   40
41 ******************************************************************************
```

After taking a closer look, we can conclude that we need the following CPS constructs:

- *Renewable:* Map the renewable resources to Equipment. The availability stated in the RESOURCEAVAILABILITIES section is the *defaultAmount*.
- *Non-Renewable:* Map the non-renewable resources to Consumables. The availability stated in the RESOURCEAVAILABILITIES section is the *initialAmount* (and *upperbound*). Set 0 as *lowerbound*.
- *Jobs:* For each job in the input data, create an entry in the Projects table in CPS. These entries will be modules (because they will have child-nodes). Set *isAlternativeNode* to `true`.
- *Precedences:* For each job, the successors are listed. In CPS, create a new "endBeforeStart"-precedence for each successor for each task.
- *Modes:* For each mode of each job, create a task in the Projects table in CPS. Set *subProjectOf* to the relevant module that was created to represent the job. Set its *duration* and use ProjectProducesConsumable and TaskNeedsEquipment to request non-renewables and renewables respectively. One default TaskType is required.

As we are focussing on makespan, the `rel.date`, `duedate` and `tardcost` are ignored. `MPM-Time` is also not relevant and doubly constrained resources do not occur in any instance. The tests were not executed with minimal tardiness cost as objective function as there were no benchmarks available to compare the results to.

The following CPS tables are empty: Calendars, Availability, TransitionMatrices, Skills, Skillsets, SkillPartOfSkillset, Employees, EquipmentHierarchy, EquipmentExtraAvailable, TaskDisablesEquipment.

The dataset is split into multiple subsets. The basis is having 2 renewables, 2 non-renewables, 16 jobs and 3 modes per job. *c15* and *c21* are examples of this. The j-series then varies the number of $j$obs. The m-series varies the number of $m$odes. The n-series varies the number of $n$onrenewables. The r-series varies the number $r$enewables. The result can be found in 5.1

Originally in the research for [SD98] 640 instances of each subset were generated. However, not all generated instances were feasible. The infeasible instances were excluded from the dataset. For *j30*, the hardest set, it was not always known if an instance was feasible or not, so all instances are included.

All subsets except *j30* are considered to be 'easy'. Because of the large number of test instances and a lack of testing time, each non-*j30* instance was assigned a time limit of $(\#Jobs + 2$ seconds. The +2 is caused by the presence of a 'supersource' job (0-duration, takes place before all other jobs) and 'sink' job (0-duration, takes place after all other jobs). The *j30* subset is considered to be 'hard' and each instance was given a $10 * (\#Jobs + 2) = 320$ seconds time limit.

5.2.1 Output

Non-*j30* subsets

As mentioned before, the non-*j30* instances are considered to be easy. Therefore, we will not discuss them in detail. The optimal solutions are known as well, so we can compare the CPS results with those. A short overview:

Set	#Renewables	#NonRenewables	#Jobs	#Modes/Job	#instances
c15	2	2	16	3	551
c21	2	2	16	3	552
j10	2	2	10	3	536
j12	2	2	12	3	547
j14	2	2	14	3	551
j16	2	2	16	3	550
j18	2	2	18	3	552
j20	2	2	20	3	554
j30	2	2	30	3	640
m1	2	2	16	1	640
m2	2	2	16	2	481
m4	2	2	16	4	555
m5	2	2	16	5	558
n0	2	0	16	3	470
n1	2	1	16	3	637
n3	2	3	16	3	600
r1	1	2	16	3	553
r3	3	2	16	3	557
r4	4	2	16	3	552
r5	5	2	16	3	546
					11182

Table 5.1: Overview of the properties of the subdatasets.

CPS Output Summary MultiMode RCPS, non-*j30*		
#instances	10542	
#no solution within time limit	15	0.1%
#normal termination (optimal)	6988	66.3%
#terminated by time limit	3539	33.6%

Of the 3539 instances terminated by the time limit, in 2748 cases (77.5% of 3539, 26.1% of 10542) CPS does find the optimal solution, but does not prove optimality (and thus continues searching).

So in 791 cases (7.5% of 10542), the optimal solution is not found. In those cases, the solution that is found by CPS is 5.1% worse on average with a maximum of 32.14%. There are only 5 cases in which the solution that is found is worse than 20%. The solutions to this instances would most likely improve with a larger time limit. Keep in mind the time limit was kept low because of the huge number of instances. These 5 instances were not investigated further, because focus shifted to the *j30* subsets.

On average over all instances, CPS performs 0.4% worse than optimal.

