ILOG OPL Studio

Technical overview

Sofiane Oussedik
soussedik@ilog.fr

Outline

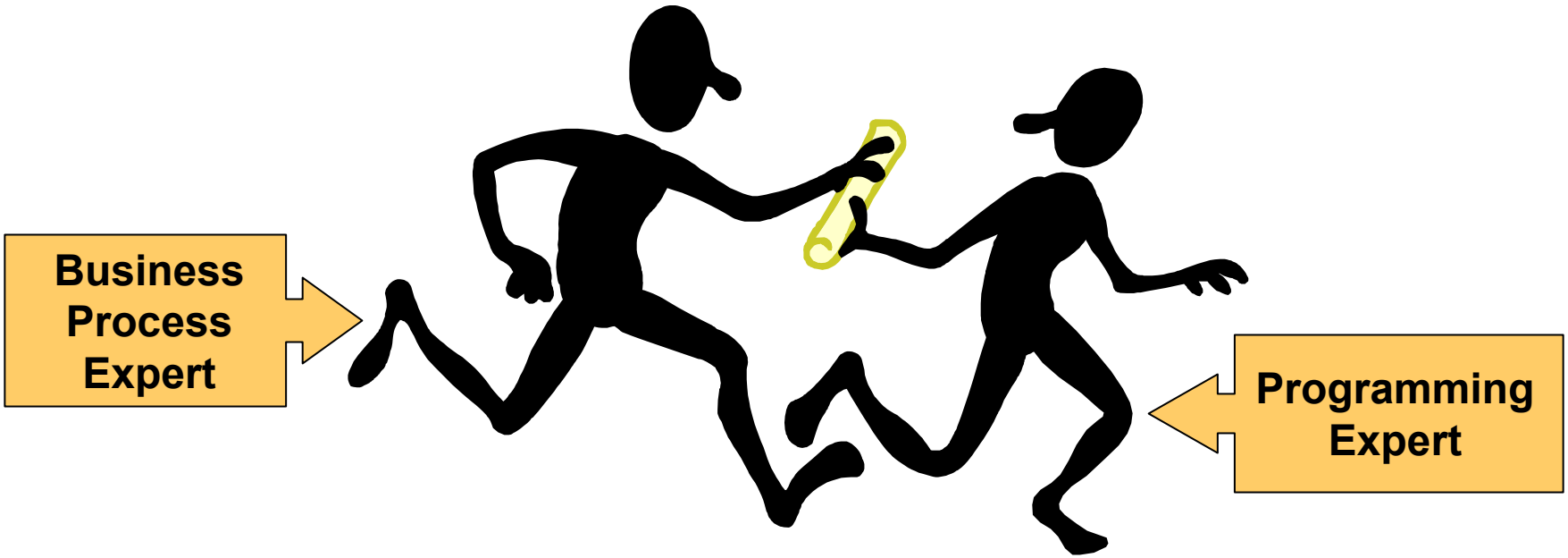
ILOG OPL Studio

- Overview
- Models and Data
- Linear Programming Example
- Constraint Programming Example
- Scheduling Example
- Data Management Features
- Iterative Applications
- OPL Component Libraries

