Detailed Scheduling in IBM ILOG CPLEX Optimization Studio

Chapter · October 2011

CITATIONS
0

READS
8,945

1 author:

Turgay Türker
Sakarya University
7 PUBLICATIONS **31** CITATIONS

SEE PROFILE

Detailed Scheduling in IBM ILOG CPLEX Optimization Studio with IBM ILOG CPLEX CP Optimizer

Contents

Introduction

IBM® ILOG® CPLEX® Optimization Studio and the embedded IBM ILOG CPLEX CP Optimizer constraint programming engine provide specialized keywords and syntax for modeling detailed scheduling problems.

This white paper introduces the concepts involved in describing a detailed scheduling problem and the Optimization Programming Languague (OPL) keywords that facilitate modeling such a problem.

About IBM ILOG OPL CPLEX Optimization Studio

CPLEX Optimization Studio supports the rapid development, deployment and maintenance of mathematical programming (MP) and constraint programming (CP) models from a powerful integrated development environment (IDE) built on the Optimization Programming Language (OPL), through programmatic APIs, or alternatively through third-party modeling environments.

OPL provides a natural representation of optimization models, requiring far less effort than general-purpose programming languages. Debugging and tuning tools support the development process, and once ready, the model can be deployed into an external application. OPL models can be easily integrated into any application written in Java, .NET or C++.

Additionally, CPLEX Optimization Studio is tightly integrated with IBM ILOG ODM Enterprise (optional), providing push-button generation of ODM Enterprise applications based on OPL models. A simple wizard-guided step produces an initial application, mapping OPL data structures to data tables in ODM Enterprise, decision variables and solution metrics to solution views, and objective functions to ODM Enterprise's interactive business goals. ODM Enterprise provides rapid development of analytical decision support applications for immediate deployment. It helps users to adjust assumptions, operating constraints and goals, and see the engine's recommendations in familiar business terminology. It also provides extensive support for what-if analysis, scenario comparison, solution explanations, and the controlled relaxation of binding constraints.

About IBM ILOG CPLEX CP Optimizer

IBM ILOG CPLEX CP Optimizer is a second-generation constraint programming (CP) optimizer for solving detailed scheduling problems, as well as certain combinatorial optimization problems that cannot be easily linearized and solved using traditional mathematical programming methods.

About the lab files

The zip file you downloaded to obtain this white paper contains the following items:

- This document in PDF format
- A directory, named **Scheduling_Labs**, that contains, under it, three directories named:
  - **Staff**
  - **Calendar**
  - **Wood**

These contain work and solution directories with the OPL files needed to complete the labs found in this white paper. These labs are compatible with all versions of CPLEX Optimization Studio, and will also work with IBM ILOG OPL Development Studio 6.0 or later. You need to have these versions installed on your computer in order to perform the exercises.

About Detailed Scheduling

CPLEX Optimization Studio provides direct access to CPLEX CP Optimizer features, which are specially adapted to solving detailed scheduling problems over fine-grained time. There are, for example, keywords particularly designed to represent such aspects as tasks and temporal constraints.

CPLEX Optimization Studio offers you a workbench of modeling features in the CPLEX CP Optimizer engine that intuitively and naturally tackle the issues inherent in detailed scheduling problems from manufacturing, construction, driver scheduling, and more.

In a detailed scheduling problem, the most basic activity is assigning start and end times to intervals. The CPLEX CP Optimizer implementation is especially useful for fine-grained scheduling. Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time and of alternative modes to perform a task.

What is a detailed scheduling problem?

Detailed scheduling can be seen as the process of assigning start and end times to intervals, and deciding which alternative will be used if an activity can be performed in different modes. Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time.

A typical scheduling problem is defined by:

- A set of time intervals—definitions of activities, operations, or tasks to be completed, that might be optional or mandatory
- A set of temporal constraints—definitions of possible relationships between the start and end times of the intervals
- A set of specialized constraints—definitions of the complex relationships on a set of intervals due to the state and finite capacity of resources
- A cost function—for instance, the time required to perform a set of tasks, cost for some optional tasks of non execution, or the penalty costs of delivering some tasks past a due date

Scheduling models in OPL

A scheduling model has the same format as other models in OPL:

- Data structure declarations
- Decision variable declarations
- Objective function
- Constraint declarations

OPL provides specialized variables, constraints and keywords designed for modeling scheduling problems.

A simple scheduling problem

The example that follows, a simple house building problem, declares a series of tasks (dvar interval) of fixed time duration (**size**) that need to be scheduled (assigned start and end times).

These tasks have precedence constraints. This means that one task must be completed before another can start. For example, in the example code, the carpentry task must be complete before the roofing task can start.

```
using CP;
dvar interval masonry size 35;
dvar interval carpentry size 15;
dvar interval plumbing size 40;
dvar interval ceiling size 15;
dvar interval roofing size 5;
dvar interval painting size 10;
dvar interval windows size 5;
dvar interval facade size 10;
dvar interval garden size 5;
dvar interval moving size 5;
```

```
subject to {
endBeforeStart(masonry, carpentry);
endBeforeStart(masonry, plumbing);
endBeforeStart(masonry, ceiling);
endBeforeStart(carpentry, roofing);
endBeforeStart(ceiling, painting);
endBeforeStart(roofing, windows);
endBeforeStart(roofing, facade);
endBeforeStart(plumbing, facade);
endBeforeStart(roofing, garden);
endBeforeStart(plumbing, garden);
endBeforeStart(windows, moving);
endBeforeStart(facade, moving);
endBeforeStart(garden, moving);
endBeforeStart(painting, moving);
}
```

Note: In OPL, the unit of time represented by an interval decision variable is not defined. As a result, the size of the masonry task in this problem could be 35 hours or 35 weeks or 35 months.

Intervals—the tasks to schedule

In OPL, tasks, such as the activities involved in house building problem, are modeled as intervals, represented by the decision variable type interval. An interval has the following attributes:

- Start
- End
- Size
- Optionality
- Intensity (calendar function)

The time elapsed between the start and the end is the length of an interval.

The size of an interval is the time required to perform the task without interruptions. An interval decision variable allows these attributes to vary in the model, subject to constraints.

Syntax:
`dvar interval` <`taskName`> <`switches`>

where <`switches`> represents one or more different modifying conditions to be applied to the interval.

Some of the switches available include:

- Setting a time window for the interval, for example:

```
dvar interval masonry in 0..20;
```

- Providing different length to the interval from its size. A task may have a fixed size, but processing may be suspended during a break, so that the length is greater than the size. For example, the windows in the house example may take five days (size) to install, but if work stops over a two-day weekend, the length of the windows interval decision variable would be 7. The declaration would then be:

```
dvar interval windows size 5 in 0..7;
```

- Optionality: interval decision variables can be declared as optional. An optional interval may or may not be present in the solution. If landscaping were an unnecessary part of building the house, the interval decision variable garden would be declared as optional:

```
dvar interval garden optional;
```

The declaration

```
dvar interval garden optional in 20..32
size 5;
```

declares that the task **`garden`**, if present, requires 5 time units to execute, and must start after time unit 20 and end before time unit 32.

Stated in everyday language, this declaration says that construction of a garden is not mandatory for building the house, but if it is to be done, (and assuming that the time units are days), the task requires five days to perform, and the five days of garden construction must happen between day 20 and day 32 in the house construction timeline.

You will find the complete syntax for **`interval`** declarations in the *Language Quick Reference* manual of the documentation.

