ILOG Concert Technology 1.1

Migration Guide

April 2001

Table of Contents

About This Manual

This manual is intended to help customers translate their existing ILOG Optimization applications into Concert Technology. Each chapter is dedicated to a product and includes mappings between class names, functions, macros, and type definitions in the current versions and their equivalents in Concert versions.

About Concert Technology

ILOG Concert Technology provides a means of modeling and solving optimization problems, enabling you to formulate optimization problems independently of the algorithms that solve the problem. It provides an extendable modeling layer adapted to a variety of ready-to-use algorithms.

The migration paths to Concert Technology are the following:

Before Concert		Concert Technology
ILOG Solver 4.4	to	ILOG Solver 5.1 + ILOG Concert Technology 1.1
ILOG Scheduler 4.4	to	ILOG Scheduler 5.1
ILOG Dispatcher 2.1	to	ILOG Dispatcher 3.1
ILOG Configurator 1.0	to	ILOG Configurator 2.1
ILOG Planner 3.3	to	ILOG CPLEX 7.1 + ILOG Concert Technology 1.1

How this Manual is Organized

Chapter 1, *The Benefits of Migrating to Concert Technology*, provides a brief overview of ILOG Concert Technology.

Chapter 2, *Migrating from Solver to Concert Technology*, explains how to translate Solver 4.4 code into Solver 5.1 and Concert code.

Chapter 3, *Migrating from Scheduler to Concert Technology*, explains how to translate Scheduler 4.4 code into Scheduler 5.1 code that uses Concert Technology.

Chapter 4, *Migrating from Dispatcher to Concert Technology*, explains how to translate Dispatcher 2.1 code into Dispatcher 3.1 code that uses Concert Technology.

Chapter 5, *Migrating from Configurator to Concert Technology*, explains how to translate Configurator 1.0 code into Configurator 2.1 code that uses Concert Technology.

Chapter 6, *Migrating from Planner to Concert Technology*, explains how to translate Planner 3.3 code into Concert Technology that uses ILOG CPLEX 7.1.

What you Need to Know

We recommend that you first read the *ILOG Concert Technology User's Manual*, as that manual introduces many of the concepts discussed in this Migration Guide.

Notation and Naming Conventions

- Throughout this manual, the following typographic conventions apply:
  - Samples of code are written in this `typeface`.
  - Important ideas are emphasized like *this*.
- Throughout this manual, the following naming conventions apply:

  The names of types, classes, and functions defined in the Concert Technology library begin with `Ilo`.

Related Documentation

- *ILOG Concert Technology User's Manual* explains how to use Concert Technology by walking through examples.
- *ILOG Concert Technology Reference Manual* documents C++ classes for representing models as well as the base classes for implementing algorithms.
- Each of the products mentioned in this guide (ILOG Solver, Scheduler, Dispatcher, Configurator, and CPLEX) is accompanied by its own reference and user's manuals.

C H A P T E R

1

The Benefits of Migrating to Concert Technology

ILOG Concert Technology provides a set of lightweight C++ objects for representing optimization problems. It is included as part of ILOG Solver and ILOG CPLEX.

ILOG Scheduler, ILOG Dispatcher, and ILOG Configurator, as extensions of ILOG Solver, are also based on Concert Technology.

Optimization models are separated from the algorithms that are used to find solutions. This allows problems to be decomposed into subproblems so that different algorithms can be applied to separate parts of a large scale problem, and makes it easy to experiment with different algorithmic strategies for the same problem. Concert Technology sets the foundation for the current and future versions of the ILOG Optimization Suite by providing a unifying C++ API for all ILOG optimization products.

Users of ILOG Solver and the other ILOG optimization products that depend on ILOG Solver will see the following benefits:

- Solution Decomposition

  Because models are separate from the search algorithms that solve the models, it is easy to try different algorithms applied to the same model, in order to find the best algorithm for a particular problem. Model/algorithm separation also allows:

  - changing a problem so that a domain of a variable expands
  - relaxing part of an infeasible problem in order to find a solution.

- Multiphase Search

  It is now easier to split a problem into smaller parts, write separate algorithms for each part, and store solutions for the individual parts.

- Cooperation

  The Concert framework facilitates:

  - Automatic model extraction. A constraint programming (CP) or linear programming (LP) model can be automatically extracted and passed to the corresponding solver.
  - Solver cooperation. Other algorithms can cooperate with constraint programming techniques.
  - Nonlinear problem solving. The Numerica domain reduction algorithms can be applied to a nonlinear problem over real variables.
  - Local search: This is now a part of ILOG Solver.

Users of ILOG Planner and ILOG CPLEX will see the following benefits:

- The time and memory required to instantiate a mathematical program is significantly reduced when using ILOG Concert Technology with ILOG CPLEX 7.1 in comparison with using ILOG Planner 3.3.
- A complete C++ API for problem modifications is provided, including adding and deleting variables and constraints, as well as modifications of individual coefficients. This allows, for example, column generation techniques to be easily implemented.
- For hybrid optimization, it is now much easier to extract the linear part of a model and treat the linear part as a single global constraint. Therefore the benefits listed for users of ILOG Solver apply to users who combine ILOG CPLEX and ILOG Solver.

C H A P T E R

2

Migrating from Solver to Concert Technology

Details of the new features and major changes in ILOG Solver 5.1 appear in the *ILOG Solver 5.1 Release Notes*. This chapter aims to show the impact of these changes by comparing Solver 4.4 code and Solver 5.1 code.

Concert Technology provides the following features for Solver users:

- clean separation of model and algorithm
- new data types
- change in the life-cycle of an algorithm
- wrapping of Solver 4.4 code
- re-implementation of search techniques
- separation of goals from demons and constraints.

We recommend that you read the *ILOG Concert Technology User's Manual*, as it introduces many of the concepts discussed in this chapter.

A Brief Introduction to Concert Technology

This section outlines the following major features in Solver 5.1 and Concert Technology:

- clean separation of model and algorithm
- new data types
- change in the life-cycle of an algorithm.

It also summarizes the steps involved in migrating from ILOG Solver 4.4 to ILOG Solver 5.1 with ILOG Concert Technology 1.1.

Clear Distinction Between Model and Algorithm

In Solver 4.4, the model (stating the problem to solve) and the algorithm (solving the problem) were part of the same structure. The objects (variables, constraints, goals) were common to both.

Concert Technology introduces a model which is separate from the Solver algorithm that will solve it. This model is filled with objects whose names are prefixed with `Ilo`. The model is then extracted by the Solver algorithm, which translates it into Solver objects that are solved by the Solver algorithm. The class `IloSolver` in Solver 5.1 is derived from the Concert class `IloAlgorithm`. Solver objects are prefixed with `Ilc`.

This new structure has two consequences:

- The algorithm has a simple solving API but not a model-stating one. For instance, Solver will not be used to create a variable in the modeling layer.
- Changes to the model are not made to the Solver objects; rather, changes are made to the model objects. The instance of `IloSolver` that has extracted the model is then informed of these changes.

New Data Types

We see that there are now two objects attached to each notion that was present in Solver 4.4. For instance, there is an `IloIntVar` and an `IlcIntVar`. `IloIntVar` is the model object, `IlcIntVar` is the Solver object. Both represent an integer variable.

The Life Cycle of an Algorithm

The life cycle of the `IloSolver` class is slightly different from the life cycle of the `IlcManager` class. In fact, `IlcManager` has been split into two classes:

- `IloEnv`, to create model objects
- `IloSolver`, to solve a model.

Thus, in a simple case, we create an instance of `IloEnv` and state the problem with it. Then we create an `IloSolver`, extract the model and solve it. This process is illustrated in the section *Wrapping Existing Solver Code*, on page 18.

How to Translate a Program from Solver 4.4 to Solver 5.1

This section of the guide outlines the major steps in migrating from ILOG Solver 4.4 to ILOG Solver 5.1 with ILOG Concert Technology 1.1. (The following sections provide examples that follow the steps of this outline in greater detail.)

1. Create the environment (an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*).
2. Create the model (an instance of `IloModel`, also documented in the *ILOG Concert Technology Reference Manual*).
3. Translate problem data into appropriate ILOG Concert Technology types. For example, values of `IlcFloat` become `IloNum` ; values of `IlcInt` become `IloInt` , and so forth. Likewise, arrays of `IlcFloatArray` become arrays of `IloNumArray`.
4. Translate constrained variables from your old application to ILOG Concert Technology variables or to ILOG Scheduler 5.1 variables. For example, `IlcIntVar` becomes `IloIntVar` and `IlcFloatVar` becomes `IloNumVar` (documented in the *ILOG Concert Technology Reference Manual*).
5. Wrap the old Solver 4.4 constraints (instances of `IlcConstraint` or its subclasses).
6. Add the wrapped constraints to the model.
7. If there are search goals in your old application, wrap the old Solver 4.4 goals (instances of `IlcGoal` or its subclasses) as shown in the section *Translating Search Code*, on page 23.
8. Create the solver (an instance of `IloSolver`) and its goal. The solve function will automatically extract the model and solve it with the goal given as argument.
9. Translate the search code from your old application according to the steps outlined in the section *Translating Search Code*, on page 23.

Wrapping Existing Solver Code

In this section, examples are provided to show how Solver 4.4 code is wrapped in Concert code.

- The simple Pheasant example is translated from Solver 4.4 to Solver 5.1.
- User-defined constraints are wrapped.
- User-defined goals are wrapped.

Translating the Pheasant Example

This section uses the simple Pheasant example to show how Solver 4.4 code is translated into Solver 5.1 code.

The code of the Pheasant example from Solver 4.4

```
#include <ilsolver/ilcint.h>

int main(){
  IlcManager m(IlcEdit);

  // define two constrained variables ranging from 0 to 100
  IlcIntVar nbRabbits(m, 0, 100), nbPheasants(m, 0, 100);

  // set up the constraint on the number of heads
  m.add( 20 == nbRabbits + nbPheasants );

  // set up the constraint on the number of legs
  m.add( 56 == 4*nbRabbits + 2*nbPheasants );

  // post and propagate constraints
  m.nextSolution();

  // print solution
  m.out() << "Rabbits: "   << nbRabbits   << endl;
  m.out() << "Pheasants: " << nbPheasants << endl;

  // release the resources of the manager
  m.end();

  return 0;
}
```

Code Sample 2.1 *The Pheasant Example from Solver 4.4*

The code of the Pheasant example as it is in Solver 5.1

```
#include <ilsolver/ilosolver.h>

int main(int argc, char **argv) {
  IloEnv env;
  try {
    IloModel model(env);

  // define two constrained variables ranging from 0 to 100
    IloIntVar  nbRabbits(env, 0, 100);
    IloIntVar  nbPheasants(env, 0, 100);

  // set up the constraint on the number of heads
    model.add(nbRabbits + nbPheasants == 20);

  // set up the constraint on the number of legs
    model.add(4*nbRabbits + 2*nbPheasants == 56);


  // create the solver and extract the model
    IloSolver solver(model);

  // solve the problem
    solver.solve();

  // print solution
    solver.out() << "Rabbits: "   << solver.getValue(nbRabbits) << endl;
    solver.out() << "Pheasants: " << solver.getValue(nbPheasants) << endl;
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }

  // release the resources of the env and on the solver
  env.end();
  return 0;
}
```

Code Sample 2.2 *Translation of the Pheasant Example in Solver 5.1*

Wrapping User-Defined Constraints

Use the `ILOCPCONSTRAINTWRAPPER` macro to wrap the existing `IlcConstraint` in Concert model objects. An `IlcConstraint` is wrapped in an `IloConstraint` then extracted, by Solver, from the model and then put back in the `IlcConstraint`.

The following code:

```
ILOCPCONSTRAINTWRAPPER1(IloFreqConstraint, solver, IloNumVarArray, _vars) {
  use(solver, _vars);
  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));
}
```

is expanded into:

```
class IloFreqConstraintI : public IloCPConstraintI {
  ILOCPCONSTRAINTWRAPPERDECL
private:
  IloNumVarArray _vars;
public:
  IloFreqConstraintI(IloEnvI*, const IloNumVarArray&, const char*);
  virtual IloExtractableI* makeClone(IloEnvI*) const;
  virtual void display(ostream  & out) const;
  IlcConstraint extract(const IloSolver&) const;
};

ILOCPCONSTRAINTWRAPPERIMPL(IloFreqConstraintI);

IloFreqConstraintI::IloFreqConstraintI(IloEnvI* env,
                                       const IloNumVarArray&   T_vars,
                                       const char* name) :
  IloCPConstraintI (env, name), _vars ((IloNumVarArray&)T_vars) {}

IloExtractableI* IloFreqConstraintI::makeClone(IloEnvI* env) const {
  IloNumVarArray targ1 = IloGetClone(env, _vars);
  return new (env) IloFreqConstraintI(env,
                                      (const IloNumVarArray &)targ1,
                                      (const char*)0);
}


void IloFreqConstraintI::display(ostream& out) const {
  out << "IloFreqConstraintI" << " (";
  if (getName()) out << getName();
  else out << getId(); out << ")" << endl;
  out << "  " << "_vars" << " " << _vars << endl;
}
```

```
IloConstraint  IloFreqConstraint(IloEnv env,
                                 IloNumVarArray _vars,
                                 const char* name=0) {
  return new (env) IloFreqConstraintI(env.getImpl(), _vars, name);
}

IlcConstraint IloFreqConstraintI::extract(const IloSolver& solver) const {
  use(solver, _vars);
  return IlcFreqConstraint(solver, solver.getIntVarArray(_vars));
}
```

Wrapping User-Defined Goals

In Solver 5.1, given an instance of the class `IloGoal`, the member function `IloSolver::extract` will create an instance of `IlcGoal` for use internally by Solver. In that context, an instance of `IlcGoal` in an existing Solver 4.4 application needs to be wrapped. You must define the virtual member function `IloGoal::extract` so that it accepts an instance of `IloSolver` and returns and instance of `IlcGoal`.

The following steps show how the goal `IlcInstantiate(IlcIntVar)` is wrapped in a Concert model object:

1. Create a subclass of the class `IloGoalI`.

```
class IloIntInstantiateI : public IloGoalI {
  IloNumVarI* _var;
public:
  IloIntInstantiateI(IloEnvI*, IloNumVarI*);
  virtual IlcGoal extract(const IloSolver) const;
  virtual IloGoalI* makeClone(IloEnvI* env) const;
  virtual void display(ILOSTD(ostream&)) const;
};
```

2. Write the constructor.

```
IloIntInstantiateI::IloIntInstantiateI(IloEnvI* e,
       IloNumVarI* v) :
  IloGoalI(e), _var(v) {}
```

3. Write the member functions `makeClone` and `display`.

```
IloGoalI* IloIntInstantiateI::makeClone(IloEnvI* env) const {
  return new(env) IloIntInstantiateI(env, (IloNumVarI*)_var->makeClone(env));
}

void IloIntInstantiateI::display(ostream& str) const {
  str << "IloIntInstantiateI(" << _var << ")";
}
```

4. Write the critical member function `extract`.

```
IlcGoal IloIntInstantiateI::extract(const IloSolver solver) const {
  IlcGoal goal;

// Check if the variable is integer
  if (solver.isInteger(_var)) {

// Get the corresponding algorithmic objects
    IlcIntVar svar = solver.getIntVar(_var);

// Return the IlcGoal
    goal = IlcInstantiate(solver, svar);
  }
  else {
    char* msg = (char*)"The variable given to IloInstantiate should be typed integer";
    throw IloSolver::VariableShouldBeInteger(msg);
  }
  return goal;
}
```

5. Write the function that builds the instance of `IloGoal`.

```
IloGoal IloInstantiate(const IloEnv env, const IloNumVar v) {
  return new (env) IloIntInstantiateI(env.getImpl(), v.getImpl());
}
```

Translating Search Code

In this section, we illustrate how simple search techniques are re-implemented using the model-algorithm paradigm introduced in Solver 5.1 with Concert Technology.

