exigen

User's Guide

process backbone

Constrainer

Exigen Rules 5.4.0

Decision Services

Document number: TP_Rules_5.4.0_Constrainer_User_1.2_SV
Revised: 12/16/2004
EXIGEN CONFIDENTIAL – FOR AUTHORIZED USERS ONLY

Important Notice

Information in this document, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice, may contain technical inaccuracies or typographical errors, and should not be construed as a commitment by Exigen Properties, Inc. and/or affiliates ("Exigen"). Exigen may make improvements and/or changes in the products and/or the programs described in this document at any time without notice. Exigen is not responsible or liable for any changes made to this document without Exigen's prior written consent. In particular, modifications in or to the data model may have occurred or may occur in a new product release after publication of this documentation. In accordance with Exigen standard support policy, any difficulties caused by a modified data model shall not be considered support issues and shall not be covered by Exigen maintenance and support. For further information, consult your support agreement.

Information published by Exigen on the Internet/World Wide Web may contain references or cross-references to Exigen products, programs and services that are not announced or available in your country. Such references do not imply that Exigen intends to announce such products, programs or services in your country. Consult your local Exigen business contact for information regarding the products, programs and services that may be available to you.

In no event will Exigen or its licensors be liable to any party for any direct, indirect, special or other consequential damages for any use of this product or its accompanying publications, including, without limitation, any lost profits, business interruption, loss of programs or other data on your information handling system or otherwise, even if we are expressly advised of the possibility of such damages. Exigen provides this product and its publications "as is" without warranties or conditions of merchantability or fitness for a particular purpose. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; in that case, this notice may not apply.

Copyright

Contact information

Telephone:		Ground mail:
1-866-4EXIGEN	North America	Exigen Properties, Inc.
1-800-GoExigen	Europe	Exigen Support Services
61-300-303-100	Australia	12 Smythe Street, 5th Floor
1-506-674-6922	All other countries	Saint John, NB, Canada, E2L 5G5
Hours of operation:		Website: http://www.exigengroup.com/
Monday through Friday 6:00 a.m. – 9:00 p.m. AST		Fax: 1-506-674-4014
Outside these hours, on-call 24x7 voicemail is answered within 15 minutes.		Email: support@exigengroup.com

Table of Contents

Preface

This preface is an introduction to the *Constrainer User's Guide*. It defines the audience, explains how to use this guide, and lists typographic conventions used throughout the guide.

The following topics are described in this section:

Audience

This guide is intended for developers who create optimization and decision support systems with constraint programming technology inside any Java development environment. The user must be familiar with the Java programming language.

How to Use This Guide

This section provides an overview of this guide's content as follows:

Information on how to use this guide	
Section	**Description**
Chapter 1	Defines Constrainer as a framework for the development of constraint-based engines.
Chapter 2	Provides several examples on how to solve basic constraint satisfaction problems, describes integer and boolean constrained variables and expressions and basic constraints.
Chapter 3	Provides more advanced constraint satisfaction problem examples. The chapter explains how to find a solution using goals, and describes constraint programming from the viewpoint of object-oriented programming.
Chapter 4	Provides an example and explains the main concepts of optimization problems.
Chapter 5	Explains how to solve scheduling problems using several examples and explains the major concepts.
Appendix A	Provides the complete source code for examples used within this guide.
Appendix B	Provides a formal description of constraint programming.
Glossary	Explains terms and abbreviations used within this guide.

Related Information

The following guides contain the additional information on Constrainer:

Related information	
Title	**Description**
Constrainer Installation Guide	Describes how to install, uninstall, and update Constrainer.

Typographic Conventions

The following styles and conventions are used in this guide:

Typographic styles and conventions	
Convention	**Description**
Bold	• Represents items such as field names, menus, menu commands, dialog boxes, windows, check boxes, option buttons, tabs, and command buttons. • Represents keys, such as **F9** or **CTRL+A.**
`Courier`	Represents file and directory names, code, system messages, and command-line commands.
Select **File > Save As**	Represents a command to perform, such as opening the **File** menu and selecting **Save As.**
Italic	• Represents any information to be entered in a field. • Represents documentation titles.
Hyperlink	Represents a hyperlink. Clicking on this field takes you to the identified place in this document.

Chapter 1: Introducing Constrainer

In the last 10 years, constraint programming (CP) became a leading technique for solving complex decision support problems in manufacturing, telecom, logistics, finance, and other industries. CP addresses problems such as job scheduling, resource allocation, planning, product configuration, and other optimization problems with many business constraints. CP provides an excellent foundation for the development of smart optimization engines for different decision support systems. Until recently, efficient constraint satisfaction environments were available only in the form of C++ or Prolog libraries, with ILOG Solver as the market leader among such products. On the Web, the combination of XML and Java became the de facto standard development environment. Consequently, an industrial strength constraint satisfaction environment for Java is rapidly becoming a must-have element in Web-based decision support. Constrainer was developed based on Java Constrainer, the first commercially available CP product for Java created by IntelliEngine, Inc. in 1999.

This chapter includes the following topics:

What Is Constrainer?

Constrainer is a subcomponent of Exigen Rules consisting of a Java package for the development of constraint-based optimization engines. It offers the methodology and tools to support real-world decision support systems. Incorporated in Exigen business solutions, Constrainer lowers cost, significantly reduces integration and deployment time, and results in effective decision support systems for financial, insurance, public sector and telecommunications organizations.

Since Constrainer is implemented as a pure Java package for constraint programming, it utilizes the following unique strengths of Java:

- user-friendly API
- platform independence
- rapid development

Constrainer supports the following features:

- integer, boolean, and floating point constrained variables
- major arithmetic, logical, and global constraints, and constrained expressions
- generic reversible environment
- efficient event notification and constraint propagation mechanisms
- reversible variables and user-defined actions

- symbolic constrained expressions
- basic scheduling classes, such as jobs and resources, and constraints on the classes
- pre-defined search goals and selectors
- ability to write problem-specific constraints and search algorithms
- smooth integration with any Java application without JNI

What Is Exigen Rules?

Exigen Rules is a framework for rapidly creating, deploying, and maintaining rules-based systems in financial services, insurance, telecommunications, and other industries. Although modern rule engines have already proven their efficiency, real-world experience shows that classifying, representing, and maintaining rules is still difficult. Exigen Rules solves this problem by providing customers with a unified methodology for building industry-specific rule templates that describe a diverse hierarchy of interrelated rules.

As part of the Exigen business solution, Exigen Rules offers extensible, responsive, and intelligent solutions to a dynamic business environment. There is a wide range of possible applications for which rules-based architecture is suitable.

Together, Exigen Rules and Constrainer comprise a business intelligence framework that allows developers to create Web-based decision support engines. While business rules could be used to define and modify the business problem, the proper optimization model can be expressed in terms of constraints and solved by a constraint engine based on Constrainer.

Tightly integrated with Exigen Rules, Constrainer can interpret business rules as constraints with an ability to violate such rules up to a certain degree as defined by the user. This hybrid approach extends rule technology to constraint-based decision support.

What Is Constraint Programming?

Constraint programming is a software technology for declarative description and effective solution of large combinatorial and optimization problems for decision support, planning, allocation resources, cutting materials, radio frequency distribution, and many other problems. The strength of constraint programming is based on separating the problem representation from problem algorithms.

Constraint programming operates in terms of variables and constraints that are imposed on these variables. The problem representation consists of the declaration of constrained variables and the imposition of problem constraints expressed in terms of the constrained variables. Solving a problem means finding such values for variables that all problem constraints are satisfied. The process of solving a problem is referred to as searching for the solution. By definition, there may be a number of solutions satisfying all the problem constraints, and it may be reasonable to find an optimal solution. Therefore, the initial problem may become an optimization problem. To compare optimality of different solutions, users must define a cost function. For information on solving optimization problems, see Chapter 4: Solving Optimization Problems.

Constraint programming involves the following distinct processes:

Constraint programming processes	
Process	**Description**
Problem representation	Declaration of the unknown problem variables and problem constraints. Constrainer provides common classes for constrained variables such as integer, float, and boolean. Each constrained variable is associated with its domain. The domain of a constrained variable defines the set of possible values for the variable. When the domain of a variable contains only one value, the variable is referred to as bound.
Searching for solution	Finding values for the constrained variables satisfying all the problem constraints. In optimization problems, Constrainer finds the solution among all solutions that minimizes the problem cost function. For simple or small-scale problems, it is sufficient to use the generic Constrainer search algorithm. For more complex or large-scale problems, users can define their own search algorithms using goal programming.

For a formal description of constraint programming, see Appendix B: Formal Description of Constraint Programming.

System Requirements

As a regular Java package, Constrainer requires only JDK 1.3 or later to be installed on the workstation.

Chapter 2: Solving Basic Problems

This section describes simple problems and Constrainer solutions to these problems.

The following topics are described in this section:

- Problem Solution Examples
- Reviewing Concepts

Problem Solution Examples

The following examples are described in this section:

Problem solution examples	
Example	**Description**
System of Equations	Describes a solution for a system of equations. This basic example demonstrates the use of symbolic constraints in Constrainer.
Map Colors Problem	Describes a solution for the map colors problem. This example demonstrates the use of constrained integer variables and expressions for combinatorial problems.
Magic Square Problem	Describes a solution for the magic square problem. It introduces a constrained integer expression array, and explains how to get and use the sum of array elements. This example also introduces the constraint `ConstraintAllDiff`.
Magic Sequence Problem	Describes a solution for the magic sequence problem. It explains how to use the method `distribute` to create powerful and concise constraints. This example also explains how to use redundant constraints to improve performance.
Family Riddle	Describes a solution for the family riddle problem. It summarizes the use of constrained integer expression arrays.

System of Equations

This example describes how to solve a system of equations.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

$X^2 + Y^2 < 20$

$X + Y = 5$

X and Y are positive integers.

Problem Solution

To solve the problem, proceed as follows:

1. Create an instance of the `Constrainer` class as follows:

```
Constrainer C = new Constrainer ("System of Equations 1");
```

2. For each unknown value, declare a constrained integer variable as follows:

```
IntVar X = C.addIntVar (-5, 5, "X");

IntVar Y = C.addIntVar (-5, 5, "Y");
```

3. It is clear that the variable modulus is less then five, so specify –5 and 5 as the minimum and maximum values of the variables.
4. Express the equations as constraints and post the letters to the `Constrainer` class as follows:

```
C.postConstraint(X.sqr().add(Y.sqr()).lt (20));

C.postConstraint(X.add(Y).eq(5));
```

5. Rewrite constraints in the symbolic form as follows:

```
C.postConstraint("X**2+Y**2<20");

C.postConstraint("X+Y=5");
```

6. Find a solution using the `Constrainer` class as follows:

```
C.execute (new GoalGenerateAll (new IntExpArray(C, X, Y)));
```

7. Run the program to produce the following output:

```
Solution 1: X[1], Y[4]
Solution 2: X[2], Y[3]
Solution 3: X[3], Y[2]
Solution 4: X[4], Y[1]
```

There are four solutions for the given system of equations.

For information on the complete example source code, see System of Equations Example Code. Complete source code is also located in the `Formula.java` file.

Map Colors Problem

This example provides a solution to the map colors problem.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

The map colors problem consists of choosing colors for the countries on a map in such a way that at most four colors are used and no two neighboring countries are the same color. This example uses the following countries:

- Belgium
- Denmark
- France
- Germany
- The Netherlands
- Luxembourg

The following table represents neighboring countries:

Neighboring countries	
Country	**Neighbors**
France	Belgium, Germany, Luxembourg.
Luxembourg	Germany, Belgium, France.
Germany	France, Denmark, Netherlands, Luxembourg.
Belgium	Netherlands, France, Luxembourg.
Denmark	Germany.
Netherlands	Belgium, Germany.

Problem Solution

To solve the problem, proceed as follows:

1. Represent the colors using integer numbers from 0 to 3.
2. Declare an integer constrained variable for each country representing its color on the map as follows:

```
IntVar Belgium = C.addIntVar (0, 3, "Belgium");
IntVar Denmark = C.addIntVar (0, 3, "Denmark");
IntVar France = C.addIntVar (0, 3, "France");
IntVar Germany = C.addIntVar (0, 3, "Germany");
IntVar Netherlands = C.addIntVar (0, 3,"Netherland");
IntVar Luxemburg = C.addIntVar (0, 3,"Luxembourg");
```

3. Combine the variables in an array to solve the problem with the aid of the `GoalGenerate` method as follows:

```
IntExpArray allVariables = new IntExpArray(
   C, Belgium, Denmark, France, Germany, Netherlands, Luxembourg);
```

4. Impose the problem constraints as follows:

```
C.postConstraint(
```

```
    ((France.ne(Belgium)).and(France.ne(Luxembourg))).and(France.ne(Germany)));
C.postConstraint((Luxembourg.ne(Germany)).and(Luxembourg.ne(Belgium)));
C.postConstraint( (Germany.ne(Netherlands)).and(Germany.ne(Denmark)));
C.postConstraint(Belgium.ne(Netherlands));
```

5. Solve the problem using the `Constrainer` class as follows:

```
C.execute(new GoalGenerate(allVariables));
```

The program provides the following colors for the countries:

```
Belgium[0]
Denmark[0]
France[1]
Germany[2]
Netherlands[1]
Luxemburg[3]
```

For information on the complete example source code, see Map Colors Problem Example Code.

Magic Square Problem

This example describes how to use the `sum()` method of the `IntExpArray` class and the `ConstraintAllDiff` constraint to solve the magic square problem.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

The problem includes finding a magic square, that is, a square matrix in which the sum of each row, column, and diagonal is identical. The numbers in the magic square are consecutive and start with 1.

Problem Solution

To solve the problem, proceed as follows:

1. To represent a magic square of size `n`, create an instance of `IntExpArray` with n^2 elements ranging from 1 to n^2 and post a constraint that states that all elements in the array are unique as follows:

```
IntExpArray vars = new IntExpArray(C, n*n, 1, n*n, "vars");
C.postConstraint(new ConstraintAllDiff(vars));
```

2. For each row, column, and diagonal, create an instance of the `IntExpArray` class with n elements ranging from 1 to n^2 and populate them from the `vars` array as follows:

```
IntExpArray[] rows = new IntExpArray[n];
IntExpArray[] columns = new IntExpArray[n];
```

```
IntExpArray diagonal1 = new IntExpArray(C, n);
IntExpArray diagonal2 = new IntExpArray(C, n);
for (i = 0; i < n; i++) {
   rows[i] = new IntExpArray(C, n);
   columns[i] = new IntExpArray(C, n);
   for (j = 0; j < n; j++) {
         rows[i].set(vars.get(i * n + j), j);
         columns[i].set(vars.get(j * n + i), j);
   }
   diagonal1.set(vars.get(i*n + i), i);
   diagonal2.set(vars.get(i*n + (n - i - 1)), i);
}
```

The sum of all numbers from 1 to n^2 is equal to $(n^2 \times (n^2 + 1))/2$, therefore the sum of every row, column, and diagonal is equal to $(n \times (n^2 + 1))/2$.

