IBM Software Group

Examples of Flow Control in OPL

Nikhila Arkalgud (narkalgu@us.ibm.com)
Scott Rux (srux@us.ibm.com)
IBM ILOG Optimization Level 2 Technical Support
2 October 2013

WebSphere® Support Technical Exchange

This session will be recorded and a replay will be available on IBM.COM sites. When speaking, do not state any confidential information, your name, company name or any information that you do not want shared publicly in the replay. By speaking during this presentation, you assume liability for your comments.

Agenda

- Introduction to Flow Control
  - Basic Structure of Flow Control
  - Templates for Flow Control
- Examples
  - Preprocessing and Postprocessing
  - Performing Iterative solves
  - Data access and modification
  - Column Generation
  - Problem Decomposition
- Important Tools
  - Debugging
  - Memory Management
- Conclusion

Introduction to Flow Control

What is Flow Control ?

Flow control enables control over how models are instantiated and solved in OPL using IBM ILOG Script:

- solve several models with different data in sequence or iteratively
- run multiple “solves” on the same base model, modifying data, constraints, or both, after each solve
- decompose a model into smaller more manageable models, and solve these to arrive at a solution to the original model (model decomposition)

What is IBM ILOG Script ?

- A script is a sequence of commands. These commands could be of various types such as declarations, simple instructions or compound instructions
- **IBM ILOG Script is different from OPL modeling language**
- IBM ILOG Script is an implementation of JavaScript
- Includes extension classes for OPL using which Model elements can be accessed and modified in Flow Control
- All IBM ILOG Script extension classes for OPL start with **Ilo,** (for example IloOplModel, IloCplex, IloCP)

Basic Structure of Flow Control

The main block

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

- To implement Flow Control include a **main** block
- A .mod file can contain at most only **one** main block
- Main block will be **first executed** regardless of where it is placed in the .mod file

The main block using CPLEX Optimizer

```
main {

 thisOplModel.generate();

 if (cplex.solve()) {
  var obj=cplex.getObjValue();
 }

}
```

- **thisOplModel** is an IBM ILOG Script variable referring to the current model instance
- **generate()** is a method used to generate the model instance
- **cplex** is an IBM ILOG Script variable available by default, that refers to the CPLEX Optimizer instance
- **solve()** calls one of CPLEX Optimizer's MP algorithms to solve the model
- **getObjValue()** is a method to access the value of the objective function

The main block using CP Optimizer

```
main {

 thisOplModel.generate();

 if (cp.solve()) {
  var obj=cp.getObjValue();
 }

}
```

- **cp** is an IBM ILOG Script variable available by default, that refers to the CP Optimizer instance
- If the **main** block is in a model file starting with **using cp;**, the **cp** variable is available by default
- If the model file starts with **using cp;**, you'll have to declare the **cplex** variable explicitly if you want to call CPLEX in that same **main** block:

var cplex = new IloCplex();

Templates for Flow Control

Templates for Flow Control

To Construct a Main Block you can apply one of the two methods:

- Start with an OPL project and create a run configuration
- Start with an OPL model and OPL data file

Script template calling a project

```
main {
 var proj = new IloOplProject("../../../../../opl/mulprod");
 var rc = proj.makeRunConfiguration();
 rc.oplModel.generate();
 if (rc.cplex.solve()) {
  writeln("OBJ = ", rc.cplex.getObjValue());
 }
 else {
  writeln("No solution");
 }
 rc.end();
 proj.end();
}
```

Script template calling a model and data

```
main {
 var source = new
IloOplModelSource("../../../../../opl/mulprod/mulprod.mod");
 var def = new IloOplModelDefinition(source);
 var opl = new IloOplModel(def,cplex);
 var data = new
 IloOplDataSource("../../../../../opl/mulprod/mulprod.dat");
 opl.addDataSource(data);
 opl.generate();
 if (cplex.solve()) {
  writeln("OBJ = ", cplex.getObjValue());
 }
 else {
  writeln("No solution");
 }
 opl.end();
 data.end();
 def.end();
 source.end();
}
```

Create source using .mod

Create model definition using source

Create model instance

Add data to model instance

Which template to use?