Basic Search Techniques

We suppose we have a common program that creates an instance of the class `IloEnv`, creates with it an instance of the class `IloModel` named `model`, and fills it with variables and constraints. We can now write small code samples that implement basic search techniques.

To Find One solution

Create a solver with the model and ask it to solve the problem. Here we ask for the first solution.

Solver 4.4 code

```
IlcManager m(IlcEdit);
// load problem

m.add(myIlcGoal);

if (m.nextSolution()) {
   solver.out() << "Found one solution" << endl;
}
else {
   solver.out() << "No Solutions" << endl;
}
```

Solver 5.1 code

```
IloEnv env;
IloModel model(env);
// load problem

IloSolver solver(model);
if (solver.solve(myIloGoal)) {
   solver.out() << "Found one solution" << endl;
}
else {
   solver.out() << "No Solutions" << endl;
}
```

To Find the Optimal Solution

Solver 4.4 code

```
IlcManager m(IlcEdit);
// load problem

m.add(myIlcGoal);

IlcBool result = IlcFalse;
while (m.nextSolution()) {
  result = IlcTrue;
}
if (result) {
  m.restart();
  m.nextSolution();
  m.out() << "Found an optimal solution" << endl;
}
else {
   m.out() << "No Solutions" << endl;
}
```

Solver 5.1 code with an Objective

To find the optimal solution, we use an instance of the class `IloObjective` to specify an objective to minimize or to maximize. The function `IloMinimize` returns an instance of the class `IloObjective`. In the following code sample, we assume `myIloGoal` is a wrapper around `myIlcGoal`. Given an objective variable `obj`, we write the following code:

```
IloEnv env;
IloModel model(env);
// load problem

model.add(IloMinimize(obj));
IloSolver solver(model);
if (solver.solve(myIloGoal)) {
   solver.out() << "Found one solution" << endl;
}
else {
   solver.out() << "No Solutions" << endl;
}
```

Solver 5.1 code with Search

You can do additional things (e.g. print the objective value) at intermediate solutions.

```
IloEnv env;
IloModel model(env);
// load problem

model.add(IloMinimize(obj));
IloSolver solver(model);
IloBool result = IloFalse;
solver.startNewSearch(myIloGoal);
while (solver.next()) {
env.out() << "Objective value = " << solver.getValue(obj) << endl;
  result = IloTrue;
}
if (result) {
   solver.restartSearch();
   solver.next();
   solver.out() << "Found an optimal solution" << endl;
}
else {
   solver.out() << "No Solutions" << endl;
}
solver.endSearch();
```

To Iterate Over Solutions

We can also use `IloSolver::startNewSearch` and `IloSolver::next` to replace code that contains the call to:

```
IlcManager::nextSolution().
```

Solver 4.4 code

```
IlcManager m(IlcEdit);
// load problem

m.add(myIlcGoal);

while (m.nextSolution()) {
   solver.out() << "Found one solution" << endl;
}
```

Solver 5.1 code

Given an `IloGoal` named `myilogoal` that wraps the `IlcGoal` used in the program, we can write:

```
IloEnv env;
IloModel model(env);
// load problem

IloSolver solver(model);
startNewSearch solver.newSearch(myIloGoal);
while (solver.next()) {
   solver.out() << "Found one solution" << endl;
}
```

```
solve.endSearch;
```

Testing a List of Constraints

Here is a search technique that is useful in an over-constrained problem. In this case, you have a list of constraints and you want to find a sublist where simple propagation does not fail.

The following code sample will iterate over this list and store accepted constraints in a second model named `store` to be used later.

```
IloEnv env;
IloModel model(env);

// create variables and basic objects

ListOfConstraints myList;
ListOfConstraintIterator iter(myList);

IloModel store(env);

IloSolver solver(model);
while (iter.ok()) {
  IloConstraint c = *iter;
  if (solver.propagate(c, IloSynchronizeAndContinue)) {
    store.add(c);
  }
  ++iter;
}

// At this point ‘store’ contains a list of constraints that do not fail upon
// simple propagation of the constraints. We can use a second solver to find a
// feasible solution.

IloSolver solver2(store);
if (solver2.solve()) {
   solver2.out() << "Found one optimal solution" << endl;
}
else {
   solver2.out() << "No Solutions" << endl;
}

env.end();
```

Multi-Phase Search with Restart

Here we illustrate how a first search will help restrict the problem before launching a full search. We assume we have an objective variable `obj` and a model `model`.

Solver 4.4 code

```
IlcManager m(IlcEdit);
// load problem

m.add(myIlcGoal1);

m.nextSolution());
IlcInt objValue = obj.getValue();

m.restart();
m.add(obj <= objValue);
m.setObjMin(obj);
m.remove(myIlcGoal1);
m.add(myIlcGoal2);

while (m.nextSolution())
  result = IlcTrue;
m.restart();
m.nextSolution();
m.out() << "Found an optimal solution" << endl;
```

Solver 5.1 code

```
IloEnv env;
IloModel model(env);
// load problem

IloSolver solver(model);

// We find a first solution
solver.solve(myIloGoal1);

// We update the model
model.add(obj <= solver.getValue(obj));
model.add(IloMinimize(obj));

// we find the optimal solution
solver.solve(myIloGoal2);
solver.out() << "Found an optimal solution" << endl;
```

Multi-Phase Search with Continue

Here we illustrate how to rewrite code that searches for one partial solution by means of a given goal, commits its preliminary results, adds a constraint, and searches for a complete solution by means of another goal.

Solver 4.4 code

```
IlcManager m(IlcEdit);
// load problem

m.add(myIlcGoal1);

m.nextSolution());
m.commit();
m.add(myIlcConstraint);
m.setObjMin(obj);
m.remove(myIlcGoal1);
m.add(myIlcGoal2);

while (m.nextSolution())
  result = IlcTrue;
m.restart();
m.nextSolution();
m.out() << "Found an optimal solution" << endl;
```

Solver 5.1 code using a solve in the first search

```
IloEnv env;
IloModel model(env);
// load problem

IloSolver solver(model);

// We find a first solution
solver.solve(myIloGoal1);

// We update the model
model.add(myIloConstraint);
model.add(IloMinimize(obj));

// we find the optimal solution
solver.solve(myIloGoal2, IloSynchronizeAndContinue);
solver.out() << "Found an optimal solution" << endl;
```

Separation of Goals from Demons and Constraints

In Solver 4.4, `IlcDemonI` derives from `IlcGoalI` and `IlcConstraintI` derives from `IlcGoalI`. The advantage is that you can always pass an instance of `IlcConstraintI` when an instance of `IlcGoalI` is expected. However, this may also lead to some confusion.

In Solver 5.1, `IlcConstraintI` derives from `IlcDemonI` and `IlcGoalI` is separated.

The advantage of this method is that it is now possible to have a clear separation between the notion of goal and the notion of constraint. A goal is used to define and control the search procedure and a constraint is used to perform some domain reductions of variables.

For compatibility with Solver 4.4, it remains possible to pass a goal when a constraint is expected. (The converse operation is also possible.)

Solver 4.4 code – part of the synopsis of `IlcGoalI`, `IlcDemonI` and `IlcConstraintI`:

```
class IlcGoalI {
  IlcGoalI(IlcManagerI*);
  IlcGoalI(IlcManager);
  virtual IlcGoal execute()=0;
  virtual void executeDemon();
};

class IlcDemonI: public IlcGoalI {
public:
  IlcDemonI(IlcManagerI* manager);
  IlcDemonI(IlcManager manager);
  virtual IlcGoal execute();
  virtual void executeDemon()=0;
};

class IlcConstraintI: public IlcGoalI {
  IlcConstraintI(IlcManagerI*);
  IlcConstraintI(IlcManager);
  virtual void propagate ()=0;
  virtual void post ()=0;
  virtual void metaPost(IlcGoalI*);
};
```

Solver 5.1 code – part of the synopsis of `IlcGoalI`, `IlcDemonI` and `IlcConstraintI`:

```
class IlcGoalI {
  IlcGoalI(IlcManagerI*);
  IlcGoalI(IlcManager);
  virtual IlcGoal execute()=0;
  virtual void propagate();
  virtual void metaPostDemon(IlcDemonI*);
};

class IlcDemonI {
public:
  IlcDemonI(IlcManagerI* manager, IlcConstraintI* ct=0);
  IlcDemonI(IlcManager manager, IlcConstraintI* ct=0);
  virtual void propagate()=0; // previously executeDemon
  virtual void metaPostDemon(IlcDemonI*);
};

class IlcConstraintI : public IlcDemonI  {
public:
  IlcConstraintI(IlcManagerI*);
  IlcConstraintI(IlcManager);
  virtual IlcGoal execute(); // for compatibility only.
  virtual void propagate ()=0;
  virtual void post ()=0;
  virtual void metaPostDemon(IlcDemonI*);
  virtual void metaPost(IlcGoalI*); // for compatibility only
};
```

There are four main changes in Solver 5.1:

1. The function `executeDemon` has been renamed as `propagate`. Therefore, if you decide to subclass `IlcDemonI` yourself (and not use the macro), you should overload the member function `propagate` instead of the member function `executeDemon`.

   For compatibility with Solver 4.4, the `ILCDEMON` macros have been changed with regard to this modification. In Solver 5.1 these macros return an instance of `IlcDemon` instead of an instance of `IlcGoal` and should no longer be used (see point 3).

2. In Solver 5.1 a demon should always be associated with a constraint, and this constraint must be passed as an argument of the constructor of `IlcDemonI`. Moreover, instead of writing code in the `executeDemon` function of an `IlcDemonI`, you should just call a member function of the associated constraint.

3. New macros, `ILCCTDEMON` and `ILCCTPUSHDEMON`, have been introduced in order to facilitate the creation of `IlcDemon`. They replace the `ILCDEMON` macros.

   `ILCCTDEMON0(name,IlcCtClass,IlcFnName)` will define a demon named `name` associated with a constraint which is an instance of the class `IlcCtClass`. The `propagate` member function of this demon will contain only the call to function `IlcFnName` of the constraint `ct->IlcFnName`.

This is a possible definition:

```
ILCCTDEMON(MyDemon,MyConstraint,fnOfCt);
```

The demon will be defined by:

```
MyDemon(ct);
```

where `ct` must be an instance of `MyConstraint`.

`ILCCTPUSHDEMON0(name, IlcCtClass)` will define a demon named `name` associated with a constraint which is an instance of the class `IlcCtClass`. The `propagate` member function of the demon will contain only the call to the `push` member function of the constraint.

4. New `whenValue`, `whenRange`, `whenDomain` functions have been introduced.

```
IlcIntExp::whenValue(const IlcDemon demon),
IlcIntExp::whenRange(const IlcDemon demon),
IlcIntExp::whenDomain(const IlcDemon demon),
IlcFloatExp::whenValue(const IlcDemon demon),
IlcFloatExp::whenRange(const IlcDemon demon),
IlcIntSetVar::whenDomain(const IlcDemon demon),
IlcIntSetVar::whenValue(const IlcDemon demon).
```

Except for co-routining, these functions should be used instead of the equivalent ones that take an instance of `IlcGoal` as a parameter.

For instance, consider the following Solver 4.4 code:

```
class IlcLessThanCtI: public IlcConstraintI {
// x < y
IlcIntVar _x;
IlcIntVar _y;
public:
IlcLessThanCtI(IlcIntVar x, IlcIntVar y):
             IlcConstraintI(x.getManager()), _x(x), _y(y){}
IlcBool isViolated() const;
IlcConstraintI* makeOpposite() const;
void post();
void propagate();
void metaPost(IlcGoalI*);
void xPropag();
void yPropag();
};

IlcBool IlcLessThanCtI::isViolated() const {
return x.getMin() > y.getMax();
}

IlcConstraintI* IlcLessThanCtI::makeOpposite() const {
return new (getManager()) IlcGreaterThanOrEqualToCt(getManager(),_x,_y);
}

ILCDEMON2(xDemon,IlcIntVar,x,IlcIntVar y){
y.setMin(x.getMin()+1);
}

ILCDEMON2(yDemon,IlcIntVar,x,IlcIntVar,y){
x.setMax(y.getMax()-1);
}

void IlcLessThanCtI::post(){
x.whenRange(xDemon(x,y));
y.whenRange(yDemon(x,y));
}

void IlcLessThanCtI::propagate(){
y.setMin(x.getMin() +1);
x.setMax(y.getMax() -1);
}

void IlcLessThanCtI::metaPost(IlcGoalI* ct){
x.whenRange(ct);
y.whenRange(ct);
}
```

In Solver 5.1 this code is translated as follows:

```
class IlcLessThanCtI: public IlcConstraintI {
// x < y
IlcIntVar _x;
IlcIntVar _y;
public:
IlcLessThanCtI(IlcIntVar x, IlcIntVar y):
             IlcConstraintI(x.getManager()), _x(x), _y(y){}
IlcBool isViolated() const;
IlcConstraintI* makeOpposite() const;
void post();
void propagate();
void metaPostDemon(IlcDemonI*); // WARNING NEW FUNCTION
};

IlcBool IlcLessThanCtI::isViolated() const {
return x.getMin() > y.getMax();
}

IlcConstraintI* IlcLessThanCtI::makeOpposite() const {
return new (getManager()) IlcGreaterThanOrEqualToCt(getManager(),_x,_y);
}

void IlcLessThanCtI::xPropag(){
y.setMin(x.getMin()+1);
}

ILCCTDEMON0(xDemon,IlcLessThanCtI,xPropag);

void IlcLessThanCtI::yPropag(){
x.setMax(y.getMax()-1);
}

ILCCTDEMON0(yDemon,IlcLessThanCtI,yPropag);

void IlcLessThanCtI::post(){
x.whenRange(xDemon(getManager(),this));
y.whenRange(yDemon(getManager(),this));
}

void IlcLessThanCtI::propagate(){
y.setMin(x.getMin() +1);
x.setMax(y.getMax() -1);
}

void IlcLessThanCtI::metaPost(IlcDemonI* ct){
x.whenRange(ct);
y.whenRange(ct);
}
```

Note that in Solver 4.4 the macro:

```
ILCDEMON2(xDemon,IlcIntVar,x,IlcIntVar y){
y.setMin(x.getMin()+1);
}
```

generates a code similar to the following lines:

```
class xDemonI : public IlcDemonI {
IlcIntVar x;
IlcIntVar y;
public:
      xDemonI(IlcManagerI* manager, IlcIntVar xx, IlcIntVar yy):
IlcDemonI(manager), x(xx), y(yy){}
      void executeDemon();
};

void xDemonI::executeDemon(){
y.setMin(x.getMin() +1);
}

IlcGoal xDemon(IlcManager m,IlcIntVar x, IlcIntVar y){
return new (m.getHeap()) xDemonI(m.getImpl(),x,y);
}
```

whereas in Solver 5.1 the macro:

```
ILCDEMON0(xDemon,IlcLessThanCtI,xPropag)
```

generates a code similar to the following lines:

```
class xDemonI : public IlcDemonI {
public:
      xDemonI(IlcManagerI* manager, IlcLessThanCtI* ctI):
IlcDemonI(manager,ct){}
      void propagate();
};

void xDemonI::propagate(){
((IlcLessThanCtI*)getConstraintI())->xPropag();
}

IlcDemon xDemon(IlcManager m,IlcLessThanCtI* ct){
return new (m.getHeap()) xDemonI(m.getImpl(),ct);
}
```

Now we can find out whether or not a demon is a constraint by using the `isAConstraint` member function:

```
IlcBool IlcDemonI::isAConstraint()
```

Moreover, we can also find out whether or not a constraint has been created at the top level (or in a goal) or added by another constraint (or a demon) by using the `getParentI` member function:

```
IlcDemonI IlcConstraintI::getParentI();
```

This member function returns the constraint itself, if the constraint has been added at the top level or in a goal. Otherwise, it references the demon/constraint that has added the constraint.

