IBM ILOG CPLEX Optimization Studio CP Optimizer Extensions User's Manual

Version 12 Release 6.0

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, and ILOG are trademarks or registered trademarks of International Business Machines Corp., in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Contents

Chapter 1. Welcome to CP Optimizer Extensions 1
Overview 1
About this manual 1
Prerequisites 2
Related documentation 2
Installing CP Optimizer 2
Typographic and naming conventions 3

Chapter 2. Using constraint propagators 5
Overview 5
Simplest propagator: writing a checker 5
Invariants: what to propagate? 8
Elementary modifiers for variables 8
Defining a new class of constraint propagator . . . 9
Example of a propagator 11
Programming tips 12

Chapter 3. Using engine extensions . . 13
Overview 13
When to use engine extensions 13
Model extraction 13
Overview 13
Extraction process 14
Engine representation of variables 14
Engine representation of constraints 15
Preprocessing 15
Engine extension API classes 16
Overview 16
Implementation classes and handles 16
Variables and expressions 16
Constraints and constraint propagation 17
Goals 18

Chapter 4. Writing complex custom constraints 21
Overview 21
Understanding constraints 21
Elementary modifiers for decision variables . . . 22
Invariants: what to propagate? 23
The constraint propagation algorithm 24
Writing your own constraint 25
Overview 25
Implementation classes and handles 26
Using ILOCPCONSTRAINTWRAPPERn to wrap constraints 27
Writing your own logical constraint 28
Using demons to propagate selectively 29
Using delta-domains to propagate efficiently . . 30
Programming tips 32
Exhaustive propagation 32
Pushing constraints 33

Chapter 5. Writing goals 37
Overview 37
Understanding goals 37
Overview 37
Using macros to create a simple goal 38
Composing goals 39
Goal stack 40
Choice points 41
Backtracking 41
Subgoals 42
Writing your own goal 42
Overview 42
Using ILCGOALn to define a new class of goals 43
Using ILOCPGOALWRAPPER to wrap the goals 44
Example of writing your own goal: implementing IlcInstantiate 45
Programming tips 46
Arguments to goals 46
Reversible data 47
Fixing all decision variables 48

Chapter 6. Writing custom scheduling constraints and goals 49
Overview 49
Working with interval variables 49
Writing a goal with interval variables 50
Writing a constraint with interval variables 50
Writing a greedy search 51
Writing a chronological scheduling goal 52
Writing a chronological scheduling goal on sequence variable 53
Writing a custom constraint on a sequence variable 53

Index 55

Chapter 1. Welcome to CP Optimizer Extensions

This is the *CP Optimizer Extensions User's Manual*.

Overview

CP Optimizer is a software library which provides a constraint programming engine.

The CP Optimizer feature of the IBM® ILOG® CPLEX® Optimizers is a software library which provides a constraint programming engine targeting both constraint satisfaction problems and optimization problems. This engine, designed to be used in a "model & run" development process, contains powerful methods for finding feasible solutions and improving them. The strength of the optimizer generally removes the need for you to write and maintain a search strategy.

CP Optimizer is based on IBM ILOG Concert Technology. Concert Technology offers a library of classes and functions that enable you to define models for optimization problems. Likewise, CP Optimizer offers a library of classes and functions that enable you to find solutions to the models. Though the CP Optimizer defaults will prove sufficient to solve most problems, CP Optimizer offers a variety of tuning classes and parameters to control various algorithmic choices.

IBM ILOG Concert Technology and CP Optimizer provide application programming interfaces (APIs) for Microsoft .NET languages, C++ and Java™. The CP Optimizer part of an application can be completely integrated with the rest of that application (for example, the graphical user interface, connections to databases and so on) because it can share the same objects.

The recommended development path for solving an optimization problem with CP Optimizer is to use IBM ILOG Concert Technology for modeling, together with the CP Optimizer search engine tuned with parameters and search phases. In this way, you benefit from all future improvements to the CP Optimizer propagation and search strategies.

While the modeling objects provided by IBM ILOG Concert Technology and the algorithms provided by the CP Optimizer engine will generally be sufficient to model and solve your problem, in some situations you may need the full flexibility of writing and maintaining the solution process in various ways, such as writing custom constraints and custom constructive search by means of goals. The extensions portion of CP Optimizer is not available in all APIs.

About this manual

The *CP Optimizer Extensions User's Manual* provides conceptual information about the advanced features of CP Optimizer.

This is the *CP Optimizer Extensions User's Manual*. It offers explanations of how to use the advanced features of CP Optimizer effectively. All of the CP Optimizer functions and classes used in this manual are documented in the *CP Optimizer*

Reference Manuals. As you study this manual, you will probably consult the appropriate reference manual from time to time, as it contains precise details on classes and their members.

Prerequisites

The *CP Optimizer Extensions User's Manual* assumes that you have a working knowledge of one of the programming languages of the available APIs and have installed CP Optimizer. The manual also assumes that you have a working knowledge of CP Optimizer.

CP Optimizer requires a working knowledge of the .NET Framework, C++ or Java. However, it does not require you to learn a new language since it does not impose any syntactic extensions on your programming language of choice.

Before using this manual, you should have some experience using CP Optimizer. You should have CP Optimizer and IBM ILOG Concert Technology installed in your development environment before starting to use this manual. Moreover, you should be able to compile, link and execute a sample program provided with CP Optimizer.

Related documentation

The *CP Optimizer Extensions User's Manual* is part of a collection of manuals. You will likely need to refer to the other manuals in the collection as you use this manual.

The following documentation ships with CP Optimizer and will be useful for you to refer to as you use this manual.

- The *Getting Started with CP Optimizer Manual* introduces CP Optimizer with tutorials that lead you through describing, modeling and solving problems.
- The *CP Optimizer User's Manual* explains the topics covered in the *Getting Started with CP Optimizer Manual* in greater depth, with individual chapters about modeling, solving and tuning the CP Optimizer search process.
- The *CP Optimizer Reference Manuals* document the IBM ILOG Concert Technology and CP Optimizer classes and functions used in the *CP Optimizer Extensions Manual*. The reference manuals also explain certain concepts more formally. There are three reference manuals; one for each of the available APIs.
- The *Release Notes® for CP Optimizer* list new and improved features, changes in the library and documentation and issues addressed for each release.

Installing CP Optimizer

The installation instructions for CP Optimizer are in the Electronic Product Delivery package.

In this manual, it is assumed that you have already successfully installed the IBM ILOG Concert Technology and CP Optimizer libraries on your platform (that is, the combination of hardware and software you are using). If this is not the case, you will find installation instructions in your IBM ILOG Electronic Product Delivery package. The instructions cover all the details you need to know to install IBM ILOG Concert Technology and CP Optimizer on your system.

Typographic and naming conventions

The typographic and naming conventions used in the *CP Optimizer Extensions User's Manual* follow the standard practices of the programming language being used.

Important ideas are *italicized* the first time they appear.

In this manual, the examples are given in C++ and Java. In the **C++ API**, the names of types, classes and functions defined in the IBM ILOG Concert Technology and CP Optimizer libraries begin with `Ilo`.

The name of a class is written as concatenated words with the first letter of each word in upper case (that is, capital). For example,

```
IloIntVar
```

A lower case letter begins the first word in names of arguments, instances and member functions. Other words in the identifier begin with an uppercase letter. For example,

```
IloIntVar aVar;
IloIntVarArray::add;
```

Names of data members begin with an underscore, like this:

```
class Bin {
public:
  IloIntVar       _type;
  IloIntVar       _capacity;
  IloIntVarArray  _contents;
  Bin (IloModel    model,
       IloIntArray capacity,
       IloInt      nTypes,
       IloInt      nComponents);
  void display(const IloCP cp);
};
```

Generally, accessors begin with the key word `get`. Accessors for Boolean members begin with `is`. Modifiers begin with `set`.

Names of classes, methods and symbolic constants in the **Java API** correspond very closely to those in the **C++ API; however in** the **Java API**, the namespaces `ilog.cp` and `ilog.concert` are used.

To make porting easier from platform to platform, IBM ILOG Concert Technology and CP Optimizer isolate characteristics that vary from system to system.

For that reason, you are encouraged to use the following identifiers for basic types in C++:

- `IloInt` stands for signed long integers;
- `IloNum` stands for double precision floating-point values ;
- `IloBool` stands for Boolean values: `IloTrue` and `IloFalse`.

You are not obliged to use these identifiers, but it is highly recommended if you plan to port your application to other platforms.

Chapter 2. Using constraint propagators

A constraint propagator is a simple way to define a custom constraint, that is, one where you define the domain reduction rules.

Overview

A constraint propagator is a simple way to define a custom constraint, that is, one where you define the domain reduction rules.

CP Optimizer offers a wide range of predefined constraints, together with powerful logical operators for combining them. These facilities usually suffice for expressing even the most specific model requirements. However, when modeling and solving your problem, you may find that it is necessary or useful to write a constraint propagator. A constraint propagator is a simple way to define a custom constraint, that is, one where you define the domain reduction rules via some C++ or Java code.

Some cases in which you might find it useful to write a constraint propagator are when a predefined constraint which implements your knowledge of the problem does not exist, when using a constraint propagator may provide the opportunity to perform stronger inferences (domain reductions within the propagation engine) than CP Optimizer would typically perform, or when running time needs improvement.

Before writing a constraint propagator in these cases, you should attempt to model the problem using the compatibility constraints (such as `IloAllowedAssignments`) or to use an alternate formulation. If these options are not practical or are inefficient, then you may find it useful to write a constraint propagator.

When you write a new constraint propagator to link certain decision variables, you must implement the inference algorithm that eliminates from the domains of those variables the values that could not participate in a solution. These reductions are carried out by means of *elementary modifiers* on the domains of these decision variables.

This chapter explains what you must do if you decide to implement a new constraint using constraint propagators. and contains the following sections.

Simplest propagator: writing a checker

A checker checks that the values assigned to the decision variables values are consistent.

When a constraint is added to a model in CP Optimizer, its purpose is to limit the possible combinations of values that can be assigned to the decision variables. A constraint propagator is intended to have the same effect. The simplest form of a propagator is a checker. Once all of the decision variables have been assigned values, the checker checks that the assigned values are consistent.

For example, a checker written to enforce the relation $x <= y$ must ensure that the value assigned to the decision variable x is not greater than the value assigned to the decision variable y.

To define a new class of constraint propagator in CP Optimizer, you implement a subclass of a predefined class. In the C++ API, this is done by defining a subclass of the `IloPropagatorI` class and a virtual member function, `execute`. In the Java API, this is done by defining a subclass of the `IloCustomConstraint` class and a member function, `execute`. In both cases the function `execute` performs the checking, ensuring that the values assigned to the decision variables are consistent in regards to the constraint. In addition, you must explicitly indicate which decision variables are involved in the propagator. Whenever the domain of one of the indicated variables changes, the `execute` method of the propagator is called. The data members of the propagator subclass include the decision variables involved in the propagator.

Consequently, the definition of a new class of constraint propagator in the C++ API follows the form:

```
class MyConstraintI : public IloPropagatorI {
private:
  // data members for the constraint
public:
  MyConstraintI(... args ...);  // constuctor
  void execute();
  IloPropagatorI * makeClone(IloEnv env) const;
};
```

A new class of constraint propagator in the Java API follows the form:

```
class MyConstraint extends IloCustomConstraint {
   // data members for the constraint
  public MyConstraint(IloCP cp, ... args ... ) throws IloException { ... }
  public void execute() { ... }
}
```

To illustrate how to define a checker by writing a new class of constraint propagator, consider the relation $x <= y$. The arguments of the propagator are obviously the decision variables `x` and `y` themselves. In the C++ API, the class declaration is:

```
class LEConstraintI : public IloPropagatorI {
 private:
   IloIntVar _x;
   IloIntVar _y;
 public:
   LEConstraintI(IloIntVar x, IloIntVar y);
   void execute();
   IloPropagatorI * makeClone(IloEnv env) const;
};
```

In the Java API, the class declaration is:

```
class LEConstraintI extends IloCustomConstraint {
  private IloIntVar _x;
  private IloIntVar _y;
  // ...
```

The role of the constructor is to initialize the data members and inform the constraint propagation engine upon which variables this propagator must work. To inform the engine that a decision variable is incident on the constraint, you call the method `IloPropagatorI::addVar` in the C++ API and `IloCustomConstraint::addVar` in the Java API with the variable as the argument. The constructor for the `LEConstraintI` in the C++ API is:

```
LEConstraintI::LEConstraintI(IloIntVar x, IloIntVar y)
   : IloPropagatorI(x.getEnv()), _x(x), _y(y) {
   addVar(x);
   addVar(y);
}
```

and in the Java API is:

```
public LEConstraintI(IloCP cp, IloIntVar x, IloIntVar y) throws IloException {
  super(cp);
  _x = x;
  _y = y;
  addVar(x);
  addVar(y);
}
```

In the method `execute` for the checker, you specify to the inference engine the logic for determining whether or not the assigned values satisfy the constraint.

For the constraint $x <= y$, this logic is easy to determine:

- if x and y are fixed to values, the value assigned to x must not exceed the value assigned to y. If the values are inconsistent, then the propagation must be forced to fail.

In the case that the values are inconsistent, you can force the propagation to produce a failure by setting the value of a decision variable to a value other than that to which it is fixed.