- The two templates are similar in behavior, both are useful for generating and solving a model
- The Run Configuration approach has the added advantage of being able to use an OPL Settings file (for example, to invoke the tuning tool on a set of models)
- If you are using the current model instance in the flow control, you can use the **thisOplModel** to refer to the current model

Preprocessing and Postprocessing in Flow Control

Preprocessing and Postprocessing

```
execute{
  writeln("Preprocessing block");
  }
minimize
...

subject to {
  ...
}
execute{
  writeln("Postprocessing block");
}
```

- Use preprocessing execute blocks to prepare the data before solving, or to set any parameters
- Preprocessing execute blocks are called by default before each solve
- Use postprocessing execute blocks to control and manipulate the solutions, such as data display or scripting log
- In flow control to execute the postprocessing blocks use:

```
thisOplModel.postProcess();
```

Examples of Preprocessing

- Use Preprocessing execute blocks to prepare the data before calling solve

```
execute{
  for(var w in workers){
   w.salary = w.salaryPerHr * w.hoursWorked;
  }
}
```

- Use Preprocessing execute blocks to set any OPL or CPLEX or CP parameter settings before calling solve

```
execute{
 cplex.numericalemphasis  = true;
}
```

Examples of Postprocessing

- Use Postprocessing execute blocks to display the output data

```
plan Plan[p in Products][t in Periods] =
  <Inside[p,t],Outside[p,t],Inv[p,t]>;

execute{
  writeln("Current Plan = ",Plan);
}
```

- Use Postprocessing execute blocks to display the current solution values

```
execute {
 writeln("Solution Objective value = ", cplex.getObjValue());
}
```

Iterative Solves using Flow Control

What is an Iterative Solve?

- Iterative Solve refers to the approach of incrementally building the mathematical model over several iterations
  - Start with a base mathematical model
  - After each iteration modify the model (either data or definition) to improve the final solution
  - If there is no further improvement, stop the iteration
  - Apart from the Final Optimal Solution, this approach also gives the Optimal Solution after each iteration
  - A powerful yet simple technique for “what-if” analysis
- We will use this technique today on a Multi-Period Production Planning Problem
  - A variation of basic production-planning problem
  - Consider the demand for the products over several periods and allow the company to produce more than the demand in a given period
  - There is an inventory cost associated with storing the additional production

Example for Iterative solves using mulprod

Multi-period production planning problem mulprod.mod

```
minimize
 sum( p in Products , t in Periods )
   (InsideCost[p]*Inside[p][t] +
    OutsideCost[p]*Outside[p][t] +
    InvCost[p]*Inv[p][t]);

subject to {
 forall( r in Resources , t in Periods )
   ctCapacity:
     sum( p in Products )
       Consumption[r][p] * Inside[p][t] <= Capacity[r];

 forall( p in Products , t in Periods )
   ctDemand:
     Inv[p][t-1] + Inside[p][t] + Outside[p][t] ==
     Demand[p][t] + Inv[p][t];

 forall( p in Products )
   ctInventory:
     Inv[p][0] == Inventory[p];
}

tuple plan {
 float inside;
 float outside;
 float inv;
}

plan Plan[p in Products][t in Periods] =
 <Inside[p,t],Outside[p,t],Inv[p,t]>;
```

A data instance from 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 ];
```

Example for Iterative solves using mulprod

```
main {
    thisOplModel.generate();
    var produce = thisOplModel;
    var capFlour = produce.Capacity["flour"];
    var best;
    var curr = Infinity;
    var ofile = new IloOplOutputFile("mulprod_main.txt");
    while ( 1 ) {
        best = curr;
        writeln("Solve with capFlour = ",capFlour);
        if ( cplex.solve() ) {
            curr = cplex.getObjValue();
            writeln();
            writeln("OBJECTIVE: ",curr);
            ofile.writeln("Objective with capFlour = ", capFlour, " is ", curr);
        } else {
            writeln("No solution!");
            break;
        }
        if ( best==curr ) break;
        capFlour++;
        for(var t in thisOplModel.Periods)
            thisOplModel.ctCapacity["flour"][t].UB = capFlour;
    }
    if (best != Infinity) {
        writeln("plan = ",produce.Plan);
    }
    ofile.close();
}
```

Using the current model instance

Obtain the starting Flour Capacity

Solving iteratively using **while** loop

Stopping criteria

Modifying the upper bound (flour capacity)

Data Access in Flow Control

Data Access in Flow Control

- Before modifying the data in the Flow Control, you must get the data elements from the OPL model instance using:

