IBM ILOG CPLEX Optimization Studio Getting Started with Scheduling in CPLEX Studio

Version 12 Release 6

Contents

Chapter 1. Scheduling with IBM ILOG CPLEX Studio

Introduces the basic building blocks of a scheduling model.

In this section, you will learn:
- The basic building blocks of a scheduling model
- How to run a sample project to ensure your installation is working correctly.

CPLEX Studio is a rapid development system for optimization models with interfaces to embed models into standalone applications.

Scheduling models can be run in the CPLEX Studio IDE or using the command line executable, `oplrun`.
- For details on how to open an example in the IDE, please refer to Getting Started with the IDE
- For details on using `oplrun`, please refer to oplrun Command Line Interface

Prerequisites

To follow the examples in this section, you should have some knowledge about optimization (math programming or constraint programming) and about modeling optimization problems.

You should also be somewhat familiar with the CPLEX Studio IDE and how to work with it. If you are not, you should complete the tutorials in Getting Started with the IDE before continuing with this manual.

Tutorial project

The `scheduling_tutorial` project is located at `<Install_dir>/opl/examples/opl/tutorial/`.

To access this project in the CPLEX Studio IDE, use the following procedure:
1. In the IDE main menu, choose **File** > **New** > **Example** to launch the New Example wizard.
2. On the first screen of the wizard, select **IBM ILOG OPL Examples** and click **Next**.
3. On the next screen of the wizard, navigate to the **Scheduling tutorial** example, highlight it, and click **Finish**.

   Note that the **Scheduling tutorial** example is located under the **Basic** group on the default **Sorted by Complexity** tab of the wizard. You can also access it quickly by typing **tutorial** into the filter text field.
4. The example opens in the IDE, and appears in the OPL Projects Navigator window with the project title **scheduling_tutorial**.

Scheduling building blocks

Scheduling is the act of creating a schedule, which is a timetable for planned occurrences. Scheduling may also involve allocating resources to activities over time. A scheduling problem can be viewed as a constraint satisfaction problem or as a constrained optimization problem, but regardless of how it is viewed, a scheduling problem is defined by:

- A set of time intervals--definitions of activities, operations, or tasks to be completed
- 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.

In CPLEX Studio, a scheduling project contains one or more model files and, optionally, one or more data files and one or more settings files. The following sections provide deeper explanations of how the OPL language facilitates the representation of scheduling problems.

The model files

Model files (`.mod` extension) contain all of the OPL statements. The following components are usually present in a model file:

- the declarations of data
- the declarations of decision variables
- the declaration of engine
- an objective function
- the constraints
- script statements.

The data, the objective function, and scripting statements are not mandatory.

Declarations of data

Data declarations allow you to name your data so that you can reference it easily in your model. For example, if your data in a table defines the cost of shipping one unit of material from location i to location j, you might want to call your item of data **cost**ij where $i=1,..., n$, $j=1,..., n$, and n is the number of locations in your model. You tell OPL that your model uses this data by declaring:

```
int n = ... ;
float cost[1..n][1..n] = ... ;
```

The ... (ellipsis) means that the actual values for your table are located in a data file, which must be listed in the current project.

You could also list the data explicitly in the model file. However, it is recommended that you construct model files without specifying values for data so that you can later easily solve many instances of the same model by simply changing the data file.

Note:

The `int` type declared means that the numbers in the data file must be integers. If the numbers in the data file are floating-point numbers, use the float type instead.

Declarations of decision variables

Variable declarations name and define the type of each variable in the model. For example, if you want to create a variable that equals the amount of material shipped from location *i* to location *j*, you can create a variable named **ship***ij*:

```
dvar int+ ship[1..n][1..n];
```

The dvar keyword indicates that you are declaring a decision variable. Since `int+` indicates that the variables are nonnegative, this statement declares an array of nonnegative integer variables.

There is a restriction for constraint programming in OPL; float decision variables cannot be used. Instead, if you need to set constraints on float expressions, you should use float dexpr; see the section Integer and float expressions in the *Language Reference Manual*.

For scheduling there are specific additional decision variables, namely:
* interval
* sequence

In OPL, activities, operations and tasks are represented as interval decision variables.

An interval has a start, an end, a length, and a size. An interval variable allows for these values to be variable within the model. The start is the lower endpoint of the interval and the end is the upper endpoint of the interval. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

Also, an interval variable may be optional, and whether or not an interval is present in the solution is represented by a decision variable. If an interval is not present in the solution, this means that any constraints on this interval acts like the interval is "not there." The exact semantics will depend on the specific constraint.

The following example contains a two-dimensional array of interval decision variables where the sizes of the interval variables are fixed:

```
dvar interval itvs  [h in Houses][t in TaskNames] size Duration[t];
```

Declaration of engine

A scheduling model starts with the declaration of the engine as follows:

```
using CP;
```

This declaration tells OPL to use the CP engine to solve the problem. In addition, it allows the use of constraints that are specific to constraint programming or to scheduling.

Objective function

The objective function is an expression that you want to optimize. This function must consist of variables and data that you have declared earlier in the model file. The objective function is introduced by either the minimize or the maximize keyword. For example:

```
minimize endOf(tasks["moving"]);
```

This statement indicates that you want to minimize the end of the interval variable tasks["moving"].

Constraints

Constraints indicate the conditions necessary for a feasible solution to your model. You declare constraints within a subject to block. For example:

```
subject to {
  forall(t in Tasks)
    forall(s in successors[t])
      endBeforeStart(tasks[t], tasks[s]);
}
```

This statement declares one set of precedence constraints.

Several types of constraints can be placed on interval variables:
- precedence constraints, which ensure that relative positions of intervals in the solution (For example a precedence constraint can model a requirement that an interval a must end before interval b starts, optionally with some minimum delay z);
- no overlap constraints, which ensure that positions of intervals in the solution are disjointed in time;
- span constraints, which ensure that one interval to cover those intervals in a set of intervals;
- alternative constraints, which ensure that exactly one of a set of intervals be present in the solution;
- synchronize constraints, which ensure that a set of intervals start and end at the same time as a given interval variable if it is present in the solution;
- cumulative expression constraints, which restrict the bounds on the domains of cumulative expressions.

Script statements

Between the various blocks of a model (declaration of data, declaration of variables, constraints) or after the constraints, it is possible to add some script statements. This is useful for instance to preprocess input data, display it, or to display result data. These statements are written in IBM ILOG Script, an extension of the JavaScript language for OPL.

IBM ILOG Script for OPL enables you to:
- add preprocessing instructions to prepare data for the model;
- control the flow while the model is solved;
- set CPLEX parameters, CP Optimizer parameters, CP Optimizer search phases, and OPL options;
- add postprocessing instructions to aggregate, transform, and format data (including results data) for display or for sending to another application, for example, a spreadsheet;
- solve repeated instances of the same model;
- create algorithmic solutions where the output of one model instance is used as the input of a second model instance.

When you use IBM ILOG Script for OPL, you avoid having to compile and link; you just add script statements to your model file.

There are two possible top-level statements:

- the `main` statement for a flow control script, and
- the `execute` statement for preprocessing and postprocessing scripts.

```
minimize endOf(tasks["moving"]);
subject to {
  ...
}

execute DISPLAY {
  writeln("end=", tasks["moving"].end);
}
```

Data files

You can organize large problems better by separating the model of the problem from the instance data. In this case, you store the instance data in one or more data files (`.dat` extension). Data files store the actual values of the data used in the model. A data file will look something like this:

```
n = 3;

c = [[0.0 1.5 2.3]
     [1.5 0.0 3.7]
     [2.3 3.7 0.0]];
```

Each data file may specify one or more connections to data sources, such as a relational database or a spreadsheet, to read and write data. From the IDE, you can export external data and internal data to a `.dat` file, which you can later use as input. Only the data actually used in the model is exported to data files.

Execute and test

In future sections, you will work through tutorials by describing, modeling, and solving problems using IBM ILOG CPLEX Studio. In this section, you are provided with a completed example model so that you can test your installation of CPLEX Studio. In the next section, *Modeling and Solving a Simple Problem: House Building*, you will learn about the language features used in this model.

