Université Catholique de Louvain

Ecole Polytechnique de Louvain

Constraint-Based Scheduling: Modeling and Filtering

Supervisor:
Pr. Pierre Schaus

Readers:
Renaud Hartert
Sascha Van Cauwelaert

Thesis submitted for the
Master degree in Computer Science
(120 credits)
option Artificial Intelligence
by

David Monjoie
Remi Piron

Louvain-la-Neuve
Academic Year 2013-2014

Abstract

Scheduling problems are critical for the industry as they have numerous applications in a broad range of fields. Solving these problems is no easy task, but people have developed systems dedicated to it. We present our contributions to one of those systems: OscaR.

We aim to contribute to OscaR on two different aspects: usability and performance. In the first part of this thesis, we tackle usability by developing a domain specific language designed especially to model scheduling problems. In the second part we try to improve OscaR performances by implementing a Timetable Edge Finding filtering algorithm for discrete cumulative resources, based on a recent algorithm presented by Petr Vilím.

Acknowledgments

We would like to thank everyone who supported us during this full year of work, except that annoying guy who kept asking how it was going just to drive us mad.

Thanks to everyone who cheered us up during the dark times when nothing was going the way we wanted.

We also thank our assistants and supervisor because they proved very patient in answering our questions during the year.

Contents

Introduction

Scheduling problems are critical for the industry as they have numerous applications in a broad range of fields. For this reason, tools like OscaR have been developed over the years. There are multiple factors playing a role in the success of such tools but we will concentrate on two of them: performance and usability. The former is required for the tool to achieve its goal, that is solving hard problems in an acceptable time. The latter, while not actually needed from a functional point of view, is essential for the tool to be used by many. It is important for people to be able to use the tool efficiently, with as low a learning curve as possible.

Other tools have been developed already to solve scheduling problems like ILOG Scheduling OPL and Comet, but they are quite complicated to use. They require to be well-versed in programming and algorithmics, as well as having a very good understanding of the problem to be modeled.

The OPL tool is primarily aimed at modeling optimization problems, and while scheduling is completely possible, a proper DSL for this specific class of optimization problems is lacking. The user is then required to manipulate problem variables directly, making it difficult for newcomers, and hindering the readability of the finished models.

In the case of Comet, the approach is a bit better due to specific modeling classes regarding scheduling. However, one still need to be familiar with the algorithmic aspects of the problem, and the models are not much more readable either. In other words, getting one's hands dirty is required to use both of these tools. Improvements have been done recently for Comet though, as the AEON system aims for high-level modeling of scheduling problems, as presented in Jean-Noël Monette's thesis. [1]

The modeling functions of those two tools are, however, limited in their readability and user-friendliness by the languages they are respectively written into. OscaR being written in Scala, we have access to all the features that makes Scala a very commonly used language in DSL development. Scala notably has parenthesis omission, implicit type conversion and multiple traits mixing features that greatly improve DSL readability and usability. Those features will be introduced later for people unfamiliar with Scala.

In this thesis we aim to tackle two of the aforementioned important aspects of modeling tools, that is usability and performance. We will present our contributions to OscaR regarding those two factors in the following pages.

Considering the usability aspect, we aim to improve OscaR by developing a domain-specific language designed for modeling scheduling problems easily and in a nice readable language. We would like our DSL to require as little programming knowledge as possible. We also wanted our design to be very friendly to additions, so developing new features for the DSL should be easy.

On the performance aspect, we will tackle the implementation of a specific filtering algorithm aiming to improve performances in solving problems using discrete cumulative resources.

To achieve our goals on the usability front, we first start by reviewing popular classes of problems, well established in the scheduling community, to see what features are required and what is often used. Then we choose a subset of those features to implement. We describe our DSL architecture, what constraints can be used and how we implemented them. As for the performance front, we start from the paper of Petr Vilím regarding Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources [2] and port it to Scala and into OscaR.

The remainder of this document is structured in two parts. The first part will tackle the usability front and describe our scheduling DSL, while the second part will present our solution on the performance front, that is the filtering algorithm we implemented in order to improve OscaR performances on specific ranges of problems. Those two parts will provide the required background materials for their respective content as well as reporting in more details about related works, before delving into our solution.

The interested reader may find the source code of our solution in the `memScheduling-dev` branch of the `OscaR - memScheduling` fork of `OscaR`, hosted on Bitbucket at https://bitbucket.org/malorne/oscar-memscheduling.

Part I

Modeling Scheduling Problems in OscaR

Chapter 1

Background Material

For people unfamiliar with Scala, we provide explanations on particular features of the language which are not standard in other programming languages. We have built our DSL around those features so it is critical to know them in order to understand the details of our implementations.

1.1 Implicits

Scala allows functions and function parameters to be defined as `implicit`.

For implicit function parameters, the user can call the function without providing a value for the implicit parameter. This is similar to default function parameters, but the value is provided by the implicit value at the call site. For example, consider the following function:

```
def function(implicit i: Int) = println(i)
```

The function can be called without actually giving the `i` as an argument if there is an implicit `Int` in scope at the call site, for example:

```
implicit val i = 1
function()
```

Implicit parameter resolution is done with respect to the parameter type, so the following is valid:

```
implicit val i = 1
implicit val j = "2"
function()
```

Because `j` is a `String`, it doesn't comply with the implicit parameter type which is an `Int`. However, the following is not valid:

```
implicit val i = 1
implicit val j = 2
function()
```

Because the compiler sees two implicit values that can be used as implicit parameter to the function call, it cannot decide between the two and will produce an error. We use implicit parameters a lot in our DSL, so we need to be careful to have only one implicit parameter of the given type at the call site.

For implicit functions, they are used to implicitly convert object between types or add functions to existing types. For example, consider the following class:

```
class BoxedInt(i: Int)
```

One can have instances of the `Int` type automatically converted into a `BoxedInt` by defining the following implicit function:

```
implicit autoBoxInt(i: Int) = new BoxedInt(i)
```

One can then write:

```
val b: BoxedInt = 1
```

And the implicit function will be used to provide the conversion from `Int` to `BoxedInt`. We use this a lot in our DSL to help the user converting types. Thanks to functions like this one, an input of type `A` can be used in a function requiring a type `B` provided there is an implicit conversion from `A` to `B`. This renders the code cleaner and more readable by avoiding explicit conversion functions.

One can also directly define the class itself as `implicit` to get the same result without having to define a specific conversion function. This can even be used to add functions on existing types by defining new functions on the implicit class:

```
implicit class BoxedInt(i: Int) {
  def printThis = println("This integer is " + i)
}
```

One can then write:

```
1.printThis
```

And the implicit conversion from `Int` to `BoxedInt` will be used to provide the `printThis` function. We also use this in some features of our DSL.

1.2 Parenthesis Omission

Scala allows users to omit parenthesis in function calls under certain conditions. For example, consider the following class:

```
class A {
  def printThis(i: Int) = println(i)
}
```

Providing variable `a` is an object of type `A`, one can write:

```
a printThis 3
```

And it will be equivalent to:

```
a.printThis(3)
```

But a lot more readable. One can also chain function calls, so considering function `f(i: Int)` of object `b` returns another object on which one can call `g(j: Int)`, the following is valid:

```
b f 1 g 2
```

We use this a lot in our DSL to build constraints that looks like actual sentences. For example we have a requirement function that reads:

```
activity needs 3 ofResource r
```

There are restrictions however. This can only be used with functions having one and only one parameter, and the function needs to be called on an object. For example one cannot write:

```
def printThis(i: Int) = println(i)
printThis 3
```

But it is valid to write `this printThis 3`, because the `printThis` function is then called on object `this`.

We had to design our functions around these limitations and it wasn't easy in some cases. Other limitations exist but are more case-specific so we will discuss them at the time they arise.

1.3 Traits

Scala has a very powerful feature called Traits. A Trait is like a Java interface, but it allows functions to be implemented. Traits can also be mixed with existing classes to add support for those interfaces to the class. Consider the following simple traits:

```
trait A {
  def a(i: Int) = println("a " + i)
}

trait B {
  def b(i: Int) = println("b " + i)
}
```

Considering a `C` class with no methods, one can mix it with one or more traits to gain their features. For example one can write:

```
val c = new C with A with B
```

Because `c` is an instance of class `C` mixed with both traits, functions `a` and `b` can be called on it. The following calls are valid:

```
c a 1
c b 2
```

The `c` variable is actually of type `C with A with B` instead of simply being an instance of class `C`, which is why it has methods `a` and `b` even though they don't exist in class `C`.

One can even design a trait extending another trait, which can then override the functions of its super trait. It can be useful to modify the current behavior of an object with respect to a special case handled by the new trait.

Actually, traits are like plugins, and they are very powerful in that regard. We designed nearly all our DSL using such traits, so one can enable or disable any feature by mixing or unmixing any trait. This also creates a very extension-friendly environment because it is very easy to create a new trait and mix it with the others to get a new feature.

1.4 Companion Objects

In Scala, companion objects are often used to build custom creation statements. For example, considering the following class:

```
class A(size: Int)
```

If the user wants to create a new instance of class `A` of size 3, he has to write:

```
new A(3)
```

There is no way of giving informations on the meaning of the class parameter, other than giving it a nice descriptive name. Still, the parameter name is lost when reading the final code. If a reader doesn't know about class `A`, there is no way for him to infer that the parameter is the size from the code alone. However, by defining a creation function on the companion object of class `A`, like this:

```
object A {
  def ofSize(size: Int) = new A(size)
}
```

The user can then create an instance of `A` by writing:

```
A ofSize 3
```

Which is a lot more expressive. One can then execute any interesting code in a companion object creation function, including mixing traits. One could return an `A` mixed with another trait without requiring the user to explicitly write the trait mixing, for example. We use this feature when creating activities in our DSL.

A special `apply` function can also be defined in Companion Objects. The user can then call the companion object itself like a function. For example, consider the following companion object:

```
object B {
  def apply(i: Int) = new A(i)
}
```

The user can simply write `B(3)` and this will actually call the `apply` method of `B`. This is very useful when designing short creation statements, as well as hiding the `new` keyword.

Chapter 2
Related Works

Work on scheduling problems modeling is not something new, as it has already been done by IBM with ILOG Scheduling OPL and Dynadec with Comet. However, those systems are not really easy to use, and require to be well-versed in computer programming and algorithmics in general. They are not really intuitive and one have to know those systems capabilities pretty well in order to find what functions to use to post the right constraints.

As an example, we will look at the code used to model a Job Shop Scheduling Problem in both languages. The Job Shop problem involves a series of tasks, grouped as jobs, that are to be processed on a set of machines. Tasks within a job are to be processed in a given order, and the machines can only process one task at a time. The goal of the problem is to minimize the makespan, that is the end of the latest task.

Here is the OPL code used to model a Job Shop Scheduling Problem, as described in the IBM Online Informations Center [3]:

```
using CP;

int nbJobs = ...; // Input from file
int nbMchs = ...; // Input from file

range Jobs = 0..nbJobs-1;
range Mchs = 0..nbMchs-1;
// Mchs is used both to index machines and operation position in job

tuple Operation {
   int mch; // Machine
   int pt;  // Processing time
};

Operation Ops[j in Jobs][m in Mchs] = ...; // Input from file

dvar interval itvs[j in Jobs][o in Mchs] size Ops[j][o].pt;
```

```
dvar sequence mchs[m in Mchs] in
          all(j in Jobs, o in Mchs : Ops[j][o].mch == m) itvs[j][o];

execute {
   cp.param.Workers = 1;
   cp.param.FailLimit = 10000;
}

minimize max(j in Jobs) endOf(itvs[j][nbMchs-1]);
subject to {
  forall (m in Mchs)
    noOverlap(mchs[m]);
  forall (j in Jobs, o in 0..nbMchs-2)
    endBeforeStart(itvs[j][o], itvs[j][o+1]);
}

execute {
  for (var j = 0; j <= nbJobs-1; j++) {
    for (var o = 0; o <= nbMchs-1; o++) {
      write(itvs[j][o].start + " ");
    }
    writeln("");
  }
}
```

It is very hard to understand what is going on unless one have a very good knowledge of the problem that is modeled and the way OPL works, or at least a strong programming background. The reader should note that the user needs to define an `Operation` tuple on his own. Because OPL only works on intervals, it has to no actual concept of machines or resources. As such, a `noOverlap` function is then used to model the behavior of the unary resources. For the precedences constraints, the `endBeforeStart` function is used. This function name is expressive, but one needs to know all existing similar functions to be able to model a problem, as there is no way for a user to discover them by himself.

Actually, OPL doesn't have a proper scheduling DSL. It is an optimization tool before anything else, and while it is possible to solve scheduling problems using it, it is not really easy to use.

Now looking at Comet, here is the code for the same problem, from the Comet manual [4]:

```
import cotfd;

int   nbJobs; // read from file
int   nbTasks; // read from file
range Jobs = 0..nbJobs-1;
range Tasks = 0..nbTasks-1;
range Machines = Tasks;
int   nbActivities = nbJobs * nbTasks;
range Activities = 0..nbActivities+1;

int   duration[Jobs,Tasks]; // read from file
int   machine[Jobs,Tasks]; // read from file

int horizon =     sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP>     cp(horizon);
Activity<CP>      a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP>      makespan(cp,0);
UnaryResource<CP> r[Machines](cp);

minimize<cp>
  makespan.start()

subject to {
  forall (j in Jobs, t in Tasks : t != Tasks.getUp())
    a[j,t].precedes(a[j,t+1]);
  forall (j in Jobs)
    a[j,Tasks.getUp()].precedes(makespan);
  forall(j in Jobs, t in Tasks)
    a[j,t].requires(r[machine[j,t]]);
}

using {
  forall(m in Machines) by (r[m].localSlack(),r[m].globalSlack())
    r[m].rank();
  makespan.scheduleEarly();
  cout << "Makespan: " << makespan.start() << endl;
}
```

This is better than OPL, we can clearly see the concepts of activities and unary resources being actual modeling classes. However, Comet usage is not completely convenient either. As one can see, the user needs to carry the `cp` variable around a lot. The makespan is not a function of the language, it is modeled as an activity with a duration of 0 which is preceded by every other activity. The constraints are similar to OPL, but we now have a real `requires` constraint, as we have proper modeling of resources. However, one still need to be well-versed in programming to be able to understand what is going on, due to the extensive use of arrays indexing.

The AEON model is more structured and a bit cleaner, as one can see in this example from Jean-Noël Monette's thesis. [1]:

```
range jobs = 1..nbjobs;
range machines = 0..nbmachines-1;
range tasks = 1..nbjobs*nbmachines;
int proc[tasks];
int mach[tasks];
int job[jobs,machines];

Schedule<Mod> s();
Job<Mod> J[i in jobs](s,IntToString(i);
Machine<Mod> M[i in machines](s,IntToString(i));
Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));
forall(i in tasks)
   A[i].requires(M[mach[i]]);
forall(i in jobs)
   J[i].containsInSequence(all(j in machines)A[job[i,j]]);
s.minimizeObj(makespanOf(s));

GreedyTabuSynthesizer synth();
Solution<Mod> sol = synth.solve(s);
sol.printSolution();
```

AEON is much closer to what we want from a DSL than were the other two. However, we can see it still carries the `cp` variable, although it is called `s`. This time the concept of makespan directly exists in the system, the user is not required to model it by himself. We can see the jobs are actual objects of the model in AEON, and the precedences constraints between the activities are posted through the `containsInSequence` function.

However, this modeling still suffers from flaws inherited from Comet. The extensive use of indexing is one of them, as well as the `<Mod>` notation, which was `<CP>` in Comet. Those notations are used to differentiate classes that are part of the modeling, which are the `<Mod>` and the ones that are part of the solver, which are the `<CP>` of Comet and are hidden from the perspective of the AEON user. This notation, along with the extensive use of `IntToString`, hinders the readability of the model, though.

We aim at designing a DSL that will be easier to read and understand, even for people with little knowledge in programming. We would like our models to be clear and readable, allowing a newcomer to understand a scheduling problem by reading one of our model. Current solutions require the user to have a good knowledge of the problem before even attempting to read any modeling code for this problem. They would directly get lost otherwise, unless they are used those systems modeling languages and can get on their feet from there. People with little knowledge in both scheduling and programming have no way of understanding such models.

We want to lower the barrier to entry for scheduling problems modeling, by designing a DSL that is easy to use, clear and produces understandable models even for people with little programming knowledge.

Chapter 3
Problem Analysis

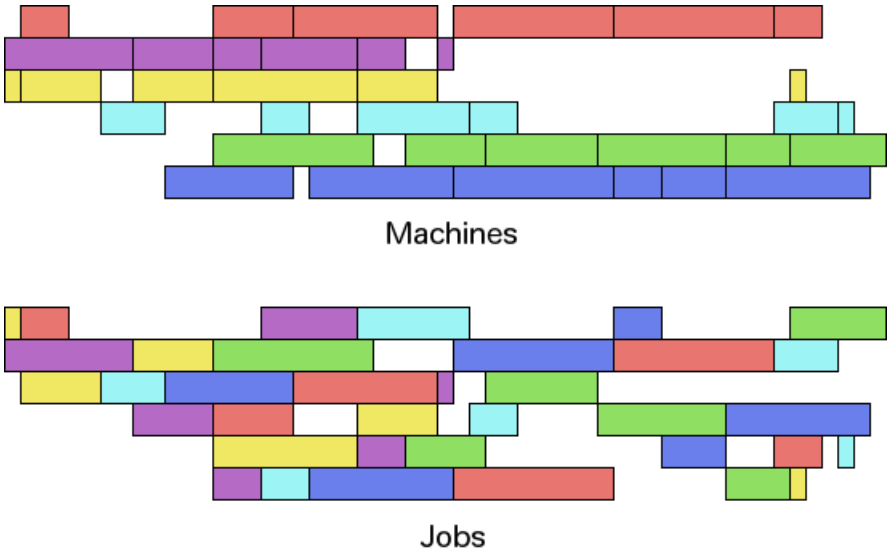
The problem we want to tackle can be stated as follow: *"How can we model scheduling problems?"* Current solutions to that problem already exist as we discussed earlier. However, we want to design our DSL with usability and readability in mind and do better on those fronts than what has already been presented. We want our DSL to be as simple to use as possible while staying expressive enough. By simple we mean easy to understand when a user reads a model designed using it, easy to guess what function one needs from its name in a list of possible functions, and easy to get a hand on without needing to know a lot about Scala or programming in general. When it comes to expressiveness, what we are going to do is review some well known scheduling problems and make a list of required or often used features. We'll then try to implement as much of them as we can. For these problems, we started with the list used by P. Laborie and D. Godard in their paper about Self-Adapting LNS [5] of well established problems in the scheduling community, then we also researched for other interesting scheduling problems.

3.1 Review of Scheduling Problems

3.1.1 The Job Shop Scheduling Problem

The goal of this problem is to minimize the completion time, also called makespan, of a set of jobs that are to be scheduled on a set of machines. A job is a sequence of tasks, also called activities, each requiring a different machine. The machines are unary resources, which means only one activity can be scheduled on a given machine at any given time. Activities are totally ordered within a given job. [6]

For example, the following figure shows the activities of a Job Shop Scheduling Problem colored by their required machines. The activities on the first visual are grouped by machines, so each activity on the same line is processed on the same machine, while the ones of the second are grouped by jobs.



One can see that the third task of the first job cannot be scheduled earlier than it is already, because it needs the purple machine, which is already fully used until then. The fourth one of the first job could be processed earlier with respect to its resources requirements, as the cyan resource is not fully used. However, it is blocked by the third one which must be completed before the next one can start, as both have to comply with the total ordering of the job.

Multiple variants of this problem exist, we describe some of them in the following:

Multi-Processor Job Shop Machines are cumulative resources instead of unary, which means multiple activities can be scheduled on the same machine at the same time. The number of activities that can be scheduled on a given machine cannot exceed the machine's capacity. [7]

Preemptive Job Shop Activities are preemptive, which means they can be interrupted at one point during execution, freeing the resource usage, and then resumed later. [8]

Job Shop with Earliness/Tardiness Each job has a due date, and generates a cost if it finishes earlier or later than its due date. The goal of the problem then becomes minimizing these costs. [9]

Job Shop with Setup Times In addition to the standard Job-Shop constraints, additional setup times are to be taken into account between

some activities when they are consecutively executed on the same resource. When an activity is to be scheduled right after another one on the same machine, there is a time that must elapse between those two activities, this time is the setup time between those activities on that resource. [10]

Considering all those variants, this problem uses the notions of precedences, unary resources requirements, cumulative resources requirements, preemptibility, earliness/tardiness and setup times.

3.1.2 The Flow Shop Scheduling Problem

This problem is itself a kind of variant of the Job Shop problem, in the sense that it has the same design of jobs and machines, but with an additional constraint that fixes the jobs requirements structure. Each jobs must be processed by each machine exactly once and in the same order. This means that every job must first execute on the first machine, then they all need to execute on the second machine and so on. [7]

There also exists multiple variants of this class of problem, like:

Multi-Processor Flow-Shop Like Multi-Processor Job Shop, multiple jobs can be processed on one machine simultaneously, up to the machine's capacity. [7]

Re-entrant Flow Shop While every job still retain the same structure, so the same machine requirements ordering, a machine can be required twice by the same job instead of exactly once. [11]

Flow Shop with Earliness/Tardiness Like Job Shop with Earliness/Tardiness, each job has a due date and the goal of the problem becomes minimizing the costs of finishing the jobs earlier or later. [12]

Flow Shop with buffers In this variant, there are buffers of fixed size between the required machines. When a job has finished executing on a given machine, it can either be processed on the next required machine if it is available, or be delayed in its respective buffer. This is not a simple relaxation of the standard Flow Shop problem however, as the job will block its current machine if it cannot exit it because the next one is not available and its buffer is already full, blocking the current machine until it can either enter the next machine or its buffer. [13]

Considering all variants of Flow Shop, this problem uses the notions of precedences, unary resources requirements, cumulative resources requirements, earliness/tardiness, and resources buffers.

3.1.3 The Open Shop Scheduling Problem

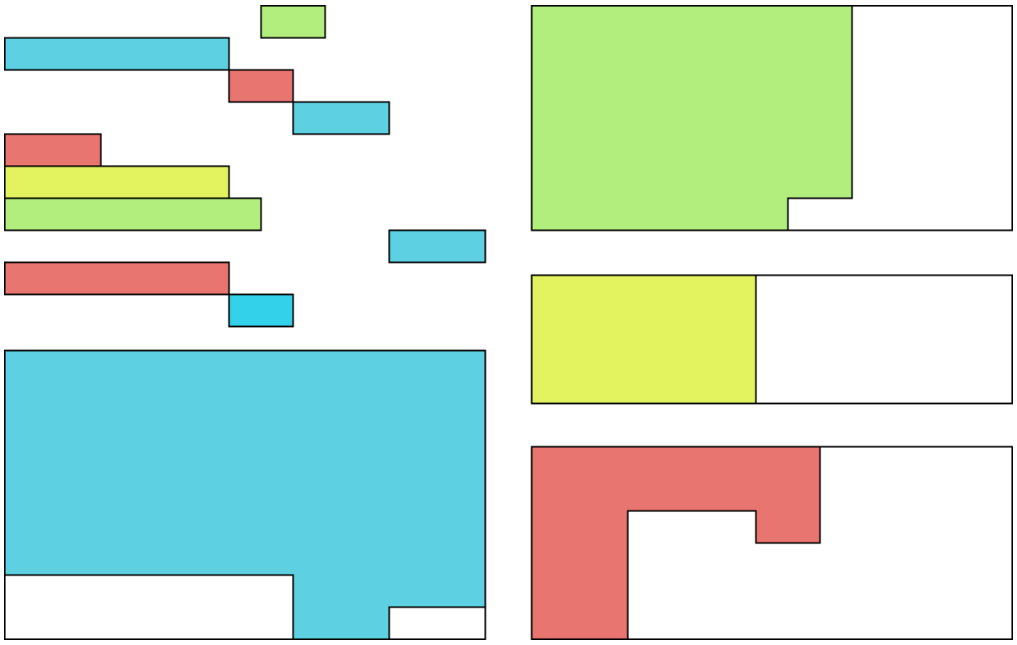
This problem is similar to the Job Shop Scheduling Problem, but has a different signification of the job concept. In the standard Job-Shop the activities are totally ordered within a given job. In the Open-Shop there is no precedence constraints within activities of a given job. However, at most one machine can process activities of a given job at any given time, which means at most one activity inside a given job can be scheduled at any given time. [14] This problem uses the notions of unary resources and a way to enforce the "one activity per job at a time" constraint, although this can also be modeled using unary resources.

3.1.4 Resource Constrained Project Scheduling Problem (RCPSP)

The goal of this problem is to minimize the completion time of a set of activities that have fixed resources requirements and are linked by precedences relations. The resources can be renewable or non-renewable. A resource is non-renewable when processing an activity on it does not return the resource capacity used by the activity after it has finished processing it. The resource capacity is consumed up to a certain level by the activity and not refunded. For example, consider taking two units of supplies from a stock of supplies. Unlike the Job Shop Scheduling Problem, precedences constraints are not linked to the concept of job and an activity can be preceded by multiple others or have multiple successors. Because of this and cumulative resources, overlap can occur between activities requiring the same resource, provided the sum of their requirements does not exceed the resource capacity. [15]

For example, the figure on the next page shows the activities of a RCPSP as well as the capacity usage of each resource. We colored the activities with respect to their required resource.

One can see the first activity starting from top starts pretty late. However, the resource it needs, the green one, is fully used by the other green activity until then. The first two red activities can overlap because their joint usage of the resource does not exceed its capacity.



RCPSP also has interesting variants, like the following:

Multi-mode RCPSP Activities can be processed in different modes, each mode having its own resources requirements. The goal of the problem is the same but now the algorithm also has to decide which of the alternative modes the activities will be executed into in order to minimize the makespan. [16]

Max. quality RCPSP Each activity has a quality variable that is dependent on its duration. The goal of the problem then becomes determining the activities durations that maximize the overall quality, standard RCPSP constraints remaining valid. [17]

RCPSP with Earliness/Tardiness Like the jobs presented earlier, each activity has a due date and penalties for finishing earlier or later than that. The goal is to minimize the ensued penalty costs. [18]

Considering those variants, this problem uses the notions of precedences, unary resources requirements, cumulative resources requirements, non-renewable resources, alternative resources, and earliness/tardiness.

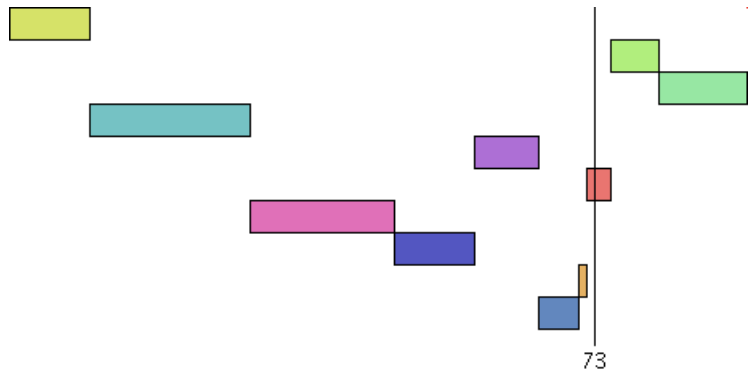
3.1.5 Single Machine Weighted Earliness/Tardiness

This problem is the simplest Earliness/Tardiness problem and its principle is the basis for the Earliness/Tardiness variants of other problems. A set of jobs is to be scheduled on a single machine, and each job has an assigned due date. The goal of the problem is to minimize the total penalty costs of jobs finishing earlier or later than their respective due date. [19] This problem uses the notions of precedences, unary resources requirements, and earliness/tardiness.