Obsolete Functions and Classes

A list of obsolete functions and classes, with their replacements, is given in the *ILOG Solver 5.1 Reference Manual*.

C H A P T E R

3

Migrating from Scheduler to Concert Technology

ILOG Scheduler 5.1 is designed to work with ILOG Concert Technology to model and to solve scheduling and resource allocation problems. The core model in such a problem is based upon a set of activities, a set of resources, and the set of scheduling constraints involving activities and resources. That set of scheduling constraints consists of the temporal constraints and the resource constraints.

Before you begin translating an existing ILOG Scheduler 4.4 application to ILOG Scheduler 5.1, please study Chapter 2, *Migrating from Solver to Concert Technology*, for details about migrating to Solver 5.1. See also the *ILOG Concert Technology 1.1 Reference Manual*, which contains the C++ classes for representing models as well as the base classes for implementing algorithms. A basic familiarity with ILOG Concert Technology and with the new features of ILOG Solver 5.1 are prerequisites to a successful migration from ILOG Scheduler 4.4 to ILOG Scheduler 5.1.

How to Translate a Program from Scheduler 4.4 to Scheduler 5.1

This section of the guide outlines the major steps in migrating from ILOG Scheduler 4.4 to ILOG Scheduler 5.1 with ILOG Concert Technology 1.1. (The following section walks you through an application following the steps of this outline in greater detail.)

1. Create the environment (an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*).
2. Create the model (an instance of `IloModel`, also documented in the *ILOG Concert Technology Reference Manual*).
3. Translate problem data into appropriate ILOG Concert Technology types. For example, values of `IlcFloat` become `IloNum` ; values of `IlcInt` become `IloInt` (or `IloNum`), and so forth. Likewise, arrays of `IlcFloatArray` become arrays of `IloNumArray`.
4. Translate the constrained elements (variables, activities, resources) from your old application to the corresponding elements in ILOG Concert Technology or ILOG Scheduler 5.1. For example, `IlcActivity` becomes `IloActivity` (documented in the *ILOG Scheduler Reference Manual*).
5. Wrap the old Solver 4.4 constraints (instances of `IlcConstraint` or its subclasses) according to the steps outlined in Chapter 2, *Migrating from Solver to Concert Technology* of this Migration Guide.

   Wrap the old Scheduler 4.4 constraints (e.g., `IlcResourceConstraint`, `IlcPrecedenceConstraint`) in the corresponding constraints in ILOG Concert Technology (e.g., `IloResourceConstraint`, `IloPrecedenceConstraint`).
6. The `IlcAltResSet` and `IlcAltResConstraint` instances are replaced by `IloAltResSet` and `IloAltResConstraint` instances. The Concert model does not define index variables on the alternative resource constraint. You can change the Solver constraints, and the meta-constraint terms, using the index variable by adding to the Concert model constraints and meta constraints built from the following two members of `IloAltResConstraint`:

   ```
   IloConstraint IloAltResConstraint::select(IloResource resource) const;

   IloConstraint IloAltResConstraint::selectSameResource
                                          (IloAltResConstraint ct) const;
   ```
7. Add the wrapped constraints to the model.
8. If you want to use the same global constraints as you used in your old problem (e.g., disjunctive constraint, edge finder) these can be managed by setting parameters in ILOG Scheduler 5.1. Translate those global constraints to Scheduler parameters.
9. If there are search goals in your old application, wrap the old Solver 4.4 goals (instances of `IlcGoal` or its subclasses) according to the steps outlined in Chapter 2, *Migrating*

from Solver to Concert Technology of this Migration Guide.

10. Create the solver (an instance of `IloSolver`) and its goal. The solve function will automatically extract the model and solve it with the goal given as argument.

11. Translate the search code from your old application according to the steps outlined in Chapter 2, *Migrating from Solver to Concert Technology* of this Migration Guide.

A Translation Example

With that outline of steps in mind, we will show now how to apply some of those steps. We will use the `bridgebr.cpp` example from the standard distribution of ILOG Scheduler 4.4 and 5.1. This example is based on the well known bridge-building problem extended by the fact that the resources have breaks. Some activities in this example are breakable, others are not.

With this example, we will demonstrate scheduling parameters in ILOG Scheduler 5.1, showing you how to translate global constraints in your Scheduler 4.4 application into the scheduling parameters of ILOG Scheduler 5.1.

Rationale

Roughly, our aim is to separate the problem data from the *model* (expressed in classes of ILOG Concert Technology) and to separate the model from the *search* (expressed in classes of ILOG Solver).

Creating the Model

In previous versions of ILOG Scheduler, an application created a manager in edit mode, declared a makespan variable, and added the makespan as an objective to the manager, like this:

```
IlcManager m(IlcEdit);
m.openLogFile("bridgebr.log");
IlcIntVar makespan;
IlcSchedule schedule = DefineProblem(m, makespan);
m.setObjMin(makespan);
```

Now, in ILOG Scheduler 5.1, we will create an environment (an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*). Within the environment, we create a model (an instance of `IloModel`, documented in the *ILOG Concert Technology Reference Manual*). We create a makespan as a numeric variable (an instance of `IloNumVar`, documented in the *ILOG Concert Technology Reference Manual*). Then we add our makespan as an objective to the model, like this:

```
IloEnv env;
IloNumVar makespan;
IloModel model = DefineModel(env, makespan);
model.add(IloMinimize(env, makespan));
```

In other words, the objective is now part of the *model* (not part of the manager as it used to be). When we create an algorithm (an instance of `IloSolver`, for example, documented in the *ILOG Solver Reference Manual*) it will extract the objective from the model for use in its search.

Extracting and Solving

In some problems, an instance of `IloSolver` will be able to extract the model and solve with no other intervention. (See the *ILOG Concert Technology User's Manual* for examples.)

In other problems, you will need a goal to guide the search for a solution. There are predefined goals, subclasses of the class `IloGoal`, suitable for use "off the shelf" in a model. These predefined goals are available in:

- ILOG Solver (see the functions `IloInstantiate`, `IloGenerate`, `IloBestInstantiate`, `IloBestGenerate`, documented in the *ILOG Solver Reference Manual*)
- ILOG Scheduler (see the functions `IloSetTimesForward`, `IloSetTimesBackward`, `IloRankForward`, `IloRankBackward`, `IloSequenceForward`, `IloSequenceBackward`, `IloAssignAlternative`, documented in the *ILOG Scheduler Reference Manual*).

At other times, you may want to write a goal yourself. To do this, you must use the facilities for extending ILOG Solver or ILOG Scheduler, namely the classes such as `IlcGoal`, or the functions `IlcInstantiate`, `IlcGenerate`, and so forth.

If your problem requires the dynamic creation of objects in your model (that is, if you need elements in the model that you can learn only "on the fly" during the solution search) then again you can use the facilities for *extending* ILOG Solver and ILOG Scheduler.

Let's return to our example to see those alternatives.

In Scheduler 4.4, you used the member function `IlcManager::nextSolution()` with a basic ranking goal, like this:

```
m.add(IlcRank(schedule, makespan));
while(m.nextSolution())
  m.out() << "current value of makespan: " << makespan << endl;
m.restart();
if (m.nextSolution())
  PrintSolution(schedule);
else
  m.out() << "No solution!" << endl;
```

In such a case, ILOG Scheduler 5.1 offers a predefined goal, `IloRankForward`, equivalent to `IlcRank`. After creating the solver and extracting the model, you call the resolution function, like this:

```
IloSolver solver(model);
    IloGoal goal = IloRankForward(env, makespan);
    if (solver.solve(goal))
      PrintSolution(solver);
    else
```

```
    solver.out() << "No solution!" << endl;
```

If you want to improve the resolution heuristics, you will define a goal yourself using the macro `ILOCPGOALWRAPPERX`, which allows you to return a search goal.

```
ILOCPGOALWRAPPER1(MyRank, solver, IloNumVar, makespan) {
  IloIntVar obj = solver.getIntVar(makespan);
  IlcScheduler schedule(solver);
  return IlcRank(schedule, obj);
}
```

The constructor `IlcScheduler(IloSolver solver)` gives you the instance of `IlcScheduler` of the solver. `IlcScheduler` is a subclass of `IlcSchedule` that enables you to write the scheduling goal, just as in Scheduler 4.4.

Accessing the Solution

After resolution, you can either directly read the data in the problem data structures or use instances of the `IloSchedulerSolution` to store the data.

You can use such stored solutions as the basis of your next attempt to search for a solution if you like. Before attempting another search, you can also make changes in a model. For example, you can add or remove constraints from the model, extract it again, and resolve. That is, the changes in a model may be monotonic or non monotonic.

Creating Activities

The creation of the activities in a model is similar to the creation of activities in ILOG Scheduler 4.4 applications except for the types of argument. In rough terms, `Ilo` replaces `Ilc` in the declaration of most arguments. Furthermore, an instance of `IlcSchedule` in an ILOG Scheduler 4.4 application becomes the Concert Technology environment (an instance of `IloEnv`) in an ILOG Scheduler 5.1 application. Likewise, a value of type `IlcInt` from a Scheduler 4.4 application becomes a value of type `IloNum` in ILOG Scheduler 5.1.

Scheduler 4.4

Here, for example, are lines creating activities in a Scheduler 4.4 application:

```
IlcActivity MakeActivity(IlcSchedule schedule,
                         const char* name,
                         IlcInt duration)
{
  IlcActivity activity(schedule, duration);
  activity.setName(name);
  return activity;
}
```

Scheduler 5.1

Here are the equivalent lines creating an activity in ILOG Scheduler 5.1:

```
IloActivity MakeActivity(IloEnv env,
                         const char* name,
                         IloNum duration)
{
  IloActivity activity(env, duration, name);
  return activity;
}
```

We now consider the fact that some activities are breakable.

Scheduler 4.4

```
IlcActivity MakeBreakableActivity(IlcSchedule schedule,
                                  const char* name,
                                  IlcInt duration)
{
  IlcBreakableActivity activity(schedule, duration);
  activity.setName(name);
  return activity;
}
```

Scheduler 5.1

The member function `IloActivity::setBreakable` allows you to declare the fact that an activity is breakable.

```
IloActivity activity = MakeActivity(env, name, duration);
activity.setBreakable(IloTrue);
```

Notice that the classes `IlcBreakableActivity` and `IlcIntervalActivity` have disappeared. By default, in Scheduler 5.1, an instance of `IloActivity` or `IlcActivity` is a non-breakable activity. The member functions `IloActivity::setBreakable` and `IlcActivity::setBreakable` allow you to control whether or not an activity is breakable.

Translating Global Constraints into Scheduling Parameters

Activities naturally fall into two groups: the breakable and the non-breakable ones. In Concert Technology we take advantage of this idea in order to save memory in the definition of the model. The characteristics of activities are managed by means of shared parameters. We start by declaring the repository of the scheduler parameters, namely an instance of `IloSchedulerEnv` like this:

```
IloSchedulerEnv schedEnv(env);
```

We change the default behavior of the parameter dealing with the breakability of the resources, like this:

```
schedEnv.getActivityBasicParam().setBreakable();
```

Then, by default, all the activities created from now on will be breakable.

```
IloActivity A1 = MakeActivity(model, "A1", 4);
```

After declaring the set of breakable activities, we will create a parameter for non-breakable activities and use it as the default, like this:

```
IloActivityBasicParam newDefaultParam(env);
schedEnv.setActivityBasicParam(newDefaultParam);
```

In consequence, from now on the activities we create are non-breakable.

```
IloActivity AB1 = MakeActivity(model, "AB1", 1);
```

The constructor of a schedule (an instance of `IlcSchedule`) in Schedule 4.4 required an origin and a horizon, that is, two integer values used as defaults to define the range of the date variable of an activity. Here is an example of that old convention:

```
IlcInt origin = 0;
IlcInt horizon = 365;
IlcManager m(IlcNoEdit);
IlcSchedule schedule(m, origin, horizon);
```

This information is no longer required in Scheduler 5.1. If you still need to make this information explicit in your application, you use the scheduler environment, like this:

```
IloNum origin = 0;
IloNum horizon = 365;
IloEnv env;
IloSchedulerEnv schedEnv(env);
schedEnv.setOrigin(origin);
schedEnv.setHorizon(horizon);
```

These values will be used by the model and the extractor as default values when needed.

Adding Constraints

In Scheduler 4.4, activities were defined in terms of constrained variables that were instances of ILOG Solver classes such as `IlcIntVar` or `IlcFloatVar`. During a search for a solution, ILOG Solver 4.4 attempted to assign values to those decision variables. Furthermore, it was possible to add constraints to a manager (an instance of `IlcManager`) while it was in edit mode. In other words, there was not an entirely clear distinction between the modeling variables of an activity and the search for a solution. For example, in ILOG Scheduler 4.4, you added temporal constraints on activities to the manager, like this:

```
manager.add(activity1.startsAfterEnd(activity2));
```

In contrast, in ILOG Scheduler 5.1 and ILOG Concert Technology, there is a clear distinction between the model, made up of decision variables, and the search for an assignment of values to the decision variables. This distinction is highlighted by the fact that an algorithm (an instance of `IloSolver`, for example) extracts information *from* a model

for its own use. You must add the temporal constraints and time bound constraints of your problem to your *model* in order for them to be extracted by an algorithm (an instance of `IloSolver`, for example) and taken into account during the search. For example, you add a temporal constraint on an activity like this:

```
model.add(activity1.startsAfterEnd(activity2));
```

In ILOG Concert Technology, when you add a constraint to a model, all the variables belonging to that constraint are implicitly added the model for you automatically. Then when your algorithm (an instance of `IloSolver`, for example) extracts the model with that added constraint, it will also extract those variables in the constraint.

Creating Resources

The creation of a resource resembles the creation of an activity; that is, you create the resource in an environment (an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*); then you add its constraints to a model; then you extract the model for an algorithm.

In Scheduler 4.4, the following lines:

```
void
MakeResource(IlcSchedule schedule,
             const char* name,
             IlcInt numberOfActivities,
             IlcActivity* activities)
{
  IlcManager m=schedule.getManager();
  IlcUnaryResource resource(schedule);
  resource.setName(name);
  for (IlcInt i = 0; i < numberOfActivities; i++)
    m.add(activities[i].requires(resource));
  resource.close();

}
```

are equivalent in Scheduler 5.1 to these lines:

```
void
MakeResource(IloModel model,
             const char* name,
             IloInt numberOfActivities,
             IloActivityArray activities)
{
  IloEnv env = model.getEnv();
  IloUnaryResource resource(env, name);
  for (IloInt i = 0; i < numberOfActivities; i++)
    model.add(activities[i].requires(resource));

}
```

Global Constraints of Scheduler 4.4 Become Parameters in Scheduler 5.1

There is also parameter management in the resources of ILOG Scheduler 5.1. Basically, the parameters are annotations of the resource. The parameters declare the exact behavior and content of a resource. In practice, these annotations allow you to enforce and precisely define constraints to apply on sets of activities requiring the resource. Those constraints on a resource might include, for example, the capacity levels of a discrete resource, the break list of a resource, or the transition time function of a resource. You have a choice between using parameters on a *set* of resources or annotating a single resource locally.