```
thisOplModel.dataElements;
```

- Every time a model data element is modified the OPL model needs to be re-generated
- If generating your OPL model is time consuming, consider directly modifying the generated optimization model
  - You can modify the CPLEX Optimizer matrix directly, without modifying the OPL model
  - For example to modify the flour capacity:

```
thisOplModel.ctCapacity["flour"].UB = flourCapacity;
```

- Only external data can be modified (for example data declared in a .mod file and initialized in a .dat file)
- Scalar data (for example int NumPeriods = 10) cannot be modified in Flow Control

Example for modifying model data using mulprod

```
var def = produce.modelDefinition;
var data = produce.dataElements;
if ( produce!=thisOplModel ) {
 produce.end();
}
produce = new IloOplModel(def,cplex);
capFlour++;
data.Capacity["flour"] = capFlour;
produce.addDataSource(data);
produce.generate();
```

- `var def = produce.modelDefinition;` — Create new model definition
- `var data = produce.dataElements;` — Reference new data elements
- `produce.end();` — End the previous OPL model instance
- `produce = new IloOplModel(def,cplex);` — Create new OPL model instance
- `data.Capacity["flour"] = capFlour;` — Change data (new flour capacity)
- `produce.addDataSource(data);` — Add new data to OPL model instance
- `produce.generate();` — Generate the new OPL model instance

Few tips to remember when performing iterative/multi model solves

- Choose wisely which template will work best for your model design. Both templates can essentially perform the same tasks, but each template has its own advantages
  - If you have multiple data instances to run, with different settings choosing the Template with OPL project with different run configurations would be a better option
  - If you have only one OPL model instance being solved iteratively then a data model template would work better
- When modifying the data elements (or model definition) of a OPL Model instance, remember to add the modified data (or the modified model definition) to the model source and generate the new OPL Model Instance
- If your OPL model generation is time consuming, consider modifying the generated optimization model directly (note: this will not modify the OPL model)
- Remember to end the objects not in use by using the **end()** method
- If using the CPLEX Studio IDE set **mainEndEnabled** to **true**

Column Generation using Flow Control

What is Column Generation?

- Classic example is the Cutting Stock problem:
  - There exist boards of a fixed width (for instance 110 inches)
  - There are also a number of shelves of different lengths (for instance, 20 inches, 45 inches, etc.)
  - It is possible to cut multiple shelves from a single board
  - There is a specified demand the number of shelves
  - Objective: minimize the number of boards used to satisfy demand
- The naïve approach is nearly impossible to write:
  - One variable per possible pattern will only work if there are very few possible patterns
  - In practice, there are too many possible patterns to make this workable
  - Impossible to scale, with even a modest number of possibilities

What is Column Generation?

- Column Generation builds these pattern variables iteratively
- The main approach starts with a rudimentary form of the original problem
- The first half solves the main model, then uses information from this main problem to initialize a sub problem
- The second half solves the sub model, then uses information from this sub problem to create new variables for main problem
- **Requires flow control to manage two different models**

Basic Approach to Column Generation

- Step 1: Solve Master Model, get some basic solution
- Step 2: Use dual information to initialize sub model
- Step 3: Solve Sub Model, get information on new pattern variable that will best improve the objective.
- If the sub model suggests that there is no more improvement, stop
- Step 4: Otherwise, use sub model solution to create new variable, go back to step 1

Basic Approach to Column Generation

- Solve Master Model

```
75    writeln("Solve master.");
76    if ( masterCplex.solve() ) {
77      curr = masterCplex.getObjValue();
78      writeln();
79      writeln("OBJECTIVE: ",curr);
80    }
81    else {
82      writeln("No solution!");
83      masterOpl.end();
84      break;
85    }
```

- Forward information from master model to sub model

```
87    subData.RollWidth = masterOpl.RollWidth;
88    subData.Size = masterOpl.Size;
89    subData.Duals = masterOpl.Duals;
90    for(var i in masterOpl.Items) {
91      subData.Duals[i] = masterOpl.ctFill[i].dual;
92    }
93
94    var subOpl = new IloOplModel(subDef, subCplex);
95    subOpl.addDataSource(subData);
96    subOpl.generate();
97
```

Basic Approach to Column Generation

- Solve Sub Model