3.1.6 The Common Due Date Problem

In this problem, a set of jobs is to be scheduled on one machine. There is a unique due date for all jobs that is usually impossible to meet, the goal of the problem being minimizing the penalty costs of the jobs that finishes earlier or later than the common due date. [20]

For example, the following figure show the optimal solution of a problem with a due date of 73 which is completely impossible to meet considering all activities require the same unary resource. It is necessary to choose which activity to prioritize with respect to the cost incurred by each of them when not finishing on the due date.



As one can see, the yellow activity has a low earliness cost because it is scheduled way too early with respect to the due date. The orange and red activities however, have high earliness and tardiness costs, because they have been given the priority with respect to the due date as they are scheduled pretty close to it.

This problem uses the notions of precedences, unary resources requirements, and earliness/tardiness.

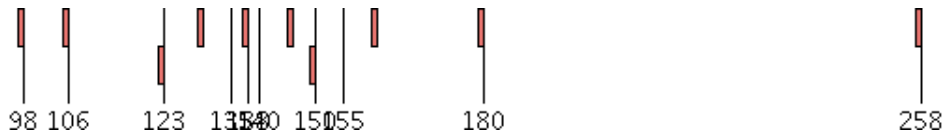
3.1.7 The Trolley Problem

This problem is based on the Job Shop Scheduling Problem, but physical distances between the machines are modeled. Each job represents the creation of an item, which has to be moved from one machine to another as it goes from a step to the next in its creation process. A trolley is used to perform this movement, which is a resource that can take multiple states, for example its position in the factory or the fact that it is loaded or not. Activities, which represent item creation process steps, require the trolley to be in a certain state to be executed, that is the trolley having successfully moved the item to the right machine. The goal is to minimize the makespan taking into account the time to load, move, and unload the trolley between two different locations. The trolley may also carry multiple items at a time, up to a given capacity. [21] This problem uses the notions of precedences, cumulative resources requirements, and state resources requirements.

3.1.8 The Aircraft Landing Problem

In this problem, we try to schedule airplane landings under the constraints of acceptable time windows and some separation criterias between a landing and successive ones. We reference the static case in [22], which means we are not scheduling the landings in real time when planes appear. We are scheduling all landings at once in their respective time windows while trying to minimize their earliness and tardiness costs with respect to their respective target landing times. The separation criterias are separation times when landing on the same runway for safety purpose. Depending on the type of planes, the separation times can be shorter or longer between two given planes than between two others.

For example, the following figure shows an optimal assignment for an Aircraft Landing Problem using two runways. The vertical lines show the target landing times, while the rectangles are the activities, which represents landings in this problem.



As one can see, there are three target landing times which are very close to each other. Because of separation times constraints, the third and seventh landings had to be scheduled on the second runway to relax the first one a little bit. Those two landings were pushing even more on the three ones which were already too close to land at the right time. It is interesting to note that using a third runway would allow each landing to be scheduled at the right target time.

This problem uses the notions of precedences, unary resources requirements, separation times and time windows.

3.1.9 Personnel Task Scheduling Problem

The goal of this problem is to minimize the overall personnel cost required to perform a given set of tasks. The personnel have shifts during which they can be assigned to a task. As the personnel is heterogenous, each of them can only be assigned to a subset of the tasks which represent the ones they are skilled into. The overall personnel cost can be a complicated function of the personnel used, but a simplified variant of this problem, called *Shift Minimization Personnel Task Scheduling Problem* only considers the cost of personnel shifts and thus aims at minimizing the number of shifts required to schedule the set of tasks. [23] This problem requires the notion of cumulative resources requirements, as we can model the shifts as activities that gives 1 unit of resources to the personnel resource, and optional activities, as we need to be able to choose how many shifts we want to schedule in order to minimize the cost.

3.2 Review of Required Features

Because it is the definition of scheduling problems themselves, all of them feature activities that requires resources and the allocation of those resources over time. We indeed need to design our DSL based on those two basic bricks of activities and resources.

Looking more closely into activities, we can see there are multiple types of activities. Atop the standard activity that is defined by its start, end and duration, we find the preemptible activity that can be interrupted and resumed, and the optional activity, that does not strictly need to be scheduled in order for the solution to be valid.

On the field of resources, we can see they also appear in multiple forms, or different types, like unary resources, cumulative resources, state resources and non renewable resources. Those resources can also be backed-up by a buffer, although it was only seen in one problem variant.

Looking now at the scheduling constraints, we can see there are constraints that affect an activity start and/or end time itself, as well as constraints linking activities among themselves with respect to certain criterias, and finally constraints that link activities to resources. These are indeed the resources requirements, which can also be alternative requirements.

In terms of constraints on the activities themselves, we notably have the time windows, in which an activity has to be processed. For constraints linking activities among themselves, they appear in various forms. We first have the precedences constraints, which are the most prominent of all, which constraint an activity start or end to occur before another activity start or end. We then have the separation times, which constraint the time that needs to elapse between two activities, and finally the setup times constraints, where the delay is only enforced if the activities are consecutively processed on the same resource.

Other interesting constraints include the Earliness/Tardiness penalties, which appear a lot as a variant of other problems. That constraint requires to set a due date for the activities. The user must then be able to retrieve an activity earliness or tardiness with respect to that due date. There are costs incurred when finishing earlier or later than the due date, and those costs are used to respectively weight the earliness and tardiness of activities.

From the previous review, it is clear that our DSL must be able to express activities resources requirements, for unary and cumulative resources at least. Precedences and earliness/tardiness are also a must have regarding the number of problems they appear into. The other features like other resources types and other scheduling constraints appear more scarcely however. Our DSL would fail with respect to the expressive power criteria if we only modeled the three aforementioned most standard aspects of scheduling problems while ignoring the other, more specific features. Because of this, we chose to develop a subset of those additional features.

We list all features expressible in our DSL in the following. This list also presents the user with a kind of roadmap of what will come next, that is the details of the usage and our implementation of said features.

- Activities
- Optional Activities
- Unary Resources
- Cumulative Resources
- Non-Renewable Resources
- Alternative Resources
- Precedences
- Separation Times
- Time Windows
- Earliness/Tardiness

Along with the support of those features of scheduling problems, we also aim our resulting DSL to produce highly clear and readable models. We want to lower the language barrier as much as possible and make scheduling possible for people with limited knowledge in programming. We want to avoid lingering variables like `cp` or notations like `<Mod>`. We also want to simplify the iteration process on sequences of elements, limiting the overhead, in terms of readability, of extensive indexing of sequences. Additionally, we also plan to have a modular design, very friendly to extensions and in which one could easily enable some extensions, disable others, and even create new ones.

Now that we know what to expect from the DSL, let's delve into the details of our solution.

Chapter 4
DSL for Scheduling Problems

In this chapter we describe our DSL. The first section introduces the structure we used for our solution, while the other sections each introduce a particular feature. Each feature section contains two distinct parts. The first one describes the feature: what can the user express with it and how can he express it, while the second part delve more into the technical details of the feature. The idea behind this pattern is that a user should be able to start using the DSL only by reading the first part of each feature section and shouldn't feel the need to read the technical details in order to simply use our solution. The technical details parts are more oriented towards discussing the development choices, how we managed to implement each feature and what is interesting to know for someone who would like to continue working on our solution.

4.1 Structure of our DSL

As all scheduling problems revolve around the concepts of activities and resources, it's not a surprise for anyone that those classes are the fundamental bricks of our DSL. At the lowest level, we have these two obvious classes: `Activity` and `Resource`. Every other feature of our DSL refers to those classes.

Notable features that have been built around those classes are activities traits. Each trait contains features that one can enable for his activities. For example the `Precedences` trait contains all the precedences constraints, while the `TimeWindows` one contains the time windows ones, and the `dueDate` trait contains every function related to earliness/tardiness with respect to a due date. All those traits are built around `Activity`. This way, one can extend `Activity` and still retain the compatibility with all other traits.

It is important to note that the `Activity` class only models the concept of an activity and nothing more. It has no function to post constraints on it. To be able to post constraints and actually model problems, one needs to mix some features traits with it. However, doing so for every problem would

be really tiresome so we provided the user with methods in `Activity`'s companion object, which creates instances of `Activity` with all the feature traits mixed in, thus enabling all supported features by default. The point of having traits is to organize each feature in its own block of code so the it is easily manageable and maintainable.

Moreover, the experienced reader will already have realized that these traits can be used like building bricks for custom implementations. As the user has access to `Activity` and all its traits, he can build custom activities very easily. One can choose to enable some features by mixing their respective traits with a custom implementation of activity, and as long as the custom activity extends our `Activity`, it will get the features for free!

By using this design, we achieve the best of both worlds in terms of beginning users needs versus power users. Beginning users have the mixing work done for them thanks to the companion object. Power users, on the other hand, can enjoy the modularity of our activities design. The former can directly use the `Activity` object and be good with it without worrying about all this programming voodoo, resulting in a more user-friendly experience for him, while the latter have endless customizing possibilities at hand without having to modify OscaR code directly as we provide him with building blocks to use and expand on.

Creating activities with all feature traits mixed in has a cost, however. We have to be careful when we design features traits that they do not register any variable or post any constraint if their functions are never called by the user. This will be discussed in more details in the `Optional` and `DueDate` traits sections of this chapter, as those were more tricky to implement in this regard.

Another important technical aspect of our DSL is the `CPScheduler` class, which extends OscaR standard `CPSolver`. Activities and resources both register themselves to it. It assigns IDs to them and posts some constraints when the user has finished modeling the problem, so when every activity and resource has been registered. It also affects the range of possible schedulings for activities over time thanks to its horizon. The `CPScheduler` will be described in more details later in the next section.

4.2 Fundamentals

In this section we introduce the `dsl` package, the `SchedulingModel` trait and the `CPScheduler` class, along with the two fundamental bricks of all scheduling problems: activities and resources. The `SchedulingModel` trait is the main trait to mix in when designing a scheduling model, as it provides the user with an implicit `CPScheduler` for creating activities and resources. As stated earlier, every of the modeling classes is based one way or another on `Activity` or `Resource`. All these classes and traits are all part of the `dsl` package. This section describes those classes and serves as a basis for the remaining of this chapter.

4.2.1 The dsl package

All the modeling classes and functions we designed can be found in the `dsl` package of the `oscar.cp.memScheduling` package of OscaR. We strongly advise the user to import the whole package in his model by adding the following line in the imports section of his code:

```
import oscar.cp.memScheduling.dsl._
```

Scala provides developers with a package scope feature, which allows us to put implicit conversions functions in a file named `package.scala`. Those implicits are then in the scope of the model as soon as one imports the whole `dsl` package, that is with the `_` notation. A user would greatly miss on the easy modeling experience if he wouldn't have those implicits in the scope of his model, thus our strong advise to import the whole package. But what will these implicit conversions actually allow the user to do that he wouldn't be able to do without it?

First of all, it will let him write `Int` values where functions require a `CPIntVar`. Nearly all our functions require `CPIntVar` as arguments, which are actual problem variables. For example an activity can require between 3 and 5 of the capacity of a resource, as the actual requirement depends on other constraints. Because our modeling functions allows the user to state variable arguments, like a variable demand in this case, their arguments are of the type `CPIntVar`. But what if the user is modeling a simpler problem, for which the requirement is already known to be 4, for example? We doesn't want him to manually box the `CPIntVar` variable whose domain would only be the value 4 just for the purpose of right typing of arguments with respect to our functions. That's why we have included implicit conversions from `Int`

to `CPIntVar`. When one writes a simple `Int` where we expected a `CPIntVar`, it is automatically converted into an already bound `CPIntVar`. Implicit conversions also allows the user to write a `CPIntVar` as a `Range`, like `3 to 5`, which would come in handy for certain cases, the variable resource requirement example being a good illustration.

Other interesting implicits enabled by importing the whole `dsl` package are the Enhanced Tuples of sequences and the Enhanced Selectors of sequences. Those are described in more details in the Additional Features section of this chapter.

Technical Details

Unfortunately, there is no way to make Scala use the `Int` to `CPIntVar` implicit when resolving a `Seq[Int]` for a `Seq[CPIntVar]` without writing another function. We then have implicits for this case as well as for the `Seq[Seq[Int]]` case. We also handle the implicit conversion of `Array[Int]` through the use of a type bound in our `Seq[Int]` function. Its argument is actually a type `S <% Seq[Int]`, which covers any type that can be viewed as a `Seq[Int]`, even if an implicit conversion is required, which is the case for `Array[Int]`. This type bound is required as a simple `S <: Seq[Int]`, covering all subclasses of `Seq[Int]` wouldn't enough because Scala would need to chain two implicits to get from `Array[Int]` to `Seq[CPIntVar]`, which it wouldn't do unless using the `<%` type bound. Sequences of ranges are also covered by similar implicits.

There are other implicit conversions implemented in the `package.scala` file, but they will be described later in this chapter, in the feature they are relevant to. Those features mainly comprises alternative requirements notation, which is described in the Alternative Requirements section, and the sequence of activities accessors described in the Creating Activities section.

4.2.2 SchedulingModel

The main purpose of the `SchedulingModel` trait is providing the user with an implicit `CPScheduler`. As discussed in the previous section, the `CPScheduler` registers every activity, resource and variable that are created in the model. It also receives the exploration method and executes it on the model variables. The `CPScheduler` is the one linking the model variables and the search, actually leading to solving the problem.

Despite its central, ever-present nature, we didn't want users to have to manipulate a scheduler variable all the way around their problem model, with functions calls like `scheduler.makespan` or `Activity(scheduler, ...)`, which one can find in OPL, Comet or AEON in the form of the `cp` variable they carry around. To this end, we designed the `CPScheduler` as an implicit parameter for most classes. This way, one can create an activity or a resource without having to write the scheduler explicitly, provided there is one and only one implicit scheduler at the creation site. This is exactly the purpose of the `SchedulingModel` trait.

To use this trait in the model, a user simply has to mix it with his model, like any other trait, by writing `extends SchedulingModel`, for example:

```
object MyModel extends SchedulingModel
```

One then have an implicit `CPScheduler` in the scope of the model. This scheduler can still be accessed like any other variable in the model by writing `scheduler` or `cp`. We prefer to refer to it as `scheduler`, but as it extends `CPSolver`, current users of OscaR may be more comfortable with the `cp` keyword and that's the reason why we also kept `cp`. The same reasoning applies to current users of the other systems.

`SchedulingModel` also lets the user easily access the makespan of the problem. The makespan is the completion time of the latest activity, so the point in time where all activities have been processed. It can be accessed directly by writing `makespan`.

It is also interesting to note that `SchedulingModel` extends the Scala `App` trait, which means that the model will directly be runnable without having to define a `main` function when `SchedulingModel` is mixed in.

Technical Details

From a technical point of view, `scheduler` is not a variable but a function, which returns the implicit scheduler which is a `private val`. The difference between the two arises when dealing with multiple schedulers as discussed in the Multiple Schedulers section in the Advanced Features of this chapter. Because we need to change which scheduler is the current implicit one in order to handle multiple schedulers, we need `scheduler` to be a function and not a `val`, because a `val` is immutable. Using a `var` would have been possible from a technical point of view, but giving the user direct access to the implicit scheduler variable would be asking for troubles. Setting it as a

`private var` to protect it would have required to write an accessor function, defeating the purpose of trying not to use a function in the first place.

The `makespan` function is simply a shortcut for `scheduler.makespan`. This allows writing `minimize makespan` without dot or parenthesis. Without this function, one would need to write `minimize scheduler.makespan` with a dot or `minimize(scheduler makespan)` without the dot but with parenthesis instead.

4.2.3 CPScheduler

As stated earlier, the `CPScheduler` is the one registering every activity, resource and variable. It is the central point of every model, even though we actually try to hide it so the user doesn't have to mess with it too much. It is still a very important part of scheduling problems modeled with our DSL, so we will describe how one can use it.

The first thing the user would want to do with the scheduler, before creating any activity, is to set its horizon. The horizon of the scheduler is the limit in time after which it won't try to schedule activities. For example, if the problem is to schedule multiple activities on multiple resources with the goal of minimizing the makespan, one would want to set the scheduler horizon to the sum of all activities processing times. There is no point in scheduling any activity after that point, because one would then have a better result just by processing them sequentially. Setting the scheduler horizon is important as it can save a lot of computation time by restraining the search space, but it can also make the search miss the optimal assignment if the horizon as been set too low and the optimal assignment would require activities to be scheduled beyond that point. One can set the scheduler horizon to an `Int` value `h` by writing:

```
scheduler horizon = h
```

One can re-access the horizon at any time by calling `horizon` on the scheduler. However, one need to set the horizon once and only once. Failure to do so will result in an error message at run time informing the user of his mistake.

The `CPScheduler` is also the class the user needs to use to define what is the goal of the model, how to explore its search space, and to start the search.

For example one can write:

```
scheduler minimize makespan
scheduler search {
  ... // search method
} start()
```

Latest versions of OscaR lets the user write the objective function and search methods without having to call the scheduler at all, so one can directly write:

```
minimize makespan
search {
  ... // search method
} start()
```

This is due to the fact that `CPScheduler` extends `CPSolver`. We didn't implement those functions as they are part of the standard OscaR build. We thought it was interesting to mention them here, though, because they are critical in solving scheduling problems as one cannot solve any problem without using them.

Technical Details

As the `CPScheduler` assigns IDs to activities and resources, it needs to keep two ID maps. One for activities, and one for resources. Activities and resources can be retrieved by calling `activities` and `resources` on the scheduler. One can also request the scheduler for a given activity or resource using its `Int id` by calling `activity(id)` or `resource(id)` on the scheduler.

The `CPScheduler` overrides the `start` function of `CPSolver` to first call the `setup` function of the registered activities and resources before starting the search. These functions are described in later sections.

Activities and resources have to call `addActivity` or `addResource` on the scheduler when they are created. This function registers them on the scheduler and returns the ID that is assigned to them.

4.2.4 Activity

An activity is defined by its start time, duration and end time, so does our `Activity` class. It also has a scheduler attribute, which is the `CPScheduler` it is registered into. Those attributes can be accessed by calling their name,

for example `start` on an activity. In addition to these attributes, it has a unique id that can be accessed like any of them. It is assigned at creation when the activity is registered into the scheduler.

Quick functions exist to access the boundaries of the start, duration and end variables. Those functions are `est`, `lst`, `ect`, `lct`, `minDur` and `maxDur`, standing for earliest starting time, latest starting time, earliest completion time, latest completion time, minimum duration and maximum duration respectively. Long name forms of these methods also exists, of the form `earliestStartingTime`. The purpose of these long name forms is to be easily identified by a non-initiated user in a list of available functions on an object, like the one displayed by Eclipse when one write a dot after an object.

Technical Details

Because it is the core of all scheduling problems, `Activity` has functions to handle resources requirements. The first two are accessors and are actually usable from the DSL. But because they return `CPIntVar`s over resources IDs, they are more directed to experienced users and that's the reason why they weren't described above. The last two are not meant to be used from the DSL at all but from the feature traits, notably the `Requirements` traits. As such, the last two functions of the following list are `protected`:

- `requiredResources()` Returns a `Seq[CPIntVar]` representing the IDs of the resources required by this activity. Alternative resources requirements result in variables having multiple ID's in their domain while simple resources requirements lead to already bound CPIntVar whose domain only contain the required activity ID.
- `alternativeTo(resource: Resource)`
  Returns a `CPIntVar` whose domain is the IDs of alternative resources to the given resource.
- `registerResource(requiredResource: Resource)`
  Registers the given resource as a requirement for this activity
- `registerAlternativeResource(alternativeResource: Seq[Resource])`
  Registers the resources of the given sequence as a single alternative resource required by this activity. This is the function that generates the `CPIntVar` variables whose domain contain multiple IDs, as it contains each of the alternative resources' IDs.

We decided alternative resources belonged in activities instead of in resources because this information is related to multiple resources at once while being centered around one activity. So instead of having the information scattered in every resource that has an alternative somewhere else for some activity, we store it in the activity which is the central point of such a requirement. Resources can then ask an activity if they have alternative for that resource by using the `alternativeTo` function, but they don't actually store the information. We think it makes sense to ask an activity if it has alternatives to a resource, while we didn't think it was reasonable for resources to store informations about other resources.

To implement those four requirements functions, `Activity` uses a private `Map[Int, ResourceVar]`, which maps the required resources IDs to an instance of class `ResourceVar`. This class only has one method `id`, which returns a lazy `CPIntVar` whose domain is the IDs of the alternative resources to that resource. For example, if an activity requires resource 0 and resource 2, the map will contain the two entries, 0 and 2, and both will point to the same lazy `CPIntVar` which domain is 0 and 2. To avoid the `requiredResources` function returning multiple copies of the same `CPIntVar`, we first convert the `Map` to a `Set`, which removes duplicates. Alternative resources and how these functions are actually used will be discussed in more details later in this chapter.

The point of using a `ResourceVar` class instead of using a `CPIntVar` directly is related to the support of optional activities and is discussed, along with other functions of `Activity` regarding this subject, in the Optional Activities section of this chapter.

Activities also have a `setup` function which is called by the scheduler when it starts the search. The point of this function is to be able to delay the computation of some factors until the search starts, which means until after the user has finished modeling his problem. The standard `Activity` class only posts the consistency constraint ($start + duration = end$) in this function, but it is used more interestingly by some feature traits that extends Activity, as we will explain later in this chapter.

4.2.5 Resource

The `Resource` class in an abstract class which serves as a super-type for instances of resources. However, resources have few shared components. This is due to the fact that there are multiple different types of resources, and they don't especially have a lot in common. Reservoir resources and State resources are good examples. The abstract `Resource` class still possess some useful attributes, like a scheduler and an id field. The id is assigned at creation and accessed by calling `id` on a resource, just like activities.

Technical Details

All resources must implement the `setup()` function, as it is abstract in the `Resource` class. It is called by the scheduler when it starts searching. The point of this function is to post the constraints related to that resource. One shouldn't post any such constraint outside of this function. The reasoning behind this is twofold. First, we avoid problems like a resource constraint posted before all activities were stated as requiring that resource, as those late activities would be ignored by the constraint. Second, it helps organizing the code, having all constraints posted by a simple function is clearer than having to call multiple functions for posting multiple constraints. This way, the scheduler just has to ask the resource to setup itself and the resource handle its own constraints.

One can retrieve the activities that were stated to be scheduled on this activity by calling the `activities` function. However, as `Resource` is highly generic, it can't provide a requirement registration function, because each type of resources have different requirements formats. A cumulative resource will require a quantity, a state resource will require a specific state, and unary resources don't even need any of those, just to cite a few. We thought it was important to have a design that could easily be extended, so we anticipated very different kinds of resources.

Furthermore, by not allowing resources requirements to be called on `Resource`, DSL functions that state resources constraints need to specify the type of resource they require in argument. This way, the user will never be able to write unclear requirements statements just because some future developer was not careful enough to restrict its function argument to specific types of resource. An example of such a problem would be a function `needs` that would take a `Resource` as argument. Such function would allow the

user to write something like `activity needs cumulativeResource`, where the quantity needed of the cumulative resource is not specified. What does this constraint mean? It could mean that the activity needs 1 of the resource, as it is the case with unary resources which have the same constraint syntax. It could also mean that the activity requires the whole resource and locks it for the time it is processed on it, which is also a valid point of view with respect to unary resources behavior. Settling for either one of those interpretations is guaranteed to result in misunderstandings for some users. Thankfully, such unclear function is not possible to develop using our DSL. Because `Resource` doesn't provide the developer with a requirement registration function, he won't be able to register anything to its resource inside of his function. He will then realize its mistake and adjust his function arguments to the right resource type, avoiding such unclear constraints problem altogether.

4.3 Modeling Classes

Now that what defines a resource and an activity has been stated, let's dive into the modeling classes we are going to use.

4.3.1 Creating Activities

As stated previously, an activity has a start time, a duration and an end time. However, as the start and end times are usually unknown, only the duration is needed to create an activity. Creating an activity of duration d that one can retrieve in a variable is then as simple as writing:

```
Activity ofDuration d
```

Even if our functions take `CPIntVar` arguments, our implicits let the user write an `Int` instead. However, that doesn't prevent the creation of activities of variable durations. One can write `Activity ofDuration (2 to 5)` to achieve such result.[1] Nearly all of our DSL functions take `CPIntVar` as arguments, so it is possible to use such variable values nearly everywhere. We won't stress it for all functions however, and will warn the user when a fixed `Int` argument is specifically required.

[1] One cannot omit the parenthesis here because Scala needs to understand that `2 to 5` is a single entity to be able to convert it to a `CPIntVar`. Without parenthesis, the compiler will think the user wants to create an `Activity` of duration 2 then call a `to` function taking an `Int` on it, which indeed doesn't exist.

As the majority of problems require the creation of multiple activities, we provide the user with functions capable of creating multiple activities at once. One can either create k activities all having a duration d, k being a `Int`, by writing:

```
Activities(k) ofDuration d
```

Or one can create multiple activities by passing a sequence of durations. Suppose a sequence of durations in a variable named processingTimes of any type that can be converted to a `Seq[CPIntVar]`, one can simply write:

```
Activities ofDurations processingTimes
```

Both methods will yield a sequence of activities one can then easily iterate on. Activities created this way inherit all functions from `Activity`, as well as those from every feature traits. These will be described in the constraints functions section. We would like to stress the reader attention over the fact that `Activities` and `ofDurations` are plural forms when one create multiple activities or provide multiple durations.

It is interesting to note that we added easy accessors for activities variables over a sequence of activities, just like the ones that are created by `Activities ofDurations`. One can retrieve a sequence of variables from a sequence of activities by writing:

```
activities ids
activities starts
activities durations
activities ends
activities requiredResources
```

Those functions simply are convenient aliases for the standard `start`, `duration` and `end` variables mapped over the whole sequence.

Technical Details

The creation syntax for activities use methods defined on the `Activity` and `Activities` companion objects of class `Activity`. What they do is creating a new `Activity` while mixing it with every feature trait, enabling every feature of our DSL by default. These methods receive the scheduler this activity has to be registered into as an implicit parameter thanks to `SchedulingModel` so the user doesn't need to specify the scheduler at all.

The collection created when using one of these methods are of the class `Vector`. We first started with `Array` as a habit from Java, but ran into complications with it. The main problem was that `Array` is invariant, which means that even if `B` extends `A`, an `Array[B]` doesn't extend `Array[A]`. This posed an obvious problem for functions requiring an `Array[Activity]` while the user actually use arrays of activities with multiple traits mixed in. We decided to use a type that extended `Seq` instead, a very common collection type in Scala which doesn't have the same typing behaviour. We chose `Vector` because it is a subclass of `Seq` and it implements fast random access.