Using Parameters to Close Resources

In the previous code sample in Scheduler 4.4, the resource was closed by the function `IlcResource::close()`. In ILOG Scheduler 5.1, the resource is considered closed by default by the algorithm that extracts the model. If you want to declare a resource not closed at resolution time, you have two alternatives:

- use the member function `IloResource::keepOpen(IloTrue)`
- use the resource parameter shared by a set of resources.

For example, using the default resource parameter of the Scheduler environment, write:

```
schedenv.getResourceParam().keepOpen(IloTrue);
```

Using Parameters to Invoke Disjunctive Constraint or Edge Finder

Another kind of parameter allows you to set the enforcement level on certain characteristics of the resource. For example, you can invoke the disjunctive constraint and the edge finder constraint in a model like this:

```
IloUnaryResource resource = /* ... */
resource.setCapacityEnforcement(IloMediumHigh);
```

You can also use the resource parameter like this:

```
schedenv.getResourceParam().setCapacityEnforcement(IloMediumHigh);
```

Using Parameters on Time Intervals

Another kind of parameter defines functions on an interval of time, for example, such functions as the maximum capacity of a resource, the time interval of an energy relaxation or a list of breaks. In Scheduler 4.4, the list of breaks was a specific object that you filled and attached to a resource, like this:

```
IlcManager m(IlcNoEdit);
IlcInt horizon = 365;
IlcSchedule schedule(m, 0, horizon);
IlcBreakList blist(schedule);
IlcInt i=7;
while(i<horizon) {
  blist.addBreak(i-2, i);
  i = i+7;
}
IlcUnaryResource resource(schedule);
resource.setBreaks(blist);
```

In Scheduler 5.1, a generic class, the `IloIntervalList` allows you to declare a list of time intervals that can be assigned as a list of breaks on resources. For example, the list of breaks can be shared by default, thanks to the schedule environment properties.

```
IloIntervalList breakParam = schedEnv.getBreakListParam();
IloInt i=7;
while(i < horizon) {
  breakParam.addInterval(i-2, i);
  i = i+7;
}
```

You can directly declare the breaks on a resource by using the `addBreak` member function:

```
void IloResource::addBreak(IloNum start, IloNum end, IloNum type =0L);
```

Moreover, as for any parameter, you can build a list of breaks and put it on a resource, like this:

```
IloIntervalList breakParam = IloIntervalList(env, 0, horizon);
IloInt i=7;
while(i < horizon) {
  breakParam.addInterval(i-2, i);
  i = i+7;
}
resource.setBreaksParam(breakParam);
```

Notice that, by default, the instance of `IloIntervalList` extracted by a resolution algorithm is closed. As for a resource, you can ask for the instance not to be considered closed during the search for a solution. This will allow new breaks on the resource to be added during the search. There are alternative ways of making a set of breaks not closed.

- You can call this member function on the instance of `IloIntervalList`:

  ```
  void IloIntervalList::keepOpen(IloBool val =IloTrue) const;
  ```

- Or you can work directly on the resource that the set applies to. For example, in the case of keeping open the list of breaks, you can call the `keepBreakListOpen` member function:

  ```
  void IloResource::keepBreakListOpen(IloBool keepOpen =IloTrue) const;
  ```

This section has simply outlined the way to translate global constraints of an application written for ILOG Scheduler 4.4 into an application of ILOG Scheduler 5.1. For more details about using the scheduling parameters of ILOG Scheduler 5.1, see the following documents:

- *ILOG Scheduler 5.1 Extensions Reference Manual.* This manual contains tables of mappings between Solver and Concert classes, plus the means the extractor uses to create the global constraints and resources from the model.
- *ILOG Scheduler 5.1 User's Manual.*

C H A P T E R

4

Migrating from Dispatcher to Concert Technology

We recommend that you read the *ILOG Concert Technology User's Manual* and, in this Migration Guide, Chapter 2, *Migrating from Solver to Concert Technology* before reading the present chapter.

Modeling is Based on Concert Technology

ILOG Dispatcher 3.1 uses ILOG Concert Technology. This means that the statement of a problem is clearly separated from the way it is solved. For this purpose Concert Technology provides the concept of the model (through the class `IloModel`). The problem is then defined by adding modeling objects (prefixed by `Ilo`) to an `IloModel`.

When the problem is to be solved, the information from the model is extracted by an algorithm. In Dispatcher terms, this translates into the following: the functionality provided by the `IlcRoutingPlan` class in version 2.1 is now provided by three `Ilo`-prefixed classes:

- `IloModel`
- `IloDispatcher`
- `IloRoutingSolution`

This means that:

- The description of the model is handled by the class `IloModel`, as with all the other optimization libraries in the ILOG Optimization Suite, which are now Concert-enabled.
- The solving of the problem, and the constraints related to it are algorithmically managed by the class `IloDispatcher`.
- The storage and manipulation of a solution to a routing problem are handled by the class `IloRoutingSolution`. (There is no equivalent to an internal solution in `IlcRoutingPlan` now, so an `IloRoutingSolution` has to be created before solving any problem).

Note also that the functions in ILOG Dispatcher 2.1 which take as argument an instance of `IlcRoutingPlan` now take, in Dispatcher 3.1 an instance of `IloEnv`.

To summarize, here is the beginning of a Dispatcher 2.1 program:

```
IlcManager m(IlcEdit);
IlcRoutingPlan plan(m);
IlcDimension2 time(plan, IlcEuclidean, "Time");
IlcDimension2 length(plan, IlcEuclidean, "Length");
IlcDimension1 weight(plan, "Weight");
```

and the corresponding beginning of a Dispatcher 3.1 program:

```
IloEnv env;
IloModel mdl(env);
...
IloDimension2 time(env, IloEuclidean, "Time");
mdl.add(time);
IloDimension2 length(env, IloEuclidean, "Length");
mdl.add(length);
IloDimension1 weight(env, "Weight");
mdl.add(weight);
IloNode depot(env, depotX, depotY);
IloVisit first(depot, "Depot");
mdl.add(first.getCumulVar(time) >= openTime);
...
IloVehicle vehicle(first, last, name);
vehicle.setCost(length, 1.0);
vehicle.setCapacity(weight, capacity);
mdl.add(vehicle);
```

Once the model is stated, it can be extracted and solved using the specific algorithms provided by Dispatcher.

```
IloSolver solver(mdl);
IloDispatcher dispatcher(solver);
IloRoutingSolution solution(mdl);
```

Search Features

The search has been modified in two ways in ILOG Dispatcher 3.1. As said previously, any object or function taking an `IlcRoutingPlan` in ILOG Dispatcher 2.1, now takes an instance of `IloEnv`. This of course applies to the solution generation goals. The following code demonstrates this:

```
IloSolver solver(mdl);
IloDispatcher dispatcher(solver);
IloRoutingSolution solution(mdl);
IloGoal instantiateCost = IloDichotomize(env,
                                         dispatcher.getCostVar(),
                                         IloFalse);
IloGoal goal = IloSavingsGenerate(env) && instantiateCost;
if (!solver.solve(goal)) {
  solver.out() << "Not enough vehicles to generate first solution" << endl;
  return 0;
}
solution.store(solver);
```

ILOG Dispatcher 3.1 now uses a new generic, open local search API, which is available in Solver 5.1. The `IlcMove` class of ILOG Dispatcher 2.1 now corresponds to the `IloNHood` class found in ILOG Solver 5.1. Also, the member functions of the class `IlcRoutingPlan` dealing with local search (`improve, gls, gts, fastGls`) are now replaced by the new search strategies based on the framework offered by ILOG Solver. The following code exemplifies this:

```
IloGoal restoreSolution = IloRestoreSolution(env, solution);
IloNHood nhood = IloTwoOpt(env)
+ IloOrOpt(env)
+ IloRelocate(env)
+ IloExchange(env)
+ IloCross(env);
IloGoal improve = IloSingleMove(env,
solution,
nhood,
IloImprove(env),
IloFirstSolution(env),
instantiateCost);
while (solver.solve(improve)) {}
solver.solve(restoreSolution);
```

Correspondence Tables

The following tables explain how to translate each member function of the Dispatcher 2.1 classes into Dispatcher 3.1. In the examples of code given:

`dispatcher`	represents an instance of	`IloDispatcher`
`visit`	represents an instance of	`IlcVisit` in Dispatcher 2.1 `IloVisit` in Dispatcher 3.1
`rplan`	represents an instance of	`IlcRoutingPlan`
`vehicle`	represents an instance of	`IlcVehicle` in Dispatcher 2.1 `IloVehicle` in Dispatcher 3.1
`solver`	represents an instance of	`IloSolver`
`solution`	represents an instance of	`IloRoutingSolution`
`model`	represents an instance of	`IloModel`

The class `IloSolver` is from ILOG Solver 5.1. The classes `IloSolution` and `IloModel` are from ILOG Concert Technology 1.1.

Translating Code for IlcRoutingPlan

An instance of `IlcRoutingPlan` passed to a function translates to an instance of the class `IloEnv`.

Dispatcher 2.1	Dispatcher 3.1
`IlcRoutingPlan`	`IloModel`, `IloDispatcher`, `IloRoutingSolution`
`addGoal`	Some of the predefined goals take a subgoal, otherwise you can add the subgoal by combining the other goals of the search with an &&.

`rplan.fastGLS()`	`IloDispatcherGLS` with `IloFirstSolution` selector.
	
	`void ImproveWithFastGLS(IloDispatcher dispatcher,`
	`IloRoutingSolution solution,`
	`IloNHood nhood) {`
	`nhood.reset();`
	`IloNumVar cost = dispatcher.getCostVar();`
	`IloEnv env = dispatcher.getEnv();`
	`IloGoal instantiateCost = IloDichotomize(env, cost, IloFalse);`
	`IloRoutingSolution rsol = solution.makeClone(env);`
	`IloRoutingSolution best = solution.makeClone(env);`
	`IloDispatcherGLS dgls(env, 0.2);`
	
	`IloGoal move = IloSingleMove(env, rsol, nhood, dgls,`
	`instantiateCost);`
	`move = move && IloStoreBestSolution(env, best);`
	`IloSolver solver = dispatcher.getSolver();`
	`IloCouple(nhood, dgls);`
	`for (IloInt i = 0; i < 300; i++) {`
	`if (solver.solve(move)) {`
	`cout << "Cost = " << solver.getMax(cost) << endl;`
	`}`
	`else {`
	`cout << "---" << endl;`
	`if (dgls.complete()) break;`
	`}`
	`}`
	`IloDecouple(nhood, dgls);`
	`IloGoal restoreSolution = IloRestoreSolution(env, best) &&`
	`instantiateCost;`
	`solver.solve(restoreSolution);`
	`rsol.end();`
	`best.end();`
	`}`

Dispatcher 2.1	Dispatcher 3.1
rplan.getBestIteration	To be done directly by the user in the local search code.
rplan.getCostPrecision	To be done directly by the user in the local search code.
rplan.getCostVar()	dispatcher.getCostVar()
rplan.getGoal()	Not applicable.
rplan.getNbOfDimensions()	dispatcher.getNbOfDimensions()
rplan.getNbOfMoves()	dispatcher.getNbOfMoves()
rplan.getNbOfNodes()	dispatcher.getNbOfNodes()
rplan.getNbOfSuccesses()	dispatcher.getNbOfSuccesses()
rplan.getNbOfUnperformedVisits()	dispatcher.getNbOfUnperformedVisits()
rplan.getNbOfVehicles()	dispatcher.getNbOfVehicles()
rplan.getNbOfVehiclesUsed()	dispatcher.getNbOfVehiclesUsed()
rplan.getNbOfVisits()	dispatcher.getNbOfVisits()
rplan.getNowhere()	IloNode(IloEnv env, IloBool everywhere, const char* name=0);
rplan.getPenFactor()	IloDispatcherGLS::getPenaltyFactor()
rplan.getTotalCost()	dispatcher.getTotalCost()
rplan.getVehicle(IlcInt)	dispatcher.getVehicle(IloInt)
rplan.getVisit(IlcInt)	dispatcher.getVisit(IloInt)

Dispatcher 2.1	Dispatcher 3.1

rplan.gls()	IloDispatcherGLS with IloMinimizeVar selector.
	
	void ImproveWithGLS(IloDispatcher dispatcher,
	IloRoutingSolution solution,
	IloNHood nhood) {
	nhood.reset();
	IloNumVar cost = dispatcher.getCostVar();
	IloEnv env = dispatcher.getEnv();
	IloGoal instantiateCost = IloDichotomize(env, cost,
	IloFalse);
	IloRoutingSolution rsol = solution.makeClone(env);
	IloRoutingSolution best = solution.makeClone(env);
	IloDispatcherGLS dgls(env, 0.2);
	
	IloSearchSelector sel = IloMinimizeVar(env,
	dgls.getPenalizedCostVar());
	IloGoal move = IloSingleMove(env, rsol, nhood, dgls,
	sel, instantiateCost);
	move = move && IloStoreBestSolution(env, best);
	IloSolver solver = dispatcher.getSolver();
	IloCouple(nhood, dgls);
	for (IloInt i = 0; i < 150; i++) {
	if (solver.solve(move)) {
	cout << "Cost = " << solver.getMax(cost) << endl;
	}
	else {
	cout << "---" << endl;
	if (dgls.complete()) break;
	}
	}
	IloDecouple(nhood, dgls);
	IloGoal restoreSolution = IloRestoreSolution(env, best) &&
	instantiateCost;
	solver.solve(restoreSolution);
	rsol.end();
	best.end();
	}

Dispatcher 2.1	Dispatcher 3.1

gts()	IloDispatcherGLS + IloDispatcherTabuSearch
	
	void ImproveWithGTS(IloDispatcher dispatcher,
	IloRoutingSolution solution,
	IloNHood nhood) {
	nhood.reset();
	IloNumVar cost = dispatcher.getCostVar();
	IloEnv env = dispatcher.getEnv();
	IloGoal instantiateCost = IloDichotomize(env, cost,
	IloFalse);
	IloRoutingSolution rsol = solution.makeClone(env);
	IloRoutingSolution best = solution.makeClone(env);
	IloDispatcherGLS dgls(env, 0.2);
	IloDispatcherTabuSearch dt(env, 5);
	IloMetaHeuristic meta = dgls + dt;
	
	IloSearchSelector sel = IloMinimizeVar(env,
	dgls.getPenalizedCostVar());
	IloGoal move = IloSingleMove(env, rsol, nhood, meta, sel,
	instantiateCost);
	move = move && IloStoreBestSolution(env, best);
	IloSolver solver = dispatcher.getSolver();
	IloCouple(nhood, meta);
	for (IloInt i = 0; i < 150; i++) {
	if (solver.solve(move)) {
	cout << "Cost = " << solver.getMax(cost) << endl;
	}
	else {
	cout << "---" << endl;
	if (meta.complete()) break;
	}
	}
	IloDecouple(nhood, meta);
	IloGoal restoreSolution = IloRestoreSolution(env, best) &&
	instantiateCost;
	solver.solve(restoreSolution);
	rsol.end();
	best.end();
	}

Dispatcher 2.1	Dispatcher 3.1

improve()	IloSingleMove with IloFirstSolution selector.
	
	void FirstAccept(IloDispatcher dispatcher,
	IloRoutingSolution solution,
	IloNHood nhood) {
	nhood.reset();
	IloNumVar cost = dispatcher.getCostVar();
	IloEnv env = dispatcher.getEnv();
	IloGoal instantiateCost = IloDichotomize(env, cost,
	IloFalse);
	IloRoutingSolution rsol = solution.makeClone(env);
	IloGoal move = IloSingleMove(env, rsol, nhood,
	IloImprove(env),
	instantiateCost);
	IloSolver solver = dispatcher.getSolver();
	while (solver.solve(move)) {
	cout << "Cost = " << solver.getMax(cost) << endl;
	}
	IloGoal restoreSolution = IloRestoreSolution(env, rsol) &&
	instantiateCost;
	solver.solve(restoreSolution);
	rsol.end();
	}

rplan.insertVisits()	IloInsertVisit for all visits.
rplan.printInformation()	dispatcher.printInformation()
rplan.refresh()	IloDistance::refresh()
rplan.removeGoal	Not applicable.
rplan.restoreSolution()	No internal solution. Not applicable.
rplan.restoreSolution(solution)	solver.solve(IloRestoreSolution(env, solution))
rplan.saveSolution()	No internal solution. Not applicable.
rplan.saveSolution(solution)	solution.store(solver)
rplan.setCostPrecision()	To be done directly by the user in the local search code.
rplan.setHook()	To be done directly by the user in the local search code or in a goal.
rplan.setMoveLimit()	To be done directly by the user in the local search code (add code).
rplan.setPenFactor()	IloDispatcherGLS::setPenaltyFactor()
rplan.setTimeLimt()	Same as setMoveLimit().