Optimization Development

Changing the rules of business™

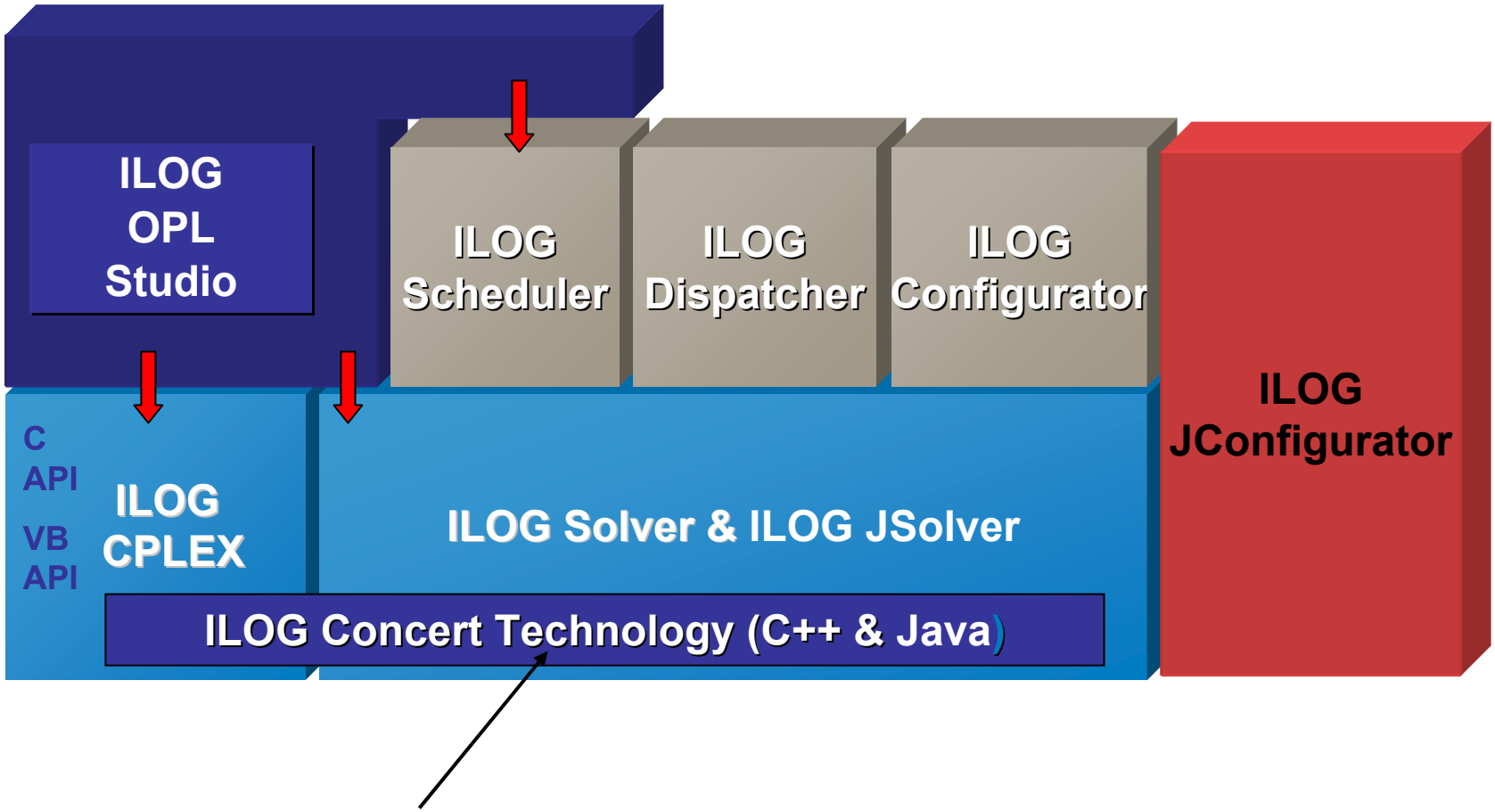
ILOG OPL Studio



ILOG OPL Studio reduces time to market for developing optimization applications

ILOG Optimization Suite

Changing the rules of business™



A Common API for CPLEX and Solver

Inside OPL Studio

Changing the rules of business™

ILOG OPL Studio

- **OPL Optimization Programming Language**
  - A language for representing optimization problems
  - Has advanced types to allow better organization of data
  - Supports constraint programming, linear and integer programming, and scheduling problems
  - Database and Microsoft Excel connectivity
  - OPLScript for iterative solving and hybrid optimization
- **Graphical User Interface for Optimization Problems**
  - Text editor with keyword colors for entering problems and data
  - Visualizations of data and solutions
  - Menus/buttons for controlling optimization
  - Online help for OPL language

OPL Component Libraries

Changing the rules of business™

ILOG OPL Studio

- **API's for embedding OPL models and OPL scripts**
  - **C++**
  - **Microsoft COM / .NET**
    - Visual Basic
    - Visual Basic for Applications (Excel, Access, etc.)
  - **Java**
  - **Web**
    - ASP, JSP
- **Links with ILOG CPLEX, Solver, and Scheduler**