3. Impose constraints stating that the sums of the rows, columns, and diagonals are equal to $(n \times (n^2 + 1))/2$ using the `sum()` method of the `IntExpArray` class as follows:

```
int sum = n * (n * n + 1) / 2;
for (i = 0; i < n; i++) {
   C.postConstraint(rows [i].sum().eq(sum));
   C.postConstraint(columns [i].sum().eq(sum));
}
C.postConstraint(diagonal1.sum().eq(sum));
C.postConstraint(diagonal1.sum().eq(sum));
```

4. Find a solution using the `GoalGenerate` method as follows:

   The `execute(Goal goal)` method of the `Constrainer` class returns true if the goal is successfully achieved. For `GoalGenerate`, it means that a solution is found. If the solution is found, it is printed:

```
if (C.execute(new GoalGenerate (vars))) {
   for (i = 0; i < n; i++) {
         for (j = 0; j < n; j++) {
               System.out.print(vars.get(i * n + j).value()+" ");
         }
 System.out.println();
}
```

   The program output for the square of size 3 is the following:

```
2 4 9
6 8 1
7 3 5
```

For information on the complete example source code, see Magic Square Problem Example Code. Complete source code is also located in the `MagicSquare.java` file.

Magic Sequence Problem

This example describes how to use the `distribute(IntExpArray cards)` method of `IntExpArray` class to create powerful and concise symbolic constraints and solve the magic sequence problem.

The following topics are described in this section:

- Problem Definition
- Problem Solution
- Optimizing the Algorithm Using Redundant Constraints

Problem Definition

The problem consists of finding a magic sequence, that is a sequence of `n + 1` integer values $(x_0, x_1, \ldots, x_n)$, such that 0 appears in the sequence x_0 times, 1 appears x_1 times, 2 appears x_2 times and so on.

Problem Solution

All elements of an array are associated with their domains. The union of the domains that comprises the array domain is actually the set of all values that can occur in the array. An instance of the `IntArrayCards` class is associated with each instance of `IntExpArray`. This instance tracks the possible number of the value occurrences in the array. It is an array that has an element per value from the array domain. For each value from the array domain, the method `distribute(IntExpArray cards)` creates a constraint that controls the number of occurrences of the value. In the `distribute(IntExpArray cards)` method, the number of occurrences for each value of the array domain is specified as the `cards` parameter.

To solve the problem, proceed as follows:

1. Represent the magic sequence using an instance of the `IntExpArray` class as follows:

```
IntExpArray sequence = new IntExpArray(C, N + 1, 0, N, "MS");
```

2. Post the symbolic constraint using the `distribute(IntExpArray cards)` method as follows:

```
C.postConstraint(sequence.distribute(sequence);
```

3. Use the `Constrainer` class to find a solution as follows:

```
C.execute(new GoalGenerate(sequence));
```

4. Print out the solution as follows:

```
for (int i = 0; i < N; i++) {
   System.out.print(sequence.get(i).value());
}
```

```
System.out.println();
```

The program output for 10-element sequence is as follows:

```
7 2 1 0 0 0 0 1 0 0
```

For information on the complete example source code, see Magic Sequence Problem Example Code. Complete source code is also located in the `MagicSequence.java` file.

Optimizing the Algorithm Using Redundant Constraints

As you can see from the code example, the following is true:

$x_1*1+ x_2*2+…+ x_N*N=N+1$.

By adding a redundant constraint, the following results are obtained:

```
IntArray coeffs = new IntArray (C, N+1);
for (int i = 0; i < N; i++) {
coeffs.set(i, i);
}
C.postConstraint(sequence.mul(coeffs).eq(N+1));
```

This method improves the example performance by a factor of 2. The example execution time on a test computer decreased from 191 ms to 100 ms.

Note: Using redundant constraints can greatly narrow the search space by more effectively reducing the constrained variable domains during constraint propagation.

Family Riddle

This example described how to solve the family riddle.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

The following facts are known about this problem:

- Both Rene’s and Leo’s families have three girls and three boys.
- The youngest child in Leo's family is a girl.
- In Rene's family, a little girl was just born. In other words, there is a girl in Rene's family younger than one year.
- Neither family includes any twins, nor any children closer in age than a year.

- All children are under the age of ten.
- In each family, the sum of the ages of the girls is equal to the sum of the ages of the boys.
- The sum of the squares of the ages of the girls is equal to the sum of the squares of the ages of the boys.
- The sum of the ages of all these children is 60.

Problem Solution

To solve the problem, proceed as follows:

1. Use the following arrays to represent the children's ages:

```
IntExpArray ReneChildren = new IntExpArray (c, 6, 0, 9, "Rene");
IntExpArray LeoChildren = new IntExpArray (c, 6, 0, 9, "Leo");
```

2. Define a separate array for boys and girls as follows:

```
IntExpArray LeoSons = new IntExpArray (c, 3);
IntExpArray ReneSons = new IntExpArray (c, 3);
IntExpArray LeoDaughters = new IntExpArray (c, 3);
IntExpArray ReneDaughters = new IntExpArray (c, 3);
```

3. Populate these arrays from the children's age arrays as follows:

```
for (int i = 0; i < 3; i++) {
   LeoDaughters.set (LeoChildren.get (i) , i);
   LeoSons.set (LeoChildren.get (i + 3) , i);
   ReneDaughters.set (ReneChildren.get (i) , i);
   ReneSons.set (ReneChildren.get (i + 3) , i);
}
```

4. Impose the problem constraints by assuming that Leo's youngest girl age is the first element of the Leo's daughters age array as follows:

```
for (int i = 1; i < 6; i++) {
   c.postConstraint (LeoDaughters.get (0).lt (LeoChildren.get (i)));
}
```

5. Assume that Rene's newborn girl's age is the first element of the Rene's daughters age array as follows:

```
c.postConstraint (ReneDaughters.get (0).eq (0));
```

6. To express the constraint that neither family includes any twins, nor any children closer in age than a year, use the `AllDiff` constraints as follows:

```
c.postConstraint (c.allDiff (LeoChildren));
c.postConstraint (c.allDiff (ReneChildren));
```

7. To state that the sum of the boys ages and girls ages are equal, use the `sum()` function in the constraint as follows:

```
c.postConstraint (LeoSons.sum ().eq (LeoDaughters.sum ()));
c.postConstraint (ReneSons.sum ().eq (ReneDaughters.sum ()));
```

8. To express the constraint stating that the sum of the squares of the ages of the girls is equal to the sum of the squares of the ages of the boys, use the array scalar product as follows:

```
c.postConstraint (
   LeoSons.mul (LeoSons).eq (LeoDaughters.mul (LeoDaughters)));
c.postConstraint (
   ReneSons.mul(ReneSons).eq (ReneDaughters.mul (ReneDaughters)));
```

9. Sum the children's age arrays as follows:

```
c.postConstraint (
   ReneChildren.sum ().add (LeoChildren.sum ()).eq (60));
```

10. Find a solution using the `Constrainer` class:

```
c.execute (
   new GoalAnd (
         new GoalGenerate (ReneChildren),
         new GoalGenerate (LeoChildren)
   )
);
```

The program output is as follows:

```
[Rene(0)[0] Rene(1)[5] Rene(2)[7] Rene(3)[1] Rene(4)[3] Rene(5)[8]]
[Leo(0)[3] Leo(1)[7] Leo(2)[8] Leo(3)[4] Leo(4)[5] Leo(5)[9]]
```

Note that the first three array items designate the girls' ages and the last three designates the boys' ages.

For information on the complete example source code, see Family Riddle Example Code. Complete source code is also located in the `Family.java` file.

Reviewing Concepts

In the previous section, several simple problem solution examples were presented. These examples introduced important basic concepts. In this section, the major concepts supported by Constrainer are reviewed, and a more detailed description is provided.

The following topics are described in this section:

- Understanding Integer Variables and Expressions
- Understanding Boolean Variables and Expressions
- Understanding Constraints

Understanding Integer Variables and Expressions

Integer variables are the most frequently used constraint variables in constraint programming.

The following topics are described in this section:

- Declaring Constrained Integer Variables
- Understanding Integer Domains
- Using Integer Variables in Expressions
- Using Constrained Boolean Expressions
- Using Symbolic Constraints on Integer Variables
- Using Arrays of Constrained Integer Variables

Declaring Constrained Integer Variables

Constrainer provides the following `Constrainer` class methods to create new constrained integer variables:

`Constrainer` class methods for creating integer variables	
Method	**Description**
`IntVar addIntVar(int min, int max)`	Creates an anonymous variable, that is, a variable with no name, with the lower bound equal to `min` and the upper bound equal to `max`.
`IntVar addIntVar(int min, int max, String name)`	Creates a variable that has a name. The string `name` is associated with the variable and is displayed whenever the variable is printed. The lower bound is equal to `min` and the upper bound is equal to `max`.
`IntVar addIntVar(IntExp exp)`	Creates an anonymous variable with the lower and upper bounds equal to the lower and upper bounds of the constrainer integer expression `exp`. The method posts the equals constraint on the expression and the new variable.
`IntVar addIntVar(int min, int max, int type)`	Creates an anonymous variable with the lower bound equal to `min` and the upper bound equal to `max`. The parameter `type` specifies the domain type of the new variable. For information on domain types, see Understanding Integer Domains.
`IntVar addIntVar(int min, int max, String name, int type)`	Creates a variable with the lower bound equal to `min` and the upper bound equal to `max`. The string `name` is associated with the variable and is displayed whenever the variable is printed. The parameter `type` specifies the domain type of the new variable. For information on domain types, see Understanding Integer Domains.

Understanding Integer Domains

A constrained variable is associated with its domain. An integer constrained variable can be associated with several types of domains. Domain types are described in the following table:

Domain types	
Type	**Description**

Domain types	
Type	**Description**

`DOMAIN_PLAIN`	Domain with only minimum and maximum values. An integer variable associated with this domain can only decrease its maximum or increase its minimum. The following is an example using this type:
	`Constrainer c = new Constrainer ("test");`
	`IntVar l_Var = c.addIntVar(0, 1000, "a_var", DOMAIN_PLAIN);`
	This domain type provides the fastest constraint propagation. In addition, through bits domains, more particular constraint propagation is possible.

`DOMAIN_BIT_FAST`	Provides a flag for each integer value in the variable range indicating whether the value belongs to the variable domain. The domain representation consumes memory in ratio to the initial domain size, but is fast. The following is an example using this type: `IntVar l_Var = c.addIntVar( 0, 1000, "a_var", DOMAIN_BIT_FAST);`
`DOMAIN_BIT_SMALL`	Consumes an order of magnitude less memory then `DOMAIN_BIT_FAST` at the expense of some performance degradation. The following is an example using this type: `IntVar l_Var = c.addIntVar(0, 1000, "a_var", DOMAIN_BIT_SMALL);`

If the integer variable domain type is not specified, Constrainer uses the following policy:

- If the domain size is less than or equal to 16, the domain type `DOMAIN_BIT_FAST` is used.
- If the domain size is less than or equal to 128, the domain type `DOMAIN_BIT_SMALL` is used.
- In all other cases, the domain type `DOMAIN_PLAIN` is used.

Using Integer Variables in Expressions

Users can compose integer expressions in Constrainer using constrained integer variables and ordinary integers. The simplest integer expression is a constrained variable. To compose integer expressions, the following functions with the parameter `value` of `IntExp` class can also be used:

Integer expressions	
Function	**Corresponding value**
`neg();`	`- this`
`add(int value);`	`this + value`
`sub(int value);`	`this - value`
`mul(int value);`	`this * value`
`div(int value);`	`this / value`
`mod(int value);`	`this % value`
`sqr();`	`this * this`
`abs();`	`Absolute value of this`
`pow(int value);`	`IntExp to the value power if value >= 0`
`minOf(IntExp value);`	Minimum of two integer expressions: `this` and `value`

Integer expressions	
`maxOf(IntExp value);`	Maximum of two integer expressions: `this` and `value`

Using Constrained Boolean Expressions

Users can create constrained boolean expressions for constrained integer expressions using the following `IntExp` methods:

`IntExp` methods used for boolean expressions	
Method	**Description**
`eq(int value);`	`this == value;`
`ne(int value);`	`this != value;`
`ge(int value);`	`this > value;`
`gt(int value);`	`this >= value;`
`le(int value);`	`this < value;`
`lt(int value);`	`this <= value;`

Constrainer provides all previously listed functions with the parameter `value` of the `IntExp` class. The following is an example of a constrained boolean expression:

```
IntBoolExp l_More = l_Exp1.gt (10);
```

Boolean constrained expressions can be used as constraints. For example, to impose a boolean expression as a constraint, use the `postConstraint(IntBoolExp ct)` method of the `Constrainer` class as follows:

```
c.postConstraint (l_More);
```

Using Symbolic Constraints on Integer Variables

One of the most powerful features provided by Constrainer is the symbolic representation of constraints. The example described in this section shows how representing constraints as symbols makes code easier to read and maintain.

Consider the following constrained integer variables:

- `IntVar l_x = c.addIntVar(0, 100, "x");`
- `IntVar l_y = c.addIntVar(0, 100, "y");`
- `IntVar l_z = c.addIntVar(0, 100, "z");`

The following is an example of code that uses the declared variables:

```
c.postConstraint ("(x + 10) * (y - z**3) > x * y * z");
```

Note that the name parameter of constrained integer variables is used to symbolically represent the constraints.

The preceding line of code is equivalent to the following line, but the preceding line is much easier to read and understand:

```
c.postConstraint (

l_x.add(10).mul(l_y.sub(l_z.pow(3))).gt(l_x.mul(l_y.mul(l_z)))

);
```

Using Arrays of Constrained Integer Variables

An array can be used to present many variables of a similar type. Constrainer provides the `IntExpArray` class for representing an array of constrained integer expressions.

