IBM ILOG OPL V6.3

IBM ILOG OPL IDE Tutorials

Copyright

Trademarks

Acknowledgement

The language manuals are based on, and include substantial material from, The OPL Optimization Programming Language by Pascal Van Hentenryck, © 1999 Massachusetts Institute of Technology.

Table of contents

IDE Tutorials

A collection of tutorials that use examples from the product distribution to illustrate typical features and use cases of the OPL IDE.

In this section

About IDE tutorials
Gives the prerequisites for reading this document and outlines the structure.

Understanding solving statistics and progress (MP models)
Shows how the IDE displays a dynamically updated progress chart for a mixed integer programming (MIP) example that takes some time to solve.

Understanding solving statistics and progress (CP models)
Shows how the IDE displays a dynamically updated progress chart for a constraint programming example that takes a few seconds to solve.

Working with external data
Explains how to connect to data sources from OPL, based on the example of databases, and then provides two practical cases.

Using IBM ILOG Script for OPL
Teaches the features of the IDE for scripts written in IBM ILOG Script for OPL, including the script debugging facilities.

Relaxing infeasible models
Uses the nurses example to demonstrate how the IDE detects conflicts and searches for relaxations in models that appear infeasible after execution.

Profiling the execution of a model

Explains how the IDE Profiler table can help you improve your model to make it run faster while consuming less memory.

Working with the solution pool

Explains how to access a project solution pool in the IDE and how you can set options and define filters on solution pool generation.

Using the performance tuning tool

Describes how to improve the combination of parameters for the CPLEX® part of your model(s).

About IDE tutorials

This manual assumes that you are familiar with the IDE graphical user interface. Read *Getting Started with the IDE* first if this is not the case.

You may also need to read How to read the OPL documentation for details of prerequisites, conventions, documentation formats, and other general information.

This manual is organized as follows:

- *Understanding solving statistics and progress (MP models)*: A relatively long MIP example presents the pause feature and the display in the Progress chart of the Output window.
- *Understanding solving statistics and progress (CP models)*: An example that uses a manufacturing problem solved by the CP Optimizer engine to describe the Statistics and Progress chart in the IDE Output window.
- *Working with external data*: An IP example presents the use of a database through SQL instructions in the data file; the example database is a Microsoft® Access file accessed through ODBC connectivity.
- *Using IBM ILOG Script for OPL*: Several examples illustrate the use of scripting through script statements in model files; the scripting language is IBM ILOG Script for OPL.
- *Relaxing infeasible models*: The nurse scheduling example illustrates how to use the IDE to search for relaxations and conflicts in an MP model that appears infeasible after execution. This feature may help you detect errors in the model or data of your project.
- *Profiling the execution of a model*: A tutorial shows how the IDE can help you improve your model to make it run faster while consuming less memory.
- *Working with the solution pool*: This section explains what solution pools are and how solution pools are populated. An example shows how you can use IBM ILOG Script to work with solution pools.
- *Using the performance tuning tool*: This section explains, with an example, how to use this IDE feature to fine-tune CPLEX® parameters for MP models.

Understanding solving statistics and progress (MP models)

Shows how the IDE displays a dynamically updated progress chart for a mixed integer programming (MIP) example that takes some time to solve.

In this section

Purpose of the tutorial
Describes the type of model used.

The scalable warehouse example
Summarizes the problem.

The scalable warehouse project
Examines the project in the OPL IDE.

Executing the warehouse project with scalable data
Explains how to pause and observe the results of the suspended execution.

Examining the statistics and progress chart (MP)
Describes the engine log, progress chart and statistics table for MP models.

Examining the engine log
Describes the Engine Log tab.

Examining the results and the data
Describes how results are displayed in the IDE after model execution.

Changing a CPLEX parameter value
Explains how to change a mathematical programming option in the IDE.

Purpose of the tutorial

The tutorial assumes that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

The tutorial uses a model solved by the CPLEX® engine to describe the statistics and progress chart in the IDE output view. However, statistics and progress chart work in a very similar manner for models solved with the CP Optimizer engine. Where slight differences can be observed, they are indicated as appropriate.

Starting from a scalable run configuration of the warehouse example presented in *The scalable warehouse example*, this tutorial covers:

- *Executing the warehouse project with scalable data*
- *Examining the results and the data*
- *Changing a CPLEX parameter value*

The scalable warehouse example

The `scalableWarehouse.mod` file is a scalable version of the warehouse location model that is described in detail in *Warehouse location* in the *Language User's Manual*.

This section assumes that you are familiar with the integer and Boolean variables used to solve this problem as explained in that document. Here is a summary the problem:

A company is considering a number of locations for building warehouses to supply its existing stores. Each possible warehouse has a fixed maintenance cost and a maximum capacity specifying how many stores it can support. Each store can be supplied by only one warehouse and the supply cost to the store differs according to the warehouse selected. The application consists of choosing which warehouses to build and which of them should supply the various stores in order to minimize the total cost, which is the sum of the fixed costs and the supply costs.

The model instance used in this section considers 50 warehouses and 200 stores. The fixed costs for the warehouses are all identical and equal to 10. The `warehouse` project folder contains the model **scalableWarehouse.mod**.

The scalable warehouse project

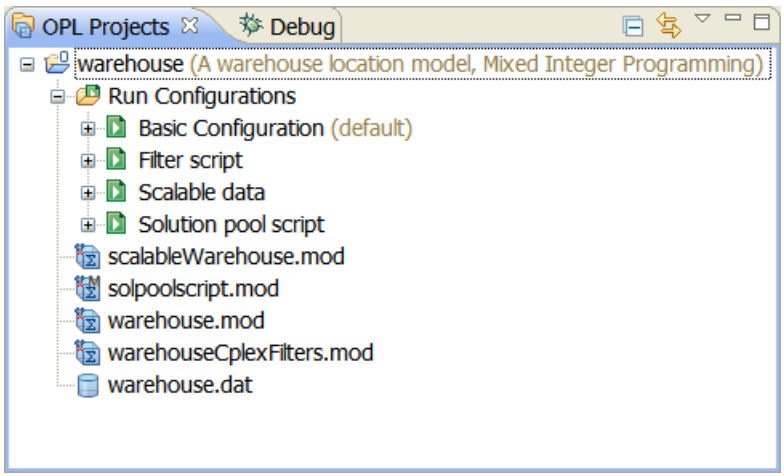
You will now take a look at this project in the IDE.

Use the **File>New>Example** menu command to open the **warehouse** example.

If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*. As a hint, the fastest way to find this example in the wizard is to choose **IBM ILOG OPL examples** on the first screen and then on the second screen type **warehouse** into the field that by default contains **type filter text**. When you do this, all other examples are filtered out, and you can double-click the **warehouse** example to open it.

The project

The `warehouse` project opens in the IDE. Its contents are shown below, with all elements in the OPL Projects Navigator expanded.



The warehouse project

Note that:

- There are two models, `warehouse.mod` and `scalableWarehouse.mod`, used in different run configurations.
- There is a `warehouse.dat` file declaring external data for the `warehouse.mod` model but there is no `scalableWarehouse.dat` file.
- There are four run configurations:
  - The first one associates the basic warehouse model and data.
  - The second one uses a variation of `scalableWarehouse.mod` for you to learn the tuning tool. See *Using the performance tuning tool*.

- The third one has been created specifically from `scalableWarehouse.mod` for you to work with scalable data.
- The fourth one uses another variation of `scalableWarehouse.mod` for you to work with solution pools. See *Working with the solution pool*.

The data

The scalable warehouse model has no associated data file. The numbers of warehouses and stores and the fixed cost are declared within the model file as shown below.

Data initialization in scalableWarehouse.mod

```
int Fixed        = 10;
int NbWarehouses = 50;
int NbStores     = 200;

assert( NbStores > NbWarehouses );

range Warehouses = 1..NbWarehouses;
range Stores     = 1..NbStores;
```

The capacity values and transportation cost values are internal data (that is, they are calculated in the model file) as shown below.

Internal data in scalableWarehouse.mod

```
int Capacity[w in Warehouses] =
  NbStores div NbWarehouses +
  w % ( NbStores div NbWarehouses );
int SupplyCost[s in Stores][w in Warehouses] =
  1 + ( ( s + 10 * w ) % 100 );
```

Note:

1. When there is no separate data file, all the variables must be initialized in the model file; there cannot be statements of the form:

   ```
   int myvar = ...;
   ```

2. The scalable warehouse model has been artificially increased in size so that the search is long enough for you to interrupt and look at feasible solutions that are not the best with respect to the objective. The resulting size is greater than the size allowed in trial mode. You therefore need a full copy of OPL to run this example.

Note the use of the integer division operator `div` in the capacity calculation and the modulus operator `mod`.

The matrix `supply [s][w]` represents the amount of the product delivered to store `s` from warehouse `w`. The total delivered to a store could be represented by a very large integer value. Instead, it is normalized to `1`, so that each supply `[s][w]` value is a proportion of 1 with a floating-point value. Thus, one warehouse could deliver 0.5 (half the total for that store), another 0.2 (a fifth of the total for that store) and so on.

You are now going to execute the run configuration with scalable data and examine the resulting statistics and chart.

Executing the warehouse project with scalable data

Basic Configuration is predefined as the default run configuration to be executed. Since it is not the configuration you want to work on, you will first make another run configuration the default one.

The scalable model has been designed to be solved by the CPLEX® engine in several seconds so that you have time to observe and interact with the execution process. To experience such interaction and observe the results, you are now going to start, then suspend, the execution and observe the output in the **Statistics** tab.

1. Right-click on **Scalable data** and choose **Set as Default**.

   This configuration name is now labeled as the new default.

2. To execute this run configuration, right-click on **Scalable data**, and select **Run this**.

3. In the Execution toolbar, the **Pause the current solve** button  appears. Click this button just after launching the solve to interrupt the process and examine the current solution.

   (See *What happens when you execute a run configuration* in the *IDE Reference* for details.)

4. Click the **Statistics** tab.

   See *Examining the statistics and progress chart (MP)* to understand what you see at this stage of the execution process.

5. The Pause button is replaced with the **Continue the current solve** button  . Click to continue and wait until execution is complete.

   The IDE returns to the running state and completes the solve.

   You will see the shape of the green line that represents the best integer solution change as more iterations are made.

6. Proceed to the next section of the tutorial.

Examining the statistics and progress chart (MP)

Describes the engine log, progress chart and statistics table for MP models.

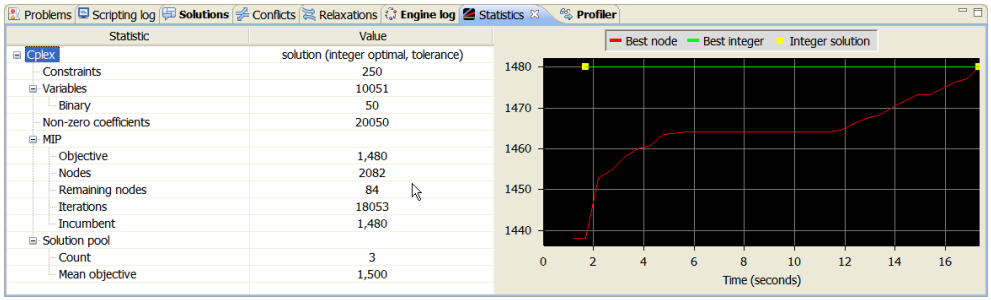
In this section

The Progress Chart (MP models)
Analyses the results of the solve process displayed as a chart in the IDE.

The Statistics table (MP models)
Analyses the results of the solve process displayed as statistics in the IDE.

The Progress Chart (MP models)

The diagram in the right part of the Statistics tab displays the solve process and results as a chart. The vertical axis of this chart is the value of the objective and the horizontal axis is time in seconds. The chart below is displayed after a pause in the solve process.



MP models: Progress Chart at feasible solution (`scalableWarehouse.mod`)

The shapes of the lines on the chart depend on exactly when you pause execution; the progress chart shown above is just an example of a first pause.

The progress chart shows the variation of the best node and best integer values and highlights the integer values found during the search:

- The green line shows the evolution of the Best Integer value, that is, the best value of the objective found that is also an integer value.
- The red line shows the evolution of the best value of the remaining open nodes (not necessarily integer) when moving from one node to another. This gives a bound on the final solution.
- The yellow point indicates a node where an integer value has been found. These points generally correspond to the stars (asterisks) in the Engine Log. See also the **Engine Log** tab.

The values in the **Value** column are dynamic and are updated every second; they change to indicate how the algorithm is progressing. The values in the **Statistic** frame are static; they indicate the model characteristics.

Since one feasible solution has been found for `scalableWarehouse.mod`, this is listed in the **Solutions** tab.

The Statistics table (MP models)

The table in the left part of the Statistics tab reflects the contents of the Engine Log hierarchically as a tree. The table shown below is displayed at the end of the solve process for `scalableWarehouse`.

Statistic	Value
Cplex	solution (integer optimal, tolerance)
Constraints	250
Variables	10051
Binary	50
Non-zero coefficients	20050
MIP	
Objective	1,480
Nodes	2082
Remaining nodes	84
Iterations	18053
Incumbent	1,480
Solution pool	
Count	3
Mean objective	1,500

MP models: results for `scalableWarehouse.mod`

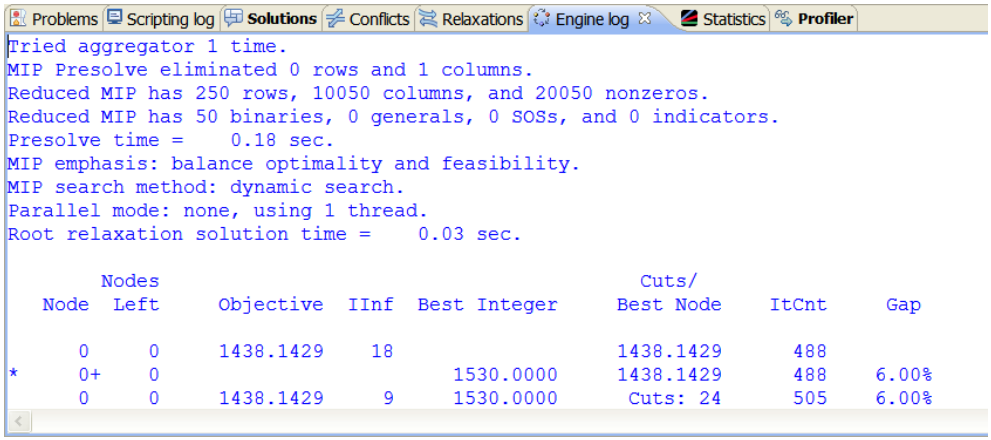
The top level of the Statistic tree indicates the solving engine (CPLEX®). You see certain statistic values change dynamically as the solving takes place.

For most algorithms, the statistics items are:

- the **CPLEX problem**
  - Number of constraints and variables, in a format similar to that of the Engine Log, with various characterizations.
  - Number of non-zero coefficients
  - Number of quadratic constraints and coefficients, when applicable
- the **presolve** stage (not shown in the `scalableWarehouse` example)
  - Number of rows and columns removed, when applicable
- the **solve** stage, with the names of the algorithms used, such as MIP, Barrier, Simplex, and so on, with the corresponding statistics for each of them.

Examining the engine log

The **Engine Log tab** in the Output window shows the CPLEX® node log when an LP model is solved.



Engine log for the warehouse example (`scalable warehouse.mod`)

When CPLEX® optimizes mixed integer programs, it builds a tree with the linear relaxation of the original MIP at the root, and subproblems to optimize at the nodes of the tree. CPLEX reports its progress in optimizing the original problem in a node log file as it traverses this tree. You control how information in the log file is recorded and displayed through two CPLEX parameters.

- The `MIPDisplay` parameter controls the general nature of the output that goes to the node log.
- The MIP node log interval parameter, `MIPInterval`, controls how frequently node log lines are printed. Its default value is 100.

These parameters can be set in the OPL settings file. See *Changing a CPLEX parameter value*. The values for these parameters are also given in CPLEX parameters and OPL parameters.

CPLEX records a line in the node log for every node with an integer solution if the parameter controlling MIP node log display information (`MIPDisplay`) is set to 1 or higher. If `MIPDisplay` is set to 2 or higher, then for any node whose number is a multiple of the `MIPInterval` value, a line is recorded in the node log for every node with an integer solution.

CPLEX logs an asterisk (*) in the left-most column for any node where it finds an integer-feasible solution or new incumbent. In the next column, it logs the node number. It next logs the number of nodes left to explore.

In the **Objective** column, CPLEX either records the objective value at the node or a reason to fathom the node. (A node is fathomed if the solution of a subproblem at the node is infeasible; or if the value of the objective function at the node is worse than the cutoff value for branch & cut; or if the linear programming relaxation at the node supplies an integer solution.) This column is left blank for lines that report that CPLEX found a new incumbent by primal heuristics. A plus (+) after the node number distinguishes such lines.

In the column labeled **IInf**, CPLEX records the number of integer-infeasible variables and special ordered sets. If no solution has been found, the column titled **Best Integer** is blank; otherwise, it records the objective value of the best integer solution found so far.

The column labeled **Cuts/Best Node** records the best objective function value achievable. If the word Cuts appears in this column, it means various cuts were generated; if a particular name of a cut appears, then only that kind of cut was generated.

The column labeled **ItCnt** records the cumulative iteration count of the algorithm solving the subproblems.

Until a solution has been found, the column labeled **Gap** is blank. If a solution has been found, the relative gap value is printed: when it is less than 999.99, the value is printed; otherwise, hyphens are printed. The gap is computed as:

```
(best integer - best node ) * objsen / (abs (best integer) + 1e-10)
```

Consequently, the printed gap value may not always move smoothly. In particular, there may be sharp improvements whenever a new best integer solution is found. Moreover, if the populate procedure of the solution pool is invoked, the printed gap value may become negative after the optimal solution has been found and proven optimal.

CPLEX also logs its addition of cuts to a model. Cuts generated at intermediate nodes are not logged individually unless they happen to be generated at a node logged for other reasons. CPLEX logs the number of applied cuts of all classes at the end.

CPLEX also indicates, in the node log file, each instance of a successful application of the node heuristic. The + denotes an incumbent generated by the heuristic. Periodically, if the MIP display parameter is 2 or greater, CPLEX records the cumulative time spent since the beginning of the current MIP optimization and the amount of memory used by branch & cut. (For example, if the `MIPInterval` parameter is set to 10, time and memory information appears either every 20 nodes or ten times the MIP interval parameter, whichever is greater.)

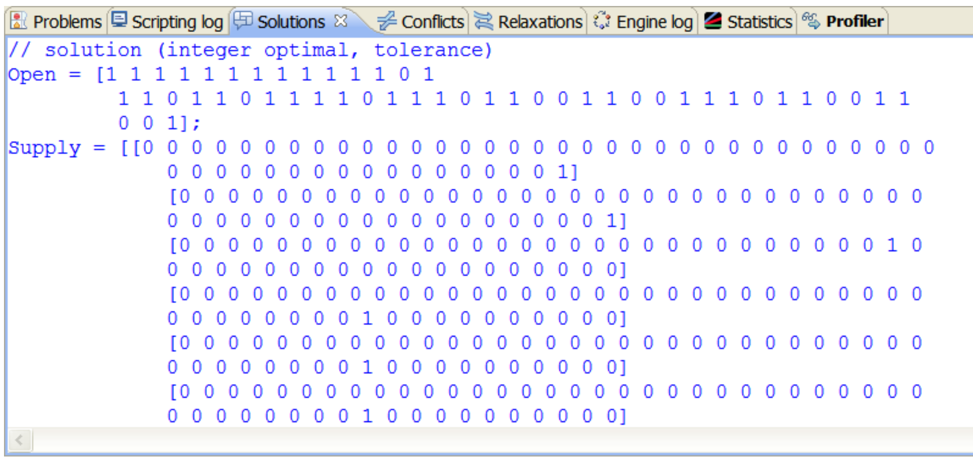
CPLEX prints an additional summary line in the log if optimization stops before it is complete. This summary line shows the best MIP bound, that is, the best objective value among all the remaining node subproblems.

Examining the results and the data

After model execution, the results are displayed in the **Solutions** tab and the data structure is displayed in the **Problem browser**.

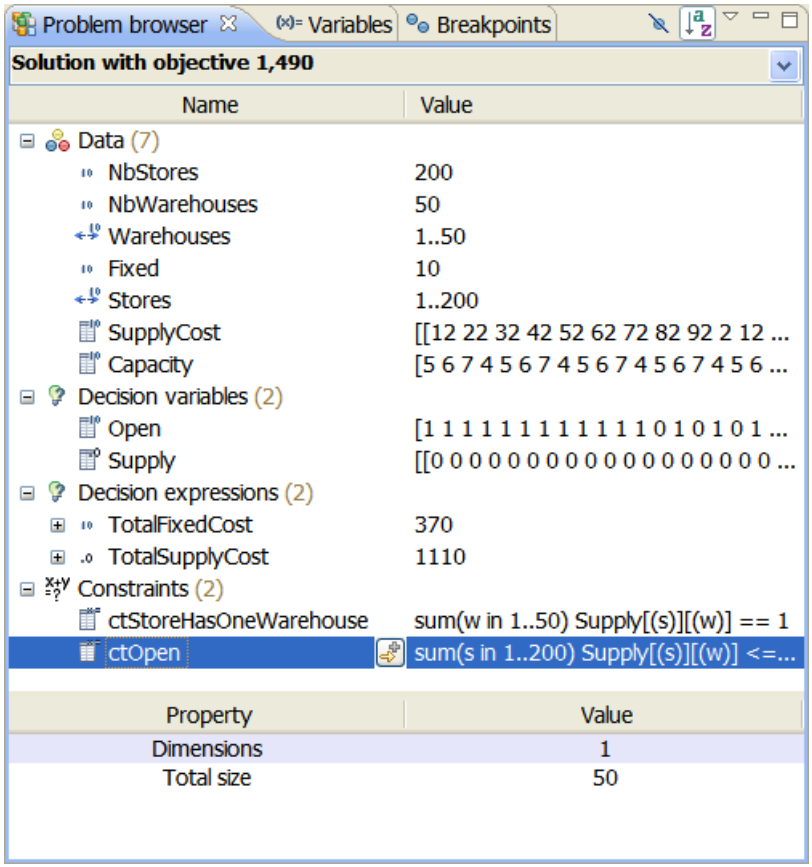
The results

If you scroll down through the **Solutions** tab, you see the results for the `Open` and `Supply` variables. The `Open` array specifies which warehouses are open. The `Supply` matrix specifies which stores are supplied by which (open) warehouses.



The data structure

After execution, the data structure for `scalableWarehouse.mod` is displayed in the Problem Browser.



Viewing the model structure in the Problem Browser (`scalableWarehouse.mod`)

The navigation tools of the IDE are available for your use at this time.

- You can view a model element in a separate table by double-clicking it.
- If you click the drop-down list at the top, you can see the final solution and, if the model expresses a MIP problem, all the solutions of the pool. See *Working with the solution pool* for details.
- You can select an item and see its property in the **Property** column.

See *Understanding the Problem Browser* in *Getting Started with the IDE* for details.

Changing a CPLEX parameter value

Changing CPLEX® parameters is useful to experiment with different values. The convenient way is to create a settings file and a different run configuration for each value, or set of values, you want to test. However, if you decide eventually that a particular setting is always needed for the model concerned, you can even set the parameter within the model by writing an `execute` IBM ILOG Script block. For an example, see *Changing CPLEX parameters* in the *Language User's Manual*.

In the scalableWarehouse example, if you increase the relative MIP gap tolerance to 0.05 (5%), the first solution found is immediately considered to be the final solution because it is at most 3.3333% from the optimal solution (as displayed in the notification message).

To set a CPLEX parameter in the IDE:

1. Right-click on **Scalable data** and select **New>Settings** to create a settings file with an `.ops` extension. Name the file **warehouse.ops**.

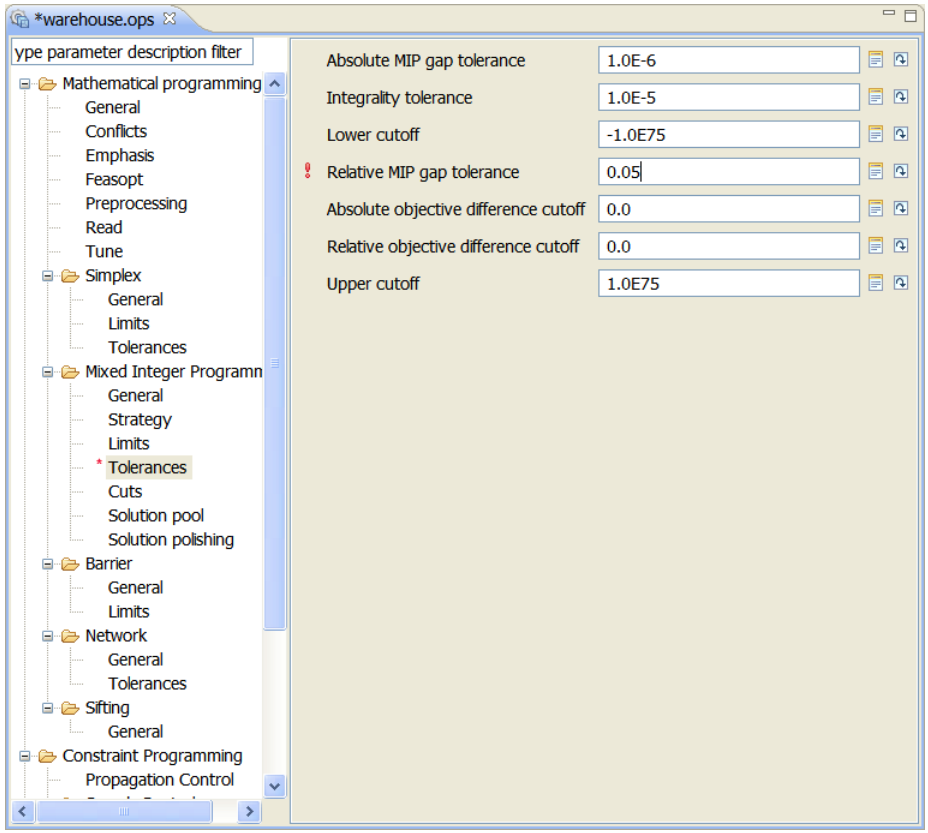
   This is explained in *Adding a settings file* in *Getting Started with the IDE*.

2. Add the new **warehouse.ops** settings file to the run configuration **Scalable data**, using drag and drop.

3. Click **Mixed Integer Programming>Tolerances** in the option tree on the left.

4. Enter the value **0.05** for **Relative MIP gap tolerance** and press **Enter**.

   Because this value is not the default value, a red exclamation mark appears to the left of the option name.



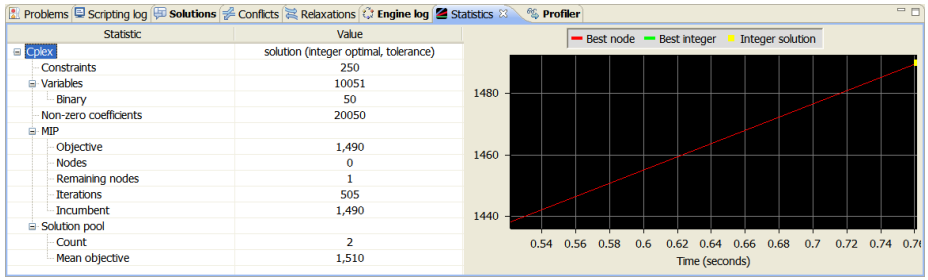
Changing the value of a CPLEX parameter

You can also create a specific run configuration as explained in *Creating and executing a different configuration* in *Getting Started with the IDE*.

5. Click the **Run** button  to rerun the same configuration (Scalable data). If you are prompted to save, click **OK**.

   Execution ends almost immediately and the progress chart reflects this fact.

> **Note**: The **Run** button re-executes the last executed configuration.



MP models: Progress Chart, New MIP gap tolerance (`scalableWarehouse.mod`)

There are fewer solutions in the **Solutions** tab and in the Problem Browser list than when the gap tolerance was not user-defined.

You can now close the warehouse example (see *Closing projects* in *IDE Reference* for details if necessary).

Understanding solving statistics and progress (CP models)

Shows how the IDE displays a dynamically updated progress chart for a constraint programming example that takes a few seconds to solve.

In this section

Purpose of the tutorial
Describes the type of model used.

The steel mill example
Summarizes the problem.

Executing the model in the OPL IDE
Explains how to solve the steel mill problem and examine the results.

Coloring of CP keywords and functions in the IDE
Constraint programming in the OPL IDE is distinguished by the colored names of the functions.

Examining the statistics and progress chart (CP)
Describes the statistics and progress chart for CP models.

Examining the engine log
Describes the contents of the CP engine log displayed in the OPL IDE.

Changing a CP parameter value
Explains how to change a constraint programming option in the IDE.

Purpose of the tutorial

In this tutorial, you will see how the IDE displays a dynamically updated progress chart for a constraint programming (CP) example.

The tutorial assumes that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

The tutorial uses a manufacturing problem solved by the CP Optimizer engine to describe the statistics and progress chart in the IDE Output window. Statistics and progress chart work in a very similar manner for models solved with the CPLEX® engine. Slight differences can be observed.