Functions that operate on intervals

A number of functions are available to operate on intervals. These are normally used in decision expressions (using the keyword, **`dexpr`**) to access an aspect of the interval, or to define a cost for a schedule. Some of these include:

- **`endOf`** – integer expression used to access the end time of an interval
- **`startOf`** – integer expression used to access the start time of an interval
- **`lengthOf`** – integer expression used to access the length of an interval
- **`sizeOf`** – integer expression used to access the size of an interval
- **`presenceOf`** – integer expression returning 1 if an optional interval is present, and 0 otherwise

All the functions related to scheduling can be found in the **OPL Functions** section of the *Language Quick Reference* manual of the documentation.

Intensity (calendar functions)

A calendar (or **intensity** function) can be associated with an interval decision variable. Intensity is a function that applies a measure of usage or utility over an interval length. For example, it can be used to specify the availability of a person or physical resource (such as a machine) during the interval.

Syntax:

```
dvar interval <taskName> intensity F;
```

where **`F`** is a stepwise function with integer values:

- The intensity is 100 percent by default, and cannot exceed this value (granularity of 100).
- If a task cannot be processed at all during a certain time window, such as a break or holiday, then the intensity for that time period is set to 0.
- When a task is processed part-time (this can be due to worker time off, interaction with other tasks, etc.) the intensity is expressed as a positive percentage.

Consider a task—for example, **`decoration`**—that is performed during an interval one week in length. In this interval a worker works five full days, one half day, and has one day off; the intensity function would be 100 percent for five days, 50 percent for one day, and zero for the last day.

You declare the intensity values using a linear stepwise function, via the OPL keyword **`stepFunction`**.

Interval size, length, and intensity are always related by the following:

- Size multiplied by granularity is equal to the integral of the intensity over the length of the interval.
- Intensity cannot exceed 100 percent, so interval size can never exceed the interval length.
- Therefore, in the proceeding example, the interval length is seven days and size equals 5.5 workdays, and would be declared as follows:

```
stepFunction F = stepwise(0->1; 100->5;
50->6; 0->7);
dvar interval decoration size 5..5 in
1..7 intensity F;
```

Scheduling constraints

CP Optimizer in OPL provides a number of specialized constraints for scheduling that you can use to build your scheduling model.

Precedence constraints

Precedence constraints are common scheduling constraints used to restrict the relative position of interval variables in a solution. These constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval. A delay, fixed or variable, can be included.

For example a precedence constraint can model the fact that an activity **a** must end before activity **b** starts (optionally with some minimum delay **z**).

List of precedence constraints in OPL:

- `endBeforeStart`
- `startBeforeEnd`
- `endAtStart`
- `endAtEnd`
- `startAtStart`
- `startAtEnd`

Example Syntax:

```
startBeforeEnd (a,b[,z]);
```

Where the end of a given time interval **a** (modified by an optional time value **z**) is less than or equal to the start of a given time interval **b**:

```
s(a) + z ≤ s(b)
```

Thus, if the ceiling had to dry for two days before the painting could begin, you would write:

```
endBeforeStart(ceiling, painting, 2);
```

The meanings of these constraints are intuitive, and you can find complete syntax and explanations for all of them at **OPL, the modeling language** > **Constraints** > **Types of constraints** > **Constraints available in constraint programming** in the *Language Reference Manual*.

Cumulative constraints

In some cases, there may be a restriction on the number of intervals that can be processed at a given time, perhaps because there are limited resources available. Additionally, there may be some types of **reservoirs** in the problem description (cash flow or a tank that gets filled and emptied).

These types of constraints on resource usage over time can be modeled with constraints on **cumulative function** expressions. A cumulative function expression is a step function that can be incremented or decremented in relation to a fixed time or an interval. A cumulative function expression is represented by the OPL keyword `cumulFunction`.

Syntax:

`cumulFunction` <`functionName`> `=` <`elementary_function_expression`>;Where <`elementary_function_expression`> is a cumulative function expression that can legally modify a `cumulFunction`. These expressions include:

- `step`
- `pulse`
- `stepAtStart`
- `stepAtEnd`

A cumulative function expression can be constrained to model limited resource capacity by constraining that the function be less than the capacity:

```
workersUsage <= NbWorkers;
```

Note: The value of a cumulative function expression is constrained to be non-negative at all times.

Example of a pulse function

`pulse`—represents the contribution to the cumulative function of an individual interval variable or fixed interval of time. Pulse covers the usage of a cumulative or renewable resource when an activity increases the resource usage function at its start and decreases usage when it releases the resource at its end time.

```
cumulFunction f = pulse(u, v, h);
cumulFunction f = pulse(a, h);
cumulFunction f = pulse(a, hmin, hmax);
```

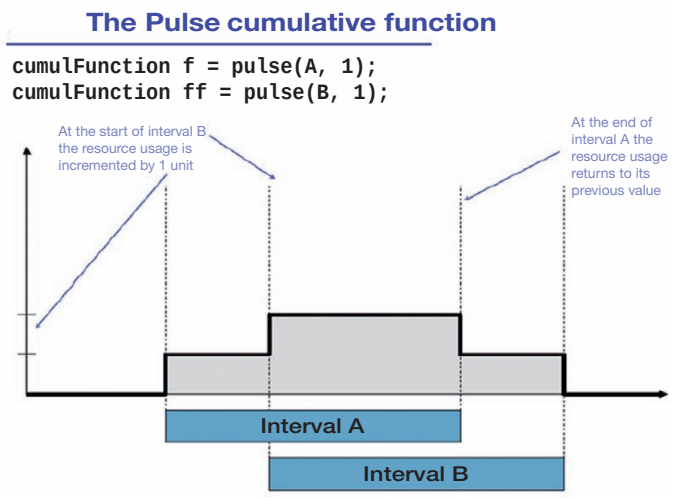
Where the pulse function interval is represented by **a** or by the start point **u** and end point **v**. The height of the function is represented by **h**, or bounded by **hmin** and **hmax**. To illustrate, consider a cumulative resource usage function that measures how much of a resource is being used:

- There are two intervals, A and B, bound in time
- Each interval increases the cumulative function expression by one unit over its duration

For each interval, this modification to the cumulative resource usage function can be made by incrementing the cumulative function with the elementary function, created with the interval and the given amount.

```
cumulFunction f = pulse(A, 1);
cumulFunction ff = pulse(B, 1);
```

Given this, the function would take the profile shown in the following graph:



The Pulse cumulative function

Example of step functions

step – represents the contribution to the cumulative function starting at a point in time:

```
cumulFunction f = step(u, h);
```

Where the time **u** is the start of production or consumption and **h** represents the height of the function.

As another example, consider a function measuring a consumable resource, similar to a budget resource:

- The level of the resource is zero, until time 2 when the value is increased to 4. This is modeled by modifying the cumulative function with the elementary cumulative function **step** at time 2:

```
cumulFunction f = step(2, 4);
```

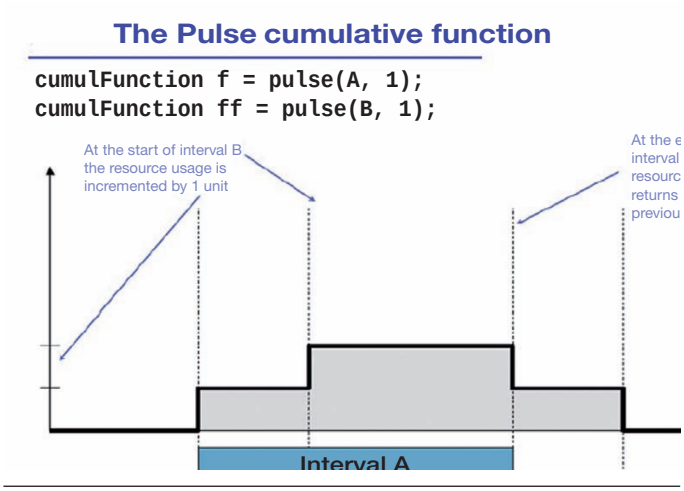
- There are two intervals, A and B, fixed in time. Interval A decreases the level of the resource by 3 at the start of the interval, modeled by applying **stepAtStart**, created with Interval A and the value 3, to the cumulative function:

```
cumulFunction ff = stepAtStart(A, -3);
```

- Interval B increases the level of the resource by 2 at the end of the interval, modeled by applying **stepAtEnd**, created with Interval B and the value 2, to the cumulative function for the interval:

```
cumulFunction fff = stepAtEnd(B, 2);
```

Given this, the function would take the profile shown in the following graph:



Step cumulative functions.