By using Constrainer, the following expressions can be created:

- constrained integer expression that represents the sum of all array elements or a cardinality of a specified integer in the array
- constraint that controls the number of occurrences of the specified values in the array
- constraint that controls that all the array elements are different

Constrainer provides the following `IntExpArray` constructors to declare a constrained integer expression array:

`IntExpArray` constructors	
Constructor	**Description**
`IntExpArray(Constrainer c, int size);`	Creates an empty array of the specified size.
`IntExpArray(Constrainer c, int size, int min, int max, String array_name)`	Creates an array of the specified size populated with constrained integer variables with domains ranging from `min` to `max` and named from `array_name[0] to array_name[size - 1]`.
`IntExpArray(Constrainer c, Vector v);` `IntExpArray(Constrainer c, FastVector v);`	Creates a constrained integer expression array from vectors of constrained integer expressions.

`IntExpArray(Constrainer c, IntExp exp0);`
…
`IntExpArray(Constrainer c, IntExp exp0, IntExp exp1, IntExp exp2, IntExp exp3, IntExp exp4, IntExp exp5, IntExp exp6, IntExp exp7, IntExp exp8);`
	Creates and simultaneously populates the array by specifying its elements up to nine items.

After creating the array, users can access its elements using the methods `get(int idx)` and `set(IntExp exp, int idx)` of the `IntExpArray` class.

Improving Performance with Redundant Constraints

Redundant constraints are logical consequences of other constraints participating in the problem statement. They cannot influence the problem solution that Constrainer finally finds, but they may improve performance by enhancing the propagation degree. For an example of redundant constraints, see Magic Sequence Problem Example Code.

Understanding Boolean Variables and Expressions

Constrained boolean variables and expressions are usually used to represent the results of logical operations of integers and float constrained expressions. Boolean variables and expressions can also be used as standalone values to represent logical constraints. Constrainer provides the `IntBoolVar` interface for constrained variables with domains consisting of the following two elements:

- `true`
- `false`

The following topics are described in this section:

- Declaring Constrained Boolean Variables
- Using Boolean Variables in Expressions
- Using Boolean Variables and Expressions as Integer Variables and Expressions

Declaring Constrained Boolean Variables

Boolean expressions and variables must be declared before being used in the program. Constrainer provides the following methods of the `Constrainer` class to create new constrained boolean variables:

`Constrainer` class methods for boolean variables creation	
Method	**Description**
`IntBoolVar addIntBoolVar();`	Creates an anonymous boolean variable.
`IntBoolVar addIntBoolVar(String name);`	Creates a variable associated with the string `name`. The string is displayed whenever the variable is printed.

Using Boolean Variables in Expressions

Users can compose constrained boolean expressions in Constrainer using constrained boolean variables, expressions, and constants. The simplest boolean expression is a constrained variable. Constrained boolean constants are instances of the `IntBoolExpConst` class. To create boolean constants, Constrainer provides the following constructor:

```
IntBoolExpConst(Constrainer c, boolean value);
```

The preceding constructor creates a boolean constant initialized to `value`.

To compose a boolean expression, use the following `IntBoolExp` and `IntBoolVar` interface methods:

- `not();`
- `and(IntBoolExp exp); and(boolean value);`
- `or(IntBoolExp exp);  or(boolean value);`
- `xor(IntBoolExp exp); xor(boolean value);`
- `implies(IntBoolExp exp); implies(boolean value);`

Using the method `asConstraint()`, a constrained boolean expression can be converted to the constraint that states that the boolean expression is true. The following is an example of creating a constraint using boolean expressions:

```
Constraint l_Cstr = l_BoolExp1.and(l_BoolExp2.implies(l_BooExp3)).asConstraint();
```

Using Boolean Variables and Expressions as Integer Variables and Expressions

The `IntBoolExp` interface is a descendant of the `IntExp` interface, and the `IntBoolVar` interface is a descendant of the `IntVar` interface. Therefore, constrained boolean expressions and variables can be treated as constrained integer expressions and variables with domains consisting of 0 corresponding to the `false` value and 1 to the `true` value, that is, you can add, subtract, multiply, divide and perform other arithmetic operations with boolean expressions just as you can with integer expressions. Using this feature, you can create constraints that are simultaneously complex and concise.

For example, a constraint stating that exactly two of the three boolean expressions are true can be written in the following way:

```
Constraint l_Cstr=l_BoolExp1.add(l_BoolExp2.add(l_BoolExp3)).eq(2).asConstraint();
```

Understanding Constraints

The following topics are described in this section:

Constraints Overview

In Constrainer, a constraint is a reflection of a constraint in the real world expressed in the terms of Constrainer variables and expressions. The most trivial constraints bind only one variable as in the following example:

```
IntVar day_of_week = C.addIntVar(1, 7);
C.postConstraint (day_of_week.gt(5)); // week-end
```

More complex constraints bind several variables as in the following example:

```
IntVar pack_size = C.addIntVar(0, 10);
IntVar pack_number = C.addIntVar(0, 1000);
IntVar required_number = C.addIntVar(0, 10000);
C.postConstraint (pack_size.mul(pack_number).eq(required_number));
```

After a constraint is composed, it can be activated immediately or some time later. The `Constrainer` methods `postConstraint(Constraint constraint)` and `postConstraint(IntBoolExp constraint)` activate the constraint immediately, while the methods `addConstraint(Constraint constraint)` and `addConstraint(IntBoolExp constraint)` postpone the constraint activation until

the method `postConstraints()` is executed. The method `postConstraints()` activates all constaints that are previously added but not activated.

Composing Constraints

In most cases, constraints in Constrainer are created using the `IntBoolExp` class, that is, problem constraints are composed as constrained boolean expressions and posted as constraints. The only exception to this method is symbolic constraints.

Symbolic constraints can be logically composed of other constraints using the `and(Constraint constraint)` method of the `Constraint` interface.

The following methods of the `IntExpArray` class are used to create symbolic constraints:

- `distribute(IntExpArray values, IntExpArray cards);`
- `distribute(IntExpArray values, IntArray cards);`
- `distribute(IntExpArray values, int [] cards);`
- `distribute(IntArray values, int n);`
- `allDiff();`

Using Metaconstraints

Boolean constrained variables and expressions are in fact integer constrained variables and expressions with a domain consisting of 0 for false and 1 for true. The representation makes it possible to use boolean constrained variables and expressions as a part of an integer expression. The following example is a constraint stating that only a certain number of other constraints are satisfied:

```
IntBoolExp  ctr1, ctr2, ctr3;
…
Constraint  ctr_2_of_3 = ctr1.add(ctr2.add(ctr3)).eq(2).asConstraint();
```

In the preceding example, there is a constraint `ctr_2_of_3` stating that exactly two of the three constraints are satisfied.

The integer representation of constraints enables solving the problem of minimizing the number of constraints that are violated in the case when the solution does not exist. If a more sophisticated cost function is used, this function could take into account an explicit hierarchy among constraints. In other words, such a function can be used to express constraint priorities.

Chapter 3: Solving Constraint Satisfaction Problems

This section describes more advanced examples that explain how to use goals to search for a solution. It also describes constraint programming from the object-oriented programming viewpoint.

This chapter describes the following topics:

- Problem Solution Examples
- Reviewing Concepts

Problem Solution Examples

The following examples are described in this section:

Problem solution examples	
Example	**Description**
Warehouse Maintenance Problem	Explains the warehouse maintenance problem. It demonstrates an advanced usage of integer constrained variable arrays.
Eight Queens Problem	Explains how to solve the eight queens problem. It demonstrates performance improvements using variable selectors.
Family Riddle 2	Explains how to solve the family riddle using object-oriented programming.

Warehouse Maintenance Problem

This example explains the warehouse maintenance problem.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

In this example, a company is considering a number of locations for building warehouses to supply its existing stores. Each possible warehouse has a fixed maintenance cost and a maximum capacity specifying how many stores it can support. In addition, each store can be supplied by only one warehouse and the supply cost of the store varies according to the warehouse selected.

The problem is to find which warehouses must be built while minimizing the total cost, consisting of warehouse maintenance costs and supply costs.

Problem Solution

To solve the problem, proceed as follows:

1. Declare initial conditions as follows:

```
// warehouse maintenance cost
int fixed = 30;
// number of stores
int nbStores = 10;
// number of warehouses
int nbWarehouses = 5;
// warehouse to store supply cost matrix
IntArray supplyCost [] = new IntArray [nbStores];
```

2. Introduce the unknown variables as follows:

```
// the array of conditions showing whether a warehouse is opened
IntExpArray openWarehouses = new IntExpArray(c, nbWarehouses);
for (int i = 0; i < nbWarehouses; i++) {
   openWarehouses.set(new IntBoolVarImpl(c), i);
}
// the array of store assignment to warehouses
IntExpArray storeAssign =
    new IntExpArray(c, nbStores, 0, nbWarehouses - 1, "SA");
```

3. Impose problem constraints on the unknown variables as follows:

```
// if a store is assigned to a warehouse then the warehouse is opened
for (int i = 0; i < nbStores; i++) {
   for (int j = 0; j < nbWarehouses; j++) {
         c.postConstraint(
         storeAssign.get(i).eq(j).implies(
               (IntBoolExp)openWarehouses.get(j)
               )
         );
         }
   }
```

4. Construct the cost function as follows:

```
// warehouse->store transition cost array
IntExpArray transCost = new IntExpArray(c, nbStores);
for (int i = 0; i < nbStores; i++) {
   transCost.set(supplyCost [i].get(storeAssign.get(i)), i);
}
// the first summand of the cost function is the transition cost sum
IntExp cost = transCost.sum ();
```

```
// the second summand of the cost function is
// the maintenance cost of warehouses
cost = cost.add (openWarehouses.sum ().mul(fixed));
```

5. Find the optimal solution using the `Constrainer` class as follows:

```
c.execute(
   new GoalFastMinimize(
         new GoalGenerate (openWarehouses.add(storeAssign)),
               cost
         )
);
```

6. Print out the solution as follows:

```
System.out.println("Optimal cost    : " + cost.value());
System.out.print("Open warehouses :" );
for (int i = 0; i < nbWarehouses; i++) {
   System.out.print("  " + openWarehouses.get(i).value());
}
System.out.println();
System.out.print("Store assignment:");
for (int i = 0; i < nbStores; i++) {
   System.out.print(" " + storeAssign.get(i).value());
}
```

The program output for the test is as follows:

```
Optimal cost: 964

Open warehouses:  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1

Store assignment: 19 15 15 19 15 15 19 15 15 19 15 15 19 15 15 19 15 15 19 15 15
```

For information on the complete example source code, see Warehouse Maintenance Problem Example Code.

Eight Queens Problem

This example explains how to solve the eight queens problem.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

Integer variable selectors can be used to solve the widely known problem of eight queens. The eight queens problem involves placing eight queens on a chessboard in such a way that none of them can capture any other using the conventional moves allowed for a queen. In other words, the problem is to select eight squares on a chessboard so that any pair of selected squares is never aligned vertically, horizontally, or diagonally. Of course, the problem can be generalized to a board of any size. In general terms, we have to select `n` squares on a board with `n` squares on each side, still respecting the constraints of non-alignment.

Problem Solution

To represent the problem, use an instance of `IntExpArray` where the index of array elements indicates the queen number and its column number simultaneously. According to the problem statement, all queens must be placed on different columns. The element instance of `IntExpArray` indicates the row number of the queen. If `x[i]` is an `i-th` element of the instance of `IntExpArray`, the relations `x[i] + i != x[j] + j and x[i] - i != x[j] - j` guarantee that `i-th` and j-th queens are placed at the distinct diagonals.

1. Using the instance of `IntExp` and the previously defined relations, declare the following:

```
// board size and simultaneously the number of queens
int board_size = 8;
Constrainer C = new Constrainer("Queens");
// array of queens rows
IntExpArray x = new IntExpArray(C, board_size);
// auxillary arrays
IntExpArray x = new IntExpArray(C, board_size);
IntExpArray x2 = new IntExpArray(C, board_size);

for(int i=0; i < board_size; i++) {
   IntVar variable =
         C.addIntVar(0, board_size -1, "q"+i, IntVar.DOMAIN_BIT_SMALL);
   x.set(variable, i);
   x1.set(variable.add(i), i);
   x2.set(variable.sub(i), i);
}
// all rows are different
C.postConstraint (C.allDiff(x));
// x[i] + i != x[j] + j
C.postConstraint (C.allDiff(x1));
// x[i] - i != x[j] - j
C.postConstraint (C.allDiff(x2));
```

```
C.printInformation(); // statistics output is on
// searching for solution
C.execute (new GoalGenerate(x));
```

The execution statistics for the test case of 24 queens is the following:

```
Choice Points: 63790 Failures: 63778   Undos: 1432239    Notifications: 625780
Memory: 242832 Time: 12908msec
```

2. Change the last string of the program as follows:

```
C.execute (new GoalGenerate(x, new IntVarSelectorMinSizeMin(x), false));
```

The preceding modification makes execution more than 100 times faster as follows:

```
Choice Points: 29    Failures: 14      Undos: 861  Notifications: 203 Memory:
57888    Time: 90msec
```

The performance improvement is achieved due to using a specialized search strategy. `GoalGenerate` provides two options for the search process to be customized. These options allow users to choose the next variable to bind, and choose the next value to try in the variable selection.

For information on the complete example source code, see Eight Queens Problem Example Code.

Family Riddle 2

This example explains how to solve the family riddle using object-oriented programming.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

For the family riddle definition, see Family Riddle.

Problem Solution

To solve the problem, proceed as follows:

1. Represent a family having three sons and three daughters as the following class:

```
public class OOFamily {
   private String m_father;
   private IntExpArray m_sons;
   private IntExpArray m_daughters;
   …
}
```

In this example, the class members `m_sons` and `m_daughters` represent the children's ages.

2. Combine all the children's ages into one array as follows:

```
private IntExpArray m_children;
```

3. Store the sum of the children's ages as follows:

```
private IntExp m_totalAge;
```

4. Define the `constructor` class as follows:

```
public OOFamily(Constrainer constrainer, String father) {
   m_father = father;
   m_constrainer = constrainer;

   m_daughters =
         new IntExpArray(constrainer, 3, 0, 9, m_father + "'girls");
   m_sons =
         new IntExpArray(constrainer, 3, 0, 9, m_father + "'s boys");
   m_children = new IntExpArray(constrainer, 6);
   for (int i = 0; i < 3; i++) {
         m_children.set(m_sons.elementAt(i), i);
         m_children.set(m_daughters.elementAt(i), i + 3);
   }
   m_totalAge = m_children.sum();
   imposeConstraints();
}
```

This class must guarantee that the newly created instance designates the following facts:

- The family has three girls and three boys.
- The sum of the ages of the girls is equal to the sum of the ages of the boys.
- The sum of the squares of the ages of the girls is equal to the sum of the squares of the ages of the boys.