The virtual member function `execute` implements the value checker; in the C++ API, the method is:

```
void LEConstraintI::execute () {
  if (isFixed(_x) && isFixed(_y))
    if (getValue(_x) > getValue(_y))
      setValue(_x, getValue(_x)+1);
}
```

and in the Java API, the method is

```
public void execute() {
  if (isFixed(_x) && isFixed(_y))
    if (getValue(_x) > getValue(_y))
      setValue(_x,getValue(_x)+1);
}
```

In the C++ API, when subclassing the `IloPropagatorI` class to create a propagator, you also need to define the method that will clone the propagator for use in the CP Optimizer engine during parallel search.

```
IloPropagatorI* LEConstraintI::makeClone(IloEnv env) const {
  return new (env) LEConstraintI(_x, _y);
}
```

To be able to use the propagator as a constraint in a C++ API model, you create a function using `IloCustomConstraint` that returns an `IloConstraint`.

```
IloConstraint LEConstraint(IloIntVar x, IloIntVar y){
  return IloCustomConstraint(x.getEnv(),
         new (x.getEnv()) LEConstraintI(x, y));
}
```

While writing a checker is an interesting introductory exercise, it is generally not an efficient manner in which to create a new constraint. When writing a constraint propagator, it is important to eliminate inconsistent values as soon as possible.

Invariants: what to propagate?

A propagator must eliminate inconsistent values from the domains of the decision variables.

When a constraint is added to a model in CP Optimizer, its purpose is to limit the possible combinations of values that can be assigned to the decision variables. A constraint propagator is intended to have the same effect; however, when you define a propagator, you must provide the logic to enable the inference algorithm to eliminate inconsistent values from the domains of the variables that are involved in that constraint propagator.

For example, a propagator written to enforce the constraint $x <= y$ must ensure that the values in the domains of x and y satisfy the constraint. From a working point of view, that propagator must eliminate values from the domains of the variables, values that are definitely inconsistent.

When you are writing a propagator, it is a good idea to start by defining the *invariant* of the constraint propagator before you implement it. (The term invariant is used here in the usual sense of software engineering.)

Consider the example of $x <= y$ in this context. Its invariant is easy to determine and can be expressed this way:

- the values in the domain of x must be less than or equal to the maximum of the values in the domain of y;
- the values in the domain of y must be greater than or equal to the minimum of the values in the domain of x.

More complicated relationships between decision variables require more complicated invariants. In general, it is best to eliminate as many values as possible from the domains of the decision variables.

Elementary modifiers for variables

Elementary modifiers for decision variables are used to remove inconsistent values from domains.

A propagator reduces the domains of the variables involved in the constraint propagator. In the C++ API, the modifiers for the domains of decision variables are available as member functions of `IloPropagatorI`. In the Java API, the modifiers for the domains of integer variables are available as member functions of `IloCustomConstraint`. In both APIs, these functions are `removeValue`, `setMax`, `setMin`, `setRange` and `setValue`. You can use these elementary modifiers to implement the inference algorithm for your constraint propagator.

For example, consider how `setValue`, one of these predefined modifiers, behaves when called on a decision variable x with current domain [0..1]. In the C++ API, the code fragment:

```
setValue(x,1);
```

reduces the domain of x to a singleton (a single element) with the value 1. This behavior literally involves a domain reduction, and this behavior is consistent throughout CP Optimizer.

For example, consider the following constraint propagator written using the C++ API:

```
void ModConstraintI::execute () {
  setValue(_x,1);
}
```

Adding this constraint propagator to the model and propagating:

```
    IloIntVar x(env, 1, 3, "x");
    model.add(ModConstraint(x));
    IloCP cp(model);
    if (cp.propagate())
        cp.out() << cp.domain(x) << std::endl;
    else
        cp.out() << "No solution." << std::endl;
    cp.end();    cp.end();
```

produces output that contains:

```
 x[1]
```

If you attempt to set the value of a constrained variable to a value that is not in the current domain, this action leads to a failure in the search. Assume that the decision variable *x* has the domain of [2..3] and you set the value to 1. This leads to a failure, and the optimizer will no longer investigate the subtree of the current node. The output produced contains

```
 No solution.
```

In the C++ API, the elementary modifiers used in a propagator are member functions of the class `IloPropagatorI`. In the Java API, the elementary modifiers are member functions of the class `IloCustomConstraint` which implements the interface `IloPropagator`.

In both APIs, the modifiers are:

- `setValue(IloIntVar x, IloInt value)` tries to make `x` equal to `value`;
- `setMin(IloIntVar x, IloInt min)` tries to make `x` greater than or equal to `min`;
- `setMax(IloIntVar x, IloInt max)` tries to make `x` less than or equal to `max`;
- `setRange(IloIntVar x, IloInt min, IloInt max)` tries to make `x` stay between `min` and `max`, inclusive;
- `removeValue(IloIntVar x, IlcInt value)` tries to make the current domain of `x` not contain `value`.

Elementary modifiers only reduce domains: they do not enlarge domains. For that reason, the following code does not modify the domain [0..1] of the decision variable and, in particular, it does not make the variable's upper bound equal to 2:

```
  setMax(x,2);
```

The next section shows how to take an implementation of an invariant and write a constraint propagator.

Defining a new class of constraint propagator

Defining a new class of constraint propagator involves writing methods to implement the logic regarding the reduction of the domains.

When defining a new class of constraint propagator, the function `execute` performs the required domain reductions, updating the domains to make the domains consistent in regards to the constraint. For the constraint $x <= y$, this logic is easy to determine:

- the values in the domain of x must be less than or equal to the maximum of the values in the domain of y;
- the values in the domain of y must be greater than or equal to the minimum of the values in the domain of x.

The invariant of the constraint $x <= y$ can be enforced using elementary modifiers. The virtual member function `execute` implements the logic regarding the reduction of the domains. For example, the function would look like this in the C++ API:

```
void LEConstraintI::execute () {
  setMax(_x, getMax(_y));
  setMin(_y, getMin(_x));
}
```

and like this in the Java API:

```
public void execute() {
  setMax(_x,(int)getMax(_y));
  setMin(_y,(int)getMin(_x));
}
```

Putting all the pieces together, the propagator in the C++ API looks like:

```
class LEConstraintI : public IloPropagatorI {
 private:
   IloIntVar _x;
   IloIntVar _y;
 public:
   LEConstraintI(IloIntVar x, IloIntVar y);
   void execute();
   IloPropagatorI * makeClone(IloEnv env) const;
};

LEConstraintI::LEConstraintI(IloIntVar x, IloIntVar y)
   : IloPropagatorI(x.getEnv()), _x(x), _y(y) {
   addVar(x);
   addVar(y);
}

void LEConstraintI::execute () {
  setMax(_x, getMax(_y));
  setMin(_y, getMin(_x));
}
IloPropagatorI* LEConstraintI::makeClone(IloEnv env) const {
  return new (env) LEConstraintI(_x, _y);
}
IloConstraint LEConstraint(IloIntVar x, IloIntVar y){
  return IloCustomConstraint(x.getEnv(),
         new (x.getEnv()) LEConstraintI(x, y));
}
```

In the Java API, the full propagator is:

```
class LEConstraintI extends IloCustomConstraint {
  private IloIntVar _x;
  private IloIntVar _y;
  // ...
public LEConstraintI(IloCP cp, IloIntVar x, IloIntVar y) throws IloException {
  super(cp);
  _x = x;
  _y = y;
```

```
  addVar(x);
  addVar(y);
}
public void execute() {
  setMax(_x,(int)getMax(_y));
  setMin(_y,(int)getMin(_x));
}
}
```

Example of a propagator

An example of a propagator is provided for illustrative purposes.

While the example in the previous section illustrates the mechanics of writing a propagator, the example is not realistic. As a more realistic example of a propagator, consider the case in which the model contains an array of decision variables and a decision variable that needs to be constrained to take the value of the index of any element that has been assigned a value that is the minimal value of the elements in the array. First, the propagator must be informed of the array of decision variables and the decision variable that is constrained to be an appropriate index of the array. Thus the constructor is:

```
ArgMinI::ArgMinI(IloIntVar y, IloIntVarArray x) :
  IloPropagatorI(y.getEnv()), _y(y), _x(x) {
  addVar(y);
  for (IloInt i=0; i< x.getSize(); i++)
    addVar(x[i]);
}
```

The `execute` method of the propagator first ensures that the domain of the decision variable representing the index is within the range of indices of the array. The method then examines the decision variables in the array and determines the minimal upper bound of the domains of all the elements of the array. It also determines, for those elements whose indices are in the domain of `_y`, the minimal value of the currently possible values, called the minimal active lower bound.

If there is any decision variable in the array whose domain is strictly greater than the calculated minimal upper bound, then it cannot be an element that takes the minimal value in the array. In this case, the value of the index is removed from the domain of the decision variable `_y`.

Then, if there is an index that is not in the domain of the decision variable `_y`, the decision variable at that index cannot take values smaller than the minimal active lower bound, and the minimum of the domain of that variable is set to the minimal active lower bound.

Putting together the pieces, the `execute` method is:

```
void ArgMinI::execute() {
  IloInt i;
  setRange(_y, 0, _x.getSize()-1);
  IloInt minUpperBound = IloIntMax;
  IloInt minActiveLowerBound = IloIntMax;
  for (i = 0; i < _x.getSize(); i++) {
    if (minUpperBound > getMax(_x[i]))
      minUpperBound = getMax(_x[i]);
    if (isInDomain(_y, i) && minActiveLowerBound > getMin(_x[i]))
      minActiveLowerBound = getMin(_x[i]);
  }
  for (i = 0; i < _x.getSize(); i++)
    if (minUpperBound < getMin(_x[i]))
      removeValue(_y,i);
```

```
  for (i=0; i < _x.getSize(); i++)
    if (!isInDomain(_y, i))
      setMin(_x[i],minActiveLowerBound);
}
```

Programming tips

Tips for writing efficient constraint propagators are provided.

All modifications made to decision variables inside a propagator are buffered. This means that the changes you make to variable domains are not visible inside the propagator, but are recorded and applied when the propagator exits. Your code needs be aware of this if it examines the domain of a variable after modifying it. For example, the following code will not cause the assertion to fail, but the lower bound of the decision variable will be set to one greater than its current value when the `execute` function of the propagator exits.

```
void BufferConstraintI::execute () {
    IloInt min = getMin(_x);
    setMin(_x, min + 1);
    assert(getMin(_x) == min);
}
```

Chapter 3. Using engine extensions

Before a model is solved, the optimizer engine extracts the modeling objects into engine objects. These engine objects contain the necessary elements for the optimization.

Overview

Engine extensions provide flexibility for writing custom constraints and custom constructive search.

The recommended development path for solving an optimization problem with CP Optimizer is to use IBM ILOG Concert Technology for modeling, together with the CP Optimizer search engine tuned with parameters and search phases. In this way, you benefit from all future improvements to the CP Optimizer propagation and search strategies.

While the modeling objects provided by IBM ILOG Concert Technology and the algorithms provided by the CP Optimizer engine will generally be sufficient to model and solve your problem, in some situations you may need the full flexibility of writing and maintaining the solution process in various ways, such as writing custom constraints and custom constructive search by means of goals.

Currently, the engine extensions API is available in C++ only.

When to use engine extensions

Engine extensions should be used only after determining that the predefined constraints and search strategies do not fit your needs.

When designing your application, you may discover that there is no predefined constraint that fits your needs and that writing a constraint propagator does not provide the flexibility that you need to write an inference algorithm which performs the most efficient domain reductions. Or, you may decide that a search technique more complex than search phases with custom evaluators and choosers would perform better for your particular problem. Or you may need to create portions of the model during the search process itself. To implement a custom search, you must also take full control of the search process.

The engine extensions API for CP Optimizer allows you much more flexibility in terms of defining new constraints and constructive search; however, the drawback is that these customizations are generally more difficult to maintain and update than the predefined constraints and search strategies provided in CP Optimizer.

Model extraction

A model is extracted to the to the optimizer before search begins.

Overview

The process of extracting a model to the optimizer is explained.

The first step in understanding how to use the engine extensions API is to understand more about what happens when a call to `IloCP::extract` is made.

Extraction process

A model is extracted to the engine recursively.

When an instance of `IloCP` extracts a model, it iterates over each extractable added to the model and works on extracting it. This mechanism is recursive in the sense that the extraction of an object may lead to the extraction of another object. For instance, the extraction of the model:

```
IloIntVar x(env, 0, 1);
IloIntVar y(env, 0, 10);
IloIntVar z(env, -1, 3);
IloModel model(env);
model.add(z);
model.add(x + 2 * y <= 3);
```

involves the extraction of the variable *z* and the constraint *x + 2 * y <= 3*. The extraction of the constraint in turn involves the extraction of variables *x* and *y*.

The purpose of extraction is to create engine objects from the extractables of the model. Theses objects can be constraints, variables or expressions.

In particular, it creates:

- one engine variable for each decision variable in the model and
- an engine constraint for a constraint or a group of constraints in the model.

These engine objects are usually prefixed with `Ilc`.

- The engine object of a decision variable is an instance of `IlcIntVar`.
- The engine object of a constraint is an instance of `IlcConstraint`.
- The engine object of an expression is an instance of `IlcIntExp` or `IlcFloatExp` depending on the type of the expression.
- Other types of extractables generally have counterparts in the engine. For instance, an instance of `IloIntTupleSet` is extracted to an instance of `IlcIntTupleSet`.

Engine representation of variables

Decision variables in the engine can be accessed via a method of the optimizer object.