Description of the problem

The problem is a house building problem in which there are ten tasks of fixed size, each of which needs to be assigned a starting time. The statements for creating the interval variables that represent the tasks are:

```
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;
```

Adding the constraints

The constraints in this problem are precedence constraints; some tasks cannot start until other tasks have ended. For example, the ceilings must be completed before painting can begin. The set of precedence constraints for this problem can be added to the model with the block:

```
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);
}
```

Here there is a special constraint, endBeforeStart, which ensures that one interval variable ends before the other starts. This constraint has a special treatment in the engine. One reason is to correctly treat the presence of intervals so that if one of the interval variables is not present, the constraint is automatically satisfied, and another reason is for stronger inference in constraint propagation.

Displaying the solution

The interval variables and precedence constraints completely describe this simple problem. An execute block is used to display a solution to the model, after values have been assigned to the start and end of each of the interval variables in the model. The last part of the code for this example is:

```
execute {
  writeln("Masonry  : " + masonry.start   + ".." + masonry.end);
  writeln("Carpentry: " + carpentry.start + ".." + carpentry.end);
  writeln("Plumbing : " + plumbing.start  + ".." + plumbing.end);
  writeln("Ceiling  : " + ceiling.start   + ".." + ceiling.end);
  writeln("Roofing  : " + roofing.start   + ".." + roofing.end);
  writeln("Painting : " + painting.start  + ".." + painting.end);
  writeln("Windows  : " + windows.start   + ".." + windows.end);
  writeln("Facade   : " + facade.start    + ".." + facade.end);
  writeln("Garden   : " + garden.start    + ".." + garden.end);
  writeln("Moving   : " + moving.start    + ".." + moving.end);
}
```

Executing the example

Run the example model <Install_dir>/opl/examples/opl/sched_intro/ sched_intro.mod either by loading it in the IDE or by using the executable oplrun. When you run the model, you should get results similar to this output:

```
<<< setup


<<< generate

 ! ---------------------------------------------------------------------------
 ! Satisfiability problem - 10 variables, 14 constraints
 ! Initial process time : 0.00s (0.00s extraction + 0.00s propagation)
```

```
!  . Log search space  : 300.0 (before), 300.0 (after)
!  . Memory usage       : 283.0 Kb (before), 283.0 Kb (after)
! -------------------------------------------------------------------------
!   Branches  Non-fixed                   Branch decision
*        13      0.00s                          -
! -------------------------------------------------------------------------
! Solution status        : Terminated normally, solution found
! Number of branches     : 13
! Number of fails        : 0
! Total memory usage     : 432.3 Kb (315.0 Kb CP Optimizer + 117.3 Kb Concert)
! Time spent in solve    : 0.00s (0.00s engine + 0.00s extraction)
! Search speed (br. / s) : 1300.0
! -------------------------------------------------------------------------

<<< solve



OBJECTIVE: no objective
Masonry  : 0..35
Carpentry: 35..50
Plumbing : 35..75
Ceiling  : 35..50
Roofing  : 50..55
Painting : 50..60
Windows  : 55..60
Facade   : 75..85
Garden   : 75..80
Moving   : 85..90

<<< post process
```

To understand the solution found by OPL to this satisfiability scheduling problem, consider the line:

```
  Masonry  : 0..35
```

The interval variable representing the masonry task, which has size 35, has been assigned the interval [0,35). Masonry starts at time 0 and ends at the time point 35.

Note:

Displaying interval variables

After a time interval has been assigned a start value (say s) and an end value (say e), the interval is written as [s,e). The time interval does not include the endpoint e. If another interval variable is constrained to be placed after this interval, it can start at the time e.

In subsequent sections, the log output is not included. You can view the log information when you run the completed projects.

Chapter 2. Modeling and solving a simple problem: house building

Presents a simple problem of scheduling the tasks to build a house in such a manner that minimizes an objective.

In this section, you will learn how to:

- use the dvar interval;
- use the constraint endBeforeStart;
- use the expressions startOf and endOf.

You will learn how to model and solve a simple problem, a problem of scheduling the tasks involved in building a house in a way that minimizes an objective. Here the objective is the cost associated with performing specific tasks before a preferred earliest start date or after a preferred latest end date. Some tasks must necessarily take place before other tasks, and each task has a given duration. To find a solution to this problem using OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to tasks in such a way that the resulting schedule satisfies precedence constraints and minimizes a criterion. The criterion for this problem is to minimize the earliness costs associated with starting certain tasks earlier than a given date and tardiness costs associated with completing certain tasks later than a given date.

For each task in the house building project, the following table shows the duration (measured in days) of the task along with the tasks that must finish before the task can start.

Table 1. House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

The other information for the problem includes the earliness and tardiness costs associated with some tasks.

Table 2. House construction task earliness costs

Task	Preferred earliest start date	Cost per day for starting early
masonry	25	200.0
carpentry	75	300.0
ceiling	75	100.0

Table 3. House construction task tardiness costs

Task	Preferred latest end date	Cost per day for ending late
moving	100	400.0

Solving the problem consists of identifying starting dates for the tasks such that the cost, determined by the earliness and lateness costs, is minimized.

Note:

In OPL, the unit of time represented by an interval 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.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are ten house building tasks, each with a given duration. For each task, there is a list of tasks that must be completed before the task can start. Some tasks also have costs associated with an early start date or late end date.

What are the decision variables or unknowns in this problem?
- The unknowns are the date that each task will start. The cost is determined by the assigned start dates.

What are the constraints on these variables?
- In this case, each constraint specifies that a particular task may not begin until one or more given tasks have been completed.

What is the objective?
- The objective is to minimize the cost incurred through earliness and tardiness costs.

Model

After you have written a description of your problem, you can use OPL to model and solve it.

Step 2: Open the example file

- Still working with the **scheduling_tutorial** project, open the **sched_time.mod** file in the IDE editing area.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model. IBM ILOG OPL gives you the means to represent the unknowns in this problem, the interval in which each task will occur, as interval variables.

Note:

Interval variable

Tasks are represented by the decision variable type `interval` in OPL.

An interval has a start, an end, a size and a length. An interval variable allows these values to be variable in the model.

The length of a present interval variable is equal to the difference between its end time and its start time. The size is the actual amount of time the task takes to process. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

An interval variable may be optional. Whether an interval is present in the solution or not is represented by a decision variable. If an interval is not present in the solution, this means that any constraint on this interval acts like the interval is "not there." Exact semantics will depend on the specific constraint.

Logical relations can be expressed between the presence statuses of interval variables, allowing, for instance, to state that whenever the interval variable `a` is present then the interval variable `b` must also be present.

In your model, you first declare the interval variables, one for each task. Each variable represents the unknown information, the scheduled interval for each activity. After the model is executed, the values assigned to these interval variables will represent the solution to the problem.

For example, to create an interval with size 35 in OPL:
```
dvar interval masonry size 35;
```

Step 3: Declare the interval variables

Add the following code after the comment `//Declare the interval variables`:
```
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;
```

In this example, certain tasks can start only after other tasks have been completed. IBM ILOG OPL allows you to express constraints involving temporal relationships between pairs of interval variables using *precedence constraints*.

Note:

Precedence constraints

Precedence constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval. The following types of precedence constraints are available; if a and b denote interval variables, both interval variables are present, and delay is a number or integer expression (0 by default), then:

- endBeforeEnd(a, b, delay) constrains at least the given delay to elapse between the end of a and the end of b. It imposes the inequality endTime(a) + delay <= endTime(b).

- endBeforeStart(a, b, delay) constrains at least the given delay to elapse between the end of a and the start of b. It imposes the inequality endTime(a) + delay <= startTime(b).

- endAtEnd(a, b, delay) constrains the given delay to separate the end of a and the end of ab. It imposes the equality endTime(a) + delay == endTime(b).

- endAtStart(a, b, delay) constrains the given delay to separate the end of a and the start of b. It imposes the equality endTime(a) + delay == startTime(b).

- startBeforeEnd(a, b, delay) constrains at least the given delay to elapse between the start of a and the end of b. It imposes the inequality startTime(a) + delay <= endTime(b).

- startBeforeStart(a, b, delay) constrains at least the given delay to elapse between the start of act1 and the start of act2. It imposes the inequality startTime(a) + delay <= startTime(b).

