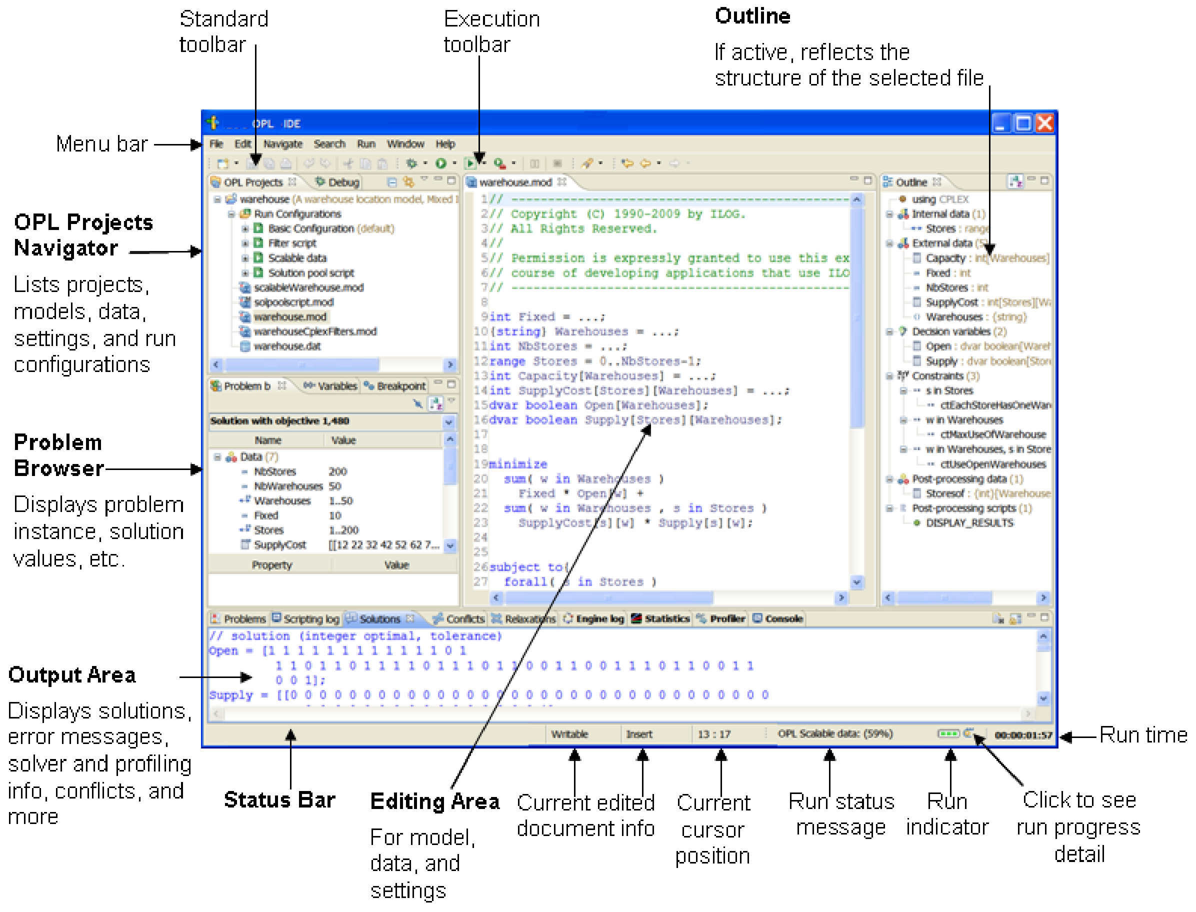
Introduction to IBM ILOG OPL Studio optimization environment

OPL (Optimization Programming Language) is language for describibing optimization problems. It resembles the usual way of writing mathematical equations. OPL Studio is an optimization environment, in which problems written in OPL can be solved using the CPLEX solver. This guide shortly introduces OPL Studio and writing OPL models.

User interface



```
// -----------------------------------------------------------
// Copyright (C) 1990-2009 by ILOG.
// All Rights Reserved.
//
// Permission is expressly granted to use this ex
// course of developing applications that use ILO
// -----------------------------------------------------------

int Fixed = ...;
{string} Warehouses = ...;
int NbStores = ...;
range Stores = 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores][Warehouses] = ...;
dvar boolean Open[Warehouses];
dvar boolean Supply[Stores][Warehouses];


minimize
  sum( w in Warehouses )
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];


subject to{
  forall( s in Stores )
```

Solution with objective 1,480

Name	Value
Data (7)	
NbStores	200
NbWarehouses	50
Warehouses	1..50
Fixed	10
Stores	1..200
SupplyCost	[[12 22 32 42 52 62 7...