You can access to the engine variables (`IlcIntVar`) for each `IloIntVar` extracted from a model by using the method `IloCP::getIntVar(IloIntVar)`. For instance, running the code:

```
IloIntVar x(env, 0, 10, "x");
IloIntVar y(env, 0, 50, "y");
IloModel model(env);
model.add(x*x >= y);
IloCP cp(env);
cp.extract(model);
cp.out() << "The engine variable of " << x << " is " << cp.getIntVar(x) << std::endl;
cp.out() << "The engine variable of " << y << " is " << cp.getIntVar(y) << std::endl;
```

produces the output:

```
The engine variable of x[0..10]  is x[0..10]
The engine variable of y[0..50]  is y[0..50]
```

The instances of IloIntVar contain just the domain information for that variable. The instances of IlcIntVar contains the same information plus some additional data structures to handle constraint propagation.

Engine representation of constraints

Constraints in the engine are built using the decision variables in the engine.

The engine object for a constraint is an instance of IlcConstraint. Basically, a model constraint or group of constraints is extracted to an instance of IlcConstraint. To illustrate constraint extraction, consider the following model:

```
IloIntVar x(env, 0, 3);
IloIntVar y(env, 0, 3);
IloModel model(env);
model.add(x < y);
```

The code:

```
IloCP cp(env);
cp.extract(model);
```

extracts the model to the IloCP instance. First, it takes the constraint $x < y$ and starts extracting it. In order to extract this constraint, it extracts the two variables x and y. The extraction of these variables creates two IlcIntVar objects: ix and iy. Then, the extraction of the constraint continues. The instance of IloCP takes the two IlcIntVar objects and creates the equivalent IlcConstraint: $ix < iy$. This inequality constraint is then added to the instance of IloCP.

The extraction creates an equivalent of the following code:

```
IlcIntVar ix(cp, 0, 3);
IlcIntVar iy(cp, 0, 3);
cp.add(ix < iy);
```

Preprocessing

CP may preprocess a model during extraction.

CP Optimizer preprocessing can aggregate expressions, constraints or decision variables in order to provide faster constraint propagation, more domain reduction or both.

As an example, a set of IloCount expressions operating on the same array of decision variables are extracted to a single instance of IlcDistribute constraint.

Variables can also be aggregated. For instance, in the following model:

```
IloIntVar x(env, 0, 10,"x");
IloIntVar y(env, 0, 50,"y");
IloModel model(env);
model.add(x == y + 2);
```

the variables x and y are aggregated. Running the code:

```
IloCP cp(env);
cp.extract(model);
cp.out() << "The engine variable of " << x << " is " << cp.getIntVar(x) << std::endl;
cp.out() << "The engine variable of " << y << " is " << cp.getIntVar(y) << std::endl;
```

produces the output:

```
The engine variable of x[0..10]  is (y[0..50] + 2)
The engine variable of y[0..50]  is y[0..50]
```

Engine extension API classes

When using the engine extensions API of CP Optimizer, you will need to use the internal engine object classes instead of only the modeling classes. These internal classes are prefixed with `Ilc`.

Overview

The internal engine object classes are needed to use the engine extensions API.

When using the engine extensions API of CP Optimizer, you will need to use the internal engine object classes instead of only the modeling classes. These internal classes are prefixed with `Ilc`.

Implementation classes and handles

An object of the handle class points to an object of the corresponding implementation class.

Throughout this chapter, it has been taken for granted that the major entities of CP Optimizer are simply objects in the full sense of C++. In fact, CP Optimizer entities depend on two classes: a handle class and an implementation class, where an object of the handle class points to an object of the corresponding implementation class. As documented in the *CP Optimizer C++ API Reference Manual*, handles are passed by value, and they are created as automatic objects, where "automatic" has the usual C++ meaning.

When using CP Optimizer, you will mostly work with handles, that is, the objects that are really pointers to corresponding implementation objects on the CP Optimizer heap. CP Optimizer takes care of all the allocation, memory-management and de-allocation of these internal implementation objects, keeping the details of these implementation classes "out of sight." However, if for some reason, you need to define new subclasses of CP Optimizer objects, you need to do so in a two-step process: you'll have to define the underlying implementation class plus the corresponding handle class.

For example, consider the predefined CP Optimizer extensions class, `IlcConstraintI`. For this class implementing an object with its own data members and virtual functions, CP Optimizer defines the handle class `IlcConstraint`. Essentially, such a handle class contains objects that are really only pointers to instances of the corresponding implementation class `IlcConstraintI`.

This has the practical effect of elevating the idea of a pointer to the level of an object itself. A number of advantages derive from this; for one thing, it provides greater certainty about how pointers are used. In particular, there is no longer any risk that the compiler will mistakenly interpret the expression `x+1` as a pointer and apply pointer arithmetic to it. Instead, pointers are consistently and systematically treated as objects.

However, the fundamental precepts of CP Optimizer that there are "No more pointers" and that "Everything is an object" mean that when you extend CP Optimizer by defining a new implementation class, you must also define a function that returns a handle.

Variables and expressions

Engine decision variables and expressions are used in the search.

When the model is extracted to an instance of IloCP, generally an instance of IlcIntVar is automatically created for each IloIntVar in the model. If no IloExtractable uses the decision variable, it will not be extracted. In this case, you can force a model to use the variable by directly adding it to the model.

Once an IloIntVar is extracted in an instance of IloCP, you can access the IlcIntVar from the modeling object IloIntVar using the accessors of the optimizer. The following code returns the IlcIntVar from an IloIntVar x:

```
IlcIntVar cpx = cp.getIntVar(x);
```

The second way of introducing search decision variables in CP Optimizer is to create them directly. Here is one constructor to do so:

```
IlcIntVar(IloCP cp, IlcInt min, IlcInt max, char* name=0);
```

The constructor declares a search decision variable that can take a value between min and max (its lower and upper boundaries). In other words, the lower and upper boundaries define the domain of the search decision variable that is being constructed. The character string name is displayed whenever the variable is printed. It is an optional argument.

This constructor is intended to be used for creation of search decision variables inside instances of other computation objects. It should not be used to declare variables that are part of the problem statement in the modeling layer. That should be done with the constructor for IloIntVar.

Domains of decision variables

In the *CP Optimizer User's Manual*, domains of decision variables are introduced as well as the manner in which domains are reduced in the inference algorithms called by the constraint propagation engine. It is important to note that the domain reductions are in fact, not performed on the modeling objects themselves, but on the corresponding internal engine objects, instances of IlcIntVar.

Constraints and constraint propagation

Engine constraints are used in the search.

When the model is extracted to an instance of IloCP, generally an instance of IlcConstraint is automatically created for each IloConstraint in the model.

When a constraint is added to the CP Optimizer engine and the search is active, the constraint is used immediately to reduce the domains of the decision variables that it involves. CP Optimizer reduces a domain by removing those values that cannot satisfy the constraint and thus cannot participate in a solution.

Posting a constraint is reversible: the constraint is removed when CP Optimizer backtracks to choice points set before that constraint was posted. If constraint propagation causes a domain to be reduced to a single value, then the decision variable will be bound to that remaining value.

In addition, when you post a constraint, the constraint is saved so that whenever any of the variables to which it applies is modified, the constraint will be activated, and the modification will be transmitted to the other variables that the constraint involves. This activity is called constraint propagation.

The algorithm used for constraint propagation in CP Optimizer is straightforward in principle. CP Optimizer maintains a queue of variables, called the constraint propagation queue. When a constrained variable is modified, that variable is put at the end of that queue if it is not already in the queue. As long as there are variables in that queue, the algorithm takes the first variable from the queue. This particular variable is said to be in process.

When a variable is processed, it is first removed from the propagation queue. Then each constraint posted on that variable is examined. For one such constraint, all the variables on which it is posted are in turn examined: their domains are reduced by removing those values that are inconsistent with it. If a variable is already in process, then this domain reduction will be deferred until it is no longer in process. If some of these variables are modified during this activity, they, too, are put into the queue if they are not yet in the queue. The algorithm continues as long as there is a variable in the queue to process. The algorithm automatically reduces domains as necessary and halts in either of two situations: when all domains contain only values consistent with the constraints, or when a domain becomes empty. For performance considerations, it does not carry out all the reductions theoretically possible.

This algorithm has several important properties:

- This algorithm always halts.
- It lets you use constraints (such as arithmetic constraints, for example) on more than two variables at a time.
- It lets you handle problems dynamically; that is, you can solve problems where new constraints can be added during the search for a solution.
- Regardless of the order in which the constraints are considered, the domains will always be the same at the end of the execution of the propagation.

Goals

Engine goals are used in the search.

Goals are the building blocks used to implement search algorithms in CP Optimizer. Both predefined search algorithms and user-defined search algorithms can be expressed in CP Optimizer through goals.

Like other CP Optimizer entities, a goal is implemented by two objects: a handle (an instance of the class `IlcGoal`) that contains a data member (the handle pointer) that points to an implementation object (an instance of the class `IlcGoalI` allocated on the CP Optimizer heap).

Among other member functions, the class `IlcGoalI` has a virtual member function, `execute`, which implements the execution of the goal. The `execute` member function must return another goal: the subgoal of the goal under execution. If the execute member function returns 0 (zero), then no subgoal has to be executed.

A goal can either succeed or fail. A goal fails if a `fail` member function (such as `IlcGoalI::fail`, for example) is called during its execution. A goal succeeds if it does not fail.

Goal execution is controlled by the member functions `IloCP::next` or `IloCP::solve` and implemented by a goal stack. The first time this member function is called, it pushes all the goals that have been added to the invoking optimizer onto the goal stack. Then it pops the top of the stack, and if there is a goal there, it executes that

goal. When the execution of the current goal is complete, its subgoal is executed (if the current goal has any subgoals). If there are no remaining subgoals, then the next goal on top of the stack is popped. The member function next stops when the goal stack is empty.

Chapter 4. Writing complex custom constraints

New, custom constraints can be written using engine extensions.

Overview

Before writing a custom constraint, predefined constraints and constraint propagators should be investigated. If these do not provide the necessary flexibility, then custom constraints can be used to link specific decision variables.

CP Optimizer offers a wide range of predefined constraints, together with powerful logical operators for combining them. These facilities usually suffice for expressing even the most specific constraints. However, when you are modeling your problem, a predefined constraint which implements your knowledge of the problem may not exist. In this case, you may find it useful to write a constraint propagator. A constraint propagator is a simple way to define a custom constraint, that is, one where you define the domain reduction rules via some C++ or Java code. Constraint propagators are discussed in detail in Chapter 2, "Using constraint propagators," on page 5.

However, there may be cases in which a constraint propagator does not provide enough flexibility. For instance, it may be that you need to solve a subproblem within the execution of the inference algorithm of your constraint.

Or it may be that be that you want to have different operations which depend on how the domain of the triggering variable has been modified; you may not want to call the propagator for each modification but only when the decision variable is fixed.

Or it may be that you may want to be able to implement an efficient algorithm in a incremental way, so you need to know which are the values that have been currently modified; these values are known as the delta domains).

When you write a new constraint class to link certain decision variables, you must implement the inference algorithm that eliminates from the domains of the variables the values that could not participate in a solution. These reductions are carried out by means of *elementary modifiers* on the domains of these variables.

Understanding constraints

Constraints link specified decision variables.

As indicated in the *CP Optimizer User's Manual*, CP Optimizer offers a wide range of predefined constraints, together with powerful logical operators for combining them. These facilities usually suffice for expressing even the most specific constraints.

However, when CP Optimizer treats a group of heterogeneous constraints together as a set, it generally still must deal with each of them locally. One way to offset this "localness" about the way constraints are treated is to use catalyzing constraints. Another--more radical approach--is to write a new constraint. This alternative may at times be somewhat more difficult.

When you write a new constraint to link certain decision variables, you must implement reductions of the domains of these variables by eliminating any values that could not participate in a solution. These reductions are carried out by means of elementary modifiers on the domains of these variables.

Elementary modifiers for decision variables

Elementary modifiers for engine decision variables are used to reduce the domain of a decision variable within the domain reduction algorithm.

In a custom constraint, the domain reduction algorithm is implemented on the internal computation (search) decision variables, the `IlcIntVar` equivalents of the `IloIntVar` model decision variables. Besides the accessors such as `getMin`, `getMax` or `getValue`, defined for integer expressions, there are also predefined elementary modifiers. You use these modifiers to implement custom constraints.

Consider how `setValue(IlcInt c)`, one of these predefined modifiers of `IlcIntExp`, behaves when called on a search decision variable *x* with a current domain of [0..1]:

```
x.setValue(1);
```

This fragment of code effectively reduces the domain of the decision variable to a singleton (a single element) with the value 1. This behavior literally involves a domain reduction, and this behavior is consistent throughout CP Optimizer: if the value 1 were not in the initial domain, CP Optimizer would have raised an error.

For example, consider the following contrasting piece of code called on a decision variable x with a current domain of [0..1]. It raises an error, a fail.

```
x.setValue(2);       // LEADS TO FAILURE
```

It is also important to note that modifiers are volatile; if a modifier does not have an immediate effect, it will have no later effect. This volatility is a way in which elementary modifiers differ from constraints. Consider the following code applied to the search decision variables *x* and *y* with current domains of [0..1] :

```
IlcIntVar x(cp,0,1), y(cp,0,1);
(x + y).setValue(1);        // NO EFFECT
```

That code has no effect because all the values of the domains can participate in a solution. If later the domain of *x* is reduced to [0], this previously applied modifier will have no effect. If the modifier is reapplied, then the domain of *y* will be reduced to the value 1.