- startAtEnd(a, b, delay) constrains the given delay to separate the start of a and the end of b. It imposes the equality startTime(a) + delay == endTime(b).

- startAtStart(a, b, delay) constrains the given delay to separate the start of a and the start of b. It imposes the equality startTime(a) + delay == startTime(b).

If either interval a or b is not present in the solution, the constraint is automatically satisfied, and it is as if the constraint was never imposed.

Step 4: Add the precedence constraints

Add the following code after the comment //Add the precedence constraints:

```
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);
```

To model the cost for starting a task earlier than the preferred starting date, you use the expression startOf that represents the start time of an interval variable as an integer expression.

For each task that has an earliest preferred start date, you determine how many days before the preferred date it is scheduled to start using the expression startOf; this expression can be negative if the task starts after the preferred date. By taking the maximum of this value and 0 using maxl, you determine how many days early the task is scheduled to start. Weighting this value with the cost per day of starting early, you determine the cost associated with the task.

The cost for ending a task later than the preferred date is modeled in a similar manner using the expression endOf. The earliness and lateness costs can be summed to determine the total cost.

Step 5: Add the objective

Add the following code after the comment //Add the objective:

```
minimize 400 * maxl(endOf(moving) - 100, 0)    +
         200 * maxl(25 - startOf(masonry), 0)   +
         300 * maxl(75 - startOf(carpentry), 0) +
         100 * maxl(75 - startOf(ceiling), 0);
```

Solve

Solving a problem consists of finding a value for each decision variable so that all constraints are satisfied. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

Step 6: Execute and display the solution

After a solution has been found, you can use the start and end properties of the interval variables to access the assigned intervals. The code for displaying the solution has been provided for you:

```
execute {
  writeln("Masonry  : " + masonry.start   + ".." + masonry.end);
  writeln("Carpentry: " + carpentry.start + ".." + carpentry.end);
  writeln("Plumbing : " + plumbing.start  + ".." + plumbing.end);
  writeln("Ceiling  : " + ceiling.start   + ".." + ceiling.end);
  writeln("Roofing  : " + roofing.start   + ".." + roofing.end);
  writeln("Painting : " + painting.start  + ".." + painting.end);
  writeln("Windows  : " + windows.start   + ".." + windows.end);
  writeln("Facade   : " + facade.start    + ".." + facade.end);
  writeln("Garden   : " + garden.start    + ".." + garden.end);
  writeln("Moving   : " + moving.start    + ".." + moving.end);
}
```

Step 7: Run the model

Run the model. You should get the following results:

```
OBJECTIVE: 5000
Masonry  : 20..55
Carpentry: 75..90
```

```
Plumbing : 55..95
Ceiling  : 75..90
Roofing  : 90..95
Painting : 90..100
Windows  : 95..100
Facade   : 95..105
Garden   : 95..100
Moving   : 105..110
```

As you can see, the overall cost is 5000 and moving will be completed by day 110.

You can also view the complete program online in the `<Install_dir>/opl/examples/opl/sched_time/sched_time.mod` file.

Chapter 3. Adding workers and transition times to the house building problem

Introduces workers and transition times to the house building problem described in the previous section.

In this section, you will learn how to:
- use the dvar interval variable sequence;
- use the constraints span and noOverlap;
- use the expression lengthOf.

You will learn how to model and solve a problem of scheduling the tasks involved in building multiple houses in a manner that minimizes the costs associated with completing each house after a given due date and with the length of time it takes to build each house. Some tasks must necessarily take place before other tasks, and each task has a predefined duration. Each house has an earliest starting date. Moreover, there are two workers, each of whom must perform a given subset of the necessary tasks, and there is a transition time associated with a worker transferring from one house to another house. A task, once started, cannot be interrupted. The objective is to minimize the cost, which is composed of tardiness costs for certain tasks as well as a cost associated with the length of time it takes to complete each house. To find a solution to this problem using IBM ILOG OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes a criterion. The criterion for this problem is to minimize the tardiness costs associated with completing each house later than its specified due date and the cost associated with the length of time it takes to complete each house.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task must be performed by a specific worker, Jim or Joe. A worker can only work on one task at a time; a task, once started, may not be interrupted. The time required to transfer from one house to another house is determined by a function based on the location of the two houses.

Table 4. House construction tasks

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing

Table 4. House construction tasks (continued)

Task	Duration	Worker	Preceding tasks
garden	5	Joe	roofing, plumbing
moving	5	Jim	windows, facade, garden, painting

For each of the five houses that must be built, there is an earliest starting date, a due date and a cost per day of completing the house later than the preferred due date.

Table 5. House construction tardiness costs

House	Earliest start date	Preferred latest end date	Cost per day for ending late
0	0	120	100.0
1	0	212	100.0
2	151	304	100.0
3	59	181	200.0
4	243	425	100.0

Solving the problem consists of determining starting dates for the tasks such that the cost, where the cost is determined by the lateness costs and length costs, is minimized.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given duration, or size. Each house also has a given earliest starting date. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and there is a transition time associated with a worker transferring from one house to another house. There are costs associated with completing each house after its preferred due date and with the length of time it takes to complete each house.

What are the decision variables or unknowns in this problem?

- The unknowns are the start and end dates of the interval variables associated with the tasks. Once fixed, these interval variables also determine the cost of the solution. For some of the interval variables, there is a fixed minimum start date.

What are the constraints on these variables?

- There are constraints that specify a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time and that it takes time for a worker to travel from one house to the other.

What is the objective?

- The objective is to minimize the cost incurred through tardiness and length costs.

Model

After you have written a description of your problem, you can use IBM ILOG OPL to model and solve it.

Step 2: Open the example file

- Still working with the **scheduling_tutorial** project, open the **sched_sequence.mod** file in the CPLEX Studio IDE.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model.

The code for reading the data into the data structures from the data file is provided.

In the related data file, the data provided includes the number of houses (NbHouses), the names of the workers (WorkerNames), the names of the tasks (TaskNames), the sizes of the tasks (Duration), the worker that can perform each task (Worker), the release date of each house (ReleaseDate), the preferred due date of each house (DueDate), the cost per day of ending each house late (Weight), and the precedence relations (Precedences).

One part of the objective is based on the length of time it takes to build a house. To model this, you use one interval variable for each house that you later will constrain to span the tasks associated with the given house. As each house has an earliest starting date, you declare each house interval variable to have a start date no earlier than that release date. The ending date of the task is not constrained, so the upper value of the range for the variable is maxint.

Step 3: Create the house interval variables

Add the following code after the comment //Create the house interval variables:

```
dvar interval houses[h in Houses] in ReleaseDate[h]..(maxint div 2)-1;
```

Each house has a list of tasks that must be scheduled. The duration, or size, of each task t is Duration[t]. Using this information, you build a matrix itvs of interval variables.

Step 4: Create the task interval variables

Add the following code after the comment //Create the task interval variables:

dvar interval itvs [h in Houses][t in TaskNames] size Duration[t];

The tasks of the house building project have precedence constraints that are added to the model.

Step 5: Add the precedence constraints

Add the following code after the comment //Add the precedence constraints:

```
forall(h in Houses)
  forall(p in Precedences)
    endBeforeStart(itvs[h][p.pre], itvs[h][p.post]);
```

To model the cost associated with the length of time it takes to build a single house, you constrain the interval variable associated with the house to start at the start of the first task of the house and end at the end of the last task. This interval variable must *span* the tasks.

Note:

Span constraint

With the constraint span, you can create a constraint that specifies that one interval variable must exactly cover a set of interval variables.

In other words, the spanning interval a is present in the solution if and only if at least one of the spanned interval variables is present and, in this case, the spanning interval variable starts at the start of the interval variable scheduled earliest in the set and ends at the end of the interval variable scheduled latest in the set

For house h, you constrain the interval variable houses[h] to cover the interval variables in itvs that are associated with the tasks of the given house.

Step 6: Add the span constraints

Add the following code after the comment //Add the house span constraints:

```
forall(h in Houses)
  span(houses[h], all(t in TaskNames) itvs[h][t]);
```

You model the times associated with the workers having to transfer between houses as a transition time tupleset.