5. In the `imposeConstraints()` method, add conditions that ensure that the boys' and girls' ages are related according to the problem statement as follows:

```
private void imposeConstraints () {
try {
   // no twins
   m_constrainer.postConstraint(m_constrainer.allDiff(m_children));
   // boys ages sum equal girls ages sum
   m_constrainer.postConstraint(m_sons.sum().eq(m_daughters.sum()));
   // boys ages squares sum equal girls ages squares sum
   m_constrainer.postConstraint(
```

```
            m_sons.mul(m_sons).eq(m_daughters.mul (m_daughters)));
      // constraints to remove symmetric solutions
      for (int i = 0; i < 2; i++) {
            m_constrainer.postConstraint(
                  m_sons.elementAt(i).lt(m_sons.elementAt(i + 1)));
            m_constrainer.postConstraint(
                  m_daughters.elementAt(i).lt(m_daughters.elementAt(i + 1)));
            }
      } catch (Exception e) {
            System.out.println (e);
      }
}
```

The constraints inside the cycle are important. The constraints align the children's ages in arrays in increasing order, so that the solutions corresponding to the transposition of any solution are represented as the only solution. These constraints help to avoid different representations of a solution.

6. Find the solution as follows:

```
Constrainer C = new Constrainer("OOFamily");

OOFamily LeoFamily = new OOFamily(C, "Leo");
OOFamily ReneFamily = new OOFamily(C, "Rene");

// Leo's last child is a girl
C.postConstraint(LeoFamily.getDaugther(0).lt(LeoFamily.getSon(0)));

// the Rene's youngest girl is under a year
C.postConstraint(ReneFamily.getDaugther(0).eq(0));

// all the children age sum is 60
C.postConstraint(
       LeoFamily.getTotalAge().add(ReneFamily.getTotalAge()).eq(60));

C.execute(
       new GoalGenerate(
              LeoFamily.getChildren().merge(ReneFamily.getChildren())
    )
);
LeoFamily.print();
```

```
ReneFamily.print();
```

The previous example shows how inherent constraints are used in the class definition and the intrinsic constraints are used in the problem representation section. An efficient constraint subdivision makes the problem representation clearer and easier to maintain.

For information on the complete example source code, see Family Riddle 2 Example Code.

Reviewing Concepts

The basic concept of search strategies in Constrainer is a goal that is an executable entity. The result of its execution can be either success or failure. Goals can be defined using other goals; and in this case, they are referred to as subgoals. Most of the predefined goals are defined using this method. During execution, Constrainer creates the goal tree.

Goals are the building blocks of search strategies that are implemented in such a way that the exact sequence of search execution is unknown in advance. This type of programming is referred to as non-deterministic programming. In non-deterministic programming, goals define only the search strategy. Furthermore, there are many solution search strategies that can be implemented using goals without managing constraints and handling constraints propagation. Constrainer provides a set of predefined goals that can be combined to implement your search strategy, or you can write your own goals according to your needs.

Goal construction is based on the following two goals:

- `GoalAnd`
- `GoalOr`

The following topics are described in this section:

Using GoalFail

The `GoalFail` goal performs only one task, that is, failure. This goal is frequently used in goal programming to terminate a search inside the current subgoal.

Using GoalAnd

The `GoalAnd` goal defines a goal composed of up to six subgoals. The subgoals are executed in the order specified in the goal constructor; that is, from left to right. Constrainer provides the following constructors for the `GoalAnd` goal:

- `GoalAnd(Goal g1, Goal g2);`
- `GoalAnd(Goal g1, Goal g2, Goal g3);`
- `GoalAnd(Goal g1, Goal g2, Goal g3, Goal g4);`
- `GoalAnd(Goal g1, Goal g2, Goal g3, Goal g4, Goal g5);`
- `GoalAnd(Goal g1, Goal g2, Goal g3, Goal g4, Goal g5, Goal g6);`

The following is an example of `GoalAnd` using three subgoals:

```
Constrainer C = new Constrainer ("PrintThree");
C.execute(
      new GoalAnd(
            new GoalPrintObject(C, new String("Print 1")),
            new GoalPrintObject(C, new String("Print 2")),
            new GoalPrintObject(C, new String("Print 3"))
      )
);
```

The preceding example produces the following output:

```
Print 1 Print 2 Print 3
```

If a subgoal fails, subsequent subgoals are not executed. The following example produces `Print 1`:

```
Constrainer C = new Constrainer ("PrintThree");
C.execute(
      new GoalAnd(
            new GoalPrintObject(C, new String("Print 1")),
            new GoalFail(C),
            new GoalPrintObject(C, new String("Print 3"))
      )
);
```

If all subgoals of a `GoalAnd` goal succeed, the `GoalAnd` goal itself also succeeds.

The limitation of having a maximum of six subgoals can be easily overcome using goal nesting, as presented in the following example:

```
new GoalAnd(
      new GoalAnd(Goal1, Goal2, Goal3, Goal4, Goal5, Goal6),
      new GoalAnd(Goal7, Goal8)
);
```

Using GoalOr

Initially Constrainer does not know the problem solution location in the search space, so it must make assumptions or guesses. If the guess turns out to be wrong, Constrainer must undo all consequences of the assumption that are the results of constraint propagation.

The `GoalOr` goal introduces the following most important concepts of constraint propagation programming:

Constraint propagation programming concepts	
Concept	**Description**
Choice point	Point at which Constrainer is allowed to make a guess and to which it returns if the guess failed.
Backtracking	Process of returning to the choice point.

The mechanisms of choice point and backtracking are provided with the `GoalOr` goal.

When implementing the `Goal` interface, the `GoalOr` constructor gets the following two parameters:

```
GoalOr(Goal g1, Goal g2);
```

The algorithm of `GoalOr` execution consist of the following steps:

1. Save the state of Constrainer so it can be restored later if needed.
2. Execute the first goal.
3. If the first goal fails, the state of Constrainer is restored and the second goal is executed.

A typical use of the `GoalOr` goal consists of the following steps:

- The first goal makes an assumption.
- The second goal imposes constraints that negate the assumption if results are inconsistent.

That is, the user makes an assumption, checks the consistency of system using the assumption, and, if the system proves to be inconsistent, negates the assumption.

Using GoalInstantiate

The `GoalInstantiate` goal selects a value from the domain of the integer constrained variable specified in the goal constructor and binds the value to the variable. Later, Constrainer automatically

propagates constraints imposed on this constrained variable. An instance of the goal can be created using the following two constructors:

```
GoalInstantiate(IntVar intvar);
GoalInstantiate(IntVar intvar, IntValueSelector selector);
```

In the second constructor, a selector is specified that defines the value selection strategy. For information on selectors, see Using Value Selectors.

The following is the `GoalInstantiate` execution algorithm:

1. If the variable domain contains only one element, the goal succeeds.
2. Otherwise, select a value from the variable domain with the value selector.
3. Bind the value to the constrained variable and perform the conditional constraint propagation.
4. If the guess produces inconsistency, perform the following tasks:
    1. Undo the variable binding to the value and the conditional constraint propagation.
    2. Remove the value from the variable domain and perform conditional constraint propagation.
    3. If the variable domain is empty, the goal fails.
    4. Go to step 1.
5. Otherwise, the goal succeeds.

The following is an implementation of the `GoalInstantiate` goal with the aid of goal programming:

```
public class MyGoalInstantiate extends GoalImpl {

private IntVar _intvar;

public MyGoalInstantiate(IntVar intvar)  {

super (intvar.constrainer());

_intvar = intvar;

}

public Goal execute() throws Failure {

if (_intvar.bound())

return null;

return new GoalOr (

new GoalSetValue(_intvar, _intvar.min()),

new GoalAnd (new GoalSetMin(_intvar, _intvar.min() + 1), this)

);

}
```

```
}
```

In the previous implementation, the following goals are used:

- goal `GoalSetValue` binds to the variable the specified value
- goal `GoalSetMin` changes the variable domain minimum to the specified value

The search strategy is defined in the `execute()` method, and the new goal is defined. For value `selection`, the selection of the minimal value from the variable domain is used.

In the preceding example, the following algorithm is executed:

1. The goal `GoalOr` executes the first subgoal `GoalSetValue`, binding the variable to its minimal value. That is, it is assumed that the variable equals its minimal value.
2. The goal `GoalOr` performs the propagation of constraints upon the assumption.

   If no controversy is detected, the solution is found and the goal `GoalOr` finishes successfully.
3. Otherwise, the goal `GoalOr` performs the following tasks:
   - restores the state saved before execution of the first subgoal
   - executes the second subgoal `GoalAnd`, which in its course excludes the wrong value, that is, the minimal value, from the variable domain using the subgoal `GoalSetMin`
   - recursively executes the instance of the goal `MyGoalInstantiate`
4. Steps 1-3 are recursively performed until a solution is found or until the variable domain is empty.

Users can easily define their own search strategies using goal programming. However, Constrainer provides a better way of defining the selection strategy for the goal `GoalInstantiate` using value selectors. For information on value selectors, see Using Value Selectors.

Using Value Selectors

This section describes a more elegant way to define a selection strategy by using value selectors. The `GoalInstantiate` constructor can assume a value selector as the second parameter as described in Using GoalInstantiate. The actual parameter must be an instance of a class implementing the `IntValueSelector` interface. A class implementing the `IntValueSelector` interface provides the following method:

```
public int select(IntVar var);
```

This method implements the selection strategy. It assumes a constrained integer variable `var` as the parameter and returns a value from the parameter domain according to its selection algorithm. For example, the algorithm can select numbers that are multiples of 11 or even numbers from the domain, and, later, the other numbers in decreasing order. Constrainer provides the following set of predefined value selectors:

Predefined value selectors	
Selector	**Description**
`IntValueSelectorMax`	Selects the maximum value from the variable domain.
`IntValueSelectorMin`	Selects the minimal value from the variable domain.

`IntValueSelectorMinMax`	Alternates the selection of the minimal and the maximal value from the variable domain.

By default, `GoalInstantiate` uses the `IntValueSelectorMin` value selector. Generally using value selectors is very simple. However, in large-scale problems it can be necessary to write your own selectors to improve the solution process performance.

Using GoalDichotomize

`GoalDichotomize` makes use of the assumption that the solution is in the upper half of the variable domain. The solution search algorithm resembles the following:

1. If the variable domain contains only one element, the goal succeeds.
2. Otherwise, the goal calculates the medium value of the variable domain as `(minimum + maximum)/2`.
3. The goal assumes that a solution is in the upper half of the variable domain using `[(minimum + maximum)/2+1, maximum]` and performs the conditional constraint propagation.
4. The goal applies the algorithm to the resulting domain.
5. If no solution is found, the goal performs the following steps:
   1. It undoes the assumption of step 3 and the conditional constraint propagation.
   2. It assumes that a solution is in the lower half of the variable domain using `[minimum, (minimum + maximum)/2]` and performs the conditional constraint propagation.
   3. It applies the algorithm to the resulting domain.
   4. If no solution is found, the goal fails.
6. Otherwise, the goal succeeds.

The following is an example of implementing `GoalDichotomize` using `GoalOr` and `GoalAnd`:

```
public class MyGoalDichotomize extends GoalImpl {

private IntVar _intvar;

public MyGoalDichotomize(IntVar intvar)  {

super (intvar.constrainer());

_intvar = intvar;

}

public Goal execute() throws Failure {

int mid = (_intvar.min() + _intvar.min()) / 2;

if (_intvar.bound())

return null;
```

```
return new GoalOr (
new GoalAnd (new GoalSetMax(_intvar, mid), this),
new GoalAnd (new GoalSetMin(_intvar, mid + 1), this)
);
}
}
```

Using GoalGenerate

The example describes how to use Constrainer to solve one-dimensional problems that represent a very narrow set of problems. Usually multi-dimensional problems must be solved with several or a multitude of unknown variables. In Constrainer, `GoalGenerate` solves multi-dimensional problems for a set of constrained integer variables.

The following topics are described in this section:

Execution Algorithm Based on GoalGenerate

`GoalGenerate` deals with variables collected together into an instance of `IntExpArray` class and operates according to the following recursive algorithm:

1. The goal chooses one of the unbound variables.

   If all the variables are bound, the solution is found and the goal succeeds.

2. The goal makes an assumption about a solution location in the chosen variable domain.

   If the variable domain is empty, the goal performs the following steps:

   1. If it is the uppermost level, the goal fails**.**
   2. Otherwise, the goal undoes all the assumptions made for the variable.
   3. The goal goes to step 4.1 of the previous level of recursion.

3. The goal propagates the effects of that assumption.
4. If a conflict arises, the goal performs the following steps:
   1. It undoes the assumption made in step 2 and the conditional constraint propagation.
   2. It removes the faulty values from the variable domain.
   3. It goes to step 2.
5. The goal recursively goes to step 1.

Execution Algorithm Based on GoalInstantiate

If `GoalGenerate` uses an assumption of `GoalInstantiate` that a variable equals a value, the `GoalGenerate` algorithm resembles the following:

1. The goal chooses one of the unbound variables.

   If all the variables are bound, the solution is found and the goal succeeds.

2. The goal selects a value from the chosen variable domain and binds it to the variable.

   If the variable domain is empty, the goal performs the following steps:

   1. If it is the uppermost level, the goal fails.
   2. It restores the chosen variable domain to the domain as it was before step 2.
   3. It goes to step 4.1 of the previous level of recursion.

3. It propagates the effects of the binding.
4. If a controversy arises, the goal performs the following steps:
   1. It undoes the assumption of step 2 and the conditional constraint propagation.
   2. It removes the previously bound value from the variable domain.
   3. It goes to step 2.
5. The goal recursively goes to step 1.