For more information, see the topics dedicated to constraint programming in the *Language User's Manual*.

The steel mill example

The steel mill problem consists in building steel coils from slabs that are available in a work-in-process inventory of semi-finished products. It is described in The steel mill problem in the Samples manual.

Executing the model in the OPL IDE

To execute the `steelmill` example:

1. Select **File>New>Example** and use the New Example wizard to open the **steelmill** example if you have not already done so.
2. To run the model, open **Run Configurations**, right-click on **Basic Configuration**, and select **Run this**.

   See *What happens when you execute a run configuration* in *IDE Reference* for details.)
3. Click the **Engine Log** tab in the Output window and examine its contents as you did in the previous tutorial.
4. Click the **Statistics** tab in the Output window.

   See *Examining the statistics and progress chart (CP)* to understand what you see at this stage of the execution process.

Coloring of CP keywords and functions in the IDE

If you double-click **steelmill.mod** to open it in the IDE editor, you will notice that the first support of constraint programming in the IDE is the colored names of the functions used in the model to define specialized constraints, for example `pack` on line 48.

Examining the statistics and progress chart (CP)

Describes the statistics and progress chart for CP models.

In this section

Statistics
Describes the contents of the CP Statistics tab displayed in the OPL IDE.

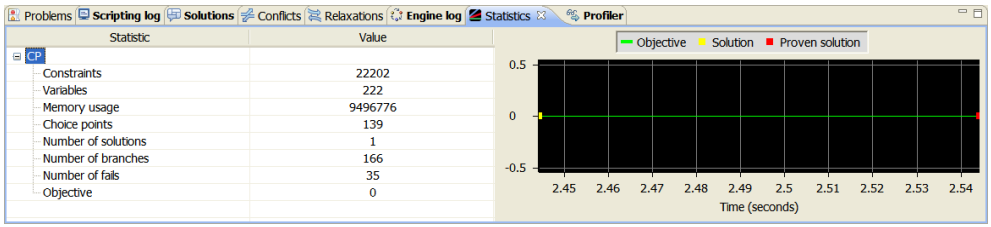
The progress chart (CP models)
Analyses the results of the solve process displayed as a chart in the IDE.

The statistics table (CP models)
Analyses the results of the solve process displayed as statistics in the IDE.

Statistics

The **Statistics** tab of the IDE Output window also provides information on the search such as number of choice points, fails, and memory usage.

When you click the **Statistics** tab in the Output window, you can see the CP Optimizer solving statistics.



Statistics for the steel mill example (default CP settings)

Pausing search

You can pause the CP Optimizer search, explore the current feasible solution (if any) in the Problem Browser and then continue the search. This is not meaningful in the steel mill example, however, as the solution is found very quickly.

Progress chart

The progress chart shows all the feasible solutions, and the final one. Again, the progress chart is not particularly interesting for the steel mill example because the optimal solution is found immediately. See the progress chart of the `route` problem in *The progress chart (CP models)* for comparison.

The progress chart (CP models)

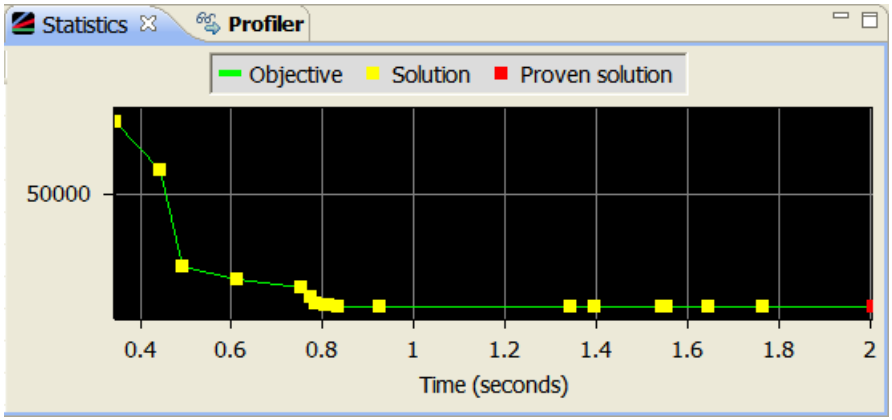
If you execute a constraint programming model solved with the CP Optimizer engine, the progress chart is slightly different from the progress chart of CPLEX® models. To observe it, you are going to open the `route` project.

1. Use the **File>New>Example** menu command to open the **Call route** example.

   The `route` project will appear in the OPL Projects navigator.

2. To run the model, open **Run Configurations**, right-click on **Basic Configuration**, and select **Run this** from the popup menu to execute the run configuration.

3. Click the **Statistics** tab.



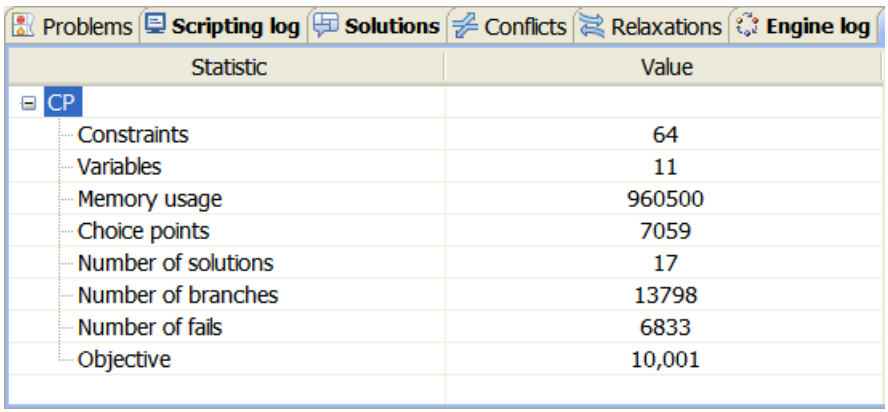
CP models: progress chart (`route` project)

The progress chart shows the variation of the objective. The yellow points show when solutions are found and the green line basically follows these points, showing the evolution of the best solution found over time.

The statistics table (CP models)

The statistics table and progress chart work in a very similar manner for models solved with the CP Optimizer engine. However, take a look at the `route` example to observe the differences.

Click the **Statistics** tab after executing the **Basic Configuration** for the `route` project. The statistics are on the left.



CP models: Statistics table (`route` project)

The top level of the Statistic tree indicates the solving engine (CP). You see certain statistic values change dynamically as the solving takes place. The statistics items are:

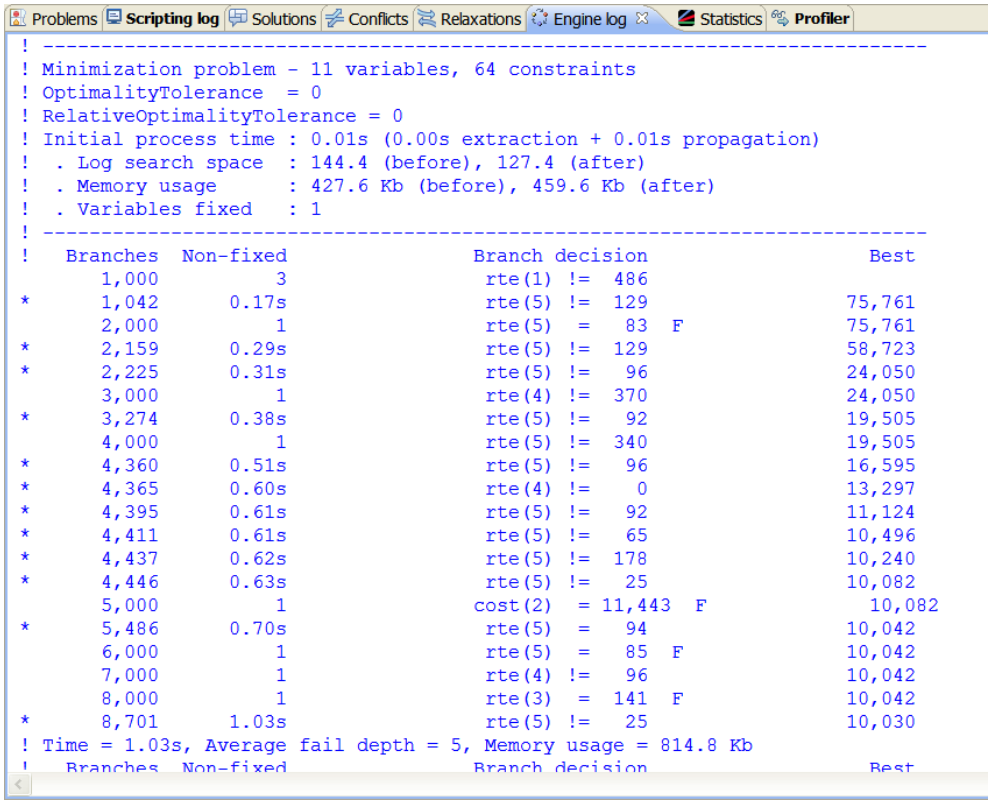
- Number of constraints after initial propagation
- Number of variables
- Memory usage (including after initial propagation)
- Number of choice points
- Number of fails
- The objective

Click on the **Solutions** tab to see the number of solutions obtained.

You can change constraint programming parameters by changing values in the **Constraint Programming** page of the settings editor. See also Constraint programming options in *IDE Reference*.

Examining the engine log

When you click the **Engine Log** tab in the Output window, you can see the CP Optimizer log.



Engine Log for the Steel Mill example (default CP settings)

The first line of the log indicates the type of problem, along with the number of decision variables and constraints in the model. In the steel mill example, there is an objective included in the model, so the problem is reported to be a Minimization problem. When the model does not include an objective, the problem type is reported as a Satisfiability problem.

In this example, we have one line each 50 branches as indicated by the parameter set in the model `steelmill.mod`.

```
cp.param.LogPeriod = 50;
```

The next three lines of the log provide information regarding the initial constraint propagation.

The **Initial process time** is the time in seconds spent at the root node of the search tree where the initial propagation occurs. This time encompasses the time used by the optimizer to load the model, called extraction, and the time spent in initial propagation.

The value for **Log search space** provides an estimate of the size of the depth-first search tree; this value is the log (base 2) of the products of the domains sizes of all the decision variables of the problem. Typically, the estimate of the size of the search tree should be smaller after the initial propagation, as choices will have been eliminated. However, this value is always an overestimate of the log of the number of remaining leaf nodes of the tree because it does not take into account the action of propagation of constraints at each node.

The memory used by the optimizer during the initial propagation is reported. Also, any parameter change from its default is displayed at the head of the search log.

In order to interpret the remainder of the log file, you may want to think about the search as a binary tree. The root of the tree is the starting point in the search for a solution; each branch descending from the root represents an alternative choice or decision in the search. Each of these branches leads to a node where constraint propagation during search will occur. If the branch does not lead to a failure and a solution is not found at a node, the node is called a choice point. The optimizer can make an additional decision and create two new alternative branches from the current node, or it can jump in the tree and search from another node.

The lines in the next section of the progress log, are displayed periodically during the search and describe the state of the search. The display frequency of the progress log can be controlled with parameters of the optimizer. See *Changing a CP parameter value*.

The progress information given in a progress log update includes:

Branches: the number of branches explored in the binary search tree.

Non-fixed: the number of uninstantiated (not fixed) model variables.

Branch decision: the decision made at the branch currently under consideration by the optimizer

Best: the value of the best solution found so far, in the case of an optimization problem.

The final lines of the log provide information about the entire search, after the search has terminated. This information about the search includes:

Solution status: the conditions under which the search terminated.

Number of branches: the number of branches explored in the binary search tree.

Number of fails: the number of branches that did not lead to a solution.

Total memory usage: the memory used by IBM ILOG Concert Technology and the IBM ILOG CP Optimizer engine

Time spent in solve: the elapsed time from start to the end of the search displayed in seconds.

Search speed: average time spent per branch.

The CP Engine Log enables you to trace the propagation (see *Changing a CP parameter value*).

Changing a CP parameter value

Changing CP Optimizer parameters is useful to experiment with different values. The convenient way is to create a settings file in the OPL IDE and a different run configuration for each value, or set of values, you want to test. However, if you decide eventually that a particular setting is always needed for the model concerned, you can even set the parameter within the model by writing an `execute` IBM ILOG Script block. For an example, see *Changing CP parameters* in the *Language User's Manual*.

For example, you can change the verbosity of the log displayed in the Engine Log tab of the Output window.

To set a CP parameter in the IDE:

1. Create a settings file for the `steelmill` project by choosing **File>New>Settings** from the main menu.

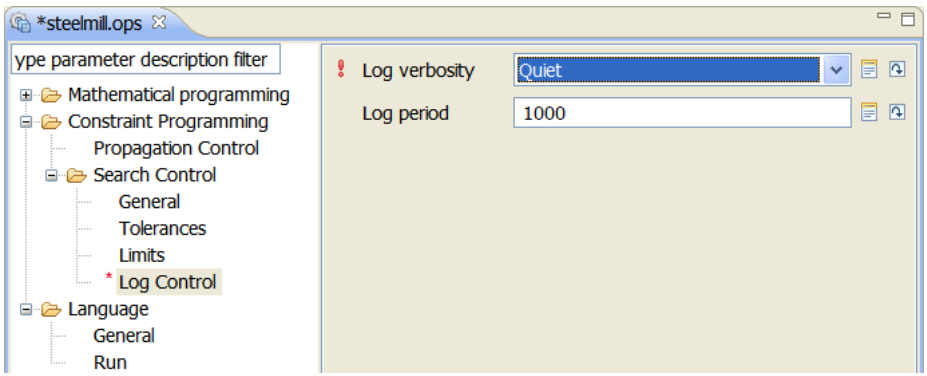
   In the popup window, type **steelmill.ops** as the name of the settings file and click **Finish**.

2. Drag and drop the **steelmill.ops** settings file into the **Basic Configuration** run configuration.

   See also *Adding a settings file* in *Getting Started with the IDE*.

3. In the settings file displayed in the IDE editor, click **Constraint Programming>Search control>Log control**.

4. Select **Quiet** from the **Log verbosity** drop-down list (see *Setting the Engine Log verbosity*).



Setting the Engine Log verbosity

5. Execute the **Basic Configuration** for the project.

   The **Engine Log** tab is empty. This is the result of the **Quiet** setting.

You can now close the **route** and **steelmill** projects in the OPL IDE (see *Closing projects* in *IDE Reference* for details if necessary).

Working with external data

Explains how to connect to data sources from OPL, based on the example of databases, and then provides two practical cases.

In this section

Purpose and prerequisites
Describes the type of model used and outlines the assumed knowledge.

Using data sources
As an example of working with databases, explains how to connect to a database using OPL instructions.

The oil database example
Describes the oil database tables and project, and its execution.

The oil sheet example
Describes the oil data spreadsheet, the oil sheet project, and its execution.

Purpose and prerequisites

IBM® ILOG® OPL allows you to read data from a data source (a database or calculation spreadsheet). In this case, the data is said to be “external” to OPL. (As opposed to the data in an internal OPL `.dat` file.)

The tutorial uses a model solved by the CPLEX® engine, but all the features described work in the same way with a model solved by the CP Optimizer engine.

Both examples are variants of a blending application. This tutorial therefore assumes that:

- you know how to define, instantiate, and initialize data: see *Data sources* in the *Language Reference Manual*,
- you know how to work with projects in the IDE: see *Getting Started tutorial*,
- you are familiar with the blending application and the solving strategy as described in *A blending problem* in the *Language User’s Manual*.

Using data sources

Databases supported

IBM® ILOG® OPL interfaces with some of the RDBMS supported by IBM ILOG DBLink. For a list of the databases supported by OPL, see *Supported databases* in Working Environment.

Connection prerequisites

- If you are working with Oracle:
  - As for most database management systems, you must have a user account and a password allowing you to connect to an existing database.
  - The Oracle client must be installed or accessible through the network.
- If you are working with ODBC, having Microsoft® Access installed will allow you to view the contents of the tables in the database. However you do not have to install this product on your computer.
- In order to follow the interactions with the database, some knowledge of the syntax of the query language SQL is useful.

Connection to a database

The OPL keyword `DBConnection` establishes a connection to a database. It requires two arguments: the database client you want to use and the connection string.

The first argument is a string indicating the name of a database system known to IBM ILOG DBLink, which is a value such as `oracle9`.

Note: Before using a database connection, you must make sure that the corresponding database client is correctly installed on your system.

The connection string passed as second argument must respect a format that depends on the target RDBMS.

For an example of connecting to an Oracle database, see *Connection to Oracle*.

For an example of connecting to a Microsoft® Access database through ODBC connectivity, see *The oil database example*.

Connection to Oracle

If you are using an Oracle database, you should adapt the `DBConnection` statement to this case. For example, in this connection string:

```
DBConnection db("oracle9", "scott/tiger@ilog");
```

the user `scott` with the password `tiger` wants to connect to the Oracle database called `ilog`.

- The string passed as first argument must take the value `oraclex`, where x depends on the version of Oracle that the DB Link driver was built upon. A possible value is `oracle9`.
- The connection string passed as second argument must respect the format:

  ```
  [user]/[password][@SQL Net id]
  ```

  where:

  - `user` and `password` indicate the user name and the password that the database administrator has already assigned to you.
  - the field `SQL Net id` has the format `<instance name>` for SQL Net V2

The oil database example

Describes the oil database tables and project, and its execution.

In this section

Description of the example
Includes what you will learn and where to find the files.

The oil database tables
Describes how to view the contents of these tables via Microsoft Access.

The oil database project
Describes the model file in the project.

The oil database data file
Describes the data file in the project.

Executing the oil database example
Execute the example and examine the result in the OPL IDE.

Viewing the result in the database
How to display the `Result` table after execution using Microsoft Access.

The Result table after execution
Shows the modified `Result` table after execution.

Description of the example

Make sure you have read the section *Using data sources* before you start.

The oil database example is a linear programming (LP) problem delivered as a run configuration of the `oil` project with the supplied database `oilDB.mdb`. In this example, the data is stored in tables in a relational database. This data reflects an optimization problem described in *A blending problem* in the *Language User's Manual*. Here is a summary the problem:

An oil company manufactures different types of gasoline by blending different types of crude oil. Each type of gasoline must satisfy quality criteria with respect to its lead content and octane rating. The company must satisfy customer demand, which is 3,000 barrels a day of super, 2,000 of regular, and 1,000 of diesel. It can purchase up to 5,000 barrels of each type of crude oil a day and process at most 14,000 barrels a day. It costs four dollars to transform a barrel of oil into a barrel of gasoline.

The company has the option of advertising a type of gasoline, in which case the demand for this type increases by ten barrels for every dollar spent.

What you are going to do

With this example, you will learn how to:

- establish a connection to a database from the IDE,
- read database tables into OPL sets,
- create a new relational table in a database from the IDE,
- write an OPL tuple set to a database by inserting the tuples as new rows in a table.

Where to find the files

For this example, you will use the following files.

- `oil` project: the oil blending example, in which one run configuration uses an ODBC connection to a Microsoft® Access database where there is data for a linear programming (LP) problem.
- `oilDB.mdb`: the Microsoft Access database that contains the data for the oil database example.

You can find the files in the `oil` folder:

`<OPL_dir>\examples\opl\oil`

where `<OPL_dir>` is your installation directory.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example,

leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

The oil database tables

The data for the Oil example is stored in the following relational tables:

```
GasData
OilData
Result
```

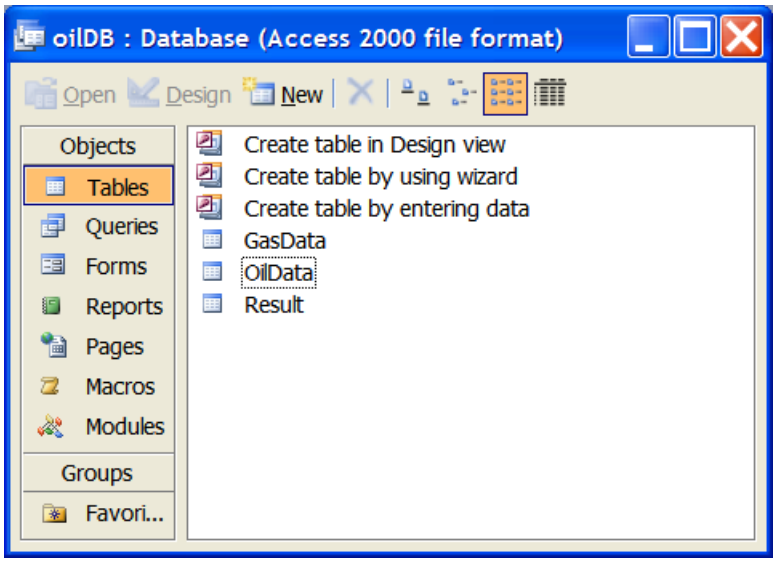
If you have Microsoft® Access installed on your computer, you can view the contents of any of these tables by simply double-clicking the database file in Windows Explorer.

You should view this file in the **copy** of the project you created when you opened the example using the New Example wizard. If you did not change any of the default settings in the New Example wizard, that should be in your default workspace, at the following location:

```
C:\Documents and Settings\<user>\Application Data\ILOG\OPL Studio IDE\<version>\
oil\oilDB.mdb
```

Microsoft® Access opens the database and displays the tables in alphabetical order. Double-click the table whose contents you want to see.

The data structure for the oil database example and the oil sheet example is the same.



The tables in the oil database

The GasData table

The five-column table `GasData` stores the following values for each type of gasoline manufactured:

```
name demand price octane lead
```

Each row is for one type of gasoline, with the product name of the gasoline stored as a character string in the first column. The daily customer demand for each type of gasoline is recorded and the price in this table is the sales price. The octane rating must be at least the value stored and the lead content must not exceed the value stored.

The following `GasData` table is the table as you see it in Microsoft® Access.

GasData : Table

name	demand	price	octane	lead
Super	3000	70	10	1
Regular	2000	60	8	2
Diesel	1000	50	6	1

The GasData table

The OilData table

The five-column table `OilData` stores the following values for each type of crude oil:

```
name capacity price octane lead
```

Each row is for one type of crude, with the name of the crude stored as a character string in the first column. The capacity figure represents the amount that can be purchased each day and the price is the purchase price. The other columns store the octane rating and the lead content.

The following `OilData` table is the table as you see it in Microsoft Access.

OilData : Table

name	capacity	price	octane	lead
Crude1	5000	45	12	0.5
Crude2	5000	35	6	2
Crude3	5000	25	8	3

The OilData table

The Result table

The table `Result` gives the results in rows for each crude/gasoline combination. It shows the blend as the number of barrels of each crude used for each type of gasoline and the amount, `a`, spent on advertising the type of gasoline.

Result : Table

	oil	gas	blend	a
	Crude1	Super	2088.889	0
	Crude1	Regular	2111.111	750
	Crude1	Diesel	800	0
	Crude2	Super	777.7778	0
	Crude2	Regular	4222.222	750
	Crude2	Diesel	0	0
	Crude3	Super	133.3333	0
	Crude3	Regular	3166.667	750
	Crude3	Diesel	200	0
▶				

The Result table

The results for the data supplied are already included in the supplied database; when you execute the model, the table `Result` is deleted and re-created.

Note: If you wanted, you could experiment by changing values in these tables before executing the model. For example, changing the prices in the GasData table to **65** (Super) and **55** (Diesel), is shown in the modified `GasData` table below:

GasData : Table

	name	demand	price	octane	lead
	Super	3000	65	10	1
	Regular	2000	60	8	2
✎	Diesel	1000	55	6	1
*					

Record: 3 of 3

The modified GasData table

You do not need to do this before continuing the tutorial. This is informational, in case you want to modify the data in the tables on your own.

The oil database project

In the IDE, use the **File>New>Example** menu command to open the **oil** example.

The IDE displays the `oil` project in the OPL Projects navigator. Open the model file `oilDB.mod` in the editing area.

Set definitions in the model

At the beginning of the model file, there are definitions of sets of string values to hold the names of the gasolines and oils.

Set definitions (`oilDB.mod`)

```
{string} Gasolines = ...;
{string} Oils = ...;
```

Tuple definitions in the model

These are followed by definitions of tuples for the data, as shown in *Tuple definitions (`oilDB.mod`)*.

Tuple definitions (`oilDB.mod`)

```
tuple gasType {
   string name;
   float demand;
   float price;
   float octane;
   float lead;
}

tuple oilType {
   string name;
   float capacity;
   float price;
   float octane;
   float lead;
}
```

These tuple definitions closely follow the structure of the rows in the `GasData` and `OilData` tables, respectively, of the database.

You will notice the type concordance between the table columns and the OPL fields. The column `name` in each table contains character strings, so the field `name` in each tuple is of type `string` ; the column `lead` in each table contains floating-point values, so the field `lead` in each tuple is of type `float`.

Note: Some columns that appear to store integer values need to be mapped to OPL fields of type `float`, rather than of type `int`. This is to avoid integer arithmetic or, if you are

using an Access database source, because the numeric values manipulated in Access are of type `float`.

Tuple sets and arrays in the model

The rows in the Access **GasData** and **OilData** tables are first mapped to sets of tuples, `gasData` and `oilData`, in OPL.

Tuple sets (`oilDB.mod`)

```
{gasType} GasData = ...;
{oilType} OilData = ...;
```

Note: OPL supports sets of tuples as well as sets of `int`, `float`, and `string` values.

The tuple sets are then preprocessed using generic indexed arrays (see *Initializing Arrays* in the *Language User's Manual*) to obtain one-dimensional arrays, `gas` and `oil`, in OPL.

Tuple sets preprocessed as generic indexed arrays (`oilDB.mod`)

```
gasType Gas[Gasolines] = [ g.name : g | g in GasData ];
oilType Oil[Oils] = [ o.name : o | o in OilData ];
```

The preprocessing is done by `execute` statements in IBM ILOG Script for OPL. *The transportation example* explains such statements.

Table loading

In the `oilDB2.mod` example, you access the one-dimensional array right from the database, as shown in the following code extract from the data file `oilDB2.dat`.

Reading database columns to a tuple (`oilDB2.dat`)

```
Gasolines,Gas from DBRead(db,"SELECT name,name,demand,price,octane,lead FROM
GasData");
Oils,Oil from DBRead(db,"SELECT name,name,capacity,price,octane,lead FROM
OilData");
```

The oil database data file

Double-click the `oilDB.dat` file in the project tree to see the contents of the data file.

Data for the Oil Database example (`oilDB.dat`)

```
DBConnection db("access","oilDB.mdb");
Gasolines from DBRead(db,"SELECT name FROM GasData");
Oils from DBRead(db,"SELECT name FROM OilData");
GasData from DBRead(db,"SELECT * FROM GasData");
OilData from DBRead(db,"SELECT * FROM OilData");
MaxProduction = 14000;
ProdCost = 4;
DBExecute(db,"drop table Result");
DBExecute(db,"create table Result(oil varchar(10), gas varchar(10), blend real,
 a real)");
Result to DBUpdate(db,"INSERT INTO Result(oil,gas,blend,a) VALUES(?,?,?,?)");
```

Notice that the data file starts with the `DBConnection` statement used to connect to the database.

Note: You can have multiple data files and, within any of them, multiple connections to databases.

Connecting to the database from OPL

The connection is established to an Access database by the following statement in `oilDB.dat`:

```
DBConnection db("access","oilDB.mdb");
```

The string passed as first argument indicates that you want to connect to an Access database. The string passed as second argument designates the Access database file. This is a helper implementation actually based on ODBC and you do not need to specify the full path name. Path names are resolved relatively to the directory of the current `.dat` file.

Reading from the database

Reading database columns

You can read columns from any table into an OPL set using the `DBRead` statement, as shown in `oilDB.dat`.

Reading database columns to an OPL set (`oilDB.dat`)

```
Gasolines from DBRead(db,"SELECT name FROM GasData");
Oils from DBRead(db,"SELECT name FROM OilData");
```

You can also read from any table into a tuple and its indexing set, as shown in the data file `oilDB2.dat`.

Reading database columns to a tuple array (`oilDB2.dat`)

```
Gasolines,Gas from DBRead(db,"SELECT name,name,demand,price,octane,lead FROM
GasData");
Oils,Oil from DBRead(db,"SELECT name,name,capacity,price,octane,lead FROM
OilData");
```

Reading database columns to a tuple array *(`oilDB2.dat`)* is more efficient than *Reading database columns to an OPL set* *(`oilDB.dat`)* because

- the code is shorter
- data is not duplicated

Reading database rows

You can read rows from any table into an OPL tuple set using the `DBRead` statement, as shown in `oilDB.dat`.

Reading database rows (`oilDB.dat`)

```
GasData from DBRead(db,"SELECT * FROM GasData");
OilData from DBRead(db,"SELECT * FROM OilData");
```

Note that the data file also initializes some variables directly; it does not only take data from the database.

Creating a new table and updating the database

At the end of the optimization process, you need to store the optimal blends and advertising expenditures in a new database table.

Use the OPL statement `DBExecute` in the data file to create a new table called `Result`, which has four columns, corresponding to the fields in the tuple `Result`.

With Microsoft Access, the instructions to delete the Result table if it exists and then (re)create it are:

Deleting and recreating the Result table

```
DBExecute(db,"drop table Result");
DBExecute(db,"create table Result(oil varchar(10), gas varchar(10), blend real,
 a real)");
```

You can then insert the `oil`, `gas`, `blend`, and `a` arrays as columns in the table `Result` using a `DBUpdate` statement.