The variable sequences functions `starts`, `durations` and `ends` actually comes from an implicit class defined in the file `package.scala` of the `dsl` package. When one writes `activities starts`, the Scala compiler first realizes it can't call `starts` on a `Seq[Activity]`, then he sees an implicit class that permits such call using a `map(_.start)` and proceeds to the conversion of the sequence to the new anonymous class to handle the `starts` call.

As discussed earlier, a simple `Activity` doesn't provide any constraint method by default, that's why we use the companion objects to mix the traits in. If one would want to create a simple `Activity` of duration d with no trait mixed in, he can simply write `new Activity(d)`. This creation method can be used to build custom activities and is discussed in the Custom Activities section in the Additional Features.

4.3.2 Creating Unary Resources

One can create a unary resource to retrieve it in a variable simply by writing[2]:

```
UnaryResource()
```

Just like activities, one can also create multiple unary resources at once to retrieve a sequence of resources.

For example one can create k unary resources, k being an `Int` by writing:

```
UnaryResources(k)
```

Again we stress the reader attention on the plural form of `UnaryResources` as we are creating multiple resources at once.

[2]Scala doesn't allow parenthesis omission here, even though they are empty. The technical reason behind this is it wouldn't be able to differentiate between the object `UnaryResource` and a parenthesis-ommitted call to its `apply` method otherwise.

Technical Details

Unary resources have an `addRequirement` function that takes an activity as parameter and adds it to the `Set` of activities registered on that resource. This set is also used to implement the `activities` function of `Resource`.

We decided to model unary resources as a proper resource type, rather than as an extension of a cumulative resource with a capacity of 1. While the two resources are very similar from a modeling point of view, the constraints one can state on them, and so the functions one can call on them, are very different. We didn't consider people stating a requirement of 3 on a unaryResource as a problem, as the scheduler would simply return an obvious failure. However, we didn't want them to be able to use all the other functions we defined on cumulative resources. Calling such functions on a unary resource without knowing it is actually implemented by a cumulative resource could only lead to less obvious failures at best.

Unary resources posts the `unaryResource` constraint of OscaR when asked to `setup`, which is an optimized resource constraint for unary resources.

This behavior can be altered, however. We created a special trait for the purpose of having a common base for different implementations of unary resources constraints, the `UnaryResourceConstraint` trait. It only has a `post` method, which takes the same parameters as the `unaryResource` constraint and is meant to create a new instance of such a constraint. We then defined a function `useConstraint` on unary resources which would take an object extending the aforementioned trait. The resource will use the given object to post the constraint through its `post` method. The user can then alter the constraint posted by unary resource by giving it another object extending our trait which would post another constraint using the same parameters.

Using this intermediate object has two advantages. First, we do not modify anything in standard OscaR code, we just build our objects around the constraint classes which have no common super type other than `Constraint`. Second, the user doesn't have to instantiate a variable of type `Constraint` himself, he can simply pass the object to the function, for example:

```
resource useConstraint UnaryResourceConstraint
```

Without having to write parenthesis, dots or anything, because we defined a `UnaryResourceConstraint` object which creates a new `unaryResource` constraint when calling its `post` method.

Indeed this is of little use for unary resources, because the `unaryResource` constraint is already well optimized, but this feature will be more interesting for cumulative resources, as we will see later on. We just thought we had to do it for unary resources because we did it for the cumulative ones.

4.3.3 Creating Cumulative Resources

For this type of resource, one have to specify a capacity. Suppose having a resource capacity value in a variable called capa, one can create a cumulative resource to retrieve it in a variable by writing:

```
CumulativeResource ofCapacity capa
```

As cumulative resources have more parameters than unary ones, there also exist other functions one can use at creation. The previous statement did not put any constraint on the resource capacity itself. It can decrease down to 0 or increase, if some activities that produces resource capacity are scheduled on it, until the maximum value for a `CPIntVar`. One can create a resource having a capacity of capa and a maximum allowed capacity of maxCapa using:

```
CumulativeResource ofMaxCapacity maxCapa startingAt capa
```

Consider a lending company trying to minimize their storage building size. They need to set a maximum capacity for their stock, as they cannot store more than what their building permits to.

Similarly, one can constraint the minimal capacity minCapa a resource cannot go under by writing:

```
CumulativeResource ofMinCapacity minCapa startingAt capa
```

Consider a resource which critical capacity it cannot go under may change according to another constraint. An example would be a machine that would risk overheating under a too high usage if the warehouse temperature is beyond a certain threshold.

One can also constraint the resource capacity both ways by writing:

```
CumulativeResource ofBoundedCapacity minCapa to maxCapa
                                                    startingAt capa
```

Just like activities, one can also create multiple cumulative resources at once using plural forms. We'll only provide an example for bounded capacity however, as translating it to the other creation statements is trivial. To create k cumulative resources, k being an `Int`, one can write:

```
CumulativeResources(k) ofBoundedCapacity minCapa to maxCapa
                                                    startingAt capa
```

One can indeed create multiple different resources by providing sequences of of `CPIntVar`, or any type convertible to it like just like activities by writing:

```
CumulativeResources ofBoundedCapacities minCapas to maxCapas
                                                    startingAt capas
```

Please note than minCapas, maxCapas and capas are ordered with respect to the resources they describe, so the first resource is described by minCapas(0), maxCapas(0) and capas(0), the next one by minCapas(1), maxCapas(1) and capas(1), and so on. We admit the syntax may be a bit long to write but it is expressive and clearly states what the user is creating, there is no unclear parameter. These last two examples are the longest ones we support, though.

Technical Details

Just like unary resources, the creation is done through methods defined on the `CumulativeResource` and `CumulativeResources` companion objects of `CumulativeResource`.

As cumulative resources handle demands in addition to activities, there is a `demands` function that returns a collection of `CPIntVar` which are the demands in capacity of the registered activities in addition to the `activities` function already described in `Resource`.

Cumulative resources have an `addRequirement` function that takes an `Activity` and a `CPIntVar` demand as parameters and adds them to the `Map` of activities registered on that resource and their demands. This map is used to implement the aforementioned `activities` and `demands` functions.

Capacity constraints are posted by the `setup` method called by the scheduler using the `SweepMaxCumulative` and `SweepMinCumulative` constraints from package `oscar.cp.constraints` of OscaR. We constraint the resource usage to never exceed its capacity by posting a `SweepMaxCumulative` on a capacity of `startingCapa - minCapa`. The user states the critical minimal capacity of the resource and its starting capacity and we do the conversion to the actual capacity for him.

The maximum capacity constraint is more tricky though. As the previously mentioned resources constraints are expressed in terms of usage but we designed resources in terms of capacity, we had to convert one concept into the other. Suppose we have a current capacity of capa and a maximum allowed capacity of maxCapa. If the resource usage is 0, then the capacity is at capa. If the capacity of the resource exceeds capa, up to maxCapa, that means the usage of the resource is below zero, as there are some activities that produces some capacity while none is consumed. If the resource has a limit over which it can no longer store the additional capacity produced, we need to constraint its usage not to go lower than `capa - maxCapa`, which is actually the amount of additional capacity that is possible to store beyond the base capacity. We can then use a `SweepMinCumulative` constraint stating such limit to be sure activities won't be able to produce more than what the resource can actually store.

As discussed in `UnaryResource`, the above constraints algorithms can be altered. One can substitute the `SweepMaxCumulative` or `SweepMinCumulative` for any object that extends our `CumulativeResourceConstraint` trait and which posts another constraint when calling its `post` method. For the moment, the only other constraint which qualifies is `TTEdgeFinding`, the constraint we develop in the second part of this thesis. However, it is very easy to create a new object extending `CumulativeResourceConstraint` and to implement its `post` method. This method simply takes the same arguments as the other cumulative resources constraints and actually posts the constraint to the scheduler. One can then write, with the `TTEdgeFinding` constraint as example:

```
cumulativeResource useCapacityConstraint TTEdgeFinding
```

To alter which constraint is posted by the resource. In this case, the `post` method of the `TTEdgeFinding` object posts a `TTEdgeFinding` constraint. The `SweepMinCumulative` can also be changed using a similar function. This function is `useMinUsageConstraint`. We will discuss minimum usage in more details in the Minimum Usage section of this chapter.

4.3.4 Creating Reservoir Resources

Reservoir resources actually are non-renewable cumulative resources. When some capacity is requested, it is never given back, it is consumed forever. This will indeed affect the way activities can be scheduled on a reservoir resource, but it doesn't change anything for the creation process with respect to a standard cumulative resource. Thanks to this, one can use any creation statement from a cumulative resource for a reservoir resource. Make sure to replace `CumulativeResource` at the beginning of the statement by `ReservoirResource`, however.

Because the minimum, maximum, and current capacities concepts fit the reservoir resource extremely well and a good picture is worth a thousand words, we include a simple schema of a reservoir resource in the figure below.

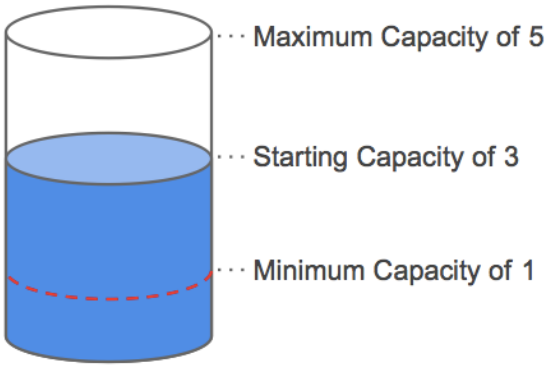


Figure 4.1: A reservoir resource

This particular reservoir resource can be created by writing:

```
ReservoirResource ofBoundedCapacity 1 to 5 startingAt 3
```

One may again imagine a variable minimum capacity. Let's assume the minimum capacity of 1 is only required when temperature is below 0, to avoid freezing if there is not enough liquid in the reservoir. One have to create a `CPIntVar` which domain is the two possible minimum capacities, that is 0 for normal temperatures and 1 for freezing ones, and then use it in the creation statement.

In practice this could be written as:

```
// Constraining minimum capacity with respect to temperature
val minCapa = CPIntVar(0, 1)
add((minCapa === 1) == (temperature <== 0))

// Creating the reservoir resource
ReservoirResource ofBoundedCapacity minCapa to 5 startingAt 3
```

Technical Details

The `ReservoirResource` class actually extends `CumulativeResource`. The only difference between one and the other is that we consider activities are scheduled from their start to the horizon for a reservoir resource. This way, their resource usage is never freed up and we can then model them just like we do it for cumulative resources.

Just like every other modeling classes, the creation is done through the methods defined on the `ReservoirResource` and `ReservoirResources` companion objects of `ReservoirResource`. Because the creation syntax is identical between `CumulativeResource` and `ReservoirResource`, we exported those functions into a `CumulativeResourceCreator` trait, which is extended by the companion objects of both cumulative and reservoir resources. Getting all those creation functions for another cumulative resource is as easy as extending `CumulativeResourceCreator`, providing a default value for the minimal and maximal capacity and a function that creates an instance of that resource given its capacity so the trait function is able to instantiate it.

4.4 Constraints Functions

Now that our activities and resources are created, we can start adding constraints to them! This section will describe the constraints one can use on activities and resources. Functions constraining activities are grouped with respect to the feature trait they are implemented into, but that's a technical detail to the DSL user as those traits are enabled by default for activities created using the `Activity` or `Activities` objects. Please note that we will refer to variables called `activities` and `resources`. Those variables are sequences of activities and resources created using plural forms of the creation statements described earlier.

4.4.1 Precedences

This section describes constraints to limit an activity possible start or end time based on another activity start or end time. We think the names of the functions are self explanatory so we are not going to describe each of them in details. One can constraint activity `a` precedence with respect to activity `b`, `a` and `b` being `Int` indices, by writing:

```
activities(a) precedes activities(b)
activities(a) endsBeforeEndOf activities(b)
activities(a) endsBeforeStartOf activities(b)
activities(a) startsBeforeEndOf activities(b)
activities(a) startsBeforeStartOf activities(b)
```

Indeed, the `precedes` function has the same effect as the `endsBeforeStartOf` function, it's simply a convenient alias.

One can also request a `CPBoolVar` which will be bound to true or false with respect to a precedence condition. The syntax is as follow:

```
activities(a) isPreceding activities(b)
activities(a) isEndingBeforeEndOf activities(b)
activities(a) isEndingBeforeStartOf activities(b)
activities(a) isStartingBeforeEndOf activities(b)
activities(a) isStartingBeforeStartOf activities(b)
```

It is very interesting because one can construct logical expressions based on `CPBoolVar`, but as this is standard OscaR behavior, we are not describing it. An example will be given when modeling the Aircraft Landing problem.

Technical Details

These functions are defined in a trait called `Precedences` which extends `Activity`. By designing them in such a trait, one can use them as it is with his own implementation of an activity, as long as it extends `Activity`, by mixing it in. This possibility will be explained in more details in the Custom Activities section of this chapter.

In terms of constraints, those functions simply translate what their name say into actual constraints on the activity variables. For example when one writes the following statement:

```
activities(a) endsBeforeStartOf activities(b)
```

It is equivalent to posting:

```
activities(a).end <= activities(b).start
```

And that's exactly what the function actually posts. This allows the user to write easily understandable constraints instead of manipulating problem variables directly, which is exactly the point of this DSL.

4.4.2 Separation Times

When one says activity a precedes activity b, it means that activity a must be processed before activity b starts. One can straighten this statement by constraining the time that can elapse between those activities, for example with `d` being a delay as a `CPIntVar`, one can write:

```
activities(a) startsAtLeast d beforeStartOf activities(b)
activities(a) startsAtLeast d beforeEndOf activities(b)
activities(a) startsAtMost d beforeStartOf activities(b)
activities(a) startsAtMost d beforeEndOf activities(b)
```

It is indeed possible to state this kind of constraints on the end variable as well. For example:

```
activities(a) endsAtLeast d beforeStartOf activities(b)
```

We aren't detailing all variations of those functions for the end variable as they are extremely similar to the ones for start.

One can constraint the delay even more by writing:

```
activities(a) starts d beforeStartOf activities(b)
activities(a) starts d beforeEndOf activities(b)
activities(a) ends d beforeStartOf activities(b)
activities(a) ends d beforeEndOf activities(b)
```

Technical Details

These functions are defined in the `SeparationTimes` trait which extends `Activity`.

Because of the high redundancy of the `beforeStartOf` and `beforeEndOf` functions, we use an intermediate `SeparationDelay` class which is invisible from the standard user. All the `starsAtLeast`, `endsAtMost`, `ends` and so on functions create a `SeparationDelay` class by passing it the variable to constrain and the constraint to apply to it. For example `startsAtLeast` creates a `SeparationDelay(start, (x: CPIntVar , y: CPIntVar) => (x <= y))` which then have the `beforeStartOf` and `beforeEndOf` functions to receive the second operand of the constraint to finally post it.

4.4.3 Synchronizations

One can also want to constraint the delay to be exactly zero, so that activity `b` starts right when activity `a` ends, thus synchronizing activity `a` start with activity `b` end. Such constraints can be expressed like this:

```
activities(a) startsAtStartOf activities(b)
activities(a) startsAtEndOf activities(b)
activities(a) endsAtStartOf activities(b)
activities(a) endsAtEndOf activities(b)
```

Technical Details

These functions are defined in the `Synchronizations` trait of the `dsl` package and extends `Activity`. They are pretty simple though, they simply post the constraint that the first activity's variable must be equal to the second activity's one. We simply put english words on numerical constraints.

4.4.4 Time Windows

It is also possible to constrain an activity start or end scheduling in time without referring to an other activity. Such constraint defines a time window during which the activity must start or end. One can constrain activity `a` in a time window between `min` and `max`, both being `CPIntVar` or types convertibles to it, by using:

```
activities(a) startsBetween min and max
activities(a) endsBetween min and max
```

Never forget that `CPIntVar` are problem variables, which means that these time windows are capable of more than fixed `Int` limits as it may be variable depending on other constraints.

Because sometimes one may want to use a time window of size 0 by saying the activity needs to start or end between a value `t` and the same value `t`, we added a shortcut for this:

```
activities(a) startsAt t
activities(a) endsAt t
```

Technical Details

Following on the pattern used by all other traits, these functions are defined in the `TimeWindows` trait extending `Activity` and translates what the english words means into actual numerical constraints on the activities variables.

4.4.5 Resources Requirements

Resources requirements are at the core of scheduling problems. The user will definitely use the functions we are going to describe in one way or another in whatever problem he will have to model. Like activities in the previous descriptions, `unaryResources`, `cumulativeResources` and `reservoirResources` are sequences of resources created using the plural form of their respective resource creation statements. The variables `a`, `r` and `s` are all `Int` indices used to access elements of the aforementioned sequences.

Let's start with unary resources requirements. Stating that activity `a` needs unary resource `r` is as easy as writing:

```
activities(a) needs unaryResources(r)
```

Cumulative resources are not very different from unary resources in terms of the syntax of their requirements function. The main difference is that one has to specify how much of the cumulative resource's capacity the activity needs. Stating that activity `a` needs a quantity `q` of the capacity of cumulative resource `r`, with `q` being a `CPIntVar` or any type convertible to it, is as easy as writing:

```
activities(a) needs q of cumulativeResources(r)
```

With cumulative resources, it is possible to have an activity that actually produces some capacity usable by the resource.

One can state that activity `a` gives `d` of capacity to cumulative resource `r` by writing:

```
activitites(a) gives q of cumulativeResources(r)
```

One could indeed express the same constraint using the `needs` function and a negative demand, but it's more straightforward for us humans to write `gives d` rather than `needs -d`.

The exact same syntax can be used for reservoir resources, so one can state activity `a` needs `q` of reservoir resource `r` by writing:

```
activities(a) needs q of reservoirResources(r)
```

The same goes for the `gives` statement, as reservoir resources basically are cumulative resources. They simply behave differently regarding to how they post their capacity constraint.

Technical Details

These functions are defined in a trait called ResourcesRequirements which extends `Activity`, allowing the experienced user to mix it with any class extending it.

This trait makes use of the `protected registerResource` requirement function of `Activity` as well as those of the different resources types. Calling a function from this trait makes it call the right requirements functions on the right activities and resources.

Reservoir resources are indeed usable in any place where a cumulative resource is expected, as the former extends the latter.

Attentive readers aleo reading the source code may have noticed that this trait not only extends `Activity` but also mixes `Requirements` in. The only method of the `Requirements` trait is a `toString` method which outputs the resources required by an activity. This toString function is meant to be shared between the two requirements traits, `ResourcesRequirements` and `AlternativeRequirements`, that we will discuss below. They are both supposed to call the `toString` method of their superclass, thus resulting in a single call to `Requirements`'s `toString` method when both traits are mixed in. If we didn't design the `toString` method this way, both traits would have outputted the resources from their own implementation of `toString`, which would have been hard to design in terms of wording of the output, and less readable in the end anyway.

4.4.6 Alternative Requirements

What if an activity needs resource `r` *or* resource `s` ? One can state such alternative resources requirements by writing *or* between the resources the activity alternatively requires. So suppose activity `a` either needs unary resource `r` or unary resource `s`, both being `Int` indices, one can write[3]:

```
activities(a) needs (unaryResources(r) or unaryResources(s))
```

But that's quite redundant and will quickly become a nightmare when requiring more than two resources using such syntax. If the resources the activity require are accessible from the same sequence, which would be the case if they were created using the plural forms of our creation statements, one can more directly write:

```
activities(a) needs unaryResources(r or s)
```

Which can then easily be expanded for more resources:

```
activities(a) needs unaryResources(r1 or r2 or r3 or r4)
```

It is very straightforward but requires that all alternatively required resources be in the same sequence to begin with. If they don't, using the longer form is required. It is indeed also possible to alternatively require more than two resources with that form.

The statements are not very different for cumulative resources, one just need to write *or* between the required quantities as well as between the resources they are required from, respectively. So suppose activity `a` either needs `q1` of resource `r1` or `q2` of resource `r2`, one can write:

```
activities(a) needs (q1 or q2) of (cumulativeResources(r1) or
                                   cumulativeResources(r2))
```

Or more directly if the cumulative resources are part of the same sequence:

```
activities(a) needs (q1 or q2) of cumulativeResources(r1 or r2)
```

Creating sequences of resources using the plural forms of their creation statements really empowers the user, allowing him to use the short forms of these functions and simplifying iterations on resources. We strongly recommend them to the user.

[3]One cannot omit parenthesis for this statement because Scala won't consider the two resources as one unique argument otherwise.

Please note however that the more verbose form will also allow the user to state alternative requirements between cumulative resources and reservoir resources alike, while the short form doesn't. Because it requires the resources to be in the same sequence, one won't be able to use it directly if having a sequence of cumulative resources and another sequence of reservoir resources. It would be possible if the user appends one sequence of resources to the other, though, but he would then be vulnerable to indexing errors when accessing the resources of such a sequence.

Don't forget to tell the scheduler to actually decide on which resource the activity will be scheduled, though! It won't try to bound the required resources variables if it is used nowhere in the search. One can access the variable to be bound for this alternative resource by using:

```
activity.alternativeTo(resource)
```

Where `resource` is one of the resources that are alternatively required. The more direct way of forcing the scheduler to decide the resource on which to schedule the activity is to use the result of the `requiredResources` function called on an `Activity` or a `Seq[Activity]` in the scheduler search. It has the same effect as calling `alternativeTo` for all resources.

Technical Details

These functions are defined in the `AlternativeRequirements` trait which extends `Activity` and `Requirements`, as discussed in the previous Technical Details section.

We think it is interesting to explain how one can write `or` in between resources and give it as an argument. We have defined an `Alternative[T]` class which extends `Seq[T]`. This class defines the `or` function, which takes a `T` and returns the `Alternative` with the new element appended. Alternative requirements functions expect to receive `Alternative[UnaryResource]`, `Alternative[CPIntVar]` or `Alternative[CumulativeResource]`, depending on the requirement. There are implicit conversions in the `package.scala` file of the `dsl` package that converts variables and resources to `Alternative` so one can call `or` on it. The `or` method can be used multiple time, thus being able to state as many alternatives as required. The more compact form use our Enhanced Selectors feature, which is described later in this chapter, where indices are of the form `Alternative[Int]`.

To model alternative resources as constraints, we inspired ourselves from the `resources` argument of `SweepMaxCumulative`. It is a `Seq[CPIntVar]` which elements are variables which domain are resources IDs. It is used by the constraint to model an activity that is not to be scheduled on the resource. For example, an activity present in a resource `SweepMaxCumulative` constraint but having its `resources` element variable bound to a different value than the resource's ID will be ignored by the constraint because the activity is actually *not* scheduled on this resource. This trait uses the `registerAlternativeResource` of `Activity`, which actually creates such variables, using the alternative resources IDs as domain. The resources then query `Activity` for those variables using the `alternativeTo` method described earlier, which are then used in the constraint.

4.4.7 Optional Activities

This trait handles the support for optional activities. From the end user point of view, the only think to write to state that an activity is optional is:

```
activity optional = true
```

After this statement, the scheduler will know it can also choose not to schedule the activity on any resource. However, it needs an incentive to do so. Problems involving optional activities usually issue some cost or drawback for not scheduling activities. To model such constraints, we designed the following function:

```
activity ifScheduled CPIntVar ifNot CPIntVar
```

This function returns a `CPIntVar` which can take the values of the two other `CPIntVar`s used as arguments. The resulting variable will take the value of the first one if the activity is scheduled, and the second one if it is not. Here is a simple example that uses `Int` values:

```
activity ifScheduled 10 ifNot 0
```

If the activity is scheduled, the resulting variable will be bound to 10, if it is not, it will be bound to 0. Such construction is useful to model the profit of scheduling an activity or the cost of not scheduling it. One can then use multiple variables like this one, and tell the scheduler to maximize or minimize their sum as the objective function, depending on the problem.

One can also retrieve the `CPBoolVar` representing if an activity is scheduled or not by accessing the `isScheduled` variable of the activity. This variable is used in the previous function implementation.

Technical Details

This trait may seem very simple from a user standpoint, as it only defines two functions, of which one is a setter method. However, it is the most complicated of all traits. The reason why it is so complicated is because it has to modify the behavior of a standard activity. The `Activity` class doesn't care about being optional, it completely ignores the concept. The `Optional` trait must then override `Activity`'s behavior but with the least code duplication possible.

We designed `Optional` functions as hooks for `Activity` ones. We would receive the call first, because of how overrides works with traits, modify the input to fit our needs and then pass the modified arguments to the superclass function. We actually use a kind of fake resource to model optional activities. We create an additional resource with no constraint on it whatsoever and hook the resource registration functions of `Activity` to add this new resource as an alternative. We then constrain the `isScheduled` variable to be `false` if the activity is scheduled on this additional resource.

For the `ifScheduled/ifNot` construction, we create a new `CPIntVar` which domain is the union of the two `CPIntVar` arguments, and constrain that `isScheduled` being true implies that the new variable takes the value of the first one, and false the second one. It is important to use an implication and not an equivalence because the two variable domains may overlap. If the variables are bound to the value they have in common, it would mean that the activity is scheduled and not scheduled at the same time, which would indeed make the constraints store fail. The intended result is for the scheduler to be indifferent whether to schedule the activity or not with respect to that value alone.

The `Optional` trait makes use of the `setup` function of `Activity` which was introduced earlier. Because we need to constrain `isScheduled` to be false if any of the activity resources variables are bound to our additional activity, we need `Activity` to actually create such variables. Which happens is we can't simply add our additional resource as a requirement for the activity without thinking much more, because at the point of resources requirement statements, we cannot be sure if the activity will be stated as optional or not by the user, as he may well do it after having stated the requirements! So what is really sent to the `Activity` requirement function is a block with a condition, which adds the additional resource only if the activity is optional. This block is not evaluated thanks to the `ResourceVar` class described ear-

lier and will be evaluated once when the resource variable is requested by a resource to post its constraint, which means when the scheduler calls the setup function. We cannot bound `isScheduled` before the model is finished, because we cannot be sure the user won't state the activity is optional before that point.

Something else that needs to be done for optional activities is tricking the scheduler to not consider them when evaluating the makespan. Because we model optional activities as activities scheduled on an additional, unconstrained resource, they are still part of the model, even "unscheduled", and will still be taken into account by the scheduler. So if an activity of duration 4 is scheduled and an activity of duration 10 is not scheduled, the scheduler will think the makespan is 10, because it doesn't know about the `Optional` trait and our additional "fake" resource. To overcome this problem, we redefine `end` for an optional activity to be equal to 0. By doing so, we also need to change either the start, the duration or both, otherwise the store will fail when adding `Activity` consistency constraint stating that the end of an activity should be equals to the sum of its start and its duration. We originally intended start and duration to be 0 for an optional non-scheduled activity, but we discovered some constraints had problems handling activities of duration 0. We then decided to "hide" the non-scheduled activities below the time 0, so we override `start` to be `-start` and `end` to be 0 so the consistency constraint still holds, we simply shift the activity away from the makespan computation.

4.4.8 Due Date

This feature trait lets the user define a due date for an activity and easily retrieve its earliness and tardiness with respect to that due date. It is very easy to set a due date `d` for activity `a`, one can simply write:

```
activities(a) isDueAt d
```

Sometimes an activity can have a minimal due date and a maximal due date which are different. The activity is too early if it finishes before the minimal due date, too late if it finished after the maximal due date, and alright if it finishes anywhere in between the two, inclusively. The idea is that there exists a time window in which the activity is preferred to be scheduled, but it is not a strict constraint imposed on the scheduler. The user is then supposed to be interested in the earliness and/or tardiness of the activity, which is usually what the problem tries to minimize when involving

due dates. One can state such a due time window between `d1` and `d2` by writing:

```
activities(a) isDueBetween d1 and d2
```

As usual, the due dates depicted here are all `CPIntVar` or any type convertible to it.