```
 98    writeln("Solve sub.");
 99    if ( subCplex.solve() ) {
100      writeln();
101      writeln("OBJECTIVE: ",subCplex.getObjValue());
102      writeln(subOpl.Use.solutionValue);
103    }
```

- If the sub model suggests that there is no more improvement, stop. Otherwise, use sub model solution to create new variable

```
111    if (subCplex.getObjValue() > -RC_EPS) {
112      subOpl.end();
113      masterOpl.end();
114      break;
115    }
116
117
118    // Prepare the next iteration:
119    masterData.Patterns.add(masterData.Patterns.size,1,subOpl.Use.solutionValue);
```

Key Points of Column Generation

- This particular approach relies on the concepts of “dual” in linear programming
- This can be extended to other ways to identify new patterns
  - The key is that the sub problem identifies new variables that will definitely improve the objective, then stops
- Similar approaches work in other industries, like crew scheduling, or vehicle routing
  - Start with a rudimentary formulation, then improve this iteratively
- **Flow control allows you to forward information from one model to another**

Problem Decomposition using Flow Control

What is Problem Decomposition?

- Classic example is a Cost Minimization problem with penalties
  - There is a standard model with demand constraints, and supply constraints, and cost coefficients
  - Some constraints can be violated. This is expressed by adding a “slack” variable on the constraints
  - Each slack variable has a very large cost coefficient
  - As the goal is to minimize cost, this will also minimize the slack variables

- The direct approach is very common, but has a couple of drawbacks
  - If the penalty cost coefficients are too large, can introduce numerical instability and related performance issues
  - If the penalty cost coefficients are too small, constraints may be violated when they should not
  - It may be difficult to prove optimality of particular model

What is Problem Decomposition?

- Problem Decomposition splits (or decomposes) the model into two or more parts
- The first broad model solves to minimize the slacks and/or other very large costs
- Once the large costs are determined, they are fixed in the main model
- The main model can then optimize on the small, fine-grained details, without letting the big variables interfere
- **Requires flow control to manage the different models**

Basic Approach to Problem Decomposition

- Basic Monolithic Model

```
21 float Cost[Products][Cities][Cities] = ...;
22
23 dvar float+ Trans[Products][Cities][Cities];
24 dvar float+ Slack[Products][Cities][Cities];
25
26
27 minimize
28   sum( p in Products , o , d in Cities )
29     Cost[p][o][d]  * Trans[p][o][d]  +
30   sum( p in Products , o , d in Cities )
31     100000000  * Slack[p][o][d];
32
33 subject to {
34   forall( p in Products , o in Cities )
35     ctSupply:
36       sum( d in Cities )
37         Trans[p][o][d] == Supply[p][o];
38   forall( p in Products , d in Cities )
39     ctDemand:
40       sum( o in Cities )
41         Trans[p][o][d] == Demand[p][d];
42    forall( o , d in Cities )
43      ctCapacity:
44        sum( p in Products )
45          (Trans[p][o][d] - Slack[p][o][d]) <= Capacity;
46 }
```

Slack Cost Coefficient

Slack Variables

Basic Approach to Problem Decomposition

- Revised Model for large costs (note objective only has large costs)

```
27 minimize
28   sum( p in Products , o , d in Cities )
29     100000000 * Slack[p][o][d];
30
31 subject to {
```

Only Slack Costs

- Revised Model for small costs (note that slacks are fixed)

```
25 float Slack[Products][Cities][Cities] = ...;
26
27 minimize
28   sum( p in Products , o , d in Cities )
29     Cost[p][o][d] * Trans[p][o][d] ;
30
31 subject to {
32   forall( p in Products , o in Cities )
33     ctSupply:
34       sum( d in Cities )
35         Trans[p][o][d] == Supply[p][o];
36   forall( p in Products , d in Cities )
37     ctDemand:
38       sum( o in Cities )
39         Trans[p][o][d] == Demand[p][d];
40    forall( o , d in Cities )
```

Fixed Slack Values

Only Normal Costs

Basic Approach to Problem Decomposition

- Flow Control to link both models

