Choco solver
http://choco.emn.fr/

Documentation

July 15, 2010

Contents

Preface

Choco is a java library for constraint satisfaction problems (CSP) and constraint programming (CP). It is built on a event-based propagation mechanism with backtrackable structures. Choco is an open-source software, distributed under a **BSD licence** and hosted by sourceforge.net. For any informations visit http://choco.emn.fr.

This document is organized as follows:

- Documentation is the user-guide of Choco. After a short introduction to constraint programming and to the Choco solver, it presents the basics of modeling and solving with Choco, and some advanced usages (customizing propagation and search).

- Elements of Choco gives a detailed description of the variables, operators, constraints currently available in Choco.

- Extras presents future works, only available on the beta version or extension of the current jar, such as the visualization module of Choco. The section dedicated to Sudoku aims at explaining the basic principles of Constraint Programming (propagation and search) on this famous game.

Part I

Documentation

The documentation of Choco is organized as follows:

- The concise introduction provides some informations about constraint programming concepts and a "Hello world"-like first Choco program.

- The model section gives informations on how to create a model and introduces variables and constraints.

- The solver section gives informations on how to create a solver, to read a model, to define a search strategy, and finally to solve a problem.

- The advanced use section explains how to define your own limit search space, search strategy, constraint, operator, variable, backtrackable structure and write logging statements.

- The applications section shows the use of Choco defined global constraints on scheduling or placement problems.

Chapter 1

Introduction to constraint programming and Choco

1.1 About constraint programming

> Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.
>
> **E. C. Freuder, Constraints, 1997.**

Fast increasing computing power in the 1960s led to a wealth of works around problem solving, at the root of Operational Research, Numerical Analysis, Symbolic Computing, Scientific Computing, and a large part of Artificial Intelligence and programming languages. Constraint Programming is a discipline that gathers, interbreeds, and unifies ideas shared by all these domains to tackle decision support problems.

Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (scheduling, routing), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), etc.

Current research in this area deals with various fundamental issues, with implementation aspects and with new applications of constraint programming.

1.1.1 Constraints

A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take, it represents some partial information about the variables of interest. For instance, the circle is inside the square relates two objects without precisely specifying their positions, i.e., their coordinates. Now, one may move the square or the circle and he or she is still able to maintain the relation between these two objects. Also, one may want to add another object, say a triangle, and to introduce another constraint, say the square is to the left of the triangle. From the user (human) point of view, everything remains absolutely transparent.

Constraints naturally meet several interesting properties:

- constraints may specify partial information, i.e. constraint need not uniquely specify the values of its variables,

- constraints are non-directional, typically a constraint on (say) two variables X, Y can be used to infer a constraint on X given a constraint on Y and vice versa,

- constraints are declarative, i.e. they specify what relationship must hold without specifying a computational procedure to enforce that relationship,

- constraints are additive, i.e. the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect,

- constraints are rarely independent, typically constraints in the constraint store share variables.

Constraints arise naturally in most areas of human endeavor. The three angles of a triangle sum to 180 degrees, the sum of the currents floating into a node must equal zero, the position of the scroller in the window scrollbar must reflect the visible part of the underlying document, these are some examples of constraints which appear in the real world. Thus, constraints are a natural medium for people to express problems in many fields.

1.1.2 Constraint Programming

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (conditions, properties) which must be satisfied by the solution.

Work in this area can be tracked back to research in Artificial Intelligence and Computer Graphics in the sixties and seventies. Only in the last decade, however, has there emerged a growing realization that these ideas provide the basis for a powerful approach to programming, modeling and problem solving and that different efforts to exploit these ideas can be unified under a common conceptual and practical framework, constraint programming.

> If you know **sudoku**, then you know **constraint programming**. See why here.

1.2 Modeling with Constraint programming

The formulation and the resolution of combinatorial problems are the two main goals of the constraint programming domain. This is an essential way to solve many interesting industrial problems such as scheduling, planning or design of timetables. The main interest of constraint programming is to propose to the user to model a problem without being interested in the way the problem is solved.

1.2.1 The Constraint Satisfaction Problem

Constraint programming allows to solve combinatorial problems modeled by a Constraint Satisfaction Problem (CSP). Formally, a CSP is defined by a triplet (X, D, C):

- **Variables**: $X = \{X_1, X_2, \ldots, X_n\}$ is the set of variables of the problem.

- **Domains**: D is a function which associates to each variable X_i its domain $D(X_i)$, i.e. the set of possible values that can be assigned to X_i. The domain of a variable is usually a finite set of integers: $D(X_i) \subset \mathbb{Z}$ (*integer variable*). But a domain can also be continuous ($D(X_i) \subseteq \mathbb{R}$ for a *real variable*) or made of discrete set values ($D(X_i) \subseteq \mathcal{P}(\mathbb{Z})$ for a *set variable*).