Other cumulative function expressions

- **`stepAtStart`** – represents the contribution to the cumulative function beginning at the start of an interval:

  ```
  cumulFunction f = stepAtStart(a, h);
  cumulFunction f = stepAtStart(a, hmin,
  hmax);
  ```

  Where the start of interval a is the start of production or consumption. The height of the function is represented by **`h`**, or bounded by **`hmin`** and **`hmax`**.

- **`stepAtEnd`** – represents the contribution to the cumulative function starting at the end of an interval:

  ```
  cumulFunction f = stepAtEnd(a, h);
  cumulFunction f = stepAtEnd(a, hmin,
  hmax);
  ```

  Where the end of interval **`a`** is the start of production or consumption. The height of the function is represented by **`h`**, or bounded by **`hmin`** and **`hmax`**.

Sequence decision variable and no overlap constraints

A scheduling model can contain tasks that must not overlap, for example, tasks that are to be performed by a given worker cannot occur simultaneously.

To model this, you use two constructs:

- The **`sequence`** decision variable
- The **`noOverlap`** scheduling constraint

Unlike precedence constraints, there is no restriction on relative position of the tasks. In addition, there may be transition times between tasks.

The sequence decision variable

Sequences are represented by the decision variable type **`sequence`**.

Syntax:

`dvar sequence` <**`sequenceName`**> **`in`** <**`intervalName`**> **`[types T];`**

Where **`T`** represents a non-negative integer.

A sequence variable represents a total order over a set of interval variables. If a sequence **`seq`** is defined over a set of interval variables **`{ a1, a2, a3, a4 }`**, a value for this sequence at a solution can be: **`(a1, a4, a2, a3)`**. A non-negative integer (the type) can be associated with each interval variable in the sequence. This integer is used by some constraints to group the set of intervals according to the type value.

Note: Absent interval variables are not considered in the ordering.

Example:

```
dvar sequence workers[w in WorkerNames] in
all(h in Houses, t in TaskNames:
Worker[t]==w) itvs[h][t] types
all(h in Houses, t in TaskNames:
Worker[t]==w) h;
```

The sequence can contain a subset of the interval variables or be empty. In a solution, the sequence will represent a total order over all the intervals in the set that are present in the solution.

The assigned order of interval variables in the sequence does not necessarily determine their relative positions in time in the schedule. To control the relative positions in time of a sequence, you use the constraints:

- **before**
- **first**
- **last**
- **prev**
- **noOverlap**

Complete syntax and explanations for these constraints are at **OPL, the modeling language** > **Constraints** > **Types of constraints** > **Constraints available in constraint programming** in the *Language Reference Manual.*

You are now going to look at the **noOverlap** constraint.

No overlap constraints

To constrain the intervals in a sequence such that they:

- Are ordered in time corresponding to the order in the sequence
- Do not overlap
- Respect transition times

OPL provides the constraint **noOverlap**.

Syntax:

```
noOverlap (<sequenceName> [,M]);
```

Where <**sequenceName**> is a previously declared sequence decision variable, and **M** is an optional transition matrix (in the form of a tuple set) that can be used to maintain a minimal distance between the end of one interval and the start of the next interval in the sequence.

In the following example:

- A set of **n** activities **A[i]** of integer type **T[i]** is to be sequenced on a machine.
- There is a sequence dependent setup time, **abs(ti-tj)** to switch from activity type **ti** to activity type **tj**.
- There should be no activity overlap.

```
tuple triplet { int id1; int id2; int
value; };
{triplet} M = { <i,j,ftoi(abs(i-j))> | i in
Types, j in Types };

dvar interval A[i in 1..n] size d[i];
dvar sequence p in A types T;

subject to {
noOverlap(p, M);
};
```

An additional interesting use of **noOverlap** is to shortcut the creation of the interval sequence variable for simple cases where the sequence is not useful:

```
noOverlap(A);
```

is equivalent to:

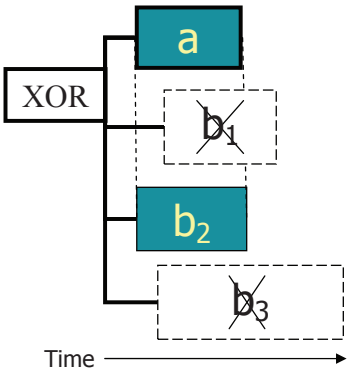
```
dvar sequence p in A;
noOverlap(p);
```

Where A is an interval decision variable (or a set of intervals) and p is a sequence decision variable.

Alternative and span constraints

The two keywords **`alternative`** and **span** provide important ways to control the execution and synchronization of different tasks.

An **`alternative`** constraint between an interval decision variable **a** and a set of interval decision variables **B** states that interval **a** is executed if and only if exactly one of the members of **B** is executed. In that case, the two tasks are synchronized. That is, interval **a** starts together with the first present interval from set **B** and ends together with it. No other members of set **B** are executed, and interval **a** is absent if and only if all intervals in the set **B** are absent:

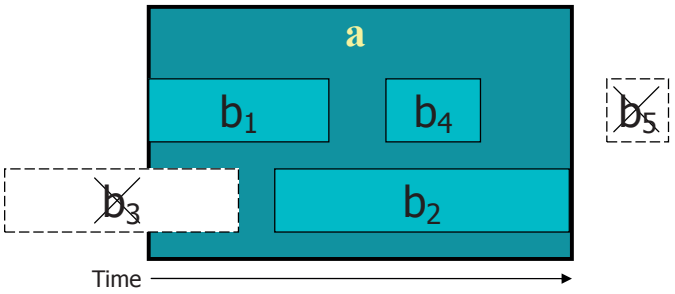


Example:
`alternative(a,[b1..bn]);`
where $\mathbf{b_2}$ is the first present interval in the set

a and $\mathbf{b_2}$ are executed simultaneously. No other intervals are

A **span** constraint between an interval decision variable **a** and a set of interval decision variables **B** states that interval **a** spans over all intervals present in the set. That is: interval **a** starts together with the first present interval from set **B** and ends together with the last present interval from set **B**. Interval **a** is absent if and only if all intervals in the set **B** are absent.

Example: `span(a,[b1..bn]);`



a starts together with $\mathbf{b_1}$ and ends together with $\mathbf{b_2}$. Intervals $\mathbf{b_3}$ and $\mathbf{b_5}$ are not executed.

Note: In both of these constraints, the array **B** must be a one-dimensional array; for greater complexity, use the keyword **`all`**.

Examples:

```
alternative(tasks[h] [t], all(s in Skills:
s.task==t) wtasks[h] [s]);
span(house[i], all(t in tasks : t.house ==
i) tasks[t]);
```

Synchronize constraint

A synchronization constraint (keyword **`synchronize`**) between an interval decision variable **a** and a set of interval decision variables **B** makes all present intervals in the set **B** start and end at the same times as interval **a**, if it is present.

Note: The array **B** must be a one-dimensional array; for greater complexity, use the keyword **`all`**.

Example:

```
synchronize(task[i], all(o in opers : o.task
== i) tiopers[o]);
```

LAB: Putting everything together—a staff scheduling problem

This lab outlines a typical scheduling problem, and takes you through the steps to use OPL Development Studio to build a model representing a staff scheduling problem.