```
// First Model
var bigModelSource = new IloOplModelSource("first.mod");
var bigModelDef = new IloOplModelDefinition(bigModelSource);
var opl = new IloOplModel(bigModelDef,cplex);
var data = new IloOplDataSource("data.dat");
opl.addDataSource(data);
opl.generate();
if (cplex.solve()) {
   writeln("FIRST OBJ = ", cplex.getObjValue());
   writeln("FIRST Slack = ", opl.Slack.solutionValue);

   // Second Model
   var smallModelSource = new IloOplModelSource("second.mod");
   var smallModelDef = new IloOplModelDefinition(smallModelSource);
   var newOpl = new IloOplModel(smallModelDef,cplex);
   var data = new IloOplDataSource("data.dat");
   newOpl.addDataSource(data);

   var newData = new IloOplDataElements();
   newData.Slack = opl.Slack.solutionValue;
   newOpl.addDataSource(newData);

   newOpl.generate();
   if (cplex.solve()) {
     writeln("SECOND OBJ = ", cplex.getObjValue());
   }
```

Solve First Model

Build Second Model

Use previous data for second model

Key Points of Problem Decomposition

- This particular approach splits a large model into smaller models
- There is substantial overlap between each model
  - This may mean extra overhead if the model design changes
- The benefit is that this allows for a more natural modeling:
  - Usually the largest costs dominate the objective, so solving them and fixing them can give a better sense than solving for everything
  - Avoiding a mix of large and small values, coefficients tends to improve numerical stability, performance, and correctness
- **Flow control allows solve these related models, and forward information between them. This is the basis of decomposition**

Debugging in Flow Control

Debugging in Flow Control: writeln

- The simplest approach is to add **writeln()** statements throughout the flow control blocks, displaying different variables and expressions
- Even though it seems simple, it is often the quickest to set up, and the simplest to evolve

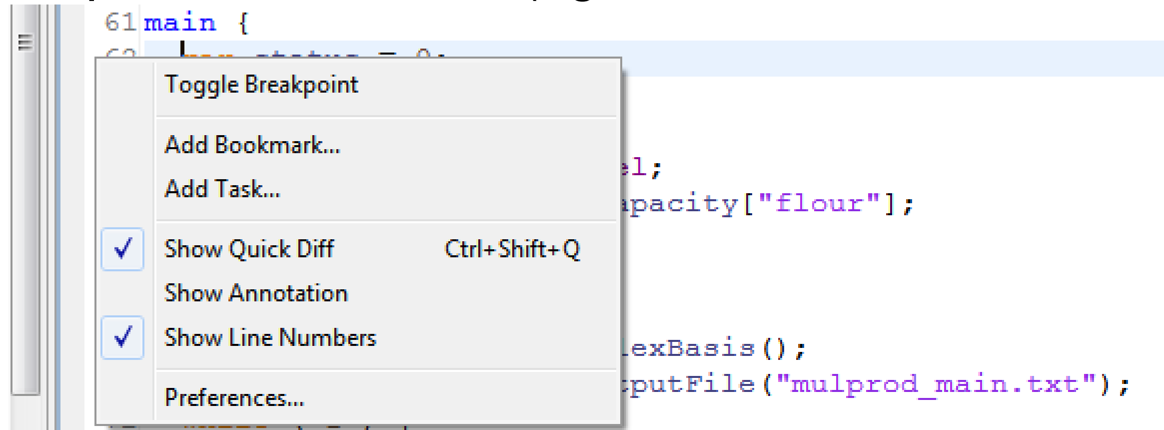
- Be careful! Too many printing statements can clutter the output, and make things less usable
- Consider using an output file (**ofile**) and redirect this extra output to an auxiliary file

Debugging in Flow Control: solve underlying model

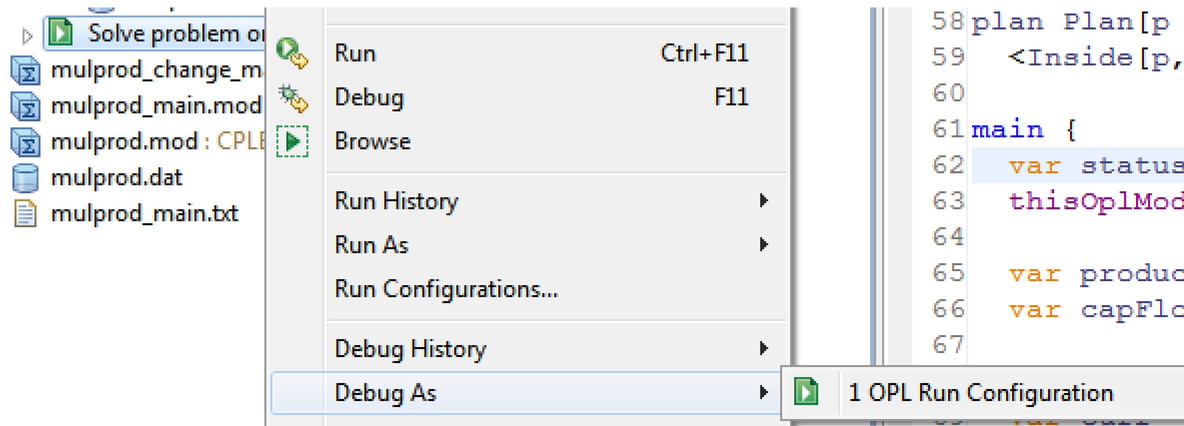