With Microsoft® Access, the insertion is made by the instruction:

Creating a Result table

```
Result to DBUpdate(db,"INSERT INTO Result(oil,gas,blend,a) VALUES(?,?,?,?)");
```

The difference between the `DBUpdate` instruction with ODBC (Microsoft Access) and the `DBUpdate` instruction with Oracle lies in the different syntax for the placeholders inside the SQL request, imposed by the two database systems. In the case of ODBC, you use a query

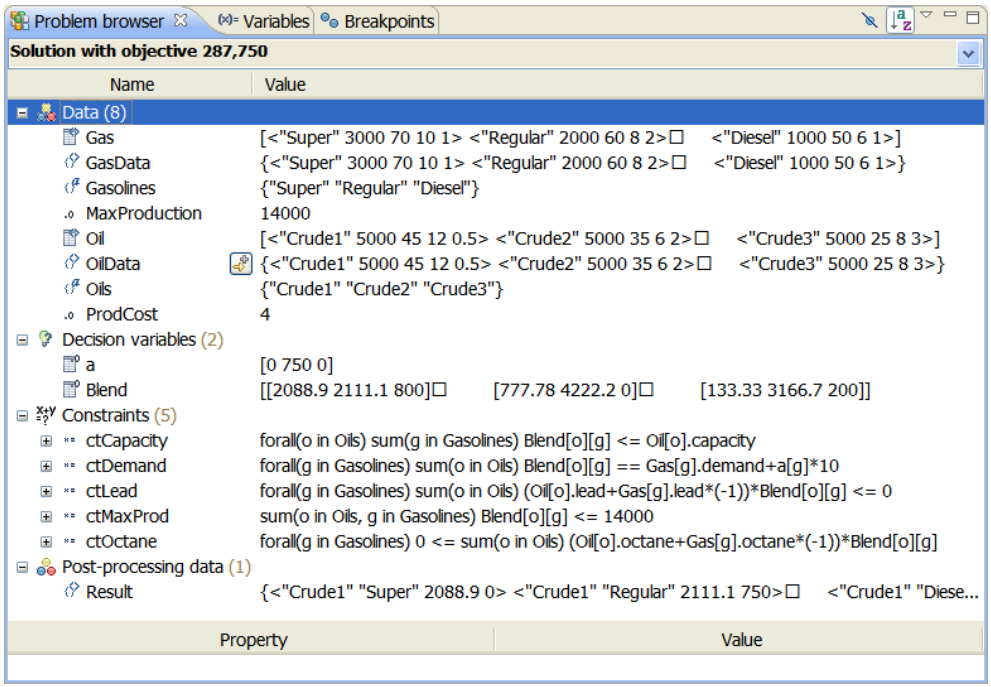
sign as a placeholder, while in Oracle you use a column sign followed by a column number, with the columns numbered starting from one.

Executing the oil database example

To execute the oil database example:

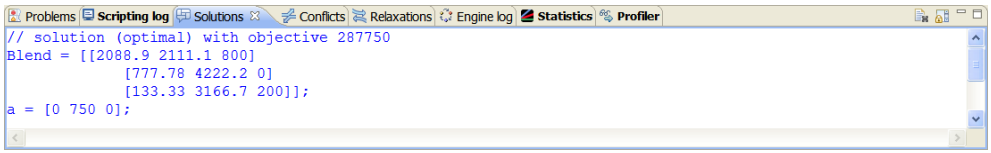
1. Right-click the run configuration **Data from database** and make it the default run configuration.
2. Right-click again and select **Run this**.

In the Problem Browser, you can examine the model in the usual manner to see the contents of the various data structures in this example. If you are not familiar with the Problem Browser, see *Understanding the Problem Browser* in *Getting Started with the IDE*.



Problem Browser after execution (`oilDB.mod`)

At the end of execution you see the following message in the **Solutions** tab).



Solutions tab (`oilDB.mod`)

You can examine the data and variables in tabular form from the data structure tree built in the Problem Browser. You can also see the table of all the results together in Microsoft Access.

Viewing the result in the database

When you have successfully executed the run configuration in the IDE, you can view the contents of the newly created table `Result` in the database.

1. Close the IDE and Microsoft® Access (if it is still open).
2. Restart Microsoft Access by double-clicking the database file `oilDB.mdb` in Windows Explorer.
3. Double-click `Result` in order to see the table's contents.

The Result table after execution

The table `Result` gives the results in rows for each crude/gasoline combination. It shows the blend as a number of barrels of each crude used for each type of gasoline and the amount spent on advertising that type of gasoline.

Result : Table

	oil	gas	blend	a
	Crude1	Super	2088.889	0
	Crude1	Regular	2111.111	750
	Crude1	Diesel	800	0
	Crude2	Super	777.7778	0
	Crude2	Regular	4222.222	750
	Crude2	Diesel	0	0
	Crude3	Super	133.3333	0
	Crude3	Regular	3166.667	750
	Crude3	Diesel	200	0
▶				

The modified Result table

The optimal blending has changed after the prices were modified but the optimal advertising expenditure has remained the same.

The oil sheet example

Describes the oil data spreadsheet, the oil sheet project, and its execution.

In this section

Description of the example
Includes what you will learn and where to find the files.

The oil data spreadsheet
Describes the data stored in Microsoft Excel spreadsheets.

The oil sheet project
Describes the model file in the project.

The oil sheet data file
Describes the data file in the project.

Executing the oil sheet example
Describes the procedure to execute the example and examines the results.

Viewing the result in the spreadsheet
How to display the `RESULT` table after execution using Microsoft Excel.

The RESULT sheet after execution
Shows the modified `RESULT` table after execution.

Description of the example

Make sure you have read the section *Using data sources* before you start.

The oil sheet example is a linear programming (LP) problem delivered within the `oil` project as a specific run configuration associating the `oil.mod` and `oilSheet.dat` files, with the spreadsheet `oilSheet.xls`. In this example, the data is stored in sheets of an Excel spreadsheet. The optimization problem it reflects is described in *A blending problem* in the *Language User's Manual*.

What you are going to do

With this example, you will learn how to:

- establish a connection to a Microsoft Excel spreadsheet from the IDE,
- read data from the spreadsheet into OPL arrays,
- write data to the spreadsheet from the IDE.

Where to find the files

For this example, you will need to use the following files:

- `oil` project: the oil blending example, in which one run configuration uses a connection to an Excel spreadsheet where there is data for a linear programming (LP) problem:
- `oilSheet.xls`: the Excel spreadsheet that contains the data for this run configuration

Both files are at the following location:

`<OPL_dir>\examples\opl\oil`

where `<OPL_dir>` is your installation directory.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

The oil data spreadsheet

The data for the oil sheet example is stored in the following sheets:

```
gas
oil
RESULT
```

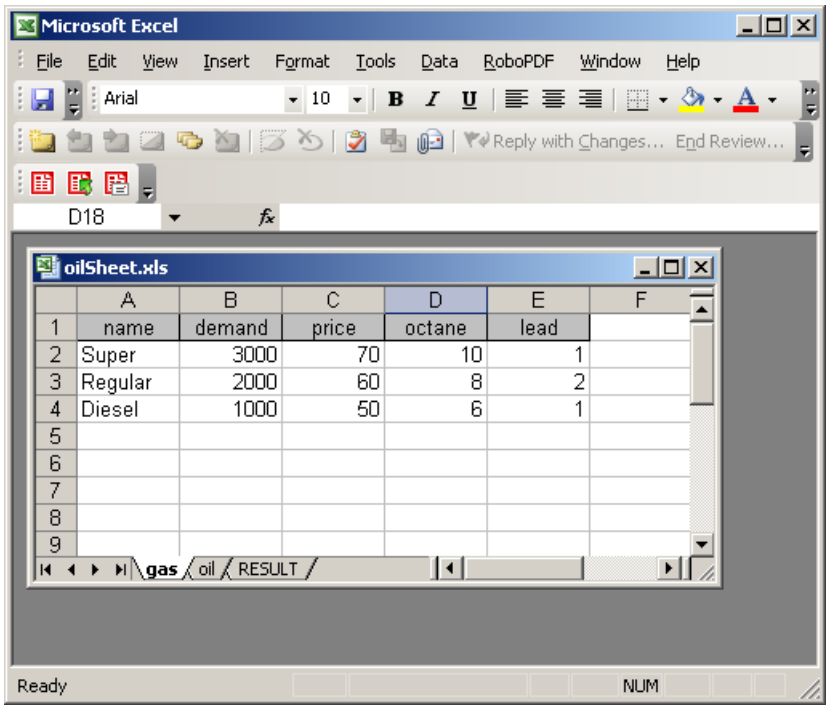
Provided that you have Microsoft® Excel installed on your computer, you can view the contents of any of these sheets by simply double-clicking the Excel file in Windows Explorer.

You should view this file in the **copy** of the project you created when you opened the example using the New Example wizard. If you did not change any of the default settings in the New Example wizard, that should be in your default workspace, at the following location:

```
C:\Documents and Settings\<user>\Application Data\ILOG\OPL Studio IDE\<version>\
oil\oilSheet.xls
```

Microsoft Excel opens the spreadsheet and you can see the first sheet and the tabs for all the sheets. Click a tab to see its contents. Notice that the `RESULT` sheet is initially empty.

The external data structure for the oil database example and the oil sheet example is the same.



The `oilSheet` spreadsheet

The gas data

Click the sheet `gas` if it is not visible when you open `oilSheet.xls`.

The five columns in `gas` store the following values for each type of gasoline manufactured:

```
name demand price octane lead
```

Each row is for one type of gasoline, with the product name of the gasoline stored as a character string in the first column. The daily customer demand for each type of gasoline is recorded and the price in this table is the sales price. The octane rating must be at least the value stored and the lead content must not exceed the value stored.

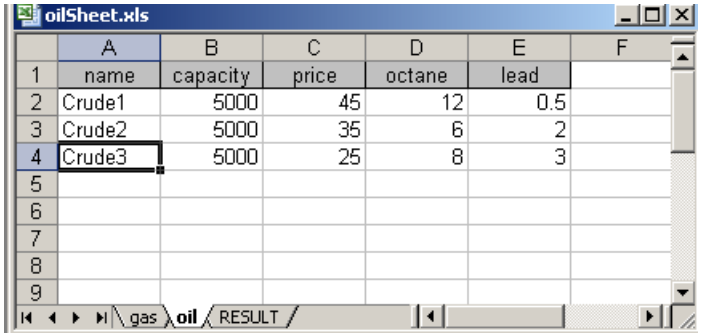
The oil data

The five columns in `oil` store the following values for each type of crude oil:

```
name capacity price octane lead
```

Each row is for one type of crude, with the name of the crude stored as a character string in the first column. The capacity figure represents the amount that can be purchased each day and the price is the purchase price. The other columns store the octane rating and the lead content.

The following graphic shows the `oil` spreadsheet as you see it in Microsoft Excel.



The `oil` spreadsheet

The oil sheet project

Use the **File>New>Example** menu command to open the **oil** example.

The IDE displays the `oil` project in the OPL Projects Navigator. Open the model file in the editing area.

Set definitions in the model

At the beginning of the model file, there are definitions of sets of string values to hold the names of the gasolines and oils:

Set definitions (`oil.mod`)

```
{string} Gasolines = ...;
{string} Oils = ...;
```

Tuple definitions in the model

These are followed by definitions of tuples for the data, as shown below.

Tuple definitions (`oil.mod`)

```
tuple gasType {
  float demand;
  float price;
  float octane;
  float lead;
}

tuple oilType {
  float capacity;
  float price;
  float octane;
  float lead;
}
```

Note that these tuple definitions follow the rows in the `gas` and `oil` data respectively but do not include the `name` column.

Tuple arrays in the model

The model in OPL then declares one-dimensional arrays, `gas` and `oil`, containing tuples for the gasolines and oils:

Tuple arrays (`oil.mod`)

```
gasType Gas[Gasolines] = ...;
oilType Oil[Oils] = ...;
```

Note: For data input from Microsoft Excel, OPL supports one-dimensional arrays of tuples as well as one or two-dimensional arrays of `int`, `float`, and `string` values.

The oil sheet data file

Double-click the `oilSheet.dat` file in the project tree to see the contents of the data file.

Data file for the Oil Sheet example (`oilSheet.dat`)

```
SheetConnection sheet("oilSheet.xls");
Gasolines from SheetRead(sheet,"gas!A2:A4");
Oils from SheetRead(sheet,"oil!A2:A4");
Gas from SheetRead(sheet,"gas!B2:E4");
Oil from SheetRead(sheet,"oil!B2:E4");
MaxProduction = 14000;
ProdCost = 4;


a to SheetWrite(sheet,"RESULT!A2:A4");
Blend to SheetWrite(sheet,"RESULT!B2:D4");
```

The data file starts with the `SheetConnection` statement used to connect to the spreadsheet.

Note: You can have multiple data files and, within any of them, multiple connections to spreadsheets.

Connecting to the spreadsheet from OPL

The connection is established by the following statement:

Connecting to a spreadsheet (MS Excel)

```
SheetConnection sheet("oilSheet.xls");
```

The name of the spreadsheet file in quotes is passed as an argument.

Note: You do not need to specify the full path name. Relative paths are resolved using the directory of the current .dat files.

Reading from the spreadsheet

Reading spreadsheet columns

You can read data from a column in any sheet into an OPL array using the `SheetRead` statement, as in the data file `oilSheet.dat`.

Reading spreadsheet columns (`oilSheet.dat`)

```
Gasolines from SheetRead(sheet,"gas!A2:A4");
Oils from SheetRead(sheet,"oil!A2:A4");
```

Note that the cells are read from 2 upward as the name of the column is not stored.

Reading spreadsheet cell ranges

You can read blocks of cells from a spreadsheet into an OPL array using the `SheetRead` statement, as in `oilSheet.dat`.

Reading spreadsheet cells (`oilSheet.dat`)

```
Gas from SheetRead(sheet,"gas!B2:E4");
Oil from SheetRead(sheet,"oil!B2:E4");
```

Note that the columns read are B to E (not A) and the cells read are from 2 upward as the `name` column is not used and the names of the other columns are not stored in the data arrays.

Note also that the data file initializes some variables directly; it does not only take data from the spreadsheet.

Writing the results to the spreadsheet

At the end of the optimization process, you need to store the optimal blends and advertising expenditures in the `RESULTS` sheet.

You can insert the `oil`, `gas`, `blend`, and `a` arrays as columns in the `RESULT` sheet using a `SheetWrite` statement.

Writing results to a spreadsheet (`oilSheet.dat`)

```
a to SheetWrite(sheet,"RESULT!A2:A4");
Blend to SheetWrite(sheet,"RESULT!B2:D4");
```

Executing the oil sheet example

To execute the oil sheet example:

1. First make sure the spreadsheet file is not read-only, then close it, so that the IDE can write to it.

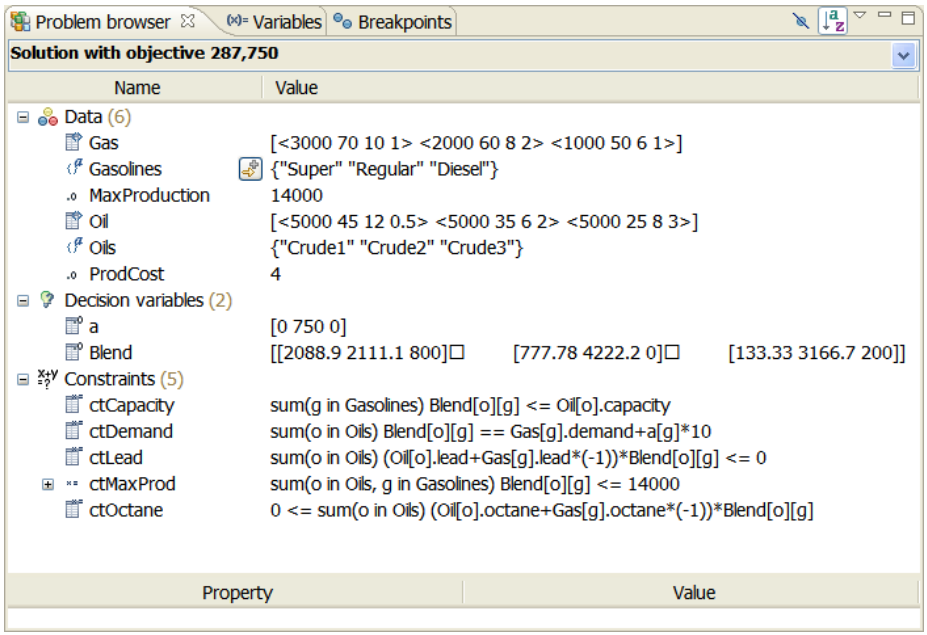
   If the spreadsheet file is read-only, the IDE displays an error message.

2. Exit Microsoft Excel.

   Otherwise, the spreadsheet file is considered read-only and the IDE reports an error message.

3. Right-click the run configuration **Data from spreadsheet** and make it the default run configuration.

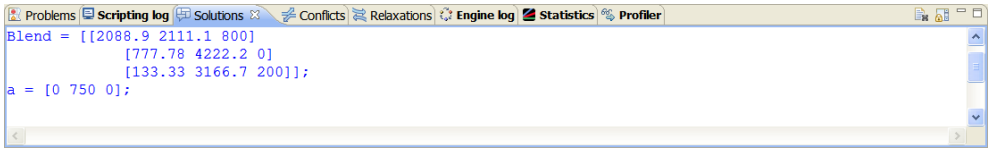
4. Right-click again and select **Run this**.



Problem Browser after execution (`oilSheet.dat`)

In the Problem Browser, you can examine the model in the usual manner to see the contents of the various data structures in this example. If you are not familiar with the Problem Browser, see *Understanding the Problem Browser* in *Getting Started with the IDE*.

At the end of execution you see the following message in the Solutions tab.



Solutions tab (`oilSheet.dat`)

You can examine the data and variables in tabular form from the data structure tree built in the Problem Browser. You can also see all the results together in Microsoft Excel.

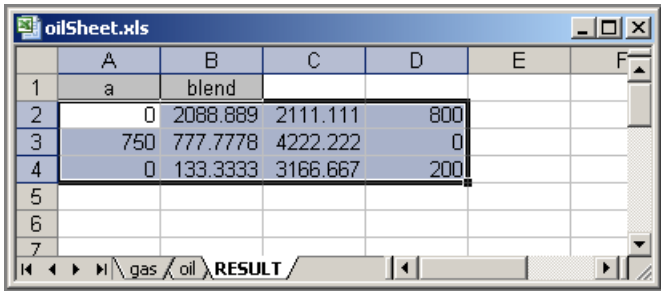
Viewing the result in the spreadsheet

When you have successfully executed the model in the IDE, you can view the contents of `RESULT` in the spreadsheet.

1. Reopen the spreadsheet file `oilSheet.xls` from Microsoft® Excel.
2. Click the `RESULT` tab to see the contents.

The RESULT sheet after execution

The RESULT sheet gives the results in rows for each crude/gasoline combination. It shows the blend as a number of barrels of each crude used for each type of gasoline and the amount spent on advertising that type of gasoline.



The RESULT sheet

Using IBM ILOG Script for OPL

Teaches the features of the IDE for scripts written in IBM ILOG Script for OPL, including the script debugging facilities.

In this section

Purpose and prerequisites
Describes the types of models used and outlines the assumed knowledge.

Features of IBM ILOG Script for OPL
Briefly explains what you can do with IBM ILOG Script for OPL and introduces the `main` and `execute` statements.

The multiperiod production planning example
Presents the example and explains how to execute it and debug the flow control script.

The transportation example
Presents the example and explains how to execute it and debug the preprocessing script.

The covering example
Presents the example and explains how to execute it and debug the postprocessing script.

Purpose and prerequisites

IBM® ILOG® OPL enables you to use a *scripting* language called IBM ILOG Script for OPL. This is an implementation of the ECMA-262 standard (also known as JavaScript™) which supports the “nonmodeling” elements of OPL.

This tutorial assumes that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

Note: This section uses several models solved by the CPLEX engine, but the features described work in the same way with models solved with the CP Optimizer engine.

The first section, *Features of IBM ILOG Script for OPL*, is a short list of what IBM ILOG Script enables you to write in your OPL projects. Then, as you go through this tutorial, you will become familiar with the following examples, each illustrating one way of using IBM ILOG Script for OPL:

- *The multiperiod production planning example*: flow control
- *The transportation example*: preprocessing
- *The covering example*: postprocessing

More complex examples of scripting are available in

`<OPL_dir>\examples\opl`

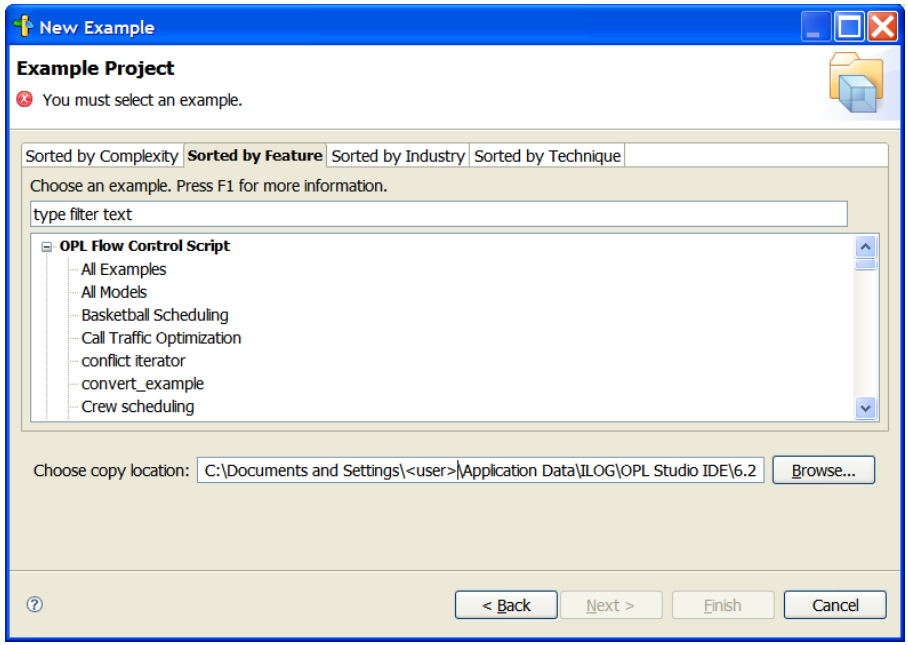
where `<OPL_dir>` is your installation directory.

They are used in *IBM ILOG Script for OPL* in the *Language User’s Manual*.

For more information, see also:

- *IBM ILOG Script for OPL* in the *Language Reference Manual*
- the *Reference Manual of IBM ILOG Script Extensions for OPL* for full details of the language.

Note: You can use the features of the New Example wizard to search for distributed OPL examples that use IBM ILOG Script or OPL Flow Control Script or other features or techniques you are interested in. For example, the following screenshot shows the New Example wizard displaying its **Sorted by Feature** tab, displaying some of the examples that show off **OPL Flow Control Script**.



Similarly, you can use the New Example wizard to search for examples by **Complexity**, **Industry** represented, or programming **Technique** used.

Features of IBM ILOG Script 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

Script statements

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
- The `execute` statement for preprocessing and postprocessing scripts

You can also write script statements in data files by using the `prepare` and `invoke` keywords.

Note: There are no separate script files; you write script statements directly in model files within “execute” or “main” blocks.

The multiperiod production planning example

Presents the example and explains how to execute it and debug the flow control script.

In this section

Presenting the multiperiod production planning example
Summarizes the problem and explains what to do and where to find the files.

Setting up the multiperiod production model and data
Presents the model and data file.

Executing a flow control script
Explains how to change the default run configuration.

Purpose of the flow control script
Explains how the script performs iterations to solve the model.

Debugging a flow control script
Describes breakpoint management, call stack display, variable object examination, and stepping.

Presenting the multiperiod production planning example

The multiperiod production planning example is a generalization of the single-period pasta production example described in *A production problem* in the *Language User's Manual*. The multiperiod version considers the demand for the products over several periods and allows the company to produce more than the demand in a given period. There is an inventory cost associated with storing the additional production.

The approach to solving this mathematical programming problem is described in detail in *A multi-period production planning problem* in the *Language User's Manual*. This section assumes that you are familiar with the solving strategy as explained in that document.

The file **mulprod_main.mod** contains the flow control script to implement the solving strategy. See the code sample *The flow control script (`mulprod_main.mod`)*.

A flow control script is encapsulated in a `main` statement:

```
main {
...
}
```

The `main` statement can be anywhere in the model file but usually follows the OPL statements for the model. See *Tutorial: Flow control and multiple searches* in the *Language User's Manual*.

What you are going to do

Working from the example described in *Presenting the multiperiod production planning example*, you will:

- open the **mulprod** example and discover its flow control script: see *Setting up the multiperiod production model and data*
- run the model: see *Executing a flow control script*
- debug the script, as explained in *Debugging a flow control script*, that is:
  - add a breakpoint
  - examine the call stack,
  - step to the next instruction,
  - abort execution or continue execution to the end of the script.

Where to find the files

To do so, you will work mainly with the run configuration **Solve models sequence by changing data** which associates the files **mulprod_main.mod** and **mulprod.dat** of the `mulprod` project, available at the following location:

`<OPL_dir>\examples\opl\mulprod`

where <OPL_dir> is your installation directory.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

Setting up the multiperiod production model and data

To start working with this example:

1. Use the **File>New>Example** menu command to open the **mulprod** example.

   The IDE displays the `mulprod` project in the OPL Projects Navigator.

2. Open the `mulprod_main.mod` model file in the editing area.

   Scroll to the `main` statement, presented here:

   The flow control script (`mulprod_main.mod`)

```
main {
  var status = 0;
  thisOplModel.generate();

  var produce = thisOplModel;
  var capFlour = produce.Capacity["flour"];

  var best;
  var curr = Infinity;
  var basis = new IloOplCplexBasis();
  var ofile = new IloOplOutputFile("mulprod_main.txt");
  while ( 1 ) {
    best = curr;
```

   Note that a semi-colon (**;**) at the end of a line is not mandatory; it is used for consistency with the OPL modeling language. IBM ILOG Script for OPL does not require a semi-colon at the end of a line; the OPL modeling language does.

3. Double-click the **mulprod.dat** file to see the data.

   The production data (`mulprod.dat`)

```
Products =  { kluski capellini fettucine };
Resources = { flour eggs };
NbPeriods = 3;

Consumption = [
                [ 0.5, 0.4, 0.3 ],
                [ 0.2, 0.4, 0.6 ]
              ];
Capacity = [ 20, 40 ];
Demand = [
           [ 10 100 50 ]
           [ 20 200 100]
           [ 50 100 100]
         ];
Inventory = [ 0 0 0];
InvCost = [ 0.1 0.2 0.1];
InsideCost = [ 0.4, 0.6, 0.1 ];
OutsideCost  = [ 0.8, 0.9, 0.4 ];
```

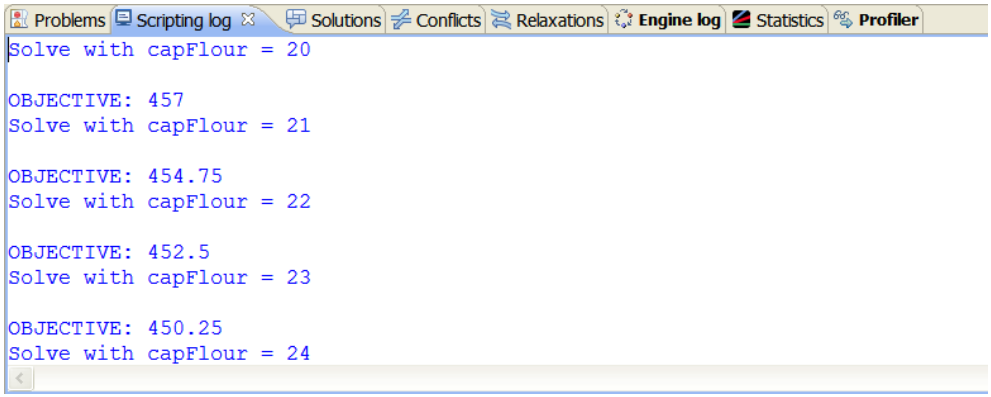
Executing a flow control script

The first run configuration, **Solve problem once**, appears as the default. Since this not the run configuration that contains the flow control script, you will first change the default run configuration.

To execute the appropriate run configuration:

1. Right-click **Solve models sequence by changing data** and select **Set as default**.
2. Right-click again and select **Run this**. (You could also right-click the project name and select **Run>Default Run Configuration**.)

The iterations appear one at a time in the **Scripting log** tab.