Of course, if you instead use constraints and add them to the model, those hazards can be avoided.

```
IloIntVar x(env,0,1), y(env,0,1);
model.add(x + y == 1);
```

This code will not directly modify any domains, but during the search for solutions, only the two solutions where the sum is equal to 1 appear. CP Optimizer stores the extracted constraint so that at each modification of the domain of either of the decision variables, the optimizer engine executes the appropriate domain modifier.

For the class `IlcIntExp`, the modifiers are:

- `setValue(IlcInt value)` which tries to make the expression equal to `value`;

- `setMin(IlcInt min)` which tries to make the expression greater than or equal to `min`;
- `setMax(IlcInt max)` which tries to make the expression less than or equal to `max`;
- `setRange(IlcInt min, IlcInt max)` which tries to make the expression stay between `min` and `max`, inclusive;

For the class `IlcIntVar`, which inherits from `IlcIntExp` the modifiers are:

- `setRange(IlcIntArray array)` which tries to make the expression include only those values in the array;
- `removeValue(IlcInt value)` which tries to make the variable different from `value`;
- `removeInterval(IlcInt min, IlcInt max)` which tries to make the variable strictly less than `min` or strictly greater than `max`. In other words, it tries to make the variable stay outside the interval defined by `min` and `max`, inclusive.

Before leaving the topic of modifiers, it must be emphasized again that they only reduce domains: they do not enlarge domains. For that reason, the following code, when applied to a search decision variable x with a current domain of [0..1] modifies nothing in the domain of *x* and, in particular, it does not make the upper boundary of *x* equal to 2.

```
x.setMax(2);          // DOES NOTHING!
```

Invariants: what to propagate?

Understanding the invariants is necessary in irder to understand what to propagate.

A constraint that links decision variables is an object with a task: that of reducing the domains of those variables in terms of the semantics of those variables. For example, once you have added the constraint *x* <= *y* on the search decision variables *x* and *y*, that constraint must ensure that the values in the domains of *x* and *y* satisfy the constraint. From a working point of view, that constraint must eliminate values from the domains of the variables, values that are definitely inconsistent.

When you are writing a constraint, it is a good idea to start by defining the invariant of that constraint before you implement it by means of elementary modifiers. (The term invariant is used here in the usual sense of software engineering.)

Consider the example of *x* <= *y* in this context. Its invariant is easy to determine, and can be expressed this way:

- the values in the domain of *x* must be less than or equal to the maximum of the values in the domain of *y*;
- the values in the domain of *y* must be greater than or equal to the minimum of the values in the domain of *x*.

With elementary modifiers, this invariant can be translated this way:

```
x.setMax(y.getMax());
y.setMin(x.getMin());
```

The constraint propagation algorithm

The constraint propagation algorithm is straightforward in principle. When the domain of a decision variable is modified, the constraints containing that variable are examined to determine whether any values in the domains of other decision variables are now inconsistent.

To see how the propagation algorithm works with constraints, consider the constraint x <= y, written with the following fragment of code:

```
IlcIntVar x(cp,0,3), y(cp,0,2);
cp.add( x <= y );
```

When this constraint is added to the optimizer, the invariant expressed in the previous section becomes active and reduces the domain of x by removing the value 3. For the moment, that is all that can be deduced from the constraint. Since the constraint has to be taken into account by CP Optimizer every time one of the variables in it is modified, the constraint itself is physically attached to these two variables.

CP Optimizer has been designed to automate and to optimize the reduction of the domains of decision variables. The CP Optimizer propagation algorithm for that purpose is straightforward in principle. When the domain of a decision variable is modified, the constraints containing that variable are examined to determine whether any values in the domains of other decision variables are now inconsistent. If this is the case, necessary domain reductions are carried out in turn.

The examination of the constraints incident on a variable is triggered by any modification of that variable. There are several different kinds of modifications, depending on the class of variable under consideration. Those modifications are referred to as propagation events.

For the class of decision variables, there are, in fact, these propagation events:

- **value** means that a value has been assigned to the decision variable, that is, the variable has been fixed;
- **range** indicates that the minimum of the domain has increased or the maximum of the domain has decreased;
- **domain** indicates that the domain of the decision variable has been modified.

When you define a new class of constraint, you must also define the propagation events for that class of constraint. You do so by means of the pure virtual member function, `post`.

Sometimes more than one event can be triggered after a variable modification. Specifically, the value event is always accompanied by the range and domain events. Likewise, a range event is always accompanied by a domain event.

Consider a search decision variable, *var*, with a domain containing only two values, *value1* and *value2*, where *value1* < *value2*. If you add a constraint that *var != value1*, three events are triggered:

- the domain event is triggered since *value1* is actually removed from the domain of *var*;
- the range event is triggered since the minimal boundary of the domain of *var* has been increased;

- the value event is triggered since the variable is fixed by the reduction of its domain to a single value.

Writing your own constraint

A custom constraint can be written using engine extensions.

Overview

A custom constraint can be written using engine extensions.

Obviously, a constraint is an object in CP Optimizer. More precisely, a constraint in CP Optimizer is an instance of a class with two pure virtual functions, `propagate` and `post`. The virtual function `propagate` implements the invariant of the constraint; the function `post` defines on which events `propagate` executes.

To define a new class of constraint, you define a subclass of `IlcConstraintI`. The data members of this subclass include, among others, the search decision variables on which the constraint is posted.

Consequently, the definition of a new class of constraint looks something like this:

```
class MyConstraintI : public IlcConstraintI {
     ... // data members of the constraint
public :
    MyConstraintI(IloCP cp, ... args ...); // constructor
    ~MyConstraintI(){} // destructor; usually empty
    void post();
    void propagate();
}
```

To clarify how to define a new class of constraint, consider the constraint *x != y*. The arguments of this constraint are the search decision variables *x* and *y*.

```
class DiffConstraintI : public IlcConstraintI {
    IlcIntVar _x, _y;
  public:
    DiffConstraintI(IloCP cp, IlcIntVar x, IlcIntVar y);
    ~DiffConstraintI(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
};
```

The role of the constructor is to initialize the data members, like this:

```
DiffConstraintI::DiffConstraintI(IloCP cp,
                                 IlcIntVar x,
                                 IlcIntVar y)
    : IlcConstraintI(cp), _x(x), _y(y) {}
```

The invariant of *x != y* is easy to determine:

- if *x* is bound to a value, the domain of *y* does not contain this value;
- likewise, if *y* is bound to a value, the domain of *x* does not contain this value.

The virtual member function `propagate` implements the reduction of the domains. Using what you've already written, you would get this:

```
void DiffConstraintI::propagate () {
    if (_x.isFixed()) _y.removeValue(_x.getValue());
    if (_y.isFixed()) _x.removeValue(_y.getValue());
}
```

The second virtual member function to write is `post`. It connects the constraint to its arguments by specifying the events that have triggered the constraint. For search decision variables, there are three kinds of attachments, one kind for each type of event.

For the class `IlcIntVar`, these member functions are as follows: (The class `IlcConstraintI` derives from the class `IlcDemonI`.)

- `whenValue(IlcDemon ct)` attaches the demon `ct` to the value event of the variable under consideration. Every time the variable takes a unique value, the `execute` function for `ct` is called.
- `whenRange(IlcDemon ct)` attaches the demon `ct` to the range event of the variable under consideration. Every time the domain of the expression gets a new boundary, the `execute` function for `ct` is called.
- `whenDomain(IlcDemon ct)` attaches the demon `ct` to the domain event. Every time the domain of the variable is modified, the `execute` function for `ct` is called.

For a constraint, the `execute` member function calls `propagate`.

In the example of the constraint $x != y$, you want to propagate the constraint every time x or y is fixed. Here is the code for doing that:

```
void DiffConstraintI::post(){
    _x.whenValue(this);
    _y.whenValue(this);
}
```

Implementation classes and handles

A handle class contains an object that is a pointer to an instance of the corresponding implementation class.

For a predefined CP Optimizer class, `IlcConstraintI`, implementing an object with its own data members and virtual functions, CP Optimizer defines the handle class `IlcConstraint`. Essentially, such a handle class contains objects that are simply pointers to instances of the corresponding implementation class `IlcConstraintI`. When you extend CP Optimizer by defining a new implementation class, you must also define a function that returns a handle.

This rule applies particularly to the definition of new classes of constraints. Handles for constraints are instances of the class `IlcConstraint`. This class provides an instance of an implementation class, or more precisely, `IlcConstraintI*`. You must define a function that returns a handle, and the minimal service that the handle provides is to manage the memory allocation of instances in your implementation class. The form that it generally takes looks like this:

```
IlcConstraint MyConstraint( /* arguments for the constraint */){
        // get the optimizer from a variable
        return new (cp.getHeap()) MyConstraintI(cp, /* ...args... */);
}
```

The body of that function builds an instance of `MyConstraintI` by calling the constructor and allocates a place on the CP Optimizer heap by calling the overloaded operator, `new (cp.getHeap() ... )`. When you use this CP Optimizer `new`, you get an efficient allocator that automatically manages how memory is recovered in case of backtracking.

In the example, *x != y*, you write a function `DiffConstraint` which creates the constraint. Here's the code for it:

```
IlcConstraint DiffConstraint(IlcIntExp x, IlcIntExp y){
  IloCP cp = x.getCP();
    return new (cp.getHeap()) DiffConstraintI(cp, x, y);
}
```

That definition, of course, has two parts: the implementation and the handle. To use this new constraint within the search, you simply write:

```
cp.add(DiffConstraint(x, y));
```

This statement informs CP Optimizer to take the constraint into account within such mechanisms as propagation, backtracking and so forth. In general, it can be said that a call to `DiffConstraint` as written builds an instance of `DiffConstraintI` and returns a handle for that object (in other words, an encapsulation of the object) to post by means of the `post` member function. The instance of the implementation class is connected to the value events of the two decision variables. Every time these variables are fixed, the `propagate` member function is executed.

Using ILOCPCONSTRAINTWRAPPERn to wrap constraints

A macro is used to wrap a constraint for use in a model.

You can use the macro `ILOCPCONSTRAINTWRAPPERn` to wrap an existing instance of `IlcConstraint` when you want to use it within IBM ILOG Concert Technology modeling objects.

This macro defines a constraint class named `_thisI` with `n` data members. When `n` is greater than zero, the types and names of the data members must be supplied as arguments to the macro. Each data member is defined by its type `ti` and a name `ai`.

```
ILOCPCONSTRAINTWRAPPER0(_this, cp)

ILOCPCONSTRAINTWRAPPER1(_this, cp, t1, a1)

ILOCPCONSTRAINTWRAPPER2(_this, cp, t1, a1, t2, a2)

ILOCPCONSTRAINTWRAPPER3(_this, cp, t1, a1, t2, a2, t3, a3)

ILOCPCONSTRAINTWRAPPER4(_this, cp, t1, a1, t2, a2, t3, a3, t4, a4)
```

In order to use an instance of `IlcConstraint` in the modeling layer, you need to follow these steps:

- Use the macro to wrap the instance of `IlcConstraint` in an instance of `IloConstraint`.
- You must use the following `IloCPConstraintI` member functions to force extraction of an extractable or an array of extractables that are used in the constraint:

  ```
  void use(const IloSolver cp, const IloExtractable ext)const;
  void use(const IloSolver cp, const IloExtractableArray extArray)const;
  ```

For more information on wrapping constraints for use with IBM ILOG Concert Technology, see the documentation for the macro `ILOCPCONSTRAINTWRAPPER` and the class `IloCPConstraintI` in the *CP Optimizer C++ API Reference Manual*.

Writing your own logical constraint

Additional virtual functions must be defined to use a custom constraint in a logical constraint.

If you want to be able to use a custom constraint in a logical constraint, you will need to define additional virtual functions (besides those you just defined for a new constraint: `propagate` and `post`). These additional virtual functions are `isViolated`, `makeOpposite` and `metaPostDemon`. This section shows you how to define these functions.

In order to express the idea that the variable *x* is different from some other variable, *y*, or that *y* is different from *z*, you can write that very simply by using the logical-OR operator provided by CP Optimizer.

```
cp.add(DiffConstraint(x, y) || DiffConstraint(y, z));
```

Once one of those two constraints proves false, CP Optimizer makes the other one true. To do so, CP Optimizer needs to know the truth value of these constraints. In fact, CP Optimizer needs only to know when one of these constraints is violated. You need to define under what conditions `DiffConstraint` is definitely not satisfied. In our example, *x != y* is violated if, and only if, variables *x* and *y* are fixed to the same value. Hence, the definition of `isViolated`:

```
IlcBool DiffConstraintI::isViolated() const {
    return (_x.isFixed() && _y.isFixed() &&
            _x.getValue()==_y.getValue());
}
```

To detect a violation, the disjunctive constraint has to be posted on the variables that are involved. This is the task of the `metaPostDemon` member function.

```
void DiffConstraintI::metaPostDemon(IlcDemonI * ct) {
    _x.whenValue(ct);
    _y.whenValue(ct);
}
```

Now, if you write:

```
cp.add(DiffConstraint(x, y) || DiffConstraint(y, z));
```

the `metaPostDemon` is called with the disjunctive constraint as its argument. Thus, you get the effect that once one of the two constraints proves false (that is, `isViolated` returns `IlcTrue`), CP Optimizer invokes the `propagate` function of the other one, insuring that the disjunctive constraint is taken into account.