OPL features

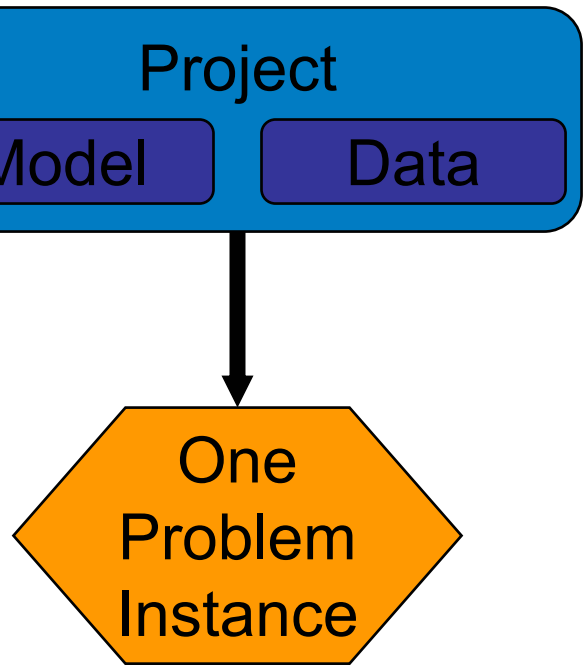
for development

What Are Models?

ILOG
Changing the rules of business™

Linear programming

- A data-independent abstraction of a problem
- OPL lets you write down the mathematical representation of a model separately from the data



A Production Planning Example

Linear programming

- A manufacturer wants to sell a product
- The product can be made either
  - **Inside the factory**
    - Scarce resources are required to build the product
    - There is a cost per unit to manufacture the product
  - **Outside the factory**
    - There is a cost per unit to purchase the product
- All demand must be satisfied
- The goal is to minimize cost

Data Declarations

Linear programming

- Sets of products and resources

```
enum Products ...;
enum Resources ...;
```

- Number of units of each resource needed to produce one unit of each product

```
float+ consumption[Products, Resources] = ...;
```

- Total number of available resources

```
float+ capacity[Resources] = ...;
```

- Number of units in demand for each product

```
float+ demand[Products] = ...;
```

- Cost per unit of inside and outside production

```
float+ insideCost[Products] = ...;
float+ outsideCost[Products]  = ...;
```

Products Could Be Jewelry

Linear programming

- **Products and Resources**

```
Products =  { rings earrings };
Resources = { gold diamonds };
```

- **Consumption**
  - A ring requires 3 units of gold and 1 diamond
  - A set of earrings requires 2 units of gold and 2 diamonds

```
consumption = [ [3, 1], [2, 2] ];
```

- **Capacity (Available units of gold and diamonds)**

```
capacity = [ 130, 180 ];
```

- **Demand (Number of rings and earrings)**

```
demand = [ 100, 150 ];
```

- **Costs (per unit for rings and earrings)**

```
insideCost = [ 250, 200 ];
outsideCost  = [ 260, 270 ];
```

Products Could Be Pasta

Linear programming

- **Products and Resources**

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

- **Consumption**
  - Kluski requires 0.5 units of flour and 0.2 eggs
  - Capellini requires 0.4 units of flour and 0.4 eggs
  - Fettucine requires 0.3 units of flour and 0.6 eggs

```
consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
```

- **Capacity (Available units of flour and eggs)**

```
capacity = [20, 40];
```

- **Demand (Number of each pasta needed)**

```
demand = [100, 200, 300];
```

- **Costs (per unit for each pasta)**

```
insideCost = [0.6, 0.8, 0.3];
outsideCost  = [0.8, 0.9, 0.4];
```

Problem Model Is Identical

ILOG
Changing the rules of business™

Linear programming

```
enum Products ...;
enum Resources ...;

float+ consumption[Products, Resources] = ...;
float+ capacity[Resources] = ...;
float+ demand[Products] = ...;
float+ insideCost[Products] = ...;
float+ outsideCost[Products]  = ...;
```