To perform it, you need the following:

- CPLEX Optimization Studio must be installed and working on your computer.
- The lab files, included with this white paper when you downloaded it, must exist somewhere on your computer or an accessible network location.
- For this exercise, you will use this OPL project directory:

```
..\Scheduling_Labs\Staff\work\
sched_staffWork
```

- You need to know how to import this project, and projects from the labs that follow, into the OPL Projects Navigator. Here are the instructions:

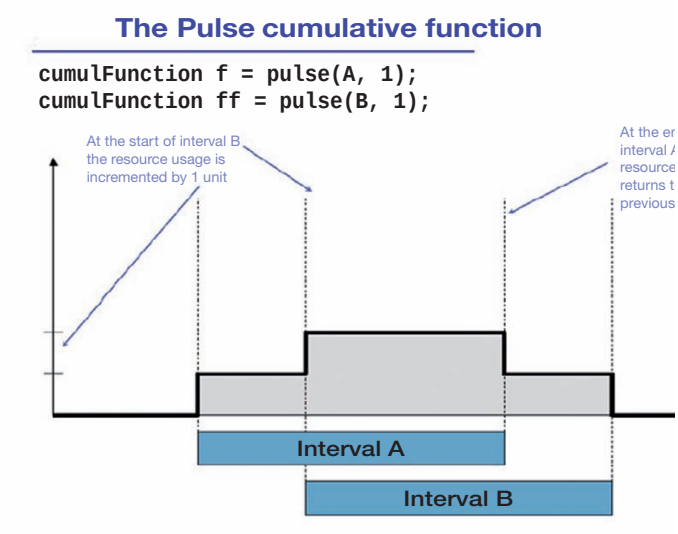
With the IDE running:

1. From the main menu, choose **File**>Import, or right-click in the OPL Projects Navigator and choose **Import**.

2. Select **Existing OPL 6.x projects.**
   The first screen of the Import Wizard is displayed:

This screen can be used to load one or more OPL projects into the OPL Projects Navigator. The general procedure for doing this is:

- Select a root directory (this is the directory on your file system where your existing OPL project is located). Alternatively you can select an archive file where the project has been stored.



First screen of the Import Wizard

- Select a project (or projects, if the root directory contains more than one project) to be imported.
- Indicate whether you want to copy the projects into the workspace.

The next steps walk you through each of the general procedures above, using the example of importing the **`sched_staffWork`** example provided with this white paper.

3. In the **Select root directory** field, enter the pathname of the directory that contains the project(s) you want to import. You can type in the pathname or use the **Browse** button to search for it. In this case, the pathname will begin at the top directory where you have installed the lab files, and continue with:
   `..\Scheduling_Labs\Staff\work\sched_staffWork`

Note: Alternatively, you can use the Select archive file field and enter the pathname of a JAR, ZIP, TAR or other compressed file that contains your project(s).

After you have selected a root directory (or archive file), the OPL projects under that directory that do not currently exist in your workspace are listed in the **Projects** view.

4. Check the box of each of the projects you want to import. (In this exercise, there is only one, **`sched_staffWork`**.)

5. Check the **Copy projects into workspace** box to create a working copy of the project in the CPLEX Optimization Studio IDE workspace—the original will remain unchanged—or you can leave the box unchecked if you want to work with the project "in place" in its current location.

6. Click **Finish** to import the project(s) into your OPL Projects Navigator.

The business problem

A telephone company must schedule customer requests for installation of different types of telephone lines:

- First (or principal) line
- Second (or additional) line
- ISDN (digital) line

Each request has a requested due date. A due date can be missed, but the objective is to minimize the number of days late.

These three request types each have a list of tasks that must be completed in order to complete the request. There are precedence constraints associated with some of the tasks. Each task has a fixed duration and also may require certain fixed quantities of specific types of resources.

The resource types are:

- Operator
- Technician
- CherryPicker (a type of crane)
- ISDNPacketMonitor
- ISDNTechnician.

The task types, along with their durations and resource requirements, are outlined in the following table:

Task type	Duration	Resources
MakeAppointment	1	Operator
FlipSwitch	1	Technician
InteriorSiteCall	3	Technician × 2
ISDNInteriorSiteCall	3	• Technician • ISDNTechnician • ISDNPacketMonitor
ExteriorSiteCall	2	• Technician × 2 • CherryPicker
TestLine	1	Technician

Each request type has a set of task types that must be executed. Some of the tasks must be executed before other tasks can start:

Request type	Task type	Preceding tasks
FirstLineInstall	FlipSwitch	
	TestLine	FlipSwitch
SecondLineInstall	MakeAppointment	
	InteriorSiteCall	MakeAppointment
	TestLine	InteriorSiteCall
ISDNInstall	MakeAppointment	
	• ISDNInteriorSiteCall • ExteriorSiteCall	MakeAppointment
	TestLine	• InteriorSiteCall • ExteriorSiteCall

The data

The following resources are available:

Resource	ID/Name
Operator	"Patrick"
Technician	"JohnTec"
Technician	"PierreT"
CherryPicker	"VIN43CP"
CherryPicker	"VIN44CP"
ISDNPacketMonitor	"EQ12ISD"
ISDNTechnician	"RogerTe"

The requests that need to be scheduled are:

Request Number	Request Type	Due date
0	FirstLineInstall	22
1	SecondLineInstall	22
2	FirstLineInstall	01
3	ISDNInstall	21
4	SecondLineInstall	21
5	ISDNInstall	22

Model the staff scheduling problem

You are now going to perform a series of steps using the lab files, based on this problem. First you'll transform the business problem into modeling language, then start to build the model. The major steps are:

1. Describe, in modeling terms, the objective and the unknowns

2. Declare task interval and precedence decision variables

3. Compute the end of each task and define the objective function

4. Define resource constraints

5. Add a surrogate constraint to accelerate search

What is the objective?

In business terms, we might say the objective is "to improve on-time performance." However, this is a qualitative statement that is difficult to model. In order to find a modeling representation (i.e. quantifiable) of the idea, we need to turn things around; instead of maximizing something abstract, we find a number that needs to be kept to a minimum. Thus, our objective becomes:

To minimize the total number of late days (days beyond the due date when requests are actually finished).

What are the unknowns?

The unknowns are:

- When each task will start
- Which resource will be assigned to each task

How to model this situation:

Here is an overview of what you are going to do. In the rest of the section, each task will be decomposed step by step to demonstrate the process.

1. The task type **`FlipSwitch`** requires a Technician, and there are two Technicians. For each task of this type, you create three **`interval`** decision variables. Two of these intervals are optional, meaning they may or may not appear in the solution.

2. To constrain such that when an optional interval is present, it is scheduled at exactly the same time as the task interval, use a **`synchronize`** constraint.

3. To ensure that the appropriate number of worker intervals are used, write a constraint that requires that the sum of present worker intervals is equal to the number of resources required.

Details of the intervals for `FlipSwitch`

- One optional represents **`JohnTec`** being assigned to the task.
- The other optional represents **`PierreT`** being assigned to the task.
- The third interval represents the task itself, and is used in other constraints.

Note: In general, if there are multiple resources with identical properties, it is best to model them as a **resource pool**. However, one can imagine that in this example, new constraints related to workers' days off could be added, so here each worker is treated as an individual resource.

Resource pools are explained in the section on **surrogate constraints.**

Modeling the precedence constraints

To model that some tasks in a request must occur before other tasks in the same request, you use the precedence constraint **`endBeforeStart`**.

While the data for this problem does not require that there be any delay between tasks, you can add a delay to the model to allow for the possibility of a delay.

Declare task interval and precedences

You will now begin the hands-on part of this exercise, and build the staff scheduling model step by step.

If you haven't already done so, import the **`sched_staffWork`** project into the OPL Projects navigator (leave the **Copy projects into workspace** box unchecked).