There's still one more point to consider. Assume that you want to affirm that the constraint `DiffConstraint(x, y)` is false; in other words, that *x == y* is true. You can use the negation operator to write this:

```
cp.add(!DiffConstraint(x, y));
```

But if you do so, how will CP Optimizer know that you want to impose the negation of a specific constraint? You have to tell CP Optimizer explicitly what the opposite constraint is, and that is the work of the member function `makeOpposite`.

In our example, the opposite constraint of *x != y* is obviously *x == y*. Remember that the constraint `x == y` is a handle. Indeed, it is an instance of the class `IlcConstraint`. With CP Optimizer, you get the object implementation from the `getImpl` member function of the handle class. For example, `(x == y).getImpl()` returns the pointer to the implementation of `x == y`.

Thus, the code is:

```
IlcConstraintI* DiffConstraintI::makeOpposite() const {
    return (_x == _y).getImpl();
 }
```

Finally, here is the complete code of the custom constraint `DiffConstraint`. (First, you'll see the definitions of the appropriate class and its member functions; then `DiffConstraint` itself near the end.)

```
class DiffConstraintI : public IlcConstraintI {
    IlcIntVar _x, _y;
  public:
    DiffConstraintI(IloCP cp, IlcIntVar x, IlcIntVar y);
    ~DiffConstraintI(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
    virtual IlcConstraintI * makeOpposite() const;
    virtual void metaPostDemon(IlcDemonI* ct);


};

DiffConstraintI::DiffConstraintI(IloCP cp,
                                 IlcIntVar x,
                                 IlcIntVar y)
    : IlcConstraintI(cp), _x(x), _y(y) {}
void DiffConstraintI::propagate () {
    if (_x.isFixed()) _y.removeValue(_x.getValue());
    if (_y.isFixed()) _x.removeValue(_y.getValue());
}
void DiffConstraintI::post(){
    _x.whenValue(this);
    _y.whenValue(this);
}
IlcBool DiffConstraintI::isViolated() const {
    return (_x.isFixed() && _y.isFixed() &&
            _x.getValue()==_y.getValue());
}

IlcConstraintI* DiffConstraintI::makeOpposite() const {
    return (_x == _y).getImpl();
 }
void DiffConstraintI::metaPostDemon(IlcDemonI * ct) {
    _x.whenValue(ct);
    _y.whenValue(ct);
}
```

All the predefined constraints of CP Optimizer are defined like this, as constraints that can be used in logical constraints.

Using demons to propagate selectively

Using demons to propagate selectively can improve performance.

Consider the `DiffConstrantI` example; each time one of its variables is fixed, `isFixed` is checked against the two variables. As another example, consider an `IloAllDifferent` constraint which strives to make every decision variable in an array be assigned a different value. If the propagate function was written similarly to the one above, it would have to check to see if each variable in the array has been fixed, which would not be efficient. You can avoid these redundant tests by using intermediate demons, that is, by posting different demons on the different variables.

You add two member functions to the class DiffConstraintI, each of them responsible for the propagation of the constraint on one variable when the other is fixed. Here is the modification.

```
class DiffConstraintI : public IlcConstraintI {
    IlcIntVar _x, _y;
  public:
    DiffConstraintI(IloCP cp, IlcIntVar x, IlcIntVar y);
    ~DiffConstraintI(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    virtual IlcBool isViolated() const;
    virtual IlcConstraintI * makeOpposite() const;
    virtual void metaPostDemon(IlcDemonI* ct);
    void xDemon(){
        _y.removeValue(_x.getValue());
    }
    void yDemon(){
        _x.removeValue(_y.getValue());
    }
};
```

Now you need to define intermediate demons on variables that call the new member functions. You do that in this way.

```
ILCCTDEMON0(DiffConstraintI_xDemon,DiffConstraintI,xDemon);
ILCCTDEMON0(DiffConstraintI_yDemon,DiffConstraintI,yDemon);
```

Finally, you change the post member function to take this modification into account, like this:

```
void DiffConstraintI::post(){
    _x.whenValue(DiffConstraintI_xDemon(getCP(),this));
    _y.whenValue(DiffConstraintI_yDemon(getCP(),this));
}
```

This new version of the DiffConstraintI is faster that the previous one since the isFixed tests in propagate are avoided. The price to pay for this efficiency is the memory used by the two intermediate demons.

Using delta-domains to propagate efficiently

Using delta-domains in propagation can improve performance.

When a propagation event is triggered for a decision variable, the variable is pushed into the propagation queue if it was not already in the queue. Moreover, the modifications of the domain of the decision variable are stored in a special set called the domain-delta. This domain-delta can be accessed during the propagation of the constraints posted on that variable. When all the constraints posted on that variable have been processed, then the domain-delta is cleared. If the decision variable is modified again, then the whole process begins again. The state of the domain-delta is reversible.

The domain-delta is a special set where the modifications of the domain of a constrained variable are stored. This domain-delta can be accessed (by means of member functions of the class of the decision variable) during the propagation of the constraints posted on that variable. When all the constraints posted on that decision variable have been processed, then the domain-delta is cleared. If the variable is modified again, then the whole process starts over again. The state of the domain-delta is reversible.

The methods of IlcIntVar that you might use when examining the domain-delta are listed in the *CP Optimizer C++ API Reference Manual* and include isInProcess, getMaxDelta, getMinDelta, getOldMax, getOldMin and isInDelta. The domain-delta can be traversed by an iterator, an instance of IlcIntVarDeltaIterator.

To illustrate the use of the domain-delta and these functions, consider a custom constrain to implement the constructive disjunction *(x==y || x==z)* where *x*, *y* and *z* are decision variables. A declaration for this class using demons is:

```
class DeltaConstraintI : public IlcConstraintI {
    IlcIntVar _x, _y, _z;
  public:
    DeltaConstraintI(IloCP cp, IlcIntVar x, IlcIntVar y, IlcIntVar z);
    ~DeltaConstraintI(){}; // empty destructor

    virtual void propagate ();
    virtual void post();
    void xDemon();
    void yDemon(IlcIntVar y, IlcIntVar z);
};
```

with the demon declarations:

```
ILCCTDEMON0(DeltaConstraintI_xDemon,DeltaConstraintI,xDemon);
ILCCTDEMON2(DeltaConstraintI_yDemon,DeltaConstraintI,yDemon,
            IlcIntVar,y,IlcIntVar,z);
```

Considering the constructive disjunction, you can see that until the decision variable *x* is fixed, no values can be removed for the domains of *y* and *z*. [Note that more complete domain reduction would include a test for a null intersection of the domains of *x* and *y* (or *x* and *z*).]

In this custom constraint, the method xDemon is called only when the value of x is fixed. In this demon, you check if the value of x is in the domain of both y and z. If it is, then nothing can be deduced. If the value of x is in the domain of only one, then you set the value of that variable to the value of x. If the value of x is in the domain of neither, then the constraint is violated.

```
void DeltaConstraintI::xDemon(){
  assert(_x.isFixed());
  IlcInt xval = _x.getValue();
  IlcInt iny = _y.isInDomain(xval);
  IlcInt inz = _z.isInDomain(xval);
  if (iny && inz)
    return;
  if (iny)
    _y.setValue(xval);
  if (inz)
    _z.setValue(xval);
  if (!iny && !inz)
    fail();
}
```

When the domain of y changes, then for each of the values that have been removed from the domain of y, you check that the removed value is in z. If it is not, then that value should be removed from x. To determine what values have been removed from the domain of y, you can use the IlcIntVarDeltaIterator.

```
void DeltaConstraintI::yDemon(IlcIntVar y, IlcIntVar z){
  assert(y.isInProcess());
  for (IlcIntVarDeltaIterator iter(y); iter.ok(); ++iter) {
     IlcInt val = *iter;
     IlcInt inz = z.isInDomain(val);
```

```
      if (!inz)
        _x.removeValue(val);
  }
}
```

Since this relationship is similar for x and z, the same demon gets called when the domain of z changes, but the arguments are reversed. Thus the post function for the custom constraint is:

```
void DeltaConstraintI::post(){
    _x.whenValue(DeltaConstraintI_xDemon(getCP(),this));
    _y.whenDomain(DeltaConstraintI_yDemon(getCP(),this,_y,_z));
    _z.whenDomain(DeltaConstraintI_yDemon(getCP(),this,_z,_y));
}
```

The propagate function, which will be executed when the constraint is posted could be:

```
void DeltaConstraintI::propagate () {
  if (_x.isFixed()) {
    xDemon();
    return;
  }
  for (IloInt i = _x.getMin(); i == _x.getNextHigher(i); _x.getNextHigher(i)) {
    IlcInt iny = _y.isInDomain(i);
    IlcInt inz = _z.isInDomain(i);
    if (iny + inz == 0)
      _x.removeValue(i);
  }
}
```

Programming tips

Programming tips for writing constraints are provided.

Exhaustive propagation

Exhaustive propagation should be used in order to have an efficient custom constraint.

In order to write an efficient custom constraint, it is important to follow the following basic steps:

- The constraint must fail when there is no solution.
- The constraint must not remove values that are consistent with the definition.
- The constraint should be propagated in all directions.
- It is usually better to reduce the domain of variables as early as possible.
- A direct consequence of the previous rule is: it is usually better to reduce the domains as much as possible.

In order to reduce the domains of constrained variables as quickly as possible, you have to carry out all domain reductions that are possible. You do so by propagating constraints in every direction. Another way of looking at this issue is to see that constraints are not directed, but must be seen as equations in the widest sense of the term. The final results of the search for a solution must be the same, regardless of the order in which constraints are posted, even user-defined constraints.

Consider the constraint $x + y == z$. The domain of z must be modified when the domain of x or of y is modified. At the same time, you must not forget that the

domain of x must be modified when the domain of y or of z is modified. A similar rule applies for y as well. This is what is meant by propagating in all directions.

More generally, when designing a new constraint class, you must not overlook any direction of propagation.

Pushing constraints

Pushing constraints makes it possible that a constraint is not propagated after each modification of each decision variable; rather, it is propagated after all its variables have been propagated.

Suppose that there is a constraint C involving the decision variables x, y, z and t. Suppose that x is modified, then C is propagated. Now if the modification of x leads to a modification of y, z and t, then these decision variables will be propagated and each of those variables will propagate C again, even if these variables have not been modified by constraints other than C.

Imagine that you want to avoid these repeated propagations because you know that the `propagate` member function of C is costly. Also imagine that you want to propagate C once for all the modifications of the variables involved in C but not for the modifications of each variable individually.

There is a mechanism in CP Optimizer that meets these aims. It is possible to have constraints that are not propagated within demons. Such a constraint is not propagated after each modification of each decision variable; rather, it is propagated after all its variables have been propagated.

In fact, in CP Optimizer there are two mechanisms for propagating the modifications of variables.

- First Level: When a decision variable is modified, all the demons linked to this variable are called. This mechanism is repeated while there is a variable for which all the demons have not yet been called.
- Second Level: The `propagate` function of some constraint known as a global constraint is called.

If, after the termination of a `propagate` member function, a variable has been modified, then constraint propagation immediately return to the first level. This means that the process at the first level has greatest priority.

In the second level, a global constraint (say, `ct`) is pushed by a call to the `push` member function.

This member function indicates to CP Optimizer that `ct` is a global constraint and must be propagated by the second level mechanism. This member function must be called within a demon.

Consider the following example simplified for illustration: you want to define the constraint $x + y = s$ without considering modifications of s.

The code of that constraint could be something like this:

```
class MySumConstraintI : public IlcConstraintI {
    IlcIntVar _x, _y, _s;
public:
    MySumConstraintI(IloCP cp, IlcIntVar x, IlcIntVar y, IlcIntVar s);
    ~MySumConstraintI(){}; // empty destructor
```

```
    virtual void propagate ();
    virtual void post();
    void pushDemon();
};
void MySumConstraintI::propagate () {
  IlcInt m;
  // compute the min of s
  m =_x.getMin() + _y.getMin();
  _s.setMin(m);
  // compute the max of s
  m=_x.getMax() + _y.getMax();
  _s.setMax(m);
}
```

A demon for each variable x and y is added and that demon is triggered when the boundary of either variable is modified (that is, when a range event occurs).

```
void MySumConstraintI::post(){
  _x.whenRange(MySumConstraintI_pushDemon(getCP(),this));
  _y.whenRange(MySumConstraintI_pushDemon(getCP(),this));
}
```

Now here is the code for the demon:

```
ILCCTDEMON0(MySumConstraintI_pushDemon, MySumConstraintI, pushDemon);
void MySumConstraintI::pushDemon() {
  push();
}
```

How does this tactic differ from the conventional propagation mechanism? In this case, the demon will be called, but the propagate member function of `MySumConstraintI` will not be called immediately. Rather, `ct` is pushed. When there are no more demons to trigger, then the constraints that have been pushed will be called. Furthermore, in this way, a constraint will be propagated only once even if it has been pushed several times.