Data

```
var float+ inside[Products];
var float+ outside[Products];
```

Decision Variables

Problem Model Is Identical (2)

Linear programming

```
minimize
   sum(p in Products)
      (insideCost[p] * inside[p] +
       outsideCost[p]* outside[p] )
```

Objective Function

```
subject to {
  forall(r in Resources)
     sum(p in Products)
       consumption[p, r]*inside[p] <= capacity[r];

  forall(p in Products)
     inside[p] + outside[p] >= demand[p];
};
```

Constraints

Demo of production.prj

Linear programming

- Run production.prj
- Display inside and outside production

Linear and Integer Programming

Linear programming

- OPL can represent linear and mixed integer programming problems
  - Objective functions and constraints are linear
    - E.g., 5 x + 3 y - 4 z
  - Variables can be floating point or integer valued
  - Constraints are inequalities or equalities (≤, ≥, =)
- OPL Studio can solve
  - Linear Programs with any of the CPLEX algorithms
  - Mixed Integer Programs using the CPLEX MIP algorithm
  - Mixed Integer Programs by using constraint programming-based search
- OPL provides access to all CPLEX algorithmic settings

Map Coloring Example

Constraint programming

- Given a list of countries

```
enum Country {Belgium,Denmark,France,Germany,
              Netherlands,Luxembourg};
```

- A set of colors to assign to countries on the map

```
enum Colors {blue,red,yellow,gray};
```

- Want to decide how to assign the colors to the countries so that no two bordering countries have the same color

```
var Colors color[Country];
```

The decision variables are values from a ***set***

Constraint Programming Model

Changing the rules of business™

Constraint programming

```
enum Country {Belgium,Denmark,France,Germany,
              Netherlands,Luxembourg};
enum Colors {blue,red,yellow,gray};
```

Data

```
var Colors color[Country];
```

Decision Variables

```
solve {
```

Find all Solutions

```
     color[France] <> color[Belgium];
     color[France] <> color[Luxembourg];
     color[France] <> color[Germany];
     color[Luxembourg] <> color[Germany];
     color[Luxembourg] <> color[Belgium];
     color[Belgium] <> color[Netherlands];
     color[Belgium] <> color[Germany];
     color[Germany] <> color[Netherlands];
     color[Germany] <> color[Denmark];
};
```

Constraints

Demo of map.mod

Constraint programming

- Run map.mod
- Show all solutions
- Rerun to show the propagation
  - Select Execution > Browse Active Model
  - Right-click on Variables > color and select display domain
  - Select Debug > Stop at Choice Point
  - Step through the model
  - Disable Debug > Stop at Choice Point when finished

Visualizing Solutions

Constraint programming

- Develop application-specific graphical output using the drawing board
  - Lines
  - Polygons
  - Arcs, Circles
  - Text labels
- Graphics can be added to the search procedure
  - Draw an object when variables are fixed to values
  - Graphics are updated automatically during the search procedure

Demo of mapgr.prj

- Run mapgr.prj
- Show all solutions

- Graphics help illustrate how many neighbors constrain each country

Constraint Programming Models

Constraint programming

- OPL allows you to represent problems using constraint programming features
  - Variables can be set to a value from a set
  - A variable can index into an array (`y[x[k]]`)
  - Constraint relations can be strict inequalities (<, >, < >)
  - Logical conditions can be modeled
    ```
    x < 4 => y > 5
    ```
  - Global constraints can be written
    ```
    alldifferent(x)
    ```
  - Constraints can have values (meta constraints)
    ```
    sum (j in myset) (x[j] > 5) = 3
    ```

Constraint programming

- OPL Studio uses ILOG Solver to solve constraint programming problems
  - Default search strategy need not be programmed
  - Users can program their own search strategies
- OPL Studio allows debugging of search strategies
  - Users can visualize the values of their variables
  - Users can step through a search procedure
  - The search tree can be visualized

Warehouse Assignment

- Want to assign S stores to W warehouses. The problem is as follows:
  - The cost of assigning store `s` to warehouse `w` is given by the array element `supplyCost[s,w]`.
  - Each warehouse `w` can have at most `capacity[w]` stores assigned to it.
  - There is a fixed cost `fixed=30` for opening up each warehouse.