Note:

Transition times

Transition times can be modeled using tuples with three elements. The first element is the interval variable type of one task, the second is the interval variable type of the other task and the third element of the tuple is the transition time from the first to the second. An integer interval variable type can be associated with each interval variable.

Given an interval variable a1 that precedes (not necessarily directly) an interval variable a2 in a sequence of non-overlapping interval variables, the transition time between a1 and a2 is an amount of time that must elapse between the end of a1 and the beginning of a2.

Step 7: Create the transition times

Add the following code after the comment //Create the transition times:

```
tuple triplet { int loc1; int loc2; int value; };
{triplet} transitionTimes = { <i,j, ftoi(abs(i-j))> | i in Houses, j in Houses };
```

Each of the tasks requires a particular worker. As a worker can perform only one task at a time, it is necessary to know all of the tasks that a worker must perform and then constrain that these intervals not overlap and respect the transition times.

A sequence variable represents the order in which the workers perform the tasks. Note that the sequence variable does not force the tasks to not overlap or the order of tasks; in a later step you will create a constraint that enforces these relations on the sequence of interval variables.

Note:

Interval sequence variable

Using the decision variable type sequence in IBM ILOG OPL, you can create a variable that represents a sequence of interval variables. The sequence can contain a subset of the 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.

The sequence variable takes an array of interval variables as well as the transition types for each of those variables. You create interval sequence variables for Jim and Joe, using the arrays of their tasks and the task locations.

Step 8: Create the sequence variables

Add the following code after the comment //Create the sequence variables:

```
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;
```

Now that you have created the sequence variables, you must constrain each sequence such that the interval variables do not overlap in the solution, that the transition times are respected, and that the sequence represents the relations of the interval variables in time. To do this, you use the constraint noOverlap.

Note:

No overlap constraint

Using the constraint noOverlap, you can constrain that the interval sequence variable passed defines a chain of non-overlapping intervals that are present in the solution. If a set of transition tuples is specified, it defines the minimal time that must elapse between two intervals in the chain.

Note that intervals which are not present in the solution are automatically removed from the sequence.

You add one no overlap constraint for the sequence interval variable for each worker.

Step 9: Add the no overlap constraint

Add the following code after the comment //Add the no overlap constraints:

```
forall(w in WorkerNames)
  noOverlap(workers[w], transitionTimes);
```

The cost for building a house is the sum of the tardiness cost and the number of days it takes from start to finish building the house. To model the cost associated with a task being completed later than its preferred latest end date, you use the expression endOf to determine the end date of the house interval variable. To model the cost of the length of time it takes to build the house, you use the expression lengthOf, which returns an expression representing the length of an interval variable. The objective of this problem is to minimize the cost as represented by the cost expression.

Step 10: Add the objective

Add the following code after the comment //Add the objective:

```
minimize sum(h in Houses)
  (Weight[h] * maxl(0, endOf(houses[h])-DueDate[h]) + lengthOf(houses[h]));
```

Solve

The search for an optimal solution in this problem could potentiality take a long time, so a fail limit has been placed on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process has been provided for you:

```
execute {
  cp.param.FailLimit = 20000;
}
```

Step 11: Execute the model

There are two ways of solving the model:

- Execute the model in the CPLEX Studio IDE. You can view the results in a Gantt chart; see "Viewing the results of sequencing problems in a Gantt chart" on page 21.
- Execute the model using oplrun. You will need to pass as arguments not only the model file but also the data file, <Install_dir>/opl/examples/opl/tutorial/ sched_sequence.dat.

You can view the complete model online in the file:

<Install_dir>/opl/examples/opl/sched_sequence/sched_sequence.mod

Viewing the results of sequencing problems in a Gantt chart

After solving a problem, the Problem Browser view shows both data and decision variable values. If the problem includes a scheduling variable, the value of the variable is displayed.
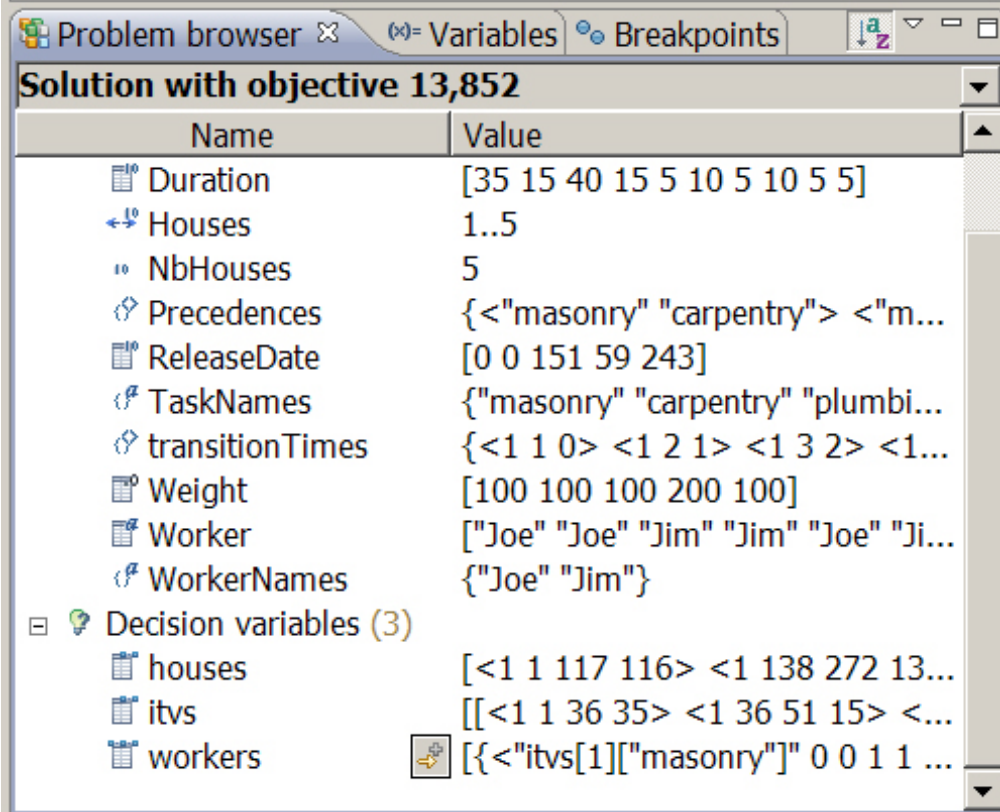
The sample model:

`<Install_dir>/opl/examples/opl/sched_sequence/sched_sequence.mod`

includes a sequence variable array. After the model is solved, the sequence value is displayed in the Problem Browser.

Detailed information can be viewed by clicking the button ![icon] **Show data view**. The button appears when the mouse hovers over a variable name.

The following screen shows the button next to the decision variable `workers`.

This button opens a new tab in the central window and shows the array in a tabular view. In the case of a sequence array, a **Gantt Chart** tab in the bottom left of the window allows you to open a Gantt Chart.

WorkerNames...	Count
["Joe"]	25
["Jim"]	25

itvs[1]["masonry"] itvs[1]["... i... itvs[4]["masonry"] itvs[... i... itvs[4]["... i... itvs[2]["masonry"]

itvs[1]["... itvs[1]["plumbing"] i... itvs[... i... itvs[4]["plumbing"] itv

Tabular Grid Gantt Chart
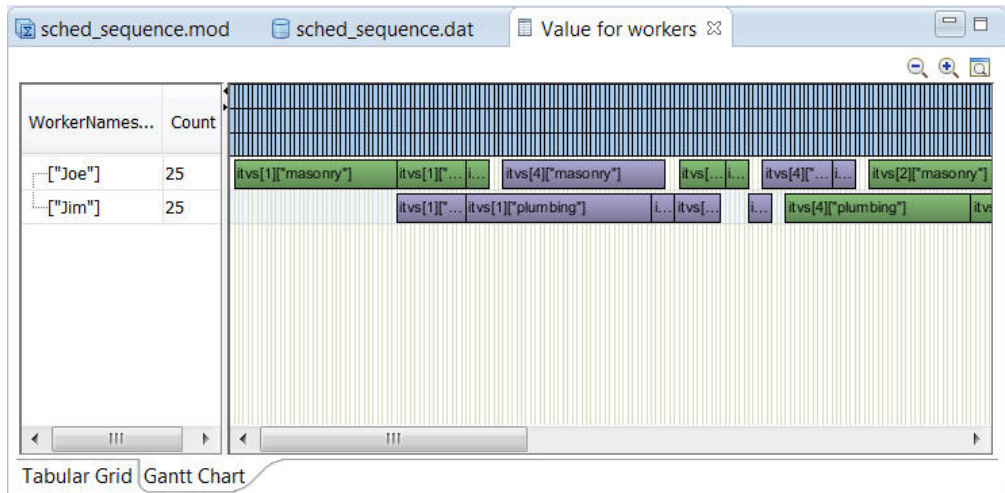
For each sequence of the array, the intervals are displayed on the same horizontal line. The noOverlap constraint, if posted in the problem model, can be checked visually on this line.