Although it may seem that there are several ordinary possibilities, this is not true. After each assignment, Constrainer propagates the problem constraints so that the search field is narrower than when using several possibilities. An assignment that led to a conflict creates the search field again. This happens due to constraint propagation after the removal of the faulty value. This is the case not only for `GoalGenerate`, but for all the search possibilities inside Constrainer. This feature represents the very substance of constraint programming. An evident consequence of the statement is that the tighter the problem constraints are, the faster the solution search process works.

Using Constructors to Create a GoalGenerate Instance

An instance of `GoalGenerate` can be created using the following constructors:

- `GoalGenerate(IntExpArray intvars);`
- `GoalGenerate(IntExpArray intvars, boolean dichotomize);`
- `GoalGenerate(IntExpArray intvars, IntVarSelector var_selector, IntValueSelector value_selector);`
- `GoalGenerate(IntExpArray intvars, IntVarSelector var_selector, boolean dichotomize);`

Execution Algorithm Based on GoalDichotomize

The following algorithm is based on `GoalDichotomize`:

1. The goal chooses one of the unbound variables.

   If all the variables are bound, the solution is found and the goal succeeds.

2. The goal assumes that a solution is in the upper half of the chosen variable domain.

   If the variable domain is empty, the goal performs the following steps:

   1. If it is the uppermost level, the goal fails.
   2. Otherwise, it undoes all the assumptions made for the variable.
   3. It goes to step 4.1 of the previous level of recursion.
3. The goal propagates the effects of that assumption.
4. If a controversy arises, the goal performs the following steps:
   1. It undoes the assumption of step 2 and the conditional constraint propagation.
   2. It removes the faulty half from the variable domain.
   3. It goes to step 2.
5. The goal recursively goes to step 1.

Using Integer Variable Selectors

An integer variable selector can be specified in the `GoalGenerate` constructors. In different tasks, it is reasonable to use different variable selection strategies to improve performance. Constrainer provides the following set of predefined integer variable selectors:

Predefined value selectors	
Selector	**Description**
`IntVarSelectorFirstUnbound`	Chooses the first unbound unknown.
`IntVarSelectorMinSize`	Chooses the unknown with the smallest domain.
`IntVarSelectorMaxSize`	Chooses the unknown with the largest domain.
`IntVarSelectorMinSizeMin`	Chooses the unknown with the smallest domain and minimal value.

You can also compose your own variable selector expressing your search strategy. Custom variable selectors are represented as classes implementing the `IntVarSelector` interface. The interface defines the following method:

```
public int select();
```

This method returns the index of the variable to be selected in an array. The variable is usually specified when creating the selector.

For an example of how to use `GoalGenerate`, see Eight Queens Problem.

Constraint Programming and Object-Oriented Programming

This section describes constraint programming from the viewpoint of object-oriented programming.

The following topics are described in this section:

- Understanding Complexity
- Classifying Constraints

Understanding Complexity

A typical way of composing a constrained-based model for a problem includes articulating the problem statement in terms of constrained variables and constraints. However, additional complexity arises in practical problems. Complexity can be divided into the following groups:

- combinatory complexity of the problem and the problem scale, that is, the problem itself
- complexity related to the data organization, transformation, and interaction among components in an application, that is, representing and solving the problem

Although you cannot reduce the first type of complexity, you can degrade the second type by using object-oriented programming in your Constrainer applications.

Classifying Constraints

From the viewpoint of object-oriented programming, constraints can by classified as follows:

Constraints classification	
Type	**Description**
Inherent or structural	The inherent constraints ensure that the object belongs to its class. These constraints also define the structure of an object, therefore, they are also referred to as structural. For example, consider the class `house`, where the constraints of having a base, walls, and roofing are inherent constraints of an object belonging to that class.
Intrinsic	Intrinsic constraints are pertinent to the instance of the class. They differentiate the instance from the other instances. For example, the constraints of having a stone base, green walls, and red roofing are intrinsic to the instance of the class.

Chapter 4: Solving Optimization Problems

Constrainer provides several mechanisms to find the best solution for a problem.

The following topics are described in this section:

Map Colors Problem 2

This example provides an algorithm of constraint violation minimization to solve the map colors problem originally described in Map Colors Problem.

The following topics are described in this section:

Problem Definition

Assume that you can use only three colors to paint the map. It is evident that some neighbors are colored the same color and there is no way to paint in such a way that all neighbors are colored with different colors. The problem becomes over-constrained.

Problem Solution

You can use the following algorithm to solve the problem:

1. Assign weights, that is, importance, to the boundaries.
2. If two neighboring countries are colored with the same color, calculate the violation as boundary weight; otherwise return 0.
3. Minimize the violation sum. Note that if the sum is 0, all the neighbors are colored with different colors.

To solve the problem, implement this algorithm as follows:

1. Define the country colors as constrained integer variables as follows:

```
IntVar Belgium = C.addIntVar(0, 2, "Belgium");
IntVar Denmark = C.addIntVar(0, 2, "Denmark");
IntVar France = C.addIntVar(0, 2, "France");
IntVar Germany = C.addIntVar(0, 2, "Germany");
IntVar Netherlands = C.addIntVar(0, 2, "Netherland");
IntVar Luxemburg = C.addIntVar(0, 2, "Luxembourg");
```

```
IntExpArray allVariables =
   new IntExpArray(
         C, Belgium, Denmark, France, Germany, Netherlands, Luxembourg
   );
```

The boundaries between France and Belgium, France and Germany, Belgium and the Netherlands, Germany and Denmark, and Germany and the Netherlands must divide the countries with different colors. That is, the hard constraints appear.

2. Post the hard constraints to the `Constrainer` class as follows:

```
C.postConstraint((France.ne(Belgium)).and(France.ne(Germany)));
C.postConstraint(Belgium.ne(Netherlands));
C.postConstraint((Germany.ne(Denmark)).and(Germany.ne(Netherlands)));
```

3. Assign weights 257, 9043, and 568 to the boundaries between France and Luxembourg, Luxembourg and Germany, and Luxembourg and Belgium respectively. That is, the second boundary is the most important, the third one is of less importance, the first one is the least important, and the remaining boundaries are not important at all. According to the weights, compose the cost function as follows:

```
IntExp weightedSum =
   (
         (France.ne(Luxembourg)).mul(257)
   ).add(
         (Luxembourg.ne(Germany)).mul(9043)
   ).add(
         (Luxembourg.ne(Belgium)).mul(568)
   ).neg();
```

4. To find the optimal solution, use `GoalFastMinimize` as follows:

```
C.execute(new GoalFastMinimize(new GoalGenerate(allVariables),weightedSum));
```

The program output is as follows:

```
Belgium: blue
Denmark: white
France: white
Germany: blue
Netherlands: white
Luxembourg: red
```

The calculation takes 551 microseconds. The maximum `WeightedSum` value is 9868.

For information on the complete example source code, see Map Colors Problem 2 Example Code.

Reviewing Concepts

This section describes the following topics:

Minimizing Cost Function

Usually, the optimization problem in Constrainer is solved as follows:

1. Declare the problem unknowns as constrained variables.
2. Impose problem constraints.
3. Create the problem cost function as an integer constrained expression.
4. Search for the cost function minimum.

The following two goals are used to solve optimization problems:

Goals used for solving optimization problems	
Goal	**Description**
`GoalMinimize`	Uses the search goal provided by the caller and expects that this goal instantiates the cost every time a solution is found. `GoalMinimize` uses the branch and bound method to search for a solution, which provides the minimal cost. It recursively splits the domain of the cost variable into two parts. The goal parameter "precision" specifies when the search should stop. When the difference between the cost maximum and minimum is less than the precision, the currently found solution is considered to be the optimal one. Otherwise, the goal replaces the cost domain by one of its halves, restores all decision variables, and calls itself again.
`GoalFastMinimize`	Uses the search goal provided by the caller and expects that this goal instantiates the cost every time a solution is found. To search for a solution that provides the minimal cost, `GoalFastMinimize` finds a solution, posts the constraint that the cost variable should be less then the found cost, and continues the search. In contrast with `GoalMinimize`, when the solution is found, `GoalFastMinimize` does not restore all decision variables, but continues the search with a more strict constraint to the cost variable.

The point in common of the preceding two goals is that the last found solution is the optimal one. By default, the goals calculate the optimal solution twice. However, you can change the default behavior. To prevent the duplicate calculation at the end of the search, set the `goal_saves_solution` mode when creating a goal.

Maximizing Cost Function

To maximize the cost function, use its negation in the `GoalMinimize` and `GoalFastMinimize` goals. For information on the `GoalMinimize` and `GoalFastMinimize` goals, see Minimizing Cost Function. For example, to maximize the cost function `profit`, you can use the following code:

```
C.execute(new GoalFastMinimize(profit.neg()), vars);
```

Using Constraint Violation Minimization

Most real-world decision support applications are over-constrained. This means that not all active rules and constraints that control the business environment can be satisfied at the same time, because some of them would be violated. In this situation, the optimal solution frequently minimizes the total constraints violations with respect to their relative importance. With the practical assumption that most real-world systems are over-constrained, Constrainer classifies the constraints in the following way:

Constraints classification	
Term	**Description**
Soft	Constraint can be violated.
Hard	Constraint cannot be violated.

This classification provides appropriate violations measurement and associates constraints, along with an efficient algorithm that finds such values for all unknown objects to minimize the total constraints violation.

For more information on minimizing constraint violations, see Family Riddle 2.

Chapter 5: Solving Scheduling Problems

This section explains how to solve scheduling problems.

Scheduler is a Java package built on top of part of Constrainer to deal with scheduling and resource allocation problems. It supplements Constrainer by adding scheduling functionality, and introduces such terms as jobs and resources. Scheduler is presented as Java package `com.exigen.ie.scheduler`.

The following topics are described in this chapter:

- Problem Solution Examples
- Reviewing Concepts

Problem Solution Examples

The following examples are described in this section:

Problem solution examples	
Example	**Description**
House Building	Provides an example of optimal segregation of house building duties. It explains how to solve basic scheduling problems.
Oven Orders	Provides an example of scheduling oven orders. It explains how to use resources in scheduling problems.
Oven Orders 2	Provides an example of scheduling orders for two ovens. It explains how to use alternative resources in scheduling problems.

House Building

This example provides the optimal segregation of the house building duties algorithm.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

A typical house building problem occurs when a user must find the optimal order of different tasks to perform to minimize the building time required. The following table describes the required tasks, their durations, and dependencies:

House building tasks		
Task	**Duration**	**Preceding jobs**
Masonry	7	None.
Carpentry	3	Masonry.

House building tasks		
Task	**Duration**	**Preceding jobs**
Plumbing	8	Masonry.
Ceiling	3	Masonry.
Roofing	1	Carpentry.
Painting	2	Ceiling.
Windows	1	Roofing.
Facade	2	Roofing, Plumbing.
Garden	1	Roofing, Plumbing.
Moving in	1	Windows, Facade, Garden, and Painting.

Problem Solution

To find a solution, proceed as follows:

1. Import the `Constrainer` and `Scheduler` packages as follows:

```
import com.exigen.ie.constrainer.*;
import com.exigen.ie.scheduler.*;
```

2. Define a class as follows:

```
public final class House1
{
```

3. Create the `main()` method:

```
public static void main(String args[]) throws Exception
{
```

4. Create a constrainer and schedule as follows:

```
Constrainer C = new Constrainer("House 1 Example");
Schedule S = new Schedule(C, 0, 30);
S.name("House 1");
```

5. Define the jobs as follows:

```
Job masonry   = S.addJob(7, "masonry");
Job carpentry = S.addJob(3, "carpentry");
Job roofing   = S.addJob(1, "roofing");
Job plumbing  = S.addJob(8, "plumbing");
Job ceiling   = S.addJob(3, "ceiling");
Job windows   = S.addJob(1, "windows");
Job facade    = S.addJob(2, "facade");
Job garden    = S.addJob(1, "garden");
Job painting  = S.addJob(2, "painting");
```

```
Job moving_in = S.addJob(1, "moving_in");
```

6. Post the `startsAfterEnd` constraints as follows:

```
carpentry.startsAfterEnd(masonry).asConstraint().post();
roofing.startsAfterEnd(carpentry).asConstraint().post();
plumbing.startsAfterEnd(masonry).asConstraint().post();
ceiling.startsAfterEnd(masonry).asConstraint().post();
windows.startsAfterEnd(roofing).asConstraint().post();
facade.startsAfterEnd(roofing).asConstraint().post();
facade.startsAfterEnd(plumbing).asConstraint().post();
garden.startsAfterEnd(roofing).asConstraint().post();
garden.startsAfterEnd(plumbing).asConstraint().post();
painting.startsAfterEnd(ceiling).asConstraint().post();
moving_in.startsAfterEnd(windows).asConstraint().post();
moving_in.startsAfterEnd(facade).asConstraint().post();
moving_in.startsAfterEnd(garden).asConstraint().post();
moving_in.startsAfterEnd(painting).asConstraint().post();
```

7. Gather the job variables as follows:

```
IntExpArray vars = new IntExpArray(C, 10);
for(int i=0; i < S.jobs().size(); ++i)
   {
         Job job = (Job)S.jobs().elementAt(i);
         vars.set(job.getStartVariable(),i);
   }
```

8. Create the solution goal as follows:

```
Goal solution = new GoalGenerate(vars);
```

9. To optimize, create a move-in date and time as follows:

```
IntExp objective = moving_in.getStartVariable();
```

10. Create `GoalMinimize` and print out the results as follows:

```
if (!C.execute(new GoalMinimize(solution,objective)))
   System.out.println("Can not minimize cost "+objective);
else
{
   System.out.println("Optimal solution = " +objective + ":");
   for(int i=0; i < S.jobs().size(); ++i)
   {
         Job job = (Job)S.jobs().elementAt(i);
```

```
            System.out.println(job);
        }
    }
```

The program provides the following output:

```
Optimal solution = moving_in.start[17]:
masonry[0 --7--> 7)
carpentry[7 --3--> 10)
roofing[10 --1--> 11)
plumbing[7 --8--> 15)
ceiling[7 --3--> 10)
windows[11 --1--> 12)
facade[15 --2--> 17)
garden[15 --1--> 16)
painting[10 --2--> 12)
moving_in[17 --1--> 18)
```

For information on the complete example source code, see House Building Example Code.