Warehouse assignment: MIP


```
var int open[Warehouses] in 0..1;
var int supply[Stores,Warehouses] in 0..1;

minimize
      sum(w in Warehouses) fixed * open[w] +
      sum(w in Warehouses, s in Stores)
            supplyCost[s,w] * supply[s,w]
subject to {
      forall(s in Stores)
            sum(w in  Warehouses) supply[s,w] = 1;
      forall(w in Warehouses, s in Stores)
          supply[s,w] <= open[w];
      forall(w in Warehouses)
            sum(s in Stores) supply[s,w] <= capacity[w];
};
```

Warehouse assignment: CP


```
var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];
var int cost[Stores] in 0..maxCost;

minimize
      sum(s in Stores) cost[s] +
      sum(w in Warehouses) fixed * open[w]
subject to {
   forall(s in Stores)
      cost[s] = supplyCost[s,supplier[s]];
   forall(s in Stores )
      open[supplier[s]] = 1;
   forall(w in Warehouses)
      sum(s in Stores) (supplier[s] = w) <= capacity[w];
};

search {
   forall(s in Stores ordered by decreasing regretdmin(cost[s]))
      tryall(w in Warehouses ordered by increasing supplyCost[s,w])
         supplier[s] = w;
};
```

MIP versus CP formulation

- **Decision variables**
  - The constraint programming formulation has 2S+W decision variables.
  - The mixed integer formulation has SW+W decision variables.
  - The CP formulation has a decision variable over a finite set of values to represent the cost of shipping for store s.
  - The MIP formulation represents the cost of shipping for store s as an implied expression.

```
sum (w in Warehouses) supplyCost[s,w] * supply[s,w]
```

- **Expressions**
  - The CP formulation uses expressions of the form `open[supplier[s]]`, which uses a decision variable to index into another decision variable.
  - The CP formulation uses the expression `(supplier[s] = w)` that evaluates to a 0/1 value.

CP includes search!


```
search {
   forall(s in Stores ordered by decreasing regretdmin(cost[s]))
      tryall(w in Warehouses ordered by increasing supplyCost[s,w])
         supplier[s] = w;
};
```

- `cost[s]` can only take on values from `supplyCost[s,w]` for the set of open warehouses `w`

  `regretdmin` = (second lowest value) - (lowest value)

- Pick the store with the largest regret, then pick the warehouse with the smallest cost
- Then open that warehouse

But Which is BETTER ??

- It depends upon the data
- It depends on the search strategy
- It depends on the combinatorial nature of the problem

- For general applications, you need tools that allow you to try both methodologies!

Scheduling Example (Part 1)

Scheduling

- Have to schedule a set of tasks to build a house

```
enum Tasks { masonry, carpentry, plumbing,
             ceiling, roofing, painting,
             windows, facade, garden, moving };
```

- Tasks require a given amount of time to be completed

```
int duration[Tasks] = [7,3,8,3,1,2,1,2,1,1];
int totalDuration =
             sum(t in Tasks) duration[t];
scheduleHorizon = totalDuration;
```

- Some tasks require other tasks to be completed

```
task[masonry] precedes task[ceiling];
task[carpentry] precedes task[roofing];
task[ceiling] precedes task[painting];
...
```

Scheduling Example (Part 2)

Scheduling

- A group of workers must build the house, but each cannot perform certain tasks

```
enum Workers { joe, jack, jim };
{Workers} cannotperform[Tasks] = #[
    masonry: { jim }, carpentry: { jack },
    plumbing: { joe, jim }, ceiling: { jack },
    roofing: { jack }, painting: { joe  },
    windows: { jack }, facade: { jim },
    garden: {}, moving: { jack } ]#;
```

- Goal is to schedule the tasks and the workers and to minimize the maximum amount of time any worker is on duty

OPL Scheduling Model (Part 1)