Chapter 4. Adding calendars to the house building problem

Introduces calendars to the house building problem.

In this section, you will learn how to:
- use the keyword stepFunction;
- use an alternative version of the constraint noOverlap;
- use intensity expression;
- use the constraints forbidStart and forbidEnd;
- use the length and size of an interval variable.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Moreover, there are two workers, each of whom must perform a given subset of the necessary tasks. Each worker has a calendar detailing on which days he does not work, such as weekends and holidays. On a worker's day off, he does no work on his tasks. His tasks may not be scheduled to start or end on these days. Tasks that are in process by the worker are suspended during his days off. To find a solution to this problem using IBM ILOG OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes a criterion. The criterion for this problem is to minimize the overall completion date.

For each task type in the house building project, the following table shows the size of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task can be performed by a given one of the two workers, Jim and Joe. A worker can only work on one task at a time; once started, a task may be suspended during a worker's days off, but may not be interrupted by another task.

Table 6. House construction tasks

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing
garden	5	Joe	roofing, plumbing

Table 6. House construction tasks (continued)

Task	Duration	Worker	Preceding tasks
moving	5	Jim	windows, facade, garden, painting

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and each worker has a calendar listing his days off.

What are the decision variables or unknowns in this problem?
- The unknowns are the start and end times of tasks which also determine the overall completion time. The actual length of a task depends on its position in time and on the calendar of the associated worker.

What are the constraints on these variables?
- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time. A task cannot start or end during the associated worker's days off.

What is the objective?
- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use IBM ILOG OPL to model and solve it.

Step 2: Open the example file
- Still working with the **scheduling_tutorial** project, open the **sched_calendar.mod** file in the IDE editing area.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model.

The code for reading the data into the data structures from the data file is provided.

In the related data file, the data provided includes the number of houses (NbHouses), the names of the workers (WorkerNames), the names of the tasks (TaskNames), the sizes of the tasks (Duration), the worker that can perform each task (Worker), the precedence relations (Precedences), and the breaks for each worker (breaks)..

To model the availability of a worker with respect to his days off, you first create a step function that represents his intensity over time. You specify that 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 with regard to his calendar.

IBM ILOG OPL provides the keyword stepFunction to represent a step function that is defined everywhere on a given interval and can be used to model the intensity of a worker.

Note:

Step function

Step functions are represented by the keyword stepFunction in IBM ILOG OPL.

Each interval [x1, x2) on which the function has the same value is called a step.

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.

For each worker, a sorted tupleset 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), and the other element represents the date at which the availability changes to this value. This tupleset, 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.

Step 3: Add the intensity step functions

Add the following code after the comment //Add the intensity step functions:

```
tuple Step {
  int v;
  key int x;
};
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; 100 };
```

This intensity function is used in creating the task variables for the workers. The intensity step function of the appropriate worker is passed to the creation of each interval variable. The size of the interval variable is the time spent at the house to process the task, not including the worker's day off. The length is the difference between the start and the end of the interval.

Step 4: Create the interval variables

Add the following code after the comment //Create the interval variables:

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

The tasks of the house building project have precedence constraints that are added to the model.

Step 5: Add the precedence constraints

Add the following code after the comment //Add the precedence constraints:

```
  forall(h in Houses) {
    forall(p in Precedences)
      endBeforeStart(itvs[h][p.pre], itvs[h][p.post]);
```

To add the constraints that a worker can perform only one task at a time, you constrain that the interval variables associated with that worker do not overlap in the solution. To do this, you use the specialized constraint noOverlap, but with a slightly different form than was used in the section Chapter 3, "Adding workers and transition times to the house building problem," on page 15.

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. You add to the model one no overlap constraint on the array of interval variables for each worker.

Step 6: Add the no overlap constraints

Add the following code after the comment //Add the no overlap constraints:

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

When an intensity function is set on an interval variable, the tasks which overlap weekends and/or holidays will be automatically prolonged. A task could still be scheduled to start or end in a weekend, but, in this problem, a worker's tasks cannot start or end during the worker's days off. OPL provides the constraints forbidStart and forbidEnd to model these types of constraints.

Note:

Forbid start constraint

With the constraint forbidStart, you can create a constraint that specifies that an interval variable must not be scheduled to start at certain times.

The constraint takes an interval variable and a step function. It the interval variable is present in the solution, then it is constrained to not start at a time when the value of the step function is zero.

OPL also provides forbidEnd and forbidExtent, which respectively constrain an interval variable to not end and not overlap where the associated step function is valued zero.

The first argument of the constraint forbidStart is the interval variable on which you want to place the constraint. The second argument is the step function that defines a set of forbidden values for the start of the interval variable: the interval variable cannot start at a point where the step function is 0.

Step 7: Create the forbidden start and end constraints

Add the following code after the comment //Create the forbidden start and end constraints (note that the final bracket is necessary to close the one opened in step 5):

```
    forall(t in TaskNames) {
      forbidStart(itvs[h][t], Calendar[Worker[t]]);
      forbidEnd  (itvs[h][t], Calendar[Worker[t]]);
    }
  }
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last). You determine the maximum completion date among the individual house projects using the expression endOf on the last task in building each house (here, it is the moving task) and minimize the maximum of these expressions.

Step 8: Create the objective

Add the following code after the comment //Add the objective:

```
minimize max(h in Houses) endOf(itvs[h]["moving"]);
```

Solve

The search for an optimal solution in this problem could potentiality take a long time, so a fail limit has been placed on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process has been provided for you:

```
execute {
  cp.param.FailLimit = 10000;
}
```

Step 9: Execute the model
- Execute the model in the CPLEX Studio IDE or using oplrun; you will need to pass as arguments not only the model file but also the data file, <Install_dir>/opl/examples/opl/tutorial/sched_calendar.dat.

You can also view the complete model online in the <Install_dir>/opl/examples/opl/sched_calendar/sched_calendar.mod file.

Chapter 5. Using cumulative functions in the house building problem

Introduces cumulative functions to the house building problem.

In this section, you will learn how to:
- use the keyword cumulFunction;
- use the keywords pulse, step, stepAtStart and stepAtEnd;

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses. Some tasks must necessarily take place before other tasks, and each task has a predefined duration. Moreover, there are three workers, and each task requires any one of the three workers. A worker can be assigned to at most one task at a time. In addition, there is a cash budget with a starting balance. Each task consumes a certain amount of the budget at the start of the task, and the cash balance is increased every 60 days. To find a solution to this problem using IBM ILOG OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes a criterion. The criterion for this problem is to minimize the overall completion date. Each task requires 200 dollars per day of the task, payable at the start of the task. Every 60 days, starting at day 0, the amount of 30,000 dollars is added to the cash balance.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. Each task requires any one of the three workers. A worker can only work on one task at a time; each task, once started, may not be interrupted.

Table 7. House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

There is an earliest starting date for each of the five houses that must be built.

House	Earliest starting date
0	31
1	0
2	90
3	120
4	90

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are five houses to be built by three workers. For each house, there are ten house building tasks, each with a given size and cost. For each task, there is a list of tasks that must be completed before the task can start. There is a starting cash balance of a given amount, and, each sixty days, the cash balance is increased by a given amount.

What are the decision variables or unknowns in this problem?
- The unknown is the point in time that each task will start. Once starting dates have been fixed, the overall completion date will also be fixed.

What are the constraints on these variables?
- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. Each task requires any one of the three workers. In addition, there are constraints that specify that a worker can be assigned to only one task at a time. Before a task can start, the cash balance must be large enough to pay the cost of the task.

What is the objective?
- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use the OPL language to model and solve it.

Step 2: Open the example file

- Still working with the **scheduling_tutorial** project, open the **sched_cumul.mod** file in the CPLEX Studio IDE.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model.

The code for reading the data into the data structures from the data file is provided.

In the related data file, the data provided includes the number of houses (`NbHouses`), the number of workers (`NbWorkers`), the names of the tasks (`TaskNames`), the sizes of the tasks (`Duration`), the precedence relations (`Precedences`), and the earliest start dates of the houses (`ReleaseDate`).

As each house has an earliest starting date, you declare the task interval variables to have a start date no earlier than that release date of the associated house. The ending dates of the tasks are not constrained, so the upper value of the range for the variables is maxint.

Step 3: Create the interval variables

Add the following code after the comment //Create the interval variables:

```
dvar interval itvs[h in Houses][t in TaskNames] in ReleaseDate[h]..(maxint div 2)-1 size Duration[t];
```

As the workers are equivalent in this problem, it is better to represent them as one pool of workers instead of as individual workers with no overlap constraints as was done in the earlier examples. The expression representing usage of this pool of workers can be modified by the interval variables that require a worker.

To model both the limited number of workers and the limited budget, you need to represent the sum of the individual contributions associated with the interval variables. In the case of the cash budget, some tasks consume some of the budget at the start. In the case of the workers, a task requires the worker only for the duration of the task. IBM ILOG OPL provides the keyword cumulFunction to represent the sum of individual contributions of interval variables.

Note:

Cumulative function expression

A cumulative function expression, represented in IBM ILOG OPL by cumulFunction, can be used to model a resource usage function over time. This function can be computed as a sum of interval variable demands on a resource over time.

An interval usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time (pulse function).

For resources that can be produced and consumed by activities (for instance the contents of an inventory or a tank), the resource level can also be described as a function of time. A production activity will increase the resource level at the start or end time of the activity whereas a consuming activity will decrease it. The

cumulated contribution of activities on the resource can be represented by a function of time, and constraints can be modeled on this function (for instance, a maximal or a safety level).

The value of the expression at any given moment in time is constrained to be nonnegative. A cumulative function expression can be modified with the atomic demand keywords:

- step, which increases or decreases the level of the function by a given amount at a given time;
- pulse, which increases or decreases the level of the function by a given amount for the length of a given interval variable or fixed interval;
- stepAtStart, which increases or decreases the level of the function by a given amount at the start of a given interval variable;
- stepAtEnd, which increases or decreases the level of the function by a given amount at the end of a given interval variable.

A cumulative function expression can be constrained to model limited resource capacity by constraining that the function be ≤ the capacity.

You create two cumulative functions, one to represent the usage of the workers and the other to represent the cash balance.

Each task requires one worker from the start to the end of the task interval. To represent the fact that a worker is required for the task, you create a cumulative function expression, workerUsage. This function will be constrained to not exceed the number of workers at any point in time. The keyword pulse adjusts the expression by a given amount on the interval. Summing these pulse atoms over all the interval variables results in an expression that represents worker usage over the entire time frame for building the houses.

Step 4: Declare the worker usage function

Add the following code after the comment //Declare the worker usage function:

```
cumulFunction workersUsage =
    sum(h in Houses, t in TaskNames) pulse(itvs[h][t],1);
```

To model the cash budget, you create a cumulative function expression, cash. To set the initial cash balance of 30,000 dollars and increase the balance by 30,000 every sixty days, you use the keyword step, which can be used to increment or decrement the cumulative function expression by a fixed amount on a given date.

Each task requires a cash payment equal to 200 dollars a day for the length of the task, payable at the start of the task. The keyword stepAtStart is used to adjust the cash balance cumulative function expression the appropriate amount for every task.

Step 5: Declare the cash budget function

Add the following code after the comment //Declare the cash budget function:

```
cumulFunction cash =
  sum (p in 0..5) step(60*p, 30000)
  - sum(h in Houses, t in TaskNames) stepAtStart(itvs[h][t], 200*Duration[t]);
```

The tasks have precedence constraints that are added to the model.

Step 6: Add the temporal constraints

Add the following code after the comment //Add the temporal constraints:

```
forall(h in Houses)
  forall(p in Precedences)
    endBeforeStart(itvs[h][p.pre], itvs[h][p.post]);
```

To add the constraint that there is a limited number of workers, you constrain the cumulative function expression representing worker usage to be no greater than the value NbWorkers.

Step 7: Add the worker usage constraint

Add the following code after the comment //Add the worker usage constraint:

```
workersUsage <= NbWorkers;
```

To add the constraints that the budget must always be nonnegative, you constrain the cumulative function expression representing the cash budget to be greater than 0.

Step 8: Add the cash budget constraint

Add the following code after the comment //Add the cash budget constraint:

```
cash >= 0;
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last). You determine the maximum completion date among the individual house projects using the expression endOf on the last task in building each house (here, it is the moving task) and minimize the maximum of these expressions.

Step 9: Add the objective

Add the following code after the comment //Add the objective:

```
minimize max(h in Houses) endOf(itvs[h]["moving"]);
```

Solve

The search for an optimal solution in this problem could potentiality take a long time, so a fail limit has been placed on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process has been provided for you:

```
execute {
  cp.param.FailLimit = 10000;
}
```

Step 9: Execute the model

There are two ways of solving the model:
- Execute the model in the CPLEX Studio IDE. You can view the results in tabular and chart form; see below *Viewing the results of cumulative functions in the IDE*.
- Execute the model using oplrun. You will need to pass as arguments not only the model file but also the data file, <Install_dir>/opl/examples/opl/tutorial/ sched_cumul.dat

You can view the complete model online in the file `<Install_dir>/opl/examples/opl/sched_cumul/sched_cumul.mod`.

## Viewing the results of cumulative functions in the IDE

After a problem is solved in the CPLEX Studio IDE, the Problem browser shows both data and decision variable values. If a scheduling model includes the `cumulFunction` keyword or a `cumulFunction` array, their values can be displayed in a dedicated view.

The sample model:

`<Install_dir>/opl/examples/opl/models/sched_atfm/sched_atfm.mod`

includes a `cumulFunction` array and, after the model is solved, the values of the cumulative functions are displayed in the Problem browser.

Detailed information can be viewed by clicking the button ⬒ **Show data view**, which appears when the mouse hovers over a variable name in the Problem browser. In the screen capture below, the button is next to the variable name r. The array is displayed in a tabular view in the central window. Each line of the table is a string representation (that can be copied and pasted) of the value of the cumulative function, as a stepwise function.
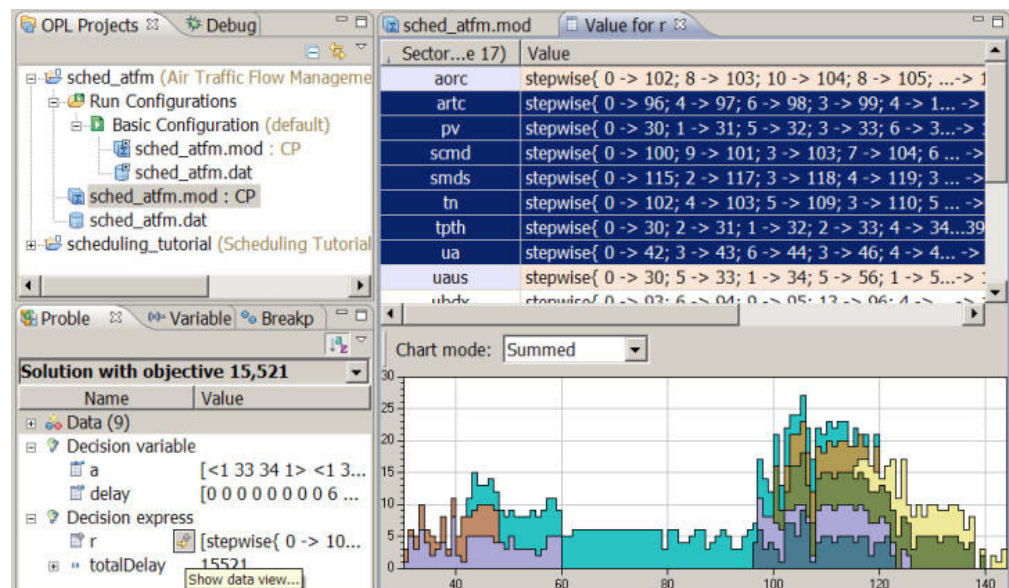
When lines are selected, a "staircase chart" below the tabular view shows the selected functions. Two types of chart mode are available: **Summed** or **Superimposed**.