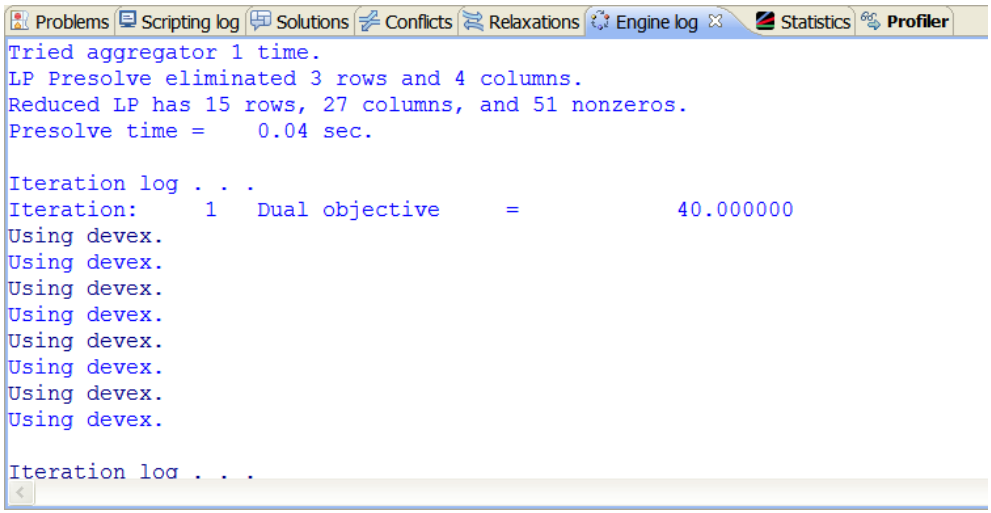


Upon completion, the IDE displays the number of solve iterations in the **Engine Log** tab. See *What happens when you execute a run configuration* in *IDE Reference* for details of the execution process.

Purpose of the flow control script

The script performs iterations to solve the model. On each iteration, it creates a new instance of the model with a changed value for the variable `capFlour` to improve the value of the objective function. The loop ends when there is no further improvement in the objective value.

After execution, the **Engine Log** ends as shown below.



Engine Log (`mulprod_main.mod`)

The last two objective values in the set of solutions are the same and so solving stops.

Debugging a flow control script

You are now going to learn how to debug a script. The debug features in the IDE are breakpoint management, call stack display, variable object examination, and stepping.

A simple debugging scenario is to place a breakpoint in a script, execute the script by means of the **Debug** button  , examine the call stack, and then interactively execute statements using the **Step Over** button.

This section covers the successive debugging steps:

- *Adding a breakpoint to a flow control script*
- *Examining the flow control call stack*
- *Stepping to the next instruction*
- *Aborting execution*
- *Continuing without stepping*
- *Ending execution*

Adding a breakpoint to a flow control script

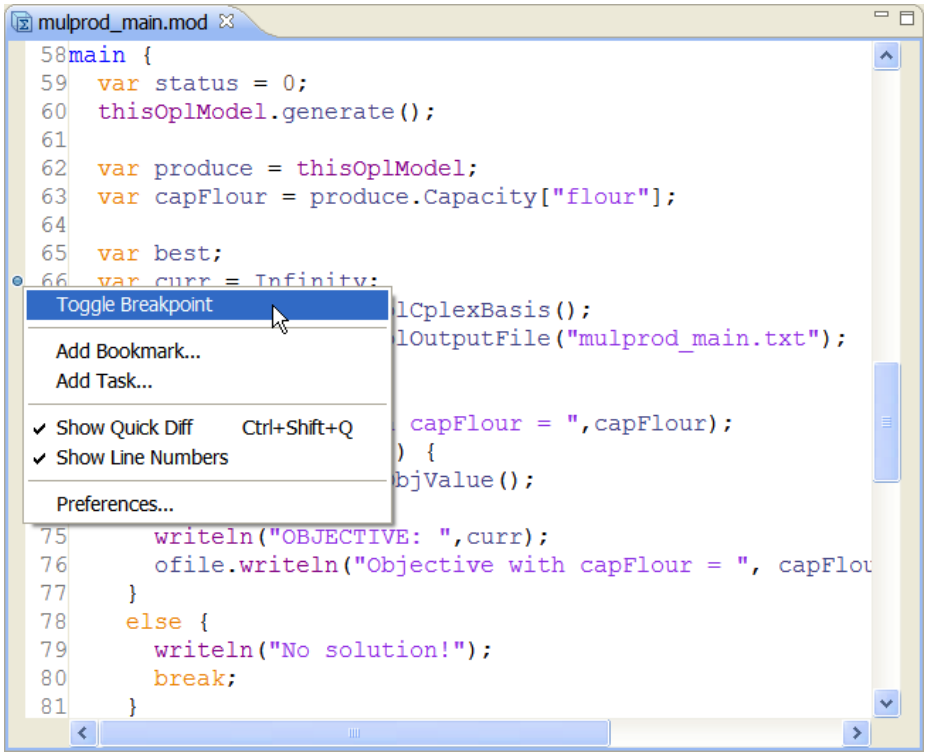
Still using the **Solve models sequence by changing data** run configuration of the `mulprod` model, you are going to add a breakpoint to set the debugging mode, then execute the script again, using this time the **Debug** button.

1. If it is not already open, double-click the **mulprod_main.mod** file in the editing area and scroll down to this line just before the loop:

   ```
   var curr = Infinity;
   ```

2. Right-click in the grey margin to the left of this line. From the popup menu, select **Toggle Breakpoint** and a blue dot appears in the margin of the editing area to signal the breakpoint.

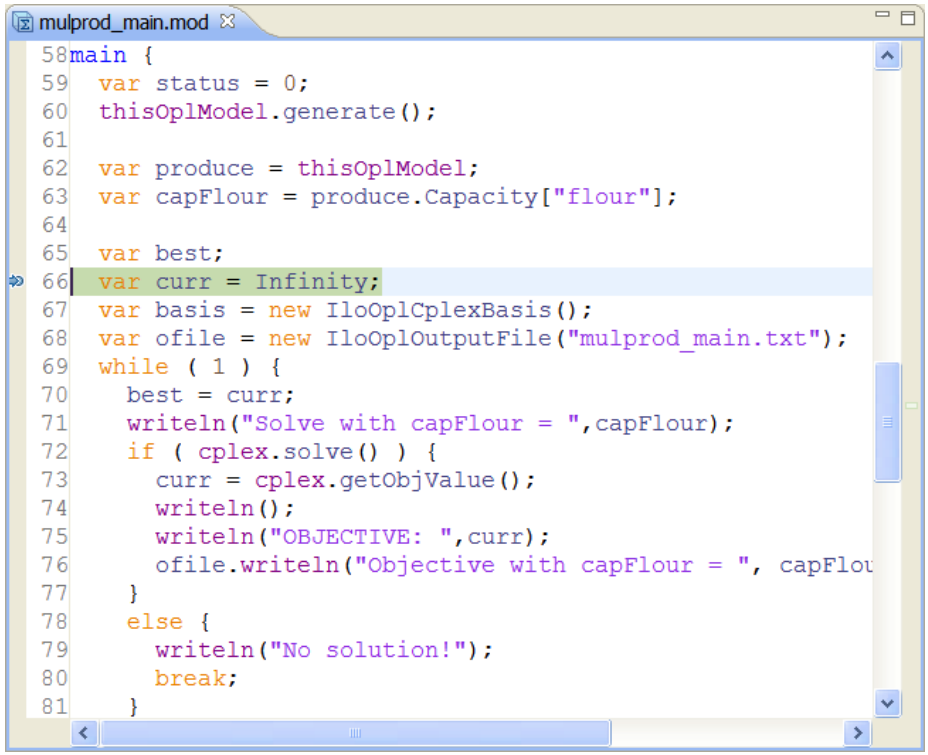
> **Note**: To remove a breakpoint, right-click and select again **Toggle Breakpoint** to make the blue dot disappear.
>
> You can also double-click in the grey margin to set and remove break points.



Inserting a breakpoint

3. In the main toolbar, click the arrow on the **Debug** button in the toolbar and select **1 mulprod Solve models sequence by changing data** to run the script.

   Execution stops at the breakpoint and a blue arrow in the margin shows the current position, as shown below.



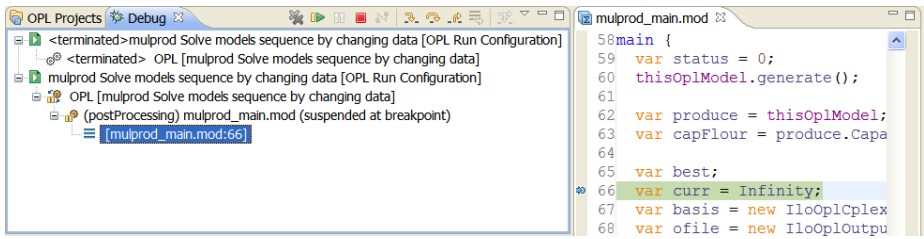
A breakpoint before the loop of the script (`mulprod_main.mod`)

Examining the flow control call stack

The call stack is now spread over two views: the Debug view, representing the call stack (showing nested function calls) and the Variable view, showing the content of the selected call frame.

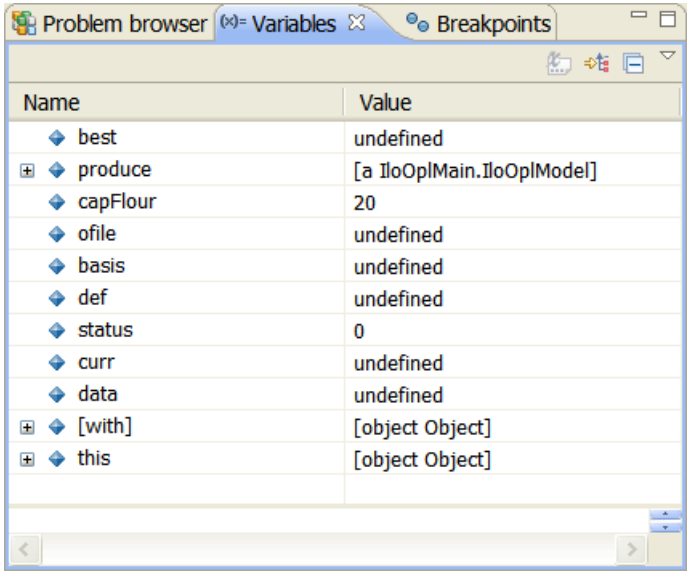
1. The call stack appears in the **Debug** view. Each function called has information in a stack frame. In this example, there is one frame, `[mulprod_main.mod:58]`.

   Click the **+** and **-** sign as necessary to expand or collapse nodes.



The call stack in the Debug view (`mulprod_main.mod`)

2. The **Variables** view shows the content of the selected call frame. In this example, there is only one call frame.



Many of the variables in this example are marked **undefined** because they are not decision variables.

Stepping to the next instruction

When you execute a run configuration in debugging mode, the **Debug** view is displayed to enable you to use the stepping buttons in its toolbar.

Debug toolbar

The **Step Over** button  allows you to step through the script instruction by instruction, executing each instruction as you go. Use it to step over:

- a function and go to the statement after the function call
- an instruction and go to the instruction after it

When you are stepping in a script, the blue arrow in the margin keeps track of the current position.

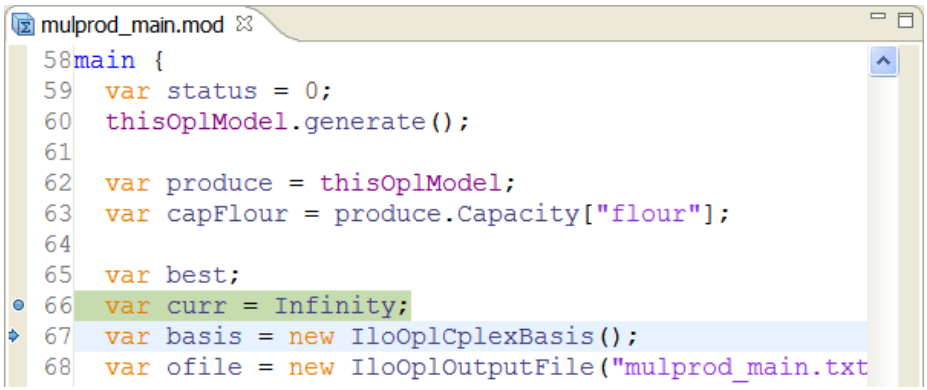
To step to the next instruction:

1. Click the **Step Over** button  .

The IDE executes the current instruction and the blue arrow in the margin moves to the next line, which is the statement for warm start:

```
var basis = new IloOplCplexBasis();
```

as shown below.



Step by step execution of a script

2. By repeatedly clicking the **Step Over** button, you can follow the execution of the loop one instruction after another.

Note that the **Step Into** button  would give the same behavior as **Step Over** in this script because there are no functions, so it just executes the current instruction.

Continuing without stepping

- At any moment while you are stepping in the script, you can ask the IDE to continue executing until completion by clicking the **Resume** button  in the toolbar of the **Debug** view.

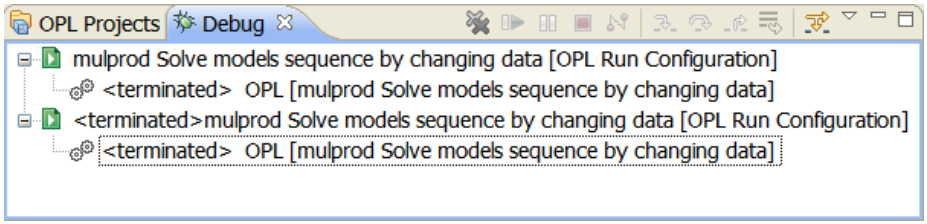
  The IDE executes the rest of the script without stopping at instructions. The possible outputs of the script are printed in the **Scripting log**.

Aborting execution

At any moment while you are stepping in the script, you can abort execution.

1. Click the **Abort the current solve** button  in the execution toolbar of the main window. (You can also click the **Terminate** button in the toolbar of the **Debug** view.)

   The **Debug** view shows that the status of the execution is 'terminated'.



Aborting the execution of a script in debugging mode

2. After aborting, you can relaunch the script by clicking the **Debug** button.

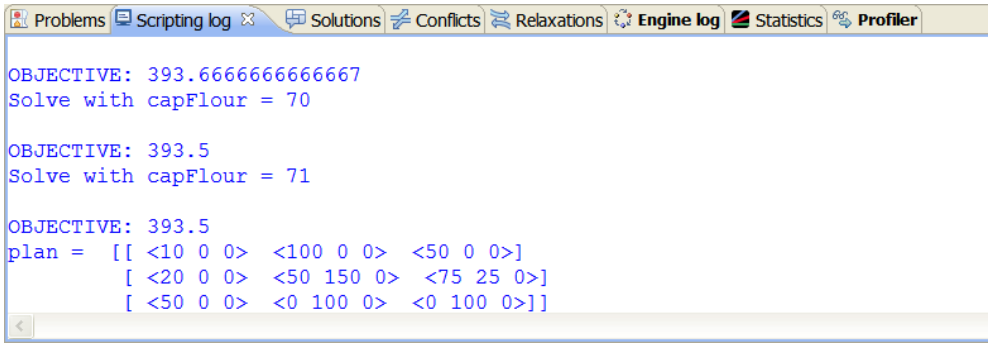
Ending execution

To summarize, while stepping through a script, you can end execution in one of three ways:

1. Click the **Continue the current solve** button  : the IDE completes the script.
2. Click the **Abort the current solve** button  : the script terminates without being completed.
3. Click the **Step Over** button  (or **Step Into** button  , if there are no functions) repeatedly; the IDE completes the script, instruction by instruction.

Examining the output

The output of execution is displayed in the **Scripting log**.



The transportation example

Presents the example and explains how to execute it and debug the preprocessing script.

In this section

Presenting the transportation example
Summarizes the problem and explains what to do and where to find the files.

Setting up the transportation model and data
Describes the model and data files.

Executing preprocessing scripts
Run a different run configuration that utilizes preprocessing scripts.

Purpose of the preprocessing scripts
Explains how to initialize an array, set a CPLEX parameter, set an OPL option, and set the display of variables.

Debugging a preprocessing script
Describes the successive debugging steps.

Presenting the transportation example

This is a multicommodity flow problem with cities that supply products and cities that demand products. In addition, there is a capacity constraint on the connections between the cities. One issue in large-scale transportation problems like this is that only a fraction of the cities are interconnected. Because of this sparsity issue, a good representation for this application consists of explicit sets of connections, routes, and costs of routes, as well as the demand and supply information.

The approach to solving this example is described in detail in *Exploiting sparsity* in the *Language User's Manual*. This section assumes that you are familiar with this example and the solving strategy as explained in that document.

The file **transp4.mod** contains preprocessing scripts to prepare data and options and to display data. See *Preprocessing statements* `(transp4.mod)`.

Preprocessing scripts are `execute` blocks declared before constraints, like this:

```
execute {
...
}
```

Important: Any `execute` statement for preprocessing must precede the objective function in the model file.

What you are going to do

Working from the example described in *Presenting the transportation example*, you will:

- set up the model and data: see *Setting up the transportation model and data*
- run the model without a breakpoint and examine specific preprocessing scripts: see *Executing preprocessing scripts*
- add a breakpoint and start debugging, as explained in *Debugging a preprocessing script*, that is:
  - examine the call stack,
  - monitor a loop by using a breakpoint to stop at each iteration,
  - step out of the `execute` function,
  - step into a function,
  - step out of the function,
  - monitor a function in a loop.

The script variant of the example is supplied as file `transp4.mod`, used in the run configuration named **Even better sparsity**, which associates the model file **transp4.mod** and the data file **transp4.dat** of the `transp` project, available at the following location:

```
<OPL_dir>\examples\opl\transp
```

where `<OPL_dir>` is your installation directory.

Setting up the transportation model and data

To start working with this example:

1. Use the **File>New>Example** menu command to open the **transp** example.

   The IDE displays the `transp` project in the OPL Projects Navigator. Open the model file in the editing area.

2. Double-click **transp4.mod** to display this model in the editor.

3. Scroll to the `execute` blocks.

   There are three preprocessing `execute` blocks before the objective, as shown below.

   Preprocessing statements (`transp4.mod`)

```
execute PARAMS {
  cplex.tilim = 100;
}

execute SETTINGS {
  settings.displayComponentName = true;
  settings.displayWidth = 40;
  writeln("Routes: ",Routes);
}

execute DISPLAY {
  function printRoute(r) {
    write("  ",r.p,":");
    writeln(r.e.o,"->",r.e.d);
  }

  writeln("Routes:");
  for (var r in Routes) {
    printRoute(r);
  }
}
```

The data is initialized in the `transp4.dat` data file. You can double-click **transp4.dat** to open that file in the editing area if you want to follow along with what is being described below.

Note that the tuple set `TableRoutes` is database-friendly in that it would allow the loading of data on routes and costs with a single SELECT statement.

In this example, the tuples in set `TableRoutes` and in arrays `Supply` and `Demand` are explicitly initialized in the data file because the matrix is sparse and only some tuples exist.

The model requires routes and costs separately, so in the model file, the tuples in tuple set `Routes` are derived from those in `TableRoutes` and the tuples in sets `Supplies` and `Customers` are then derived from those in `Routes`.

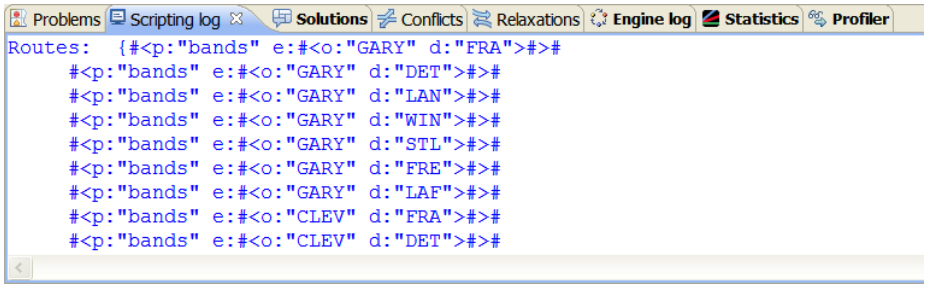
Executing preprocessing scripts

Basic Configuration is the default run configuration in this project, but it does not contain the preprocessing scripts you will be working with. So in the steps below you run the **Even better sparsity** run configuration.

- Right-click the run configuration **Even better sparsity** and choose **Run this**.

> **Note**: You could also right-click **Even better sparsity** and choose **Set as default**, and then run it by right-clicking the project and choosing **Run>Default Run Configuration**.

Upon completion, the IDE displays the results in the **Scripting log**.



See *What happens when you execute a run configuration* in *IDE Reference* for details of the execution process.

Purpose of the preprocessing scripts

In this model, preprocessing scripts are used for:

- *Initializing an array*
- *Setting a CPLEX parameter*
- *Setting an OPL option*
- *Setting the display of variables*

Initializing an array

The recommended syntax to initialize arrays is via generic indexed arrays, as shown in the code extract below, which sets up a cost array for the routes.

Generic indexed arrays, an example (`transp4.mod`)

```
float Cost[Routes] = [ <t.p,<t.o,t.d>>:t.cost | t in TableRoutes ];
```

This cost array is used in the objective, which aims to minimize the sum of transportation costs along all routes. See also *As generic indexed arrays* in the *Language Reference Manual*.

However, you can also use a preprocessing `execute` block as shown in the following code extract, which contains a script named `INITIALIZE`.

Preprocessing script: initializing an array (`transp4.mod`)

```
float Cost[Routes];
execute INITIALIZE {
  for( var t in TableRoutes ) {
      Cost[Routes.get(t.p,Connections.get(t.o,t.d))] = t.cost;
   }
}
```

Setting a CPLEX parameter

The script named `PARAMS` sets a CPLEX® parameter for the algorithm.

Preprocessing script: setting a CPLEX parameter (`transp4.mod`)
This parameter sets a time limit on each call to the optimizer. See *Changing CPLEX parameters* in the *Language User's Manual*.

Setting an OPL option

The script statement named `SETTINGS` sets the display of the component name on 40 characters.

Setting the display of variables

The script statement named DISPLAY displays the routes in the Routes tuple set in the format:

```
product: origin->destination
```

The DISPLAY script uses a function to do this. You can see this display of the routes in the **Scripting Log** tab.

Preprocessing script: displaying variables (transp4.mod)

```
execute DISPLAY {
  function printRoute(r) {
    write("  ",r.p,":");
    writeln(r.e.o,"->",r.e.d);
  }

  writeln("Routes:");
  for (var r in Routes) {
    printRoute(r);
  }
}
```

Debugging a preprocessing script

This section covers the successive debugging steps:

- *Adding a breakpoint to a preprocessing script*
- *Examining the preprocessing call stack*
- *Stepping out of an execute function*
- *Stepping into a function*
- *Stepping out of a lower-level function*
- *Monitoring a function in a loop*

Adding a breakpoint to a preprocessing script

You are going to add a breakpoint to set the debugging mode, then execute the script again, using this time the **Debug** button.

To add a breakpoint:

1. In the **transp4.mod** file, scroll to the line containing the `printRoute` function in the `DISPLAY execute` block.

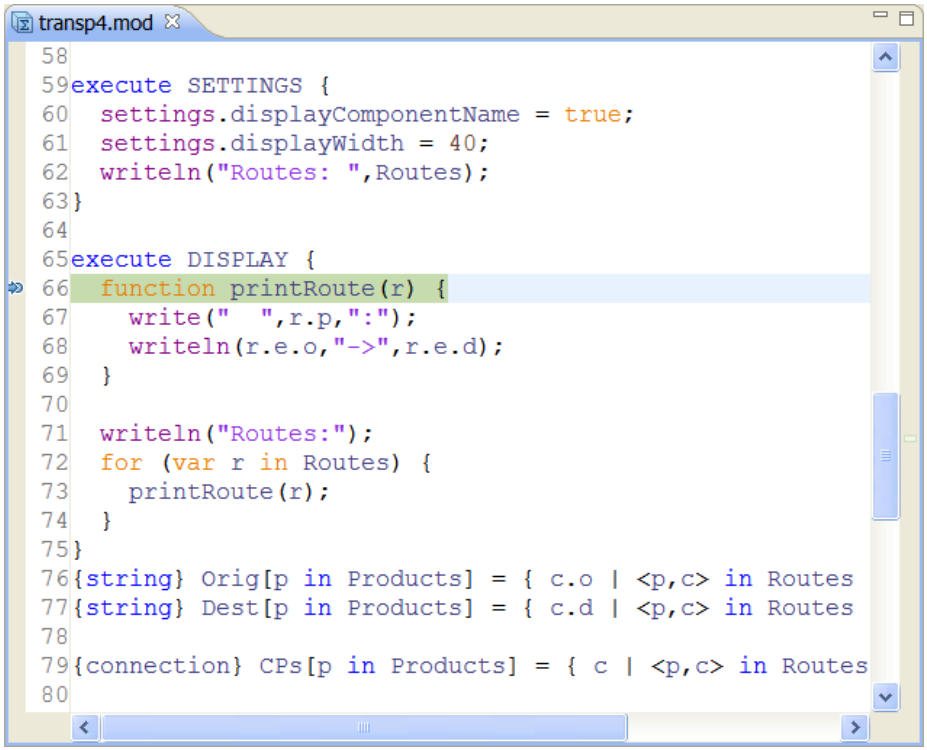
   ```
   function printRoute(r)
   ```

2. Double-click in the grey margin to the left of the line.

   A blue dot appears in the margin of the editing area to signal the breakpoint.

3. In the main toolbar, click the arrow to the right of the **Debug** button  and select **1 transp Even better sparsity** to run the script.

   Execution stops at the breakpoint and information appears in the Variables view and the **Scripting log**.

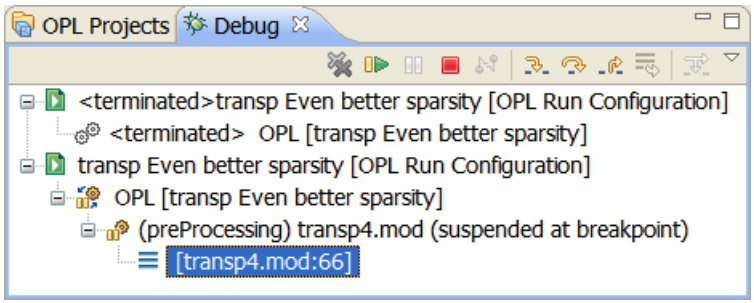


Execution stopped at the breakpoint in `transp4.mod`

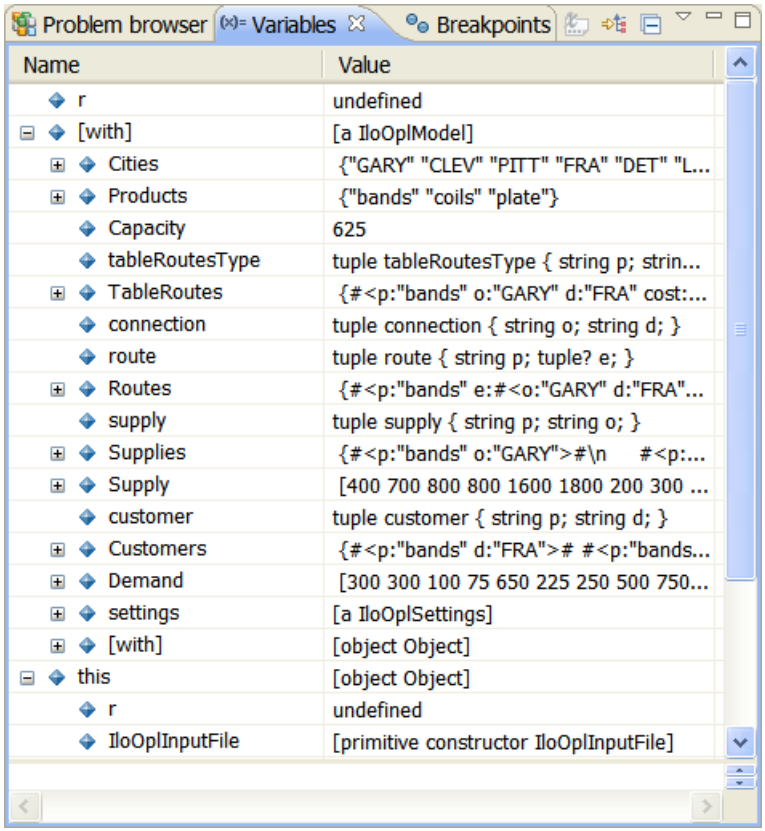
Examining the preprocessing call stack

The call stack, showing nested function calls, is displayed in the **Debug** view. The content of a selected call frame is displayed in the **Variables** view.

In this example, the call stack contains one call frame, `[transp4.mod:66]`.