j30 subset

Out of the 640 *j30* instances, there are 88 instances that are not listed in the 'best results so far' list for MultiMode RCPS in Philippe Laborie's library. For all those 88 instances, CPS can also not find a solution within 320 seconds. It is therefore assumed that these instances are infeasible and they are excluded from the analysis below. In the non-*j30*

instances, the 'known best' are optimal solutions. For the *j30* instances, the 'known best' are not necessarily optimal.

CPS Output Summary MultiMode RCPS, *j30*		
#instances	552	
#no solution within time limit	1	0.2%
#normal termination (optimal)	332	60.1%
#terminated by time limit	219	39.7%

Out of the 552 remaining instances, CPS performs better in 2 cases (0.4%), worse in 98 cases[2] (17.8%) and equal in 452 cases (81.9%). The table below gives an overview of the performance of CPS compared to the known best solutions. A graphical overview (like for MultiSkills RCPS) was constructed, but it is not suitable here because it is unreadable caused by the large number of instances.

Instances	**CPS performs**	σ	`max`
all instances	0.6% worse	1.6%	11.9% worse
instances CPS does better	2.2% better	0.1%	2.3% better
instances CPS does worse	3.7% worse	1.9%	11.9% worse

It was determined that the generic CPS model performs 0.6% worse on average, compared to the best known solutions, on the Multi Mode RCPS dataset. This is considered to be a very acceptable result.

5.3 Rail Maintenance Scheduling

Confidential part of thesis.

5.4 Aircraft Assembly

Confidential part of thesis.

[2]In the 1 instance where CPS does not find a solution within the time limit, but a solution is known, CPS is considered to perform 'worse', but the entry is not included in the calculation of the averages.

Chapter 6

Conclusion

This final chapter will conclude the thesis. First, the developed CPS model will be evaluated. After that some notes regarding possible future work will be made.

6.0 Evaluation

As Chapter 2 stated, the goal of this master project is to develop a generic optimization model for the CPSP, using CPO.

To evaluate the developed CPS model, one of the things to be measured is how *generic* it is. Because it is generic, a limited loss in performance is acceptable. *Performance* is a relevant keyword in that sentence. Another important keyword is *realistic*: CPS extends RCPS so it can be used in practice. *Generic*, *realistic*, and *performance* are the main topics for the evaluation of the developed CPS model.

Generic As stated in Chapter 5, the CPS model supports 8 out of 12 datasets that were found in RCPS-related literature, and covers Aircraft Assembly and Rail Maintenance Scheduling. Also, it covers the requirements specified in Section 2.0.0. Thus, the CPS model is considered to be sufficiently generic. Of course, it could be even more generic, but in its current state it can cover many cases of 'constructing a big object' and more: maintainance projects like rail maintenance scheduling.

Realistic In most academic research concerning scheduling, makespan is the relevant objective function. In practice, however, finishing a project as fast as possible is not the only factor; other objectives can be important as well, like minimizing costs. CPS therefore supports the minimization of costs: a weighted sum of task-specific, employee-specific, equipment-specific costs, et cetera (for a full overview, see Section 2.1.4). The configurable weights make this objective function very flexible and thus powerful.

Performance The performance of the CPS model was analyzed with various use cases in Chapter 5. Based on the results of the MultiSkills RCPS and MultiMode RCPS datasets, we can conclude that the performance of CPS is very acceptable for theoretical cases not involving preemption.
The dataset of Rail Maintenance Scheduling is based on real-life data. *Confidential part of thesis.*

In the Aircraft Assembly case, we experimented with an objective function based on costs instead of makespan. *Confidential part of thesis.*

The CPS model is also well documented. The requirements in Section 2.0.0, the formal definition in Section 2.1, the description of the design in Section 3.0, and the explanation of the design decisions in Section 3.1, together with the exhaustive commments in the OPL code document the CPS model clearly. This detailed documentation, together with the unprecedented file structure[1], makes it well maintainable.

However, when the CPS model will be used as basis for case specific solutions by IBM, it will most likely have to be adapted to fit the case perfectly and to increase its performance. A case will have its own GUI, its own connections to other data sources and also its own specifics that allow for optimizations.

Because CPS has a clean, generic design, has acceptable performance in various application domains, is well documented, and is well maintainable, it is considered to be well suited as a stepping stone for future work, for both practical implementation of domain specific cases IBM will come across, and continued development of the model.

6.1 Future Work

In this section some topics for possible future work are discussed.