One can then retrieve activity `a` earliness and tardiness just like any other activity variable:

```
activities(a) earliness
activities(a) tardiness
```

Technical Details

Like the other activities features traits, these functions are defined in a trait called `DueDate` which extends `Activity`, allowing the experienced user to mix it with any class extending `Activity`.

We struggled a lot on this feature because of the `subjectTo` function of the `CPSolver` which has been removed by now. The problem was that the `CPSolver`, and the `CPScheduler` by extension, required a `subjectTo` function to be called on it before being able to start solving a problem. This method received a block of functions which it executed when called, and did some more work on the solver side to consider the model closed. The problem is that such a `subjectTo` block was a kind of bait for constraints. By the look of the function, one would like to put his constraints only in the `subjectTo` block because it looked like it was the only place where they were registered, judging by the name of the function, even though this was not true in practice. Because of this, the user had to state the objective function of his model before stating the constraints, like this:

```
// Variables definitions
minimize(objective) subjectTo {
    // Constraints
}
```

But if the objective to be minimized revolves around the earliness or tardiness of an activity, what should be the return of these functions when the due date has yet to be defined, as the user would likely define it in the `subjectTo` block? If we return `CPIntVar` bound to 0 when the due date is not defined, then we block the variable entirely and the due date defined

later will be ignored or will trigger a failure of the `CPStore` when later adding the constraint, depending on the implementation. If we return an unbound variable and constrain it when the due date is defined, we then require the user to set a due date for every activity from which he requested the earliness or tardiness. We think the user would expect the earliness and tardiness of activities for which he *doesn't* state a due date to be 0, however that wouldn't be the case with the proposed implementation.

To solve this problem, we came up with the `setup` method of `Activity` which was introduced earlier, and the `OptionalVar` class. This class behavior is something in between a `lazy val` and an `Option[CPIntVar]`. We need the `setup` method to be able to bound the earliness and tardiness variables of activities when we are sure the user as finished modeling his problem, that is right before the scheduler starts the search. However, we only need to do it if the user required the earliness or tardiness variable to begin with, otherwise it is not a problem that he didn't set a due date. Because feature traits like `DueDate` cannot create variables unless their features are required, all variables must have a lazy behavior at least. However, we have no means to know in the `setup` method if the user actually required any of earliness or tardiness because trying to evaluate its value would actually create the lazy variable. This could be solved using two private variables which would default to false and be set to true when the user would require earliness or tardiness respectively. However, that would require to create one boolean variable per problem variables affected by this issue, and that also applies to possible future feature traits. To allow future developers to easily overcome this issue, we designed the `OptionalVar` class, which is a wrapper for `CPVar` and is implemented using an `Option[V <: CPVar]`. What it does is wrapping any `CPVar`, without creating it thanks by-name parameters, into a class on which one can ask easily ask if the variable has been required or not, which is already better than a lazy val for that matter. However, that behavior alone could also have been obtained using a single `var earliness: Option[CPIntVar] = None`. We can ask `isDefined` and we can erase None with the real value when needed. However, care must then be taken when developing with this design because nothing prevents changing the value of the variable *twice*, if the user calls some function more than once for example, losing the reference to the previously stored variable. Our `OptionalVar` class does not have this problem, because when one require the variable using the `require` function, its value is evaluated, stored in the class once, and returned. It's a kind of `getOrSet` method for `Option`, which returns the value if it set or set it once and for all if it was not already and returns it. This is safe, easy to use, and solves our problem completely.

We really think it's an interesting tool when developing feature traits, so we made it accessible for power users wanting to develop their own trait by putting it in the `dsl` package.

4.4.9 Resources Usage

Activities already had a lot of constraints described, but resources also have some. For a `CumulativeResource`, one can state a minimum usage for that resource. Suppose a resource that is so costly to run that one doesn't want to run it under 80% capacity usage. With `u` being the minimum usage to run the resource `r`, one can state that if resource `r` is used by any activity, then its capacity must be at least used by a quantity of `u` by writing:

```
cumulativeResources(r) hasMinUsageOf u
```

This constraint will require that `u` capacity is to be required from the resource if *any* capacity is required to begin with. Those functions indeed only apply to cumulative and reservoir resources, as unary resources don't even have a capacity.

Technical Details

In practice, the maximum capacity of `CumulativeResource` is already defining a minimum usage, as discussed earlier, of `startingCapa - maxCapa`. When stating a new minimum usage, the higher one will prevail when posting the constraint.

The `SweepMinCumulative` constraint of OscaR exactly matches the definition of function `hasMinUsageOf`. It constraints the resource usage to be at least of a certain amount, but only during the time the resource is actually used. If no activity requires the resources at a certain time, then the usage will be 0. It can either be 0 or at least the minimum usage.

Earlier versions of `CumulativeResource` had a variant of this minimum usage constraint, which imposed the minimum capacity to be respected from the beginning to the end of the problem. We implemented that behavior by adding a fake activity which spanned the whole horizon, simulating an all time resource requirement to trigger `SweepMinCumulative`. However, we decided to remove it. The reason behind this decision is that while the constraint was nice, we also wanted to provide the user with another variant, which constrained such strict usage between the start of the first activity on that resource and the end of the last one. Three versions of minimum

usage functions seemed like we were going too far into specific constraints, like we were trying to provide each problem with a single function doing all the needed modeling, but then being limited to this sole problem scope. We decided to only provide the weakest version of the constrained as explained in the description of this section, and let the user straighten it if needed.

4.5 Additional Features

This section introduces additional features we thought interesting to add to our DSL. These features weren't strictly required as of the problem analysis of the previous chapter, but we thought they significantly improved the DSL experience.

4.5.1 Reader

This feature is in the additional features because it does not posts any constraints. However, it is critical for the user. The `Reader` trait gives access to methods for parsing an input file. If one want to use this feature, he needs to mix the `Reader` trait with the model. Considering the `SchedulingModel` trait, one then needs to write something like:

```
object MyModel extends SchedulingModel with Reader
```

Doing so gives access to the `read` variable, which can be used to read files. The first thing one wants to do is define the file to read. This can be done by calling the `fromFile` method and giving it a filepath. For example:

```
read fromFile "path/to/file"
```

The input files described in the multiple papers we referenced to in the Problem Analysis section all had important values on the first line, like the number of activities or the number of resources. One can read `k Int` values, `k` itself being an `Int`, by writing:

```
read fileFor k as Int
```

Thanks to Scala pattern matching feature, one can define variables directly when reading the file, for example for two variables to read, one called `nJobs` and the other `nResources`:

```
val Seq(nJobs, nResources) = read fileFor 2 as Int
```

Because this method returns a `Seq[Int]`, one needs to write `Seq` in front of the variables for pattern matching to succeed.

One can also read entire columns and lines from the input file. For example one can write:

```
val Seq(starts, ends) = read fileFor nbTasks x 2 asColumnsOf Int
```

To read a table of size `nbTasks x 2` and receive it indexed by columns, so 2 columns, and name each of them at reading time thanks to pattern matching. One can also retrieve the table indexed by lines by using `asLinesOf` instead of `asColumnsOf`. If one doesn't want to to convert the values to `Int` or `Double`, the two types we support conversions to, he may retrieve the default `String` result by using `asLines` or `asColumns`.

Sometimes however, the file is structured so strangely with respect to line returns, that neither reading it by lines or columns will suffice. For this type of files, we added the following functions:

```
read fileFor i unitsOf j asLinesOf Int
read fileFor i unitsOf j asColumnsOf Int
```

Using `unitsOf` actually tells the parser to ignore line returns. It will read the file until it has parsed `i` units each consisting of `j` values. One can then retrieve the result indexed by lines or by columns, resulting in either `i` units or `j` columns of the units elements.

Something that is really important to know is that our reader is designed to read only what the user asks it to read. Elements that were not returned to the user are put back in the file for further reading. This is necessary to support reading by columns, where Scala files are normally read by lines. Doing so allows the user to write:

```
val Seq(col1, col2, col3) = read fileFor 3 columnsOf Int
```

And still retain the next columns in the file, maybe to read them as `Double`, or because they have a strange structure. A good example of such structure is one we found in RCPSP files. One of the columns indicates how much elements one have to read on each line. So the file have different line lengths and one cannot know the length before parsing it. To handle such files, we allowed the user to use a `Seq[Int]` to indicate the number of columns to read for each line, instead of a fixed `Int` applying to each line.

One can then read such a file, where `nbTasks` is the number of lines to read by writing:

```
val Seq(col1, col2, remainingElements) = read fileFor 3 columnsOf Int
read fileFor nbTasks x remainingElements asLinesOf Int
```

One cannot, however, request the result as columns in this case because they are not regular.

One can also drop parts of the file using:

```
read dropping k lines
read dropping k columns
read dropping k words
```

The first two dropping functions are obvious. The third one is actually useful to read more descriptive files which contains lines like "Jobs = 40" instead of simply "40".

A big improvement came late in the development of the `Reader` trait. We added the possibility to read the file according to given patterns. For example, the three following statements are equivalent ways of reading 3 columns, although the second and fourth ones will not stop at `nbTasks` lines:

```
read fileFor nbTasks x 3 asColumnsOf Int
read fileFor 3 columnsOf Int
read fileFor nbTasks withPattern (Int, Int, Int)
read fileFor (columnOf(Int), columnOf(Int), columnOf(Int))
```

The second version is the most powerful one because it lets the user write any of the above statements as part of a pattern. So one can for example write:

```
read fileFor nbTasks withPattern (Int, Int, Double, 3 unitsOf 5
                                                  asColumnsOf Int)
```

For each of the `nbTasks` elements, the file will be read for two `Int`s, one `Double` and a table of 15 `Int`s ignoring line returns. All the user has to do is tell the reader how to parse one element of information from the file, and how much of those elements it needs to read. This construction alone suffice to read nearly every file we found. Exceptions are the one where we do not know how much lines there is to read, where it is required to use `read k columnsOf` or the last statement of the four equivalent ones above.

Because these functions use tuples instead of `Seq`, one can write:

```
val (jobs, requiredResources, durations) = read fileFor nbTasks
                                    withPattern (Int, Int, Int)
```

Without needing to write `Seq` in front of the variable definitions. Because each time the reader reads one elements it reads 3 `Int`s, the result is three columns of `Int`.

The `withPattern` function also supports additional patterns like `Int(k)`, which will read `k Int` but output a matrix with `k` columns retrievable in a single variables instead of `k` columns which would then need `k` variables. Another additional pattern is `dependsOnNext`, that we had to define for the RCPSP file. This pattern reads the next `Int` and uses its value to know how much other values it needs to read.

Patterns support came very late in the development of the `Reader`, so we still support the more outdated way of reading the file which was described first in this section. However, we think the pattern-based version is easier to use and more powerful. We were actually able to parse all our files using this pattern-based approach.

Additionally, the `dependsOnNext` pattern example prove the endless possibilities of designing any type of strange patterns to read certain files. We are convinced that the pattern-based approach is a good one and that it should definitely be expanded on, even outside of the context of this thesis. File parsing is often a tedious task, and such pattern-based approach could prove very useful in overcoming it.

Technical Details

To be able to read entire columns of the file while Scala only supports to read lines, we design the `DataFile` class. It can be found in the package `oscar.cp.memScheduling.parser`. The `DataFile` class actually extends an `Iterator[Seq[String]]`, in the sense that it outputs lines of `String` split into separate elements on a given regexp. What is different from a standard `Iterator` though, is that it has a `putBack` method. When we want to read a column of `k` elements, we read `k` lines, we then store the values of the first element of each line in a variable, `putBack` the rest of each line, and return the stored elements.

The `DataFile` is actually implemented using the real `Iterator[String]` of the file, and a `Stack[Seq[String]]` which holds the lines we put back. When calling `hasNext` or `next` on the `DataFile`, it will first try to empty the `Stack` before handing us a brand new line of the file.

Because we have to provide a return type for the `withPattern` function, we had to use tuples. If we used `Seq`, we'd lose the underlying type of each resulting element. because `Seq` is not meant to contain elements of different types. However, because of the tuples implementation in Scala, we do not support patterns consisting of more than 7 elements. The limitations of `Seq` versus the ones of tuples are discussed in more details in the Technical Details section of the Enhanced Tuples feature.

4.5.2 Visualization

The `Visualization` trait is a model-level trait like `SchedulingModel` or `Reader`, which means it needs to be mixed in with the user model object to be used. When mixed-in, one have access to the `visualization` object, which has a single method `show`, which accepts 4 different types of arguments, for 4 different types of elements to display.

First of all, one can display activities ordered and colored with respect to some value. For example, if one has a sequence of activities and a sequence of jobs that are `Int` values representing the job to which an activity belongs, one can write:

```
visualization show activities by jobs
```

To display the activities ordered and colored with respect to their job on a Gantt chart, resulting in a window similar to the one on figure 4.2.

If no coloring factor exists other than the activities themselves, one can color the activities with respect to their IDs, which will color each activity differently, by writing:

```
visualization show activities by activities.ids
```

Using the `ids` method defined over a `Seq[Activity]` by an implicit method in `package.scala`, to get a window the like of the previous one, albeit with each activity separated and colored differently.

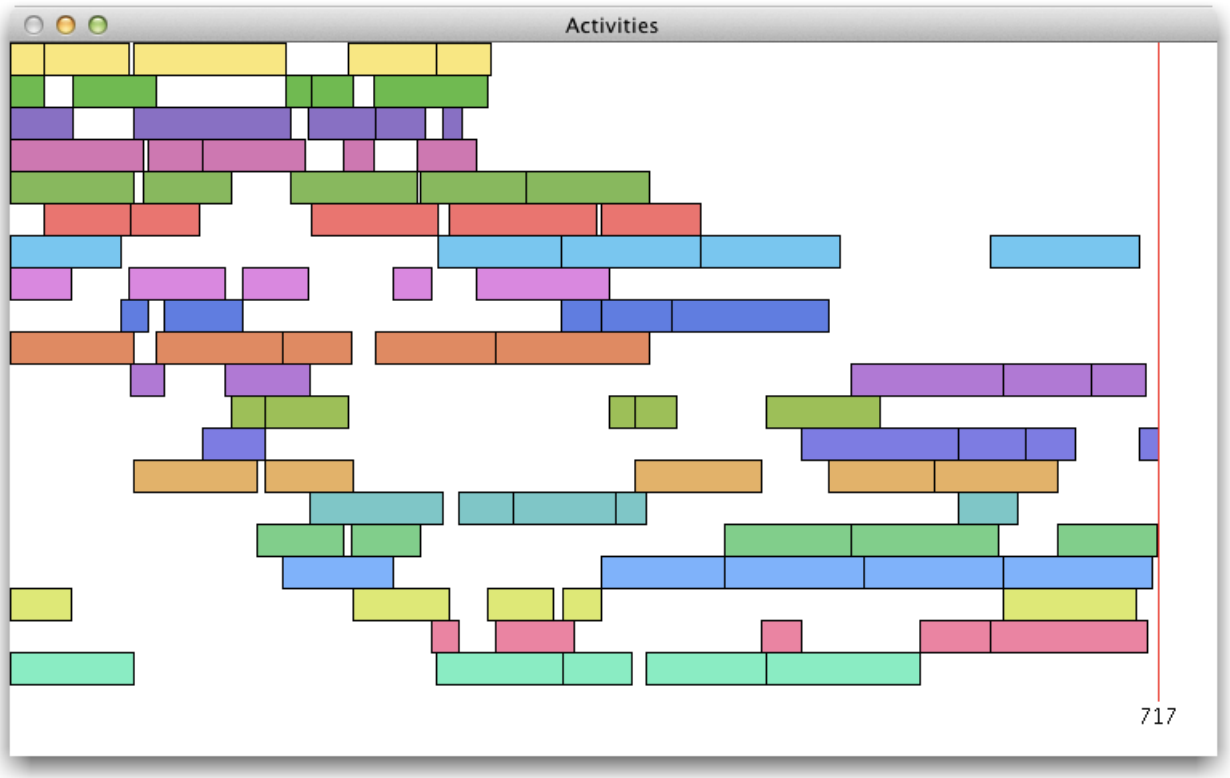


Figure 4.2: Activities colored with respect to their job

One can also display the activities scheduled on a sequence of unary resources by writing:

```
visualization show unaryResources
```

Which will result in a Gantt chart like the one on figure 4.3.

It is very interesting to give the `show` method a sequence of unary resources to display instead of calling `show` for each unary resource, because doing so results in a Gantt chart very similar to the one of activities. One can then see the activities colored with respect to their jobs on the resources chart, which is very useful. One can have the exact opposite coloring, that is having the activities in the jobs chart colored with respect to their resources by first calling `show` on the resources then on the jobs, as the first coloring prevails on the next ones. An example is shown in figure 4.4.

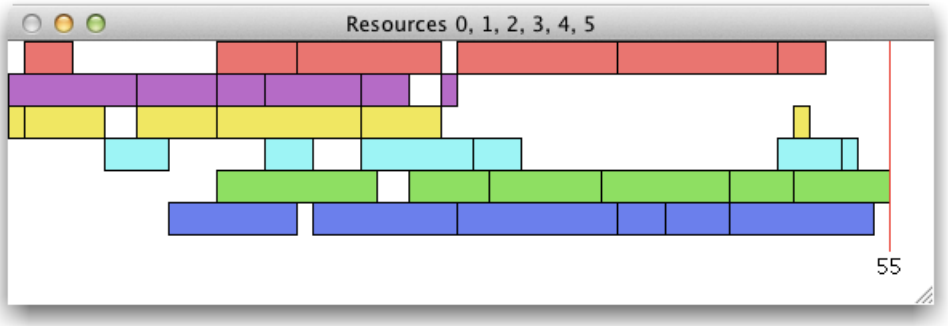


Figure 4.3: Unary Resources

One can also set the title of a visual by adding `withTitle` at the end of the visualization statement. Here is a complete example for unary resources and jobs:

```
visualization show machines withTitle "Machines"
visualization show activities by jobs withTitle "Jobs"
```

Which would result in a visual like the one on figure 4.4.

It is also possible to modify the main display windows title using:

```
visualization withTitle "Main Title"
```

Indeed, one can also visualize cumulative resources usage, the syntax being identical:

```
visualization show cumulativeResources
```

The cumulative resources displays capacity usage as a function of time like one can see on figure 4.5. Because of the nature of this graph, calling `show` on a sequence of cumulative resources have the same result as calling it on each one separately.

The last type of arguments the `show` method accepts is a simple `Int`. It draws a vertical line on all displays at the specified time. For example to display a due date of 92 as a vertical line on the visuals, one can write:

```
visualization show 92
```

Which will add a vertical line on all graphs like the one on figure 4.6.

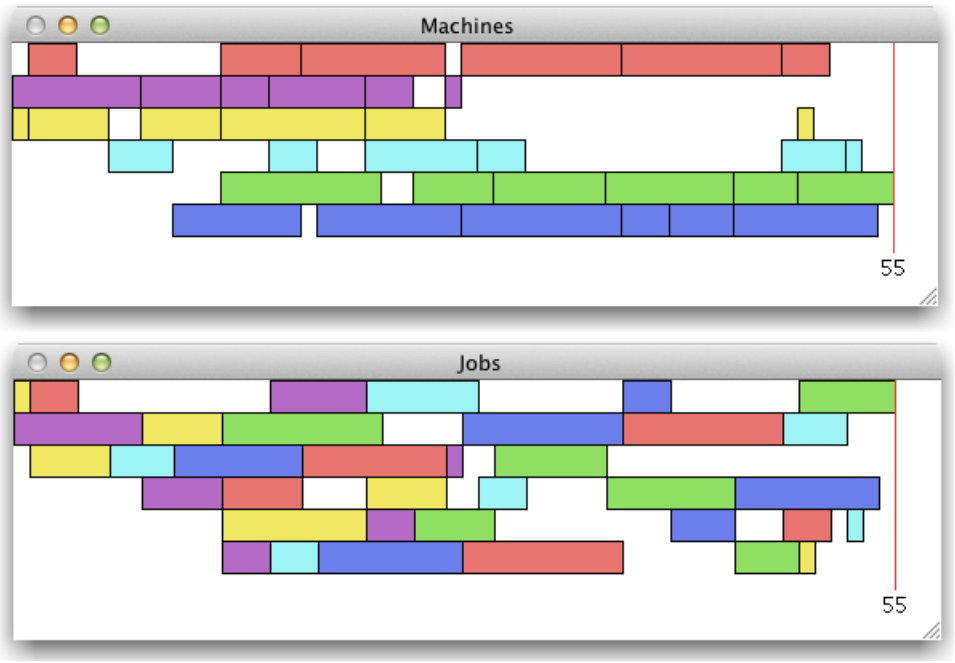


Figure 4.4: Unary Resources and jobs

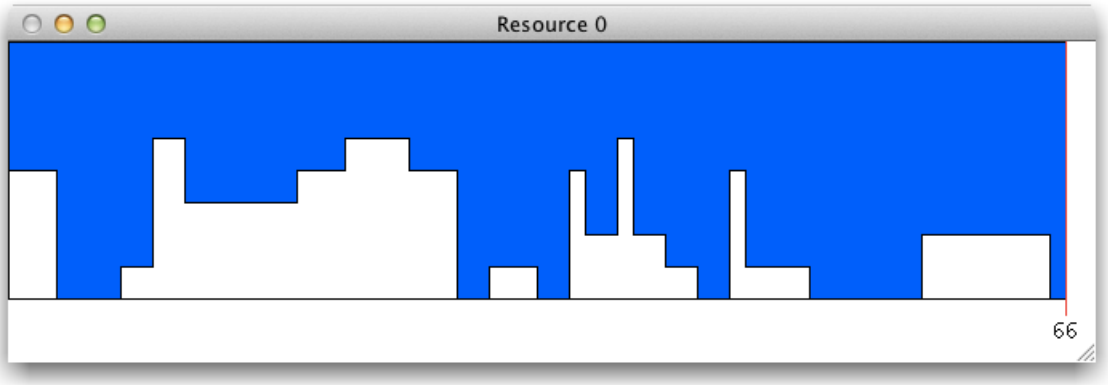


Figure 4.5: Cumulative Resource

One can zoom the visuals to a `Int` value `z` by writing:

```
visualization zoomedBy z
```

The attentive reader may have noticed we used a zoom of 10 for the visuals involving resources, as the windows are the same scale as the Activities example, but the makespan value is ten times lower.

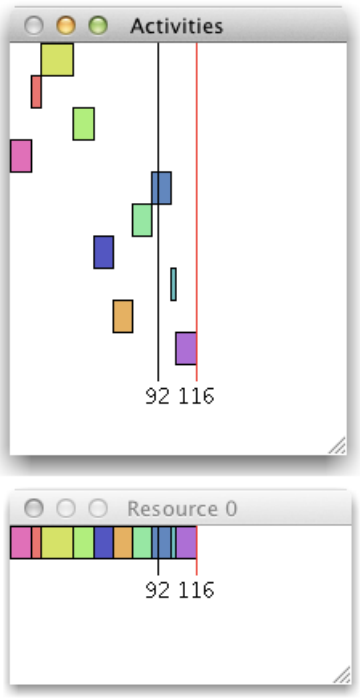


Figure 4.6: A due date displayed as a vertical line

Technical Details

Updating those visuals required to trigger some update code when a solution was found. Hopefully, recent versions of OscaR `CPSolver` have an handy `onSolution` method that receives a block of code and executes it each time a solution is found, that's what we used to trigger the visuals update methods.

Please note that we didn't implement those visuals. What we did is integrating them with our DSL trait-based design and providing easy functions to display them. The Visualization feature was however a low-priority one on our development schedule, so it might still have some rough edges. An example of the sort is the size of the windows. The visualization windows will span the whole screen and resize themselves with respect to the number of windows. So if one requests 5 visualization, each of them will use a fifth of the screen, which will hinder the ability to read any them. One can resize the windows manually though, so it's not a major problem. That's simply something that could surely be done automatically but we chose to concentrate on developing other features instead.

4.5.3 Enhanced Tuples

For those unfamiliar with Scala, one can iterate over multiple sequences, let's say a sequence of activities and a sequence of resources, by writing:

```
for ((a, r) <- (activities, resources).zipped) {
  ...
}
```

This construction allows the user to receive the first element of the first sequence in `a` and the first of the second one in `r`, then the second of both sequences at the next iteration and so on. The problem with this statement, in addition to being forced to call `zipped`, which we thought isn't really obvious, is that one cannot iterate over more than three sequences using such construction. We decided to write implicit functions that would let the user zip up to 5 sequences without even writing `.zipped`, so he could write:

```
for ((a, b, c, d, e) <- (seq1, seq2, seq3, seq4, seq5)) {
  ...
}
```

Which is not possible to write in standard Scala, even using `zipped`. We thought the syntax was already lengthy enough with 5 elements to not allow zipping with more than 5 elements, though. There is actually a limitation in how Scala handles tuples that prevented our implicit functions to work for any size of tuples, as we will discuss in the Technical Details section below.

For power users interested in using the full possibilities of Enhanced Tuples, we also implemented `Seq` functions `filter`, `foreach` and `map`, so, a `Tuple2[Seq[A], Seq[B]]` will react as a `Seq[Tuple2[A, B]]` for example. Consider the following code snippet:

```
val seqA = Seq(1, 2, 3)
val seqB = Seq(1.5, 2.5, 3.5)
(seqA, seqB) foreach (t => println(t._1, t._2))
```

Would have the `Tuple2[Seq[A], Seq[B]]` behave like a `Seq[Tuple2[A, B]]` and output:

```
1 1.5
2 2.5
3 3.5
```

But because the `foreach` requires a function taking a `Tuple2[A, B]` as argument, the user needs to use the tuples accessors like `_1`, which aren't very pretty when it comes to functions names. To overcome this limitation, we added `filter`, `foreach` and `map` functions that would accept tuples elements directly as input, so one could then write:

```
(seqA, seqB) foreach ((a, b) => println(a, b))
```

Which leads to the same output but feels a lot more natural. The Enhanced Tuples feature is immediately available when importing the DSL package using the `_` notation.

Technical Details

Scala's tuples sizes are actually hard-coded in their typing. A tuple of arity 2 is a Tuple2, a tuple of arity 3 is a Tuple3 and so on. There is no underlying type that would allow a generic implementation of our implicit `zipped` feature for all sizes of tuples. Scala developers actually stopped implementing tuples longer than 22, as there is no Tuple23 class or higher. We then decided to provide the feature up to 5 elements inclusively, as we thought it should be enough for most problem models. We do not prevent the user to use a more conventional for loop with an index and query each sequence element using the index if he really needs to loop over more than 5 sequences at once, though.