To zoom into the chart, use Ctrl + left click + mouse move. To zoom out, use Shift + left click + mouse move.

Chapter 6. Using alternative resources in the house building problem

Describes how to use alternative resources in the house building problem.

In this section, you will learn how to:
- use the constraints alternative and presenceOf;
- use the keyword optional.

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses. Some tasks must necessarily take place before other tasks, and each task has a predefined duration. Each house has a maximal completion date. Moreover, there are three workers, and one of the three is required for each task. The three workers have varying levels of skills with regard to the various tasks; if a worker has no skill for a particular task, he may not be assigned to the task. For some pairs of tasks, if a particular worker performs one of the pair on a house, then the same worker must be assigned to the other of the pair for that house. The objective is to find a solution that maximizes the task associated skill levels of the workers assigned to the tasks. To find a solution to this problem using IBM ILOG OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and maximizes a criterion. The criterion for this problem is to maximize the task associated skill levels of the workers assigned to the tasks.

For each task type in the house building project, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. A worker can only work on one task at a time; each task, once started, may not be interrupted.

Table 8. House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

Every house must be completed within 300 days. There are three workers with varying skill levels in regard to the ten tasks. If a worker has a skill level of zero for a task, he may not be assigned to the task.

Table 9. Worker-task skill levels

Task	Joe	Jack	Jim
masonry	9	5	0
carpentry	7	0	5
plumbing	0	7	0
ceiling	5	8	0
roofing	6	7	0
painting	0	9	6
windows	8	0	5
façade	5	5	0
garden	5	5	9
moving	6	0	8

For Jack, if he performs the roofing task or facade task on a house, then he must perform the other task on that house. For Jim, if he performs the garden task or moving task on a house, then he must perform the other task on that house. For Joe, if he performs the masonry task or carpentry task on a house, then he must perform the other task on that house. Also, if Joe performs the carpentry task or roofing task on a house, then he must perform the other task on that house.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are five houses to be built by three workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. Each worker has a skill level associated with each task. There is an overall deadline for the work to be completed on the five houses.

What are the decision variables or unknowns in this problem?
- The unknown is the point in time that each task will start. Also, unknown is which worker will be assigned to each task.

What are the constraints on these variables?

- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that each task must have one worker assigned to it, that a worker can be assigned to only one task at a time and that a worker can be assigned only to tasks for which he has some level of skill. There are pairs of tasks that if one task for a house is done by a particular worker, then the other task for that house must be done by the same worker.

What is the objective?

- The objective is to maximize the skill levels used.

Model

After you have written a description of your problem, you can use IBM ILOG OPL to model and solve it.

Step 2: Open the example file

- Still working with the **scheduling_tutorial** project, open the **sched_optional.mod** file in the CPLEX Studio IDE.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model.

The code for reading the data into the data structures from the data file is provided.

In the related data file, the data provided includes the number of houses (NbHouses), the names of the workers (Workers), the names of the tasks (Tasks), the sizes of the tasks (Durations), the precedence relations (Precedences), and the overall deadline for the construction of the houses (Deadline).

The data also includes a tupleset, Skills. Each tuple in the set consists of a worker, a task, and the skill level that the worker has for the task. In addition, there is a tupleset, Continuities, which is a set of triples (a pair of tasks and a worker). If one of the two tasks in a pair is performed by the worker for a given house, then the other task in the pair must be performed by the same worker for that house.

You create two matrices of interval variables in this model. The first, tasks, is indexed on the houses and tasks and must be scheduled in the interval [0..Deadline]. The other matrix of interval variables is indexed on the houses and the Skills tupleset. These interval variables are optional and may or may not be present in the solution. The intervals that are performed will represent which worker performs which task.

Step 3: Create the interval variables

Add the following code after the comment //Create the interval variables:

```
dvar interval tasks [h in Houses][t in Tasks] in 0..Deadline size Durations[t];
dvar interval wtasks[h in Houses][s in Skills] optional;
```

The tasks in the model have precedence constraints that are added to the model.

Step 4: Add the temporal constraints

Add the following code after the comment //Add the temporal constraints:

```
forall(p in Precedences)
  endBeforeStart(tasks[h][p.pre], tasks[h][p.post]);
```

To constrain the solution so that exactly one of the interval variables wtasks associated with a given task of a given house is to be present in the solution, you use the specialized constraint alternative.

Note:

Alternative constraint

With the specialized constraint alternative, you can create a constraint between an interval and a set of intervals that specifies that if the given interval is present in the solution, then exactly one interval variable of the set is present in the solution.

In other words, consider an alternative constraint created with an interval variable a and an array of interval variables bs. If a is present in the solution, then exactly one of the interval variables in bs will be present, and a starts and ends together with this chosen interval.

Step 5: Add the alternative constraints

Add the following code after the comment //Add the alternative constraints:

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

You add the constraints that certain tasks must be performed by the same worker. To represent whether a task is performed by a worker, you use the expression presenceOf. The constraint presenceOf is true if the interval variable is present in and is false if the interval variable is absent from the solution.

For each house and each given pair of tasks and worker that must have continuity, you constrain that if the interval variable for one of the two tasks for the worker is present, the interval variable associated with that worker and the other task must also be present.

Step 6: Add the same worker constraints

Add the following code after the comment //Add the same worker constraints:

```
forall(c in Continuities,
      <c.worker, c.task1, l1> in Skills,
      <c.worker, c.task2, l2> in Skills)
  presenceOf(wtasks[h,<c.worker, c.task1, l1>]) ==
  presenceOf(wtasks[h,<c.worker, c.task2, l2>]);
```

To add the constraints that a given worker can be assigned only one task at a given moment in time, you use the constraint noOverlap.

Step 7: Add the no overlap constraints

Add the following code after the comment //Add the no overlap constraints:

```
forall(w in Workers)
  noOverlap(all(h in Houses, s in Skills: s.worker==w) wtasks[h][s]);
```

The presence of an interval variable in `wtasks` in the solution must be accounted for in the objective. Thus for each of these possible tasks, you increment the cost by the product of the skill level and the expression representing the presence of the interval variable in the solution.

The objective of this problem is to maximize the skill levels used for all the tasks, so you maximize the expression.

Step 8: Add the objective

Add the following code after the comment `//Add the objective`:

```
maximize sum(h in Houses, s in Skills) s.level * presenceOf(wtasks[h][s]);
```

Solve

The search for an optimal solution in this problem could potentiality take a long time, so a fail limit has been placed on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process has been provided for you:

```
execute {
  cp.param.FailLimit = 10000;
}
```

Step 9: Execute the model

- Execute the model in the CPLEX Studio IDE or using `oplrun`; you will need to pass as arguments not only the model file but also the data file, `<Install_dir>/opl/examples/opl/tutorial/sched_optional.dat`.

You can also view the complete model online in the `<Install_dir>/opl/examples/opl/sched_optional/sched_optional.mod` file.

Chapter 7. Using state functions: house building with state incompatibilities

Describes how to use state functions to take into account incompatible states as tasks finish.

In this section, you will learn how to:
- use the keyword stateFunction.
- use the constraint alwaysEqual;

You will learn how to model and solve a house building problem, a problem of scheduling the tasks involved in building multiple houses. Some tasks must necessarily take place before other tasks, and each task has a predefined size. Moreover, there are two workers, and each task requires either one of the two workers. A subset of the tasks require that the house be clean, whereas other tasks make the house dirty. A transition time is needed to change the state of the house from dirty to clean. To find a solution to this problem using IBM ILOG OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes an expression. The objective for this problem is to minimize the overall completion date.