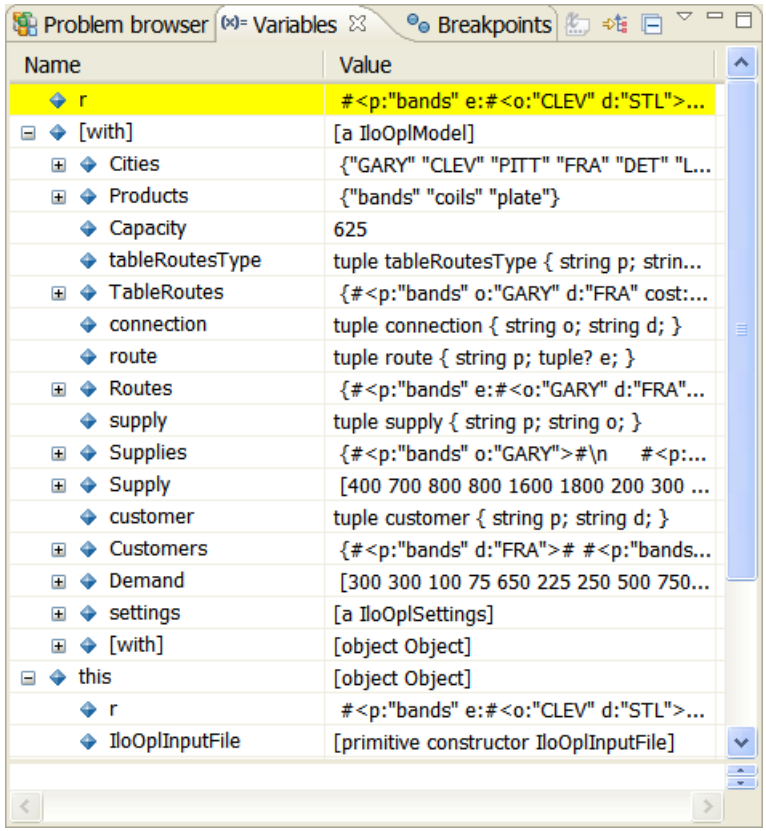
Debug view for `transp4.mod`



Variables view for call frame `[transp4.mod:66] (transp4.mod)`

For a large tuple set, the values may not all be visible within the window. In this case, an ellipsis appears at the end of the cell. Pass the cursor over the column to display all the values in a tooltip.

- Click the **Step Over**  or **Step Into** button  repeatedly to watch the value of the variables change in the call stack. The variables are highlighted when their values change.



The Variables view after stepping

Stepping out of an execute function

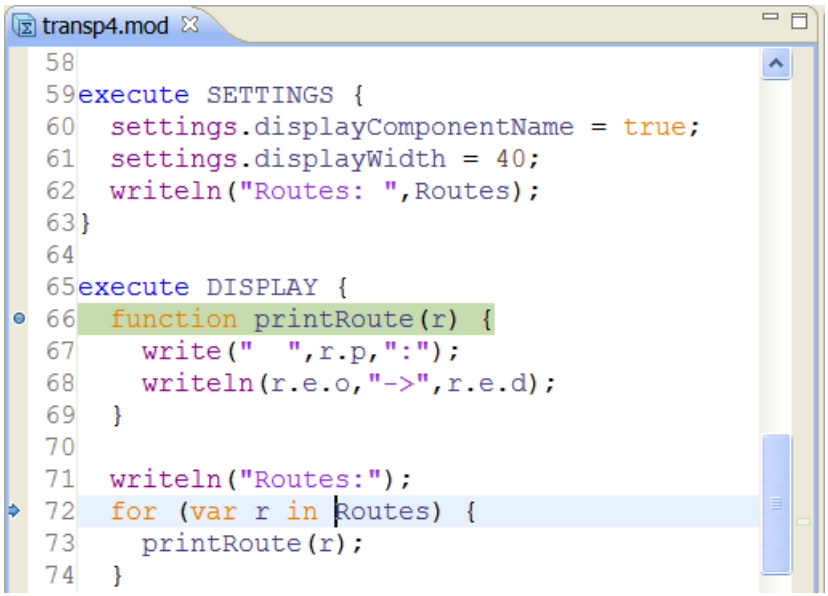
If you are stepping in an `execute` statement and the current instruction is within a loop (`for`, `forall`, `while`, or `repeat`), you can make the IDE execute the loop without stopping at instructions by stepping out of the `execute` function using the **Step Return**  button. The IDE executes the entire loop, then stops at the first instruction after the loop.

The **Step Return** button allows you to step out of a function in a script, to the statement following the function call. For example, if you were to click the **Step Return** button at this statement:

```
writeln("Routes":);
```

the current instruction becomes:

```
for (var r in Routes) {
```



Stepping out of an `execute` function `(transp4.mod)`

1. Click the **Step Over** button  to execute the instruction.

   If you open the nodes in the call stack, you can see the CPLEX® parameter setting listed for the predefined `cplex` object.

2. Click **Step Over** four more times.

   The current instruction is now:

   ```
   printRoute(r);
   ```

Note that **Step Over** does not move to the statement within the function because it is at a lower level.

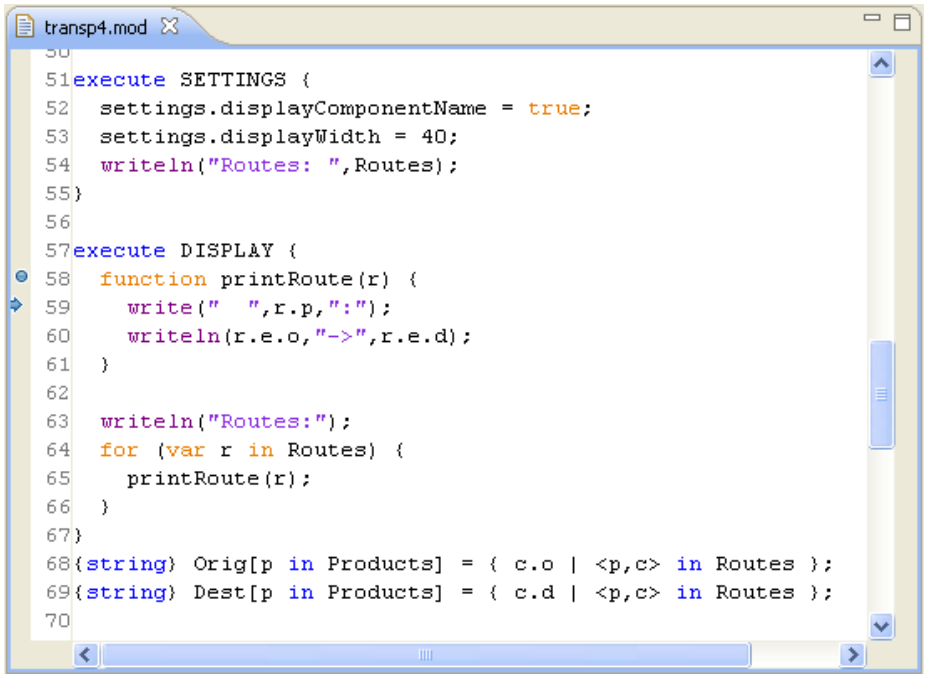
Stepping into a function

The **Step Into** button  allows you to step into a function. Use it to step to:

- the first statement, or
- to the instruction after the current one if there is no function called.
  - To step into the `printRoute(r)` function, click the **Step Into** button.

The IDE executes the current instruction and the blue arrow in the margin moves to the first line in the function:

```
write("  ",r.p,":");
```



Stepping into a function *(transp4.mod)*

Stepping out of a lower-level function

- Click the **Step Return** button  to execute the `write` and `writeln` statements, and step out of the `printRoute(r)` function.

  The current instruction becomes:

  ```
  for (var r in Routes) {
  ```

  because the loop iteration is completed and so the next instruction is the `loop` statement.

```
56
57execute DISPLAY {
58  function printRoute(r) {
59    write("  ",r.p,":");
60    writeln(r.e.o,"->",r.e.d);
61  }
62
63  writeln("Routes:");
64  for (var r in Routes) {
65    printRoute(r);
66  }
67}
```

Stepping out of the PrintRoute(r) function `(transp4.mod)`

Monitoring a function in a loop

- To continue execution, stopping at each pass through the `printRoute(r)` function, click the **Step Into** button  repeatedly.

The routes appear one at a time in the **Scripting log**.

Note that the **Step Into** button can continue iterating through the loop as well as stepping into the function. When the loop is completed, the statement is also complete and so the model is solved.

At any point you can continue execution to the end of the script without stopping by clicking the **Resume** button  in the Debug view.

The covering example

Presents the example and explains how to execute it and debug the postprocessing script.

In this section

Presenting the covering example
Summarizes the problem and explains what to do and where to find the files.

Setting up the covering model and data
Describes the model and data files.

Purpose of the postprocessing script
Enables you to postprocess solutions by placing an `execute` statement after the objective.

Executing a postprocessing script
Explains how to execute a run configuration and examine the result in the IDE.

Postprocessing a feasible solution
Discusses changing a language option and adding a settings file before postprocessing.

Presenting the covering example

The `covering` example is described in detail in *Set covering* in the *Language User's Manual*. This section assumes that you are familiar with this integer programming problem and the solving strategy as explained in that document. Here is a summary of the problem.

A set covering problem involves selecting items to fill (cover) a need. In this case, the need is to build a house and the items are workers. The construction of a house can be divided into a number of tasks, each requiring one or more skills, such as plumbing or masonry. A worker may or may not be able to perform a task, depending on his or her skills, and the cost of hiring a worker also depends on his or her skills (qualifications). The problem consists of selecting a set of workers to perform all tasks, while minimizing the cost. A 0/1 variable is associated with each worker to represent whether or not the worker is hired.

The file **covering.mod** contains the postprocessing script which manipulates the results data to show the resulting crew of hired workers. See *Postprocessing script, `covering.mod`*.

A postprocessing script is encapsulated in an `execute` statement:

```
execute {
...
}
```

The `execute` statement for postprocessing must **follow** the objective function in the model file.

What you are going to do

Working from the `covering` example described above, you will:

- set up the model and data: see *Setting up the covering model and data*
- run the default configuration, including the postprocessing script, as explained in *Executing a postprocessing script*, and then:
  - examine the Scripting Log tab
  - change an OPL Language option
  - rerun the model and see the difference in the Scripting Log tab

Where to find the files

The covering example is supplied in the `covering` project, at the following location:

`<OPL_dir>\examples\opl\covering`

where `<OPL_dir>` is your installation directory.

The model for the `covering` project is contained in the file **covering.mod**. Data for the model `covering.mod` is contained in the file **covering.dat**.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

Setting up the covering model and data

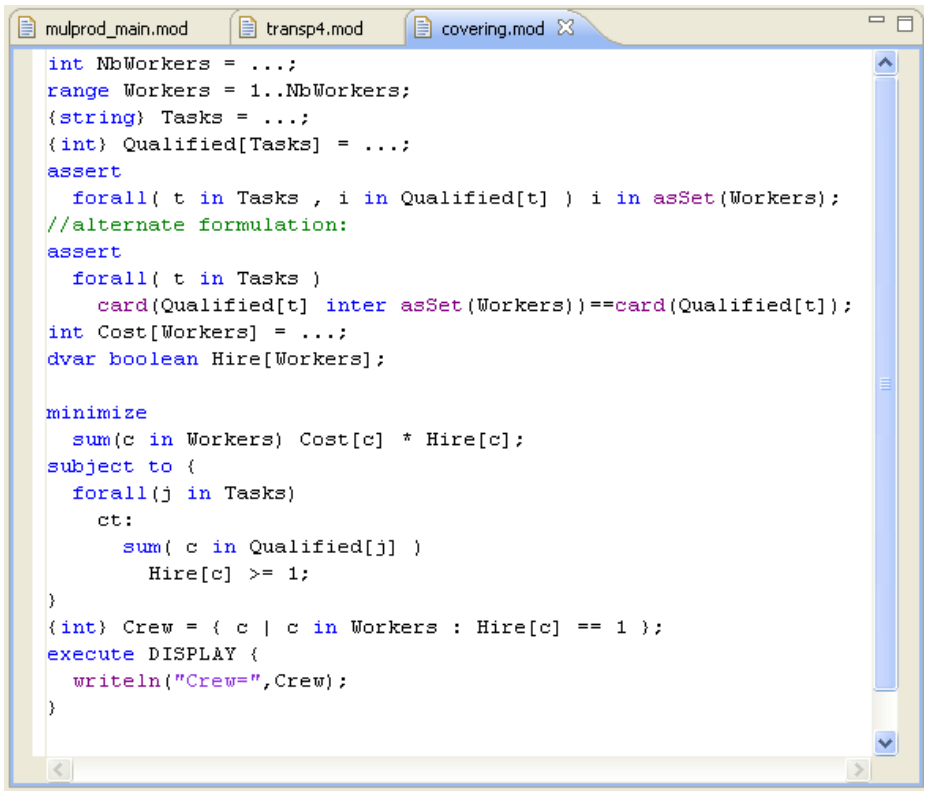
To start working with this example:

1. Use the **File>New>Example** menu command to open the **covering** example.

   The IDE displays the `covering` project in the OPL Projects Navigator.

2. Open the `covering.mod` model file in the editing area.

   Scroll to the `DISPLAY` postprocessing script, in the `execute` block just after the objective.



Postprocessing script, `covering.mod`

3. Open the `covering.dat` file.

   The data is initialized in the data file. Notice that 32 workers are available.

The covering data (`covering.dat`)

```
NbWorkers = 32;
Tasks = { masonry, carpentry, plumbing, ceiling,
     electricity, heating, insulation, roofing,
     painting, windows, facade, garden,
```

```
     garage, driveway, moving };
Qualified = [
   { 1  9 19  22  25  28  31 }
   { 2 12 15 19 21 23 27 29 30 31 32 }
   { 3 10 19 24 26 30 32 }
   { 4 21 25 28 32 }
   { 5 11 16 22 23 27 31 }
   { 6 20 24 26 30 32 }
   { 7 12 17 25 30 31 }
   {  8 17 20 22 23  }
   {  9 13 14  26 29 30 31 }
   {  10 21 25 31 32 }
   {  14 15 18 23 24 27 30 32 }
   {  18 19 22 24 26 29 31 }
   {  11 20 25 28 30 32 }
   {  16 19 23 31 }
   {  9 18 26 28 31 32 }
];
Cost = [ 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 6 6 6 7 8 9
 ];
```

Purpose of the postprocessing script

Any `execute` IBM® ILOG® Script statement placed after the objective enables you to postprocess the solutions found by the execution process. For example, in the statement named `DISPLAY` in `covering.mod`, it allows you to write to the **Scripting log** (see *Postprocessing script, `covering.mod`*).

Executing a postprocessing script

The `covering` example defines only one run configuration.

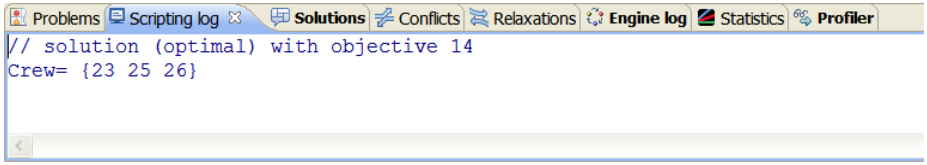
To execute the run configuration:

1. Right-click on **Run Configurations** and select **Run>Basic Configuration (default)**.

   Upon completion, the IDE displays the number of solutions found in the Solutions tab. See *What happens when you execute a run configuration* in *IDE Reference* for details of the execution process.

2. Examine the **Scripting log**.

   At the end of execution, the **Scripting log** contains the line written by the postprocessing script.



Scripting log output written by the engine and a script (`covering` project)

The line written to the Scripting Log by the postprocessing script shows the crew, that is, the workers that are hired. Three out of thirty-two workers are used in this solution.

Postprocessing a feasible solution

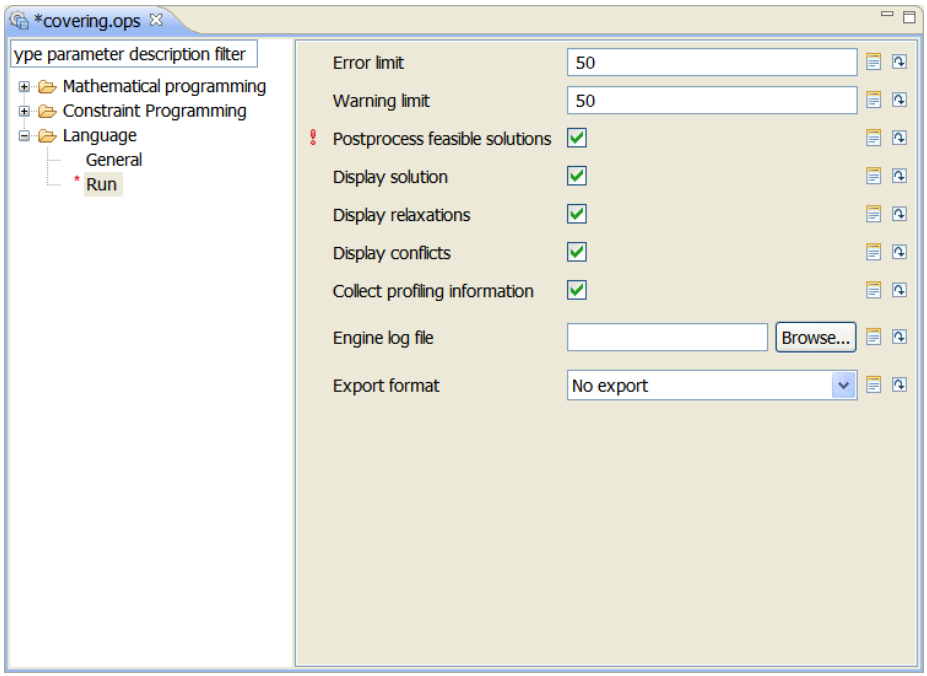
You are now going to change an **OPL Language** option for the `covering` project and observe the different output in the Scripting Log after execution. First, you need to add a settings file to that project if it does not already exist.

To change an OPL Language option for a project:

1. Right-click the **covering** project in OPL Projects Navigator and choose **New>Settings** to create a settings file as explained in *Adding a settings file* in *Getting Started with the IDE*.
2. Name it **covering.ops** and add it to the default run configuration using drag and drop.

   After you have done this the various MP, CP, and OPL Language options become visible in the settings editor.
3. Click **Language>Run,** then check the **Postprocess feasible solutions** box and save the settings file.

   Note the red exclamation mark indicating that the option is set to a user-defined value.

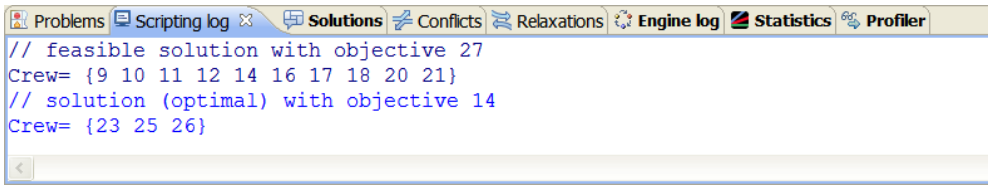


Turning on the Postprocess feasible solutions option

4. Re-run **Basic Configuration** in the **covering** project to execute the covering example again and postprocess any feasible solutions found before the final solution.

Result

At the end of execution, observe the difference in the **Scripting log**. It contains two sets of output written by the engine and by the postprocessing script, one for the feasible solution, and one for the final solution.



Postprocessing a feasible solution (`covering` project)

The non optimal feasible solution appears because the **Postprocess feasible solutions** was checked.

Relaxing infeasible models

Uses the nurses example to demonstrate how the IDE detects conflicts and searches for relaxations in models that appear infeasible after execution.

In this section

Presenting the nurse scheduling example
Summarizes the problem and explains what to do and where to find the files.

Setting up the nurses model and data
Describes the elements of the model.

Executing the nurses project (1)
Describes how to observe infeasibility, conflicts and proposed relaxations.

Working on the execution results
Describes how to study the conflict and suggested relaxation, and change the data to remove infeasibility.

Executing the nurses project (2)
Explains how to observe the new result after changing the data.

How relaxation and conflict search works
Relaxations and conflicts both express the infeasibility of a model and propose steps towards feasibility. After you have had hands-on experience with the nurse scheduling example, learn how to differentiate between Relaxations and Conflicts.

Presenting the nurse scheduling example

The tutorial assumes that you know how to work with projects in the IDE. If this is not the case, first read *Getting Started with the IDE*.

Note: You can also search for relaxations and conflicts using the IBM® ILOG® Script methods `printRelaxations` and `printConflicts`. See *Searching for relaxation and conflicts*.

The nurses example

The nurses example describes a nurse scheduling problem: the hospital Human Resources department needs to create work schedules for nurses and nurse teams. A good schedule is an optimal one, that is, a schedule that meets as many of the hospital's overall goals as possible. If some of these goals prove incompatible, then the solution consists in nevertheless finding a schedule that will work out, in other words, a *feasible* schedule. The goals include:

- staffing each hospital department with the proper number of nurses at all times;
- matching a nurse's skills, such as a Board Certification in Cardiac Care, with the requirements of the department;
- establishing a minimum and maximum number of hours worked per week;
- maximizing fairness in how nurses are allocated to shifts, that is, making sure that no nurses are scheduled for 50 hours a week when others are scheduled for only 25 hours;
- incorporating best practice guidelines, such as trying to schedule nurses with compatible skills or with a proven history of working well together on the same team;
- taking into account individual nurse preferences for days off as much as possible;
- keeping salary costs to a reasonable level.

Sometimes, several of these goals will conflict, for example, during a week when many nurses are on vacation. In such a case, the solving engine finds no solution. As this is not acceptable because a hospital needs a nurse schedule, you have to make choices, prioritize the goals, and relax some constraints accordingly. You will see in this tutorial how the IDE guides you through this task.

What you are going to do

Working from the nurse scheduling example, you will:

- become familiar with the nurses project: see *Setting up the nurses model and data*
- solve the model and discover that it is infeasible: see *Executing the nurses project (1)*
- analyze and understand the suggested relaxation: see *Studying the suggested relaxation*

- analyze and understand suggested conflicts: see *Studying the suggested conflicts*
- change some data: see *Changing the data*
- execute the project again and find a solution to the feasible model: *Executing the nurses project (2)*
- learn more about *How relaxation and conflict search works*

Important: Conflicts and relaxations are supported only for models solved by the CPLEX engine. They are not currently supported with the CP Optimizer engine.

Where to find the files

The project folder is at the following location:

`<OPL_dir>\examples\opl\nurses\`

where `<OPL_dir>` is your installation directory.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

Setting up the nurses model and data

Use the **File>New>Example** menu command to open the **nurses** example.

The IDE displays the `nurses` project in the OPL Projects Navigator.

Open the model in the editing area. Since this is a rather long model, it cannot be shown entirely. Scroll down or resize the main window to see more of it.

The model instance, **nurses.mod**, contains the following elements:

- nurses
- nurse shifts
- nurse teams
- maximal and per-nurse work time
- nursing skills requirements
- hospital departments
- department incompatibilities
- vacations
- required assignments

Double-click **nurses.dat** in the OPL Projects Navigator to display the data instances associated with this model in the editing area.

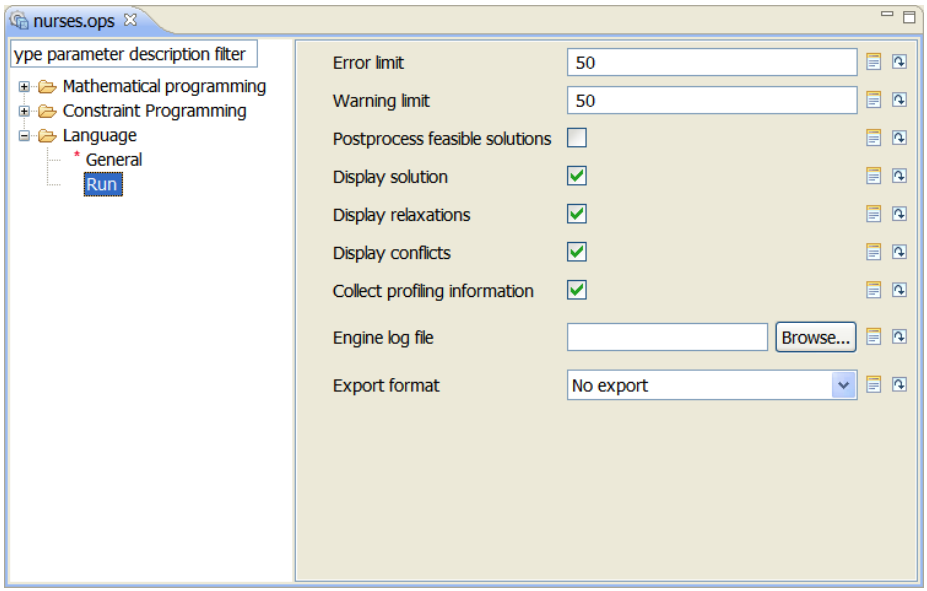
Notice that the Outline window presents a convenient view of the model or data elements.

Executing the nurses project (1)

The nurse scheduling project has been designed with infeasible data. The IDE lets you know when a project is infeasible, proposes relaxations, and displays conflicts to help you make it feasible. In this section, you are going to execute the model and observe infeasibility.

1. Double-click on the settings file, **nurses.ops**, to open it in the editing area.
2. In the displayed settings file, select **Language>Run** in the left panel and make sure that the **Display relaxations** and **Display conflicts** options are checked.

   These options, which are turned on by default, tell the IDE to compute and display suggested relaxations and conflicts, when they exist, in the **Conflicts** and **Relaxations** tabs.

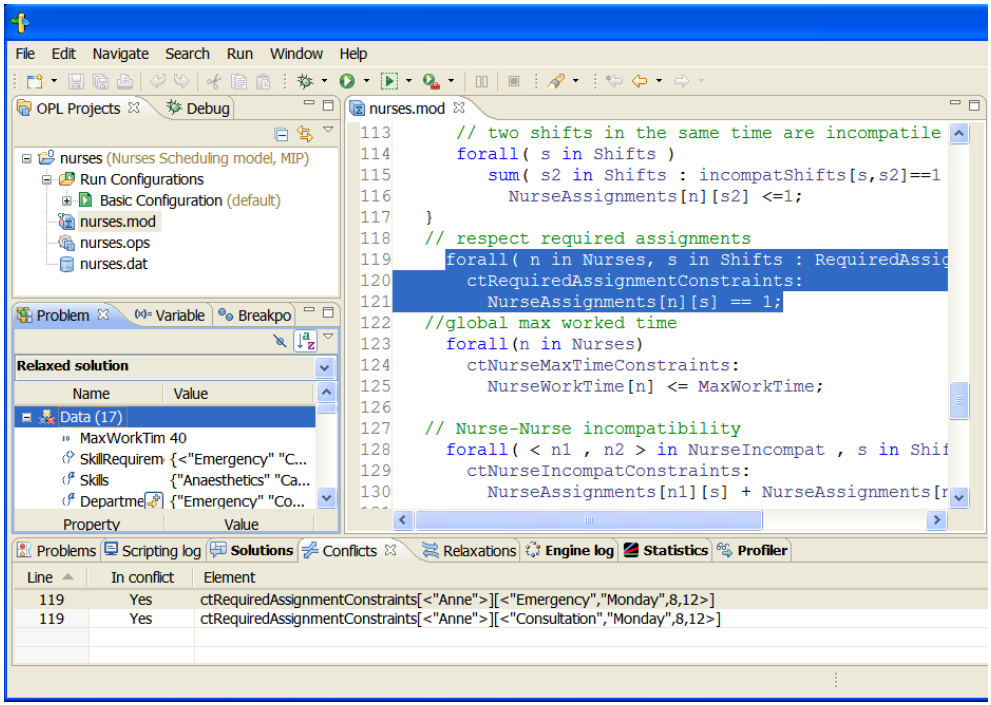


Relaxations and conflicts checked by default

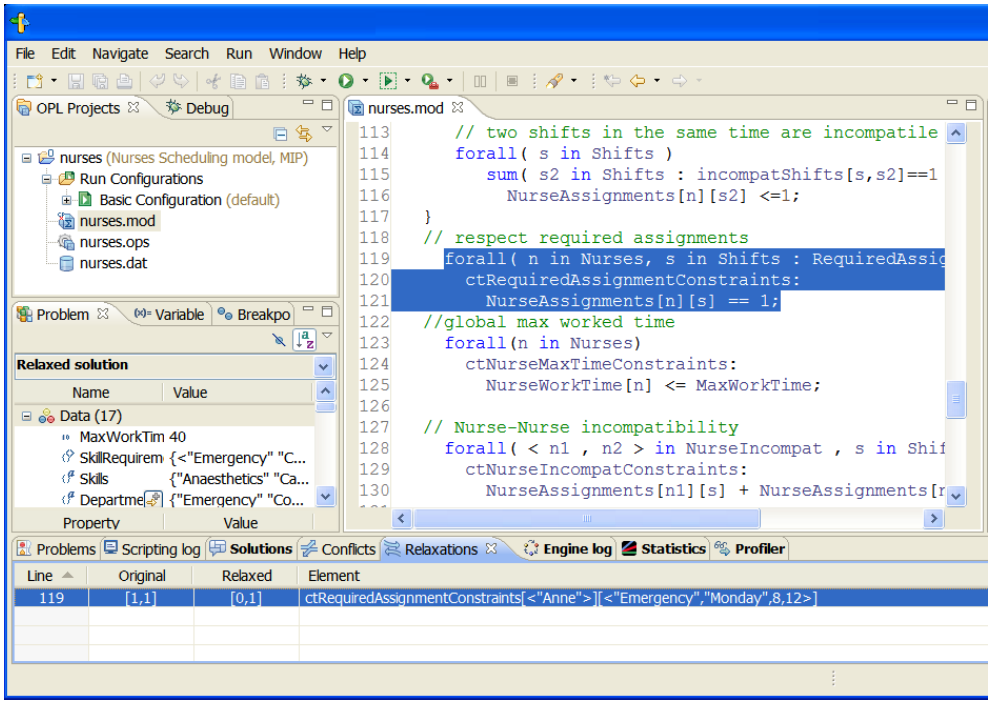
3. Right-click on the **nurses** project and select **Run>Basic configuration**.