- It can be very tricky to debug master models and sub models, especially if they are built iteratively
- Sometimes, it is useful to create a new OPL project, using the original model
- The OPL data file must be created separately (or manually)
  - This can be made easier by printing out various data with **writeln()**
- This approach is very useful to determine if there are errors in the flow control statements, or with the model itself

Debugging in Flow Control: use the IDE

- Create a breakpoint in the model (right-click on the line numbers)

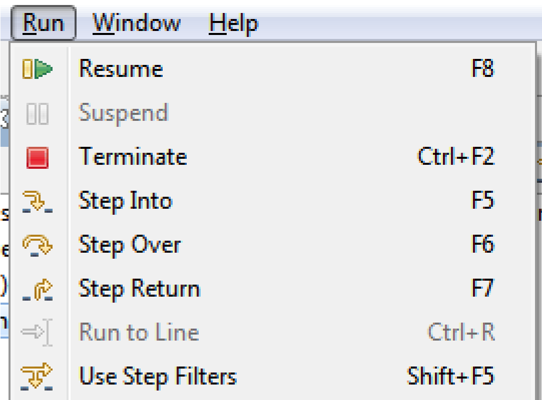


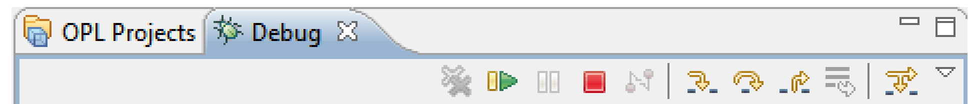
- Start the debugger



Debugging in Flow Control: use the IDE

- Use the Debug options to step through the flow control model line-by-line





- Use the expression window to keep track of variables

Problem browser | (x)= Variables | Breakpoints

Name	Value
status	0
produce	[a IloOplMain.IloOplModel]
capFlour	24
best	452.5
curr	450.25
ofile	[a IloOplOutputFile]
basis	[a IloOplMain.IloOplCplexBasis]
def	[a IloOplMain.IloOplModelDefinition]
data	[a IloOplMain.IloOplDataElements]
[with]	[object Object]

111
112
113
114
115
116
117
118
119
120
121
122

Memory Management in Flow Control

Ending Objects in Flow Control

- Iterative solves can lead to out of memory errors if the objects are not handled correctly

- IBM ILOG Script provides the **end()** methods to end the objects in Flow Control

- The **end()** method is disabled by default in CPLEX Studio IDE, to enable it use:

thisOplModel.settings.mainEndEnabled = true;

- Use caution when calling **end()** method, if you try access the object after it has been deleted could lead to a crash

Ending Objects in Flow Control

Template calling a model and data

```
main {
var source = new
IloOplModelSource("../../../../../opl/mulprod/mulprod.mod
");
var def = new IloOplModelDefinition(source);
var opl = new IloOplModel(def,cplex);
var data = new
IloOplDataSource("../../../../../opl/mulprod/mulprod.dat"
);
opl.addDataSource(data);
opl.generate();

opl.settings.mainEndEnabled = true;

if (cplex.solve()) {
writeln("OBJ = ", cplex.getObjValue());
}
else {
writeln("No solution");
}
opl.end();
data.end();
def.end();
source.end();
}
```

Template calling a project