Oven Orders

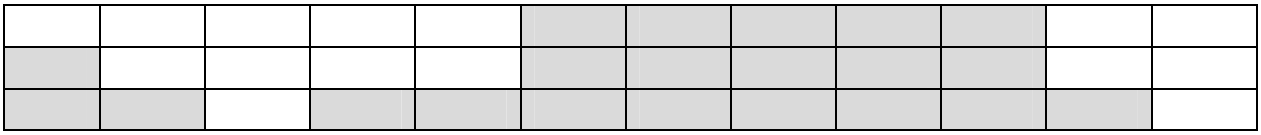
This example provides an example of scheduling oven orders.

The following topics are described in this section:

- Problem Definition
- Problem Solution

Problem Definition

Assume an oven in which you can fire batches of bricks. There are five orders to fire X batches during Y days. Schedule all the orders to be done in no more than 11 days, taking into consideration the following oven availability:



0 1 2 3 4 5 6 7 8 9 10 11

The order size is the following:

Order size		
Job	**Batches**	**Days**
JobA	2	1
JobB	1	4
JobC	1	4
JobD	1	2
JobE	2	4

Problem Solution

To solve the problem, proceed as follows:

1. Import the `Constrainer` and `Scheduler` packages.
2. Create a class.
3. Create the `main()` method.
4. Define the jobs as follows:

```
Job jA = S.addJob(1, "JobA");
Job jB = S.addJob(4, "JobB");
Job jC = S.addJob(4, "JobC");
Job jD = S.addJob(2, "JobD");
Job jE = S.addJob(4, "JobE");
```

5. Define a resource oven as follows:

```
Resource oven = S.addResource(3,"oven");
```

6. Change oven capacity according to the problem definition as follows:

```
oven.capacity(0,2);
oven.capacity(1,1);
oven.capacity(2,0);
oven.capacity(3,1);
oven.capacity(4,1);
oven.capacity(10,1);
```

7. Post the requirement constraints as follows:

```
jA.requires(oven,2).post();
jB.requires(oven,1).post();
jC.requires(oven,1).post();
jD.requires(oven,1).post();
jE.requires(oven,2).post();
```

8. Create an array of instantiated variables as follows:

```
IntExpArray vars = new IntExpArray(C, 5);
for(int i=0; i < S.jobs().size(); ++i)
{
   Job job = (Job)S.jobs().elementAt(i);
   vars.set(job.getStartVariable(),i);
}
```

9. Solve the problem as follows:

```
Goal solution = new GoalGenerate(vars);
```

```
if (!C.execute(solution))
   System.out.println("Can not solve "+solution);
else
{
   System.out.println("1st solution:");
   for(int i=0; i < S.jobs().size(); ++i)
   {
         Job job = (Job)S.jobs().elementAt(i);
         System.out.println(job);
   }
}
```

The program provides the following output:

```
JobA[5 --1--> 6) => oven
JobB[3 --4--> 7) => oven
JobC[7 --4--> 11) => oven
JobD[0 --2--> 2) => oven
JobE[6 --4--> 10) => oven
```

For information on the complete example source code, see Oven Orders Example Code.

Oven Orders 2

This example explains how to schedule orders for two ovens.

The following topics are described in this section:

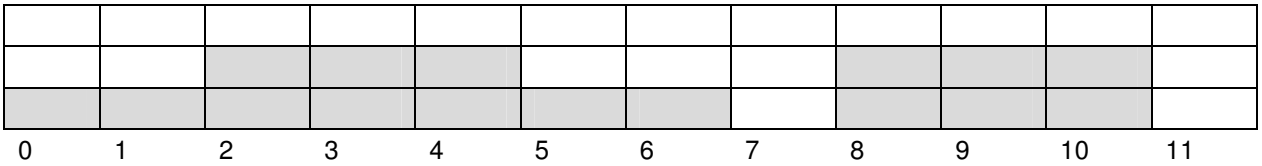
- Problem Definition
- Problem Solution

Problem Definition

This problem is similar to the problem described in Oven Orders, with the following difference:

- There are two ovens.
- There are 10 orders.
- Each order should use one of the two ovens.

The second oven has the following availability:



The remaining five orders are presented in the following table:

Order size		
Job	**Batches**	**Days**
JobF	1	1
JobG	1	3
JobH	2	3
JobI	1	2
JobJ	3	1

Problem Solution

To solve the problem, proceed as follows:

1. Import the `Constrainer` and `Scheduler` packages.
2. Create a class.
3. Create the `main()` method.
4. Define the jobs as follows:

```
Job jA = S.addJob(1, "JobA");
Job jB = S.addJob(4, "JobB");
Job jC = S.addJob(4, "JobC");
Job jD = S.addJob(2, "JobD");
Job jE = S.addJob(4, "JobE");
Job jF = S.addJob(1, "JobF");
Job jG = S.addJob(3, "JobG");
Job jH = S.addJob(3, "JobH");
Job jI = S.addJob(2, "JobI");
Job jJ = S.addJob(1, "JobJ");
```

5. Define the resources and capacities as follows:

```
Resource oven1 = S.addResource(3,"oven1");
Resource oven2 = S.addResource(3,"oven2");
oven1.capacity(0,1,2);
oven1.capacity(1,2,1);
oven1.capacity(2,3,0);
oven1.capacity(3,5,1);
oven1.capacity(5,10,3);
oven1.capacity(10,11,1);
oven2.capacity(0,2,1);oven2.capacity(2,5,2);
oven2.capacity(5,7,1);
```

```
oven2.capacity(7,8,0);

oven2.capacity(8,11,2);
```

6. Post the requirement constraints as follows:

```
jA.requires(oven1,2).or(jA.requires(oven2,2)).post();
jB.requires(oven1,1).or(jB.requires(oven2,1)).post();
jC.requires(oven1,1).or(jC.requires(oven2,1)).post();
jD.requires(oven1,1).or(jD.requires(oven2,1)).post();
jE.requires(oven1,2).or(jE.requires(oven2,2)).post();
jF.requires(oven1,1).or(jF.requires(oven2,1)).post();
jG.requires(oven1,1).or(jG.requires(oven2,1)).post();
jH.requires(oven1,2).or(jH.requires(oven2,2)).post();
jI.requires(oven1,1).or(jI.requires(oven2,1)).post();
jJ.requires(oven1,3).or(jJ.requires(oven2,3)).post();
```

7. Create an array of variables to be instantiated as follows:

```
IntExpArray vars = new IntExpArray(C, 10);

for(int i=0; i < S.jobs().size(); ++i)

{

   Job job = (Job)S.jobs().elementAt(i);

   System.out.println("* "+job);

   vars.set(job.getStartVariable(),i);

}
```

8. Solve the problem as follows:

```
Goal solution = new GoalGenerate(vars);
if (!C.execute(solution))
   System.out.println("Can not solve "+solution);
else
{
   System.out.println("1st solution:");
   for(int i=0; i < S.jobs().size(); ++i)
   {
         Job job = (Job)S.jobs().elementAt(i);
         System.out.println(job);
   }
}
```

The program provides the following output:

```
JobA[0 --1--> 1) => oven1          JobF[1 --1--> 2) => oven1
JobB[0 --4--> 4) => oven2          JobG[6 --3--> 9) => oven1
JobC[2 --4--> 6) => oven2          JobH[8 --3--> 11) => oven2
JobD[3 --2--> 5) => oven1          JobI[9 --2--> 11) => oven1
JobE[6 --4--> 10) => oven1         JobJ[5 --1--> 6) => oven1
```

For information on the complete example source code, see Oven Orders 2 Example Code.

Reviewing Concepts

Scheduler is designed to solve scheduling and resource allocation problems.

The following topics are described in this section:

- Scheduler Parts
- Using Scheduler with Constrainer
- Creating a Schedule
- Finding a Solution
- Understanding Jobs
- Understanding Recourses
- Reviewing Major Concepts

Scheduler Parts

Scheduler consists of the following parts:

Scheduler parts	
Part	**Description**
class `Schedule`	Main container representing Scheduler itself.
interface `Job`	Interface for jobs, which represent schedule activities.
class `JobInterval`	Concrete job interval implementation.
class `Resource`	Resource implementation.
class `ConstraintRequires`	Resource requirement constraint implementation.
class `ConstraintRequiresOr`	Implementation of when a job can require different resources. This is a logical OR operation.
class `IntExpEmployed`	Implementation of expression reflecting resource employment.

Using Scheduler with Constrainer

Scheduler is based on Constrainer. Scheduler contains objects that represent aspects of scheduling and resource allocation problems. The user can easily extend Scheduler by adding more specialized classes representing the objects of a particular problem.

Scheduler defines the following general directions of the user-defined objects:

- implementing particular jobs and resources
- implementing specialized constraints
- introducing new heuristics in search algorithms

Creating a Schedule

To create a schedule, create the `Constrainer` and `Schedule` objects as shown in the following example:

```
Constrainer C = new Constrainer("Example");

Schedule S = new Schedule(C, 0, 30);

S.name("Schedule Example");
```

The following table describes Scheduler objects used in the example:

Scheduler objects in example	
Objects	**Description**
`Constrainer C = new Constrainer("Example");`	Creates the `Constrainer` object.
`Schedule S = new Schedule(C, 0, 30);`	Creates the `Scheduler` object. This constrainer has the following parameters: • constrainer object to associate with • two integers representing a schedule horizon The horizon is a time interval for which the user wants to calculate the schedule.
`S.name("Schedule Example");`	Names the schedule. This parameter is optional.

Finding a Solution

The following example is the simplest method of finding the solution:

```
IntExpArray vars = new IntExpArray(C, vars_size);

// assigning variables to the array

// gathering them from defined scheduling objects

…
```

```
// the simplest case: just generate desired variables

if (!C.execute(new GoalGenerate(vars)))

{

// success

}

else

{

// failure

}
```

The preceding method of finding the solution is the same for other Constrainer problems. It is flexible and gives the user a method to handle scheduling and resource allocation problems in the same way as another problems.

Understanding Jobs

This section describes main working job concepts.

The following topics are described in this section:

- Defining a Job
- Constraining Jobs
- Accessing Job Variables

Defining a Job

Job is a class that represents any activity.

To define a job, use the `addJob(duration, name)` method of the `Schedule` object.

The following is an example of defining a job:

```
Job masonry = S.addJob(7, "masonry");
```

Constraining Jobs

To introduce job dependency constraints, use the following expressions:

Expressions for constraining jobs	
Expression	**Description**
`A.startsAfterEnd(B)`	Job A starts after job B is finished.
`A.startsAfterStart(B)`	Job A starts after job B is started.

Expressions for constraining jobs	
`A.endsAfterStart(B)`	Job A finishes after job B is started.
`A.endsAfterEnd(B)`	Job A finishes after job B is finished.

For example, to define that job A precedes job B, use the following expression:

```
IntBoolExp b_after_a = jobB.startsAfterEnd(jobA);
```

You can use this as an expression in further calculations or activate it immediately as a constraint as follows:

```
jobB.startsAfterEnd(jobA).asConstraint().post();
```

Accessing Job Variables

To perform a search, you must gather variables to pass them to a desired goal, for example `GoalGenerate`. Scheduler provides a simple mechanism for querying schedule jobs and their internal variables. `Schedule` class provides the `jobs()` method that returns `FastVector` of `Job` objects. `Job` provides the `getStartVariable()` method used for this purpose.

The following is a code example that shows how to create an array containing the start variables of all schedule jobs:

```
// creating an array of S.jobs().size() variables

IntExpArray vars = new IntExpArray(C, S.jobs().size(););


// setting array elements to jobs variables

for(int i=0; i < S.jobs().size(); ++i)

{

Job job = (Job)S.jobs().elementAt(i);

vars.set(job.getStartVariable(),i);
```

Understanding Resources

A resource is an entity that is consumed by jobs. This chapter explains the main concepts of working with resources. The following topics are described in this section:

- Defining a Recourse
- Recourse Constraints

Defining a Resource

To define a resource, use the `addResource` method as shown in the following example:

```
Resource res1 = S.addResource(5,"resource1");
```

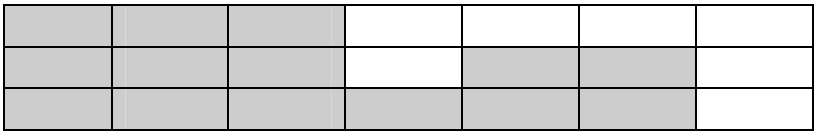
The `addResource` method has the following possibilities:

addResource method expressions	
Expression	**Description**
`addResource(capacity, name)`	Adds a resource with a specified capacity and the time available during all scheduling processes.
`addResource(capacity, avail_start, avail_end, name)`	Adds a resource with a specified capacity and limited availability, that is, it is available only during a defined period `[avail_start, avail_end)`.

When a resource requires a time-varied capacity, you can lower capacities using the `setCapacityMax()` method as it is described in the following table:

`setCapacityMax` method expressions	
Expression	**Description**
`setCapacityMax(time_start, time_end, capacity)`	Changes a resource capacity during the specified period `[time_start, time_end]` to a new value.
`setCapacityMax(time, capacity)`	Is equal to the call of the previous `setCapacityMax(time, time+1, capacity)`, that is, it changes capacity at the particular moment.

The following example provides a definition of a resource with varying capacity according to the timetable:



0 1 2 3 4 5 6

The timetable is represented by the following code:

```
Resource res = S.addResource(3, 0, 6, "time varying");

res.setCapacityMax(3, 1);

res.setCapacityMax(4, 6, 2);
```

You can query the remaining capacity of a resource any time it is available using the calling method `capacity(time)`.

Resource Constraints

A resource is an entity that is consumed by some jobs. This fact represents the `ConstraintRequires` constraint. This constraint is created using the `requires(resource, capacity)` method of the `Job` interface. The simplest method assumes that `capacity=1`.

The following example illustrates how to use `ConstraintRequires` constraint:

```
Resource res = S.addResource(3, ”my resource”);

Job job = S.addJob(7, ”my job”);

ConstraintRequires creq = job.requires(res,2);

creq.post();
```

Sometimes jobs can require similar resources. In this case, `ConstraintRequiresOr` is used as the helper constraint that keeps and observes all resources the job can require and chooses the suitable one. Creation of this constraint is initiated by the predefined method `or(Constraint)` of `ConstraintRequires`. It is important to call exactly the method `ConstraintRequires` because the base class `Constraint` has another `or()` method that is designed for regular constraints, not for resource constraints.

Appendix A: Source Code for Examples

This appendix contains code source of examples listed in this guide.