6.1.0 Search Phases

Currently, the CPS model only uses the 'default' search of CPO. There was some experimentation with 'Extended' inference and configuration of search phases (in the confidential part of the thesis). Search phases can be used to tell CPO what part of the solution to solve first. For example, it is possible to instruct CPO to first determine which tasks (out of alternatives) should be present and solve the rest afterwards.

Using a parameter (in the Parameters table) to set the inference level would be trivial. However, generic support of search phases is a big challenge.

6.1.1 Confidential Part of Thesis

Confidential part of thesis.

6.1.2 Research into optionality

The CPS model supports optionality in a limited sense. A root project has the attribute *forcePresence* which can be set to false if excluding that project tree is an option. The CPS model does not support optionality on individual tasks. The decision to do this is based on the fact that without optional tasks in the project network, a stronger 'push-down-presence' construct could be created (see Section 3.1.0 and the example reasoning: when you are building an airplane, attaching the right wing is not optional).

[1] Acccording to Wim Nuijten, Philippe Laborie, and Stéphane Michel

Optionality can still be 'faked' in CPS, by giving a task a 0-duration alternative task. The performance difference in CPO between this 'fake' optionality and real optionality has not been tested.

6.1.3 More performance tests on cost as Objective Function

Most of the tests in Chapter 5 were performed with makespan as objective function. Only in the Aircraft Assembly case, performance was tested with a cost-based objective function.

Almost all research in the area of RCPS uses makespan as objective function. Therefore, most of the generated/constructed instances are designed for this. The weighted cost objective function of CPS is very flexible and supports many different real-life cases. However, not many performance tests have been done in this area. This remains a subject of future research.

6.1.4 Confidential Part of Thesis

Confidential part of thesis.

6.1.5 Disruption Management

Creating an optimal schedule at the start of the project is undeniably useful. However, in practice, it frequently happens that a schedule is no longer feasible, because of disruptions (illness for example). In such cases, the schedule would have to be *repaired*. This is usually referred to as disruption management.

A very basic form of repairing a schedule, which could be done in the CPS model, is removing completed tasks from the input and having CPO process it again. This has a lot of drawbacks. One major drawback is that the newly generated schedule could look very different from the original schedule, which is not very practical. In order to solve this, the original schedule would have to be part of the input, with a progress indication for each task. Constraints could be designed that limit the differences between the original schedule and the repaired schedule. With a new objective function, such a difference could be minimized.

As an extension of the CPS model, this is a very interesting area of research.

List of Abbreviations

CPS(P)	Complex Project Scheduling (Problem)
RCPS(P)	Resource-Constrained Project Scheduling (Problem)
RMS	Rail Maintenance Scheduling
COS	IBM ILOG CPLEX Optimization Studio
OPL	Optimization Programming Language, used in COS
CPO	IBM ILOG CP Optimizer, the technology in COS that will be used to solve the problem
ODME	IBM ILOG Optimization Decision Manager Enterprise

List of Figures

Bibliography

[AFC07] E. Alba and J. Francisco Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, 2007.

[Bak74] K.R. Baker. *Introduction to Sequencing and Scheduling.* 1974.

[BJ86] Cellary W. Slowinski R. Blazewicz, J. and Weglarz J. Scheduling under Resource Constraints: Deterministic Models. *Annals of operations research*, 7, 1986.

[BK12] P. Brucker and S. Knust. Resource-constrained project scheduling. *Complex Scheduling*, pages 117–238, 2012.

[BLK83] J. Blazewicz, J.K. Lenstra, and AHG Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[BPN01] Ph. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* 2001.

[BVQ08] F. Ballestín, V. Valls, and S. Quintanilla. Pre-emption in Resource-Constrained Project Scheduling. *European Journal of Operational Research*, 189(3):1136–1152, 2008.

[Cra96] J.M. Crawford. An approach to Resource Constrained Project Scheduling. In *Proceedings of the 1996 Artificial Intelligence and Manufacturing Research Planning Workshop*, page 35, 1996.

[DRH99] B. De Reyck and W. Herroelen. The multi-mode resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 119(2):538–556, 1999.

[Har01] S. Hartmann. Project scheduling with multiple modes: a genetic algorithm. *Annals of Operations Research*, 102(1):111–135, 2001.

[HK10] C. Heimerl and R. Kolisch. Scheduling and staffing multiple projects with a multi-skilled workforce. *OR spectrum*, 32(2):343–368, 2010.

[JMR+01] J. Józefowska, M. Mika, R. Różycki, G. Waligóra, and J. Weglarz. Simulated annealing for multi-mode resource-constrained project scheduling. *Annals of Operations Research*, 102(1):137–155, 2001.