   After a few seconds, the **Conflicts** and **Relaxations** tabs indicate where the relaxations and conflicts are, as shown in the following screen shots.

   When a conflict or relaxation is selected in either the **Conflicts** and **Relaxations** tabs, the corresponding code is highlighted in the model.



Displaying conflicts for nurses.mod



Displaying relaxations for nurses.mod

When you get such messages after execution of a model, the next step consists in *Studying the suggested relaxation*.

Working on the execution results

Describes how to study the conflict and suggested relaxation, and change the data to remove infeasibility.

In this section

Studying the suggested relaxation
Suggests places where you can change the data or the way constraints are expressed to remove incompatibilities.

Studying the suggested conflicts
Explains the conflict in the model and how to avoid it.

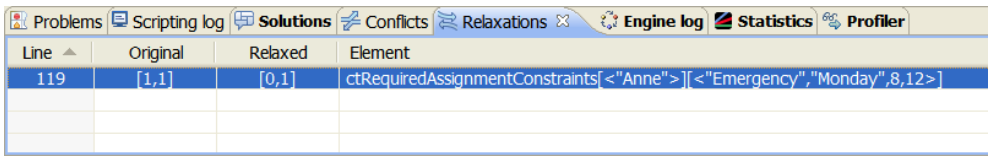
Changing the data
Shows how to change the data in the `nurses.dat` file.

Studying the suggested relaxation

When a model proves infeasible, the IDE suggests the places where you can change the data or the way constraints are expressed so as to remove the incompatibilities.

Note: Only ranged constraints are relaxable. Logical constraints are not.

Click the **Relaxations** tab. For the `nurses` example, you should see what is shown below.



Suggested relaxation (`nurses.mod`)

The proposed relaxation consists in changing the bound of the given constraint to `[0,1]` instead of `[1,1]`. The relaxation refers to the constraint on required assignments.

```
// respect required assignments
  forall( n in Nurses, s in Shifts : RequiredAssignments[n][s] == 1 )
    ctRequiredAssignmentConstraints:
      NurseAssignments[n][s] == 1;
```

This means that the equality to 1 would no longer be mandatory in the relaxed model. The constraint itself seems correctly expressed. Practically, it means that a feasible solution can be found, and hence a schedule can be worked out, if Anne is not necessarily assigned to Consultation on Monday from 8 to 12. In other words, changing this data is **sufficient** for the model to be feasible (see *Changing the data*).

Important: Be aware, however, that infeasibility may be the consequence of an error in the modeling of another constraint.

Let us now take another view at infeasibility by *Studying the suggested conflicts*.

Studying the suggested conflicts

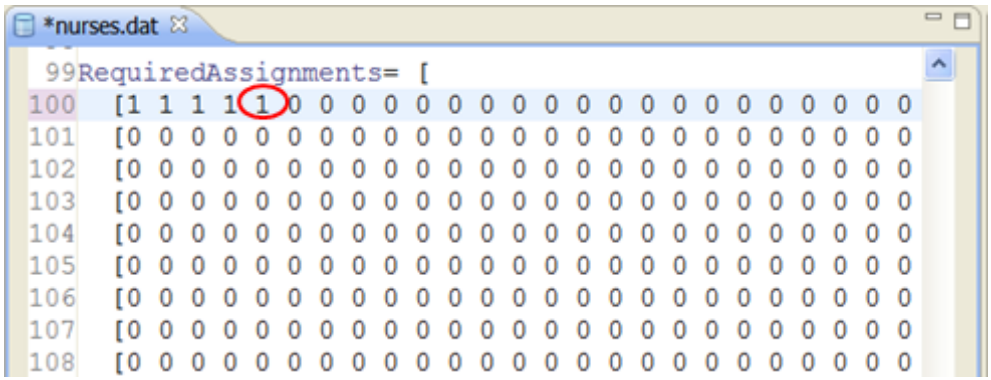
When a model is proved infeasible, OPL also searches possible conflicts between the constraints of the model. A conflict is a set of constraints that cannot be all true at the same time. At least one of them must be removed or modified to avoid the conflict. See *How relaxation and conflict search works* for more information.

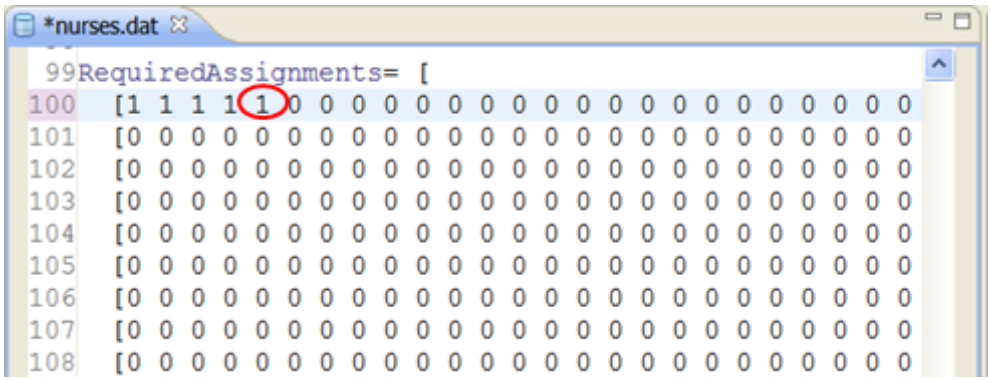
Return to the **Conflicts** tab if necessary. For the nurses example, the IDE displays the proposed conflict in the **Conflicts** tab.

The conflict is that Anne cannot be assigned simultaneously to both the Emergency and the Consultation shifts. In other words, to avoid this particular conflict, one of the required shifts (that is, one of the constraints on assignments to shifts) **must** be removed. If no other conflicts exist in the model, the model then becomes feasible.

Changing the data

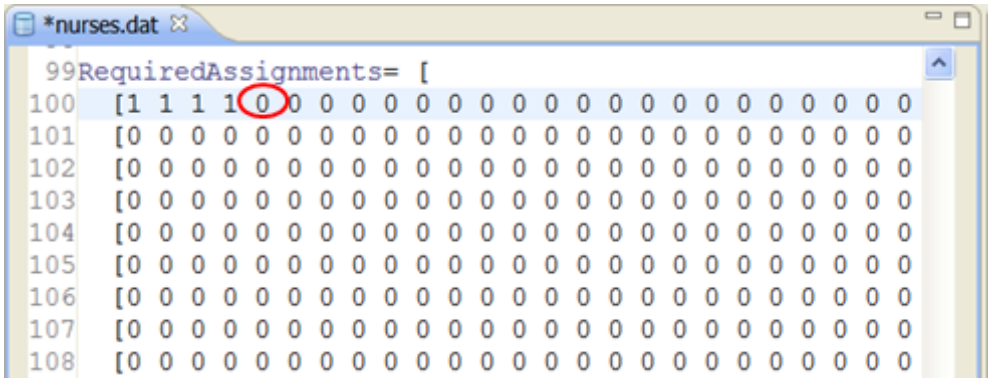
The list of required assignments is given in the data file `nurses.dat`.





In this example, it appears that the assignment to relax is the fifth value from the left in the top row.

1. Open **nurses.dat** in the Editing Area and scroll to the RequiredAssignments section.
2. Change it to **0** to make the model feasible.


*nurses.dat
99 RequiredAssignments= [
100 [1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
101 [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
102 [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
103 [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
104 [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
105 [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
106 [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
107 [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
108 [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

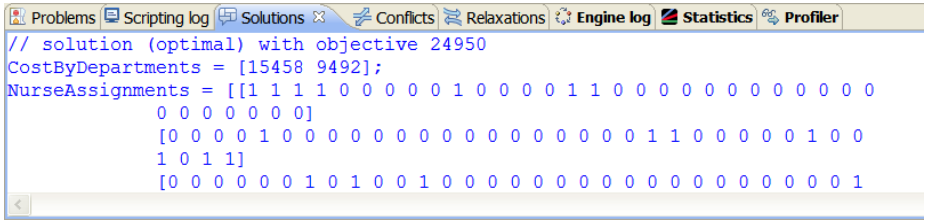
Executing the nurses project (2)

After you have changed an assignment value as shown in *Changing the data*, you are going to execute the project again and see the new result.

To observe the new result:

1. Right-click on the **nurses** project and select **Run>Basic Configuration**. Click **OK** in the popup to save the modified file.

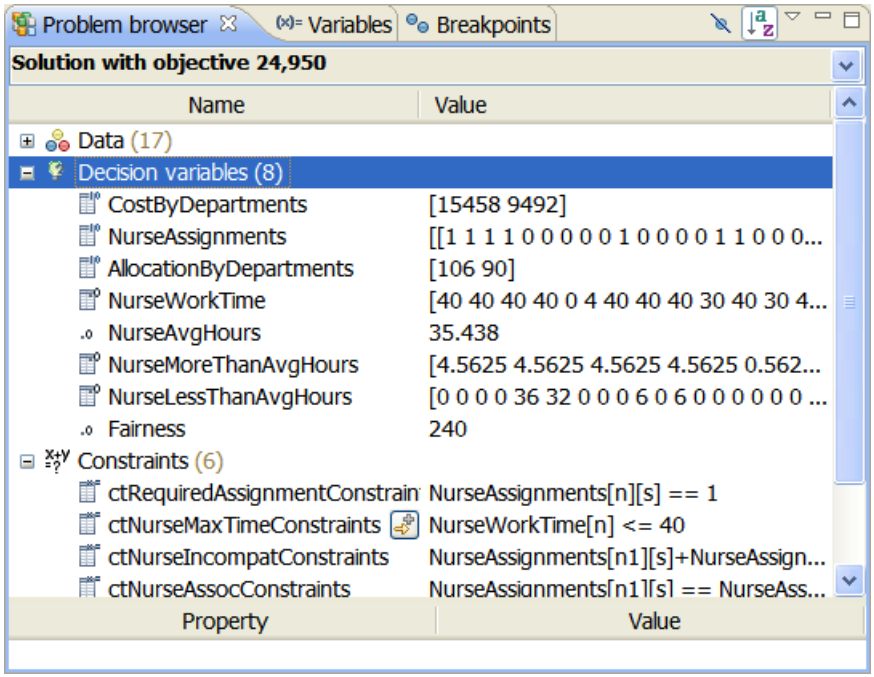
   The **Conflicts** and **Relaxations** tabs are now empty while the **Solutions** tab displays the solutions found. You now get a feasible solution with cost **24950**, as shown below.



Feasible model after change in data

   You can also see that Anne is assigned to Emergency on the “Monday 8 to 12” shift (not to Consultation). To view Anne’s assignment, do as follows.

2. In the Problem Browser, scroll down in the Name column to **Decision variables** / `NurseAssignments` as shown:



Problem Browser, scrolling to variables/NurseAssignments

3. Click the **Show data view** button in the `NurseAssignments` row.

   The table of nurse assignments opens, as shown.

*nurses.dat | Value for NurseAssignments

Nurses (size 32) | Shifts (size 35)

name	departmentName	day	startTime	endTime	Value
"Anne"	"Emergency"	"Monday"	2	8	1
"Anne"	"Emergency"	"Monday"	8	12	1
"Anne"	"Emergency"	"Monday"	12	18	1
"Anne"	"Emergency"	"Monday"	18	2	1
"Anne"	"Consultation"	"Monday"	8	12	0
"Anne"	"Consultation"	"Monday"	12	18	0
"Anne"	"Emergency"	"Tuesday"	8	12	0
"Anne"	"Emergency"	"Tuesday"	12	18	0
"Anne"	"Emergency"	"Tuesday"	18	2	0
"Anne"	"Consultation"	"Tuesday"	8	12	1
"Anne"	"Consultation"	"Tuesday"	12	18	0
"Anne"	"Emergency"	"Wednesday"	2	8	0
"Anne"	"Emergency"	"Wednesday"	8	12	0
"Anne"	"Emergency"	"Wednesday"	12	18	0
"Anne"	"Emergency"	"Wednesday"	18	2	1
"Anne"	"Consultation"	"Wednesday"	8	12	1
"Anne"	"Consultation"	"Wednesday"	12	18	0
"Anne"	"Emergency"	"Thursday"	2	8	0
"Anne"	"Emergency"	"Thursday"	8	12	0
"Anne"	"Emergency"	"Thursday"	12	18	0
"Anne"	"Emergency"	"Thursday"	18	2	0
"Anne"	"Consultation"	"Thursday"	8	12	0
"Anne"	"Consultation"	"Thursday"	12	18	0

A table view of NurseAssignments

The second line shows that Anne is assigned to Emergency, Monday 8-12.

For further information on views of model elements, see *Understanding the Problem Browser* in *Getting Started with the IDE*.

Now you may want to know more on *How relaxation and conflict search works*.

How relaxation and conflict search works

Relaxations and conflicts both express the infeasibility of a model and propose steps towards feasibility. After you have had hands-on experience with the nurse scheduling example, learn how to differentiate between Relaxations and Conflicts.

In this section

Relaxations
Provides a definition of a relaxation.

Setting the relaxation level
Explains how to set the relaxation level in the OPL IDE.

Conflicts
Discusses conflicts and potential conflicts.

Relaxations

In this context, a relaxation is a modified model where some bounds (of variables and/or constraints) have been made wider than in the first version. For example, an integer variable with the original bound `[0,100]` could be relaxed to `[0,200]`, or a range constraint `expr <= 10` could be relaxed to `expr <= 12`.

The relaxation search process looks for a way to make the model and its data more flexible so that the problem becomes feasible while keeping modifications to a minimum. In other words, the suggested relaxation in the **Relaxations** tab is a **sufficient minimal change** to make the model feasible.

More about constraint relaxation

More specifically, only constraints that have been labeled are taken into account for possible relaxation and conflicts. As variables are systematically named, they are always considered by the search process. Constraints, however, may be unlabeled. If you choose not to label the constraints in your model, they will not be taken into account for relaxation or conflict search. In this case, only the variables will be affected by the relaxation process, which may lead to surprising suggestions. Moreover, some existing conflicts may not be found. This is another reason why labeling constraints is a recommended practice (see *Constraint labels* in the *Language Reference Manual*).

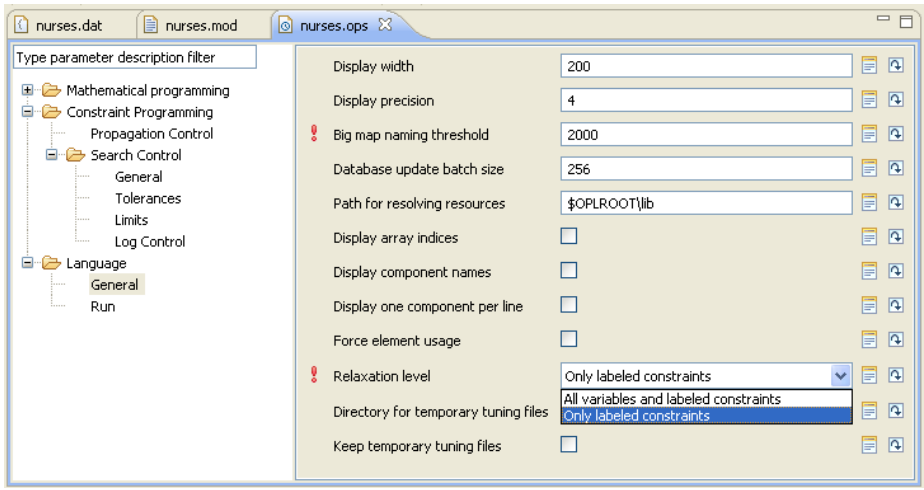
Setting the relaxation level

You can set whether you want OPL to relax only the labeled constraint or also the variables.

To set the relaxation level:

- In the settings file, `nurses.ops`, click **Language>General** and choose an option from the **Relaxation level** list.

  By default, both variables and constraints are relaxed. You can choose to relax only labeled constraints.



Setting the relaxation level

Conflicts

The conflict search process looks for a way to remove as many constraints as possible while it stays infeasible. Consequently, a conflict is the subset of constraints from the infeasible model in which removing any of the constraints makes that set feasible. When a conflict is displayed in the **Conflicts** output window, it expresses the **necessary change** to make the model feasible: you must remove or modify at least one of the conflicting constraints.

Important:

1. Be aware that minimality may not be achieved if the search process is stopped by a time limit, an unexpected interruption, or some similar event.
2. If the model happens to contain multiple independent causes of infeasibility, it may be necessary for the user to repair one cause and then repeat the process with a further refinement.

A basic example summarizes the difference still more clearly.

```
dvar int+ x;
dvar int+ y;

maximize y;
subject to {
   ct1: x >= 10;
   ct2: x <= 0;
}
```

This model contains two incompatible constraints. To remove this incompatibility, you can:

- either change `ct1` to `x <= 0` : this is relaxing the constraint;
- or remove `ct1` or `ct2` : this is removing the conflicting constraint.

Potential conflicts

The CPLEX® engine differentiates in infeasible models between *identified* conflicts, as described in this tutorial from the `nurses` example, and *possible* conflicts.

It may happen that the engine is not capable of completely proving the conflict. In this case, it returns some constraint that “may be” in the conflict. This corresponds to the `ConflictPossibleMember` status of the method `IloCplex:getConflict`.

Profiling the execution of a model

Explains how the IDE Profiler table can help you improve your model to make it run faster while consuming less memory.

In this section

Purpose and prerequisites
Explains what to do in this tutorial and where to find the files.

Identifying slow and memory-consuming model elements
The information displayed in the table of the Profiler enables you to identify slow or memory-consuming elements during execution of the profiler model, and, possibly, to infer what changes in your model might improve execution performance. This part of the tutorial uses the profiler example to demonstrate this.

Examining model extraction and solving
In addition to time and memory consumption, the Profiler table displays the extraction and solving phases of the model execution. This part of the tutorial uses the scalable configuration of the warehouse example to demonstrate this.

Purpose and prerequisites

When you execute a run configuration, information about execution time, memory consumption, and model extraction is collected as part of the execution. When execution stops, the information appears as a table in the Profiler output window.

This tutorial uses models solved with the CPLEX engine but all the features described work in the same way with models solved by the CP Optimizer engine.

Important: In this tutorial, “faster” and “consuming less memory” may apply both to creating the model and to searching for solutions. This tutorial gives hints only on how to improve the model creation part. For the model solving part, the tutorial explains what kind of information the profiler can display. Using this information to make the model solve faster falls out of the scope of this documentation.

In other words, this tutorial does not explain how to write a better model that finds an optimal solution faster.

Prerequisites

It is assumed that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

What you are going to do

The tutorial is based on two different examples:

- Working with the `profiler` example described in *Presenting the profiler example*, you will:
  - solve the model: see *Executing the profiler model*
  - become familiar with the profiling information: see *Description of the profiling information*
  - identify slow or memory consuming elements: see *Examining the profiling information*
- Then, working with the scalable configuration of the warehouse example described in *Presenting the scalable run configuration*, you will:
  - solve the model: see *Executing the scalable run configuration*
  - look at the information on model extraction and engine search: see *Examining the extraction and search information*

Where to find the files

The profiler example is supplied as the model **profiler.mod** included in the `profiler` project at the following location:

<OPL_dir>\examples\opl\profiler\

The scalable warehouse example is a run configuration of the warehouse project at the following location:

<OPL_dir>\examples\opl\warehouse\

where <OPL_dir> is your installation directory.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

Identifying slow and memory-consuming model elements

The information displayed in the table of the Profiler enables you to identify slow or memory-consuming elements during execution of the profiler model, and, possibly, to infer what changes in your model might improve execution performance. This part of the tutorial uses the profiler example to demonstrate this.

In this section

Presenting the profiler example
Discusses the model `profiler.mod`.

Executing the profiler model
Explains how to obtain profiling information.

Description of the profiling information
Describes the information about execution time, memory consumption and model extraction, collected after execution of the model.

Examining the profiling information
Explains how to analyze time and memory consumption after execution of the model.

Presenting the profiler example

The profiler example is a variation of the transportation example (see Profiling in the *Samples* manual).

Use the **File>New>Example** menu command to open the **profiler** example.

The IDE displays the `profiler` project in the OPL Projects Navigator. Open the model in the editing area.

Model of the Profiler example (`profiler.mod`)

```
int n = 300;
range r = 1..n;
int Values1[r][r];

execute INIT_Values1 {
  for( var i in r )
    for( var j in r )
      if ( i == 2*j )
        Values1[i][j] = i+j;
  writeln(Values1);
}

int Values2[i in r][j in r] = (i==2*j) ? i+j : 0;

execute INIT_Values2 {
  writeln(Values2);
}

tuple T {
  int i;
  int j;
}
{T} indexes = { < i , 2 * i > | i in r };
int Values3[<i,j> in indexes] = i+j;

execute INIT_Values3 {
  writeln(Values3);
}
```

The `profiler` example is supplied with no data file. This example is a dummy unrealistic model where the same data is created in three different ways so as to demonstrate how the profiler helps you find which data is slow and/or memory consuming.

The code extract above shows that the identical data to be created is a structure of values depending on two indices `i` and `j`. Non-null values exist only when `i` is equal to `2*j`. The values are then `i+j`.

In the supplied example, `i` and `j` range from 1 to 300. This range makes the execution run in an acceptable amount of time on an average laptop computer. Still, the model is not too small, so that differences in modeling significantly affect execution time and memory consumption. You can change the value of the upper range limit `n` to adjust the execution time if it is too fast or too slow on your own machine.

Executing the profiler model

To execute the model and display the Profiler table:

1. Right-click on the profile project and select **Run>Basic Configuration**. (If necessary, see *Executing a project* in *Getting Started with the IDE* for a reminder of the execution process.)

   The model solves with no error message.

2. Click the **Profiler** tab to display profiling information.

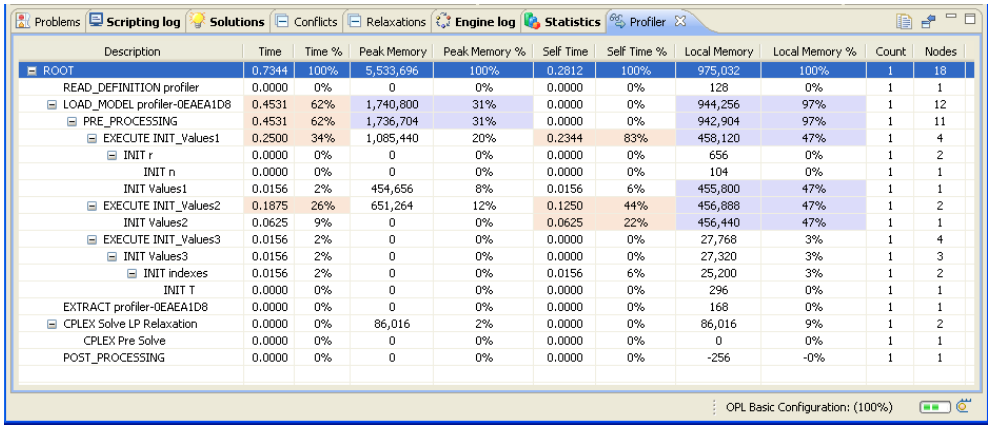
Next, read the *Description of the profiling information*.

Description of the profiling information

When execution stops, the information collected about execution time, memory consumption, and model extraction is organized in tabular form in the Profiler tab, as shown below.

Note:
1. If the profiler table is empty and the run configuration includes a settings file, first make sure the “Collect profiling information” option is turned on in the Language/General window of the settings editor.
2. The figures shown in the illustrations may be different on your machine.

Each column header is a sort criterion (see *Sorting*) and there are two icons at the top right of the Profiler tab for the commands **Copy contents to clipboard** and **Customize thresholds**. The Description column presents the execution steps in sequential order as a tree. The root item corresponds to the full execution.



Description	Time	Time %	Peak Memory	Peak Memory %	Self Time	Self Time %	Local Memory	Local Memory %	Count	Nodes
ROOT	0.7344	100%	5,533,696	100%	0.2812	100%	975,032	100%	1	18
READ_DEFINITION profiler	0.0000	0%	0	0%	0.0000	0%	128	0%	1	1
LOAD_MODEL profiler-0EAEA1D8	0.4531	62%	1,740,800	31%	0.0000	0%	944,256	97%	1	12
PRE_PROCESSING	0.4531	62%	1,736,704	31%	0.0000	0%	942,904	97%	1	11
EXECUTE INIT_Values1	0.2500	34%	1,085,440	20%	0.2344	83%	458,120	47%	1	4
INIT r	0.0000	0%	0	0%	0.0000	0%	656	0%	1	2
INIT n	0.0000	0%	0	0%	0.0000	0%	104	0%	1	1
INIT Values1	0.0156	2%	454,656	8%	0.0156	6%	455,800	47%	1	1
EXECUTE INIT_Values2	0.1875	26%	651,264	12%	0.1250	44%	456,888	47%	1	2
INIT Values2	0.0625	9%	0	0%	0.0625	22%	456,440	47%	1	1
EXECUTE INIT_Values3	0.0156	2%	0	0%	0.0000	0%	27,768	3%	1	4
INIT Values3	0.0156	2%	0	0%	0.0000	0%	27,320	3%	1	3
INIT indexes	0.0156	2%	0	0%	0.0156	6%	25,200	3%	1	2
INIT T	0.0000	0%	0	0%	0.0000	0%	296	0%	1	1
EXTRACT profiler-0EAEA1D8	0.0000	0%	0	0%	0.0000	0%	168	0%	1	1
CPLEX Solve LP Relaxation	0.0000	0%	86,016	2%	0.0000	0%	86,016	9%	1	2
CPLEX Pre Solve	0.0000	0%	0	0%	0.0000	0%	0	0%	1	1
POST_PROCESSING	0.0000	0%	0	0%	0.0000	0%	-256	-0%	1	1

Profiling information for the `profiler` example

The Description tree

Most categories below the root item include two important kinds of item, as shown in *Execution tree items*.

Execution tree items

Execution steps	Notation
Script block execution	EXECUTE <block_name>
Data initialization	INIT <data_element_name>

Then, for each execution step, the elements listed below are displayed in columns.

Profiling information measured for each execution step

Information for each execution step	Definition
Time	The total time consumed by the task...
Percentage of time	...as a proportion of the total time
Peak Memory	The maximal memory used to process the OPL problem. See also the Glossary.
Percentage of peak memory	... as a proportion of the total memory
Self time	The time used by the task minus the time used by the subtasks...
Percentage of self time	...as a proportion of the total time
Local Memory	The memory usage observed by comparing the start and the end of a step. See also the Glossary.
Percentage of local memory	...as a proportion of the total memory
Count	The number of times that same node is repeated on this tree level
Nodes	The number of nodes in the branch starting at this node

Processing time vs. user time

All figures indicated in the **Time** column are **processing** times, not **user** times. This means that if your computer is executing other tasks at the same time or is paging because it has too much memory to handle, these extra times are not taken into account by the profiler.

In other words, the OPL profiler records the user+kernel time (which Microsoft® calls the *process time*) given the following definitions.

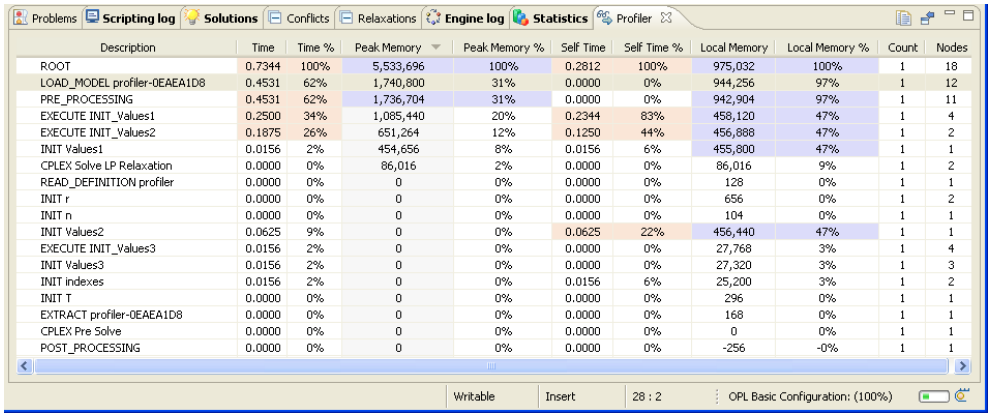
elapsed time	The total elapsed (wall-clock) time required for an operation. This includes time spent waiting for I/O, synchronization objects, timers, scheduling, and other delays.
user+kernel time	The amount of time executing code in the thread. It does not include time spent waiting for devices or servicing other processes.

Sorting

By default, the Profiler table rows are sorted sequentially in the order of execution, as reflected by the Description tree. However, you can sort the table, in ascending or descending order, on the figures of any column by clicking on that column header name.

For example, in *Profiler table sorted on the peak memory column (`profiler.mod`)*, the figures in the Peak Memory column are sorted in descending order and as a consequence, the Description tree is no longer displayed sequentially. Compare to *Profiling information for the `profiler` example*, which shows the default, sequential order.

Click the Description column header to restore the default order.