```
main {
var proj = new
IloOplProject("../../../../../opl/mulprod");
var rc = proj.makeRunConfiguration();
rc.oplModel.generate();

rc.oplModel.settings.mainEndEnabled = true;

if (rc.cplex.solve()) {
writeln("OBJ = ", rc.cplex.getObjValue());
}
else {
writeln("No solution");
}
rc.end();
proj.end();
}
```

Conclusion

Summary

- Flow control is a powerful concept, allowing you to control how different models are solved
- Preprocessing and postprocessing can be used to initialize, prepare, and output data in different ways
- Solving multiple related models in sequences is possible with a simple loop and flow control statements
- The flow control approach also works to decompose larger complicated models into smaller models
- **Flow control allows you to approach more problems than a pure OPL model approach**

Further ILOG Optimization Support Resources

- We are on Facebook!
  - https://www.facebook.com/ILOGOptimizationSupport
- We are on YouTube!
  - https://www.youtube.com/OptimizationSupport
- Don’t forget the IBM Support Portal
  - http://ibm.com/support/

Additional WebSphere Product Resources

- Learn about upcoming WebSphere Support Technical Exchange webcasts, and access previously recorded presentations at: http://www.ibm.com/software/websphere/support/supp_tech.html
- Discover the latest trends in WebSphere Technology and implementation, participate in technically-focused briefings, webcasts and podcasts at: http://www.ibm.com/developerworks/websphere/community/
- Join the Global WebSphere Community: http://www.websphereusergroup.org
- Access key product show-me demos and tutorials by visiting IBM Education Assistant: http://www.ibm.com/software/info/education/assistant
- View a webcast replay with step-by-step instructions for using the Service Request (SR) tool for submitting problems electronically: http://www.ibm.com/software/websphere/support/d2w.html
- Sign up to receive weekly technical My Notifications emails: http://www.ibm.com/software/support/einfo.html

Connect with us!

1. **Get notified on upcoming webcasts**
   Send an e-mail to wsehelp@us.ibm.com with subject line "wste subscribe" to get a list of mailing lists and to subscribe
2. **Tell us what you want to learn**
   Send us suggestions for future topics or improvements about our webcasts to wsehelp@us.ibm.com
3. **Be connected!**
   Connect with us on Facebook
   Connect with us on Twitter

Questions and Answers

This Support Technical Exchange session will be recorded and a replay will be available on IBM.COM sites. When speaking, **do not state any confidential information, your name, company name or any information you do not want shared publicly in the replay**. By speaking in during this presentation, you assume liability for your comments.

Join Industry Solutions Client Success Essentials Community

Easily find important Support resources

- Connect with the Expert:
  - Support Technical Exchanges
  - Ask the Expert Sessions
- Product Support Newsletters
- Blog & Forums
- Training videos, IEA modules
- Event Readiness
- Proactive Services Offerings
- Essential Links to key sites:
  - IBM Support Portal
  - Client Success Portal
  - Fix Central

Welcome to the IBM Industry Solutions Client Success Essentials Community!

Thank you for joining the IBM Industry Solutions Client Success Essentials community. This community brings together users of IBM Industry Solutions software to share, collaborate and connect with each other virtually. In this community, you'll find training videos, upcoming events, blogs, important web links, and more.

View Welcome Video

This community is available only to clients and business partners.
Click here for more information on how to join this community.

Learn and Collaborate:

Click here to view the Segment-Product Directory

Smarter Cities
Products | Blog | Forums

Smarter Commerce
Products | Blog | Forums

Smarter Content (ECM)
Products | Blog | Forums

Join the new Industry Solutions Client Success Essentials Community   Quick start instructions

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON IBM’S CURRENT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO NOR SHALL HAVE THE EFFECT OF CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCT OR SOFTWARE.

Copyright and Trademark Information

IBM, The IBM Logo and IBM.COM are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks and others are available on the web under “Copyright and Trademark Information” located at www.ibm.com/legal/copytrade.shtml.