Examine data and model files

1. Open the **`step1.mod`** and **`data.dat`** files.
   The **`mod`** file represents a part of what the finished model will look like. Most of the model is already done.
   Note that there is no objective function. At this point, you have what is called a **satisfiability problem**. Running it will determine values that satisfy the constraints, without the solution necessarily being optimal.

2. Examine closely how the data declarations for the model are formulated.

3. Note especially, the declaration of the set **`demands`**. This creates a set whose members come from the tuple **`Demand`**, which is a tuple of tuples (**`RequestDat`** and **`TaskDat`**). This set is made sparse by filtering it such that only task/request pairs that are found in the tuple set **`recipes`** are included. Effectively, it creates a sparse set of required tasks to be performed in a request and operations (the same tasks associated with a given resource). Only valid combinations are in the set.
   Note: This is a good example of the power of tuple sets to create sparse sets of complex data.

4. The file **`data.dat`** instantiates the data as outlined in the problem definition. It instantiates:

- **`ResourceTypes`**
- **`RequestTypes`**
- **`TaskTypes`**
- **`resources`**
- **`requests`**
- **`tasks`**
- **`recipes`**
- **`dependencies`**
- **`requirements`**

Think about how model and data files are related.

Define the tasks and precedences

You are now ready to start declaring decision variables and constraints. At this point, we will define only the tasks, and the precedence rules that control them.

1. Declare an interval decision value to represent the time required to do each request/operation pair in the set **demands**. Name the decision variable **titasks**:

```
dvar interval titasks[d in demands]
size d.task.ptime;
```

2. An important aspect of the modeling is expressing the precedence constraints on the tasks (demands). These constraints can be expressed using the constraint **endBeforeStart**.

   The **step1.mod** file already contains the preparatory declarations:

```
forall(d1, d2 in demands, dep in depend-
encies :
  d1.request == d2.request &&
  dep.taskb == d1.task.type &&
  dep.taska == d2.task.type)
```

   Examine these declarations and try to understand clearly what they mean.

3. Write the **endBeforeStart** constraint.

4. Compare with the solution in **..\Scheduling_Labs\Staff\solution\sched_staffSolution\step1.mod**

Solve the model and examine the output

1. Solve the model by right clicking the **Step1** run configuration and selecting **Run this** from the context menu.

2. Look at the results in the **Solutions** output tab. Can you determine what the displayed values represent?

3. Look at the **Engine log** and **Statistics** output tabs, and note that this model is, for the moment, noted as a "Satisfiability problem."

4. Close **Step1.mod**.

Compute the end of each task and define the objective

To model the objective you need to determine the end time of each request. The request itself can be seen as an interval, with a variable length. The request interval must cover, or span, all the intervals associated with the tasks that comprise the request.

You create the objective by finding the difference between the end time and the due date, and minimizing it.

Compute the time needed for each request

1. Open **Step2.mod** for editing.

2. The requests are modeled as interval decision variables. Write the following declaration in the model:

```
dvar interval tirequests[requests];
```

3. Write a **span** constraint to link this decision variable to the appropriate **titasks** instances.

   Hint: use the **all** quantifier to associate the required tasks for each request with the appropriate duration.

4. Check your solution against **..\Scheduling_Labs\Staff\solution\sched_staffSolution\Step2.mod**

Transform the business objective into the objective function

The business objective requires the model to minimize the total number of late days (days beyond the due date when requests are actually finished). To do this in the model, you need to write an objective function that minimizes the time for each request that exceeds the due date.

1. Calculate the number of late days for each request:

- The data element **`requests`** is the set of data that instantiates the tuple **`RequestDat`**. The **`duedate`** is included in this information.
- The interval **`tirequests`** represents the time needed to perform each request.
- Subtract the **`duedate`** from the date on which **`tirequests`** ends.

  Hint: Use the **`endof`** function to determine the end time of **`tirequests`**.

2. Include a test that discards any negative results (requests that finish early) from the objective function.

   Hint: Use the **`maxl`** function to select the greater of:

- The difference between due date and finish date
- 0

3. Minimize the sum of all the non-negative subtractions, as calculated for each request.

4. Check the solution in

   `..\Scheduling_Labs\Staff\solution\sched_staffSolution\Step2.mod`

Solve the model and examine the output

1. Solve the model by right clicking the **`Step2`** run configuration and selecting **Run this** from the context menu.

2. Look at the **Engine log** and **Statistics** output tabs, and note that the model is now reported as a "Minimization problem," after the addition of the objective function. Scroll down a little further and notice the number of fails reported.

3. Look at the results in the Solutions output tab. You will notice that an objective is now reported, in addition to the values of **`titasks`** and **`tirequests`**.

4. Close **`Step2.mod`**.

Define resource constraints

So far, you have defined the following constraints as identified in the business problem:

- For each demand task, there are exactly the required number of task-resource **`operation`** intervals present.
- Each task precedence is enforced.
- Each **`request`** interval decision variable spans the associated **`demand`** interval decision variables.

It's time, now, to define the constraints on resources.

Review the needs

You now need to meet the following needs, not yet dealt with in the model:

- Each task-resource **`operation`** interval that is present is synchronized with the associated task's **`demand`** interval.
- There is no overlap in time amongst the present **`operation`** intervals associated with a given resource.
- At any given time, the number of overlapping task-resource **`operation`** intervals for a specific resource type do not exceed the number of available resources for that type.

Modeling the alternative resources

The task type **`FlipSwitch`** requires a technician. There are two technicians, i.e. there are two **alternative resources**, available to do the same task. You want to be able to optimize how each of these is used relative to the objective.

To create the alternative resources, you declare optional intervals for each possible task/resource pair.

To constrain a resource so that it cannot be used by more than one task at a given time, you need to ensure that there is no overlap in time amongst the intervals associated with a given resource.

How to model this situation:

1. Create a **`sequence`** decision variable from those intervals.

2. Place a **`noOverlap`** constraint on the **`sequence`** decision variable.

Assign workers to tasks

It is now time to deal with the question of who does what. We know from the data that there is more than one resource, in some cases, capable of doing a given task. How do we decide who is the best one to send on a particular job?

The key idea in representing a scheduling problem with alternative resources is:

- Model each possible task-resource combination with an optional interval decision variable.
- Link these with an interval decision variable that represents the entire task itself (using a **synchronize** constraint).

Procedure:

1. Open **`Step3.mod`** for editing.

2. You will see that a new data declaration has been added:

```
tuple Operation {
     Demand dmd;
      ResourceDat resource;
};
{Operation} opers = {<d, r >| d in
demands, m in requirements, r
in resources : d.task.type == m.task &&
r.type == m.resource};
```

   The members of the tuple set opers are the set of tasks assigned to a resource.

3. There is also a new decision variable associated with this tuple set that calculates the time required for each **`operation`**:

```
dvar interval tiopers[opers] optional;
```

   Note that this variable is optional. If one of the optional interval variables is present in a solution, this indicates that the resource associated with it is assigned to the associated task.

   Important: Remember that in this model a task is called a **`demand`**, and a task-resource pair is called an **`operation`**.

4. Declare a **`sequence`** decision variable named **`workers`**, associated with each resource.

   Hint: Use **`all`** to connect each **`resource`** used in an **`operation`** to its related **`tiopers`** duration:

```
dvar sequence workers[r in resources]
in all(o in opers : o.resource == r)
tiopers[o];
```

5. Constrain this decision variable using a **`noOverlap`** constraint to indicate the order in which a resource performs its **`operations`**.