[KH06] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for Resource-Constrained Project Scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.

[Lab09] P. Laborie. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 148–162, 2009.

[LG07] P. Laborie and D. Godard. Self-Adapting Large Neighborhood Search: Application to single-mode scheduling problems. *Proceedings MISTA-07, Paris*, pages 276–284, 2007.

[PBMN07] C. Pessan, O. Bellenguez-Morineau, and E. Néron. Multi-skill project scheduling problem and total productive maintenance. *Proceedings of MISTA*, pages 608–610, 2007.

[PV10] V.V. Peteghem and M. Vanhoucke. A genetic algorithm for the preemptive and non-preemptive multi-mode resource-constrained project scheduling problem. *European Journal of Operational Research*, 201(2):409–418, 2010.

[RBL07] J. Roca, F. Bossuyt, and G. Libert. PSPSolver: An Open Source Library for the RCPSP. *PlanSIG 2007*, page 124, 2007.

[RVBW06] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*, volume 35. Elsevier Science, 2006.

[SD98] A. Sprecher and A. Drexl. Multi-mode Resource-Constrained Project Scheduling by a simple, general and powerful sequencing algorithm. *European Journal of Operational Research*, 107(2):431–450, 1998.

[ST11] M. Santos and A. Tereso. On the Multi-mode, Multi-skill Resource Constrained Project Scheduling Problem–A Software Application. *Soft Computing in Industrial Applications*, pages 239–248, 2011.

[ZLS11] J. Zhu, X. Li, and W. Shen. Effective genetic algorithm for resource-constrained project scheduling with limited preemptions. *International Journal of Machine Learning and Cybernetics*, 2(2):55–65, 2011.

[ZLT06] H. Zhang, H. Li, and CM Tam. Particle swarm optimization for preemptive scheduling under break and resource-constraints. *Journal of construction engineering and management*, 132:259, 2006.

Appendix A

Assertions Input Data

The typing of the parameters and return values covers a lot of trivial assertions.

Employees

A.E.0: Availability windows do not overlap:

$$\forall\, e \in E : \forall\, w_1, w_2 \in availability(e) \mid w_1 \neq w_2 \\ : end(a_1) < start(a_2) \;||\; start(a_1) > end(a_2)$$

Equipment

A.Q.0: Availability windows do not overlap: similar to the one for Employees.

Consumables

A.C.0: Initial amount between lowerbound and upperbound:

$$\forall\, c \in C : lowerbound(c) < initialAmount(c) < upperbound(c)$$

Projects

A.P.0: Allowed time windows do not overlap:

$$\forall\, p \in P : \forall\, w_1, w_2 \in allowedWindows(p) \mid w_1 \neq w_2 \\ : end(a_1) < start(a_2) \;||\; start(a_1) > end(a_2)$$

A.P.1: "There are no circular constructs in the Project hierarchy".

A.P.2: For tasks, *maxPartDuration* is larger than *minFirstPartDuration*, etc:

$$\forall\, t \in T : maxPartDuration(t) \geq minFirstPartDuration(t) \\ \wedge\, maxPartDuration(t) \geq minLastPartDuration(t) \\ \wedge\, maxPartDuration(t) \geq minMiddlePartsDurations(t) \\ \wedge\, duration(t) \geq minFirstPartDuration(t) \\ \wedge\, duration(t) \geq minLastPartDuration(t) \\ \wedge\, duration(t) \geq minMiddlePartsDurations(t)$$

$$\wedge\ maxPartSeparation(t) \geq minPartSeparation(t)$$

A.P.3: Never produce/consume more of a consumable than available (maximum):

$$\forall\, p \in P, c \in C : abs(producesXOfConsumable(p, c)) \leq upperbound(c) - lowerbound(c)$$

Precedences

A.R.0: Precedence points to different projects:

$$\forall\, r \in R : project1(r) \neq project2(r)$$

A.R.1: *maxDelay* is larger than *minDelay*:

$$\forall\, r \in R : maxDelay(r) \geq minDelay(r)$$

A.R.2: Precedence is not about children in an alternative construction:

$$\begin{aligned}\forall\, r \in R \mid\ & hasParent(project1(r)) \\ & \wedge\ hasParent(project2(r)) \\ & \wedge\ parent(project1(r)) = parent(project2(r)) \\ & : !isAlternativeNode(\ parent(project1(r))\ )\end{aligned}$$