Dispatcher 2.1	Dispatcher 3.1

steepest()	IloSingleMove with IloMinimizeVar selector.
	
	void BestAccept(IloDispatcher dispatcher,
	IloRoutingSolution solution,
	IloNHood nhood) {
	nhood.reset();
	IloNumVar cost = dispatcher.getCostVar();
	IloEnv env = dispatcher.getEnv();
	IloGoal instantiateCost = IloDichotomize(env, cost, IloFalse);
	IloRoutingSolution rsol = solution.makeClone(env);
	IloGoal move = IloSingleMove(env, rsol, nhood,
	IloImprove(env),
	IloMinimizeVar(env, cost),
	instantiateCost);
	IloSolver solver = dispatcher.getSolver();
	while (solver.solve(move)) {
	cout << "Cost = " << solver.getMax(cost) << endl;
	}
	IloGoal restoreSolution = IloRestoreSolution(env, rsol) &&
	instantiateCost;
	solver.solve(restoreSolution);
	rsol.end();
	}

Dispatcher 2.1	Dispatcher 3.1

rplan.tabu()	IloDispatcherTabuSearch
	
	void ImproveWithTabu(IloDispatcher dispatcher,
	IloRoutingSolution solution,
	IloNHood nhood) {
	nhood.reset();
	IloNumVar cost = dispatcher.getCostVar();
	IloEnv env = dispatcher.getEnv();
	IloGoal instantiateCost = IloDichotomize(env, cost,
	IloFalse);
	IloRoutingSolution rsol = solution.makeClone(env);
	IloRoutingSolution best = solution.makeClone(env);
	IloDispatcherTabuSearch dts(env, 12);
	
	IloGoal move = IloSingleMove(env, rsol, nhood, dts,
	IloMinimizeVar(env, cost),
	instantiateCost);
	move = move && IloStoreBestSolution(env, best);
	IloSolver solver = dispatcher.getSolver();
	cout << "Tenure = 12" << endl;
	for (IloInt i = 0; i < 150; i++) {
	if (i == 70) {
	cout << "Tenure = 20" << endl;
	dts.setTenure(20);
	}
	if (i == 85) {
	cout << "Tenure = 5" << endl;
	dts.setTenure(5);
	}
	if (i == 105) {
	dts.setTenure(12);
	cout << "Tenure = 12" << endl;
	}
	if (solver.solve(move)) {
	cout << "Cost = " << solver.getMax(cost) << endl;
	}
	else {
	if (dts.complete()) break;
	}
	}
	IloGoal restoreSolution = IloRestoreSolution(env, best) &&
	instantiateCost;
	solver.solve(restoreSolution);
	rsol.end();
	best.end();
	}

rplan.unsetLimit()	Not applicable.

Translating Code for IlcVisit

In the right-hand column of the table: case 1) returns an `Ilc` instance, case 2) returns an `Ilo` instance.

Dispatcher 2.1	Dispatcher 3.1
`IlcVisit`	`IloVisit`
`visit.disable()`	`solution.remove(visit); model.remove(visit);`
`visit.enable()`	`solution.add(visit); model.add(visit);`
`visit1.getCostTo(visit2, vehicle)`	`dispatcher.getCost(visit1, visit2, vehicle)`
`visit.getIndex()`	`dispatcher.getIndex(visit);`
`getManager()`	Not applicable.
`visit.getNext()`	`dispatcher.getVisit(dispatcher.getNextVar(visit).getValue())`
`visit.getNextVar()`	1) `dispatcher.getNextVar(visit)` 2) `visit.getNextVar()`
`visit.getPrev()`	`dispatcher.getVisit(dispatcher.getPrevVar(visit).getValue())`
`visit.getPrevVar()`	1) `dispatcher.getPrevVar(visit)` 2) `visit.getPrevVar()`
`visit.getRankVar()`	1) `dispatcher.getRankVar(visit)` 2) `visit.getRankVar()`
`visit.getQuantity(dim)`	`dispatcher.getTransitVar(dim).getValue()`
`visit.getQuantityVar(dim)`	1) `dispatcher.getTransitVar(visit, dim)` 2) `visit.getTransitVar(dim)`
`visit.getSavedNext()`	`solution.getNext(visit)`
`visit.getSavedPrev()`	`solution.getPrev(visit)`
`visit.getTransitVar(dim)`	1) `dispatcher.getTransitVar(visit, dim)` 2) `visit.getTransitVar(dim)`
`visit.getTravelVar(dim)`	1) `dispatcher.getTravelVar(visit, dim)` 2) `visit.getTravelVar(dim)`
`visit.getVehicle()`	`dispatcher.getVehicle(dispatcher.getVehicleVar(visit).getValue())`

Dispatcher 2.1	Dispatcher 3.1
visit.getVehicleVar()	1) dispatcher.getVehicleVar(visit) 2) visit.getVehicleVar()
visit.getWaitVar()	1) dispatcher.getWaitVar(visit) 2) visit.getWaitVar()
visit.insert()	solver.solve(IloInsertVisit(env, visit))
visit.insert(vehicle)	solver.solve(IloInsertVisit(env, visit, vehicle))
visit1.insert(visit2)	solver.solve(IloInsertVisit(env, visit1, visit2))
isDisabled()	Not applicable.
visit1.isJustAfter(visit2)	visit1.getPrevVar() == visit2
visit1.isJustBefore(visit2)	visit1.getNextVar() == visit2
visit.makeIncompatibleWith(vehicle)	visit.getVehicleVar() != vehicle
visit1.makeIncompatibleWith(visit2)	visit1.getVehicleVar() != visit2.getVehicleVar()
visit1.sameVehicle(visit2)	visit1.getVehicleVar() == visit2.getVehicleVar()
visit.setDelay(dim, value)	model.add(visit.getDelayVar(dim) == value)
visit.setDelay(dim2, dim, value)	model.add(visit.getDelayVar(dim2) == value * IloAbs(visit.getTransitVar(dim)))
visit.setDelayVar(dim, exp)	model.add(visit.getDelayVar(dim) == exp)
visit1.setNext(visit2)	dispatcher.setNext(visit1, visit2)
visit1.setPrev(visit2)	dispatcher.setPrev(visit1, visit2)
visit.setQuantity(dim, value)	model.add(visit.getTransitVar(dim) == value)
visit.setQuantityVar(dim, var)	Not applicable.
visit1.setSavedNext(visit2)	solution.setNext(visit1, visit2)
visit1.setSavedPrev(visit2)	solution.setPrev(visit1, visit2)
visit.setVehicle(vehicle)	dispatcher.setVehicle(visit, vehicle)
IlcVisitIterator	IloIterator<IloVisit>, IloRoutingSolution::VisitIterator
IlcUnperformedVisitIterator	IloDispatcher::UnperformedVisitIterator, IloRoutingSolution::UnperformedVisitIterator

Translating Code for IlcVehicle

Dispatcher 2.1	Dispatcher 3.1
`IlcVehicle`	`IloVehicle`
`vehicle.disable()`	`solution.remove(vehicle);` `model.remove(vehicle);`
`enable()`	`solution.add(vehicle);` `model.add(vehicle);`
`generateRoute()`	Not applicable.
`vehicle.getIndex()`	`dispatcher.getIndex(veh)`
`getNbOfVehicleBreakConstraints()`	Not applicable.
`getRouteCompleteVar()`	Not applicable.
`isDisabled()`	Not applicable.
`isRouteComplete()`	Not applicable.
`IlcVehicleIterator`	`IloIterator<IloVehicle>,` `IloRoutingSolution::VehicleIterator`
`IlcRouteIterator`	`IloDispatcher::RouteIterator,` `IloRoutingSolution::RouteIterator`

Translating Code for Dimensions

Dispatcher 2.1	Dispatcher 3.1
`IlcDimension::getRoutingPlan()`	Not applicable.

`IlcDimensionIterator`	`IloIterator<IloDimension>,`
	`IloIterator<IloDimension1>,`
	`IloIterator<IloDimension2>`

Translating Code for Distances

Distance functions consider three parameters in ILOG Dispatcher 3.1 (two nodes and a vehicle) compared to two in previous versions of ILOG Dispatcher (two nodes). These simpler distance functions can still be implemented using `IloSimpleDistanceFunction` and `IloSimpleDistanceEvalI`.

Dispatcher 2.1	Dispatcher 3.1
`IlcDistance`	`IloDistance`
`IlcDistanceEvalI`	`IloDistanceEvalI`, `IloSimpleDistanceEvalI`
`IlcDistanceFunction`	`IloDistanceFunction`, `IloSimpleDistanceFunction`
`IlcDistanceI`	`IloDistanceI`
`IlcDistMax`	`IloDistMax`
`IlcEuclidean`	`IloEuclidean`
`IlcExplicitDistance`, `IlcExplicitDistanceI`	Removed.
`IlcGeographical`	`IloGeographical`
`IlcManhattan`	`IloManhattan`

Translating Code for Goals

Dispatcher 2.1	Dispatcher 3.1
`IlcAllUnperformedGenerate`	When added to an instance of `IloRoutingSolution`, the saved state of an instance of `IloVisit` is unperformed.
`IlcEmptyPlanGenerate`	Create an empty solution and add vehicles only.
`IlcGenerate`	`IloDispatcherGenerate`
`IlcInsertionGenerate`	`IloInsertionGenerate`
`IlcInstantiateTransits`	`IloInstantiateTransits`
`IlcInstantiateVehicleBreak`	`IloInstantiateVehicleBreak`
`IlcInstantiateVehicleBreaks`	`IloInstantiateVehicleBreaks`
`IlcInstantiateVehicleBreakDuration`	`IloInstantiateVehicleBreakDuration`
`IlcInstantiateVehicleBreakPosition`	`IloInstantiateVehicleBreakPosition`
`IlcInstantiateVehicleBreakStart`	`IloInstantiateVehicleBreakStart`
`IlcNearestAdditionGenerate`	`IloNearestAdditionGenerate`
`IlcNearestDepotGenerate`	`IloNearestDepotGenerate`
`IlcSavingsGenerate`	`IloSavingsGenerate`
`IlcSweepGenerate`	`IloSweepGenerate`

Translating Code for Solutions

Dispatcher 2.1	Dispatcher 3.1
`IlcRoutingSolution`	`IloRoutingSolution`
`solution.getCost()`	`solution.getSolution().getObjectiveValue()`
`read()`	Not applicable.

Translating Code for Neighborhoods

Dispatcher 2.1	Dispatcher 3.1
`IlcCross`	`IloCross`
`IlcExchange`	`IloExchange`, `IloSwapPerform` for unperformed visits.
`IlcExchangePair`	`IloExchange`, `IloSwapPerform` for unperformed visits.
`IlcMove`	Equivalent of `IloNHood`.
`IlcOrOpt`	`IloOrOpt`
`IlcRelocate`	`IloRelocate`, `IloMakePerformed` and `IloMakeUnperformed` for unperformed visits.
`IlcRelocatePair`	`IloRelocate`, `IloMakePerformed` and `IloMakeUnperformed` for unperformed visits.
`IlcThreeOpt`	Removed.
`IlcTwoOpt`	`IloTwoOpt`

Translating Code for Breaks

In the right-hand column of the table: case 1) returns an `Ilc` instance, case 2) returns an `Ilo` instance.

Dispatcher 2.1	Dispatcher 3.1
`IlcVehicleBreakConstraint`	`IloVehicleBreakCon`
`break.getDurationVar()`	1) `dispatcher.getDurationVar(break)` 2) `break.getDurationVar()`
`break.getPosition()`	`dispatcher.getPosition(break)`
`break.getPositionVar()`	1) `dispatcher.getPositionVar(break)` 2) To constrain the position of break, you can use: `break.justAfter() break.getStartVar()`
`break.getStartVar`	1) `dispatcher.getStartVar(break)` 2) `break.getStartVar()`
`break.isPerformed()`	`dispatcher.isPerformed(break)`
`break.isUnperformed()`	`dispatcher.isUnperformed(break)`
`IlcVehicleBreakConstraintIterator`	`IloIterator<IloVehicleBreakCon>`

Miscellaneous Translations

Dispatcher 2.1	Dispatcher 3.1
`IlcNode`	`IloNode`
`IlcNodeIterator`	`IloIterator<IloNode>`
`IlcOutputManip`	`IloOutputManip`
`IlcRPHook`	Removed.
`ILCSETWINDOW(visit.getCumulVar(dim), begin <= var && var <= end)`	`model.add(begin <= visit.getCumulVar(dim) <= end)`
`IlcTerse(IlcRoutingPlan)`	`IloTerse(IloDispatcher)`
`IlcVerbosei(IlcRoutingPlan)`	`IloVerbose(IloDispatcher)`

C H A P T E R

5

Migrating from Configurator to Concert Technology

To translate the code of a Configurator 1.0 application into Configurator 2.1 code, you need to do the following:

- Replace the `IlcConfigurator` objects with an `IloCatalog`, an `IloConfiguration` and an `IloConfigurator` object.
- Declare type hierarchies, type-tables, ports, attributes, and constraints to the catalog.
- Build the catalog instead of linking the configurator.
- Create the initial components in the `IloConfiguration` object.
- Declare the requirement constraints to the `IloConfiguration` object instead of adding them to the `IlcManager`.
- Extract the `IloConfiguration` model and solve it with an `IloSolver` algorithm.
- Replace the `IlcConfigStrategy` declarations with an instance of `IloConfigStrategy`, to which the corresponding generic goals and their attached component types are declared.
- Replace the `Ilc` selectors with the corresponding `Ilo` selectors.
- Replace the interchangeability classes with substitutability classes, and pass them as arguments of the goals, instead of directly attaching them to component types.
- Add the minimization object to the `IloConfiguration` object, instead of adding it to the `IlcManager` with `setObjMin`.

The following table gives the correspondence between the Configurator 1.0 API and the new Configurator 2.1 API.