Changing the rules of business™

Scheduling

```
enum Tasks { masonry, carpentry, plumbing,
             ceiling, roofing, painting,
             windows, facade, garden, moving };
int duration[Tasks] = [7,3,8,3,1,2,1,2,1,1];
int totalDuration = sum(t in Tasks) duration[t];
scheduleHorizon = totalDuration;
enum Workers { joe, jack, jim };
{Workers} cannotperform[Tasks] = #[
    masonry: { jim }, carpentry: { jack },
    plumbing: { joe, jim }, ceiling: { jack },
    roofing: { jack }, painting: { joe  },
    windows: { jack }, facade: { jim },
    garden: {}, moving: { jack } ]#;
```

Data

```
Activity task[t in Tasks](duration[t]);
var int durationWorkers[Workers] in 0..totalDuration;
Activity attendance[w in Workers](durationWorkers[w]);
UnaryResource worker[Workers];
AlternativeResources s(worker);
```

Decision Variables

OPL Scheduling Model (Part 2)

ILOG
Changing the rules of business™

Scheduling

```
minimize max(w in Workers) durationWorkers[w]
```

Objective Function

```
subject to {
 task[masonry] precedes task[carpentry];
 task[masonry] precedes task[plumbing];
 task[masonry] precedes task[ceiling];  task[carpentry] precedes task[roofing];
 task[ceiling] precedes task[painting]; task[roofing] precedes task[windows];
 task[roofing] precedes task[facade];   task[plumbing] precedes task[facade];
 task[roofing] precedes task[garden];   task[plumbing] precedes task[garden];
 task[windows] precedes task[moving];   task[facade] precedes task[moving];
 task[garden] precedes task[moving];    task[painting] precedes task[moving];

 forall(t in Tasks)  task[t] requires s;

 forall(t in Tasks) forall(w in cannotperform[t])
   not activityHasSelectedResource(task[t],s,worker[w]);

 forall(t in Tasks) forall(w in Workers)
   activityHasSelectedResource(task[t],s,worker[w]) =>
           attendance[w].start <= task[t].start &
           attendance[w].end >= task[t].end;
};
```

Constraints

Demo of house3.mod

Scheduling

- Run house3.mod
- Show the Gantt chart for task activities and worker resources

Scheduling Models

Scheduling

- **OPL supports modeling entities for scheduling problems**
  - **Activities**
    - Have durations
    - Can be breakable
    - Constraints to state that activities precede other activities
  - **Resources**
    - Unary Resources
    - Discrete Resources
    - Reservoirs
    - State Resources with Transition Times
- **Can also add other kinds of constraints using constraint programming**

Structured Data

Data Management

- Given a set of cities

```
enum Cities { MIA, EWR, SFO, BOS };
```

- Origin/destination pairs can be represented as

```
struct Pair { Cities  o; Cities d; };
```

- A set of pairs would be represented as

```
setof(Pair) odpairs =
  {<MIA,EWR>,<MIA,SFO>,<SFO,BOS>,<EWR,SFO>};
```

- The origins can be computed as

```
setof(Cities) origins = { o | <o,d> in odpairs};
```

- A decision variable array over the set of odpairs would be written as:

```
var float+ flow[odpairs];
```

Database connectivity via SQL

Changing the rules of business™

Data Management

- **Reading**

```
struct Precedence {
   string before;      //Task
   string after;       //Task
};
setof(Precedence) precedences from
  DBread(db, "select * from PRECEDENCE");
```

- **Writing**

```
struct Schedule {
  string task;
  int startTime;
  int endTime;
};
setof(Schedule) resultSet = { /* Set of results */ };
DBupdate(db, "insert into Result
              (task, startTime, endTime)
              values (?, ?, ?)")(resultSet);
```

Excel connectivity

Changing the rules of business™

Data Management

- **Reading**

```
setof(int) TimePeriods from
  SheetRead(sheetData,"Time");
float+ avail[TimePeriods] from
  SheetRead(sheetData,"avail");
setof(string) Products from
  SheetRead(sheetData,"Product");
float+ revenue[Products,TimePeriods] from
  SheetRead(sheetData,"Revenue");
```