Description	Time	Time %	Peak Memory	Peak Memory %	Self Time	Self Time %	Local Memory	Local Memory %	Count	Nodes
ROOT	0.7344	100%	5,533,696	100%	0.2812	100%	975,032	100%	1	18
LOAD_MODEL profiler-0EAEA1D8	0.4531	62%	1,740,800	31%	0.0000	0%	944,256	97%	1	12
PRE_PROCESSING	0.4531	62%	1,736,704	31%	0.0000	0%	942,904	97%	1	11
EXECUTE INIT_Values1	0.2500	34%	1,085,440	20%	0.2344	83%	458,120	47%	1	4
EXECUTE INIT_Values2	0.1875	26%	651,264	12%	0.1250	44%	456,888	47%	1	2
INIT Values1	0.0156	2%	454,656	8%	0.0156	6%	455,800	47%	1	1
CPLEX Solve LP Relaxation	0.0000	0%	86,016	2%	0.0000	0%	86,016	9%	1	2
READ_DEFINITION profiler	0.0000	0%	0	0%	0.0000	0%	128	0%	1	1
INIT r	0.0000	0%	0	0%	0.0000	0%	656	0%	1	2
INIT n	0.0000	0%	0	0%	0.0000	0%	104	0%	1	1
INIT Values2	0.0625	9%	0	0%	0.0625	22%	456,440	47%	1	1
EXECUTE INIT_Values3	0.0156	2%	0	0%	0.0000	0%	27,768	3%	1	4
INIT Values3	0.0156	2%	0	0%	0.0000	0%	27,320	3%	1	3
INIT indexes	0.0156	2%	0	0%	0.0156	6%	25,200	3%	1	2
INIT T	0.0000	0%	0	0%	0.0000	0%	296	0%	1	1
EXTRACT profiler-0EAEA1D8	0.0000	0%	0	0%	0.0000	0%	168	0%	1	1
CPLEX Pre Solve	0.0000	0%	0	0%	0.0000	0%	0	0%	1	1
POST_PROCESSING	0.0000	0%	0	0%	0.0000	0%	-256	-0%	1	1

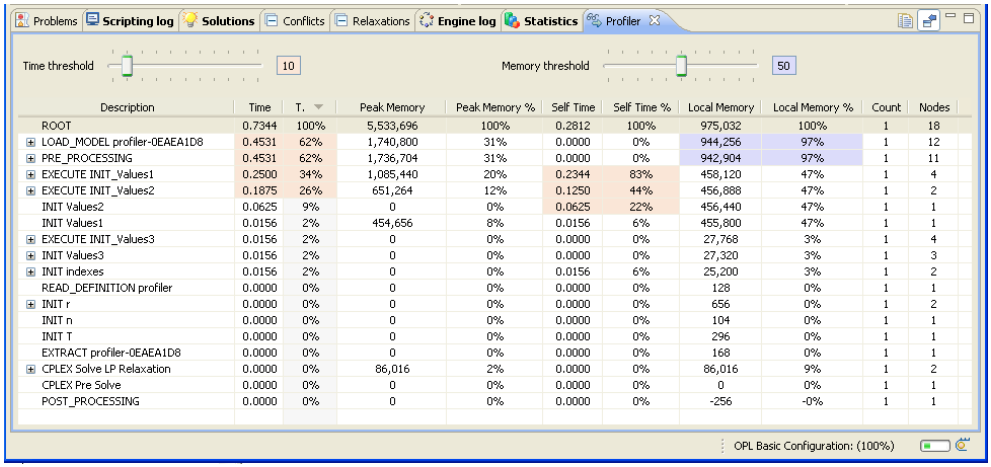
Profiler table sorted on the peak memory column (`profiler.mod`)

Thresholds

Click the **Customize thresholds** icon  at the top right of the Profiler tab to display the sliders.

The **Time % threshold** and **Memory % threshold** sliders, at the top of the Output window, enable you to dynamically change the minimum percentage of time or memory above which you want the figures to appear on a blue (time) or pink (memory) background. Compare *Time % threshold set to 10 and memory % threshold set to 50 (`profiler.mod`)* to *Profiling information for the `profiler` example*: more time figures are shown against a blue background when the time threshold is set to 10% and fewer memory figures appear against a pink background when the memory threshold is set to 50%.

Note: The figures shown in the illustrations may be different on your machine.

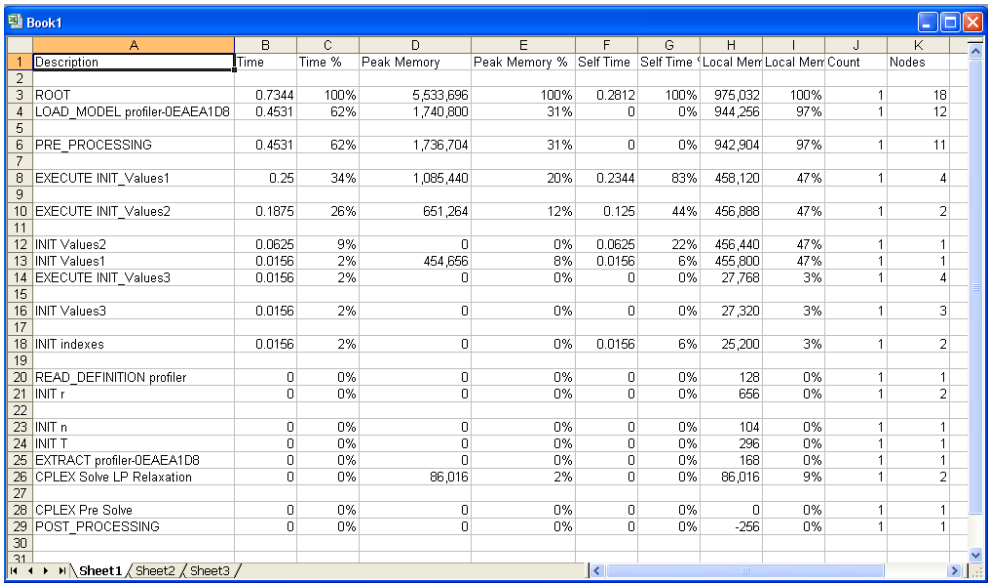


Description	Time	T.	Peak Memory	Peak Memory %	Self Time	Self Time %	Local Memory	Local Memory %	Count	Nodes
ROOT	0.7344	100%	5,533,696	100%	0.2812	100%	975,032	100%	1	18
⊞ LOAD_MODEL profiler-0EAEA1D8	0.4531	62%	1,740,800	31%	0.0000	0%	944,256	97%	1	12
⊞ PRE_PROCESSING	0.4531	62%	1,736,704	31%	0.0000	0%	942,904	97%	1	11
⊞ EXECUTE INIT_Values1	0.2500	34%	1,085,440	20%	0.2344	83%	458,120	47%	1	4
⊞ EXECUTE INIT_Values2	0.1875	26%	651,264	12%	0.1250	44%	456,888	47%	1	2
INIT Values2	0.0625	9%	0	0%	0.0625	22%	456,440	47%	1	1
INIT Values1	0.0156	2%	454,656	8%	0.0156	6%	455,800	47%	1	1
⊞ EXECUTE INIT_Values3	0.0156	2%	0	0%	0.0000	0%	27,768	3%	1	4
⊞ INIT Values3	0.0156	2%	0	0%	0.0000	0%	27,320	3%	1	3
⊞ INIT indexes	0.0156	2%	0	0%	0.0156	6%	25,200	3%	1	2
READ_DEFINITION profiler	0.0000	0%	0	0%	0.0000	0%	128	0%	1	1
⊞ INIT r	0.0000	0%	0	0%	0.0000	0%	656	0%	1	2
INIT n	0.0000	0%	0	0%	0.0000	0%	104	0%	1	1
INIT T	0.0000	0%	0	0%	0.0000	0%	296	0%	1	1
EXTRACT profiler-0EAEA1D8	0.0000	0%	0	0%	0.0000	0%	168	0%	1	1
⊞ CPLEX Solve LP Relaxation	0.0000	0%	86,016	2%	0.0000	0%	86,016	9%	1	2
CPLEX Pre Solve	0.0000	0%	0	0%	0.0000	0%	0	0%	1	1
POST_PROCESSING	0.0000	0%	0	0%	0.0000	0%	-256	-0%	1	1

Time % threshold set to 10 and memory % threshold set to 50 (`profiler.mod`)

Copying Profiler contents to a clipboard

Click the **Copy contents to clipboard** icon  at the top right of the Profiler tab to copy the Profiler table and then paste it to a text editor or spreadsheet. The following screen shows the Profiler table in Microsoft® Excel.



	A	B	C	D	E	F	G	H	I	J	K
1	Description	Time	Time %	Peak Memory	Peak Memory %	Self Time	Self Time %	Local Mem	Local Mem	Count	Nodes
2											
3	ROOT	0.7344	100%	5,533,696	100%	0.2812	100%	975,032	100%	1	18
4	LOAD_MODEL profiler-0EAEA1D8	0.4531	62%	1,740,800	31%	0	0%	944,256	97%	1	12
5											
6	PRE_PROCESSING	0.4531	62%	1,736,704	31%	0	0%	942,904	97%	1	11
7											
8	EXECUTE INIT_Values1	0.25	34%	1,085,440	20%	0.2344	83%	458,120	47%	1	4
9											
10	EXECUTE INIT_Values2	0.1875	26%	651,264	12%	0.125	44%	456,888	47%	1	2
11											
12	INIT Values2	0.0625	9%	0	0%	0.0625	22%	456,440	47%	1	1
13	INIT Values1	0.0156	2%	454,656	8%	0.0156	6%	455,800	47%	1	1
14	EXECUTE INIT_Values3	0.0156	2%	0	0%	0	0%	27,768	3%	1	4
15											
16	INIT Values3	0.0156	2%	0	0%	0	0%	27,320	3%	1	3
17											
18	INIT indexes	0.0156	2%	0	0%	0.0156	6%	25,200	3%	1	2
19											
20	READ_DEFINITION profiler	0	0%	0	0%	0	0%	128	0%	1	1
21	INIT r	0	0%	0	0%	0	0%	656	0%	1	2
22											
23	INIT n	0	0%	0	0%	0	0%	104	0%	1	1
24	INIT T	0	0%	0	0%	0	0%	296	0%	1	1
25	EXTRACT profiler-0EAEA1D8	0	0%	0	0%	0	0%	168	0%	1	1
26	CPLEX Solve LP Relaxation	0	0%	86,016	2%	0	0%	86,016	9%	1	2
27											
28	CPLEX Pre Solve	0	0%	0	0%	0	0%	0	0%	1	1
29	POST_PROCESSING	0	0%	0	0%	0	0%	-256	0%	1	1
30											
31											

The next step consists in *Examining the profiling information*.

Examining the profiling information

You are now going to analyze the time and memory figures of the `profiler.mod` example.

Note: The figures shown in the illustrations may be different on your machine.

Slow elements

In the profiling context, slow elements are elements with a bad execution time. All three data structures are equivalent (see *Presenting the profiler example*). However, if you look at the first column (Time) of the Profiler table, you can see that execution of `EXECUTE INIT_Values1` (which includes initializing the structure plus setting the values in the script) takes nearly twice as long as execution of `EXECUTE INIT_Values2` (which includes initialization of `Values2`).

PRE_PROCESSING	0.7188	94%	5,394,432	99%	0.0000	0%	942,904	97%	1	11
EXECUTE INIT_Values1	0.2969	39%	1,404,928	26%	0.2969	633%	458,120	47%	1	4
INIT r	0.0000	0%	0	0%	0.0000	0%	656	0%	1	2
INIT n	0.0000	0%	0	0%	0.0000	0%	104	0%	1	1
INIT Values1	0.0000	0%	442,368	8%	0.0000	0%	455,800	47%	1	1
EXECUTE INIT_Values2	0.4219	55%	3,989,504	73%	0.3281	700%	456,888	47%	1	2
INIT Values2	0.0938	12%	569,344	10%	0.0938	200%	456,440	47%	1	1

Comparing execution time (`profiler.mod`)

In other words, inline initialization of the `Values2` array is faster than initialization in the script.

Memory consuming elements

If you now examine the Profiler table from the point of view of memory consumption and look at the Local Memory column, you can see that initialization of `Values2` uses much more memory than initialization of `Values3`. It even increases the Peak Memory.

EXECUTE INIT_Values2	0.4219	55%	3,989,504	73%	0.3281	700%	456,888	47%	1	2
INIT Values2	0.0938	12%	569,344	10%	0.0938	200%	456,440	47%	1	1
EXECUTE INIT_Values3	0.0000	0%	0	0%	0.0000	0%	27,768	3%	1	4
INIT Values3	0.0000	0%	0	0%	0.0000	0%	27,320	3%	1	3
INIT indexes	0.0000	0%	0	0%	0.0000	0%	25,200	3%	1	2
INIT T	0.0000	0%	0	0%	0.0000	0%	296	0%	1	1

Memory consumption (`profiler.mod`)

This shows the interest of using tuple-based indexing when the structure is very sparse.

Examining model extraction and solving

In addition to time and memory consumption, the Profiler table displays the extraction and solving phases of the model execution. This part of the tutorial uses the scalable configuration of the warehouse example to demonstrate this.

In this section

Presenting the scalable run configuration
Discusses the model `scalableWarehouse.mod`.

Executing the scalable run configuration
Explains how to modify the default run configuration.

Examining the extraction and search information
After execution of the model.

Turning off the Profiler
To save execution time and memory.

Drawing conclusions
Contains some recommendations for improving your model.

Presenting the scalable run configuration

The warehouse location model is described in detail in Warehouse location problems in the *Samples* manual. It includes a run configuration **Scalable data** defined by the model **scalableWarehouse.mod** supplied with the `warehouse` project at the following location:

```
<OPL_dir>\examples\opl\warehouse\
```

where `<OPL_dir>` is your installation directory.

There is no data file in that configuration. The scalable data is declared in the `scalableWarehouse.mod` model, as shown in *Model of the scalableWarehouse example (`scalableWarehouse.mod`)*.

Use the **File>New>Example** menu command to open the **warehouse** example.

The IDE displays the project in the OPL Projects Navigator. Open the model in the editing area.

Model of the scalableWarehouse example (`scalableWarehouse.mod`)

```
int Fixed        = 10;
int NbWarehouses = 50;
int NbStores     = 200;

assert( NbStores > NbWarehouses );

range Warehouses = 1..NbWarehouses;
range Stores     = 1..NbStores;
int Capacity[w in Warehouses] =
  NbStores div NbWarehouses +
  w % ( NbStores div NbWarehouses );
int SupplyCost[s in Stores][w in Warehouses] =
  1 + ( ( s + 10 * w ) % 100 );
dvar int Open[Warehouses] in 0..1;
dvar float Supply[Stores][Warehouses] in 0..1;
dexpr int TotalFixedCost = sum( w in Warehouses ) Fixed * Open[w];
dexpr float TotalSupplyCost = sum( w in Warehouses, s in Stores )  SupplyCost
[s][w] * Supply[s][w];
minimize TotalFixedCost + TotalSupplyCost;

subject to {
  forall( s in Stores )
    ctStoreHasOneWarehouse:
      sum( w in Warehouses )
        Supply[s][w] == 1;
  forall( w in Warehouses )
    ctOpen:
      sum( s in Stores )
        Supply[s][w] <= Open[w] * Capacity[w];
}
```

Executing the scalable run configuration

Basic Configuration is predefined as the default run configuration. Since it is not the configuration you want to work on, you will first make another run configuration the default one.

To execute the run configuration:

1. Right-click **Scalable data** and choose **Set as default** .

   The word 'default' appears in parentheses after the run configuration name.

2. Right-click on **Scalable data** and select **Run this** . (If necessary, see *Executing a project* in *Getting Started with the IDE* for a reminder of the execution process.)

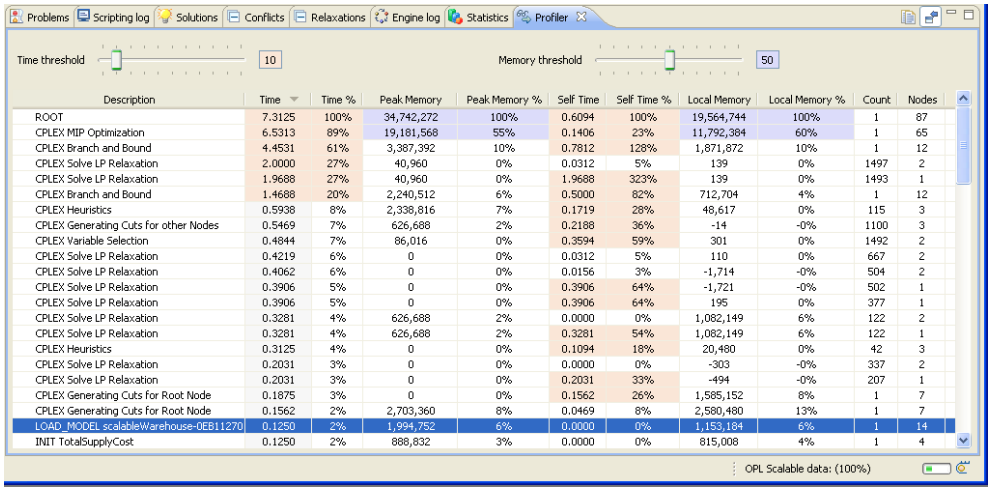
   The model solves with no error message.

3. Click the **Profiler** tab in the Output window.

The next step consists in *Examining the extraction and search information*.

Examining the extraction and search information

The extraction, search, and execution information is collected in a table as explained in *Description of the profiling information*.



Description	Time	Time %	Peak Memory	Peak Memory %	Self Time	Self Time %	Local Memory	Local Memory %	Count	Nodes
ROOT	7.3125	100%	34,742,272	100%	0.6094	100%	19,564,744	100%	1	87
CPLEX MIP Optimization	6.5313	89%	19,181,568	55%	0.1406	23%	11,792,384	60%	1	65
CPLEX Branch and Bound	4.4531	61%	3,387,392	10%	0.7812	128%	1,871,872	10%	1	12
CPLEX Solve LP Relaxation	2.0000	27%	40,960	0%	0.0312	5%	139	0%	1497	2
CPLEX Solve LP Relaxation	1.9688	27%	40,960	0%	1.9688	323%	139	0%	1493	1
CPLEX Branch and Bound	1.4688	20%	2,240,512	6%	0.5000	82%	712,704	4%	1	12
CPLEX Heuristics	0.5938	8%	2,338,816	7%	0.1719	28%	48,617	0%	115	3
CPLEX Generating Cuts for other Nodes	0.5469	7%	626,688	2%	0.2188	36%	-14	-0%	1100	3
CPLEX Variable Selection	0.4844	7%	86,016	0%	0.3594	59%	301	0%	1492	2
CPLEX Solve LP Relaxation	0.4219	6%	0	0%	0.0312	5%	110	0%	667	2
CPLEX Solve LP Relaxation	0.4062	6%	0	0%	0.0156	3%	-1,714	-0%	504	2
CPLEX Solve LP Relaxation	0.3906	5%	0	0%	0.3906	64%	-1,721	-0%	502	1
CPLEX Solve LP Relaxation	0.3906	5%	0	0%	0.3906	64%	195	0%	377	1
CPLEX Solve LP Relaxation	0.3281	4%	626,688	2%	0.0000	0%	1,082,149	6%	122	2
CPLEX Solve LP Relaxation	0.3281	4%	626,688	2%	0.3281	54%	1,082,149	6%	122	1
CPLEX Heuristics	0.3125	4%	0	0%	0.1094	18%	20,480	0%	42	3
CPLEX Solve LP Relaxation	0.2031	3%	0	0%	0.0000	0%	-303	-0%	337	2
CPLEX Solve LP Relaxation	0.2031	3%	0	0%	0.2031	33%	-494	-0%	207	1
CPLEX Generating Cuts for Root Node	0.1875	3%	0	0%	0.1562	26%	1,585,152	8%	1	7
CPLEX Generating Cuts for Root Node	0.1562	2%	2,703,360	8%	0.0469	8%	2,580,480	13%	1	7
LOAD_MODEL scalableWarehouse-0EB11270	0.1250	2%	1,994,752	6%	0.0000	0%	1,153,184	6%	1	14
INIT TotalSupplyCost	0.1250	2%	888,832	3%	0.0000	0%	815,008	4%	1	4

Profiling information for the scalable configuration of the warehouse example

You can see the two columns **Count** and **Nodes**.

- The **Count** column displays the number of times that same node is repeated on this tree level. This count is 1 most of the time, except for extractions of labeled leaf constraints inside a `forall` statement, and during the CPLEX® search.
- The **Nodes** column displays the number of subnodes in the branch starting at this node. The node count for leaves is 1.

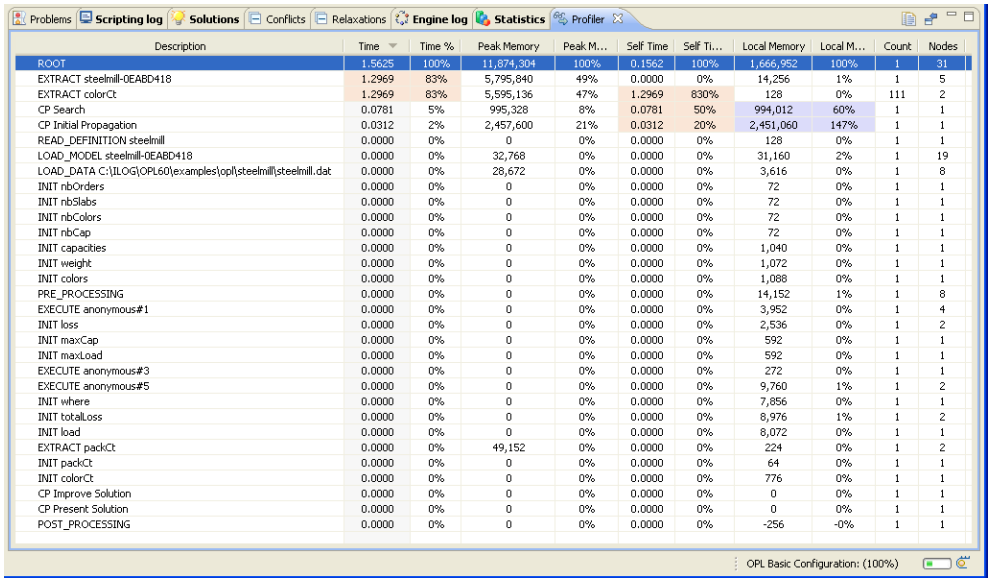
In the Description tree, you can see an item called `EXTRACTING <model_name>-<random_number>`. This branch of the execution tree includes all of the extraction phase of the model from OPL to the CPLEX® engine. Each labeled constraint is shown as a leaf as it is extracted. A constraint may take much more time and memory than another even if it appears very simple in OPL. This reflects the fact that the aggregate `forall` construct and the slicing feature enable you to write very complex sets of linear constraints in a compact OPL formulation.

Even a simple `forall` statement may be constructed on top of a huge set and then translated into a lot of constraints in the CPLEX matrix.

You can see another branch, at the top of the tree, called `CPLEX MIP Optimization` which reflects the CPLEX search. Different CPLEX suboperations are described. You may see relevant information such as the amount of time spent generating cuts and using heuristics to find solutions.

Profiler for a CP model

For a constraint programming model, the extraction and solving information appears as shown in *CP models: Profiler (`steelmill` project)*.



Description	Time	Time %	Peak Memory	Peak M...	Self Time	Self Ti...	Local Memory	Local M...	Count	Nodes
ROOT	1.5625	100%	11,874,304	100%	0.1562	100%	1,666,952	100%	1	31
EXTRACT steelmill-0EABD418	1.2969	83%	5,795,840	49%	0.0000	0%	14,256	1%	1	5
EXTRACT colorCt	1.2969	83%	5,595,136	47%	1.2969	830%	128	0%	111	2
CP Search	0.0781	5%	995,328	8%	0.0781	50%	994,012	60%	1	1
CP Initial Propagation	0.0312	2%	2,457,600	21%	0.0312	20%	2,451,060	147%	1	1
READ_DEFINITION steelmill	0.0000	0%	0	0%	0.0000	0%	128	0%	1	1
LOAD_MODEL steelmill-0EABD418	0.0000	0%	32,768	0%	0.0000	0%	31,160	2%	1	19
LOAD_DATA C:\ILOG\OPL60\examples\opl\steelmill\steelmill.dat	0.0000	0%	28,672	0%	0.0000	0%	3,616	0%	1	8
INIT nbOrders	0.0000	0%	0	0%	0.0000	0%	72	0%	1	1
INIT nbSlabs	0.0000	0%	0	0%	0.0000	0%	72	0%	1	1
INIT nbColors	0.0000	0%	0	0%	0.0000	0%	72	0%	1	1
INIT nbCap	0.0000	0%	0	0%	0.0000	0%	72	0%	1	1
INIT capacities	0.0000	0%	0	0%	0.0000	0%	1,040	0%	1	1
INIT weight	0.0000	0%	0	0%	0.0000	0%	1,072	0%	1	1
INIT colors	0.0000	0%	0	0%	0.0000	0%	1,088	0%	1	1
PRE_PROCESSING	0.0000	0%	0	0%	0.0000	0%	14,152	1%	1	8
EXECUTE anonymous#1	0.0000	0%	0	0%	0.0000	0%	3,952	0%	1	4
INIT loss	0.0000	0%	0	0%	0.0000	0%	2,536	0%	1	2
INIT maxCap	0.0000	0%	0	0%	0.0000	0%	592	0%	1	1
INIT maxLoad	0.0000	0%	0	0%	0.0000	0%	592	0%	1	1
EXECUTE anonymous#3	0.0000	0%	0	0%	0.0000	0%	272	0%	1	1
EXECUTE anonymous#5	0.0000	0%	0	0%	0.0000	0%	9,760	1%	1	2
INIT where	0.0000	0%	0	0%	0.0000	0%	7,856	0%	1	1
INIT totalLoss	0.0000	0%	0	0%	0.0000	0%	8,976	1%	1	2
INIT load	0.0000	0%	0	0%	0.0000	0%	8,072	0%	1	1
EXTRACT packCt	0.0000	0%	49,152	0%	0.0000	0%	224	0%	1	2
INIT packCt	0.0000	0%	0	0%	0.0000	0%	64	0%	1	1
INIT colorCt	0.0000	0%	0	0%	0.0000	0%	776	0%	1	1
CP Improve Solution	0.0000	0%	0	0%	0.0000	0%	0	0%	1	1
CP Present Solution	0.0000	0%	0	0%	0.0000	0%	0	0%	1	1
POST_PROCESSING	0.0000	0%	0	0%	0.0000	0%	-256	-0%	1	1

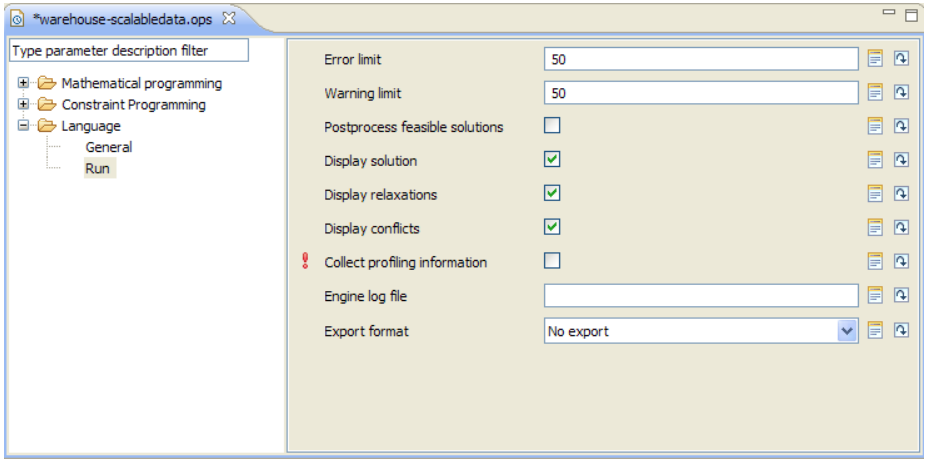
CP models: Profiler (`steelmill` project)

Turning off the Profiler

The profiling functionality is controlled by an OPL language option, which tells the IDE to compute and display the profiling information in the Profiler tab. This option is turned on by default but you can turn it off to save execution time and memory.

To turn off the profiler:

1. Create an `.ops` file as explained in *Adding a settings file* in *Getting Started with the IDE*, and add it to the run configuration (drag and drop).
2. Click **Language>Run**.
3. Uncheck the **Collect profiling information** box, as shown below.



Turning off Collect profiling information

Drawing conclusions

The Profiler can help you improve your model so as to reduce or eliminate slow or memory consuming structures. For example, in `profiler.mod`, we would recommend using the third construct. Likewise, in `scalableWarehouse.mod`, you might want to use the optimization part of the Description tree to fine-tune the CPLEX® search by changing some CPLEX parameter values.

Working with the solution pool

Explains how to access a project solution pool in the IDE and how you can set options and define filters on solution pool generation.

In this section

Purpose and prerequisites
Explains what to do in this tutorial and where to find the files.

The solution pool in the OPL IDE
Explains how the solution pool works, how to use it from the IDE Problem Browser, and how to set options.