Configurator 1.0 API	Configurator 2.1 API
`IlcAllConnect`	Obsolete
`IlcCard(component type)`	Obsolete
`IlcCardIterator`	`IlcCardPortIterator`
`IlcCardSelect` `  selectCard(IlcPort p)`	`IloCardSelect` `  select(IloConfigurator cfg, IloPort p)`
`IlcCardSelectI` `  selectCard(IlcPort p)`	`IloCardSelectI` `  select(IloConfigurator cfg, IloPort p)`
`IlcCloseType`	`IloClose`

`IlcCompatibility`
`  IlcCompatible`
`  IlcIncompatible`
	`IloCompatibility`
	`  IloCompatible`
	`  IloIncompatible`

`IlcCompatibilityIterator`	`IloCompatibilityIterator`

`IlcComponent`
`  configure()`
`  getConfigurator()`
`  getBaseType()`
`  getIntAttr(const char* name)`
`  getManager()`
`  getName()`
`  getPort(const char* name)`
`  getType()`
`  getTypeVar()`
`  hasProperty(const char* name)`
`  isConfigured()`
`  isWildCard()`
	`IloComponent`
	`  IloGoal IloConfigure(IloComponent,IloConfigStrategy)`
	`  obsolete`
	`  getBaseType()`
	`  getNumAttr(const char* name)`
	`  obsolete`
	`  getName()`
	`  getPort(const char* name)`
	`  getType()`
	`  getTypeAttr()`
	`  hasAttribute(const char* name)`
	`  obsolete`
	`  isWildCard()`

`IlcComponentI`	`IloComponentI`

Configurator 1.0 API	Configurator 2.1 API

IlcComponentType
  add(IlcConstraint ct)
 	
  addCardPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcPropertyMode mode=IlcComputed);
 	
  addCardPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcInt cardValue,
  IlcPropertyMode mode=IlcComputed);
 	
  addCardPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcInt cardMin, IlcInt cardMax,
  IlcPropertyMode mode=IlcComputed);
 	
  addIntAttr(const char* name,
  IlcInt min, IlcInt max,
  IlcPropertyMode mode=IlcComputed);
 	
  addIntAttr(const char* name,
  IlcIntArray values,
  IlcPropertyMode mode=IlcComputed);
 	
  addPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcPropertyMode mode=IlcComputed);
 	
  addPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcInt cardValue,
  IlcPropertyMode mode=IlcComputed);
 	
  addPort(IlcPropertyRole role,
  IlcComponentType aType,
  const char* name,
  IlcInt cardMin, IlcInt cardMax,
  IlcPropertyMode mode=IlcComputed);
 	
  close();
	IloComponentType
	  add(IloConstraint ct)
	
	  addCardPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloAttrMode mode=IloComputed);
	
	  addCardPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloInt cardValue,
	  IloAttrMode mode=IloComputed);
	
	  addCardPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloInt cardMin, IloInt cardMax,
	  IloAttrMode mode=IloComputed);
	
	  addIntAttr(const char* name,
	  IloInt min, IloInt max);
	
	  addIntAttr(const char* name,
	  IloNumArray values);
	
	  addPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloAttrMode mode=IloComputed);
	
	  addPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloInt cardValue,
	  IloAttrMode mode=IloComputed);
	
	  addPort(IloAttrRole role,
	  IloComponentType aType,
	  const char* name,
	  IloInt cardMin, IloInt cardMax,
	  IloAttrMode mode=IloComputed);
	
	  IloConfiguration::close(IloComponentType)
	  IloConfigurator ::close(IloComponentType)

Configurator 1.0 API	Configurator 2.1 API
`configure();`	`IloGoal IloConfigueAll(IloComponentType t, IloConfigurator cfg, IloConfigStrategy st)`
`configure(IlcInstanceSelect selector);`	`IloGoal IloConfigueAll(IloComponentType t, IloConfigurator cfg, IloConfigStrategy st, IloInstanceSelect sel)`
`getConfigurator()`	Obsolete
`getDepth()`	`getDepth()`
`getInstance(const char* name)`	`IloConfiguration::getInstance (IloComponentType t, const char* name)` `IloConfigurator::getInstance (IloComponentType t, const char* name)`
`getIntAttr(const char* name)`	`getNumAttr(const char* name)`
`getInterchangeability()`	Obsolete
`getLabel()`	Obsolete
`getName()`	Obsolete
`getNumberOfInstances()`	`IloConfiguration::getNumberOfInstances (IloComponentType t)` `IloConfigurator::getNumberOfInstances (IloComponentType t)`
`getPort(const char* name)`	`getPort(const char* name)`
`getStrategy()`	Obsolete
`getTypeVar()`	`getTypeAttr()`
`hasProperty(const char* name)`	Obsolete
`isClosed()`	`IloConfiguration::isClosed(IloComponentType t)` `IloConfigurator::isClosed(IloComponentType t)`
`isLeaf()`	`isLeaf()`
`isRoot()`	`isRoot()`
`isSubTypeOf(IlcComponentType aType)`	`isSubTypeOf(IloComponentType aType)`
`isSuperTypeOf(IlcComponentType aType)`	`isSuperTypeOf(IloComponentType aType)`

Configurator 1.0 API	Configurator 2.1 API
`makeInstance(const char* name=0)`	`IloConfiguration::makeInstance(IloComponentType t, const char* name=0)`
`makeInstances(IlcInt number)`	`IloConfiguration::makeInstances(IloComponentType t, IloInt number)` `IloConfigurator::makeInstances(IloComponentType t, IloInt number)`
`printProperties()`	Obsolete
`setIntAttrMax(const char* name, IlcInt max)`	`setNumAttrMax(const char* name, IloNum max)`
`setIntAttrMin(const char* name, IlcInt min)`	`setNumAttrMin(const char* name, IloNum min)`
`setIntAttrValue(const char* name, IlcInt value)`	`setNumAttrValue(const char* name, IloNUm value)`
`setInterchangeability (IlcInterchangeability it)`	Obsolete
`setLabel(const char* label)`	Obsolete
`setPortCardinality(const char* name, IlcInt value)`	`setPortCardinality(const char* name, IloInt value)`
`setPortCardinality(const char* name, const char* subType, IlcInt value)`	`setPortCardinality(const char* name, const char* subType, IloInt value)`
`setPortCardinalityMax (const char* name,IlcInt max)`	`setPortCardinalityMax(const char* name, IloInt max)`
`setPortCardinalityMax(const char* name, const char* subType, IlcInt max)`	`setPortCardinalityMax(const char* name, const char* subType, IloInt max)`
`setPortCardinalityMin(const char* name, IlcInt min)`	`setPortCardinalityMin(const char* name, IloInt min)`
`setPortCardinalityMin(const char* name, const char* subType, IlcInt min)`	`setPortCardinalityMin(const char* name, const char* subType, IloInt min)`
`setStrategy(IlcConfigStrategy aStrategy)`	`IloConfigStrategy::setGoal(IloComponentType t, IloGoal g);`

Configurator 1.0 API	Configurator 2.1 API
`ILC_CONFIG_STRATEGYn`	Obsolete
`IlcConfigStrategy`	Obsolete

`IlcConfigStrategyI`
 	
`classifyInstances(IlcPort p,)`
`IlcBool recursive=IlcTrue)`
 	
`generateCards(IlcPort p)`
 	
`generateCards(IlcPort p,`
`IlcCardSelect selectVar,`
`IlcIntSelect selectValue)`
 	
`getComponent()`
 	
`instantiate(IlcIntAttr x)`
 	
`instantiate(IlcIntAttr x,`
`IlcIntSelect selector)`
 	
`instantiate(IlcTypeVar tvar)`
 	
`instantiate(IlcTypeVar tvar)`
`IlcTypeSelect selector)`
 	
`instantiate(IlcPort p,`
`IlcBool recursive=IlcTrue)`
 	
`instantiate(IlcPort p,`
`IlcConnectionSelect selector,`
`IlcBool recursive=IlcTrue)`
 	
`maxInstantiate(IlcPort p,`
`IlcBool recursive=IlcTrue)`
	Obsolete
	
	`IloGoal IloInstantiate(IloEnv, IloPort)`
	`IloGoal IloDeepInstantiate(IloEnv, IloPort,`
	`IloConfigStrategy)`
	
	`IloGoal IloGenerateCards(IloEnv, IloPort)`
	
	`IloGoal IloGenerateCards(IloEnv, IloPort,`
	`IloCardSelect selectVar,`
	`IlcIntSelect selectValue)`
	
	Obsolete
	
	`IloGoal IloInstantiate(IloEnv, IloNumAttr x)`
	
	`IloGoal IloInstantiate(IloEnv, IloNumAttr x,`
	`IlcIntSelect select)`
	
	`IloGoal IloInstantiate(IloEnv, IloTypeAttr x)`
	
	`IloGoal IloInstantiate(IloEnv, IloTypeAttr x)`
	`IloTypeSelect select)`
	
	`IloGoal IloConnectOne(IloEnv, IloPort p)`
	`IloGoal IloDeepConnectOne(IloEnv, IloPort p,`
	`IloConfigStrategy)`
	
	`IloGoal IloConnectOne(IloEnv, IloPort p,`
	`IloConnectionSelect select)`
	`IloGoal IloDeepConnectOne(IloEnv, IloPort p,`
	`IloConfigStrategy,`
	`IloConnectionSelect)`
	
	`IloGoal IloInstantiate(IloEnv env, IloPort p,`
	`IloMaxSetFirst)`
	`IloGoal IloDeepInstantiate(IloEnv env, IloPort p,`
	`IloMaxSetFirst,`
	`IloConfigStrategy st)`

Configurator 1.0 API	Configurator 2.1 API
`maxInstantiate(IlcPort p, IlcConnectionSelect selector, IlcBool recursive=IlcTrue);`	`IloGoal IloInstantiate(IloEnv env, IloPort p, IloMaxSetFirst, IloConnectionSelect sel)` `IloGoal IloDeepInstantiate(IloEnv env, IloPort p, IloMaxSetFirst, IloConfigStrategy st, IloConnectionSelect sel)`
`minInstantiate(IlcPort p, IlcBool recursive=IlcTrue);`	`IloGoal IloInstantiate(IloEnv env, IloPort p, IloMinSizeFirst)` `IloGoal IloDeepInstantiate(IloEnv env, IloPort p, IloMinSizeFirst, IloConfigStrategy st)`
`minInstantiate(IlcPort p, IlcConnectionSelect selector, IlcBool recursive=IlcTrue);`	`IloGoal IloInstantiate(IloEnv env, IloPort p, IloMinSizeFirst, IloConnectionSelect sel)` `IloDeepInstantiate(IloEnv env, IloPort p, IloMinSizeFirst, IloConfigStrategy st, IloConnectionSelect sel)`
`apply()`	Obsolete
`getManager()`	Obsolete

Configurator 1.0 API	Configurator 2.1 API

IlcConfigurator
  IlcConfigurator(IlcManager m,
  IlcInt cardMax=10000)
 	
  addType(const char* name,
  IlcComponentI* proto=0,
  IlcInt estimatedSize=256)
 	
  addType(const char* name,
  const char* superName)
 	
  addTypeTable(IlcComponentType t1,
  IlcComponentType t2,
  IlcCompatibility compat,
  const char* name=0)
 	
  close()
  error(const char* message)
  existType(const char* name)
  getCardMax()
  getHeap()
  getManager()
  getPhase()
  getType(const char* name)
  getTypeTable(const char* name)
  isLinked()
  link()
  link(ostream& os)
  newPhase()
	IloConfigurator
	  IloConfigurator(IloSolver solver)
	  IloCatalog::setCardMax(IloInt cardMax)
	
	  IloCatalog::addType(const char* name,
	  IloInt estimatedSize=256)
	
	  IloCatalog::addType(const char* name,
	  const char* superName)
	
	  IloCatalog::addTypeTable(IloComponentType t1,
	  IloComponentType t2,
	  IloCompatibility compat,
	  const char* name=0)
	
	  IloConfigurator::close(IloComponentType t)
	  IloConfigError
	  IloCatalog::existType(const char* name)
	  IloCatalog::getCardMax()
	Obsolete
	  getSolver()
	Obsolete
	  IloCatalog::getType(const char* name)
	  IloCatalog::getTypeTable(const char* name)
	  IloCatalog::isBuilt()
	  IloCatalog::build()
	  IloCatalog::build(ostream& os)
	Obsolete

IlcConnect(IlcComponent c, IlcPort aPort)	IloConnect(IloComponent c, IloPort aPort) - in the model IlcConnect(IloComponent c, IlcPort aPort) - in the solver

IlcConnectionSelect
  selectComponent(IlcPort p)
	IloConnectionSelect
	  select(IloConfigurator cfg, IloPort p)

IlcConnectionSelectI
  selectComponent(IlcPort p)
	IloConnectionSelectI
	  select(IloConfigurator cfg, IloPort p)

IlcDomainIteration	IlcDomainIteration
IlcEvalInstance	IloEvalComponent
IlcEvalType	IloEvalType
IlcInstanceIterator	IloInstanceIterator
IlcInstances	Obsolete

Configurator 1.0 API	Configurator 2.1 API

IlcInstanceSelect
  select(IlcComponentType aType)
	IloInstanceSelect
	  select(IloConfigurator cfg,
	  IloInstanceIterator& it)

IlcInstanceSelectI
  select(IlcComponentType aType)
	IloInstanceSelectI
	  select(IloConfigurator cfg,
	  IloInstanceIterator& it)

IlcIntAttr	IloNumAttr (in the model) IlcIntAttr (in the solver)
IlcInterchangeability	IloSubstituability
IlcInterchangeabilityI	IloSubstituabilityI
IlcNotConnect(IlcComponent c, IlcPort aPort)	IloNotConnect(IloComponent c, IloPort aPort) - in the model IlcNotConnect(IloComponent c, IlcPort aPort) - in the solver
IlcNotSpecialize(IlcTypeVar var, IlcComponentType t)	IloNotSpecialize(IloTypeAttr, IloComponentType t) IlcNotSpecialize(IlcTypeAttr, IloComponentType t)
IlcPort	IloPort (in the model) IlcPort (in the solver)
IlcPortDeltaIterator	IlcConnectionDeltaIterator
IlcPortIterator	IloConnectionIterator (in the model) IlcConnectionIterator (in the solver)

IlcPropertyRole
  IlcHasPart
  IlcPartOf
  IlcUses
  IlcIntAttribute
	IloAttrRole
	  IloHasPart
	  IloPartOf
	  IloUses
	  IloNumAttribute

IlcSpecialize(IlcTypeVar var, IlcComponentType t)	IloSpecialize(IloTypeAttr, IloComponentType t) IlcSpecialize(IlcTypeAttr, IloComponentType t)
IlcSubTypeIterator	IloSubTypeIterator
IlcSuperTypeIterator	IloSuperTypeIterator

IlcTypeCompatiblity(IlcTypeVar var,
  IlcPort aPort,
  IlcTypeTable aTable)
	IloTypeCompatiblity(IloTypeAttr var,
	  IloPort aPort,
	  IloTypeTable aTable)

IlcTypeCompatiblity(IlcTypeVar var1,
  IlcTypeVar var2,
  IlcTypeTable aTable)
	IloTypeCompatiblity(IloTypeAttr var1,
	  IloTypeAttr var2,
	  IloTypeTable aTable)

Configurator 1.0 API	Configurator 2.1 API
`IlcTypeIterator`	`IloTypeIterator`
`IlcTypeSelect` `  selectType(IlcTypeVar var)`	`IloTypeSelect` `  select(IloConfigurator cfg, IloTypeAttr var)`
`IlcTypeSelectI` `  selectType(IlcTypeVar var)`	`IloTypeSelectI` `  select(IloConfigurator cfg, IloTypeAttr var)`
`IlcTypeTable`	`IloTypeTable`
`IlcTypeVar`	`IloTypeAttr` (in the model) `IlcTypeAttr` (in the solver)
`IlcTypeVarDeltaIterator`	`IlcTypeAttrDeltaIterator`
`IlcTypeVarIterator`	`IlcTypeAttrIterator`
`IlcWildCard`	`IlcWildCard`

6

Migrating from Planner to Concert Technology

In ILOG Planner 3.3, the functionalities of `IlcLinOpt` fall roughly into two categories:

- those that can be used in edit mode only and correspond to CPLEX basic functionalities,
- those that can be used in search mode only, and program the search and perform linear relaxation with or without constraint propagation.