Property	Value

```
// solution (integer optimal, tolerance)
Open = [1 1 1 1 1 1 1 1 1 1 1 1 0 1
         1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 1 0 0 1 1
         0 0 1];
Supply = [[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Projects / folders

IBM® ILOG® OPL uses the concept of a *project* to associate a model (`.mod`) file with, usually, one or more data (`.dat`) files and one or more settings (`.ops`) files.

A project containing only a single model file is valid; data and settings files are optional. However, one project can contain several sets of model, data and settings files, the relationships between them maintained using *run configurations*.

The *model file* declares data elements but does not necessarily initialize them. The *data files* contain the initialization of data elements declared in the model.

The `.project` file in the root folder for the OPL project organizes all the related model, data and settings files. Run configurations, which are maintained in an `.oplproject` file, also provide a convenient way to maintain the relationship between related files and runtime options for the environment (see also the Run configurations section).

When you are about to write a new model in IBM ILOG, the dialog box that appears allows you to name your project, give your project a description, and choose whether you want to create a data file or a settings file. The description of the project may be useful later to better differentiate projects with similar names. This is explained in detail in Getting Started with the OPL IDE and in the IDE Reference manual.

A minimal project has:

- one model file
- one default run configuration referencing that same model file

A typical project has:

- one or more model files
- any number of data files or no data file
- any number of settings files or no settings file
- one or more run configurations referencing various combinations of those model, data, and settings files. (A run configuration cannot have more than one model file.)

Model files

Model (`.mod`) files contain all your OPL statements. The data and the objective function are not mandatory and there may be more optional components, such as scripting statements. Note that you can also generate a model file in a compiled form (`.opl`) from the IDE for execution through the OPL interface libraries (see Generating a compiled model). The components of a model file are covered in the following sections.

Declarations of data

Data declarations allow you to name your data so that you can reference them easily in your model. For example, if your data in a table define 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* and *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 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. See also the Run configurations section.

Note that 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

In OPL context, as opposed to IBM ILOG Script and to the general programming context, variables are decision variables. Declarations of decision variables name and give 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 float+ ship[1..n][1..n];
```

That statement declares an array of non-negative floating-point variables. (That is what `float+` means). The `dvar` keyword indicates that you are declaring a decision variable.

An objective function

The objective function is a function 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 sum(i,j in 1..n) cost[i][j]*ship[i][j];
```

That statement indicates that you want to minimize the sum of the shipping costs for each origin-destination pair.

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(j in 1..n) sum(i in 1..n) ship[i][j] == demand[j];

}
```

That statement declares one set of constraints. There is a constraint for each destination. (That is what the `forall` keyword indicates.) The constraint for each destination states that the sum of material shipped to that destination must equal the demand at that destination. The symbol == indicates equals within a constraint block. The symbol <= indicates less than or equal to. The symbol >= indicates greater than or equal to.

Data files

You can organize large problems better by separating the model of the problem from the instance data, each set of data stored in a separate data file, with a `.dat` extension.

In this case, you store the instance data in one or more data files (`.dat`). Data files (`.dat`) store the values of the data used in the model. If you declare the data as suggested in this tutorial, your 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.

Settings files

Settings files (`.ops`) are where your user-defined values are stored when you decide to change the default values of OPL language options, constraint-programming (CP Optimizer) parameters, or mathematical-programming (CPLEX®) parameters.

OPL settings apply only to the model included in the run configuration, not to the submodels loaded and solved.

Run configurations

Run configurations are a way of handling model, data, and settings files within a project.

Basically, a run configuration is a variation of a given project for execution purposes. It combines at least a model file and, optionally, one or more data files and one or more settings files within the project, while addressing the same mathematical problem. You can define as many run configurations as you need within a given project. Typically, you use run configurations to test, improve, and fine-tune your OPL projects.

For example, you can:

- keep two sets of data: a simple one for quick prototyping and a larger one to work closer to your business case;
- keep one configuration for each set of MP options (CPLEX parameters) that makes sense for your problem.

Practically, run configurations appear as sublevels in the Projects tree.