The reason why Scala tuples are implemented like that is because they can contain different types of variables. For example a `Tuple3` is actually a `Tuple3[A, B, C]`. This allows a user to put three variables of three different types inside a tuple, which cannot really be done with a `Seq[T]` because the underlying type `T` of the `Seq` would then become `Any` in presence of variables of different incompatible types. That's the reason why we couldn't provide a generic implementation of our Enhanced Tuples for all sizes, because we couldn't use `Seq` and tuples require to define a specific function for each size. Because the only underlying type shared by all tuples is the trait `Product`. This trait has a `productElement` function to retrieve one element of the tuple, but because of typing, the return type of this function is `Any`. It would be possible to implement generic versions of some tuples-related functions though, by resorting to casting the `Any` value into a well-defined expected type. We definitely cannot do that here, however, as the user can actually put a sequence of any type in our tuples to iterate on, so we have no idea to what type we'd need to cast the `Any` besides `Seq[Any]`.

Enhanced Tuples actually mimics the behavior of a `Seq`. For example a `EnhancedTuple2[A, B]`, created from a `Tuple2[Seq[A], Seq[B]]` will react to `Seq` functions `filter`, `foreach` and `map` just like a `Seq[Tuple2[A, B]]` would. We actually build such a `Seq` in the Enhanced Tuples, and we pass the functions to it. Earlier implementations even had Enhanced Tuples extending `Seq`, however this prevented the user to use the Enhanced Selector feature on Enhanced Tuples. This issue will be described in more details in the next Technical Details section.

4.5.4 Enhanced Selectors

In Scala like many other languages, the standard way to query a sequence for its elements is by using an element index, so if one want the resource at index 3 of the `Seq[Resource]` resources, one would write `resources(3)`. But what if the user want to iterate over resources 3, 5, 7 and 12 ? It is possible to write:

```
for (i <- Seq(3, 5, 7, 12)) {
  activity needs resources(i)
  ...
}
```

But that's not convenient, and if he needs to post multiple constraints on the resources, he will have to write resource(i) quite a lot. In addition to having those nasty parenthesis in the constraints statements, which hinders the readability of the model, he also have to carry on with the `i`. With our Enhanced Selectors, one can write:

```
for (r <- resources(3, 5, 7, 12)) {
  activity needs r
  ...
}
```

The constraints we post in the loop are then shorter and more efficient, because we don't have to query the resources sequence every time we want to access one of its resource.

If the user has a sequence of indices, he can also use it to select the elements. For example if one have a `Seq[Int] requiredResources`, which contains the indices of the required resources, one can write:

```
for (r <- resources(requiredResources)) {
  activity needs r
}
```

This construction is extremely readable and it clearly states *for each resource r in the requiredResources of sequence resources*. That's a very convenient way of querying the resources, which reaches its full potential when used in conjunction with `Reader` to read indices by columns, because then one directly has a sequence of indices to use in such construction.

One can also select elements of a sequence by using a `Range`, for example:

```
for (r <- resources(3 to 7)) {
  activity needs r
}
```

The Enhanced Selectors feature goes very well with the Enhanced Tuples one. Combining both features, for example with a range selector, one can write:

```
for ((d, r) <- (demands(3 to 7), resources(3 to 7)) {
  activity needs d of r
}
```

If the sequences have the same indexing scheme, we even pushed the power of Enhanced Tuples and Enhanced Selectors even further to allow the user to write:

```
for ((d, r) <- (demands, resources)(3 to 7)) {
  activity needs d of r
}
```

We think it would be quite hard to allow a shorter way of writing the above iteration, while still keeping the final result understandable.

There is a last selector than the user can use, and we actually already encountered it in this chapter. It is the alternative selector, used for compact forms of alternative requirements:

```
activity needs (d1 or d2) of resources(1 or 2)
```

It simply converts the selected elements of the sequences to the `Alternative` type we described earlier.

The Enhanced Selector feature is immediately available when importing the `dsl` package using the `_` notation.

Technical Details

Enhanced Selectors are implemented using an implicit class in `package.scala` which takes a `Seq[T]` as argument. It has one `apply` function for each type of selectors and returns a `Seq` containing the requested elements.

For Enhanced Tuples selection support we added implicits from the tuples to Enhanced Selectors. We originally intended the extension of type `Seq` by Enhanced Tuples to be enough to trigger the implicits of Enhanced Selectors, given a `T, S <% Seq[T]` type bound for the Enhanced Selectors implicits, but it didn't work out. The reason it didn't work in practice, is because for Enhanced Tuples to extend `Seq`, we had to define an `apply(Int)` method, which conflicted with the Enhanced Selectors one when resolving implicits. Taking the good example of the above `(demands, resources)(3 to 7)`, the compiler sees a `Tuple2(Seq[CPIntVar], Seq[Resource])`, to which we try to call an `apply(Range)` method. We thought that the compiler would see that the only implicit conversion allowing such a call was the Enhanced Selector one after conversion to Enhanced Tuples thanks to the `<%` type bound. However, the compiler doesn't look at function arguments when resolving implicits. It then stopped on ambiguous implicits, because it saw that the Enhanced Selectors implicit could provide an `apply` method for `Tuple2`, through the type bound, but it also saw the `apply` method of the `EnhancedTuple2` class itself, the one we had to define to extend `Seq`. Without evaluating the function argument type, the compiler had no way of knowing which implicit to use. We then had to drop the `Seq` extension of Enhanced Tuples and add new implicits from tuples to Enhanced Selectors, using the underlying `Seq` of the Enhanced Tuples.

4.5.5 Multiple Schedulers

The Multiple Schedulers feature is implemented in the `MultipleSchedulers` trait. To use it, one needs to mix it with his model object:

```
object MyModel extends SchedulingModel with MultipleSchedulers
```

This trait has to handle multiple schedulers while ensuring there is always one and only one implicit `CPScheduler` available at a time, otherwise the compiler won't know which one to use and will fail to provide any. Having multiple schedulers can be useful to be able to use different objective functions or search methods for the same model. Users need to be able to define an objective function or a search method specifically for a given scheduler, but without having to re-write the model to register it to every scheduler!

To accommodate for those needs, we provide a `use` function on the scheduler, as well as an `apply` selection function.

Suppose the testing of three objective functions, one then wants the model to be registered to three schedulers but doesn't want to write the creation functions and constraints more than once. One can then write:

```
for (i <- 1 to 3) {
  scheduler use i

  ... // Model definition
}
```

This way, the implicit scheduler that will be used in the model definition will be the scheduler indexed by i, the for loop will register the activities, resources and constraints to each scheduler, and the user only had to write the model once. Without using an implicit scheduler, he would have needed to define the model with the scheduler hardcoded in the model statements, which would have required him to write the same model for each scheduler, with only the `scheduler` or `cp` variable changing.

The objective functions, however, are not going to be defined in a loop, as the point of this is to be able to define a different one for each scheduler. We know that we can access the implicit scheduler by writing `scheduler`, but mixing `MultipleSchedulers` in allows the user to request a specific scheduler indexed by `Int i` by writing `scheduler(i)`. One can then define a different objective function for each of them, like in the following:

```
scheduler(1) minimize(...) // Objective 1
scheduler(2) maximize(...) // Objective 2
scheduler(3) minimize(...) // Objective 3
```

The same can be applied to `search` methods. One can also add specific constraints to one of the schedulers, for example:

```
scheduler use 3
activity endsAt 20
```

The additional constraint will only be registered to scheduler 3 because it was the implicit scheduler at the constraint call site. One can even directly add the constraint to the right scheduler without changing which scheduler is the implicit one by writing:

```
scheduler(3).add(...)
```

One can also request a brand new scheduler using:

```
scheduler use new CPScheduler
```

When using the `MultipleSchedulers` trait, the scheduler gains an `id` field which contains its index to be used later in other calls to `use`. This is mandatory for the above, because otherwise, it would be impossible for the user to know the index assigned to a brand new scheduler without knowing how we implemented the `use` function.

This last `use` function actually allows a power user to use his own scheduler as the implicit one. Suppose he wants to define his own scheduler, with added functions or whatever he may want to add, he just have to extend our `CPScheduler` and makes it the implicit one using the `use` function. Suppose he have is custom scheduler in a variable named `myCustomScheduler`, he just needs to write `scheduler use myCustomScheduler`, and his scheduler will now become the implicit one.

Technical Details

The `MultipleSchedulers` trait extends `SchedulingModel`, as it has to override its function. Where `SchedulingModel` only has a single scheduler, which is instantly returned when calling `cp` or `scheduler`, this trait has to keep a `Map` of all schedulers and their IDs. Calling the two aforementioned functions would then retrieve the scheduler corresponding to the current implicit scheduler index of the `Map`, the default scheduler being indexed at 0. We needed to implement `scheduler` as a function, even though it made little sense from the perspective of the `SchedulingModel` alone, because we would otherwise override a stable `val` by a function in `MultipleSchedulers`, which is forbidden by the compiler.

The scheduler selection functions using an index are implemented as requests to the schedulers `Map`, with the difference that a new `CPScheduler` is created if none currently exists for this index. For custom schedulers, we assign them the lowest positive non-already used index. The reason for that choice is that a user is more likely to use scheduler 5 after having used scheduler 4 than going back and using scheduler 3 that he didn't used already. So assigning the new scheduler to 5 was risking the user would request scheduler 5 later thinking it would create a new one but returning the custom one instead. The safest way to deal with indices is to use the `id` field of the returned scheduler, though.

To add an `id` to `CPScheduler`, we actually defined a new `CPScheduler` class in trait `MultipleSchedulers` and we designed it such that it extends `oscar.cp.memScheduling.dsl.CPScheduler`, which has the effect of shadowing the standard `CPScheduler` in the model. When a user creates a new scheduler by writing `new CPScheduler` while having the `MultipleSchedulers` trait mixed in, he actually creates one of this new type of `CPScheduler` which is automatically registered to the `Map` of the `MultipleSchedulers` trait.

4.5.6 Custom Activities

For the recall, the activity a user creates using our previously discussed creation statements are actually objects of type `Activity` with all the feature traits mixed in.

In practice, our feature traits can be used like building blocks to tailor the capabilities of activities in the model. For example, a user may want to only enable some features. For doing so, he can create a new activity and mix it with the features he wants to enable:

```
new Activity(duration) with ResourcesRequirements with DueDate
```

If one needs to create such activities in different places of the model, we advise to define a new activity class with those traits mixed in, for example if we name the new class `MyTailoredActivity`:

```
class MyTailoredActivity(duration: Int) extends Activity(duration)
  with ResourcesRequirements
  with DueDate
```

But removing traits is less interesting than adding new ones. If one wants to add a new trait called `MyNewTrait`, it is as easy as extending `Activity`:

```
trait MyNewTrait extends Activity {
  def myNewConstraintMethod {
   ... // Posting the new constraint
  }
}
```

One can then create an `Activity` with this new trait mixed in by adding `with MyNewTrait` to the previous statements. The resulting activity will support the `myNewConstraintMethod` constraint as well as every other ones from the activated traits.

We think it is very easy and powerful, we are very fond of the trait-based design of our activities and their features.

Technical Details

All these possibilities to tailor custom activities by adding or removing traits, along with possibly implementing new activities are possible thanks to our trait-based design. This is a design we chose to use since the beginning of the implementation. The gains in terms of modularity, code organization and extensions are tremendous. The possibilities of expansions are nothing short of endless.

However, this trait-based design has a drawback. One cannot use the creation methods we described earlier for `Activity` when dealing with custom activities. The attentive reader may have noticed that the user had to write `new Activity(duration)` instead of `Activity ofDuration duration`. It means that one cannot use the plural forms of the creation statements of `Activities` either. The user then needs to populate his own `Seq[Activity]` using Scala standard way of `tabulate` or `fill`, which is not very convenient. However, there is no clean way to overcome this problem in Scala.

We would love something similar to:

```
Activities ofDurations durations with (Precendes, DueDate, ...)
```

But this would require a function `with` taking a sequence of traits, which is already not easily done because of the typing of `Seq` as we discussed earlier with Enhanced Tuples. It would be possible to define an `Int` corresponding to each trait and request a `Seq[Int]`, though. However, that's the implementation of the `with` method which is impossible to do. For the JVM, an `A` class mixed with a trait `T` is a new class type. The class `A with T` *is* a class from the JVM standpoint. Because of this, we cannot define a return type for the `with` method, because we don't know beforehand what final type the activity will have because we don't know which traits will be mixed in. We also cannot dynamically add a trait to an already created instance.

To be fair, it is possible to do so, but it requires reflection and a terrible hack using the Java `ClassLoader`. [24] We didn't want to resort to such implementation, but we thought it was interesting enough to mention it.

Chapter 5
Validation

The validation of our DSL is done in three parts. First we have to validate our code behavior, then we have to validate our DSL usefulness, by actually modeling scheduling problems, then we have to compare it with the other existing systems.

5.1 Code Behavior

For the code behavior, we have to test if our functions actually do what they are meant to. When one writes `scheduler use 3`, is the new implicit scheduler really the right one? When one writes `activities(a) needs (q1 or q2) resources (r1 or r2)`, does it actually posts the right constraints to express this requirement? We validated our code behavior through unit testing using Scalatest. We won't detail them here, but the interested reader may find them in package `memScheduling` of `oscar-cp/src/test/scala/`.

5.2 Scheduling Problems Modeling

As for the DSL itself, we will show how one can model the problems introduced in the Problem Analysis section, using our features and functions.

We start with the Job Shop Scheduling Problem. The instances we used are structured in two part. The first one contains 3 integers on one line, respectively for the number of jobs, the number of tasks per jobs and the number of machines. The other describes each task as three integers, respectively for the id of the job from which the task is part, the id of the resource the task require, and the duration of the task.

```
// Parsing
read fileFrom "data/memScheduling/jobshop/ft10"
val Seq(nbJobs, nbTasks, nbMachines) = read fileFor 3 as Int
val Seq(jobs, required, durations) = read fileFor 3 columnsOf Int
```

Once the reading is done, we need to set the scheduler horizon, create the activities and resources, and post the constraints. We post the resources requirements of the activities using the Enhanced Selectors to retrieve the required resources from the resources sequence, and the Enhanced Tuples for easy writing of the `for` loop. We then post the precedences constraints, which apply for consecutive activities of the same job.

```
// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val machines   = UnaryResources(nbMachines)

for ((a, r) <- (activities, machines(required))) {
  a needs r
}

for (i <- 0 until jobs.length-1 if jobs(i) == jobs(i+1)) {
  activities(i) precedes activities(i+1)
}
```

We show visuals for the resources and the activities, ordered and colored by jobs. We then tell the scheduler to minimize the makespan of the problem, and to search using the `setTimes` searching method.

```
// Visualizing
visualization withTitle "OscaR Example: JobShop"
visualization show machines withTitle "Machines"
visualization show activities by jobs withTitle "Jobs"

// Searching
scheduler minimize makespan search {
  setTimes(activities.starts, activities.durations, activities.ends)
} start()
```

For the big picture, the whole code for this problem, including the parsing, can be found in the appendix, to avoid cluttering this chapter.

As one can see, the reading of the file is done in two lines, because we use one line for each different part of the file. The modeling is pretty straightforward, and we are pretty sure the attentive reader wouldn't even need our explanation of the code in order to understand it. The function names are

self explanatory, so anyone with no knowledge of OscaR or our DSL would still be able to understand this code at a glance. The only part that would perhaps require an explanation for complete neophytes to our DSL is the selection of required resources from the resources sequences using the Enhanced Selectors features. However, one may read the for loop as "for each activity and resource in all activities and the required resources", so we think it is quite straightforward as well. One simply has to understand we are using a sequence of indices to select the sequence of resources corresponding to those indices. We would even have been able to write the entire modeling in one for loop. However, this would have required line returns as the page width is rather short considering the code fonts. We preferred using two separate loops, to demonstrate how one can use the enhanced tuples features to access elements of the model or still use the more common way of using indices. If we had used only one for loop however, and not considering the parsing as a part of the modeling proper, we would have modeled the Job Shop problem in 6 lines of code, including the horizon definition. We think it's a pretty honorable result, particularly when considering the readability of the final result.

We won't be showing visualization and search details for every problem though, as they are quickly going to be redundant. We will still show them again when they use new features or are interesting in some way.

We introduced multiple variants of the Job Shop problem in the Problem Analysis section. The Multiprocessor variant uses Cumulative Resources instead of Unary ones, so the resources can execute multiple activities at once, up to their capacities, that we will name `capas`. Handling such problem would require to modify the resource definition line by something like:

```
val machines   = CumulativeResources ofCapacities capas
```

And the resource requirement constraint like this:

```
a needs 1 of r
```

We especially designed our constraints functions to be readable and clearly stating what they were for. It is then really easy to understand which line does what in a code using our DSL, spot which one to modify, and modify it accordingly.

The Job Shop with Earliness/Tardiness would simply require to add a due date to each job and change the objective of the scheduler, but we won't detail it here. Please refer to the Single Machine with Earliness/Tardiness

and Common Due Date problems for everything related to due dates and Earliness/Tardiness, as it is the core of these problems and the basis used for the E/T variants of the others.

Preemptible Job Shop and Job Shop with Setup Times are not supported by our DSL however, as we had to decide on a subset of features to implement, we were not able to implement everything needed for every problem. However, we can give some starting tracks. For example, the `Activity` class posts its consistency constraint via a method, so one can create a trait which would override the method to prevent it from posting its consistency, which would otherwise create problems with preemptible activities. One can then mix this trait with `Activity` and implement the other features that would come with preemptibility. For Setup Times, we will use a kind of weaker version when modeling the Aircraft Landing problem.

The second class of problem we introduced is the Flow Shop Scheduling Problem. However, it is very similar to the standard Job Shop. Only what is allowed as a value of the problem is different, not the way it is modeled, so it is redundant with the Job Shop modeling. The Multi-Processor and E/T variants of Flow Shop are similar to the Job Shop ones. The Re-entrant variant only changes the values that can be used in the problem, not the model itself either. The Flow Shop has an interesting variant however, in the one which uses Buffers. We did not implement such buffers in our DSL, but one could use the same trait-based approach we used for activities to add features to resources. One of these features may then be a Buffer feature. Our design is highly modulable so there is endless possibilities for extensions.

As for the Open Shop Scheduling Problem, it is also very similar to the standard Job Shop. The only difference is that instead of precedences constraints, only one task per job may be scheduled on a resource at any one time. It is possible to model such behavior by adding one unary resource per job and constrain every task of a job to require the job's respective resource in addition to the resource the activity already require. For example one would replace the precedences part of the Job Shop modeling above by:

```
val jobResources = UnaryResources(nbJobs)
for ((a, r) <- (activities, jobResources(jobs))) {
  a needs r
}
```

Because each task of a same job will require the same unary resource, they will not be able to overlap.

Resource Constrained Project Scheduling Problem, RCPSP, is the next type of problem we introduced in the Problem Analysis section. It comprises activities and cumulative resources, but it can also use non-renewable resources, which is our `ReservoirResource`. The instances of RCPSP we used are composed of 3 parts. The first one contains 2 integers on one line, respectively for the number of tasks and the number of resources. The second one contains 4 integers which are the capacities of the 4 resources. The last part is more complicated, and is itself composed of two parts. The first one is 5 columns of integers, respectively for the duration of each activity and the demands of each activity with respect to the 4 resources. The second part is a column of integers telling the number of successors of each activity, then on each line the successors of each activity. So the instance file has unregular line lengths. Hopefully this is not a problem to read it.

```
// Parsing
read fileFrom "data/memScheduling/RCPSP/J301_1.RCP"
val Seq(nActivities, nResources) = read fileFor 2 as Int
val capaResources     = read fileFor 4 as Int
val Seq(durations)    = read fileFor nActivities x 1 asColumnsOf Int
val allDemands        = read fileFor nActivities x 4 asLinesOf Int
val Seq(nPrecedences) = read fileFor nActivities x 1 asColumnsOf Int
val precedences = read fileFor nActivities x nPrecedences asLinesOf Int
```

Once the reading is done, we have to set the horizon of the problem, and create the activities and resources. We then post the precedences constraints, as well as the requirements ones.

```
// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val machines = CumulativeResources ofCapacities capaResources

for ((a, demands, preceded) <- (activities, allDemands, precedences)) {
  for (b <- preceded) {
    a precedes activities(b-1)
  }

  for ((d, m) <- (demands, machines)) {
    a needs d of m
  }
}
```

We then show the cumulative resources of this problem. The objective is to minimize the makespan, but it is redundant.

```
// Visualizing
visualization zoomedBy 10 withTitle "OscaR Example: RCPSP"
visualization show machines
```

For the big picture, the whole code for this problem, including the parsing, can be found in the appendix, to avoid cluttering this chapter.

This time we used the old reading system instead of the pattern-based one so we can show both of them. One can see it is a bit of a problem when reading only one column, because as it is supposed to output a series of column, we have to match it with a `Seq` to retrieve the sole column of the sequence. It is much more clean to use the pattern-based version of the reader, using the `dependsOnNext` pattern. One can than write:

```
val (nTasks, nResources) = read fileFor (Int, Int)
val capaResources  = read fileFor 4 as Int
val tasks = read fileFor nTasks withPattern
                            (Int, Int(4), dependsOnNext as Int)
val (durations, allDemands, precedences) = tasks
```

Besides the reading, the modeling is straightforward, this is actually a testament to how easy our DSL is. The problem has resources requirements, for which it gives the quantity required of each resource in a given column for each resource, and precedences, for which it gives the list of activities preceded by each activity. We simply have to iterate on each activity, demands of this activity and other activities preceded by it, and post the precedence constraints and the resources requirements.

For the multi-mode variant of RCPSP, one have to register multiple modes for each activity. This is actually a use of alternative resources. The instances files contains the number of modes, and the quantity of each resource required in each mode. For example, to model activity `a` having two different modes (`m1` and `m2`) each requiring two resources (`r1a` and `r1b` for `m1` and `r2a` and `r2b` for `m2`), one would write, `d` being the sequence of demands and `r` the sequence of resources:

```
activities(a) needs (d(m1)(r1a) or d(m2)(r2a)) of (r(m1)(r1a) or r(m2)(r2a))
activities(a) needs (d(m1)(r1b) or d(m2)(r2b)) of (r(m1)(r1b) or r(m2)(r2b))
```

But we then need to constraint that if one alternative is chosen over the other, then the other required resources must be from the same mode:

```
add((a.alternativeTo(r(r1a)) === r(r1a).id)
 == (a.alternativeTo(r(r1b)) === r(r1b).id))
add((a.alternativeTo(r(r2a)) === r(r2a).id)
 == (a.alternativeTo(r(r2b)) === r(r2b).id))
```

It would have been interesting to implement an alternative requirements functions which would have accepted a whole sequence of resources as one alternative to simplify this. Unfortunately we designed the alternative requirements functions on very small problems and we originally thought this design would be enough.

The Max. quality variant would be rather easy to do, because the user would just have to build its quality function on the activities durations, accessible from a sequence of activity by using `.durations` to retrieve a sequence of durations. Please refer to the next two problems for Earliness/Tardiness considerations.

For Single-Machine Weighted Tardiness, the instances we used contained 40 tasks, and were described in three lines. One for the durations, one for the weights, and one for the due dates.

```
// Parsing
read fromFile "data/memScheduling/weighted-tardiness/wt40_1"
val nbTasks   = 40
val durations = read fileFor nbTasks as Int
val weights   = read fileFor nbTasks as Int
val dueDates  = read fileFor nbTasks as Int
```

Once the reading is done, we have to set the scheduler horizon. We then create the activities, using the sequence of durations, and the single machine. We then assign its due date to each activity and states it requires the single machine. We have to post a `weightedSum` constraint to weight the activities tardiness with respect to their assigned due date, and tell the scheduler to minimize it.

```
// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val singleMachine = UnaryResource()
```

```
for ((a, d) <- (activities, dueDates)) {
  a needs singleMachine
  a isDueAt d
}

val weightedTardiness = weightedSum(weights, activities.map(_.tardiness))

// Searching
scheduler minimize weightedTardiness
```

The whole code for this problem, including the parsing, can be found in the appendix.

It actually took us less than 5 minutes to model this single machine weighted tardiness problem, including parsing the file. Models using our DSL may even be used to explain a scheduling problem to someone. Usually one would want to read descriptions and explanations on a given scheduling problem before starting reading its code modeling. This is because other modeling languages require a deep knowledge of the language to understand what is going on, so knowing the problem well can greatly help in understanding the code. Howevern, if the code is clear, readable and straightforward, one could *learn* the problem by *reading the code*. We think this could be done using problems modeled with our DSL. The above is a very good example, and so is the next one, the Common Due Date Scheduling Problem.

This problem is similar to the previous one, but this time all activities have a common due date, which is indeed strictly impossible as all activities need to be scheduled on one machine. We then try to minimize the joint sum of weighted earliness and tardiness of the activities with respect to their common due date. The instances files are composed of two parts. The first one is a single Int giving the number of tasks of this problem, and the second is three columns, describing for each task its duration, its earliness cost and its tardiness cost respectively.

```
// Parsing
read fileFrom "data/memScheduling/common-due-date/sch10/cdd10_10.txt"
val nbJobs = read fileFor Int
val Seq(durations, earlyCost, lateCost) = read fileFor 3 columnsOf Int
```

Once we have read the file, we build the due date which is, by definition of this problem, the sum of the durations times h, a parameter to be set by the user to have a more or less restrictive due date, rounded down. We can

then set the scheduler horizon. However, it is important to note that using the sum of durations is a bad idea this time. Because we want activities to approach the common due date as much as possible, noting requires that activities scheduling starts at time 0. Because of this, activities may be scheduled at a time beyond the sum of the durations, and setting such a restrictive horizon for this problem would miss the optimal solution. Like the previous problem, we state the resource requirement for each activity, as well as its due date. Here, the due date is the same for each activity, which wasn't the case for the previous example. We weight both the tardiness and earliness this time.

```
// Modeling
val h = 0.4 // External parameter, to be set by user.
val dueDate = (durations.sum * h).toInt

scheduler horizon = dueDate + durations.sum

val activities = Activities ofDurations durations
val resource   = UnaryResource()

for (a <- activities) {
  a needs resource
  a isDueAt dueDate
}

val weightedEarliness = weightedSum(earlyCost, activities.map(_.earliness))
val weightedTardiness = weightedSum(lateCost,  activities.map(_.tardiness))
```

We show the due date on the visualization charts, and we then tell the scheduler to minimize the sum of the weighted earliness and tardiness.

```
// Visualizing
visualization show activities by activities.ids
visualization show dueDate

// Searching
scheduler minimize(weightedEarliness + weightedTardiness)
```

The whole code for this problem can also be found in the appendix.

As discussed for the Weigthed Tardiness problem, people not knowing anything about our DSL or the scheduling problem we are currently modeling, but still having a bit of programming knowledge, would quickly understand what we are doing. There is very little noise, nearly everything that is written in the code refers to the problem, we see the scheduler very little and most of the work is already done for the user, so he can concentrate on what is really interesting, that is modeling his scheduling problem.

The Trolley problem however, cannot be modeled using our DSL because we didn't implement State Resources so we cannot model the trolley itself. However, one may write a trait as an extension of our existing resources, which would modify which activity can execute at a given time, considering the current state of the resource. We will expand on the traits possibilities for resources in the Future Work section.