In Concert Technology, these categories are reflected by two classes.

- for category 1, the class `IloCplex` handles all previous `IlcLinOpt` functionalities,
- for category 2, there is now the class `IloLinConstraint`.

This chapter contains two sections, the first one describing the changes devoted to category 1 use, the second describing the changes devoted to category 2 use.

Translating a Planner Application in Edit Mode

With Planner 3.3, an application in edit mode uses only CPLEX features and not Solver search.

With Concert Technology, an application in edit mode requires the installation of Concert 1.1 and CPLEX 7.1 – Solver is no longer needed for edit mode. At installation, you must include the header file `ilcplex/ilocplex.h`.

Creating the Working Object

The class that handles linear constraints in edit mode is now `IloCplex`. It replaces the class `IlcLinOpt`. The declarations:

```
IlcManager m(IlcEdit);
IlcLinOpt lo(m);
```

are converted to:

```
IloEnv env;
IloCplex cplex(env);
```

Adding Constraints

The member function:

```
void IlcLinearSolver::add(IlcConstraint ct, IlcBool postAll);
```

does not have a direct equivalent in `IloCplex`. In Concert Technology, constraints are added to a model via the memberfunction:

```
IloExtractable IloModel::add(const IloExtractable x) const;
```

See Chapter 2, *Migrating from Solver to Concert Technology*, to find out more about how linear constraints of the type `IlcConstraint` are now created and added to an instance of `IloModel`.

All the linear constraints and the objective function of a model are extracted by the `extract` member function of the class `IloCplex`:

```
void IloCplex::extract(const IloModel model);
```

Setting the Objective Function

Setting and changing the objective function is done in the model in Concert. Consequently, the following lines:

```
IlcFloatVar IlcLinearSolver::setObjMin(IlcFloatExp exp,
                                       IlcBool postAll = IlcFalse) const;

IlcIntVar IlcLinearSolver::setObjMin(IlcIntExp exp,
                                     IlcBool postAll = IlcFalse) const;
```

are now replaced by:

```
model.add(IloMinimize(<concert linear expression>))
```

where `<concert linear expression>` is the conversion of expression `exp` to a Concert expression.

Optimizing

To optimize a problem, `IlcLinOpt` identifies the various LP solvers and the MIP solver by means of the following functions:

```
IlcSolutionStatus IlcLinOpt::primalOpt() const;
IlcSolutionStatus IlcLinOpt::dualOpt() const;
IlcSolutionStatus IlcLinOpt::networkPrimalOpt() const;
IlcSolutionStatus IlcLinOpt::networkDualOpt() const;
IlcSolutionStatus IlcLinOpt::networkPrimalOpt(IlcInt level) const;
IlcSolutionStatus IlcLinOpt::networkDualOpt(IlcInt level) const;
IlcSolutionStatus IlcLinOpt::barrierOpt() const;
IlcSolutionStatus IlcLinOpt::barrierPrimalOpt() const;
IlcSolutionStatus IlcLinOpt::barrierDualOpt() const;
IlcSolutionStatus IlcLinOpt::mipOpt() const;
```

In `IloCplex`, a call to the member function:

```
IloBool IloCplex::solve() const
```

solves the problem by taking into account its type. That is, if the problem is an LP one, the default LP solver is used. If it is a MIP problem, the MIP solver is used. The optimization status can be obtained with the member function:

```
IloAlgorithm::Status IloCplex::getStatus().
```

To force an LP optimization of a MIP problem, the user must specify the LP solver with the function:

```
void IloCplex::setRootAlgorithm(IloCplex::Algorithm alg);
```

where `alg` can be one of the values in the following enumeration:

```
enum IloCplex::Algorithm {
  IloCplex::Primal,
  IloCplex::Dual,
  IloCplex::Barrier,
  IloCplex::NetworkPrimal,
  IloCplex::NetworkDual,
  IloCplex::DualBarrier
};
```

Then the relaxed problem is solved with the function:

```
IloBool IloCplex::solveRelaxed();
```

For example, the code:

```
IlcManager m(IlcEdit);
IlcLinOpt lo(m);
...
IlcSolutionStatus status = lo.primalOpt();
```

is translated into:

```
IloEnv env;
IloCplex cplex(env);
...
cplex.setRootAlgorithm(IloCplex::Primal);
cplex.solveRelaxed();
IloAlgorithm::Status status = cplex.getStatus();
```

Getting the Results

The Planner 3.3 member functions to obtain the value of an expression:

```
IlcFloat IlcLinearSolver::getCurrentValue(IlcFloatExp var) const;
IlcFloat IlcLinearSolver::getCurrentValue(IlcIntExp var) const;
```

are converted in Concert Technology to:

```
IloNum IloCplex::getValue(const IloExpr e) const;
```

Since the classes `IloExpr` and `IloNumVar` are unrelated, we also have:

```
IloNum IloCplex::getValue(const IloNumVar x) const;
```

Similarly, the memberfunctions:

```
IlcFloat IlcLinearSolver::getReducedCost(IlcFloatVar var) const;
IlcFloat IlcLinearSolver::getReducedCost(IlcIntVar var) const;
```

are converted to:

```
IloNum IloCplex::getReducedCost(const IloNumVar x) const;
```

Also the member function:

```
IlcFloat IlcLinOpt::getDualValue(IlcConstraint c) const;
```

is translated to the member functions:

```
IloNum IloCplex::getDual(const IloRange range);
```

Similarly, the member function:

```
IlcFloat IlcLinOpt::getSlackValue(IlcConstraint c) const;
```

is translated to the member function:

```
IloNum IloCplex::getSlack(const IloRange range);
```

To obtain the objective value, the member function:

```
IlcFloat IlcLinOpt::getObjValue() const;
```

is converted to:

```
IloNum IloCplex::getObjValue() const;
```

The correspondence between Planner 3.3 and Concert/CPLEX functions for obtaining statistics is given in the following table:

Planner 3.3 Functions	Concert 1.1 and CPLEX 7.1 Functions
`IlcInt IlcLinOpt::getNbOfConstraints() const;`	`IloInt IloCplex::getNrows() const;`
`IlcInt IlcLinOpt::getNbOfVars() const;`	`IloInt IloCplex::getNcols() const;`
`IlcInt IlcLinOpt::getNbOfNonZeros() const;`	`IloInt IloCplex::getNnz() const;`
`IlcInt IlcLinOpt::getNbOfIterations() const;`	`IloInt IloCplex::getNiterations() const;`

Exceptions

The call to the member function:

```
void IlcLinOpt::useExceptions(IlcBool useIt = IlcTrue) const;
```

has no equivalent in `IloLinConstraint` since exceptions are now always used in Concert and CPLEX. The class `IlcLinOptException` of exceptions thrown in Planner 3.3 is now replaced by `IloException`, documented in the *ILOG Concert Technology Reference Manual*.

Bound Changes

In Planner 3.3, bounds are changed by means of the member functions:

```
void IlcLinOpt::chgBounds(IlcIntVar x, IlcInt min, IlcInt max) const;
void IlcLinOpt::chgBounds(IlcFloatVar x, IlcFloat min, IlcFloat max) const;
void IlcLinOpt::chgLowerBound(IlcIntVar x, IlcInt min) const;
void IlcLinOpt::chgLowerBound(IlcFloatVar x, IlcFloat min) const;
void IlcLinOpt::chgUpperBound(IlcIntVar x, IlcInt max) const;
void IlcLinOpt::chgUpperBound(IlcFloatVar x, IlcFloat max) const;
```

In Concert and CPLEX, bounds are changed directly on the variables of the model by means of the member functions:

```
void IloNumVar::setLb(IloNum lb) const;
void IloNumVar::setUb(IloNum ub) const;
```

Constraint Removal

You can remove a constraint from a problem in an instance of `IlcLinOpt` with the member function:

```
void IlcLinOpt::remove(const IlcConstraint& ct) const;
```

In Concert and CPLEX, this operation is performed directly on the model by the member function:

```
void IloModel::remove(const IloExtractable x) const;
```

Parameter Handling

To change a parameter on an instance of `IlcLinOpt`, the following member functions, that depend on the parameter type, are used:

```
void IlcLinOpt::setParam(IlcBoolParam, IlcBool) const;
void IlcLinOpt::setParam(IlcIntParam, IlcInt) const;
void IlcLinOpt::setParam(IlcFloatParam, IlcFloat) const;
void IlcLinOpt::setParam(IlcStringParam, const char*) const;
```

In Concert and CPLEX, this is done with the member functions:

```
void IloCplex::setParam(BoolParam   parameter, IloBool value);
void IloCplex::setParam(IntParam    parameter, IloInt value);
void IloCplex::setParam(NumParam  parameter, IloNum value);
void IloCplex::setParam(StringParam parameter, const char* value);
```

The Planner 3.3 member function to reset all parameters to their default values:

```
void IlcLinOpt::setParamsDefault() const;
```

is now, in Concert and CPLEX:

```
void IloCplex::setDefaults();
```

The member functions that read parameters:

```
IlcBool IlcLinOpt::getParam(IlcBoolParam) const;
IlcInt IlcLinOpt::getParam(IlcIntParam) const;
IlcFloat IlcLinOpt::getParam(IlcFloatParam) const;
char* IlcLinOpt::getParam(IlcStringParam, char* buffer) const;
```

are now called:

```
IloBool     IloCplex::getParam(BoolParam   parameter) const;
IloInt      IloCplex::getParam(IntParam    parameter) const;
IloNum      IloCplex::getParam(NumParam    parameter) const;
const char* IloCplex::getParam(StringParam parameter) const;
```

To read default values and the range of values of parameters, the member functions:

```
IlcBool IlcLinOpt::getParamDefault(IlcBoolParam) const;
IlcInt IlcLinOpt::getParamDefault(IlcIntParam) const;
IlcFloat IlcLinOpt::getParamDefault(IlcFloatParam) const;
char* IlcLinOpt::getParamDefault(IlcStringParam, char* buffer) const;
IlcInt IlcLinOpt::getParamMin(IlcIntParam) const;
IlcFloat IlcLinOpt::getParamMin(IlcFloatParam) const;
IlcInt IlcLinOpt::getParamMax(IlcIntParam) const;
IlcFloat IlcLinOpt::getParamMax(IlcFloatParam) const;
```

are now called:

```
IloBool     IloCplex::getDefault(BoolParam   parameter) const;
IloInt      IloCplex::getDefault(IntParam    parameter) const;
IloNum      IloCplex::getDefault(NumParam  parameter) const;
const char* IloCplex::getDefault(StringParam parameter) const;
IloInt      IloCplex::getMin(IntParam    parameter) const;
IloNum      IloCplex::getMin(NumParam  parameter) const;
IloInt      IloCplex::getMax(IntParam    parameter) const;
IloNum      IloCplex::getMax(NumParam  parameter) const;
```

Basis Handling

In Planner 3.3, a basis is a mapping between variables and constraints in the `IlcLinOpt` object on one hand and a status on the other hand.

In Concert 1.1 and CPLEX 7.1, a basis is represented by two arrays of status values in instances of:

```
IloCplex::BasisStatusArray
```

one for the variables and one for the constraints.

To get or to fill a basis from an instance of `IlcLinOpt`, these member functions are called:

```
IlcBasis IlcLinOpt::getBasis() const;
void IlcLinOpt::fillBasis(IlcBasis) const;
```

To get a basis from an instance of `IloCplex`, create two arrays of basis status values (`cstat` for the variables and `rstats` for the ranges) and fill them by means of the member function:

```
void IloCplex::getStatuses(BasisStatusArray cstat, const IloNumVarArray var,
              BasisStatusArray rstat, const IloRangeArray con) const;
```

To set a basis for an instance of `IlcLinOpt`, this member function is called:

```
void IlcLinOpt::setBasis(IlcBasis)const;
```

To set a basis for an instance of `IloCplex`, the two arrays of basis status values (`cstat` for the variables and `rstats` for the ranges) must be given to the following member function:

```
void IloCplex::setStatuses(const BasisStatusArray cstat,
                           const IloNumVarArray var,
                           const BasisStatusArray rstat,
                           const IloRangeArray con);
```

Handling Problem Types

There is no problem type handling in `IloCplex` as there was in `IlcLinOpt`. The following member function lets you know whether the problem extracted is a MIP:

```
IloBool IloCplex::isMIP() const;
```

You can solve the relaxed problem with the member function:

```
IloBool IloCplex::solveRelaxed();
```

You can solve the problem where all the integer variables are fixed to their current values with the member function:

```
IloBool IloCplex::solveFixed();
```

Output

To output a file in a CPLEX format such as `.lp`, `.mps` or `.sav`, the member function:

```
void IlcLinOpt::writeProblem(const char* filename) const;
```

is converted to:

```
void IloCplex::exportModel(const char* filename) const;
```

Priority Orders

In Planner 3.3, it is possible to specify a priority order for branching on variables with the member function:

```
void setPriOrder(IlcIntVarArray vars,
IlcIntArray priorities,
IlcIntArray directions) const;
```

where `priorities` contains the priorities for the variables in the array `vars` and `directions` contains the branching directions for the variables in `vars`.

In Concert and CPLEX, priorities for an array of variables are specified with the member function:

```
void IloCplex::setPriorities(const IloNumVarArray var, const IloNumArray pri);
```

and the branching directions are specified with the member function:

```
void IloCplex::setDirections
                 (const IloNumVarArray var, const BranchDirectionArray dir);
```

The branching directions are translated in the following way:

`IlcBranchGlobal`	becomes	`IloCplex::BranchGlobal`
`IlcBranchUp`	becomes	`IloCplex::BranchUp`
`IlcBranchDown`	becomes	`IloCplex::BranchDown`

Callbacks

In Planner 3.3, callbacks are defined with the macro `ILCCALLBACK`; its body is the main code to execute. Then callback is set to the `IloLinOpt` instance as an LP callback with a call to the member function:

```
void IloLinOpt::setLPCallback(IlcCallBack)
```

or as a MIP callback with a call to the member function:

```
void IloLinOpt::setMIPCallback(IlcCallBack)
```

In Concert 1.1 and CPLEX 7.1, callbacks are defined in a similar way. However, there are several types of callbacks:

- An LP callback is defined with the macro `ILOLPCALLBACK`.
- A MIP callback is defined with the macro `ILOMIPCALLBACK`.
- More specific callbacks can be defined with other macros. For instance:
  - `ILOPRESOLVECALLBACK` defines a callback called from the `IloCplex` presolve routine.
  - `ILOCROSSOVERCALLBACK` defines a callback called from the `IloCplex` crossover routine.

Callbacks are then set to the `IloCplex` instance with a call to the member function:

```
IloCplex::Callback use(IloCplex::Callback);
```

Planner 3.3 provides the following member functions to read information from the code of a callback:

```
IlcFloat IlcLinOpt::getCallbackInfo(IlcFloatCallbackInfo whichInfo) const;
IlcInt   IlcLinOpt::getCallbackInfo(IlcIntCallbackInfo whichInfo) const;
```

where `IlcFloatCallbackInfo` and `IlcIntCallbackInfo` are enumerated types of information that can be read.