"Just enough" constraint

Another constraint states that for each demand task, there are exactly the required number of task-resource operation intervals present ("just enough" to do the job—not more or less). The presence of an optional interval can be determined using the **`presenceOf`** constraint:

```
forall(d in demands, rc in requirements :
rc.task == d.task.type) {
  sum (o in opers : o.dmd == d &&
  o.resource.type == rc.resource)
  presenceOf(tiopers[o]) == rc.quantity;
```

- Write this into the model file.

Check your work and solve the model

1. Compare your results with the contents of **`..\Scheduling_Labs\Staff\solution\sched_staffSolution\Step3.mod`**

   Important: Do not copy the synchronization constraint into your work copy yet. First you are going to solve the model and observe the results.

2. Solve the model by right clicking the **Step3** run configuration and selecting **Run this** from the context menu.

3. Look at the **Engine log** and **Statistics** output tabs, and note that the number of variables and constraints treated in the model has increased slightly.

4. The results in the **Solutions** output tab show values for four decision values now, as well as the solution.

Synchronize simultaneous operations and observe the effects

1. Declare a constraint that synchronizes each task-resource **`operation`** interval that is present with the associated task's **`demand`** interval:

   ```
   forall (r in requests, d in demands :
   d.request == r)
     synchronize(titasks[d], all(o in opers :
     o.dmd == d)
     tiopers[o]);
   ```

2. Solve the model by right clicking the **Step3** run configuration and selecting **Run this** from the context menu.

3. Look at the **Engine log** and **Statistics** output tabs. The number of variables and constraints treated in the model has increased significantly, as has the number of fails.

4. Look at the results in the **Solutions** output tab, and note, especially how values for **`workers`** have changed from the previous solve.

5. Close **`Step3.mod`**.

Add a surrogate constraint to accelerate search

While these constraints completely describe the model, at times it is beneficial to introduce additional constraints that improve the search. Along with treating the resources individually, you can also treat the set of resources of a given type as a **resource pool**. A resource pool can be modeled using a cumulative function expression.

Each resource type has one **`cumulFunction`** associated with it. Between the start and end of a task, the **`cumulFunction`** for any required resource is increased by the number of instances of the resource that the task requires, using the **`pulse`** function.

A constraint that the **cumulFunction** never exceeds the number of resources of the given type is added to the model.

Note: These surrogate constraints on the **cumulFunction** expressions are crucial, as they enforce a stronger constraint when the whole set of resources of the tasks is not chosen.

Declare the cumulative function

1. Open **Step4.mod** for editing.

2. To model the surrogate constraint on resource usage, a cumulative function expression is created for each resource type. Each **cumulFunction** is modified by a **pulse** function for each demand. The amount of the **pulse** changes the level of the **cumulFunction** by the number of resources of the given type required by the demand:

```
cumulFunction cumuls[r in ResourceTypes] =
sum (rc in
  requirements, d in demands :
  rc.resource == r && d.task.type ==
  rc.task) pulse(titasks[d], rc.quantity);
```

Constrain the cumulative function

1. You will see that a new intermediate data declaration exists:

```
int levels[rt in ResourceTypes] =
sum (r in resources : r.type == rt) 1;
```

   This is used to test for the presence of a given resource in a resource type.

2. Write a constraint that requires, when a resource is present in a resource type, that the value of the **cumulFunction** must not exceed the value of **levels**.

3. Compare your results with the contents of **..\Scheduling_Labs\Staff\solution\sched_staffSolution\Step4.mod**

Solve the model and examine the results

- Solve the model by right clicking the **Step4** run configuration and selecting **Run this** from the context menu.
- If you look at the **Engine log** and **Statistics** output tabs, you will note a dramatic improvement in the number of fails, thanks to the surrogate constraint.

LAB: A house building calendar problem

You are now going to consider a problem for scheduling the tasks involved in building multiple houses in a manner that minimizes the overall completion date of the houses. It has many aspects in common with the previous lab, and adds the concept of calendars for resources.

To perform this lab, you need the following:

- CPLEX Optimization Studio must be installed and working on your computer
- The lab files, included with this white paper when you downloaded it, must exist somewhere on your computer or an accessible network location.
- For this exercise, you will use this OPL project directory:

..\ Scheduling_Labs\Calendar\work\sched_calendarWork

You'll need to know how to import this project into the OPL Projects Navigator—see the IDE documentation for instructions.

The business problem

There are five houses to be built. As usual, some tasks must take place before other tasks, and each task has a predefined size.

There are two workers, each of whom must perform a given subset of the necessary tasks.

A worker can be assigned to only one task at a time.

Each worker has a calendar detailing the days on which he does not work, such as weekends and holidays, with the following constraints:

- On a worker's day off, he does no work on his tasks.
- A worker's tasks may not be scheduled to start or end on a day off.
- Tasks that are in process by the worker are suspended during his days off.

Task	Size	Worker	Preceding tasks
Masonry	35	Joe	
Carpentry	15	Joe	Masonry
Plumbing	40	Jim	Masonry
Ceiling	15	Jim	Masonry
Roofing	05	Joe	Carpentry
Painting	10	Jim	Ceiling
Windows	05	Jim	Roofing
Façade	10	Joe	Roofing
			Plumbing
Garden	05	Joe	Roofing
			Plumbing
Moving	05	Jim	Windows
			Façade
			Garden
			painting

What is the objective?

In business terms, the objective is to minimize the total number of days required to build five houses.

What are the unknowns?

The unknowns are when each task will start. The actual length of a task depends on its position in time and on the calendar of the associated worker.

What are the constraints?

The constraints specify that:

- A particular task may not begin until one or more given tasks have been completed.
- A worker can be assigned to only one task at a time.
- Tasks that are in process are suspended during the associated worker's days off.
- A task cannot start or end during the associated worker's days off.

Model the house building calendar problem

You are now going to perform a series of steps using the lab files, based on this problem. The major steps are:

1. Define a calendar for each worker

2. Declare the decision variable

3. Define the objective function

4. Define constraints

Modeling the workers' calendars

A **`stepFunction`** is used to model the availability (**`intensity`**) of a worker with respect to his days off. This function has a range of [0..100], where the value 0 represents that the worker is not available and the value 100 represents that the worker is available for a full work period with regard to his calendar.

Note: While not part of this model, any value in 0…100 can be used as the intensity. For instance, the function could take the value 50 for a time window in which a resource works at half-capacity.

For each worker, a sorted tuple set is created. At each point in time where the worker's availability changes, a tuple is created. The tuple has two elements:

- The first element is an integer value that represents the worker's availability (0 for on a break, 100 for fully available to work, 50 for a half-day).
- The second element represents the date at which the availability changes to this value.

This tuple set, sorted by date, is then used to create a **`stepFunction`** to represent the worker's intensity over time. The value of the function after the final step is set to 100.

Define a calendar for each worker

1. Import the **`sched_calendarWork`** project into the OPL Projects Navigator (Leave the **Copy projects into workspace** box unchecked) and open the **`calendar.mod`** and **`calendar.dat`** files for editing.

2. Examine the first data declarations and their instantiations in the **`.dat`** file:

- The first two declarations, **`NbHouses`** and **`range Houses`** establish simple declarations of how many houses to build, and a range that is constrained between 1 and that total number.
- The next two declarations instantiate sets of strings that represent, respectively, the names of the workers and the names of the tasks to perform.
- The declaration **`int Duration [t in TaskNames] = ...;`** instantiates an array named **`Duration`** indexed over each **`TaskNames`** instance.
- The declaration **`string Worker [t in TaskNames] = ...;`** instantiates an array named **`Worker`** indexed over each **`TaskNames`** instance.
- The tuple set Precedences instantiates task pairings in the tuple **`Precedence`**, where each tuple instance indicates the temporal relationship between two tasks: the task in **`before`** must be completed before the task in **`after`** can begin.
- The tuple **`Break`** indicates the start date, **`s`**, and end date, **`e`** of a given break period. A list of breaks for each worker is instantiated as the array **`Breaks`**. Each instance of this array is included in a set named **`Break`**.