The next problem we introduced in the Problem Analysis section is the Aircraft Landing Problem, where each plane as a time window during which it must land and a target time for the landing, which is modeled as a due date. Additionally, each landing has a separation time with respect to other landings if they have to happen next. The instances for this problem are the stranger ones we had to deal with, and are actually not possible to read comfortably if not using the pattern based approach. The file is not structured by lines or by columns at all. Instead, each plane is described one after the other. Each plane is comprised of four integers, its appearance time, earliest landing time, latest landing time and target landing time respectively. Then come two doubles, for the earliness cost and tardiness cost incurred when not meeting the target landing time. After that, each separation time enforced by this plane's landing with respect to each other plane is then described, but on multiple lines! This pattern is repeated over the file for each plane, and is the only structure of the file. We have to read it by plane instead of reading it by lines or columns. That being said, it is very easy to read such file using the pattern-based reader !

```
// Parsing
read fileFrom "data/memScheduling/aircraft-landing/airland1.txt"
val (nbPlanes, freezeTime) = read fileFor (Int, Int)
val planes = read fileFor nbPlanes withPattern
                  (Int, Int, Int, Int, Double, Double, nbPlanes as Int)
val (appearances, earliests, targets, latests,
     earlyCosts, lateCosts, delays) = planes
```

Once we have read the file, we can set the scheduler horizon, which is the highest latest landing time, because these no plane can land beyond that. We then create the activities, which are actually the landings, and as such are treated as points in time as they have no duration assigned. We then set their duration to 0. The duration is actually irrelevant, but we have to set one. One could use any number, but if using a number different than 0, be sure to state the constraints on the end of the activity instead of the start, because earliness and tardiness are computed with respect to the end of an activity. Once the landings are created, we can constraint them to be in the predefined time window and set their target landing time. We then go through each plane separation times and set for each other plane that if they are preceded by this plane, then they have to respect the separation time. Using strict separation times without the conditional form would render the problem impossible, because then each plane would have a forced separation time with respect to every other plane. Once this is done, we than have to convert the costs to integers, because OscaR doesn't support real values, and weight each landing earliness and tardiness.

```
// Modeling
scheduler horizon = latests.max
val landings = Activities(nbPlanes) ofDuration 0

for ((landing, min, max, target) <- (landings, earliests, latests, targets)) {
  landing startsBetween min and max
  landing isDueAt target
}

// For each landing and its associated delays
for ((i, delays_i) <- (landings, delays)) {
  // For each other landing and its associated delay with respect to i
  for ((j, d) <- (landings, delays_i) if (j != i)) {
    add((i isPreceding j) ==> (i isStartingAtLeast d beforeStartOf j))
  }
}

// We need to convert the costs because Oscar doesn't support real values.
val elc = earlyCosts.map(c => (c*100).toInt)
val llc = lateCosts.map(c => (c*100).toInt)

val weightedEarliness = weightedSum(elc, landings.map(_.earliness))
val weightedTardiness = weightedSum(llc, landings.map(_.tardiness))
```

We also show the target landing of each plane on the gantt chart.

```
visualization show landings by landings.ids zoomedBy 3
for (t <- targets) {
  visualization show t
}
```

The whole code for this problem can be found in the appendix.

We referenced the static case for this problem so we don't use the values from freeze time and appearances. Because we model this as a single runway problem, we don't even need to model the runway at all. Modeling it with multiple runways would require to create the runways as unary resources and state alternative requirements with each runway as alternative. Please note that it would also require the landings durations to be greater than 0, otherwise the `unaryResource` constraint would fail.

```
val runways  = UnaryResources(2)

for (landing <- landings) {
  landing needs runways(0 or 1)
}
```

But the biggest challenge would be the separation times, because this time we would need an additional condition for the separation time to be true. We would need to add that the landings use the same runway as an additional condition to trigger the separation times.

```
add(((i isEndingBeforeEndOf j)
     & (i.alternativeTo(runways(0)) === j.alternativeTo(runways(0))))
     ==> (i isEndingAtLeast d beforeEndOf j))
```

It is a bit long to write, but it is a complicated constraint. Besides, the user needs to use the `add` function because of how `CPBoolVar` works in OscaR. To be able to chain logical expressions, creating a `CPBoolVar` doesn't imply it is `true`, so when one writes:

```
a ==> b
```

It doesn't post the constraint that a implies b, it creates a new `CPBoolVar` which can be used in chaining like:

```
(a ==> b) ==> c
```

To constraint that the logical expression must be true, the user has to use the `add` method.

```
add((a ==> b) ==> c)
```

The next problem on the list is the Shift Minimization Personnel Task Scheduling Problem. The instances are very verbose and contains a lot of text, so the reading is a bit long because we have to drop the useless descriptive words.

The first four lines are descriptions, so we drop them.[1] Then comes the line containing the number of jobs. Unfortunately, unlike other problem instances, this line reads "Jobs = 40" instead of simply "40". We then need to drop 2 words to be able to read the number of jobs. Which comes after is two columns of integers for the starts and ends of each task respectively. We then have the number of shifts, in the same form as the number of jobs, so we first have to drop 2 words then read the value.

After this, we have the same kind of pattern we saw with the RCPSP, where the first number on the line indicates how much other numbers there are to read on this line. However, because this file is very verbose, the numbers of this first column have a colon right to them, so the values read like "26:" instead of "26". We did not expect this, so we have to read the column without converting it to Int, remove the colons, then convert it to Int. We can then read the lines of unregular lengths by giving the newly retrieved lines lengths to the reader.

```
// Parsing
read fromFile "data/memScheduling/shift-minimization/data_1_23_40_66.dat"
read dropping 4 lines;
read dropping 2 words
val nbJobs = read fileFor Int
val (starts, ends) = read fileFor nbJobs withPattern (Int, Int)
read dropping 2 words
val nbShifts = read fileFor Int
val nbQualifiedJobs = (read fileFor column).map(_.replaceAll("[:]", "").toInt)
val qualifiedJobs = read fileFor nbShifts x nbQualifiedJobs asLinesOf Int
```

[1] We need to either use parenthesis for the call to `lines` or add a semicolon at the end of the call, because Scala doesn't really like parenthesis omission for functions with no parameters. We wouldn't need to do it if the next statement was a variable definition though, but here we are calling another function right after it.

We can then set the scheduler horizon, which is 1440 by definition of the problem, the number of minutes in 24 hours. The shifts are activities which spans 24 hours so 1440 as well. They are optional and gives 1 of capacity to their assigned `worker` resource, which are otherwise empty. The only way to get workers resources is to schedule some shifts.

```
// Modeling
scheduler horizon = 1440
val shifts = Activities(nbShifts) ofDuration 1440
val workers = CumulativeResources(nbShifts) ofCapacity 0
for ((shift, worker) <- (shifts, workers)) {
  shift optional = true
  shift gives 1 of worker
}
```

The jobs to be scheduled have, unlike every other problem until now, fixed starts and ends, so the file doesn't provide durations for them, we then have to do a little trick to get them in our DSL. We then retrieve for each job the workers resources that are qualified to do that job. For example the shift 4 gives 1 of the resource worker 4, which are qualified for jobs 0, 1, 2, 6, etc. So each job alternatively require one of the qualified workers for that job. They can't be done by other workers. Unfortunately, we didn't understand how to model the problem at first, and because of that, our implementation of alternative resources fell short to model it. Our pretty `or` notation would require us to write each qualified workers with that notation, which would be really tiresome. We then added an `alternatively` method, which converts any sequence to an `Alternative`, so it is considered as if we wrote `or` between each of them. However, because demands can be different for each alternative, we need to create an `Alternative` of the same size filled with 1.

```
val jobs = (starts, ends).map((s, e) => Activity ofDuration e-s)
for (i <- 0 until nbJobs) {
  jobs(i) startsAt starts(i)
  jobs(i) endsAt ends(i)
  val qualified = (0 until nbShifts).filter(j => qualifiedJobs(j).contains(i))
  val demand = alternatively(Seq.fill(qualified.length)(1))
  jobs(i) needs demand of alternatively(workers(qualified))
}
```

We have to admit our implementation of alternative resources falls a little short when working with big numbers of alternatives.

When this is done though, we have to build the sum of scheduled shifts to use them in the minimization of the scheduler. It is important to force the search to bound the values of the jobs required resources though.

```
val nbShiftUsed = sum(shifts.map(s => s ifScheduled 1 ifNot 0))

// Searching
scheduler minimize nbShiftUsed
search {
  binaryFirstFail(jobs.requiredResources :+ nbShiftUsed)
} onSolution {
  shifts foreach println
} start()
```

For the big picture, the whole code for this problem can be found in the appendix.

This last problem was the most tricky to model, notably because of the way we model alternative resources. It is usable thank to the `alternatively` method, but the repeated demand value still remains. On the other hand, allowing the `needs` function to be called with a single `Int` argument would conflict with the method of the same name of the standard resources requirements, both resulting in an object on which one could call the `of` method, so Scala wouldn't be able to tell the difference. Even if the different arguments of the `of` methods would be enough for us to tell the difference, the compiler wouldn't.

Apart for some quirks with alternative resources, we are able to model nearly every problem we described in the Problem Analysis section, along with most of their variants. We are pretty correct on the criteria of expressiveness. Our functions are sometimes a bit long to write, however. It is quite hard to come up with clear expressive names while still complying with parenthesis omission rules.

The problems we can't model are related to some features we didn't implement, as we chose to develop a subset of them as discussed in the Problem Analysis section. Still, our trait-based design being extremely friendly to extension, these would probably be easy to add in the future. We discuss some ideas of extensions in the Future Work section.

5.3 Comparative View

Now that we have discussed how our DSL fared with respect to the modeling of the scheduling problems we described in the Problem Analysis section, it is interesting to compare it to the other existing systems. For this, we will return to the Job Shop example we used to analyze those systems back in the Related Work section and compare their modeling code with our own. To be fair when comparing with those systems, we will ignore the Parsing and Visualizing sections of our model. Those are additional features we thought interesting to add into the work of this thesis, but they are not strictly required to *model* scheduling problems.

For the recall, and to avoid painful returns to the Related Works section, the modeling parts of the other existing systems are in the following.

```
tuple Operation {
   int mch; // Machine
   int pt;  // Processing time
};

Operation Ops[j in Jobs][m in Mchs] = ...; // Input from file

dvar interval itvs[j in Jobs][o in Mchs] size Ops[j][o].pt;
dvar sequence mchs[m in Mchs] in
          all(j in Jobs, o in Mchs : Ops[j][o].mch == m) itvs[j][o];

minimize max(j in Jobs) endOf(itvs[j][nbMchs-1]);
subject to {
  forall (m in Mchs)
    noOverlap(mchs[m]);
  forall (j in Jobs, o in 0..nbMchs-2)
    endBeforeStart(itvs[j][o], itvs[j][o+1]);
}
```

Figure 5.1: Modeling of a Job Shop problem in OPL

```
int horizon =     sum(j in Jobs,t in Tasks) duration[j,t];
Scheduler<CP>     cp(horizon);
Activity<CP>      a[j in Jobs,t in Tasks](cp,duration[j,t]);
Activity<CP>      makespan(cp,0);
UnaryResource<CP> r[Machines](cp);

minimize<cp>
  makespan.start()

subject to {
  forall (j in Jobs, t in Tasks : t != Tasks.getUp())
    a[j,t].precedes(a[j,t+1]);
  forall (j in Jobs)
    a[j,Tasks.getUp()].precedes(makespan);
  forall(j in Jobs, t in Tasks)
    a[j,t].requires(r[machine[j,t]]);
}
```

Figure 5.2: Modeling of a Job Shop problem in Comet

```
Schedule<Mod> s();
Job<Mod> J[i in jobs](s,IntToString(i);
Machine<Mod> M[i in machines](s,IntToString(i));
Activity<Mod> A[i in tasks](s,proc[i],IntToString(i));

forall(i in tasks)
   A[i].requires(M[mach[i]]);
forall(i in jobs)
   J[i].containsInSequence(all(j in machines)A[job[i,j]]);
s.minimizeObj(makespanOf(s));
```

Figure 5.3: Modeling of a Job Shop problem in AEON

Now let's look at our implementation using our DSL:

```
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val machines   = UnaryResources(nbMachines)

for ((a, r) <- (activities, machines(requiredRes))) {
  a needs r
}

for (i <- 0 until jobs.length-1 if jobs(i) == jobs(i+1)) {
  activities(i) precedes activities(i+1)
}

scheduler minimize makespan
```

Figure 5.4: Modeling of a Job Shop problem using our DSL

We think the comparison is crystal-clear. Where all the other systems have coding artifacts like `dvar interval`, `<CP>` or `<Mod>`, we only have the standard, nice and short `val` of Scala. There is no noise in our code. Everything that is displayed is useful for the modeling. The only exception is the two instances of `scheduler`. To be fair, the second one can be omitted in recent versions of OscaR. For the first one, if we had defined a `horizon` function, it would have prevented the user to create a `horizon` variable. Our Enhanced Tuples and Enhanced Selectors feature are both used in this example, in the `for` loop stating resources requirements. Both features are very powerful in reducing noise around constraints. Without these, the same statement should be written:

```
for ((a, r) <- (activities, requiredResources).zipped) {
  a needs machines(r)
}
```

Or without tuples entirely:

```
for (i <- 0 until activities.length) {
  activities(i) needs machines(requiredResources(i))
}
```

Both cases introduce noises in the code that hinder its readability. That's why we really like our Enhanced Tuples and Selectors features, they let the user focus on the constraints instead of the intricacies of the language. This way, constraints are clean and readable.

An interesting concept from AEON is the job concept. Precedences constraints doesn't appear in the AEON model because they are implied by `containsInSequence`. We explored with a similar concept early in development, but we found some problems with this approach. First, the name of the function is too generic, what if the user wants to order the activities of the job according to something else? We experimented with some functions like `orderedByPrecedence`, but we were unhappy of the result considering all the possibilities. There would have needed to be `orderedByCompletionTime` and `orderedByDuration` at least. Then what about ordering on the number of required resources? The list goes on. We prefer to let the user the freedom to order the activities the way he wants rather than trying to provide a pre-made function for all possible orderings. The second problem we found with this approach, is the concept of job itself. A job has different meanings according to which problem the user is modeling. In some problems, jobs are a series of activities ordered by precedences. In other, there are series of activities with no ordering, but only one activity per job can be processed at any one time. Yet in some problems, a job is simply an activity, a single task. We didn't want to make a decision on which one of those concepts of jobs was better than the others.

As we already discussed a bit with Single-Machine Weigthed Tardiness and Common Due Date, we think models written using our DSL are somewhat actually easier to understand than textual descriptions of the problems themselves. Indeed that's only valid for people with enough programming knowledge to understand basic constructions like a `for` loop, as a textual description would still be better for people with no knowledge in programming at all. Still we think the comparison with the other systems is interesting. When we read the code of the models written in the other systems, we needed to have a strong understanding of the problem that was modeled in order to understand the functions used. We would have stood no chance of understanding how the problem worked given only the modeling code from the other systems. We think our DSL fare a lot better in this regard thanks to the advantages of Scala. The parenthesis omissions really help a lot in cleaning the code and making it more readable, and the functions defined on the companion objects make for clear and expressive coding.

Something that was only mentioned until now but we still think is important is function discovery. When, as students, we use a new Java or Scala library in Eclipse or any other modern IDE, we usually write a dot right next to an object and the IDE then displays a list of possible functions on that object. This feature allows us to discover what is possible to do with an object without having to read a manual or browsing online. Provided the functions names are expressive enough, we can find what we were looking for and start using the library instantly. This is not possible with OPL because they use functions that take multiple parameters, like `endBeforeStart(a, b)`. There is no way to find this function unless the user already knows it exists. Because we defined such functions on an activity instead, we can write `activities(a)`, then a dot, and receive a list of every function that is possible on that activity, with respect to enabled traits. One may argue that Comet and AEON also use the same function call pattern, but the last time we used the Comet editor from Dynadec, that is last year, it didn't offer a function discovery feature like Eclipse does, provided the functions used were designed in this way.

All in all, we think our DSL is a lot more user-friendly than other existing systems, even to people with less programming knowledge. We think it may well be used to teach scheduling to newcomers, while the other systems require to be well-versed in the domain before even attempting to model problems using them. However, we currently only support a limited set of features. Even though we already have quite a lot of possible functions and constraints, we are still a long shot away from being able to model every variant of every existing scheduling problem.

However, we really think it is a good framework to model scheduling problems, especially given the endless possibilities of extensions it offers. As to extensions themselves, we will discuss more about the various ideas we have in the Future Work chapter.

Chapter 6
Future Work

There are plenty of possibilities for future work on this DSL, because scheduling problems are vast and there is always some room to improve.

First of all, we would have liked to integrate more some features that came late in development. A good example we already discussed about is the pattern-based approach of the Reader. Currently we have the two approaches, but the pattern-based one is clearly more powerful. However, the old approach is more dense and requires the user to write less code. Keeping both approaches have its merits from the user standpoint, but we are pretty sure it would be possible to unify them at the code level. It would be cleaner from a developer standpoint, while still keeping both features for the user.

Speaking of the reader, it would be really interesting to provide the user with an access to the split regexp of the `DataFile` class. It already as such `String` regexp in the class so it wouldn't be that difficult. By doing so, one could even create predefined regexp which would only parse numbers and delete letters for example. This would greatly simplify the reading of more verbose instances files. Other predefined regexp may be imagined for commonly used instances formats in order to minimize the work of reading them.

Something that came pretty late in the development is the `setup` method of activities. We were first reluctant to add it because we thought we may be able to develop all features without it, be we finally had to add it for `Optional`. We modified `DueDate` to use it because it solved one minor problem we add with it at the time, but we thought it may be interesting to review the other feature traits to see if they could be improved by using the `setup` function.

The conditional forms of the constraints also came pretty late in development, with forms like `isPreceding`. It should be interesting to look back at the other constraints to implement functions like that. For example we can currently compare two activities requirements by comparing the result of their `alternativeTo` function, but we may also have a function like

`isRequiringSameResourceAs`, although we would need a better name. It also poses the problem of the `add` function that the user is required to use in order to actually post a `CPBoolVar` to be true. It would be interesting to think of a way of bounding such unbounded `CPBoolVar`. We though about a `===>` method, but we think it is already tricky enough for the user to differentiate between both `==>` and `=>` functions. We also thought of a kind of enhanced `CPBoolVar`, which would take a predicate as parameter. The predicate would default to `true`, unless created from another `CPBoolVar`. For example, something like: following:

```
class EnhancedCPBoolVar(v: CPBoolVar,
                        predicate: CPBoolVar = CPBoolVar(true))
```

Using the standard `CPBoolVar` functions like this:

```
a == b
```

Would result in an `EnhancedCPBoolVar` with its predicate set to true. But the `EnhancedCPBoolVar` would override each logical function of the standard `CPBoolVar` to create an `EnhancedCPBoolVar` instead, using `this` as a predicate when creating the new resulting variable. It's only a concept though, we are not sure if it would be possible in practice. Even if it was, it is a lot of code to automatically post `CPBoolVar`s, but we thought the concept was interesting enough to mention it.

We realize the support of alternative resources by our DSL falls a bit short when the problems involve more than a few alternative resources. The `alternatively` method we implemented when we were modeling Shift Minimization Personnel Task Scheduling helps a bit, but it does not solve the problem entirely because we think it lies in our design of alternative resources itself. We designed it for very simple problems involving alternative resources, and one can feel it when dealing with more complex problems. That is a part of our DSL we wished we had designed differently from the ground up.

However, as a general rule with our DSL, we designed it extremely extension-friendly by using traits everywhere, so there are endless possibilities to add new features to our activities or modify the ones we provide. Something interesting we thought about would be a due date for the start of the activity. We think the concept of due date is interesting, because it does not actually constrain the activity. When one says the due date of an activity is 92, the solver won't fix the end of the activity to 92. Instead, the user will build

some functions using the earliness and/or tardiness of the activity, which will make the solver tend to approach the due date. We think it adds an interesting layer of granularity, and we would like this to exist for the start of an activity as well, something like a due starting date.

Something that may be interesting to explore is using traits for resources as well. What if a `Resource` could become a unary resource when mixed with the `Unary` trait, a cumulative resource with the `Capacity` trait, and a reservoir resource with both the `Capacity` and `NonRenewable` traits ? A state resource could then be a `Resource` with the `State` trait. What is interesting with this design is one could build a non-renewable cumulative resource with a state by mixing `Resource` with `Capacity`, `NonRenewable` and `State`. If not using traits, such a resource would require a brand new class tailored especially for this purpose, which is not really efficient. Another example would be adding the resource buffers that were needed for one of the problems variants. Adding a buffer to a resource would then be as easy as mixing the `Buffer` trait in. One cannot hope to create a specialized class for each type of resource of each type of problems, but one *can* achieve that by providing the user with a trait-based design that can be used to tailor resources at will.

That's what we already did with the activities, we provided the user with ready-to-use building blocks to build activities that meet his needs. We also provided users with a pre-built easy `Activity` constructor with all features enabled for the ones not interested in building custom activities. The possibilities are open for anyone to add any interesting feature easily.

Part II

Filtering Discrete Cumulative Resources

Chapter 1

Background Material

Before talking about algorithms we'd like to recall some concepts of constraint programming that we will use a lot throughout this part.

1.1 Activity

An activity is a task which has an earliest start time *(est)*, a latest completion time *(lct)*, a processing time *(p)* and some resources requirements *(d)*. With these attributes we can compute the latest start time *(lst = lct - p)*, the earliest completion time *(ect = est + p)* and the energy *(e = d*p)*. Activities are non-preemptive, meaning that an activity starting at time t must be processed without interruption until t + p. Finally we maintain a decision variable for the start time $[est, lst]$ and the end time $[ect, lct]$ of the activity. These domains will further be reduced by the constraint.

All of those decision variables constitute a domain D. The main goal of constraint programming is to use constraints in order to progressively reduce the current domain by discarding inconsistent values in D.

1.2 Resource

The activities need one or many resources to complete their tasks. All the following suppose that the resource is a single discrete cumulative resource. A resource is said cumulative (by opposition to disjunctive) if it can process more than one activity at the same time.

Chapter 2

Related Works

In this section we introduce some existing techniques used to propagate cumulative resource constraints.

2.1 Timetabling

The goal of the timetabling algorithm is to spot activities i with a latest starting time smaller than its earliest completion time, i.e $lct_i - p_i < est_i + p_i$. In this case we know that these activities must use the resource between $lct_i - p_i$ and $est_i + p_i$ (Figure 2.1). Therefore we can compute a timetable by aggregating all of these intervals which will represent the minimum use of the resource over time. The timetable will detect if the maximum capacity of the resource is exceeded and adjust activities' bounds if necessary. More details on the sweep technique of Letort, Carlsson and Beldiceanu can be found in [25].

2.2 Edge Finding and Extended Edge Finding

The Edge Finding algorithm bases its propagation rule on a set of activities instead of all of them. We define a set of activities Ω in which the total energy e_Ω is the sum of all activities included in the set. This set must be executed in the interval $[est_\Omega, lct_\Omega]$, where :

- for $a \in \Omega$, $est_\Omega = est_a$ iff $est_a \leq est_{a\setminus\Omega}$ and
- for $b \in \Omega$, $lct_\Omega = lct_b$ iff $lct_b \geq lct_{b\setminus\Omega}$

We suppose that this energy is not greater than the total energy available between the interval $[est_\Omega, lct_\Omega]$. Then Edge Finding determines whether another activity i can start or end after (or before) set Ω by checking the available energy during the interval $[est_i, est_\Omega]$ and $[lct_\Omega, lct_i]$ (Figure 2.2). In other words, we search some ordering relations between the activities, like the edges in a graph. More details on these propagation techniques can be found in Baptiste, Le Pape and Nuijten works in [26].

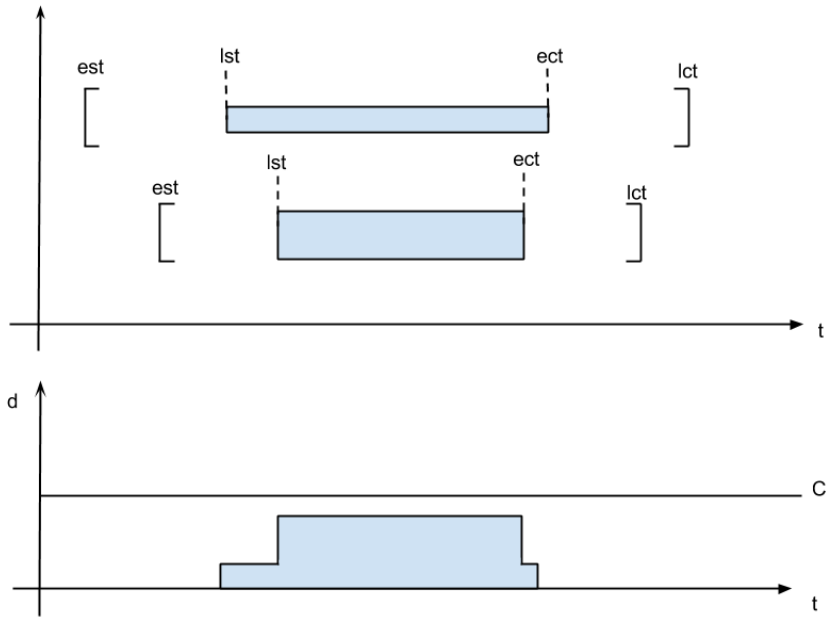


Figure 2.1: Example for the Timetabling algorithm. The first graph records two activities and the duration where they must use the resource. The second graph illustrates the minimum capacity profile.

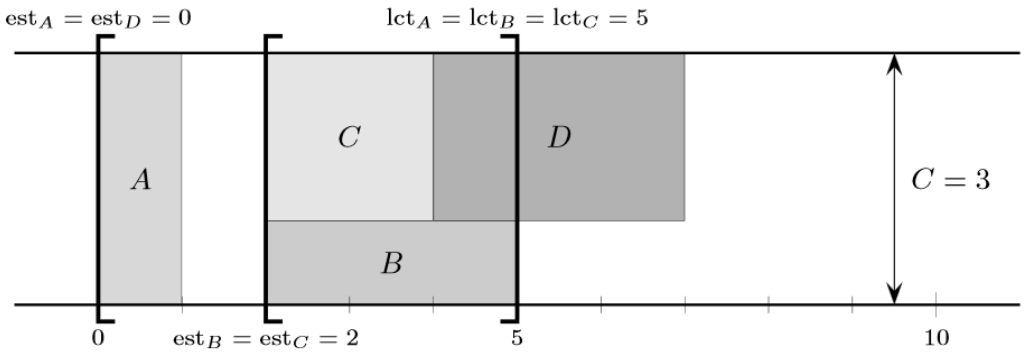


Figure 2.2: Example of Edge Finding from [2]. est_D can be updated from 0 to 4 because there will be an overflow if we schedule D at 0.

Chapter 3
Problem Analysis

Edge Finding filtering gives us some very good results for disjunctive resources. However this is not the case when we apply it to cumulative resources. Indeed Edge Finding doesn't heed activities which overlap a set Ω, therefore the filtration power will be reduced.

Mercier and Van Hentenryck have provided some adjustments to Edge Finding and Extended Edge Finding algorithm to overcome this problem by using dynamic programming. This was later improved by Vilim. [27] [28]

Moreover Vilim himself has recently proposed a new approach to these problems. The idea is to mix Edge Finding filtering with the Timetabling technique in order to build a stronger algorithm. More precisely he modified the propagation rule of Edge Finding in a way that it could use the timetable to do a better pruning. Our task in this part is to use Vilim's notes to implement this algorithm in Oscar.