If you want to refine the propagation mechanism, for example, if you want to treat the variables that have been modified in a special way, you have to manage that treatment by using internal data structures within the constraint. For example, suppose that if x is modified, you want to call your own function `propagateX()` and if y has been modified, you want to call your own `propagateY()`. To do so, you add reversible Boolean data members to `MySumConstraintI`, like this:

```
    IlcRevBool _xIsModified;
    IlcRevBool _yIsModified;
```

Now you define special demons for x and for y, like this:

```
ILCCTDEMON0(MySumConstraintI_xDemon, MySumConstraintI, xDemon);
ILCCTDEMON0(MySumConstraintI_yDemon, MySumConstraintI, yDemon);
void MySumConstraintI::xDemon() {
  memorizeThatXisModified();
  push();
}
void MySumConstraintI::yDemon() {
  memorizeThatYisModified();
  push();
```

Now the propagate member function will look like this:

```
void MySumConstraintI::propagateX () {
  IlcInt m;
  // compute the min of s
  m =_x.getMin() + _y.getMin();
  _s.setMin(m);
```

```
  // compute the max of s
  m=_x.getMax() + _y.getMax();
  _s.setMax(m);
}
void MySumConstraintI::propagateY () {
  IlcInt m;
  // compute the min of s
  m =_x.getMin() + _y.getMin();
  _s.setMin(m);
  // compute the max of s
  m=_x.getMax() + _y.getMax();
  _s.setMax(m);
}
void MySumConstraintI::propagate(){
  if (_xIsModified){
    propagateX();
    _xIsModified.setValue(getCP(),IlcFalse);
  }
  if (_yIsModified){
    propagateY();
    _yIsModified.setValue(getCP(),IlcFalse);
  }
}
```

Thus you have created a global constraint that will be propagated only once and propagated only after the propagation of variables in that constraint.

Chapter 5. Writing goals

Customized constructive search can be implemented using search goals.

Overview

A custom search algorithm can be implemented by implementing the decision making logic using goals.

CP Optimizer offers a wide range of predefined search techniques, together with powerful tools such as search phases and custom evaluators and selectors that can be used with these techniques. These facilities usually suffice for solving the vast majority of problems. However, when you are developing an optimization application, there may be special circumstances in which search phases do not provide you with the flexibility to solve your specific problem. In this case, you may find it useful to write a custom search algorithm by implementing the decision making logic using goals.

This chapter explains what you must do if you decide to implement a custom search by subclassing the `IlcGoalI` class in the CP Optimizer engine extensions API.

Understanding goals

Customizable search goals enable you to write your own search strategy.

Overview

Customizable search goals enable you to write your own search strategy.

Goals are the building blocks used to implement search strategies in CP Optimizer, and the availability of customizable goals enables you to write your own search strategy. Goals implement algorithms where the exact sequence of operations to follow is not known in advance. This kind of programming is often called non-deterministic. To help with the task of writing a custom search, some simple strategies are provided as pre-defined goals, such as `IlcInstantiate`, which works to fix one decision variable, and `IlcGenerate`, which works to fix an array of variables.

When executed, a goal generally analyzes some information based on the current domains of the decision variables and then determines a division that would dichotomize the search space. To split the space, the goal creates two alternatives in the search. The goal will specify to the optimizer to try one alternative and save the other to try if the first choice leads to a failure. Along with trying an alternative, the goal execution generally calls another goal, perhaps itself again, to continue dichotomizing the search space.

Goals, as they are represented in Concert Technology, depend on two classes: `IloGoal` and `IloGoalI`. An instance of the class `IloGoal` (a handle) contains a data member (the handle pointer) that points to an instance of the class `IloGoalI` (its implementation object). Likewise, goals, as they are represented in CP Optimizer search, depend on two classes: `IlcGoal` and `IlcGoalI`. An instance of the class

IlcGoal (a handle) contains a data member (the handle pointer) that points to an instance of the class IlcGoalI (its implementation object) which is allocated on the CP Optimizer heap.

To define a new class of goals, you can use the macro ILCGOALn. If you want to use a new class of goals in a Concert Technology model, you can wrap it using the macro ILOCPGOALWRAPPERn.

A goal is executed by a call to IloCP::solve or an IloCP::startNewSearch/ IloCP::next pair. A goal can either succeed or fail. A goal fails if a fail member function (such as IlcGoalI::fail or IloCP::fail, for example) is called during its execution. Such a call may happen automatically when the domain of a decision variable becomes empty through domain reduction. A goal succeeds if it does not fail.

Using macros to create a simple goal

A custom search goal can be implemented by using a macro.

To illustrate the implementation of custom goals, consider a simple goal called HelloWorld. This goal, when executed, creates the output "Hello World" and succeeds. The easiest way to implement the "Hello World" goal is to use the macro ILCGOAL0:

```
ILCGOAL0(HelloWorld){
  IloCP cp = getCP();
  cp.out() << "Hello World" << std::endl;
  return 0;
}
```

This macro, when expanded, generates code similar to the following code:

```
class HelloWorldI : public IlcGoalI {
   public:
     HelloWorldI(IloCP cp);
     IlcGoal execute();
 };
 HelloWorldI::HelloWorldI(IloCP cp)
   :IlcGoalI(cp){}
 IlcGoal HelloWorld(IloCP cp){
   return new (cp.getHeap()) HelloWorldI(cp.getImpl());
 }
 IlcGoal HelloWorldI :: execute() {
   IloCP cp = getCP();
   cp.out() << "Hello World" << std::endl;
   return 0;
 }
```

With this example, you see that defining a goal using the ILCGOAL0 macro creates a subclass of IlcGoalI and an execution function:

- The subclass contains a virtual function used to execute the goal. Its name is execute. Its body is the body of the macro. When this function is called, the goal is *executed*.
- The function creates an instance of the goal. The name of this function is the name used as the argument to the macro.

To use this new class of goals in a Concert Technology model, you must wrap it using the macro ILOCPGOALWRAPPERn. To wrap the "Hello World" goal, you write:

```
ILOCPGOALWRAPPER0(HelloWorldConcert,cp) {
  return HelloWorld(cp);
}
```

This macro, when expanded, generates code similar to the following code:

```
class HelloWorldConcertI : public IloGoalI {
public:
  HelloWorldConcertI (IloEnvI*);
  ~HelloWorldConcertI ();
  virtual IlcGoal extract(const IloCP) const;
  virtual IloGoalI* makeClone(IloEnvI*) const;
};

HelloWorldConcertI::HelloWorldConcertI(IloEnvI* env) :
  IloGoalI(env) {}

HelloWorldConcertI::~HelloWorldConcertI () {}

IloGoalI* HelloWorldConcertI::makeClone(IloEnvI* env) const {
  return new (env) HelloWorldConcertI(env);
}

IloGoal HelloWorldConcert (IloEnv env) {
  return new (env) HelloWorldConcertI (env.getImpl());
}

IlcGoal HelloWorldConcertI::extract(const IloCP cp) const {
  return HelloWorld(cp);
}
```

With this example, you see that wrapping a goal using the ILOCPGOALWRAPPER0 macro creates a subclass of `IloGoalI` and an extraction function:

- The subclass contains a virtual function used to extract the goal. Its name is `extract`. Its body is the body written in the macro. When this function is called, the goal is *extracted*.
- The function creates an instance of the modeling layer goal. The name of this function is the name used as the argument to the macro.

At the Concert Technology level, an `IloGoal` is passed as an argument to the member functions `IloCP::solve` or `IloCP::startNewSearch`. The following code executes the “Hello World” goal from the main program:

```
IloModel model(env);
IloCP cp(model);
cp.setParameter(IloCP::LogVerbosity,IloCP::Quiet);
if (cp.solve(HelloWorldConcert(env)))
  cp.out() << "Success" << std::endl;
else
  cp.out() << "Fail" << std::endl;
  cp.end();
```

The execution of the goal succeeds and produces the output:

```
Hello World
Success
```

Composing goals

A goal can be defined as a sequence of goals that must all succeed.

A goal can be defined as a sequence of goals that must all succeed. At the Concert Technology level, goals can be composed using the `operator&&` function. For example, consider a custom goal “Greetings”:

```
ILCGOAL0(Greetings){
  IloCP cp = getCP();
  cp.out() << "Greetings" << std::endl;
```

```
  return 0;
}
ILOCPGOALWRAPPER0(GreetingsConcert,cp) {
  return Greetings(cp);
}
```

If this goal is composed with the custom "Hello World" goal from the previous section, calling the CP Optimizer search with the code:

```
IloModel model(env);
IloCP cp(model);
cp.setParameter(IloCP::LogVerbosity,IloCP::Quiet);
if (cp.solve(HelloWorldConcert(env) && GreetingsConcert(env)))
  cp.out() << "Success" << std::endl;
else
  cp.out() << "Fail" << std::endl;
cp.end();
```

produces the following results:

```
Hello World
Greetings
Success
```

Goals can be composed at the search level using the function `IlcAnd`. This function defines a goal composed of several subgoals. The subgoals are executed from left to right. For example, the previous goal composition could alternatively be written as:

```
ILOCPGOALWRAPPER0(ComposedGoalConcert,cp) {
  return IlcAnd(HelloWorld(cp), Greetings(cp));
}
```

The return type of the `execute` function of a subclass of `IlcGoalI` is an `IlcGoalI`, which provides yet another way to compose goals. Changing the final line of code of the function `HelloWorldI::execute` to be

```
  return Greetings(cp);
```

produces the same results as composing the goals with the `IlcAnd` function.

For a goal that is a conjunction of subgoals, every subgoal must succeed in order for the goal to succeed. If one of the subgoals in the conjunction fails, then the goal fails.

Goal stack

The goal stack determines the order of execution of the goals.

The member functions `IloCP::solve` and `IloCP::next` control the execution of goals. The first time one of these member functions is called, it creates a stack of goals, called the *goal stack*.

Note:

Goals must be called by `IloCP::solve`, a `IloCP::startNewSearch/IloCP::next` pair, or by other goals. Otherwise, they are not executed.

At the Concert Technology level, an instance of an `IloGoal` is passed as an argument to the member functions `IloCP::solve` or `IloCP::startNewSearch`. The function `IloCP::solve` pushes the corresponding search layer goal onto the goal stack of the invoking `IloCP` optimizer object and then executes the subgoal at the

top of the stack. The function IloCP::startNewSearch pushes the corresponding search layer goal onto the goal stack of the invoking IloCP optimizer object, but does not execute it immediately. The member function IloCP::next pops the goal which is on top of the goal stack, if any, and executes it. In both cases, when the current goal execution is finished, the goal at the top of the stack is popped and executed. Thus goals are executed in the order first in, last out. If the goal stack is empty, the call to solve or next terminates and returns IloTrue.

Note:

For goals composed of conjunctions (subgoals joined with IlcAnd or operator&&), they subgoals are pushed onto the goal stack starting with the rightmost goal in the sequence.

Choice points

A choice point is a choice between two subgoals of a goal.

A goal can be defined as a choice between subgoals; this choice of subgoals is referred to as a choice point and is the mechanism with which CP Optimizer can try alternatives in the search. In order for the goal to succeed, only one of the subgoals of the choice point must succeed. At the Concert Technology level, a choice point can be created using the operator|| function. For example, a choice point between the custom goals "Hello World" and "Greetings":

```
IloModel model(env);
IloCP cp(model);
cp.setParameter(IloCP::LogVerbosity,IloCP::Quiet);
if (cp.solve(HelloWorldConcert(env) || GreetingsConcert(env)))
  cp.out() << "Success" << std::endl;
else
  cp.out() << "Fail" << std::endl;
cp.end();
```

produces the following results:

```
Hello World
Success
```

Because the "Hello World" subgoal is executed first and succeeds, the "Greetings" subgoal is not pushed onto the goal stack and is not executed.

At the search level, choice points are implemented by the function IlcOr.

The member function IloCP::next produces, if possible, one successful execution of a given goal and stops. To continue to try the other choices, you should place IloCP::next within a loop.

Backtracking

Backtracking occurs when there is a failure in a goal and goal execution resumes at the most recent choice point with untried subgoals.

Within CP Optimizer search, the implicit generation of combinations of values for decision variables uses *constructive search strategies*. A constructive strategy attempts to build a solution by choosing a non-fixed decision variable and a value for this variable. The chosen variable is then fixed to the chosen value and constraint propagation is triggered. This operation is called *branching*, and the fixing is also called a "branch". Constraint propagation then reduces the domains of variables and, consequently, the currently possible combinations. After propagation

terminates, another non-fixed variable, if one exists, is chosen, and the process repeats until all decision variables are fixed. However, if a fixing fails because it cannot lead to a solution, the constructive strategy *backtracks* and chooses another value for the variable.

The fixing of a decision variable must be seen as a guess: if it leads to inconsistencies, it should be undone, and another value should be tried. CP Optimizer implements these guesses and "undos" by using choice points. Choice points are implemented at the search level in CP Optimizer by the function `IlcOr`. A choice point defines a goal in terms of a choice between subgoals.

CP Optimizer executes a choice point between two or more subgoals like this:

- The state of CP Optimizer (including the state of all variables and constraints and the state of the goal stack) is saved, so that it can be restored if needed. This is called setting the *choice point*.
- The first subgoal is pushed onto the top of the goal stack.
- The other subgoals are saved as untried subgoals for the choice point.
- Then the first subgoal is popped from the goal stack and executed. If this subgoal fails, the state of CP Optimizer is restored, and the first untried choice is pushed onto the goal stack. This activity is called backtracking, and it continues until a subgoal is found that succeeds.
- If all subgoals fail, the choice point itself fails.

When the function `IloCP::fail` is called, goal execution resumes at the most recent choice point with untried subgoals. It is possible to make goal execution resume at an earlier choice point if you associate labels with choice points. Then the function `IloCP::fail` can be called with a label. In such a case, goal execution resumes at the most recent choice point having the same label.

Subgoals

When writing a new class of goals, you must determine what the goal will return.