3. Declare a tuple named **`Step`** with two elements:

- An integer value, **`v`**, that represents the worker's availability at a given moment (0 for on a break, 100 for fully available to work, 50 for a half-day)
- An integer value, **`x`**, that represents the date at which the availability changes to this value. Make this element the key for the tuple.

```
tuple Step {
   int v;
   key int x;
};
```

4. Create a sorted tuple set such that at each point in time where the worker's availability changes, an instance of the tuple set is created. Sort the tuple set by date, and use a **`stepfuncion`** named **`calendar`** to create the intensity values to be assigned to each **`WorkerName`**

   Hint: use a **`stepwise`** function:

```
sorted {Step} Steps[w in WorkerNames] = {
<100, b.s >| b in
   Breaks[w] } union { <0, b.e >| b in
Breaks[w] };
stepFunction Calendar[w in WorkerNames] =
stepwise (s in
   Steps[w]) { s.v - >s.x; 00 };
```

Note: When two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

Modeling the unknowns

You need to know the dates when each task will start. If you have that information, the end dates are known, since the size of the task is known and the breaks that will determine the length of the task are also known.

Write an expression that calculates the start date of each task for each house, using this information.

Associate the step function **`Calendar`** with an interval variable using the keyword **`intensity`** to take the worker's availability dates into account.

This has been prepared for you already in the lab.

Declare the decision variable

1. Continue looking at the model file—the following **`interval`** decision variable is declared:

```
dvar interval itvs[h in Houses, t in
TaskNames]
  size Duration[t]
  intensity Calendar[Worker[t]];
```

2. Can you see how the **`Calendar`** function is associated with the decision variable in order to ensure that the worker's availability is taken into account?

Define the objective function

The objective of this problem is to minimize the total number of days required to build five houses. To model this, you minimize the overall completion date—i.e. the span of time from the start date of the first house to the completion date of the house that is completed last.

Transform the business objective into the objective function

To minimize the overall completion time span, you need to write an objective function that minimizes the maximum time needed to build each house and arrive at a minimum final completion date for the overall five-house project:

1. Determine the maximum completion date for each individual house project using the expression **`endOf`** on the last task in building each house (the moving task).

2. Minimize the maximum of these expressions.

Check the solution in
`..\Scheduling_Labs\Calendar\solution\sched_calendarSolution\calendar.mod`

Define constraints

You need three kinds of constraints in this model:

- Precedence constraints to determine the dependencies of one task on another
- No overlap constraints specify that a worker cannot be assigned to two tasks at once
- Forbidden start/end constraints specify days when a task cannot begin or end (due to worker unavailability)

Write the precedence constraint

The precedence constraints in this problem are simple **`endBeforeStart`** constraints with no delay.

1. Write a single constraint that can be applied via the tuple set **`Precedences`** to each instance of the interval decision variable **`itvs`**. Use filtering on **`(p in Precedences)`** to separate out start dates and end dates. Use arrays of the form **`[p.before]`** and **`[p.after]`**.

2. Check your work against the file **`..\Scheduling_Labs\Calendar\solution\sched_calendarSolution\calendar.mod.`**

Modeling the noOverlap constraint

To add the constraints that a worker can perform only one task at a time, the interval variables associated with that worker are constained to not overlap in the solution using the specialized constraint **noOverlap**:

```
forall(w in WorkerNames)
  noOverlap( all(h in Houses, t in
  TaskNames: Worker[t]==w)
  itvs[h][t]);
```

Write the noOverlap constraint

1. Write a constraint that says the interval variables associated with a worker are constrained to not overlap in the solution.

2. Check your work against the file
   ..\Scheduling_Labs\Calendar\solution\sched_calendarSolution\calendar.mod

You may be surprised by the form of the **noOverlap** constraint in the solution. This form is a shortcut that avoids the need to explicitly define the interval sequence variable when no additional constraints are required on the sequence variable.

Modeling forbidden start/end periods

When an intensity function is set on an interval variable, the tasks which overlap weekends and/or holidays will be automatically prolonged. An option could be available to start or end a task on a weekend day, but in this problem, a worker's tasks cannot start or end during the worker's days off.

A forbidden start or end is represented in IBM ILOG OPL by the constraints **forbidStart** and **forbidEnd**, which respectively constrain an interval variable to not end and not overlap where the associated step function has a zero value.

Write the forbidden start/end period constraint

1. Write a constraint, using **forbidStart** and **forbidEnd**, that forbids a task to start or end on the associated worker's days off (i.e. when **intensity** = 0).

2. Check your work against the file
   ..\Scheduling_Labs\Calendar\solution\sched_calendarSolution\calendar.mod.

Matters of State: Understanding State Functions

In some cases, there may be a restriction on what types of tasks can be processed simultaneously. For instance, in the house building problem, a "clean task" like painting cannot occur at the same time as a "dirty" task like sanding the floors.

Moreover, some transition may be necessary between intervals with different states, such as needing to wait for the paint to dry before floor sanding can take place.

This type of situation is called a state function. A **state function** represents the changes in state over time, and can be used to define constraints.

OPL provides the keyword **stateFunction** to model this.

Syntax:
stateFunction <**functionName**> **[with M];**

Where <**functionName**> is a label given to the function, and **M** is an optional transition matrix that needs to be defined as a set of integer triplets (just as for the **noOverlap** constraint). Thus this matrix is a tuple set.

For example, for an oven with three possible temperature levels identified by indexes 0, 1 and 2 we could have:

- **[start=0, end=100]: state=0**
- **[start=150, end=250]: state=1**
- **[start=250, end=300]: state=1**
- **[start=320, end=420]: state=2**
- **[start=460, end=560]: state=0**

In ordinary terms, this represents a set of non-overlapping intervals, each beginning at time **start** and ending at time **end**, over which the function maintains a particular non-negative integer value indicated by **state** (in this example, oven temperature).

In between those intervals, the state of the function is not defined (for example, between time 100 and time 150), typically because of an ongoing transition between two states (such as the time needed to change oven temperatures from **state 0** to **state 1**).

To model the oven example:

```
tuple triplet {
  int start;
  int end;
  int state;
};
{ triplet } Transition = ...;
//...
stateFunction ovenTemperature with
Transition;
```

State constraints

You can use constraints to restrict the evolution of a state function. These constraints can specify:

- That the state of the function must be defined and should remain equal to a given state everywhere over a given fixed or variable interval (**alwaysEqual**).
- That the state of the function must be defined and should remain constant (no matter its value) everywhere over a given fixed or variable interval (**alwaysConstant**).
- Those intervals requiring the state of the function to be defined cannot overlap a given fixed or variable interval (**alwaysNoState**).
- That everywhere over a given fixed or variable interval, the state of the function, if defined, must remain within a given range of states **[vmin, vmax] (alwaysIn)**.

Additionally, **alwaysEqual** and **alwaysConstant** can be combined with synchronization constraints to specify that the given fixed or variable interval should have its start and/or end point synchronized with the start and/or end point of the interval of the state function that maintains the required state (notions of start and end alignment).

Example:

```
int MaxItemsInOven = ...;
int NbItems = ...;
range Items = 1..NbItems;
int DurationMin[Items] = ...;
int DurationMax[Items] = ...;
int Temperature[Items] = ...;
tuple triplet { int start; int end; int
state; };
{ triplet } Transition = ...;
dvar interval treat[i in Items] size
DurationMin[i]..DurationMax[i];
stateFunction ovenTemperature with
Transition;
cumulFunction itemsInOven = sum(i in Items)
pulse(treat[i], 1);
constraints {
itemsInOven <= MaxItemsInOven;
forall(i in Items)
alwaysEqual(ovenTemperature, treat[i],
Temperature[i], 1, 1);
}
dvar interval maintenance ...;
constraints {
// ...
alwaysIn(ovenTemperature, maintenance, 0,
4);

}
```

This example models the oven problem described above (with more possible values for **`state`**), and adds the notion of synchronization. Certain items can be processed at the same time in the oven as they require the same temperature. There are limits, however, on the number of items that can be in the oven at one time, and on item size.