Chapter 4
Solution

In this chapter we will describe our implementation of the TimeTable Edge-Finding algorithm. To do that we will essentially follow the same steps as Vilim [2] since it's for us the easiest way to understand it. Each part will be illustrated by the code created in Oscar. Some minors changes can occur in our illustration in comparison to the implementation in Oscar in order to facilitate the reading

First we will discuss about activities and some choices of design we made to manipulate them more easily. Then we will concentrate on the Timetable part (how we constructed it, how it will be useful) and see the algorithm for checking infeasibility (the overload checking algorithm). After that we will talk about the full propagation rule and how we use it to update the activities *est*. Finally we will see how we handle symmetry to update the activities *lct*.

4.1 Activities modeling

Propagators in OscaR don't use the classes from the DSL but actual problem variables. The arguments of the propagator are then arrays of `CPIntVar` corresponding to the starts, ends, durations, demands etc. of activities. However these type of arguments are not especially relevant in our algorithm. For example we only use the *est*, *lst*, *ect* and *lct* in the algorithm, and since the *lst* and *ect* can be found by using *lct* and *est* respectively, we only need the *start.min* and *end.max* of all activities. Furthermore we don't care about activities with an empty duration or an empty demand. Loop over those activities won't change anything to the propagation in the best case, and could slow down the propagation in the worst case. So we filter them as soon as possible.

Another important point is that we need a way to keep trace of activities in the algorithm. Indeed we will often sort the activities in different ways and we must ensure that the data of the correct activity will be used. That's

why we decided to add a field `id` to each activity.

Here is our activity class with the following variables : an *est*, a duration, a *lct*, a demand and an *id*.

```
class Activity(s: Int, p: CPIntVar,
  e: Int, d: CPIntVar, private var i: Int) {
    def est = s
    def lct = e
    def dur = p
    def demand = d
    def id = i
  }
```

We then create a method to generate all the useful activities. To avoid having an array bigger than the actual number of activities we compute first a set of id.

```
private val validActivities = (0 until durations.length).filter(
    i => durations(i).min > 0 && demands(i).min > 0)
```

The next method will generate the activities.

```
private def generateActivities(): Unit = {
    nActivities = 0
    for (i <- validActivities){
        activities(nActivities) = new Activity(
  starts(i).min,durations(i),ends(i).max,demands(i),i)
    }
  }
```

Finally, the algorithm often needs to loop over the activities in precise orderings. For example in the adjustment's algorithm we need to loop over activities in an non-increasing order by *est*. We introduced a method to create the arrays we need in the whole propagator : increasing and decreasing order by *est*, increasing order by *est* with tie-breaking, decreasing order by *ect*, decreasing order by *lst*, decreasing order by *lct* and increasing order by *est+duration* of the free part *(see next section for explanation on the free part)*. To do this we created for each order an object ordering which redefine the `compare` method. For example, with tie breaking :

```
object estOrderingTie extends Ordering[Activity]{
  def compare(a:Activity, b:Activity) = {
    val comp =  a.est compare b.est
    if(comp != 0){
      comp
    }
    else{
      b.est + b.freePart compare a.est + a.freePart
    }
  }
}
```

Then in the `generateArrays()` method we clone the array activities and reorder it with `quickSort` :

```
def generateArrays() {
  actEstTie = activities.clone
  Sorting.quickSort(actEstTie)(estOrderingTie)
  ...//Other arrays
}
```

The advantage of using arrays instead of queues is that we can loop over it many times without recomputing it.

4.2 Timetable

We know that Edge Finding uses a set of activities Ω and we want to improve the energy consumption between $[est_\Omega, lct_\Omega]$ by using the timetable. In order to do that, we must compute the fixed part and the free part of each activity. The fixed part of an activity *(named p^{TT})* is the interval in which the activity must be executed. The free part, p^{EF} is the remaining duration. We also define the energy of the free part : $e^EF = d * p^{EF}$. These three variables are added in `Activity` as well as generated with the other variables:

```
class Activity(s: Int, d: CPIntVar, e: Int, dem: CPIntVar,
   fix : Int, fp: Int, efp: Int, private var i: Int) {
    ...//Other variables
    def fixedPart = fix
    def freePart = fp
    def energyFreePart = efp
```

```
  }

 private def generateActivities(): Unit = {
    nActivities = 0
    for (i <- validActivities){
        val fixed = math.max(0, (starts(i).min + durations(i).min) - (
          ends(i).max - durations(i).min))
        val freePart = durations(i).min - fixed
        val energyFreePart = freePart*demands(i).min
        activities(nActivities) = new Activity(
          starts(i).min,durations(i),ends(i).max,demands(i),
            fixed, freePart, energyFreePart,i)
    }
  }
```

The timetable will contain the sum of capacities of all fixed parts at each time t, with $t \in [0, horizon]$. However we mainly need to know how much energy the timetable contains between $[est_\Omega, lct_\Omega]$. This energy can be calculated easily : we take all the energy present in the timetable from est_Ω and we subtract it the energy from lct_Ω. Since all activities can potentially be in Ω, we need two arrays to store the energy from the *est* and the *lct* for each of them, respectively `ttAfterEst` and `ttAfterLct`.

```
private var ttAfterEst = Array.ofDim[Int](nTasks)
private var ttAfterLct = Array.ofDim[Int](nTasks)
```

Computing the whole timetable can be costly. Instead we will only concentrate on the four different events that can occur over time. If we consider time to go backward we can associate those events to the *est*, *lct*, *ect* and *lst* of an activity *i*. At time t we can:

- Compute `ttAfterEst` if $est_i == t$
- Compute `ttAfterLct` if $lct_i == t$
- Add d_i to the timetable if $est_i + p_i == t$
- Remove d_i from the timetable if $lct_i - p_i == t$

To do that we created a class Event with three variables : an activity, an event and the event's date. The events types will respectively be numbered from 0 to 3.

```
private class Event(a: Activity, e: Int, d: Int){
  def act = a
  def event = e
  def date = d
}
```

Then we must generate those events. We first create an array to contains the ones whose size will be four times the number of valid activities since there are four events by activity.

```
private val eventArray = new Array[Event](validActivities.length*4)
```

Next we fill the array with the events and we order it by date in decreasing order with the function `stableSort`

```
 def generateEvent() {
   var i = 0
   var count = 0
   while(i < activities.length){
     eventArray(count) = new Event(tasks(i),0,tasks(i).lct)
     eventArray(count+1) = new Event(tasks(i),1,tasks(i).est)
     eventArray(count+2) = new Event(
       tasks(i),2,tasks(i).est + tasks(i).dur.min)
     eventArray(count+3) = new Event(
   tasks(i),3,tasks(i).lct - tasks(i).dur.min)
     count+=4
     i+=1
   }
   stableSort(eventArray, 0, count, (a: Event, b: Event) => a.date > b.date)
 }
```

Finally we have to explain how to maintain the total energy stored. Between the event we maintain the actual level of demand in a variable called `actDem`. Each time we encounter an event of type 2 or 3 this level is respectively increased or decreased by the activity's demand if the activity has a fixed part. But before each event we must compute the energy between the event and the previous one by simply multiplying the actual demand by the interval. Of course we keep the previous event's date in a variable `prev` that we recompute after each event. This gives the following code to compute the arrays `TTAfterLct` and `TTAfterEst`.

```
def computeTT() = {
  var totalEnergy = 0
  var actDem = 0
  var i = 0
  var prev = 0
  while(i < eventArray.length){
    val a = eventArray(i)
    val act = a.act
    if(prev != 0){
        totalEnergy+=(prev-a.date)*actDem
      }
    a.event match {
      //Event 0 = Lct
      case 0 => ttAfterLct(act.id) = totalEnergy
      //Event 1 = Est
      case 1 => ttAfterEst(act.id) = totalEnergy
      //Event 2 = Ect
      case 2 => if(act.fixedPart > 0) actDem += act.demand.min
      //Event 3 = Lst
      case 3 => if(act.fixedPart > 0) actDem -= act.demand.min
    }
    prev = a.date
    i+=1
  }
}
```

4.3 Overload Checking

We now need a way to check, before running the algorithm, if the total of both fixed and free part don't overflow, i.e, if there is enough capacity during the task intervals to resolve the problem. To do that we compute the remaining energy between $[est_\Omega, lct_\Omega]$. Since the fixed parts are already in $ttAfterEst(\Omega)$ and $ttAfterLct(\Omega)$, we just need a way to add the free parts' energy to the total energy required.

We warn the user that we will not follow Vilim in this section. Indeed, he bases his Overload Checking algorithm only on activities which have a non null free part. However, it will not detect an overflow when the activities have only fixed parts. Let's take for example three activities with respectively demands of 2, 2 and 1, all durations of 2 and a single resource with

capacity of 2. The horizon is assigned to 5. We can easily see that this example is infeasible since we can't perform two activities at the same time without overloading the resource.

Pseudo code of Vilim is the following :

```
//a and b are activities, energy = energy of free parts
if ((b.lct - a.est)*capacity.max - energy - (
  ttAfterEst(a) - ttAfterLct(b)) < 0)
    return fail
```

On the first run there is no fixed part. So `ttAfterEst(a)` and `ttAfterLct(b)` are null. The energy will be 2*2+2*2+2*1=10 and the energy available will be 5*2 = 10. No overload at this point.
On the second run we say that the search assigns the start of the first activity to 0. We add 2 during the interval $[0, 2[$ in the timetable. Therefore, `ttAfterEst` of all activities will be 4. However in the overload checking we will only consider the last two activities. So the energy will be 2*2+2*1 = 6 and the energy available will still be 5*2=10.
On the third run the search assigns the start of the second activity to 0. Again we add 2 during the interval $[0, 2[$ in the timetable and `ttAfterEst` of all activities will be 8. But the overload checking will only consider the last activity and will not detect the overflow.

Therefore we searched over an another Overload Checking algorithm and we found a functional one in a paper of Schutt, Feydy and Stuckey [29]. In this algorithm we first iterate over activities in non-increasing order by *lct* to select the one with the highest *lct*, which will represent lct_Ω. Each time we select a new activity we check if its *lct* differs from the previous one because there is no need to redo the set Ω more than once. Then we initialize the total energy of the free parts to 0 and we iterate over the activities in non-decreasing order by *est*. This allow us to progressively add activities to the set and maintain the total energy consumption of the free parts.

```
def overloadChecking() : CPOutcome = {
    var end = Double.PositiveInfinity
    var y = 0
    while(y < activities.length){
      val b = actLct(y)
      if(b.lct != end){
        end = b.lct
```

```
        var energy = 0
        var x = 0
        while(x < activities.length){
          ...//Task interval computation
          }
          x +=1
        }
      }
      y+=1
    }
    return CPOutcome.Suspend
  }
```

Once an activity is selected we look at its *est* to be sure that it forms a task intervals with b. Then we check if a will finish before b to see if we must take into account the whole energy of a's free part. If not it means that a is on the right with respect to b. Therefore, we only add to the energy the part before the end of b. This is another difference between this algorithm and the Vilim one which does not take into account this last case.

```
//Task interval computation
while(x < activities.length){
  val a = actEstReverse(x)
  if(a.est < end){
    var begin = a.est
    if(a.lct <= end){
      energy += a.energyFreePart
      }
    else if(a.lct-a.freePart < end){
      energy +=  (b.lct - (a.lct-a.freePart))*a.demand.min
      }
...//Overflow checking
}
x+=1
}
```

Finally we check if the reserve is non-negative. However we didn't want to exclude problems with variable capacity. That's why we check the reserve with an minimum capacity. If this leads to an overflow, we try to increase the capacity by one and we redo the check. We continue to increase the capacity until it does not generate an overflow anymore or we reach the maximum capacity, in which case the problem is infeasible

```
  //Overflow checking
while(((end - begin)*capacity.min - energy -ttAfterEst(a.id)
                                          + ttAfterLct(b.id)) < 0){
  if(capacity.updateMin(capacity.min + 1) == CPOutcome.Failure){
    return CPOutcome.Failure
    }
}
```

We can prove that the above algorithm find all the overflow. If we take back the previous example and run it through the algorithm:

On the first run there is no overload since nothing is fixed.
On the second run the algorithm will iterate the activities until it takes the first activity as $[lct_\Omega]$. The total energy required during $[0, 2[$ will be 0*(no free part)* $+ 4$ which equals the energy available *2*2 = 4*
However, on the third run the total energy required will be $0 + 8$ which overflows the energy available. So this algorithm works correctly with all kind of overflows.

We must precise that we will now ignore activities with no free part. Indeed they will never modify the energy consumption anymore since they are already in the timetable. The remaining activities will form the set T^{EF}.

4.4 Time Bound Adjustment

How much additional energy will an activity ι require during $[est_\Omega, lct_\Omega]$ when it overlaps with Ω ? This is the main question of our algorithm. At this point we just know that this energy must be inferior or equal to the reserve of Ω. If not, the activity's *est* must be updated. This section will be dedicated on how we compute this update. Reminder : all the fixed parts are in the timetable, so we only concentrate on the free parts of the activities !

4.4.1 Propagation Rule

First we need to define the propagation rule that will be used. There is two parts in that rule : the condition and the update. Like we said earlier the condition is that the reserve is inferior to the additional energy ($addEn(\iota)$) given by the activity ι. However this energy depends on the position of the activity. So we distinguish four cases:

- Inside Ω: the additional consumption will simply be the energy of the free part of $\iota \rightarrow addEn(\iota) = e_\iota^{EF}$.
- Right of Ω: the consumption will be set by the duration during est_ι and $lct_\Omega \rightarrow addEn(\iota) = (lct_\Omega - est_\iota) * d_i$.
- Left of Ω: this time we don't want the interval between est_ι and est_Ω. Therefore we will take the free part and we subtract it $est_\Omega \rightarrow addEn(\iota) = (est_\iota + p_\iota^{EF} - est_\Omega) * d_\iota$.
- Through Ω: this one is simple, we add the combinaition of the demand of ι with the task interval $\rightarrow addEn(\iota) = (lct_\Omega - est_\Omega) * d_\iota$.

Notice that this decomposition will lead our main algorithm too.

For the update we must consider the maximum duration that the activity can spend during the task interval. To do that we cut the activity in two parts :

- Mandatory part: this is the part of ι which is already in the timetable and which overlaps with the task interval. In other words it's the intersection between the interval and the fixed part, i.e between $[est_\Omega, lct_\Omega]$ and $[lct_\iota - p_\iota, est_\iota + p_\iota]$. We compute it in Oscar by giving est_Ω, lct_Ω and ι to the following function:

```
def mandatoryIn(est : Int, lct : Int, i : Activity) : Int = {
    math.max(0,math.min(lct,i.est + i.dur.min)-
  math.max(est,i.lct-i.dur.min))
  }
```

- Optional part: this is the free part of ι which can be executed during the task interval considering the reserve of Ω:

```
def maxAddIn(reserve: Int, i : Activity) : Int = {
    math.floor(reserve/i.demand.min).toInt
  }
```

The remaining duration must be spent after lct_Ω. Therefore, if ι must be updated, its new *est* will have the following value:

$est_\iota = lct_\Omega - mandatoryIn(est_\Omega, lct_\Omega, \iota) - maxAddIn(reserve, \iota)$

This gives us the following full propagation rule:

$\forall \Omega \subset T^E, \forall \iota \in T^{EF} \backslash \Omega : reserve(\Omega) < addEn(\iota) \rightarrow$
$est_\iota := lct_\Omega - mandatoryIn(est_\Omega, lct_\Omega, \iota) - maxAddIn(reserve, \iota)$

We want to point out that this new *est* will not necessary be consistent with all sets Ω. This is not a problem since TimeTable Edge Finding assume that if there is still an update to do, it will find it out in a next run and update the *est* again.

4.4.2 Adjustment Algorithm

Now that we have the propagation rule we can finally create the main algorithm of the propagator. Foremost we must copy the actual *est* of all activities in a data structure because we don't want to update bounds until the end of the algorithm. Because we will do the update in a separate method to gain clarity, we will also return an array at the end.

```
def estAdjustment() : Array[Int] = {
  var est = Array.ofDim[Int](nTasks)
  var t = 0
  while(t < activities.length){
    if(activities(t).energyFreePart > 0){
      est(activities(t).id) = activities(t).est
    }
    t+=1
  }
  ...//Adjustment's rules
  return est
}
```

Like in the Overload Checking function we will iterate over sets Ω and compute maximum value of $addEn(\iota)$ over all activities that are not in Ω. However this value depends on the relative position of activities and Ω. Therefore we split the algorithm in three phases according to the positions of activities with respect to Ω.

Inside and Right position

In these positions we will iterate over set Ω in the same way as the overload checking algorithm. We first sweep over activities b in non-increasing order

by *lct* and we compute the task interval by iterating over activity a in non-increasing order by *est*. We only need to ensure that it is a task interval by checking a's *est* and b's *lct*.

```
//Adjustment's rules
  var i = 0
var end = Double.PositiveInfinity
  while(i < activities.length){
    val b = actLctReverse(i)
    if(b.energyFreePart > 0 && b!= end){
      var energyEF = 0
      var iota = -1
      var act = 0
      while (act < actEstReverse.length){
        val a = actEstReverse(act)
        if(a.energyFreePart > 0 && a.est < b.lct) {
          ...//Inside case
        }
        ...//Right case
        ...//Update
        act+=1
      }
      ...//Through case
    }
  }
  ...//Left case
```

Now that we have a time interval we have to see if the activity is inside Ω or on the right by checking the *lct*. If it's inside we know by the propagation rule that the additional consumption will be the free part's energy of the activity.

```
  //Inside case
  if(a.energyFreePart > 0 && a.est < b.lct) {
    if(a.lct <= b.lct){
      energyEF += a.energyFreePart
    }
    ...//Right case
    ...//Update
  }
```

If it's in right position, the additional consumption will be the duration between est_a and lct_Ω multiplied by a's demand. However we are not sure that there is no fixed part during this interval. That's why we must take the minimum between it and the actual free part's energy of the activity to ensure that we don't take too much energy. Moreover we search over the maximum additional consumption so we must compare it to the additional consumption of ι. If it's higher, a become the new ι.

```
//Right case
else if(iota == -1 || math.min(
  a.energyFreePart,(b.lct-a.est)*a.demand.min) > math.min(
  iota.energyFreePart,(b.lct-iota.est)*iota.demand.min)){
              iota = act
  }
...//Update
```

After that we compute the reserve with the following function based on the Overload checking rule:

```
def computeReserve(a : Activity, b: Activity, energy: Int) : Int = {
    val capacityOmega = (b.lct - a.est)*capacity.max
    capacityOmega - energy - (ttAfterEst(a.id) - ttAfterLct(b.id))
  }
```

If it's smaller than the additional consumption, we can update the activity's *est* following the propagation rule.

```
 //Update
 var reserve = computeReserve(a,b,energyEF)
   if (iota != -1 && reserve < math.min(
   iota.energyFreePart,(b.lct-iota.est)*iota.demand.min)){
       est(iota.id) = math.max(est(iota.id),b.lct-mandatoryIn(
   a.est,b.lct,iota)- maxAddIn(reserve,iota))
    }
```

Through position

At this point we have swept over all activities. To find the ones which are in the through position we will iterate in reverse order, i.e. in non-decreasing order by *est*. This way we will progressively reduce Ω by removing activities from it. We also break ties by non-increasing $est + p^{EF}$ to have the best chance to quickly update iota *(see later)*.

```
  //Through case
  iota = -1
  act = 0
  while (act < actEstTie.length){
    val a = actEstTie(act)
    if(a.energyFreePart > 0 && a.est < b.lct) {
    ...//Update of est
    }
  ...//Computation of iota
  act+=1
  }
```

Then we check if the activity was effectively part of Ω. If so, we first check if iota's *est* can be updated by computing the reserve and compare it to the additional consumption of iota before removing its free part's energy from the total energy.

```
  //Update of est
  var reserve = computeReserve(a,b,energyEF)
  if (iota != -1 && reserve < (b.lct-a.est)*iota.demand.min){
    est(iota.id) = math.max(est(iota.id),b.lct-mandatoryIn(
  a.est,b.lct,iota)-maxAddIn(reserve,iota))
    }
  energyEF -= a.energyFreePart
}
```

Wheter the activity was in Ω or not we must check that it is in through position by looking if it exceeds b's *lct*. That's why we break ties with $est + p^{EF}$: at the same *est* we select the activity that has the highest free part: it will be the one which add the most energy. Then we finally check if the additional consumption of a is higher than the one of iota. If it's true, we change iota to a.

```
  //Computation of iota
  if(a.est + a.freePart >= b.lct && (
    iota == -1 || a.demand.min > iota.demand.min)){
      iota = act
  }
```

Left Position

This part is a bit more tricky because we do the opposite of above. We first sweep over activities a in an non-decreasing order by *est*. Then we

iterate over activity b in non-decreasing order by *est* in order to gradually add activities b into Ω if their *est* is higher than a's *est*.

```
  //Left case
var begin = Double.NegativeInfinity
  var j = 0
  while(j < activities.length){
    val a =  activities(j)
    if(a.energyFreePart > 0 && a != begin){
      var energyEF = 0
      var iota = -1
      var act2 = 0
      while (act2 < actEst.length){
        var act = 0
        if(a.est <= b.est){
          ...//Computation of iota
        }
        ...//Update
        act2+=1
      }
    }
  }
```

After that we add its free part's energy to the total energy. Now that we have Ω we must check the activities to see if they are in left position vis-a-vis the task interval. But instead of checking all of them we can check only those which $est + p^{EF}$ don't exceed b's *lct*. This can be done easily by ordering the activities according to this order. If the activity pass the check we finally compare its additional consumption to the one of ι according to the propagation rule.

```
  //Computation of iota
  val i = actEstFreePart(act)
  if(i.freePart > 0){
    if(i.est < a.est && a.est < i.est + i.freePart &&
      (iota == -1 || ((i.est + i.freePart - a.est)*i.demand.min >
        (iota.est + iota.freePart - a.est)*iota.demand.min))){
          iota = act
    }
  }
  act+=1
```

We end this part by the usual computation of the reserve and the update of activity's *est* if needed.

```
//Update
var reserve = computeReserve(a,b,energyEF)
if(iota != -1 && reserve < (
iota.est + iota.freePart - a.est)*iota.demand.min){
    est(iota.id) = math.max(est(iota.id),b.lct-mandatoryIn(
      a.est,b.lct,iota)-maxAddIn(reserve,iota))
}
```

4.4.3 Update of est and ect

Now that we have an array with the updated *est* we can prune the activities' domains. We will use the activities' id to ensure that we update each activity with the correct *est*. If the update leads to a failure, i.e. if the new *est* is not in the domain of the activity, then we return a failure. We do the same thing to update activities' *ect*.

```
def updateBounds(est: Array[Int]): CPOutcome = {
  var t = 0
  var task = 0
  while(t < activities.length){
    if (activities(t).energyFreePart > 0){
      task = activities(t).id
      if (starts(task).updateMin(est(task)) == CPOutcome.Failure){
          return CPOutcome.Failure
      }
      if (ends(task).updateMin(
           est(task) + durations(task).min) == CPOutcome.Failure){
          return CPOutcome.Failure
      }
    }
    t+=1
  }
  return CPOutcome.Suspend
}
```

4.5 Setup and propagate method

This is a reminder of the two usual method used in Oscar to launch the propagator. The `propagate` method regroup all the methods we've seen before.

```
override def setup(l: CPPropagStrength): CPOutcome = {
  priorityL2 = 0
  val oc = propagate()
  if (oc == CPOutcome.Suspend) {
    for (i <- Tasks) {
      if (!starts(i).isBound){
        starts(i).callPropagateWhenBoundsChange(this)
      }
      if (!durations(i).isBound){
        durations(i).callPropagateWhenBoundsChange(this)
      }
      if (!ends(i).isBound){
        ends(i).callPropagateWhenBoundsChange(this)
      }
      if (!demands(i).isBound){
        demands(i).callPropagateWhenBoundsChange(this)
      }
      if (!resources(i).isBound)
        resources(i).callPropagateWhenDomainChanges(this)
      }
    }
    capacity.callPropagateWhenBoundsChange(this)
  }
  return oc
}

override def propagate(): CPOutcome = {
  generateActivities
  generateArrays(activities)
  generateEvent(activities)
  computeTT

  if (overloadChecking == CPOutcome.Failure){
    return CPOutcome.Failure
  }
```

```
    val est = estAdjustment(activities)

    if(updateBounds(est) == CPOutcome.Failure){
      return CPOutcome.Failure
    }
    return CPOutcome.Suspend
  }
```

4.6 Update of Lct

The above algorithm only updates *est* and *ect*, but we can use it to update *lct* and *lst* by feeding it with symmetrical data, i.e. the inverse of the *est* and *lct*. For example if an activity had an *est* at 1 and a *lct* at 5, we consider now that it has an *est* at -5 and a *lct* at -1. This way we don't have to rewrite all the methods and we can simply use them by operating some slight changes.

- We have to create another array with the opposite activities. This is done at the same time with the regular activities.

```
private def generateActivities(): Unit = {
  nActivities = 0
  for (i <- validActivities){
...//Regular activities' computation
  val fixedOpp = math.max(0, ((-ends(i).max) + durations(i).min) - (
    (-starts(i).min) - durations(i).min))
  val freePartOpp = durations(i).min - fixedOpp
  val energyFreePartOpp = freePartOpp*demands(i).min
  activitiesOpp(nActivities) = new Activity(
    (-ends(i).max),durations(i),(-starts(i).min),
      demands(i),fixedOpp, freePartOpp, energyFreePartOpp,i)
    nActivities += 1
  }
}
```

- Then we can feed the method `generateArrays()` with this new data structure. For example :

```
def generateArrays(tasks : Array[Activity]) {
  actEst = tasks.clone
  Sorting.quickSort(actEst)(estOrdering)
```

```
    ...//Other arrays
}
```

- We also feed the `estAdjustment()` algorithm with the array of symmetrical activities:

```
def estAdjustment(tasks : Array[Activity]) : Array[Int] = {
  var est = Array.ofDim[Int](nTasks)
  var t = 0
  while(t < validActivities.length){
    if(tasks(t).energyFreePart > 0){
      est(tasks(t).id) = tasks(t).est
    }
    t+=1
  }
...//Adjustment's rules
```

Therefore `estAdjustment()` will give us an array of negative *est*. To update the *lst* and *lct* we have to sweep over the array of symmetric activities and update them with the inverse of the array.

```
def updateBounds(est: Array[Int], lct:Array[Int]): CPOutcome = {
  var t = 0
  var task = 0
  while(t < validActivities.length){
    ...//est and ect adjustment
    if(activitiesOpp(t).energyFreePart > 0){
      task = activitiesOpp(t).id
      if (ends(task).updateMax(-lct(task)) == CPOutcome.Failure){
        return CPOutcome.Failure
      }
      if (starts(task).updateMax(
        -lct(task) - durations(task).min) == CPOutcome.Failure){
          return CPOutcome.Failure
      }
    }
    t+=1
  }
  return CPOutcome.Suspend
}
```

- Finally here is the actual `propagate()` method :

```
override def propagate(): CPOutcome = {
    generateActivities
    generateArrays(activities)
    generateEvent(activities)
    computeTT

    if (overloadChecking == CPOutcome.Failure){
      return CPOutcome.Failure
    }
    val est = estAdjustment(activities)

    generateArrays(activitiesOpp)
    generateEvent(activitiesOpp)
    computeTT

    if (overloadChecking == CPOutcome.Failure){
      return CPOutcome.Failure
    }

    val lct = estAdjustment(activitiesOpp)

    if(updateBounds(est, lct) == CPOutcome.Failure){
      return CPOutcome.Failure
    }
    return CPOutcome.Suspend
  }
```

Chapter 5
Validation

The validation of the propagator is done in two parts. First we test it through the `TestCumulativeConstraint` class, then we test it on the open instances of the RCPSP problem from PSPLIB. [30]

5.1 TestCumulativeConstraint

The `TestCumulativeConstraint` class is part of Oscar. We extended it to use `TTEdgeFinding` instead of `SweepMaxCumulative`. It generates a hundred random instances and launch them twice: once with the chosen propagator and the other with the `CumulativeDecomp` propagator, which returns the right results. It then compares the set of solutions of the two propagators for each instance, and return a failure if they differ. It is possible to choose the size of the generated instances.