The return value of the `execute` function of a subclass of `IlcGoalI` is an `IlcGoalI`. When writing a new class of goals, you must determine what the goal will return. If there are no subgoals of the goal, then you can specify a return value of 0. This indicates that the goal has succeeded.

Otherwise, the goal must return a goal, a conjunction of goals, a choice point or some combination of these. It is important to note that the subgoal specified can be a constraint. If a constraint is specified as a subgoal, CP Optimizer adds the constraint to the optimizer. This constraint addition is reversible; when the optimizer backtracks over this subgoal, the constraint is removed.

Although the most common forms of constraint added are `==` and `!=`, you can also dichotomize with other more general constraints.

Writing your own goal

A custom goal can be written using a macro.

Overview

A custom goal can be written using a macro.

You can use the macro ILCGOALn to define a new class of goals. If you want to use this new class of goals in a Concert Technology model, you must first wrap them using the macro ILOCPGOALWRAPPERn.

Using ILCGOALn to define a new class of goals

A macro can be used to define a new class of goals.

You define a goal using the macro ILCGOALn. The definition consists of three parts:

- a name for the goal,
- a list of typed parameters, n being the number of parameters and
- a body which defines how to execute the goal.

For example, the following goals merely print something, but they are valid goal definitions:

```
ILCGOAL0(Print){
  IloCP cp = getCP();
  cp.out() << "Print: executing a goal without parameters\n";
  return 0;
}
ILCGOAL1(PrintX, IlcInt, x){
  IloCP cp = getCP();
  cp.out() << "PrintX: executing a goal with one parameter\n";
  cp.out() << x << std::endl;
  return 0;
}
ILCGOAL2(PrintXY, IlcInt, x, IlcFloat, y){
  IloCP cp = getCP();
  cp.out() << "PrintXY: executing a goal with two parameters\n";
  cp.out() << x << std::endl;
  cp.out() << y << std::endl;
  return 0;
}
```

The macro for creating a goal class also generates a function for creating an instance of the goal. For example, the following statements are valid calls in the search layer to create instances of the goals just defined:

```
  IlcGoal goal = Print(cp);
  IlcGoal goalX = PrintX(cp, 2);
  IlcGoal goalXY = PrintXY(cp, 1, 2.);
```

The goal creation function can be declared in a header file. Its signature has the same name and arguments as the goal, and it returns a pointer to an object of the type IlcGoal. It does not execute the goal.

Note:

Such pointers must not be stored since goals are automatically de-allocated by CP Optimizer after they have been executed. They can be passed as arguments as long as the goal has not yet been executed.

For example, the following declarations are valid declarations of these same goals:

```
IlcGoal Print(IloCP cp);
IlcGoal PrintX(IloCP cp, IlcInt x);
IlcGoal PrintXY(IloCP cp, IlcInt x, IlcFloat y);
```

You can also define a goal using subgoals. The following goal has three subgoals:

```
ILCGOAL0(PrintAll){
  IloCP cp = getCP();
  return IlcAnd(Print(cp), PrintX(cp, 2), PrintXY(cp, 1, 2.));
}
```

The execution of the goal `PrintAll` produces the following output:

```
Print: executing a goal without parameters
PrintX: executing a goal with one parameter
2
PrintXY: executing a goal with two parameters
1
2
```

Using ILOCPGOALWRAPPER to wrap the goals

A custom goal must be wrapped to be used in a model.

There are several situations in which you might need to wrap a goal in an instance of `IloGoal`:

- You want to use the goal as an argument of `IloCP::solve`.
- You want to use the goal as an argument of `IloCP::startNewSearch`.

Any time a goal is to be used within the Concert Technology layer, it must be wrapped. In such situations, use the macro `ILOCPGOALWRAPPERn` to wrap your goal.

For example, you can wrap the `PrintAll` goal in an instance of `IloGoal` in this way:

```
ILOCPGOALWRAPPER0(PrintAllConcert,cp) {
  return PrintAll(cp);
}
```

To create an instance of this goal you write:

```
  IloGoal goal = PrintAllConcert(env);
```

The `IloGoal` object returned can then be passed as an argument to the `IloCP::solve` and `IloCP:startNewSearch` functions. Both of these functions extract the instance of the `IloGoalI` to the optimizer. In the body of the macro `ILOCPGOALWRAPPERn`, you must take the arguments to the modeling layer goal and determine the extracted counterparts of these objects in order to create the search layer counterpart of the goal. For example, if the modeling layer goal takes a decision variable as an argument, then in the body of the macro you must determine the engine object extracted by the optimizer and pass this search decision variable to the search layer goal.

For example, if the goal takes a decision variable as an argument:

```
ILCGOAL1(MycGoal, IlcIntVar, var) {
  ...
  return 0;
}
```

Then in the macro to wrap the goal you need to determine the search decision variable that has been extracted from the model decision variable:

```
ILOCPGOALWRAPPER1(MyoGoal, cp, IloIntVar, x){
  return MycGoal(cp, cp.getIntVar(x));
}
```

The extraction of a goal does not extract and must not extract its arguments. Therefore, when using a goal you must make sure that the arguments will be extracted by the model. A good way to ensure this is to add an object to the model before using it as an argument to a goal.

Example of writing your own goal: implementing IlcInstantiate

The instantiation goal is used as an example of writing your own goal.

As an example of goal programming, you are going to look closely at how the function `IlcInstantiate` is implemented. This function takes a decision variable as its argument and fixes it. More precisely, this function selects a value in the domain of the decision variable and assigns that value to the decision variable. The CP Optimizer engine automatically propagates the constraints incident on this decision variable.

With the ideas about choice points and backtracking that you have just learned in mind, now you can implement a goal that is similar to `IlcInstantiate`.

To do so, you will use the following algorithm:

- If the decision variable is fixed (that is, it has already been assigned a value), do nothing and succeed.
- Otherwise, set a choice point between two subgoals. If a selector is specified, the first subgoal assigns a valued based on the selection criteria. Otherwise, the minimum value of the current domain of the decision variable is assigned to it.
- If a contradiction is detected (the domain of a decision variable is reduced to the empty set), execute the second subgoal of the choice point. The second subgoal removes the tried and failed value from the domain of the constrained variable and executes the goal again. Indeed, this execution will try another value from the domain of the decision variable and repeat the process.

Assigning a value to (or removing a value from the domain of) a decision variable is straightforward. To do so, use the CP Optimizer constraint operators `==` and `!=`.

The CP Optimizer code corresponding to `IlcInstantiate` for decision variables is quite simple. You will see a couple versions of it to clarify a few points about goals.

To define a goal, you use a macro of the form `ILCGOALn`, where `n` is the number of arguments of the goal. The arguments of `ILCGOALn` are the name of the goal, followed by the types and names of the goal's arguments. Then the macro is followed by the body of a C++ function. This body defines how to execute the goal. The return value of that function is the subgoal of the goal defined by the macro itself. When there are no such subgoals, the function must return 0.

Now you can define `MycInstantiate` as a choice point between two goals.

Before setting a choice point, CP Optimizer checks whether the decision variable has been fixed, and if not, CP Optimizer selects a value in its domain. The second subgoal recursively calls `IlcInstantiate`.

Here is a first version of `MycInstantiate`:

```
ILCGOAL1(MycInstantiate, IlcIntVar, var) {
  if (var.isFixed())
    return 0;
  IlcInt value = var.getMin();
```

```
  return IlcOr( var == value,
                IlcAnd( var != value,
                        MycInstantiate(getCP(), var)));
}
```

As the ILCGOALn macros define classes of IlcGoalI, you can use the standard C++ keyword this to refer to the current goal. With that possibility in mind, here is a second version of MycInstantiate:

```
ILCGOAL1(MycInstantiate, IlcIntVar, var) {
  if (var.isFixed())
    return 0;
  IlcInt value = var.getMin();
  return IlcOr( var == value,
                IlcAnd( var != value,
                        this));
}
```

(The actual implementation of the predefined goal IlcInstantiate is slightly different since the choice of the value can be indicated by a parameter, but these two versions indicate the substance of the implementation.)

If you want to have an IloGoal object that can then be passed as an argument to the IloCP::solve and IloCP:startNewSearch functions, use the macro ILOCPGOALWRAPPER1.

For example, you can define MyoInstantiate this way:

```
ILOCPGOALWRAPPER1(MyoInstantiate, cp, IloIntVar, x){
  return MycInstantiate(cp, cp.getIntVar(x));
}
```

To create an instance of this modeling layer goal you write:

```
  IloGoal goal = MyoInstantiate(env, x);
```

Programming tips

Programming tips are provided to help with writing efficient custom goals.

Arguments to goals

arguments to goals are computed immediately, even though goals are not executed immediately.

Goals are not executed immediately when they are created. However, their arguments are computed immediately, and these arguments are saved together with the goal on the goal stack. Those arguments and computations are used when the goal is executed.

Note:

For that reason, arguments should not contain pointers to automatic objects, where automatic objects are objects allocated on the C++ stack, that is, without using the new operator.

Consider the following goals:

```
ILCGOAL1(PrintI, IlcInt*, i){
  IloCP cp = getCP();
  cp.out() << *i << std::endl;
  return 0;
```

```
}
ILCGOAL1(CallPrintI, IlcInt*, i){
  IlcInt i = 0;
  return PrintI(getCP(), &i); // error here: i is an automatic
}
```

If the goal `CallPrintI` is executed, an instance of `PrintI` is created. This (among other effects) stores the address of `i`. Then the function executing the body of `CallPrintI` returns. Thus the address of `i` is no longer a valid address. Finally, the goal `PrintI` may be executed, causing an error since the address of `i` is no longer valid.

Another subtle point is that arguments are computed before goal execution actually takes place. That fact can lead to some problems. A simple example requires the use of a decision variable.

Consider the following incorrect code:

```
ILCGOAL1(WrongGoal, IlcIntVar, x){
  return IlcAnd(IlcInstantiate(x),
                PrintX(getCP(), x.getValue())); // error: x is not fixed
}
```

The goal was intended to be executed as: the decision variable should be fixed by the execution of the goal `IlcInstantiate`; then the execution of the goal `PrintX` should print its value. However, the argument of `PrintX` is computed before `IlcInstantiate` is executed. That order of events causes an error since `x` is not yet fixed. Indeed, `IlcInstantiate` is pushed onto the goal stack, but it is not executed immediately.

Here is the correct way to pass `x` as an argument (instead of passing its value):

```
ILCGOAL1(PrintVar, IlcIntVar, x){
  IloCP cp = getCP();
  cp.out() << x.getValue() << std::endl;
  return 0;
}
ILCGOAL1(RightGoal, IlcIntVar, x){
  return IlcAnd(IlcInstantiate(x),
                PrintVar(getCP(), x));   // no error here
}
```

As indicated in this lesson the macros `ILCGOALn` define subclasses of the class `IlcGoalI`. Arguments to the macro are data members of the new class. The most significant consequence of that fact is that modifications of those arguments in the body of the macro are saved. Such modifications often have unfortunate results, and you are strongly advised not to make such modifications.

Reversible data

Reversible data objects can be used to return data to its previous state upon backtracking.

Consider an object, such as a goal, that has a data member with a value that is modified during the search for a solution. The value assigned to this data member depends on the choices and the hypotheses made before this assignment.

If those choices fail to find a solution further on, the optimizer backtracks, and the computed value for this data member is no longer valid. For that reason, this data member must be a reversible data member. That is, the modifier of this data member must save the initial value of the data member before any modification, so

the optimizer is able to restore the value of the data member to its value before modification if any failure occurs later. CP Optimizer provides you with the predefined class `IlcRevInt`, the class of reversible integers, for just such a purpose.

In this example is a custom goal that works to assign values to the variables in an array, starting at the smallest index. Each time the goal is executed, the index is incremented. However, when the optimizer backtracks, this integer is not decremented, leading to a situation in which some decision variables are not fixed. For example, the following goal will not ensure that all decision variables in the array are fixed when the search terminates:

```
ILCGOAL2(MycBadGenerate, IlcIntVarArray, x,int,index){
  IloCP cp = getCP();
  index++;
  if (index > x.getSize())
    return 0;
  return IlcAnd(IlcInstantiate(x[index-1]),this);
}

ILOCPGOALWRAPPER1(MyoBadGenerate,cp,IloIntVarArray,x) {
  IlcInt index=0;
  return MycBadGenerate(cp,cp.getIntVarArray(x),index);
}
```

In order for the state of the index to be restored when the optimizer backtracks, you can use an instance of an `IlcRevInt`, as in the following code segment:

```
ILCGOAL2(MycGoodGenerate, IlcIntVarArray, x, IlcRevInt *, revIndex){
  IloCP cp = getCP();
  revIndex->setValue(cp,revIndex->getValue()+1);
  if (revIndex->getValue() > x.getSize())
    return 0;
  return IlcAnd(IlcInstantiate(x[revIndex->getValue()-1]),this);
}

ILOCPGOALWRAPPER1(MyoGoodGenerate,cp,IloIntVarArray,x) {
  IlcRevInt* revIndex = new (cp.getHeap()) IlcRevInt(cp);
  revIndex->setValue(cp,0);
  return MycGoodGenerate(cp,cp.getIntVarArray(x),revIndex);
}
```

Fixing all decision variables

All decision variables must be fixed as necessary.

When you use a custom goal, the optimizer may solve the problem and leave some variables not fixed. If you choose to use a custom goal, it is your responsibility to ensure that the decision variables are fixed as necessary.