Time	avail
1	40
2	40
3	32
4	40

Revenue	1	2	3	4
bands	25	26	27	27
coils	30	35	37	39

- **Writing**

```
var float+ Make[Products,TimePeriods];
SheetWrite(sheetResult,"A2:D3")(Make);
```

- **Use named ranges or sheet references**

OPLScript

Iterative Applications

- Scripting language to control execution of OPL models and solution methods
- Applications:
  - Solve a sequence of related models
  - Solve a model with varying data
  - Complex decomposition strategies
  - Hybrid optimization

OPLScript Example

ILOG
Changing the rules of business™

Iterative Applications

```
Model produce("mulprod.mod","mulprod.dat") editMode;
import enum Resources produce.Resources;
int+ capFlour := produce.capacity[flour];

forall(i in 1..4) {
  produce.capacity[flour] := capFlour;
  produce.solve();
  cout << "Flour capacity" << capFlour <<
       "Objective Function: " <<
        produce.objectiveValue() << endl;
  Basis b(produce);
  produce.reset();
  produce.setBasis(b);
  capFlour := capFlour + 1;
}
```

Declare Model

Set data, then Solve!

Output Answer

Try additional capacity, using advanced basis

ILOG
Changing the rules of business™

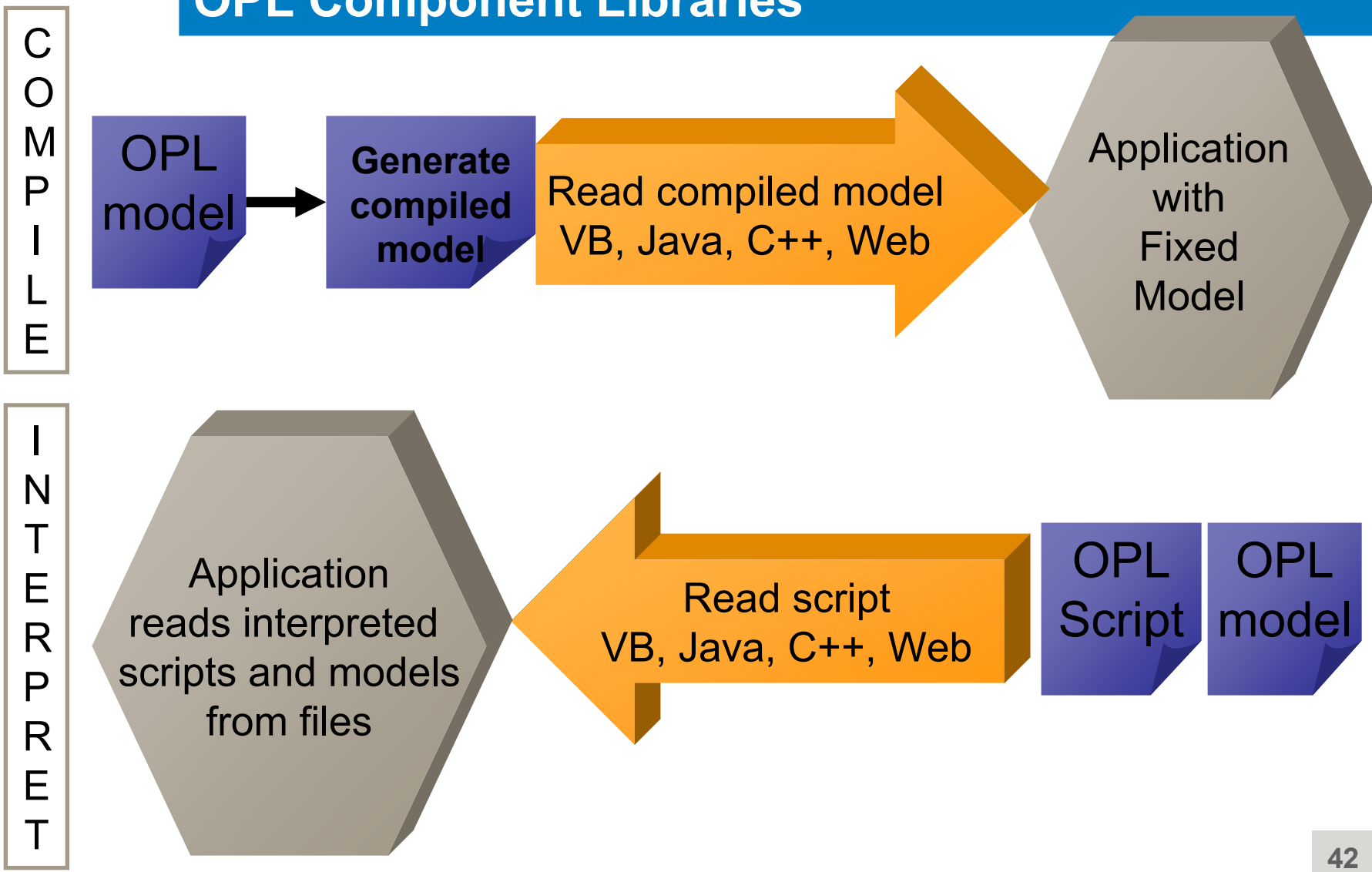
OPL features

for deployment

Deployment Options

Changing the rules of business™

OPL Component Libraries



Excel Demo

Changing the rules of business™

Steel Production Spreadsheet Application

- Open steelt2D.xls
- Show optimization
- Show how if numbers change, answer changes

Visual Basic Example

OPL Component Libraries

- **Initialize solver**

```
Dim solver As COPLsolver
Set solver = New COPLsolver
```

- **Model and data loaded from file or memory**

```
Call solver.loadCompiledModelFileAndDataFile
  ("mulprod.opl", "mulprod.dat", 1)
```

- **Modeling entities are accessed via strings**

```
Dim capacity As IOPLarray
Set capacity = solver.getArray("capacity")
Dim resources As IOPLenum
Set resources = solver.getEnum("Resources")
Dim flour As IOPLenumValue
Set flour = resources.getValue("flour")
Dim capFlour As IOPLint