Example: Total tardiness minimization on single machine

This example is directly from the lecture material:

$$\text{Min} \sum_{\forall i} f_i$$

$$t_i + P_i - D_i \le f_i, \qquad \forall i$$

$$t_i \ge 0,\ f_i \ge 0, \qquad \forall i$$

$$My_{ij} + (t_i - t_j) \ge P_j, \qquad \forall i \in \{1,..,I-1\},\ j \in \{i+1,..,I\}$$

$$M(1 - y_{ij}) + (t_j - t_i) \ge P_i, \qquad \forall i \in \{1,..,I-1\},\ j \in \{i+1,..,I\}$$

$$y_{ij} \in \{0,1\}, \qquad \forall i, j$$

Data

I = number of jobs

P_i = duration of job i

D_j = due date of job i

M = Large number

Decision variables

f_i = tardiness of job i
t_i = starting time of job i
y_{ij} = 1, if job i precedes job j, 0 otherwise

Objective

Minimize total tardiness

Constraints

1 Tardiness is difference between due date and finishing time of job *i*

2 Jobs’ starting times and tardiness must be positive

3 Jobs must not overlap

4 y_{ij} take only binary values

The construction of an OPL model file (`.mod`) follows the same format exactly. The problem can be formulated in IBM ILOG OPL like this:

```
//Data
int I = ...;
int M = ...;
float P[1..I] = ...;
float D[1..I] = ...;

//Variables
dvar float+ t[1..I];
dvar boolean y[1..I][1..I];
dvar float+ f[1..I];

//Objective
minimize sum (i in 1..I)f[i];

//Constraints
subject to{
  forall (i in 1..I) f[i] >= t[i] + P[i] - D[i];
  forall (i in 1..I-1,j in i+1..I) M * y[i][j] + (t[i] - t[j]) >= P[j];
  forall (i in 1..I-1,j in i+1..I) M * (1 - y[i][j]) + (t[j] - t[i]) >= P[i];
// FIFO simulation:
// forall (i in 1..I - 1)t[i+1] >= t[i];
}
```

Notice that:

- the `...` (ellipsis) syntax means that the data is initialized externally, that is, from a data file (`.dat`):

```
I = 5;
M = 1000;
P = [4,6,8,3,9];
D = [9,20,36,15,19];
/* Data reading and writing to Excel worksheet
SheetConnection sheet("TutorialDatafile.xls");
 P from SheetRead(sheet,"A1:A5");
 D from SheetRead(sheet,"B1:B5");
 t to SheetWrite(sheet,"C1:C5");
 f to SheetWrite(sheet,"D1:D5");
*/
```

Debugging and dealing with error messages

OPL checks for errors in the model and data files.

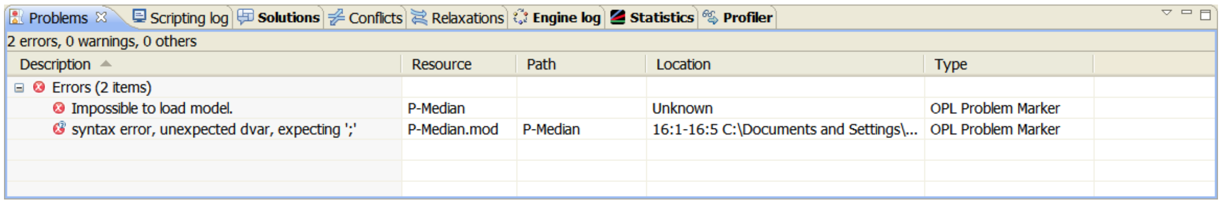
Syntax errors

Syntax and semantic errors are displayed dynamically in the Issues tab as you type.

For example, a common error is to forget to put a semicolon at the end of a statement. If you omit the semicolon at the end of the line

```
int P = ...;
```

the **Problems** tab displays the description, location, and source file of the error.



2 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (2 items)				
Impossible to load model.	P-Median		Unknown	OPL Problem Marker
syntax error, unexpected dvar, expecting ';'	P-Median.mod	P-Median	16:1-16:5 C:\Documents and Settings\...	OPL Problem Marker

Generally, error messages will look similar to this example.

Solving errors

Immediately after you run your project, OPL checks for errors that prevent the solver from running. If such errors are found, one or more error messages will be displayed in the **Problems** tab. (See *The Main window* section.)

Displaying solutions

It is possible for you to view solutions while the solver is running as well as after it has finished. In addition to the **Solutions** tab of the Output Area, you can view a solution in tabular form through the **Problem Browser**. (See The Main window.) If your model expresses a MIP problem that generates feasible solutions, you can see the solution pool in the Problem Browser and further populate it with more nonoptimal solutions. (You can also see feasible solutions in the Solutions tab if certain Language settings are selected; see After running a project and Setting language options.)

You can see variable values in the Problem Browser, which also contains information about data structures, data values, labeled constraints and sensitivity data, as well as postprocessing data.