For each task type in the house building project, the following table shows the duration of the task in days along with state of the house during the task. A worker can only work on one task at a time; each task, once started, may not be interrupted.

Table 10. House construction tasks

Task	Duration	State	Preceding tasks
masonry	35	dirty	
carpentry	15	dirty	masonry
plumbing	40	clean	masonry
ceiling	15	clean	masonry
roofing	5	dirty	carpentry
painting	10	clean	ceiling
windows	5	dirty	roofing
facade	10		roofing, plumbing
garden	5		roofing, plumbing
moving	5		windows, facade, garden, painting

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:
- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?
- There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. There are two workers. There is a transition time associated with changing the state of a house from dirty to clean.

What are the decision variables or unknowns in this problem?
- The unknowns are the date that each task will start. The cost is determined by the assigned start dates.

What are the constraints on these variables?
- There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. Each task requires either one of the two workers. Some tasks have a specified house cleanliness state.

What is the objective?
- The objective is to minimize the overall completion date.

Model

After you have written a description of your problem, you can use IBM ILOG OPL to model and solve it.

Step 2: Open the example file
- Still working with the **scheduling_tutorial** project, open the **sched_state.mod** file in the CPLEX Studio IDE.

This file is an OPL model that is only partially completed. You will add the missing code in each step of this lesson. At the end, you will have completed the OPL model.

The code for reading the data into the data structures from the data file is provided.

In the related data file, the data provided includes the number of houses (NbHouses), the number of workers (NbWorkers), the names of the tasks (TaskNames), the sizes of the tasks (Duration), the precedence relations (Precedences), and the cleanliness state of each task (AllStates).

Each house has a list of tasks that must be scheduled. The duration, or size, of each task t is Duration[t]. Using this information, you build a matrix task of interval variables.

Step 3: Create the interval variables

Add the following code after the comment //Create the interval variables:

```
dvar interval task[h in Houses][t in TaskNames] size Duration[t];
```

As in the example Chapter 5, "Using cumulative functions in the house building problem," on page 29, each task requires one worker from the start to the end of the task interval. To represent the fact that a worker is required for the task, you create a cumulative function expression, workers. This function is constrained to not exceed the number of workers at any point in time. The keyword pulse adjusts the expression by a given amount on the interval. Summing these pulse atoms over all the interval variables results in an expression that represents worker usage over the entire time frame for building the houses.

Step 4: Declare the worker usage functions

Add the following code after the comment //Declare the worker usage functions:

```
cumulFunction workers = sum (h in Houses, t in TaskNames)
  pulse(task[h][t], 1);
```

The transition time from a dirty state to a clean state is the same for all houses. As in the example Chapter 3, "Adding workers and transition times to the house building problem," on page 15, you create a tupleset to represent the transition time ttime between cleanliness states.

Step 5: Create the transition times

Add the following code after the comment //Create the transition times:

```
tuple triplet { int loc1; int loc2; int value; };
{triplet} ttime = {
  <Index["dirty"], Index["clean"], 1>,
  <Index["clean"], Index["dirty"], 0>
};
```

Certain tasks require the house to be clean, and other tasks cause the house to be dirty. To model the possible states of the house, you use the stateFunction keyword to represent the disjoint states through time.

Note:

stateFunction

A state function is a function describing the evolution of a given feature of the environment. The possible evolution of this feature is constrained by interval variables of the problem. For example, a scheduling problem may contain a resource whose state changes over time. The resource state can change because of scheduled activities or because of exogenous events; some activities in the schedule may need a particular resource state in order to execute.

Interval variables have an absolute effect on a state function, requiring the function value to be equal to a particular state or in a set of possible states.

Step 6: Declare the state function

Add the following code after the comment //Declare the state function:

```
stateFunction state[h in Houses] with ttime;
```

To model the state required or imposed by a task, you create a constraint that specifies the state of the house throughout the interval variable representing that task.

Note:

alwaysEqual

The constraint alwaysEqual, specifies the value of a state function over the interval variable.

The constraint takes a state function, an interval variable, and a state value. Whenever the interval variable is present, the state function is defined everywhere between the start and the end of the interval variable and remains equal to the specified state value over this interval.

You constrain the state function to take the appropriate values during the tasks that require the house to be in a specific state.

To add the constraint that there can be only two workers working at a given time, you constrain the cumulative function expression representing worker usage to be no greater than the value NbWorkers.

Step 7: Add the constraints

Add the following code after the comment //Add the constraints:

```
subject to {
  forall(h in Houses) {
    forall(p in Precedences) {
      endBeforeStart(task[h][p.pre], task[h][p.post]);
    }
    forall(s in States) {
      alwaysEqual(state[h], task[h][s.task], Index[s.state]);
    }
  }
  workers <= NbWorkers;
}
```

The objective of this problem is to minimize the overall completion date (the completion date of the house that is completed last).

Step 8: Add the objective

Add the following code after the comment //Add the objective:

```
minimize max(h in Houses) endOf(task[h]["moving"]);
```

Solve

The search for an optimal solution in this problem could potentiality take a long time, so a fail limit has been placed on the solve process. The search will stop when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process has been provided for you:

```
execute {
  cp.param.FailLimit = 10000;
}
```

Step 9: Execute the model

There are two ways of solving the model:

- Execute the model in the CPLEX Studio IDE. You can view the results in a table or a Gantt chart; see "Viewing the results of state functions in a Gantt chart."
- Execute the model using `oplrun`. You will need to pass as arguments not only the model file but also the data file, `<Install_dir>/opl/examples/opl/tutorial/sched_state.dat`.

You can view the complete model online in the file `<Install_dir>/opl/examples/opl/sched_state/sched_state.mod`.

Viewing the results of state functions in a Gantt chart

After a problem is solved in the CPLEX Studio IDE, the Problem browser shows both data and decision variable values. If a scheduling model includes the `stateFunction` keyword or a `stateFunction` array, their values can be displayed in a dedicated view.

The sample model:

`<Install_dir>/opl/examples/opl/sched_state/sched_state.mod`

includes a `stateFunction` array and, after the model is solved, the values of the state functions are displayed in the Problem browser.

Detailed information can be viewed by clicking the button **Show data view**, which appears when the mouse hovers over a variable name in the Problem browser. In the screen capture below, the button is next to the variable name `state`. The array is displayed in a tabular view in the central window. Each line of the table is a string representation (that can be copied and pasted) of the value of the state function, as a stepwise function.

When lines are selected, a Gantt chart below the tabular view shows the selected functions. Each line in the Gantt corresponds to one state function, as a series of intervals. Each interval is labeled with the state value for this interval. Undefined intervals are omitted. The zoom buttons are on the top right of the Gantt chart.

Index

A
alternative constraint   38
atomic demand   31

C
capacity   31
constraint   4
  alternative   4
  cumulative expression   4, 33
  no overlap   4, 19, 26, 38
  precedence   4, 11
  span   18
  specialized   2
  synchronize   4
  temporal   2
  types   4
constraint alternative   38
cumulative constraint   33
cumulative function   31
cumulative functions   29
cumulative functions, viewing in the
  IDE   34
cumulFunction keyword   31

D
data declaration   2
decision variable   3
demand
  atomic   31

E
end property   13
endBeforeStart constraint   11
endOf
  expression   12
endOf expression   27
engine declaration   3

F
fail limit   20, 27, 33, 39, 44
forbidEnd constraint   26
forbidExtend constraint   26
forbidStart constraint   26

I
interval   2
interval definition   3
interval keyword   11
interval variable   11
  end   12
  length   11, 20
  modifier   17
  optional   11, 37, 38

interval variable *(continued)*
  sequence   19

L
lengthOf expression   20
limit
  fail   20, 27, 33, 39, 44

M
model file   2

N
noOverlap constraint   19, 26, 38

O
objective function   3

P
presenceOf expression   38
pulse expression   31

S
scheduling building blocks   2
script statement   4
sequence interval variable   19
sequence keyword   19
span constraint   18
start property   13
state function, viewing results in the
  IDE   45
state functions   41
step expression   31
stepAtEnd expression   31
stepAtStart expression   31
stepFunction keyword   24

T
transition time   18

V
variable   3

IBM ®

Product Number: 1234-SS1

Printed in USA