The following examples are described in this section:

System of Equations Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

public class Formula
{
  public static void main(String[] args) {
    try {
      Constrainer c = new Constrainer("Formula");

      // define variables
      IntVar X = c.addIntVar(0, 100, "X");
      IntVar Y = c.addIntVar(0, 100, "Y");

      // add "formula" constraints
//      c.addConstraint("X**2 + Y**2 < 20"); // in symbolic forms
      c.addConstraint(X.mul(X).add(Y.mul(Y)).lt(20)); // in direct form

//      c.addConstraint("X + Y == 5"); // in symbolic forms
      c.addConstraint(X.add(Y).eq(5)); // in direct form

      // post all the constraints
      c.postConstraints();

      // Search for all the solutions
```

```
      IntExpArray vars = new IntExpArray(c, X, Y);
      Goal goal = new GoalGenerateAll(vars);
      c.execute(goal);
    } catch(Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
  }
}
```

Map Colors Problem Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

public class MapColoring{
  static final String[] colors = {"red","green","blue","yellow"};

  static public void main (String[] argv){
    try{

      Constrainer C = new Constrainer("Map-coloring");
      IntVar Belgium = C.addIntVar(0,3,"Belgium");
      IntVar Denmark = C.addIntVar(0,3,"Denmark");
      IntVar France = C.addIntVar(0,3,"France");
      IntVar Germany = C.addIntVar(0,3,"Germany");
      IntVar Netherlands = C.addIntVar(0,3,"Netherland");
      IntVar Luxemburg = C.addIntVar(0,3,"Luxemburg");

      IntExpArray allVariables = new
IntExpArray(C,Belgium,Denmark,France,Germany,Netherlands,Luxemburg);
      C.postConstraint(
((France.ne(Belgium)).and(France.ne(Luxemburg))).and(France.ne(Germany)) );
      C.postConstraint( (Luxemburg.ne(Germany)).and(Luxemburg.ne(Belgium)) );
      C.postConstraint(  Belgium.ne(Netherlands) );
      C.postConstraint( (Germany.ne(Netherlands)).and(Germany.ne(Denmark)) );

      C.execute( new GoalAnd(new GoalGenerateAll(allVariables),
                             new GoalPrint(allVariables)));
    }
   catch (Exception ex) {
    ex.printStackTrace();
   }
  }
}Magic Square Problem Example Code

package sampleproblem;

import com.exigen.ie.constrainer.*;

public class MagicSquare {
  public static void main(String[] args) {
    try {
```

```
    String arg = (args.length==0)?"5":args[0];
    int n = Integer.parseInt(arg);
    int sum = n*(n*n+1)/2;

    int i, j;
    // create Constrainer instance
    Constrainer C = new Constrainer ("Magic Square");
    // create all magic square elements
    IntExpArray vars = new IntExpArray(C, n*n, 1, n*n, "vars");
    // all elements must be unique
    C.postConstraint(C.allDiff(vars));

    // create arrays of rows and columns
    IntExpArray[] rows = new IntExpArray[n];
    IntExpArray[] columns = new IntExpArray[n];

    // create arrays for diagonals
    IntExpArray diagonal1 = new IntExpArray(C, n);
    IntExpArray diagonal2 = new IntExpArray(C, n);

    for (i = 0; i < n; i++) {
      // create arrays for the i-th row and column
      rows[i] = new IntExpArray(C, n);
      columns[i] = new IntExpArray(C, n);
      // populate the arrays
      for (j = 0; j < n; j++) {
        rows[i].set(vars.get(i * n + j), j);
        columns[i].set(vars.get(j * n + i), j);
      }
      diagonal1.set(vars.get(i*n + i), i);
      diagonal2.set(vars.get(i*n + (n - i - 1)), i);
      // the i-th row and colunm are populated,
      // let's impose constraint on their sums
      C.postConstraint(rows [i].sum().eq(sum));
      C.postConstraint(columns [i].sum().eq(sum));
    }

    // the diagonals are populated, let's impose constraint on their sums
    C.postConstraint(diagonal1.sum().eq(sum));
    C.postConstraint(diagonal1.sum().eq(sum));

    // search a solution
    if (C.execute(new GoalGenerate (vars))) {
      // print the solution found
      for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
          System.out.print(vars.get(i * n + j).value()+" ");
        }
        System.out.println();
      }
    } else {
      System.out.println("No solutions");
    }
  } catch (Failure e) {
    System.out.println(e);
    e.printStackTrace();
  }
```

```
  }
}
```

Magic Sequence Problem Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

 public class MagicSequence {
  static Constrainer C = new Constrainer("MagicSequence");

  public static void main(String[] args) {
    try{
      String arg = (args.length==0)?"10":args[0];
      int N = Integer.parseInt(arg);

      // the problem variable declaration
      IntExpArray sequence = new IntExpArray(C,N+1,0,N, "MS");

      // imposing the problem constraints
      C.distribute(sequence,sequence);
      int[] coeffs = new int[N+1];
      for (int i=0;i<coeffs.length;i++){
        coeffs[i] = i;
      }
      // imposing the redundant constraint
      C.postConstraint(C.scalarProduct(sequence,coeffs).eq(N+1));

      C.printInformation();
      Goal gen = new GoalGenerate(sequence);
      boolean flag = C.execute(gen);
      if (!flag){
        throw new Failure();
      }
      for (int i = 0; i < N; i++) {
        System.out.print(sequence.get(i).value() + " ");
      }
      System.out.println();
    }
    catch (Failure f){
      System.out.println("There is no such a sequence");
    }
    catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

Family Riddle Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

public class Famlily {
  public static void main (String [] args) {
    // constrainer creation
    Constrainer c = new Constrainer ("Family Riddle");

    // unknowns declaration
    // all the children are under age ten that is [0 .. 9]
    IntExpArray ReneChildren = new IntExpArray (c, 6, 0, 9, "Rene");
    IntExpArray LeoChildren = new IntExpArray (c, 6, 0, 9, "Leo");
    IntExpArray LeoSons = new IntExpArray (c, 3);
    IntExpArray ReneSons = new IntExpArray (c, 3);
    IntExpArray LeoDaughters = new IntExpArray (c, 3);
    IntExpArray ReneDaughters = new IntExpArray (c, 3);

    // girls goes first then boys follows
    for (int i = 0; i < 3; i++) {
      LeoDaughters.set (LeoChildren.get (i) , i);
      LeoSons.set (LeoChildren.get (i + 3) , i);

      ReneDaughters.set (ReneChildren.get (i) , i);
      ReneSons.set (ReneChildren.get (i + 3) , i);
    }

    try {
      // constaint imposition

      // the youngest child in Leo's family is a girl
      for (int i = 1; i < 6; i++) {
        c.postConstraint (LeoChildren.elementAt (0).lt (
          LeoChildren.elementAt (i)));
      }

      // Rene girl is just arrived
      c.postConstraint (ReneDaughters.elementAt (0).eq (0));

      // neither family includes any twins, nor any children closer
      // in age than a year.
      c.postConstraint (c.allDiff (LeoChildren));
      c.postConstraint (c.allDiff (ReneChildren));

      // boys' ages sum equal to girls' ages sum
      c.postConstraint (LeoSons.sum ().eq (LeoDaughters.sum ()));
      c.postConstraint (ReneSons.sum ().eq (ReneDaughters.sum ()));

      // boys' age squares sum equal to girls' age squares sum
      c.postConstraint (LeoSons.mul (LeoSons).eq (
        LeoDaughters.mul (LeoDaughters)));
      c.postConstraint (ReneSons.mul(ReneSons).eq (
        ReneDaughters.mul (ReneDaughters)));
```

```
      // the sum of all children ages is equal to 60
      c.postConstraint (ReneChildren.sum ().add (LeoChildren.sum ()).eq (60));

      c.execute (
        new GoalAnd (
          new GoalGenerate (ReneChildren),
          new GoalGenerate (LeoChildren)
        )
      );

      System.out.println("\n" + ReneChildren +"\n"+ LeoChildren);

    } catch (Exception e) {
      System.out.println (e);
      e.printStackTrace ();
    }
  }
}
```

Warehouse Maintenance Problem Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

 public class Warehouse4 {

  public static void main(String[] args) throws Failure {
    // warehouse maintenance cost
    int fixed = 300;
    // number of stores
    int nbStores = 21;
    // number of warehouses
    int nbWarehouses = 20;

    Constrainer c = new Constrainer ("Warehouse");

    // warehouse to store supply cost matrix
    IntArray    supplyCost [] = new IntArray [nbStores];

    // the matrix initialization
    supplyCost [0] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [1] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [2] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [3] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [4] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [5] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [6] = new IntArray (c, new int []
```

```
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [7] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [8] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [9] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [10] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [11] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [12] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [13] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [14] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [15] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [16] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [17] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});
    supplyCost [18] = new IntArray (c, new int []
{20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4});
    supplyCost [19] = new IntArray (c, new int []
{29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44,20,52,37,48});
    supplyCost [20] = new IntArray (c, new int []
{110,24,83,4,29,62,43,44,20,52,37,48,110,24,83,4,29,62,43,44});

    // the array of conditions showing whether a warehouse is opened
    IntExpArray openWarehouses = new IntExpArray(c, nbWarehouses);
    for (int i = 0; i < nbWarehouses; i++) {
      openWarehouses.set(c.addIntBoolVar(), i);
    }
    // the array of store assignment to warehouses
    IntExpArray storeAssign =
      new IntExpArray(c, nbStores, 0, nbWarehouses - 1, "SA");

    // if a store is assigned to a warehouse then the warehouse is opened
    for (int i = 0; i < nbStores; i++) {
      for (int j = 0; j < nbWarehouses; j++) {
        c.postConstraint(
          storeAssign.get(i).eq(j).implies(
            (IntBoolExp)openWarehouses.get(j)
          )
        );
      }
    }

    // warehouse->store transition cost array
    IntExpArray transCost = new IntExpArray (c, nbStores);
    for (int i = 0; i < nbStores; i++) {
      transCost.set(supplyCost [i].elementAt (storeAssign.elementAt (i)), i);
    }

    // the first summand of the cost function is the transition cost sum
```

```
    // the second summand of the cost function is the maintenance cost of
warehouses
    IntExp cost = transCost.sum ().add (openWarehouses.sum ().mul (fixed));

    c.printInformation ();
    c.execute (
      new GoalFastMinimize (
        new GoalAnd (
          new GoalGenerate (openWarehouses),
          new GoalGenerate (storeAssign)
        ),
        cost
      )
    );
    System.out.println("Optimal cost    :" + cost.toString ());
    System.out.print("Open warehouses :" );
    for (int i = 0; i < nbWarehouses; i++) {
      System.out.print("  " + openWarehouses.get(i).value());
    }
    System.out.println();
    System.out.print("Store assignment:");
    for (int i = 0; i < nbStores; i++) {
      System.out.print(" " + storeAssign.get(i).value());
    }
  }
}
```

Eight Queens Problem Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

public class Queens {
  public static void main(String[] args) {
    try {
      String arg = (args.length==0)?"24":args[0];

      // board size and simultaneously the number of queens
      int board_size = Integer.parseInt(arg);
      Constrainer C = new Constrainer("Queens");

      // array of queens rows
      IntExpArray x = new IntExpArray(C, board_size);
      // auxillary arrays
      IntExpArray xplus = new IntExpArray(C, board_size);
      IntExpArray xminus = new IntExpArray(C, board_size);

      for(int i=0; i < board_size; i++) {
        IntVar variable =
          C.addIntVar(0, board_size-1, "q"+i, IntVar.DOMAIN_BIT_SMALL);
        x.set(variable, i);
        xplus.set(variable.add(i), i);
        xminus.set(variable.sub(i), i);
```

```
      }

      // all rows are different
      C.postConstraint (C.allDiff(x));
      // x[i] + i != x[j] + j
      C.postConstraint (C.allDiff(xplus));
      // x[i] - i != x[j] - j
      C.postConstraint (C.allDiff(xminus));

      C.printInformation();
      // searching for solution
      // C.execute (new GoalGenerate(x)); // not optimized search
      // optimized search with variable selector and dichotomize
      C.execute (new GoalGenerate(x,new IntVarSelectorMinSizeMin(x), true));

      // print the found solution
      System.out.println(x);
    } catch(Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
  }
}
```

Family Riddle 2 Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;

public class OOFamily {
    private String      m_father;
    private IntExpArray m_sons;
    private IntExpArray m_daughters;
    private IntExpArray m_children;
    private Constrainer m_constrainer;
    private IntExp      m_totalAge;

    public OOFamily (Constrainer constrainer, String father) {
      m_father = father;
      m_constrainer = constrainer;

      m_daughters = new IntExpArray (constrainer, 3, 0, 9, m_father + "'girls");
      m_sons = new IntExpArray (constrainer, 3, 0, 9, m_father + "'s boys");
      m_children = new IntExpArray (constrainer, 6);
      for (int i = 0; i < 3; i++) {
        m_children.set (m_sons.elementAt (i), i);
        m_children.set (m_daughters.elementAt (i), i + 3);
      }
      m_totalAge = m_children.sum ();
      imposeConstraints ();
    }

    public IntExp getTotalAge () { return m_totalAge; }
```

```
  public IntExpArray getChildren () { return m_children; }
  public IntExp getDaugther (int idx) { return m_daughters.elementAt (idx); }
  public IntExp getSon (int idx) { return m_sons.elementAt (idx); }

  public void print () {
    System.out.println (m_father + "'s family:");
    System.out.println (m_sons);
    System.out.println (m_daughters);
  }

  private void imposeConstraints () {
    try {
      // no twins
      m_constrainer.postConstraint (m_constrainer.allDiff (m_children));
      // boys' ages sum eq girls' ages sum
      m_constrainer.postConstraint (m_sons.sum ().eq (m_daughters.sum ()));
      // boys' ages squares sum eq girls' ages squares sum
      m_constrainer.postConstraint (m_sons.mul (m_sons).eq (
        m_daughters.mul (m_daughters)));
      // constraints to remove symmetric solutions
      for (int i = 0; i < 2; i++) {
        m_constrainer.postConstraint (
          m_sons.elementAt (i).lt (m_sons.elementAt (i + 1)));
        m_constrainer.postConstraint (
          m_daughters.elementAt (i).lt (m_daughters.elementAt (i + 1)));
      }
    } catch (Exception e) {
      System.out.println (e);
    }
  }

public static void main(String[] args) {
  try {
    Constrainer C = new Constrainer ("OOFamily");

    OOFamily LeoFamily = new OOFamily (C, "Leo");
    OOFamily ReneFamily = new OOFamily (C, "Rene");

    // Leo's last child is a girl
    C.postConstraint (LeoFamily.getDaugther (0).lt (LeoFamily.getSon (0)));

    // the Rene's youngest girl is under a yeah
    C.postConstraint (ReneFamily.getDaugther (0).eq (0));

    // all the children age sum is 60
    C.postConstraint (
      LeoFamily.getTotalAge ().add(ReneFamily.getTotalAge ()).eq (60));

    C.execute (
        new GoalGenerate(LeoFamily.getChildren().merge(ReneFamily.getChildren()))
    );
    LeoFamily.print();
    ReneFamily.print();
  } catch (Exception e) {
    System.out.println (e);
    e.printStackTrace ();
  }
```

```
  }
}
```