In Concert and CPLEX, each piece of information is obtained with a specific member function of `IloCplex::LPCallback` and `IloCplex::MIPCallback`.

With regard to information available in LP callbacks, the following table maps the Planner 3.3 enumerated type and the Concert and CPLEX member function.

Planner 3.3	Concert 1.1 and CPLEX 7.1
`IlcCallbackInfoPrimalObj`	`IloNum IloCplex::LPCallback::getObjValue()`
`IlcCallbackInfoDualObj`	`IloNum IloCplex::LPCallback::getObjValue()`
`IlcCallbackInfoPrimalInfMeas`	`IloNum IloCplex::LPCallback::getInfeasibility() const`
`IlcCallbackInfoDualInfMeas`	`IloNum IloCplex::LPCallback::getInfeasibility() const`
`IlcCallbackInfoPrimalFeas`	`IloBool IloCplex::LPCallback::isFeasible() const`
`IlcCallbackInfoDualFeas`	`IloBool IloCplex::LPCallback::isFeasible() const`
`IlcCallbackInfoItCount`	`IloInt IloCplex::LPCallback::getNiterations() const`

Crossover information is accessible only in crossover callbacks defined with the macro ILOCROSSOVERCALLBACK. The mapping is:

Planner 3.3	Concert 1.1 and CPLEX 7.1
IlcCallbackInfoCrossoverPPush	IloInt IloCplex::CrossoverCallback::getNprimalPushes() const
IlcCallbackInfoCrossoverPExch	IloInt IloCplex::CrossoverCallback::getNprimalExchanges() const
IlcCallbackInfoCrossoverDPush	IloIn IloCplex::CrossoverCallback::getNdualPushes() const
IlcCallbackInfoCrossoverDExch	IloInt IloCplex::CrossoverCallback::getNdualExchanges() const
IlcCallbackInfoCrossoverSbCnt	IloInt IloCplex::CrossoverCallback::getNsuperbasics() const

For information available in MIP callbacks the mapping is:

Planner 3.3	Concert 1.1 and CPLEX 7.1
IlcCallbackInfoNodeCount	IloInt IloCplex::MIPCallback::getNnodes() const
IlcCallbackInfoNodesLeft	IloInt IloCplex::MIPCallback::getNremainingNodes() const
IlcCallbackInfoMIPIterations	IloInt IloCplex::MIPCallback::getNiterations() const
IlcCallbackInfoCliqueCount	IloInt IloCplex::MIPCallback::getNcliques() const
IlcCallbackInfoCoverCount	IloInt IloCplex::MIPCallback::getNcovers() const
IlcCallbackInfoBestInteger	IloNum IloCplex::MIPCallback::getIncumbentObjValue() const;
IlcCallbackInfoBestRemaining	IloNum IloCplex::MIPCallback::getBestObjValue() const;
IlcCallbackInfoCutoff	IloNum IloCplex::MIPCallback::getCutoff() const;

Presolve information is available only in the presolve callback defined with the macro ILOPRESOLVECALLBACK. The mapping is:

Planner 3.3	Concert 1.1 and CPLEX 7.1
IlcCallbackInfoPresolveRowsGone	IloInt IloCplex::PresolveCallback::getNremovedRows() const
IlcCallbackInfoPresolveColsGone	IloInt IloCplex::PresolveCallback::getNremovedCols() const
IlcCallbackInfoPresolveAggSubst	IloInt IloCplex::PresolveCallback::getNaggregations() const
IlcCallbackInfoPresolveCoeffs	IloInt IloCplex::PresolveCallback::getNmodifiedCoeffs() const

Translating a Planner Application in Search Mode

In Planner 3.3, an application in search mode uses CPLEX features and Solver search.

With Concert Technology, the same application requires the installation of Concert 1.1, CPLEX 7.1, and Solver 5.1. At installation, you must include the header file `ilsolver/ilohybrid.h`.

Creating the Working Object

The class that handles linear constraints in search mode is now `IloLinConstraint`. It replaces `IlcLinOpt`. The declarations:

```
IlcManager m(IlcEdit);
IlcLinOpt lo(m);
```

are replaced in Concert 1.1 and Solver 5.1 by:

```
IloEnv env;
IloSolver solver(env);
IloLinConstraint lc(solver);
```

Adding Constraints

The member function:

```
void IlcLinearSolver::add(IlcConstraint ct, IlcBool postAll);
```

does not have a direct equivalent in `IloLinConstraint`. Constraints are added to a model using the associated instance of `IloSolver`. We distinguish two cases: the call of this function in edit mode and the call in search mode.

- In the first case, the function call is part of the model definition and is equivalent to the addition of a constraint to a model via the member function:

  ```
  IloExtractable IloModel::add(const IloExtractable x) const;
  ```

  See Chapter 2, *Migrating from Solver to Concert Technology*, to learn more about how linear constraints of type `IlcConstraint` are now created and added to an instance of `IloModel`. All linear constraints of a model are extracted to each instance of `IloLinConstraint` created with a solver when the following member function is called:

  ```
  void IloSolver::extract(const IloModel model) const;
  ```

- In the second case, the constraint is created in search mode, and it must be of type `IlcConstraint`. A linear constraint of a model is added to each instance of `IloLinConstraint` created with a solver when the following member function is called:

  ```
  void IloSolver::add(IlcConstraint ct);
  ```

The `postAll` argument determines whether the constraint was added to Solver and Planner (`postAll = IlcTrue`) or to Planner only (`postAll = IlcFalse`). With `IloLinConstraint`, constraints are, by default, added to `IloSolver` and to `IloLinConstraint`. To force a constraint to be added to `IloLinConstraint` only, use the member function:

```
void IloLinConstraint::setSynchronization(IloLinConstraint::Synchronization s);
```

When `s = IloLinConstraint::LinConstraintOnly`, constraints are extracted and added to the instance of `IloLinConstraint` but not to the instance of `IloSolver`. When `s = LinConstraintAndSolver`, constraints are added to both instances.

Setting the Objective Function

In ILOG Concert Technology, setting and changing the objective function is done in the model. So, the function equivalent to:

```
IlcFloatVar IlcLinearSolver::setObjMin(IlcFloatExp exp,
                                       IlcBool postAll = IlcFalse) const;

IlcIntVar IlcLinearSolver::setObjMin(IlcIntExp exp,
                                     IlcBool postAll = IlcFalse) const;
```

is now replaced by:

```
model.add(IloMinimize(<concert linear expression>))
```

where `<concert linear expression>` is the translation of expression `exp` as a Concert expression.

When you set an objective function in Planner 3.3, a constraint is added to Solver to equalize the objective function and the variable returned by the `setObjMax` or `setObjMin` function. If `postAll = IlcTrue`, this constraint is also added to Solver.

For this constraint to be added to Solver in Concert Technology, it is now necessary to call the function:

```
void IloLinConstraint::setSynchronization(IloLinConstraint::Synchronization s);
```

with `s = LinConstraintAndSolver` before extracting the model.

Note that `IloLinConstraint` allows you to change the objective function in a reversible way with the member functions:

```
void IloLinConstraint::setObjMin(const IlcFloatExp e, IloBool propagate = IloTrue);
void IloLinConstraint::setObjMin(const IlcIntExp e, IloBool propagate = IloTrue);
```

The translation is similar for the member functions:

```
IlcFloatVar IlcLinearSolver::setObjMax(IlcFloatExp exp,
                                       IlcBool postAll = IlcFalse) const;

IlcIntVar IlcLinearSolver::setObjMax(IlcIntExp exp,
                                     IlcBool postAll = IlcFalse) const;
```

Getting Results

The member functions:

```
IlcFloat IlcLinearSolver::getCurrentValue(IlcFloatExp var) const;
IlcFloat IlcLinearSolver::getCurrentValue(IlcIntExp var) const;
```

are converted to:

```
IloNum IloLinConstraint::getValue(const IloExpr e) const;
```

Since the classes `IloExpr` and `IloNumVar` are unrelated, we also have:

```
IloNum IloLinConstraint::getValue(const IloNumVar x) const;
```

Observe that for `Ilc...` expressions created during search we also have:

```
IloNum IloLinConstraint::getValue(const IlcIntExp e) const;
IloNum IloLinConstraint::getValue(const IlcFloatExp e) const;
```

Similarly, the member functions:

```
IlcFloat IlcLinearSolver::getReducedCost(IlcFloatVar var) const;
IlcFloat IlcLinearSolver::getReducedCost(IlcIntVar var) const;
```

can be converted using one of these member functions:

```
IloNum IloLinConstraint::getReducedCost(const IlcIntVar x) const;
IloNum IloLinConstraint::getReducedCost(const IlcFloatVar x) const;
IloNum IloLinConstraint::getReducedCost(const IloNumVar x) const;
```

Also the member function:

```
IlcFloat IlcLinOpt::getDualValue(IlcConstraint c) const;
```

is translated using the member functions:

```
IloNum IloLinConstraint::getDual(const IlcConstraint ct);
IloNum IloLinConstraint::getDual(const IloRange range);
```

Similarly, the member function:

```
IlcFloat IlcLinOpt::getSlackValue(IlcConstraint c) const;
```

is translated using the member functions:

```
IloNum IloLinConstraint::getSlack(const IlcConstraint ct);
IloNum IloLinConstraint::getSlack(const IloRange range);
```

To get the objective value, the member function:

```
IlcFloat IlcLinOpt::getObjValue() const;
```

is translated to:

```
IloNum IloLinConstraint::getObjValue() const;
```

The following table gives the mappings between new and previous member functions for obtaining statistics.

Planner 3.3	Concert 1.1 and CPLEX 7.1
`IlcInt IlcLinOpt::getNbOfConstraints() const;`	`IloInt IloLinConstraint::getNrows() const;`
`IlcInt IlcLinOpt::getNbOfVars() const;`	`IloInt IloLinConstraint::getNcols() const;`
`IlcInt IlcLinOpt::getNbOfNonZeros() const;`	`IloInt IloLinConstraint::getNnz() const;`
`IlcInt IlcLinOpt::getNbOfIterations() const;`	`IloInt IloLinConstraint::getNiterations() const;`
`IlcInt IlcLinOpt::getMemoryUsed() const;`	`IloInt IloLinConstraint::getMemoryUsage() const;`

Optimization Algorithms

In Planner 3.3, the member function for setting the root node algorithm is:

```
void IlcLinOpt::setFirstMethod(IlcOptMethod method, IlcInt level = 1) const;
```

where `method` is one of the enumerated values:

```
  enum IlcOptMethod {
  IlcPrimal,
  IlcDual,
  IlcNetworkPrimal,
  IlcNetworkDual,
  IlcBarrierPrimal,
  IlcBarrierDual,
};
```

In Concert 1.1 and Solver 5.1, it is:

```
void IloLinConstraint::setRootAlgorithm(IloCplex::Algorithm algo,
                                        IloInt level = 1);
```

where `algo` is one of the enumerated values:

```
  enum IloCplex::Algorithm {
    IloCplex::Primal,
    IloCplex::Dual,
    IloCplex::Barrier,
    IloCplex::NetworkPrimal,
    IloCplex::NetworkDual,
    IloCplex::DualBarrier
  };
```

The correspondence between new and old names is the following:

Planner 3.3	Concert 1.1 and CPLEX 7.1
IlcPrimal	loCplex::Primal
IlcDual	IloCplex::Dual
IlcNetworkPrimal	IloCplex::NetworkPrimal
IlcNetworkDual	IloCplex::NetworkDual
IlcBarrierPrimal	IloCplex::BarrierPrimal
IlcBarrierDual	IloCplex::BarrierDual

The member function:

```
void IlcLinOpt::setSearchMethod(IlcOptMethod method) const;
```

is now:

```
void IloLinConstraint::setNodeAlgorithm(IloCplex::Algorithm algo);
```

Precision Handling

Precision is handled the same way in `IloLinConstraint` as in `IlcLinOpt`. However, the name for accessing the tolerance value for integrality checking and for the rounding procedure has changed. The member function:

```
IlcFloat IlcLinOpt::getFuzzValue() const
```

is converted to:

```
IloNum IloLinConstraint::getFeasTolerance() const;
```

and the member function:

```
void IlcLinOpt::setFuzzvalue(IlcFloat p)
```

is converted to:

```
void IloLinConstraint::setFeasTolerance(IloNum tol) const;
```

Variable Instantiation

The Planner 3.3 member function:

```
void IlcLinOpt::trySolution()
```

used to instantiate all variables appearing in the invoking object to their current value is now:

```
void IloLinConstraint::fixAllVarsToValue()
```

Note that Concert also supports member functions that enable you to fix variables, and arrays of variables, independently.

Exceptions

The call to the Planner 3.3 member function:

```
void IlcLinOpt::useExceptions(IlcBool useIt = IlcTrue) const;
```

has no equivalent in `IloLinConstraint` since exceptions are always used in Concert 1.1 and Solver 5.1.

The exceptions thrown from the Planner 3.3 class `IlcLinOptException` are now replaced by `IloException`.

Goals

The member function:

```
IlcGoal IlcLinOpt::getPiecewiseLinearGenerateGoal() const;
```

is now replaced by the member function:

```
IloGoal IloGoalPiecewiseLinear(IloEnv env, IloLinConstraint lc);
```

that creates a Concert goal, and by:

```
IlcGoal IloGoalPiecewiseLinear(IloSolver, IloLinConstraint lc);
```

that creates a Solver goal for enumerating on segments of a piecewise linear expression which has been extracted from the model.

Miscellaneous Functions

The following rounding functions from Planner 3.3 have a direct translation in Concert 1.1 and Solver 5.1.

Planner 3.3	Concert 1.1 and CPLEX 7.1
`IlcInt IlcLinearSolver::nearest(IlcFloat x) const;`	`IloNum IloLinConstraint::nearest(IloNum a) const;`
`IlcInt IlcLinearSolver::trunc(IlcFloat x) const;`	`IloNum IloLinConstraint::trunc(IloNum a) const;`
`IlcFloat IlcLinearSolver::frac(IlcFloat x) const;`	`IloNum IloLinConstraint::frac(IloNum a) const;`
`IlcFloat IlcLinearSolver::distToInt(IlcFloat x) const;`	`IloNum IloLinConstraint::` `distToInt(IloNum a) const;`
`IlcFloat IlcLinearSolver::floor(IlcFloat x) const;`	`IloNum IloLinConstraint::floor(IloNum a) const;`
`IlcFloat IlcLinearSolver::ceil(IlcFloat x) const;`	`IloNum IloLinConstraint::ceil(IloNum a) const;`

The member functions for print information on a stream:

```
void IlcLinearSolver::printInformation() const;
void IlcLinearSolver::printInformation(ostream& str) const;
```

are translated to:

```
void IloLinConstraint::printInformation(ostream& stream) const;
void IloLinConstraint::printInformation() const;
```

Variable Selection

The selection functions over an array of integer variables that were defined by the macro:

```
IlcChooseMixedIndex1(name, criterion, type)
```

can be defined by the more general macro `ILOSELECTVARn`. For example, the macro:

```
IlcChooseMixedIndex1(MostNotInteger, -linsolver.distToInt(val), IlcIntVar)
```

defines the function `MostNotInteger(IlcIntVarArray x, IloLinOpt lo)`. This function selects the variable that has the "least integer" value. The same function can be defined with the macro `ILOSELECTVARn`:

```
ILOSELECTVAR2(MostNotInteger, IloNumVar, x, IloLinConstraint, lc){
  IloNum dist = lc.distToInt(lc.getValue(x));
  return (dist > lc.getFeasTolerance()) ? dist : -IlcInfinity;
}
```

Index

T

V