Chapter 6. Writing custom scheduling constraints and goals

Custom constraints and goals for scheduling can be written using engine extensions.

Overview

Custom constraints and goals for scheduling can be written using engine extensions.

This chapter illustrates how to use the scheduling extension classes of CP Optimizer to:

- write new constraints;
- write new search strategies and, more particularly, implement chronological schedule building.

The aim of the samples is not to be more efficient than the automatic search procedures of CP Optimizer. Instead, the aim of the samples is to provide the user advice for deciding whether custom algorithmic development is necessary, and, if so, to provide search procedure heuristics principles, design implementation hints, and source code for building blocks. The source code of the examples discussed in this chapter can be found in the directory `examples/src/cpp` of your CP Optimizer distribution.

Working with interval variables

Scheduling constraints in the engine are built using the interval variables in the engine.

The primary reason for developing a custom search goal is the recognition of global knowledge of the model which allows for an efficient search to be built:

- The requirements of the problem do not fit with the automatic search, for example, in the sample `schedsearch_greedyrelax.cpp`, the objective is to build the schedule greedily at each step of the relaxation.
- The automatic search does not find a good enough solution to efficiently start the automatic optimization procedure. The solution the user-defined goal is then used as starting point for the optimization.
- You want to write a sophisticated search procedure. Examples of this, given in an academic context, are studies of search procedures which take advantage of the propagation algorithm, the reversibility properties and the search tree traversal delivered by CP Optimizer.

The basic principles useful in writing a goal involving instances of `IlcIntervalVar` include:

- building a chronological schedule by fixing the starting point and, eventually, the end point after candidate selection at the frontier of the partial schedule and
- interleave a goal that raises precedences or extends sequences with a chronological schedule decision.

An important point when writing a user-defined search procedure is fixing the presence of an interval. A general strategy is to decide its presence value before scheduling it (or making any actual decision that affects the interval domain).

In a model, there exist instances of interval (or integer) variables whose fixing depend on the fixing of some other interval and integer variables. The classic example of this dependence in CP Optimizer is the master interval of a spanning or alternative constraint; a master interval is fixed whenever the spanned or alternative intervals are fixed. Chronological schedule building decisions should not be made on this type of master interval. Eventually, due to specific knowledge of the problem, you are aware that there are some intervals whose fixing heavily depends upon some dominant intervals. In schedule building, it is best to wait for the dominant intervals to be fixed before making any (residual) decisions on the dominated interval.

The function `IlcGoal IlcSimpleCompletionGoal(IloCP cp);` returns a goal that fixes all variables of the model. This goal should be used whenever the user-define goal terminates with some variables remaining unfixed. Using this goal is mandatory whenever the model contains state functions (instances of `IloStateFunction`).

Writing a goal with interval variables

Custom search goals enable you to write your own search strategy.

The basic principles involved in writing a goal involving instances of `IlcIntervalVar` include:

- building a chronological schedule by fixing the start point and, eventually the end point after candidate selection at the frontier of the partial schedule;
- interleave a goal that raises precedences or extends sequences with a chronological schedule decision.

The primary reason for developing your own search goal is the recognition of global knowledge of the model which allows you to build an efficient search. In practice, there generally exist instances of interval variables whose fixing depend on the fixing of some other interval variables. The classic example of this dependence in CP Optimizer is the master interval of a spanning or alternative constraint; a master interval is fixed whenever the spanned or in alternative intervals are fixed. Chronological schedule building decisions should not be made on this type of master interval. The master intervals are dominated for decision building, and the spanned or alternative intervals are dominant in decision building. Due to specific knowledge of your problem, you will be aware that there are some intervals whose fixing heavily depends upon some dominant intervals; that is, they are essentially dominated. In schedule building, you should wait for the dominant intervals to be fixed before making any (residual) decisions on the dominated interval.

Note:

Note that when the presence of an interval is not fixed, a general strategy is to decide its presence value before scheduling it.

Writing a constraint with interval variables

Constraints link specified decision variables.

Writing a new constraint involving instances of `IlcIntervalVar` involves the complications of managing the presence status of interval variables. All constraint conditions must support the notion that "An absent interval does not belong to the schedule solution." In other words, discovering an absent interval cannot result in a filtering algorithm raising a failure.

Consider a set of constraints defined by a condition `C` on an interval variable `v`. If, given the actual ranges of the start, end or length of the interval, the condition `C` is violated, you must set the interval to be absent. The demon of event on the interval domain of `v` should look like.

```
if (v.isAbsent())
  return;
else if (isViolatedC(v))
  v.setAbsent();
else
  updateIntervalDomainUnderC(v);
```

where `isViolatedC(v)` and `updateIntervalDomainUnderC(v)` are member functions of the constraint class, where the latter method updates the start, end and length ranges of v.

Now, consider a set of constraints defined by a condition `C` on a pair of interval variables `v1` and `v2`. The change in the interval domain of `v1` has consequences on `v2` if and only if presence of `v2` implies presence of `v1` (that is the case in particular if `v1` is present). As a consequence, when this pairwise condition is violated, it can be concluded that `v1` and `v2` cannot be both present. For optional intervals, you generally know from the model the relationship between the presence statuses of the intervals in the pair; it is important to store this information in the constraint data structure. The demon of event on the interval domain of `v1` would look like:

```
if (isDone())
  return;
else if (v1.isAbsent() || v2.isAbsent())
  markDone();
else if (isViolatedC(v1, v2)) {
  IlcPresenceImpliesNot(v1, v2);
  markDone();
} else if (presenceImplies(v1, v2))
  updateIntervalDomainUnderCFrom1(v1, v2);
```

where `isDone()` and `presenceImplies(v1, v2)` are member functions of the constraint class.

The update of the interval domain must support hypothetical reasoning. That is, if the presence of `v1` implies the presence of `v2`, a change in the domain interval of `v2` may affect the domain interval of `v1`.

Writing a greedy search

A simple example is used to illustrate custom search for scheduling.

The example `schedsearch_greedandrelax.cpp` illustrates the design aspects of writing your own search by showing you how to:

- write a search that builds a chronological schedule with a very simple search that does not have a branch and bound tree traversal;
- write a dynamic selector for chronological list scheduling of the interval variables which only considers variables that are the actual decision variables;

- use the internal solve method of the instance of `IloCP` to iterate on schedule building procedures. This is a key point as writing an iterative improvement method follows the same principle.

To define a new search, you will have to create at least two separate sets of data. The first set of data, the model context, collects the information you need from the model, such as the instances of `IloIntervalVar` that need to be fixed, as well as all the structural and invariant information that may be needed to build the search. This set of data should be allocated on the `IloEnv` instance or by your own allocator. The second set of data, the engine context, contains all of the information that is used by the search algorithm to define the tree traversal and, eventually, the constraints that you may add to the solving engine. The engine context contains the counterparts of the model variables that are needed during the search (for example instances of `IlcIntervalVar`), a local copy of all structural and invariant information you may need to build the search and that may change during the search as well as reversible data (instances of `IlcRevInt` and `IlcRevAny`). This data set is allocated on the heap of the optimizer, the `IloCP` instance. The model context should never be used in the search, since the model environment data is not under the control of the solving engine. Conversely, the engine context must never point to data allocated outside of heap of the instance of `IloCP`. An instance of `IloGoal` in the model context should be wrapped in a goal wrapper that returns an instance of `IlcGoal`. The instance of the `IlcGoal` is created in the engine context and can be used to launch the search tree traversal. This design is strongly recommended as it clearly separates the data from the model declaration and the data structure of the solving procedure.

Writing a chronological scheduling goal

Custom goals on interval variables can be written using engine extensions.

In resource scheduling, the number of time points is often quite large, generally much larger than the number of interval variables. Any part of the problem manipulation, from modeling to solving and displaying, should avoid enumerating the time dimension. IBM ILOG Concert Technology and CP Optimizer allow a user to formally describe a scheduling problem without time enumeration by providing classes of interval variables and the resource description framework as sequences of functions defined on consecutive integer intervals. For solving, the same practice must also hold. Other than some very specific cases like proving optimality, the predefined algorithms of CP Optimizer, such as propagation, variable selection, decision making, search tree traversal or cost improvement methods, enumerate only the intervals. Two samples based on classical academic problems named the Resource Constraint Project Scheduling Problem (RCPSP) are provided, these examples illustrate how to build such a chronological schedule for two cases. The first example, `schedsearch_settimes.cpp`, illustrates the case in which all interval variables are constrained by the model to be present in the solution. In the other example, `schedsearch_optionalsettimes.cpp`, there are optional interval variables in the model. The interval variables are constrained by precedences and cumul function constraints (with alternative of cumul for the second sample). The samples present several variants of Set Times Chronological Schedule building, from the less generic and efficient to the most generic and efficient. In particular, the second sample illustrates managing non-scheduling decisions, specifically presence fixing, in a set times algorithm.

Writing a chronological scheduling goal on sequence variable

Custom constraints on interval sequence variables can be written using engine extensions.

When solving scheduling problems, the main difficulty is often fixing sequence variables with no overlap constraints. For these types of problems, a good practice is to build the subsequences before scheduling the sequenced interval variables. An advantage of this practice is that the decisions to make are fixing the presence of the intervals and fixing the immediate successor of the intervals in the sequence; that is, the decisions depend only on the number of intervals and are enumerable. To build a chronological schedule, the Head-Tail graph of sequence variables is perfectly suited. The example `schedsearch_sequencing.cpp` illustrates how to interleave sequencing decisions and schedule building. This sample is based on the classical academic problem called the Job Shop Scheduling Problem (JSSP); it is a simple example for which a greedy scheduling algorithm is enough to build the schedule. Each interval variable is constrained by precedences and belongs to a sequence (with alternative of sequence). The sample shows how to use the Head-Tail Graph of a sequence to implement a sequencing algorithm and how to interleave non-scheduling decisions (here sequencing) and partial scheduling completion.

Writing a custom constraint on a sequence variable

Custom constraints on interval sequence variables can be written using engine extensions.

Sequencing of non-overlaping intervals is an important concept in scheduling applications. To model a scheduling problem, the specific topology of a sequence of intervals or between sequences of intervals may need to be constrained. In some cases, a complex assembly of constraints may be used to express the constraint. For example, the sample `sched_pflowshop.cpp` illustrates modeling a permutation flow shop problem. This practice has limitations in expressivity and performance; in `sched_pflowshop.cpp`, a constraint per interval variable is added to the model, and these expressions are quite complex. The model for `sched_pflowshop.cpp` is limited to a perfect permutation. Supporting stronger conditions or relaxing conditions on the permutation is difficult. For example, this model does not work if some interval variables are optional and may be absent in the solution.

An interval sequence variable in the engine of CP Optimizer is an instance of the class `IlcIntervalSequenceVariable`. The internal data structure specialized for chronological scheduling is the head-tail graph. The sample `schedsearch_permutation.cpp` illustrates writing a permutation constraint that allows some interval variables to be absent in the solution.

Following the design in the example, you can write your own constraint on sequence variables. The events on the head-tail graph include changes in presence in the sequence of an interval variable, head and tail extensions, and the global update of the set of not sequenced intervals.

The filtering algorithms associated with head or tail extensions or changes in presence of an interval in the head or tail include:

- a filtering algorithm that processes the “last interval in head” changes that extend the head or remove a candidate for the head;

- a filtering algorithm that processes the “last interval in tail” changes that extend of the tail or remove of a candidate for the tail;
- a filtering algorithm that processes the “last interval in tail” or “last interval in head” changes when the sequence variable is sequenced by propagating the next relationship between the last interval in head and the last interval in tail;
- the maintenance of incremental support on an interval in head for which the condition of the constraint is bound before it in the head: the head boundary;
- the maintenance of incremental support on an interval in tail for which the condition of the constraint is bound after it in the tail: the tail boundary;
- an algorithm that propagates the change of an interval variable in head after the head boundary that considers the neighboring in head of this interval.
- an algorithm that propagates the change of an interval variable in tail after the tail boundary that considers the neighboring in tail of this interval;
- If possible, associated with the non sequenced event change set, a filtering algorithm that tries to remove candidate head and tail based on knowledge of the last present intervals in head and tail.

Index

Special characters

.NET 3

B

backtrack 41
branch 41

C

C++ 3
choice point 41
constraint
  custom 21
  example 25
  propagation 24
constraint propagator 5
constructive search 41
custom goal
  example 45

D

decision variable
  modifier 5

E

extension
  using 13
extractable
  constraint 17
  decision variable 17
  goal 18
extraction 14
  constraint 15
  decision variable 14

G

goal
  argument 46
  defining custom 43
  definition 37
goal stack 40

H

handle class 16

I

IlcAnd function 39
IlcConstraint class 17
IlcConstraintI class 25
IlcGoal class 18, 37
IlcIntVar class 17
IloCP class
  extract method 14
ILOCPCONSTRAINTWRAPPERn
  macro 27
ILOCPGOALWRAPPERn macro 37, 38
  using 44
IloCustomConstraint class 8
  execute method 5
IloGoal class 37
IloPropagatorI class 8
  addVar method 5
  execute method 5
implementation class 16
  goal 37
installing 2
invariant 8, 23

J

Java 3

M

modifier
  buffered 12
  elementary 5, 8, 22

P

platform specific information 3
preprocessing 15
propagation
  event 24
propagation queue 17
propagator 5
  defining 5, 10

S

search
  constructive strategy 41
  custom 37
search strategy 37

Printed in USA