To extends this class we created the `TestTTEdgeFinding` class which can be found in the `oscar.cp.test.memScheduling` package. We then simply overrided the `cumulative` method of the superclass to launch our propagator instead of the cumulative one.

We launched the tests ten times on each instances' size between 3 and 7. Beyond this size, the tests took too much time because the set of solutions for one instance can be over several millions. All tests are successful.

It's a good start for our propagator but we must emphasize that all the instances has a single resource with no other constraint than the resource's capacity. Therefore, we must test it deeper with more complicated instances.

To do this, we modeled RCPSP problems from PSPLIB. [30]

5.2 RCPSP problem

RCPSP problem has already been defined in the first part of this document. Our main idea is to compare our propagator to the `SweepMaxCumulative` one in terms of solutions founds, traversed nodes, number of fails and speed by using three different searches : `binaryFirstFail`, `binaryStatic` and `setTimes`. We first run 20 problems of the J30 instances, then 20 problems of the J60 instances. We also set an arbitrary time limit of two minutes. We expect that our propagator will be slower than the cumulative one, because of the presence of two nested cycles in the `estAdjustment`. However, the pruning could be better, resulting in a lesser number of nodes and failures in a majority of instances. The results of the three different searches can be found in tables 5.1 to 5.4. Note that there are no results for `setTimes` because our propagator never got to a solution using it, regardless of the instances used.

5.3 Analyzis of the results

The implementation of the TimeTable Edge Finding propagator was not really a success given the results we have obtained. With the `binaryFirstFail` search, our propagator doesn't reach the same bound as the cumulative one in more than 50% of cases. Moreover, in those cases the search exceeds the time limit. However, when the bounds are the same, we see that our propagator is the slowest, like we expected. But it also has bigger number of nodes and fails while we expected the opposite.

It was a bit better with the `binaryStatic` search. We can see that the bound is the same in more than one instance out of two. But it has also some strange behavior since, even if the search has found the optimal bound, it cannot terminate itself. Finally the `setTimes` search was by far the worst one since it gave us no result at all.

It is pretty hard to find the error since a little mistake can have big consequences in the constraint propagation. However, it seems that the propagtor rely to much on the `overloadChecking` algorithm, probably due to a bad pruning computation in the `estAdjustment` method. However, our implementation cannot be outrageously wrong because we pass the test of the `TestCumulativeConstraint` class while very early implementations of our propagator did not, so it is showing we are on the right track.

Moreover, we analyzed the propagator many times and meticulously searched for errors step by step in the algorithm through a lot of break points. The only suspect behavior we've seen was a huge amount of fail within the `overloadChecking` algorithm on some instances. But this corresponds to the above analysis. To be sure, we tested it with different algorithms and had the same problem.

We also computed another adjustment's algorithm detailed in the same paper as our Overload Checking's implementation in [29]. This way we thought that if we've done a small mistake in the previous one, it will be corrected in the new one. However, this resulted in the same behavior and results as the Vilim's one. We therefore kept the basic's one.

Instances	J301_1	J301_2	J302_1	J302_2	J303_1	J303_2	J304_1
Bounds(TTEF)	48	47	41	56	72	40	49
Bounds(Cumul)	43	47	38	51	72	40	49
Nodes(TTEF)	/	/	/	/	163	97160	64
Nodes(Cumul)	124	116	110	164	64	150	64
Fail(TTEF)	/	/	/	/	69	48581	33
Fail(Cumul)	63	59	56	83	33	76	33
Time(TTEF)	>120	>120	>120	>120	0.2	9.6	0.15
Time(Cumul)	0.18	0.2	0.19	0.23	0.08	0.16	0.08

Instances	J304_2	J305_1	J305_2	J306_1	J306_2	J307_1	J307_2
Bounds(TTEF)	60	80	83	67	/	55	42
Bounds(Cumul)	60	59	82	59	51	55	42
Nodes(TTEF)	62	/	/	/	/	122	142
Nodes(Cumul)	62	/	448006	276200	6010	64	62
Fail(TTEF)	33	/	/	/	/	62	72
Fail(Cumul)	33	/	229014	138101	3006	33	32
Time(TTEF)	0.15	>120	>120	>120	>120	0.23	0.27
Time(Cumul)	0.09	>120	109.74	31.58	1.47	0.1	0.1

Instances	J308_1	J308_2	J309_1	J309_2	J3010_1	J3010_2
Bounds(TTEF)	44	51	/	/	52	85
Bounds(Cumul)	44	51	90	107	42	56
Nodes(TTEF)	64	64	/	/	/	/
Nodes(Cumul)	64	62	/	/	12414	41394
Fail(TTEF)	33	33	/	/	/	/
Fail(Cumul)	33	32	/	/	6208	20698
Time(TTEF)	0.17	0.17	>120	>120	>120	>120
Time(Cumul)	0.08	0.09	>120	>120	2.18	4.22

Table 5.1: Results for BinaryFirstFrail on J30 instances.

Instances	J601_1	J601_2	J602_1	J602_2	J603_1	J603_2	J604_1
Bounds(TTEF)	/	/	65	82	60	69	84
Bounds(Cumul)	77	74	65	82	60	69	84
Nodes(TTEF)	/	/	584	1354	164	264	124
Nodes(Cumul)	202	/	128	124	124	122	124
Fail(TTEF)	/	/	293	678	83	133	63
Fail(Cumul)	102	/	65	63	63	62	63
Time(TTEF)	>120	>120	0.50	0.82	0.29	0.3	0.38
Time(Cumul)	0.2	>120	0.15	0.14	0.14	0.15	0.13

Instances	J604_2	J605_1	J605_2	J606_1	J606_2	J607_1	J607_2
Bounds(TTEF)	60	108	178	/	74	/	85
Bounds(Cumul)	60	84	118	66	67	77	85
Nodes(TTEF)	124	/	/	/	/	/	278
Nodes(Cumul)	122	/	/	/	6752	122	122
Fail(TTEF)	63	/	/	/	/	/	140
Fail(Cumul)	62	/	/	/	3377	62	62
Time(TTEF)	0.3	>120	>120	>120	>120	>120	0.5
Time(Cumul)	0.15	>120	>120	>120	2.4	0.21	0.2

Instances	J608_1	J608_2	J609_1	J609_2	J6010_1	J6010_2
Bounds(TTEF)	64	61	/	119	/	65
Bounds(Cumul)	64	61	97	102	85	62
Nodes(TTEF)	124	208	/	/	/	88626
Nodes(Cumul)	120	124	/	/	114	212
Fail(TTEF)	63	105	/	/	/	44315
Fail(Cumul)	61	63	/	/	58	107
Time(TTEF)	0.34	0.4	>120	>120	>120	97.33
Time(Cumul)	0.19	0.2	>120	>120	0.21	0.28

Table 5.2: Results for BinaryFirstFrail on J60 instances.

Instances	J301_1	J301_2	J302_1	J302_2	J303_1	J303_2	J304_1
Bounds(TTEF)	49	51	38	51	72	40	49
Bounds(Cumul)	43	47	38	51	72	40	49
Nodes(TTEF)	/	/	/	/	11196	/	64
Nodes(Cumul)	198	/	128	72	114	64	64
Fail(TTEF)	/	/	/	/	5599	/	33
Fail(Cumul)	100	/	65	37	58	33	33
Time(TTEF)	>120	>120	>120	>120	11.57	>120	0.16
Time(Cumul)	0.25	>120	0.18	0.15	0.09	0.09	0.08

Instances	J304_2	J305_1	J305_2	J306_1	J306_2	J307_1	J307_2
Bounds(TTEF)	60	62	85	64	56	55	42
Bounds(Cumul)	60	59	84	59	54	55	42
Nodes(TTEF)	64	/	/	/	/	92	148
Nodes(Cumul)	64	/	/	78258	/	64	64
Fail(TTEF)	33	/	/	/	/	47	75
Fail(Cumul)	33	/	/	39130	/	33	33
Time(TTEF)	0.15	>120	>120	>120	>120	0.16	0.23
Time(Cumul)	0.09	>120	>120	8.59	>120	0.1	0.09

Instances	J308_1	J308_2	J309_1	J309_2	J3010_1	J3010_2
Bounds(TTEF)	44	51	100	109	55	60
Bounds(Cumul)	44	51	95	109	42	56
Nodes(TTEF)	64	64	/	/	/	/
Nodes(Cumul)	64	64	/	/	234	41394
Fail(TTEF)	33	33	/	/	/	/
Fail(Cumul)	33	33	/	/	118	20698
Time(TTEF)	0.16	0.17	>120	>120	>120	>120
Time(Cumul)	0.08	0.09	>120	>120	0.35	>120

Table 5.3: Results for BinaryStatic on J30 instances.

Instances	J601_1	J601_2	J602_1	J602_2	J603_1	J603_2	J604_1
Bounds(TTEF)	80	84	65	82	62	69	84
Bounds(Cumul)	77	73	65	82	60	69	84
Nodes(TTEF)	/	/	388	238	/	222	124
Nodes(Cumul)	196	/	128	124	192	124	124
Fail(TTEF)	/	/	195	120	/	112	63
Fail(Cumul)	99	/	65	63	91	63	63
Time(TTEF)	>120	>120	0.43	0.35	>120	0.3	0.23
Time(Cumul)	0.2	>120	0.15	0.14	0.19	0.13	0.12

77 73 65 82 60 69 84

Instances	J604_2	J605_1	J605_2	J606_1	J606_2	J607_1	J607_2
Bounds(TTEF)	60	98	120	66	1	77	91
Bounds(Cumul)	60	91	119	60	67	77	85
Nodes(TTEF)	124	/	/	/	/	430	/
Nodes(Cumul)	122	/	/	21358	277912	172	364
Fail(TTEF)	63	/	/	/	/	216	/
Fail(Cumul)	62	/	/	10680	138957	87	83
Time(TTEF)	0.24	>120	>120	>120	>120	0.5	>120
Time(Cumul)	0.15	>120	>120	4.57	89.65	0.18	0.27

Instances	J608_1	J608_2	J609_1	J609_2	J6010_1	J6010_2
Bounds(TTEF)	64	61	106	107	85	69
Bounds(Cumul)	64	61	100	98	85	62
Nodes(TTEF)	124	124	/	/	6718	/
Nodes(Cumul)	124	124	/	/	336	792
Fail(TTEF)	63	63	/	/	3360	/
Fail(Cumul)	63	63	/	/	169	397
Time(TTEF)	0.31	0.24	>120	>120	6.18	>120
Time(Cumul)	0.16	0.17	>120	>120	0.27	0.58

Table 5.4: Results for BinaryStatic on J60 instances.

Chapter 6
Future Work

In this chapter we will discuss some improvements that could be made to the propagator.

Obviously the main improvement would be to find why the propagator does not have the expected behavior. We believe we have described our algorithm in enough details that someone else could analyze it and maybe find the source of the problem. Because we have explored all the tracks that we had in mind, we won't be able to give more precision on where to look, however.

The `updatingBounds` method could also be improved. Instead of sweeping over all activities we could only sweep over the ones that have actually been modified. This could be done by computing an array containing the activities' id whose *est* has been modified in the `estAdjustment` method. In fact this was part of our implementation but we had to remove it at some point due to problems with symmetric activities.

Finally, Vilim has proposed an improvement to the `estAdjustment` method in [2]. The idea is the following: est_i is updated iff it causes an overload in $[est_\Omega, lct\Omega]$. So there exists at least one activity within Ω which must end before i can start. Therefore we define the minimum $est_j + p_j$ over all activities j $\in \Omega$ and we say that the new est_i must be bigger than this minimum. This new version could produce more pruning in theory. However, we didn't implement it because of the difficulty we already had with the standard one.

Conclusion

Because it is divided in two parts, we thought it may be interesting to provide a general conclusion to our thesis. We start with the filtering algorithm because it is the one we just discussed.

Unfortunately, we were unable to meet our goal with the algorithm. We searched in vain for the problem for months. We even restarted the development at some point, but it didn't help. However, we are not that far from the cumulative one. Some instances almost provide the same result so it is not completely wasted. By describing our algorithm in the very details, we hope that someone else may be able to find the mistake we made that eluded us until now. We would appreciate if such person could contact us upon finding our error, because we really have been searching for it very hard so we would be really interested in finding what it was.

On the DSL side, we think we ended up with a lot more features that we were planning to. These were primarily language simplification features than scheduling ones, though. We think having clear, readable scheduling models is as important as being able to model problems itself. Besides, we are very proud of our trait-based design. It can easily be expanded on, and we hope it will find uses in the future to integrate more and more scheduling features into a nice clean DSL for OscaR.

Part III

Appendix

Appendix A

Scheduling Models Source Code

To regain a bit of really needed horizontal space, we unindented the code of the models inside the scope of the model object.

The codes of the following pages each comprise multiple sections. The Parsing, Modeling and Searching sections are always present, and often is the Visualizing one.

To be fair when comparing with other scheduling models, one would only compare the Modeling section. The Parsing and Visualizing ones are additional features of our DSL.

The following pages contain the whole code for running instances of the following problems:

- Job Shop Scheduling Problem
- Resource Constrained Scheduling Problem
- Single-Machine Weighted Tardiness Problem
- Common Due Date Problem
- Aircraft Landing Problem
- Shift Minimization Scheduling Problem

```
object JobShop extends SchedulingModel with Reader with Visualization {

// Parsing
read fileFrom "data/memScheduling/jobshop/ft10"
val Seq(nbJobs, nbTasks, nbMachines)  = read fileFor 3 as Int
val Seq(jobs, required, durations) = read fileFor 3 columnsOf Int

// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val machines   = UnaryResources(nbMachines)

for ((a, r) <- (activities, machines(required))) {
  a needs r
}

for (i <- 0 until jobs.length-1 if jobs(i) == jobs(i+1)) {
  activities(i) precedes activities(i+1)
}

// Visualizing
visualization withTitle "OscaR Example: JobShop"
visualization show machines withTitle "Machines"
visualization show activities by jobs withTitle "Jobs"

// Searching
scheduler minimize makespan search {
  setTimes(activities.starts, activities.durations, activities.ends)
} start()
}
```

Figure A.1: The Job Shop Scheduling Problem

```
object RCPSP extends SchedulingModel with Reader with Visualization {

// Parsing
read fileFrom "data/memScheduling/RCPSP/J301_1.RCP"
val Seq(nActivities, nResources) = read fileFor 2 as Int
val capaResources     = read fileFor 4 as Int
val Seq(durations)    = read fileFor nActivities x 1 asColumnsOf Int
val allDemands        = read fileFor nActivities x 4 asLinesOf Int
val Seq(nPrecedences) = read fileFor nActivities x 1 asColumnsOf Int
val precedences = read fileFor nActivities x nPrecedences asLinesOf Int

// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val machines = CumulativeResources ofCapacities capaResources

for ((a, demands, preceded) <- (activities, allDemands, precedences)) {
  for (b <- preceded) {
    a precedes activities(b-1)
  }

for ((d, m) <- (demands, machines)) {
    a needs d of m
  }
}

// Visualizing
visualization zoomedBy 10 withTitle "OscaR Example: RCPSP"
visualization show machines

// Searching
scheduler minimize(makespan) search {
  binaryFirstFail(activities.map(_.start))
} start()
}
```

Figure A.2: The Resource Constrained Scheduling Problem (RCPSP)

```
object WeightTard extends SchedulingModel with Reader with Visualization {

// Parsing
read fromFile "data/memScheduling/weighted-tardiness/wt40_1"
val nbTasks   = 40
val durations = read fileFor nbTasks as Int
val weights   = read fileFor nbTasks as Int
val dueDates  = read fileFor nbTasks as Int

// Modeling
scheduler horizon = durations.sum
val activities = Activities ofDurations durations
val singleMachine = UnaryResource()

for ((a, d) <- (activities, dueDates)) {
  a needs singleMachine
  a isDueAt d
}

val weightedTardiness = weightedSum(weights, activities.map(_.tardiness))

// Searching
scheduler minimize weightedTardiness
search {
  setTimes(activities.starts, activities.durations, activities.ends)
} start()
}
```

Figure A.3: The Single-Machine Weighted Tardiness Problem

```
object CommonDueDate extends SchedulingModel with Reader with Visualization {

// Parsing
read fileFrom "data/memScheduling/common-due-date/sch10/cdd10_10.txt"
val nbJobs = read fileFor Int
val Seq(durations, earlyCost, lateCost) = read fileFor 3 columnsOf Int

// Modeling
val h = 0.4 // External parameter, to be set by user.
val dueDate = (durations.sum * h).toInt

scheduler horizon = dueDate + durations.sum
val activities = Activities ofDurations durations
val resource   = UnaryResource()

for (a <- activities) {
  a needs resource
  a isDueAt dueDate
}

val weightedEarliness = weightedSum(earlyCost, activities.map(_.earliness))
val weightedTardiness = weightedSum(lateCost,  activities.map(_.tardiness))

// Visualizing
visualization withTitle "OscaR Example: CommonDueDate"
visualization show activities by activities.ids
visualization show dueDate
visualization zoomedBy 5

// Searching
scheduler minimize(weightedEarliness + weightedTardiness)
onSolution{
  activities.map(println)
  println
} search {
  setTimes(activities.starts, activities.durations, activities.ends)
} start()
}
```

Figure A.4: The Common Due Date Problem

```
object AircraftLanding extends SchedulingModel with Reader with Visualization {

// Parsing
read fileFrom "data/memScheduling/aircraft-landing/airland1.txt"
val (nbPlanes, freezeTime) = read fileFor (Int, Int)
val planes = read fileFor nbPlanes withPattern
                (Int, Int, Int, Int, Double, Double, nbPlanes as Int)
val (appearances, earliests, targets, latests,
     earlyCosts, lateCosts, delays) = planes

// Modeling
scheduler horizon = latests.max
val landings = Activities(nbPlanes) ofDuration 0

for ((landing, min, max, target) <- (landings, earliests, latests, targets)) {
  landing startsBetween min and max
  landing isDueAt target
}

// For each landing and its associated delays
for ((i, delays_i) <- (landings, delays)) {
  // For each other landing and its associated delay with respect to i
  for ((j, d) <- (landings, delays_i) if (j != i)) {
    add((i isPreceding j) ==> (i isStartingAtLeast d beforeStartOf j))
  }
}

val elc = earlyCosts.map(c => (c*100).toInt) //We need to convert the costs
val llc = lateCosts.map(c => (c*100).toInt)  //because Oscar needs integers.

val weightedEarliness = weightedSum(elc, landings.map(_.earliness))
val weightedTardiness = weightedSum(llc, landings.map(_.tardiness))

// Visualizing
visualization show landings by landings.ids zoomedBy 3
targets.foreach(t => visualization show t)

// Searching
scheduler minimize(weightedEarliness + weightedTardiness) search {
  binaryFirstFail(landings.starts)
} onSolution {
  landings foreach println
} start()
}
```

Figure A.5: The Aircraft Landing Problem

```
object ShiftMini extends SchedulingModel with Reader with Visualization {

// Parsing
read fromFile "data/memScheduling/shift-minimization/data_1_23_40_66.dat"
read dropping 4 lines;
read dropping 2 words
val nbJobs = read fileFor Int
val (starts, ends) = read fileFor nbJobs withPattern (Int, Int)
read dropping 2 words
val nbShifts = read fileFor Int
val nbQualifiedJobs = (read fileFor column).map(_.replaceAll("[:]", "").toInt)
val qualifiedJobs = read fileFor nbShifts x nbQualifiedJobs asLinesOf Int

// Modeling
scheduler horizon = 1440
val shifts = Activities(nbShifts) ofDuration 1440
val workers = CumulativeResources(nbShifts) ofCapacity 0
for ((shift, worker) <- (shifts, workers)) {
  shift optional = true
  shift gives 1 of worker
}

val jobs = (starts, ends).map((s, e) => Activity ofDuration e-s)
for (i <- 0 until nbJobs) {
  jobs(i) startsAt starts(i)
  jobs(i) endsAt ends(i)
  val qualified = (0 until nbShifts).filter(j => qualifiedJobs(j).contains(i))
  val demand = alternatively(Seq.fill(qualified.length)(1))
  jobs(i) needs demand of alternatively(workers(qualified))
}

val nbShiftUsed = sum(shifts.map(s => s ifScheduled 1 ifNot 0))

// Searching
scheduler minimize nbShiftUsed
search {
  binaryFirstFail(jobs.requiredResources :+ nbShiftUsed)
} onSolution {
  shifts foreach println
} start()
}
```

Figure A.6: The Shift Minimization Personnel Task Scheduling Problem

Bibliography

[1] J.-N. Monette, Y. Deville, and P. Van Hentenryck, "Aeon: Synthesizing scheduling algorithms from high-level models," in *Operations Research and Cyber-Infrastructure* (J. Chinneck, B. Kristjansson, and M. Saltzman, eds.), vol. 47 of *Operations Research/Computer Science Interfaces*, pp. 43–59, Springer US, 2009.

[2] P. Vilím, "Timetable edge finding filtering algorithm for discrete cumulative resources," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 230–245, Springer, 2011.

[3] IBM, "Where to find the CP examples," *IBM ILOG ODM Entreprise Information Center*, http://pic.dhe.ibm.com/infocenter/odmeinfo/v3r5/index.jsp, last visited 2014-05-29.

[4] Dynamic Decision Technologies Inc., *Comet Tutorial*. 2010.

[5] P. Laborie and D. Godard, "Self-adapting large neighborhood search: Application to single-mode scheduling problems," *Proceedings MISTA-07, Paris*, pp. 276–284, 2007.

[6] R. H. Storer, S. D. Wu, and R. Vaccari, "New search spaces for sequencing problems with application to job shop scheduling," *Management Science*, vol. 38, no. 10, pp. 1495–1509, 1992.

[7] M. Perregaard, "Branch and bound method for the multiprocessor jobshop and flowshop scheduling problem," *Master's Thesis, Department of Computer Science, University of Copenhagen*, 1995.

[8] C. L. Pape and P. Baptiste, "An experimental comparison of constraint-based algorithms for the preemptive job-shop scheduling problem," 1997.

[9] P. Baptiste, M. Flamini, and F. Sourd, "Lagrangian bounds for just-in-time job-shop scheduling," *Computers & Operations Research*, vol. 35, no. 3, pp. 906 – 915, 2008. Part Special Issue: New Trends in Locational Analysis.

[10] P. Brucker and O. Thiele, "A branch & bound method for the general-shop problem with sequence dependent setup-times," *Operations-Research-Spektrum*, vol. 18, no. 3, pp. 145–161, 1996.

[11] J.-S. Chen, J. C.-H. Pan, and C.-K. Wu, "Hybrid tabu search for re-entrant permutation flow-shop scheduling problem," *Expert Systems with Applications*, vol. 34, no. 3, pp. 1924–1930, 2008.

[12] E. Danna and L. Perron, "Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs," in *Principles and Practice of Constraint Programming–CP 2003*, pp. 817–821, Springer, 2003.

[13] P. Brucker, S. Heitmann, and J. Hurink, "Flow-shop problems with intermediate buffers," *OR Spectrum*, vol. 25, no. 4, pp. 549–574, 2003.

[14] P. Brucker, J. Hurink, B. Jurisch, and B. Wöstmann, "A branch & bound algorithm for the open-shop problem," *Discrete Applied Mathematics*, vol. 76, no. 1, pp. 43–59, 1997.

[15] R. Kolisch and A. Sprecher, "Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program," *European Journal of Operational Research*, vol. 96, no. 1, pp. 205–216, 1997.

[16] M. Mori and C. C. Tseng, "A genetic algorithm for multi-mode resource constrained project scheduling problem," *European Journal of Operational Research*, vol. 100, no. 1, pp. 134–141, 1997.

[17] N. Policella, X. Wang, S. F. Smith, and A. Oddi, "Exploiting temporal flexibility to obtain high quality schedules," in *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 20, p. 1199, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005.

[18] M. Vanhoucke, E. Demeulemeester, and W. Herroelen, "An exact procedure for the resource-constrained weighted earliness–tardiness project scheduling problem," *Annals of Operations Research*, vol. 102, no. 1-4, pp. 179–196, 2001.

[19] F. Sourd and S. Kedad-Sidhoum, "An efficient algorithm for the earliness-tardiness scheduling problem," *Optimisation Online,(1205)*, 2005.

[20] D. Biskup and M. Feldmann, "Benchmarks for scheduling on a single machine against restrictive and unrestrictive common due dates," *Computers & Operations Research*, vol. 28, no. 8, pp. 787–801, 2001.

[21] I. I. O. Studio, "Ilog scheduler 6.1 users manual," *France: ILOG Corporation*, 2005.

[22] J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha, and D. Abramson, "Scheduling aircraft landings—the static case," *Transportation science*, vol. 34, no. 2, pp. 180–197, 2000.

[23] M. Krishnamoorthy, A. Ernst, and D. Baatar, "Algorithms for large scale shift minimisation personnel task scheduling problems," *European Journal of Operational Research*, vol. 219, no. 1, pp. 34–48, 2012.

[24] S. Judkins, "Dynamic mixin in scala - is it possible?," http://stackoverflow.com/questions/3254232/dynamic-mixin-in-scala-is-it-possible?answertab=active, 2014-05-25.

[25] A. Letort, M. Carlsson, and N. Beldiceanu, "A synchronized sweep algorithm for the k-dimensional cumulative constraint," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 144–159, Springer, 2013.

[26] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling: applying constraint programming to scheduling problems*, vol. 39. Springer, 2001.

[27] L. Mercier and P. Van Hentenryck, "Edge finding for cumulative scheduling," *INFORMS Journal on Computing*, vol. 20, no. 1, pp. 143–153, 2008.

[28] P. Vilím, "Edge finding filtering algorithm for discrete cumulative resources in o(kn log n)," in *Principles and Practice of Constraint Programming-CP 2009*, pp. 802–816, Springer, 2009.

[29] A. Schutt, T. Feydy, and P. J. Stuckey, "Explaining time-table-edge-finding propagation for the cumulative resource constraint," in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 234–250, Springer, 2013.

[30] "Project scheduling problem library," http://www.om-db.wi.tum.de/psplib/datasm.html, 2014-05-20.