Set capFlour = capacity.getInt(flour)
```

C++ and Java almost identical

OPL Component Libraries

- Initialize solver
  - C++

```
OPLsolver solver;
```

  - Java

```
OPLsolver solver = new OPLsolver();
```

- Load model and data from file or memory

```
solver.loadCompiledModelFileAndDataFile("mulprod.opl",
                                        "mulprod.dat", 1);
```

- Modeling entities are accessed via strings

```
OPLarray capacity = solver.getArray("capacity");
OPLenum resources = solver.getEnum("Resources");
OPLenumValue flour = resources.getValue("flour");
OPLint capFlour = capacity.getInt(flour);
```

Web JSP Demo

Changing the rules of business™

Steel Production Spreadsheet Application

- Open Staffing Demo
- Show optimization

OPL Studio Key Features

Changing the rules of business™

Summary

- Powerful Modeling Language (OPL) for
  - Constraint Programming
  - Scheduling
  - Linear and Mixed Integer Programming
- Graphical User Interface for
  - Entering Models and Data
  - Organizing Projects
  - Visualizing data and solutions
  - Controlling optimization
- Linked with ILOG optimization tools
  - ILOG Solver, Scheduler and CPLEX
- Database access, Spreadsheet access and Scripting
- OPL Component Libraries accelerate application deployment

Rapid Application Deployment

Changing the rules of business™

Summary

- Develop your model in OPL Studio, maintaining model/data separation
- Refine your algorithm with OPL search strategies
  - Use visualization to enhance understanding
  - Step through search procedures
  - Use iterative or hybrid approaches
- Incorporate model via OPL Component Libraries
  - Use C++, Visual Basic, Java, ASP, JSP
  - Integrate external data
  - Use the answer wherever needed
- Links with ILOG CPLEX, Solver, Scheduler

Get more info

Changing the rules of business™

Web seminar Series: soon on webseminar.ilog.com

- Overview of CPLEX
- Building CPLEX applications using OPL Studio
- Getting started with OPL Studio

Email: **seminarinfo@ilog.com**

Visit: **http://www.ilog.com**