Map Colors Problem 2 Example Code

```
package sampleproblem;

import com.exigen.ie.constrainer.*;
public class ColorMin
{
  static final int MAX_COLORS_NUM = 2;
  static final String[] colors = {"blue", "white", "red", "green"};
  public static void main(String[] args)
  {
      try{

        Constrainer C = new Constrainer("Map-coloring");
        // define the country colors as constrained integer variables
        IntVar Belgium = C.addIntVar(0,3,"Belgium");
        IntVar Denmark = C.addIntVar(0,3,"Denmark");
        IntVar France = C.addIntVar(0,3,"France");
        IntVar Germany = C.addIntVar(0,3,"Germany");
        IntVar Netherlands = C.addIntVar(0,3,"Netherland");
        IntVar Luxemburg = C.addIntVar(0,3,"Luxemburg");

        IntExpArray allVariables = new IntExpArray
          (C,Belgium,Denmark,France,Germany,Netherlands,Luxemburg);

        // post the hard constraints to Constrainer
        C.postConstraint((France.ne(Belgium)).and(France.ne(Germany)));
        C.postConstraint(Belgium.ne(Netherlands));
        C.postConstraint((Germany.ne(Denmark)).and(Germany.ne(Netherlands)));

        // compose the cost function
        IntExp weightedSum = ((France.ne(Luxemburg)).mul(257)).add(
                              (Luxemburg.ne(Germany)).mul(9043)).add(
                              (Luxemburg.ne(Belgium)).mul(568)).mul(-1);

        for (int i=0;i<allVariables.size();i++){
            C.postConstraint((allVariables.get(i)).le(MAX_COLORS_NUM-1));
        }
        // find the optimal solution
        Goal all = new GoalFastMinimize(new
GoalGenerate(allVariables),weightedSum);

        C.printInformation();
        C.execute(all);

        for (int i=0;i<allVariables.size();i++){
          System.out.println(
            allVariables.get(i).name()+" : "+colors[allVariables.get(i).value()]);
        }
        System.out.println ();
        System.out.println (
```

```
          "Maximum WeightedSum value is : " + (-1)*weightedSum.value ());
        System.out.println ();
      }
      catch(Failure ex){
        System.out.println(" There is no solution...");
        ex.toString();
      }
      catch(Exception ex){
        ex.printStackTrace();
      }
  }
}
```

House Building Example Code

```
package scheduler;

import com.exigen.ie.constrainer.*;
import com.exigen.ie.scheduler.*;

public final class House
{
    public static void main(String args[]) throws Exception
    {
        Constrainer C = new Constrainer("House 1 Example");
        Schedule S = new Schedule(C, 0, 30);
        S.setName("House 1");

        // defining jobs
        Job masonry = S.addJob(7, "masonry");
        Job carpentry = S.addJob(3, "carpentry");
        Job roofing = S.addJob(1, "roofing");
        Job plumbing = S.addJob(8, "plumbing");
        Job ceiling = S.addJob(3, "ceiling");
        Job windows = S.addJob(1, "windows");
        Job facade = S.addJob(2, "facade");
        Job garden = S.addJob(1, "garden");
        Job painting = S.addJob(2, "painting");
        Job moving_in = S.addJob(1, "moving_in");

        //    Posting "startsAfterEnd" constraints
        carpentry.startsAfterEnd(masonry).asConstraint().post();
        roofing.startsAfterEnd(carpentry).asConstraint().post();
        plumbing.startsAfterEnd(masonry).asConstraint().post();
        ceiling.startsAfterEnd(masonry).asConstraint().post();
        windows.startsAfterEnd(roofing).asConstraint().post();
        facade.startsAfterEnd(roofing).asConstraint().post();
        facade.startsAfterEnd(plumbing).asConstraint().post();
        garden.startsAfterEnd(roofing).asConstraint().post();
        garden.startsAfterEnd(plumbing).asConstraint().post();
        painting.startsAfterEnd(ceiling).asConstraint().post();
        moving_in.startsAfterEnd(windows).asConstraint().post();
        moving_in.startsAfterEnd(facade).asConstraint().post();
        moving_in.startsAfterEnd(garden).asConstraint().post();
```

```
        moving_in.startsAfterEnd(painting).asConstraint().post();

        C.printInformation();
        Goal solution = new GoalSetTimes(S.jobs());

        IntExp objective = moving_in.getStartVariable();
        if (!C.execute(new GoalMinimize(solution,objective)))
            System.out.println("Can not minimize cost "+objective);
        else
        {
            System.out.println("Optimal solution with objective="+objective+":");
            for(int i=0; i < S.jobs().size(); ++i)
            {
                Job job = (Job)S.jobs().elementAt(i);
                System.out.println(job);
            }
        }
    }
}
```

Oven Orders Example Code

```
package scheduler;

import com.exigen.ie.constrainer.*;
import com.exigen.ie.scheduler.*;

public final class Oven
{

    static class MySelector implements JobVariableSelector {
      public IntVarSelector getSelector(IntExpArray vars)
      {
        return new IntVarSelectorMinSizeMin(vars);
      }
    }

    public static void main(String args[]) throws Exception
    {
        Constrainer C = new Constrainer("Oven Scheduling Example");
        Schedule S = new Schedule(C,0,11);
        S.setName("Oven");

        long executionStart = System.currentTimeMillis();

        Job jA = S.addJob(1, "JobA");
        Job jB = S.addJob(4, "JobB");
        Job jC = S.addJob(4, "JobC");
        Job jD = S.addJob(2, "JobD");
        Job jE = S.addJob(4, "JobE");

        Resource oven = S.addResourceDiscrete(3,"oven");

        oven.setCapacityMax(0,2);
```

```
        oven.setCapacityMax(1,1);
        oven.setCapacityMax(2,0);
        oven.setCapacityMax(3,1);
        oven.setCapacityMax(4,1);
        oven.setCapacityMax(10,1);

        jA.requires(oven,2).post();
        jB.requires(oven,1).post();
        jC.requires(oven,1).post();
        jD.requires(oven,1).post();
        jE.requires(oven,2).post();

        Goal solution = new GoalSetTimes(S.jobs(), new MySelector());

        long solveStart = System.currentTimeMillis();

        C.printInformation();

        if (!C.execute(solution))
            System.out.println("Can not solve "+solution);
        else
        {
            System.out.println("1st solution:");
            for(int i=0; i < S.jobs().size(); ++i)
            {
                Job job = (Job)S.jobs().elementAt(i);
                System.out.println(job);
            }
        }
        System.out.println(oven);

        long solveTime = System.currentTimeMillis() - solveStart;
        long executionTime = System.currentTimeMillis() - executionStart;

        System.out.println("Execution time: "+executionTime+" msec");
        System.out.println("Solving time: "+solveTime+" msec");

    }
}
```

Oven Orders 2 Example Code

```
package scheduler;

import com.exigen.ie.constrainer.*;
import com.exigen.ie.scheduler.*;

public final class TwoOvens
{
    final static int OVENS = 2;

    static class MySelector implements JobVariableSelector {
      public IntVarSelector getSelector(IntExpArray vars)
      {
```

```
        return new IntVarSelectorFirstUnbound(vars);
      }
    }

    public static void main(String args[]) throws Exception
    {
        Constrainer C = new Constrainer("TwoOvens Scheduling Example");
        Schedule S = new Schedule(C,0,11);
        S.setName("TwoOvens");

        long executionStart = System.currentTimeMillis();

        Job jA = S.addJob(1, "JobA");
        Job jB = S.addJob(4, "JobB");
        Job jC = S.addJob(4, "JobC");
        Job jD = S.addJob(2, "JobD");
        Job jE = S.addJob(4, "JobE");
        Job jF = S.addJob(1, "JobF");
        Job jG = S.addJob(3, "JobG");
        Job jH = S.addJob(3, "JobH");
        Job jI = S.addJob(2, "JobI");
        Job jJ = S.addJob(1, "JobJ");

        // creating resources for two ovens
        AlternativeResourceSet res = new AlternativeResourceSet();

        Resource oven1 = S.addResourceDiscrete(3,"oven1");
        Resource oven2 = S.addResourceDiscrete(3,"oven2");

        res.add(oven1);
        res.add(oven2);

        oven1.setCapacityMax(0,1,2);
        oven1.setCapacityMax(1,2,1);
        oven1.setCapacityMax(2,3,0);
        oven1.setCapacityMax(3,5,1);
        oven1.setCapacityMax(5,10,3);
        oven1.setCapacityMax(10,11,1);

        oven2.setCapacityMax(0,2,1);
        oven2.setCapacityMax(2,5,2);
        oven2.setCapacityMax(5,7,1);
        oven2.setCapacityMax(7,8,0);
        oven2.setCapacityMax(8,11,2);

        C.postConstraint(jA.requires(res,2));
        C.postConstraint(jB.requires(res,1));
        C.postConstraint(jC.requires(res,1));
        C.postConstraint(jD.requires(res,1));
        C.postConstraint(jE.requires(res,2));
        C.postConstraint(jF.requires(res,1));
        C.postConstraint(jG.requires(res,1));
        C.postConstraint(jH.requires(res,2));
        C.postConstraint(jI.requires(res,1));
        C.postConstraint(jJ.requires(res,3));

        Goal solution = new GoalSetTimes(S.jobs(), new MySelector());
```

```
        long solveStart = System.currentTimeMillis();

        // C.traceExecution();
        // C.traceFailures();
        // C.trace(vars);

        C.printInformation();

        if (!C.execute(solution))
            System.out.println("Can not solve "+solution);
        else
        {
            System.out.println("1st solution:");
            for(int i=0; i < S.jobs().size(); ++i)
            {
                Job job = (Job)S.jobs().elementAt(i);
                System.out.println(job);
            }
        }
        System.out.println(oven1);
        System.out.println(oven2);

        long solveTime = System.currentTimeMillis() - solveStart;
        long executionTime = System.currentTimeMillis() - executionStart;

        System.out.println("Execution time: "+executionTime+" msec");
        System.out.println("Solving time: "+solveTime+" msec");

    }
}
```

Appendix B: Formal Description of Constraint Programming

This appendix formally describes the concept of constraint programming.

The following topics are described in this section:

Constraint Satisfaction Problems

Constraint programming deals with how to solve constraint satisfaction problems (CSP). Formally speaking, CSP is defined by a set of variables $X_1, X_2 \ldots X_n$, and a set of constraints $C_1, C_2 \ldots C_m$. Each variable X_i has a non-empty domain D_i of possible values. Each constraint C_i involves a subset of variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all variables, $\{X_i=v_i; X_j=v_j\}$. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints, that is a complete consistent assignment.

The simplest kind of CSP involves variables that are discrete and have finite domains. If the maximum domain size of any variable in a CSP is d, then the number of possible complete assignments is $O(d^n)$. That is, it is exponential in the number of variables. Discrete variables can also have infinite domains, for example, the set of integers or the set of strings. Constraint satisfaction problems with continuous domains are very common in the real world and are widely studied in the field of the operations research.

Constraint Types

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the unary constraint that restricts the value of a single variable. A binary constraint relates two variables. Higher-order constraints involve three or more variables. Special solution algorithms, which are not discussed in this guide, exist for linear constraints on integer variables, that is, constraints in which each variable appears only in linear form. It is proven that no algorithm exists for solving general non-linear constraints on integer variables.

Constraint Solution Search Types

The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. Plain backtracking is an uninformed algorithm, so it is not expected to be very effective for large problems.

The following domain-independent heuristics help to improve the search performance regardless of the problem:

Heuristic types	
Type	**Description**
Minimum remaining values heuristic	Selects the variable with the fewest valid values for the next assignment. It picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. This heuristic is also known as the most constrained variable or fail-first heuristic.
Degree heuristic	Attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

A way to make use of constraints during search is called forward checking. When a variable *X* is assigned, the forward checking process looks at each unassigned variable *Y* that is connected to *X* by a constraint and deletes from *Y*'s domain any value that is inconsistent with the value chosen for *X*. Problem-specific heuristics help to improve the problem solution search performance. The heuristics are invented by constraint programmers, and rely on deep knowledge of the problem.

Glossary

Glossary	
arithmetic constraint	Constraint that restricts numerical variables in arithmetical expressions using comparison operators, for example a + b = 10. Arithmetic constraints can be combined using logical operations.
backtracking	Process of abandoning one line of search, usually after failure, returning to a given state in the system, and restoring variables to their previous values in that state so that another alternative can be explored and used when searching for a solution.
constraint	Condition that must be satisfied by the solution of a problem.
domain	Constrained variable associated with its set of potential values that the variable can assume. The set is referred to as the variable domain. The process of constraint propagation monotonically reduces domains to the values satisfying the specified constraints.
failure	Situation when the constrained variable domain is reduced to the empty domain during the constraint propagation process so the problem constraints could not be satisfied.
goal	Building block of search algorithms in Constrainer. A goal execution result is success or failure. Goals can be defined using other goals. In that case, the subsequent goals are referred to as subgoals. Constrainer puts goals into its stack, and the order of goals in the stack corresponds to their execution order.
inherent constraint	Constraint that deals with parameters that relate objects to their classes.
propagation	Reduction of a variable domain can arise from the reduction of some other variables due to problem constraints. The sequential process of domain reduction caused by constraints is referred to as constraint propagation.
redundant constraint	Logical consequence of other constraints participating in the problem statement. These constraints do not affect the problem solutions, but can lead to Constrainer performance improvement.
specific constraint	Constraint that deals with parameters that identify an object inside its class.
symbolic constraint	Shortcut for complex relations between variables. Such relations can be expressed with an exponential number of arithmetic constraints or cannot be expressed in such a way at all. An example of symbolic constraints is the constraint assuring that all elements of an array are different from one another.

Index