Finally, the last constraint models a required maintenance period where oven temperature cannot go beyond level 4.

Lab: A wood cutting problem

This lab features a problem that involves a process: cutting different kinds of logs into wood chips. It demonstrates a simple use of state constraints.

To perform this lab, you need the following:

- CPLEX Optimization Studio must be installed and working on your computer
- The lab files, included with this white paper when you downloaded it, must exist somewhere on your computer or an accessible network location.
- For this exercise, you will use this OPL project directory:

```
..\
Scheduling_Labs\Wood\work\sched_woodWork
```

You'll need to know how to import this project into the OPL Projects Navigator—see the IDE documentation for instructions.

The business problem

A wood factory machine cuts stands (processed portions of log) into chips. Each stand has these characteristics:

- Length
- Diameter
- Species of wood

The following restrictions apply:

- The machine can cut a limited number of stands at a time with some restriction on the sum of the diameters that it can accept.
- The truck fleet can handle a limited number of stands at a given time.
- Stands processed simultaneously must all be of the same species.
- Each stand has a fixed delivery date and a processing status of one of:
  - Standard
  - Rush

  Any delay on a rush stand will cost a penalty.

The wood cutting company needs to minimize costs per unit time, and reduce penalty costs resulting from late deliveries of rush stands to a minimum.

What are the unknowns?

The unknowns are the completion date of the cutting of the stands. An interval variable is associated with each of the stands. The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species.

What are the constraints?

The constraints are:

- At a given time, the machine can cut:
- A limited number of stands
- A limited sum of stand diameters
- Only one species.
- The trucks can carry a limited number of stands.

What is the objective?

In business terms, the objective is to minimize the combined total of cutting costs (expressed as a function of time spent on the machine) and the cost for any penalties due to late delivery of rushed orders.

Model the wood cutting problem

You are now going to perform a series of steps using the lab files, based on this problem. Almost all of this model is already done. You are going to examine it, and then write a state function and a state constraint to complete it. The major steps are:

1. Examine the completed parts of the model

2. Define the one species constraint

3. Examine the objective function

If you haven't already done so, import the **sched_woodWork** project into the OPL Projects Navigator. Leave the **Copy projects into workspace** box unchecked.

Examine the completed parts of the model

- Open the **sched_wood.mod** file for editing and examine it.

Modeling the processing of the stands

An interval variable is associated with each of the stands. The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species:

```
dvar interval a[s in stands] size (s.len *
cutTime[s.species]);
```

Modeling the quantity constraint

The number of stands being processed at a time can be modeled by a cumulative expression function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by 1 using the **pulse** function. A constraint that the cumul function never exceeds the stand capacity of the machine is added to the model:

```
cumulFunction standsBeingProcessed = sum
(s in stands) pulse(a[s], 1);
  standsBeingProcessed <= maxStandsTogether;
```

Modeling the diameter constraint

The total diameter of the stands being processed at a time can be modeled by a cumulative function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by the diameter using the **pulse** function. A constraint that the cumul function never exceeds the diameter capacity of the machine is added to the model:

```
cumulFunction diameterBeingProcessed = sum
(s in stands) pulse(a[s],
  s.diameter);
    diameterBeingProcessed <= maxDiameter;
```

Modeling the fleet constraint

The constraint on the number of trucks being used can be placed on the cumul function for the number of stands being processed:

```
cumulFunction trucksBeingUsed =
standsBeingProcessed;
  trucksBeingUsed <= nbTrucks;
```

Define the one species constraint

In this model, the wood cutting company can profit from processing multiple stands at the same time in the same batch, provided that certain constraints are met. One of these is that the cutting machine can only process one species of wood at a time.

Declare the state function

To express this constraint in the model, you are first going to declare a state function called **species**.

- Do this now in the model file.

Write an alwaysEqual constraint

1. Write a constraint that says that the value **species** in each member of the tuple set **stands** is equal when being processed by the cutting machine.

   Hint: Use the **ord** keyword to order the species together, and the scheduling constraint **alwaysEqual** to constrain the state function **species**.

2. Check your work against the file **..\Scheduling_Labs\Wood\solution\sched_woodSolution\sched_wood.mod**

Examine the objective function

In business terms, the objective is to minimize the combined total of cutting costs (expressed as a function of time spent on the machine) and the cost for any penalties due to late delivery of rushed orders.

In mathematical terms, this objective translates into minimizing the sum of:

- The product of the maximum cutting time per stand and the cost per time unit
- The product of the length of rushed stands that are late and the cost per unit of length for being late

Transform the business objective into the objective function

The objective requires the model to minimize the sum of two calculations. The first is the product of the maximum cutting time per stand and the cost per time unit.

- In the model, the maximum cutting time per stand is defined by a decision expression using the **dexpr** OPL keyword:

```
dexpr int makespan = max (s in stands)
endOf(a[s]);
```

- The first part of the objective function calculates the product of makespan and the cost per time unit:

```
minimize makespan * (costPerDay /
nbPeriodsPerDay)
```

The second quantity to be minimized is the product of the length of rushed stands that are late and the cost per unit of length for being late.

- To calculate the length (in feet, in this case) of stands identified as "rush" orders that are to be delivered late, we use another decision expression, named **lateFeet**:

```
dexpr float lateFeet = sum (s in stands :
s.rush == 1) s.len *
  (endOf(a[s])  >s.dueDate);
```

- We can now complete the objective function by adding the calculation of cost of late rushed footage. The entire objective function is:

```
minimize makespan * (costPerDay /
nbPeriodsPerDay) +
  costPerLateFoot * lateFeet;
```

Spend some time examining the solution in **..\Scheduling_Labs\Wood\solution\sched_woodSolution\sched_wood.mod**.

Pay special attention to how the state constraint interacts with the objective.

Summary

In this paper, you have looked at scheduling problems and how to model them in IBM ILOG CPLEX Optimization Studio.. Areas you covered include:

- Interval decision variables:
  - An interval variable has a start, an end, a size and a length, each of which may be fixed or variable.
  - An interval variable can be declared to be optional.
  - A calendar (step function) can be associated with an interval variable.

- Sequence decision variables:
  - A sequence variable takes an array of interval variables and fixes these in a sequence.
  - Such a sequence represents a total order over all the intervals in the set that are present in the solution.

- Specialized scheduling constraints:
  - No overlap constraints
  - Precedence constraints
  - Cumulative constraints
  - Calendar constraints
  - State constraints

You also learned how to work with the special tools provided by IBM ILOG CPLEX CP Optimizer in IBM ILOG CPLEX Optimization Studio for modeling detailed scheduling problems.

For more information

For more information on IBM ILOG CPLEX CP Optimizer and scheduling, please visit **ibm.com**/software/integration/optimization/cplex-optimization-studio/ or contact your IBM representative.

For more information on IBM ILOG CPLEX Optimization Studio, please visit **ibm.com**/software/integration/optimization/cplex-optimization-studio/ or contact your IBM representative.

Additionally, financing solutions from IBM Global Financing can enable effective cash management, protection from technology obsolescence, improved total cost of ownership and return on investment. Also, our Global Asset Recovery Services help address environmental concerns with new, more energy-efficient solutions. For more information on IBM Global Financing, visit: **ibm.com**/financing

IBM Global Services
Route 100
Somers, NY 10589
U.S.A.

Produced in the United States of America
November 2010

WSW14077-USEN-01