Filtering the solution pool
Describes how to write filters by means of IBM® ILOG® Script statements.

Flow control script and solution pool
Describes how to use the IBM® ILOG® Script API to populate the solution pool and use the solutions found.

Purpose and prerequisites

OPL supports the CPLEX® solution pool feature for mixed integer programming (MIP) models. The solution pool is a way of generating and keeping more than one solution to a MIP problem. This allows you to evaluate and explore alternative solutions. For example, you might prefer a solution which also meets a secondary objective over the incumbent.

Prerequisites

The tutorial assumes that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

What you are going to do

Working with various run configurations of the `warehouse` project based on different models, you will:

1. Learn how the IDE supports the solution pool for MIP projects. See *The solution pool in the OPL IDE*. You will:
   a. learn about the default pool of feasible solutions
   b. obtain more non-optimal solutions
   c. set some solution pool options
2. Learn about solution filters and add a range filter. See *Filtering the solution pool*.
3. Use the IBM ILOG Script API, instead of OPL settings. See *Flow control script and solution pool*. You will:
   a. populate the solution pool and use the solutions
   b. add a range filter to control which solutions are kept in the pool

Where to find the files

The tutorial is based on the `warehouse` project available at the following location:

`<OPL_dir>\examples\opl\warehouse`

where `<OPL_dir>` is your installation directory.

The warehouse location problem is described in Warehouse location problems in the *Samples* manual.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

The solution pool in the OPL IDE

Explains how the solution pool works, how to use it from the IDE Problem Browser, and how to set options.

In this section

How the solution pool works
For a mixed-integer problem (MIP) that generates multiple solutions.

Examining pool solutions in the Problem Browser
Explains how to change the default run configuration and analyze the result after a new run.

Obtaining more solutions
Explains how to set CPLEX parameters to obtain additional feasible solutions.

Setting solution pool options
To change the gap tolerance, the number of solutions stored, and the criteria on which solutions are substituted for others when the pool has reached its maximal capacity.

How the solution pool works

The solution pool feature allows you to generate and store multiple solutions to a mixed integer programming (MIP) model. This feature uses an extension of the CPLEX® branch-and-cut algorithm to generate multiple solutions in addition to the optimal solution.

If your model expresses a mixed-integer problem (MIP) and generates intermediate feasible solutions, the OPL IDE displays the pool of solutions obtained after execution at the top of the Problem Browser (see *Solution pool in the Problem Browser (`scalableWarehouse.mod`)*). You can set a number of options (CPLEX parameters) to change how the pool is populated.

There are two ways to collect pool solutions: the default accumulation of feasible solutions over the incumbent and the explicit choice of populating the pool according to default or custom settings.

- In the **default mode**, any feasible solution found during the regular search by the MIP optimization algorithm is listed as a pool solution. The solution pool may contain only the incumbent solution or it may contain more. Sometimes, CPLEX finds a solution that is worse than the current incumbent. In this case, the worse solution may enter the pool although the incumbent callback is not called.
- **Populate** is the mode where, after the MIP search, you explicitly make the choice of finding more solutions by turning on an option that activates a special heuristic (by means of a call to the `populate` method).

Examining pool solutions in the Problem Browser

After you execute a run configuration of a MIP problem, the information on the solutions found is displayed as follows:

- The **Statistics** tab displays the number of solutions in the pool.
- The **Solutions** tab displays the optimal solution. If there are pool solutions in the Problem Browser, by clicking on a pool solution you display the details of that intermediate feasible solution in the Solution tab.
- The drop-down list at the top of the **Problem Browser** numbers each solution other than the final one and identifies it by its objective value.

To view pool solutions:

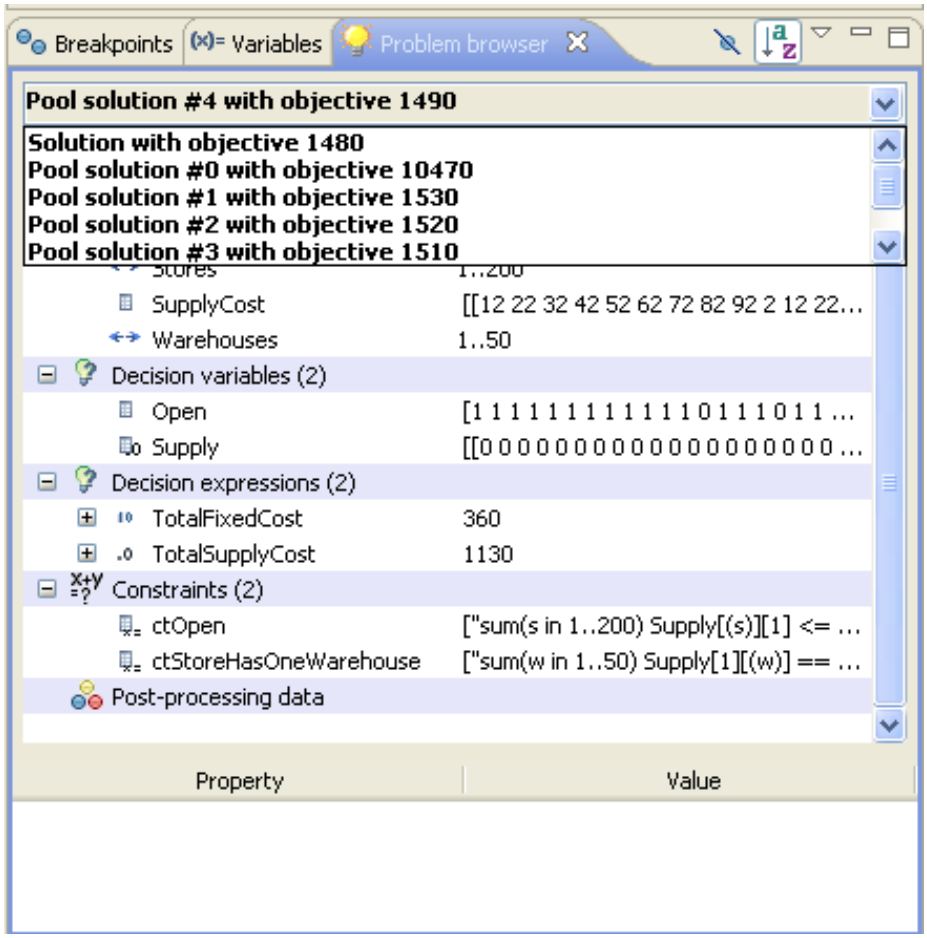
1. In the `warehouse` example, right-click on **Basic Configuration** and set it to default (if it is not the default). This run configuration should contain `warehouse.mod + warehouse.dat`.
2. Right-click on Basic Configuration and select **Run this**.

   You obtain only one solution. In the Solutions tab:

   ```
   // optimal solution with objective 383.
   ```
3. You decide to see what solutions you get with different data.

   Right-click the configuration **Scalable data** and select **Set as Default**.
4. Right-click on **Scalable data** and select **Run this** to execute the run configuration.

   The **Solutions** tab displays the optimal solution. The other feasible solutions are summarized in the solution pool in the Problem Browser.



Solution pool in the Problem Browser (`scalableWarehouse.mod`)

The solutions in the pool are identified by an index number starting from 0 (zero). Generally, the lowest index numbers are for the solutions found earliest. The index numbers may change after re-populating the pool, depending on the strategy in use when the pool reaches its size limit.

When you select a solution from the list in the Problem Browser, the values are displayed in the Solutions tab. If you select pool solution #1, this solution will be displayed in the Solutions tab. If you select pool solution #2, both #1 and #2 will be displayed.

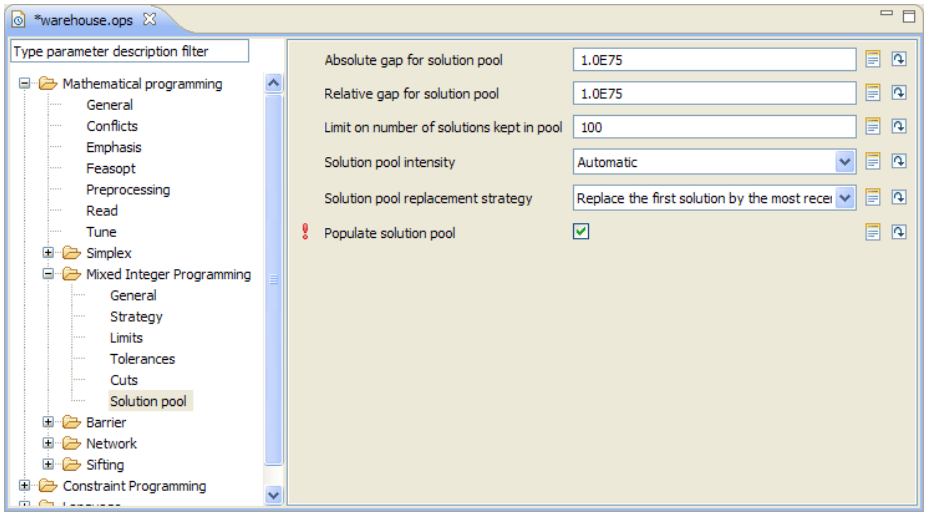
By default, the pool collects the feasible solutions, if any; that is, all the solutions that worked out at the cost of not satisfying the objective by a certain gap. You can use option settings to collect more of these non-optimal solutions.

Obtaining more solutions

If you realize after a MIP optimization that you need additional solutions, you can set the IDE to show more of them by activating the **Populate solution pool** option. The number of solutions you see depends on the values you select for the associated **MIP/Solution pool** options. These options set CPLEX® to spend some extra time after the search to find more feasible solutions by using some specific strategies.

To obtain more solutions:

1. Add a settings file to the **Scalable data** configuration, as explained in *Adding a settings file* in *Getting Started with the IDE*.
2. In the settings editor, select **Mixed Integer Programming>Solution pool**.
3. Check the **Populate solution pool** box.



Turning on the Populate solution pool option

4. Run the configuration again.

   The Problem Browser displays more solutions for you to examine. The number of additional solutions collected relies on a number of associated CPLEX parameters described in *Setting solution pool options*.

Setting solution pool options

When you activate the **Populate solution pool** option, you can use the other options of the **Mixed Integer Programming>Solution pool** page to change the gap tolerance, the number of solutions stored, and the criteria on which solutions are substituted for others when the pool has reached its maximal capacity.

You can also set a limit to the number of solutions generated by the populate phase. In the tree on the left, select the **Mixed Integer Programming>Solution pool**. Then modify the value for **Limit on number of solutions kept in pool**.

As an example:

1. Set the **Basic Configuration** as the default run configuration.
2. Create a settings file and drag it to the **Basic Configuration**.
3. In the settings file, select **Mixed Integer programming>Solution pool** from the tree on the left. Check the **Populate solution pool** box and set the value of **Limit on number of solutions kept in pool** to 3. Then execute the run configuration.

   The Problem Browser displays only 3 feasible solutions.
4. Click the **Reset** button  to restore the **Limit on number of solutions kept in pool** to its default value, then set the **Solution pool intensity** option to **Aggressive** and execute again.

   The execution yields many more solutions.

Note: Other parameters may also have an effect on the solution pool, just as they affect MIP optimization or feasibility generally. See *CPLEX parameters and OPL parameters* in Mathematical programming options for more information.

See also the part dedicated to the solution pool in the CPLEX User's Manual, in particular to understand the difference between collecting solutions and populating the pool.

Filtering the solution pool

Describes how to write filters by means of IBM® ILOG® Script statements.

In this section

Solution pool filters
Explains diversity filters and range filters.

Adding a range filter
Also explains how to add a diversity filter.

Solution pool filters

You can use IBM® Script statements to filter out solutions from the pool. If, for example, you are not satisfied with the solutions yielded by the **Scalable data** configuration with the **MIP>Solution pool** options you set, you may decide to go further and create arrays of variables and coefficients to add range or diversity filters.

There are two kinds of filters for the solution pool.

- Diversity filters allow you to generate solutions that are similar to (or different from) a set of reference values that you specify for a set of binary variables.
- Range filters allow you to generate solutions that obey a new constraint, specified as a linear expression within a range.

See the section about filtering the solution pool in the *CPLEX User's Manual* for details on how these filters work.

Adding a range filter

A range filter adds a constraint over a linear expression. You add a range filter if, for example, you have used the solution pool to get many solutions, but you are interested only in the solutions where the number of open warehouses is between 30 and 43. The `warehouseCplexFilters.mod` model has been written for this purpose as a variation of `scalableWarehouse.mod`. This model contains the `execute` block shown in the code extract below.

Preprocessing script to filter solutions (`warehouseCplexFilters.mod`)

```
// define a range filter applied to solution pools
execute {
    var vars = new Array();
    var coefs = new Array();
    for (var w in Warehouses) {
        vars[w-1] = Open[w];
        coefs[w-1] = 1;
    }
    cplex.addRangeFilter(30, 43, vars, coefs);
}
```

With this data instance:

```
int Fixed        = 30;
int NbWarehouses = 50;
int NbStores     = 100;
```

when the pool is populated, 6 solutions out of the range defined in the filter are removed (26 without the filter, 20 with the filter).

To observe filter scripts:

1. Select the `warehouse` project.
2. Right-click the run configuration name **Filter script** and choose **Set as Default**.
3. Add a settings file, as explained in *Adding a settings file* in *Getting Started with the IDE*, to the **Filter script** configuration.
4. In the settings editor, scroll down the option tree and click **Mixed Integer Programming>Solution pool**.
5. Check the **Populate solution pool** box.
6. Execute the run configuration.

You can also add diversity filters using the method `cplex.addDiversityFilter`. See the *IBM ILOG Script Reference Manual*.

Flow control script and solution pool

Describes how to use the IBM® ILOG® Script API to populate the solution pool and use the solutions found.

In this section

How the flow control script works
Illustrated via the model `solpoolscript.mod`.

Executing the flow control script
Using the `warehouse` project.

How the flow control script works

The model `solpoolscript.mod` is a variation of `scalableWarehouse.mod` and has been written to illustrate how the flow control script works. It contains the `main` flow control script shown in the following code extract.

Flow control script for pool solutions (`solpoolscript.mod`)

```
main {
    thisOplModel.generate();
    cplex.solve();
    cplex.populate();
    var nsolns = cplex.solnPoolNsolns;
    writeln("Nsolns = ",nsolns);
    for (var s=0; s<nsolns; s++) {
      thisOplModel.setPoolSolution(s);
      writeln("---------");
      writeln("sol ", s, " objective = ", cplex.getObjValue(s));
      for (var i in thisOplModel.Warehouses)
        writeln("Open[",i,"] = ", thisOplModel.Open[i]);
    }
```

This flow control script

1. creates and extracts the model, and populates the solution pool after a regular MIP solve

   ```
   thisOplModel.generate();
   cplex.solve();
   cplex.populate();
   ```

2. gets the number of solutions in the solution pool

   ```
   var nsolns = cplex.solnPoolNsolns;
   writeln("Number of solutions found = ",nsolns);
   ```

3. uses the solutions in the model

   ```
   writeln();
   for (var s=0; s<nsolns; s++) {
     thisOplModel.setPoolSolution(s);
   ```

4. gets the objective values for those solutions

   ```
     writeln("solution #", s, ": objective = ", cplex.getObjValue(s));
   ```

5. writes the results to the Scripting Log

   ```
     write("Open = [ ");
     for (var i in thisOplModel.Warehouses)
       write(thisOplModel.Open[i], " ");
     writeln("]");
     writeln("---------");
   ```

Executing the flow control script

To execute the flow control script:

1. Select the `warehouse` project.
2. Right-click the run configuration name **Solution pool script** and choose **Set as Default**.
3. Execute the run configuration.

   Notice that:

   - The result of the script execution is reported in the **Scripting log**. The number of solutions found is written at the top of the Scripting log tab.
   - You cannot see the list of solutions from the Problem Browser.
   - If you add an `.ops` file and check the option **Populate solution pool**, it is not taken into account because script statements prevail over IDE settings.

For more information

– on the solution pool in general: see the CPLEX documentation, in particular the specific chapter in the *CPLEX User's Manual*.

– on the IBM ILOG Script API for the solution pool: see the *IBM ILOG Script Reference Manual*.

Using the performance tuning tool

Describes how to improve the combination of parameters for the CPLEX® part of your model(s).

In this section

Purpose and prerequisites
Introduces the CPLEX performance tuning tool and tells you where to find the files for the tutorial.

How performance tuning works
Describes the CPLEX performance tuning tool.

How to use performance tuning in the IDE
Two ways of using the performance tuning tool, first without fixed settings, then by specifying the settings you want to use.

Purpose and prerequisites

The purpose of the CPLEX® performance tuning tool is to find the best combination of parameters for the CPLEX part of your model(s) that can be set in the OPL settings. The tuning can be done for one or several run configurations so that the search performs best when your application is deployed. Because, in a deployed application, data changes between each invocation of the engine, performance tuning can be executed on a set of problem instances.

Prerequisites

The tutorial assumes that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

Where to find the files

The tutorial is based on the `warehouse` project available at the following location:

`<OPL_dir>\examples\opl\warehouse`

where `<OPL_dir>` is your installation directory.

The warehouse location problem is described in Warehouse location problems in the *Samples* manual.

Note: You will open this OPL project and all projects in these tutorials using the New Example wizard, which allows you to open and work with a copy of the distributed example, leaving the original example in place. If you need a reminder of how to use the New Example wizard, see *Opening distributed examples in the OPL IDE*.

How performance tuning works

The tuning tool works as follows:

- It considers one or more run configurations you specify.

  If the run configurations selected contain different models, the tuning process generates a `.sav` file for each selected configuration and optimizes the parameters for all these `.sav` files considered together.

  Even if the run configurations use the same model, the tuning process always passes different `.sav` files to CPLEX® because the data may be different from one configuration to the next.

- It ignores any change to the default settings specified in `execute` script statements.

- If you want it to define some CPLEX parameters that should remain unchanged, you must pass them as a specific `.ops` file, known as the Fixed Settings file.

- The tuning process executes the models several times in the background, each time with a different set of settings (which may affect the solving time).

- If there is a main block, tuning won't work.

At the end of the solve iterations, the process generates an `.ops` file containing the OPL settings (CPLEX parameters) that it has found to be the best to solve the problem the fastest. This tutorial focuses on the IDE implementation but the performance tuning feature also works from the oplrun command. See the document *oplrun Command Line Interface*.

For a more complete description of the tuning process, please refer to the CPLEX documentation.

How to use performance tuning in the IDE

Two ways of using the performance tuning tool, first without fixed settings, then by specifying the settings you want to use.

In this section

Tuning without fixed settings
Provides a procedure for tuning a model in the OPL IDE.

Results
Discusses the new settings file created at the end of the tuning process.

Tuning with fixed settings
How to tune parameters to get the best performance in a specific context.

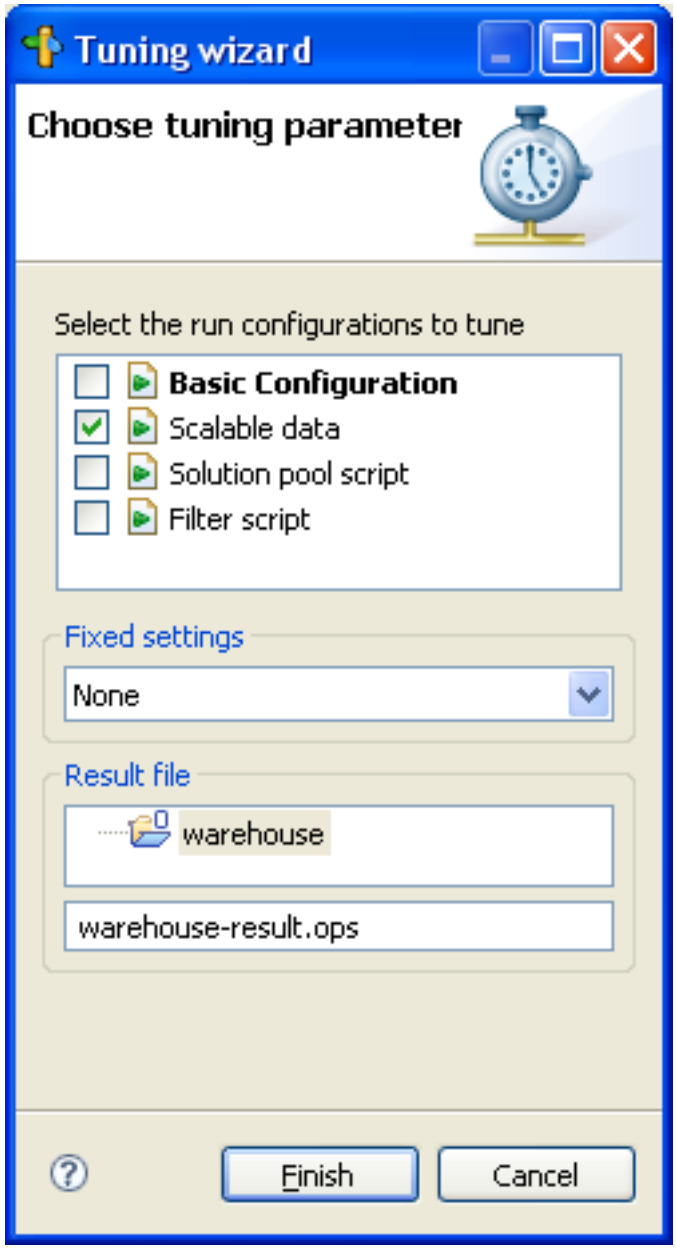
Tuning options
To set the level of information reported by the tuning process.

Temporary tuning files
Temporary files created by the tuning process can be saved to disk to examine later.

Tuning without fixed settings

To observe this:

1. Right-click on the `warehouse` project and select **Tune project**  .
   The tuning wizard opens.



Tuning wizard

By default, all the run configurations defined in the project are selected. Uncheck the ones you don't want to tune. If all run configurations are unchecked, the **Finish** button is not available.

2. In this exercise, uncheck all configurations except **Scalable data**.
3. In the **Result file** area, you need to provide a name for the settings file (`.ops`) output.
4. Click **Finish**. During the tuning process, the Engine Log reports the percentage of progress, the settings worked on at each solve iteration, the execution time, the objective value, the best bound value. "Time limit exceeded" means that the particular tuning iteration took longer than the fastest iteration run so far.

> **Note**: Please note that the progress of the tuning operation is shown in two different locations — in the **Engine Log** tab, and at the right of the IDE's Status Bar. The percentages shown in these two locations may not be identical.

Results

When the tuning process is completed, a new `.ops` file is created and automatically added to the project. If you double-click it, you can see in the editor what values have been modified.

Depending on your hardware configuration, the execution speed may change, hence the results of the tuning process may be different.

If you add the tuning result file to the scalable data run configuration and execute this configuration, notice that the execution time is shorter (almost reduced by half) and that the Engine Log is different.

Tuning with fixed settings

Sometimes, you may want to tune parameters to get the best performance in your own specific context. For example, in your context the best performance may be, instead of the optimal solution, a solution that is within a certain gap from the optimal one. To achieve this, you can pass the performance tuning tool a set of fixed parameters which the tuning process cannot change as they will be used in your model. These settings are taken into account “as is” to calculate the tuning results.

To observe how tuning results differ if you pass a file of fixed settings.

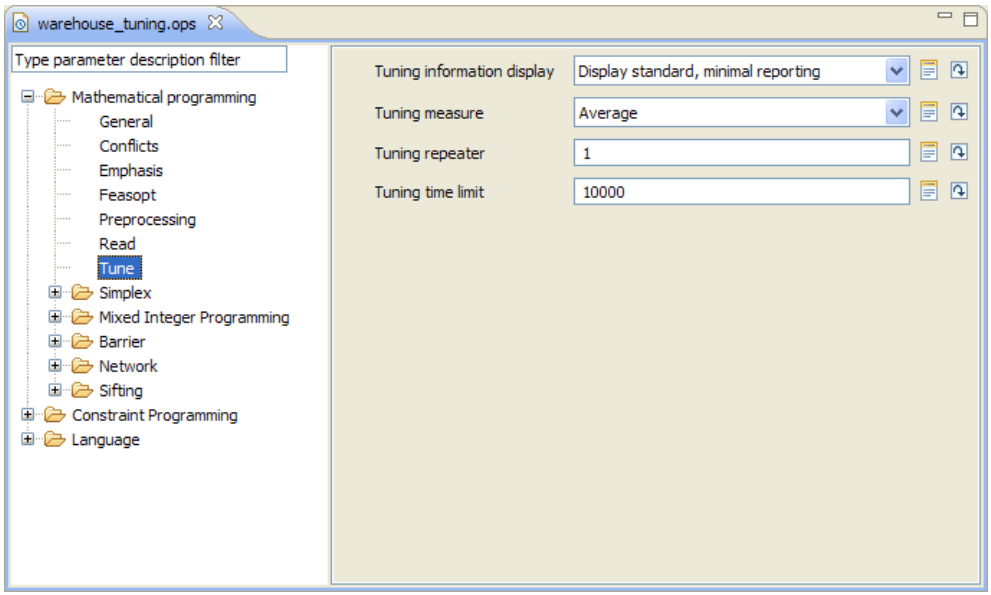
1. Select the `warehouse` project.
2. Add a new settings file, as explained in *Adding a settings file* in *Getting Started with the IDE* to the **Scalable data** configuration.
3. In the **Mixed Integer Programming >Tolerances** page, change the **Relative MIP gap tolerance** to 0.02.
4. Right-click on the `warehouse` project and select **Tune project**  .
5. Make sure that only the run configuration **Scalable data** is selected.
6. In the **Fixed settings** field, select the name of the `.ops` file you have just created with a custom value for Relative MIP gap tolerance.
7. Change the name of the tuning-result file so as not to overwrite the existing one.
8. Click **Finish**.

   The **Relative MIP gap tolerance** value you have set appears in the tuning results.

> **Note**: Please note that the progress of the tuning operation is shown in two different locations — in the **Engine Log** tab, and at the right of the IDE's Status Bar. The percentages shown in these two locations may not be identical.

Tuning options

The **Mathematical programming>Tune** options enable you to set the level of information reported by the tuning process, the measure for evaluating the progress, the number of times the process should be repeated on perturbed versions, and the time limit per run configuration and test set. These options are documented in Mathematical programming options, section *CPLEX parameters and OPL parameters*.



MP Tune options

The possible **Tune** settings are:

- **Tuning information display** — Specifies the level of information reported by the tuning tool as it works. Options are:
  - **No display** (0)
  - **Display standard, minimal reporting** (1 — the default)
  - **Display standard report plus parameter settings** (2)
  - **Display exhaustive report and log** (3)
- **Tuning measure** — Controls the measure for evaluating progress when a suite of models is being tuned. You can specify whether you want the best average performance or the least worst performance across a set of models. Options are:
  - **Average** (mean time — the default)
  - **Minmax** (minmax time)

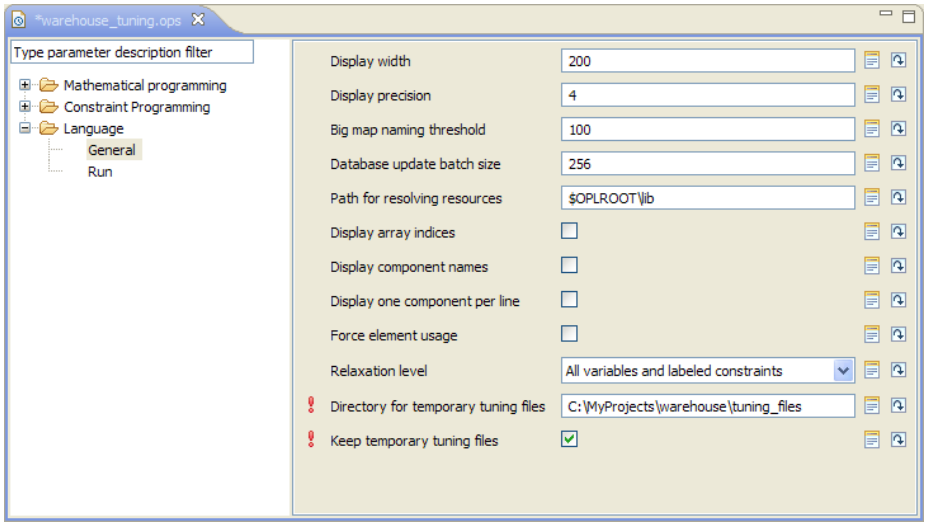
- **Tuning repeater** — Specifies the number of times tuning is to be repeated on reordered versions of a given problem. Options are:
  - Any non-negative integer (Default: **1**)
- **Tuning time limit** — Sets a time limit per model and per test set (that is, suite of models) applicable in tuning. Options are:
  - Any non-negative number (Default: **10000**)

Temporary tuning files

You can save the temporary `.sav` files created by the tuning process to disk to examine their content later.

To save the temporary tuning files:

1. Add a settings file as explained in *Adding a settings file* in *Getting Started with the IDE*.
2. Make sure you add it to the run configurations you want to tune.
3. In the new settings file, check the option **Language > General > Keep temporary tuning files**.
4. Enter a target directory for the temporary files.



Keeping temporary tuning files

Next time you tune run configurations of the project, you will find the corresponding `.sav` file in the directory you have selected. You can reuse this file in IBM ILOG CPLEX.

Index

W