- **Constraints**: $C = \{C_1, C_2, \ldots, C_m\}$ is the set of constraints. A constraint C_j is a relation defined on a subset $X^j = \{X_1^j, X_2^j, \ldots, X_{n^j}^j\} \subseteq X$ of variables which restricts the possible tuples of values $(v_1, \ldots, v_{n^j})$ for these variables:

$$(v_1, \ldots, v_{n^j}) \in C_j \cap (D(X_1^j) \times D(X_2^j) \times \cdots \times D(X_{n^j}^j)).$$

Such a relation can be defined explicitly (ex: $(X_1, X_2) \in \{(0, 1), (1, 0)\}$) or implicitly (ex: $X_1 + X_2 \leq 1$).

Solving a CSP is to find a tuple $v = (v_1, \ldots, v_n) \in D(X)$ on the set of variables which satisfies all the constraints:

$$(v_1, \ldots, v_{n^j}) \in C_j, \quad \forall j \in \{1, \ldots, m\}.$$

For optimization problems, one need to define an **objective function** $f : D(X) \to \mathbb{R}$. An optimal solution is then a solution tuple of the CSP that minimizes (or maximizes) function f.

1.2.2 Examples of CSP models

This part provides three examples using different types of variables in different problems. These examples are used throughout this tutorial to illustrate their modeling with Choco.

Example 1: the n-queens problem.

Let us consider a chess board with n rows and n columns. A queen can move as far as she pleases, horizontally, vertically, or diagonally. The standard n-queens problem asks how to place n queens on an n-ary chess board so that none of them can hit any other in one move.

The n-queens problem can be modeled by a CSP in the following way:

- **Variables**: $X = \{X_i \mid i \in [1, n]\}$.

- **Domain**: for all variable $X_i \in X$, $D(X_i) = \{1, 2, \ldots, n\}$.

- **Constraints**: the set of constraints is defined by the union of the three following constraints,

  - queens have to be on distinct lines:
    * $C_{lines} = \{X_i \neq X_j \mid i, j \in [1, n], i \neq j\}$.
  - queens have to be on distinct diagonals:
    * $C_{diag1} = \{X_i \neq X_{j+j-i} \mid i, j \in [1, n], i \neq j\}$.
    * $C_{diag2} = \{X_i \neq X_{j+i-j} \mid i, j \in [1, n], i \neq j\}$.

Example 2: the ternary Steiner problem.

A ternary Steiner system of order n is a set of $n*(n-1)/6$ triplets of distinct elements taking their values in $[1, n]$, such that all the pairs included in two distinct triplets are different. See http://mathworld.wolfram.com/SteinerTripleSystem.html for details.

The ternary Steiner problem can be modeled by a CSP in the following way:

- let $t = n*(n-1)/6$.

- **Variables**: $X = \{X_i \mid i \in [1, t]\}$.

- **Domain**: for all $i \in [1, t]$, $D(X_i) = \{1, ..., n\}$.

- **Constraints**:

  - every set variable X_i has a cardinality of 3:
    * for all $i \in [1, t]$, $|X_i| = 3$.
  - the cardinality of the intersection of every two distinct sets must not exceed 1:
    * for all $i, j \in [1, t]$, $i \neq j$, $|X_i \cap X_j| \leq 1$.

Example 3: the CycloHexane problem.

The problem consists in finding the 3D configuration of a cyclohexane molecule. It is described with a system of three non linear equations:

- **Variables**: x, y, z.
- **Domain**: $]-\infty; +\infty[$.
- **Constraints**:

$$y^2 * (1 + z^2) + z * (z - 24 * y) = -13$$
$$x^2 * (1 + y^2) + y * (y - 24 * x) = -13$$
$$z^2 * (1 + x^2) + x * (x - 24 * z) = -13$$

1.3 My first Choco program: the magic square

1.3.1 The magic square problem

In the following, we will address the magic square problem of order 3 to illustrate step-by-step how to model and solve this problem using choco.

Definition:

A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A standard magic square contains the integers from 1 to n^2.

The constant sum in every row, column and diagonal is called the magic constant or magic sum M. The magic constant of a classic magic square depends only on n and has the value: $M(n) = n(n^2 + 1)/2$.

More details on the magic square problem.

1.3.2 A mathematical model

Let x_{ij} be the variable indicating the value of the jth cell of row i. Let C be the set of constraints modeling the magic square as:

$$
\begin{aligned}
& x_{ij} \in [1, n^2], && \forall i, j \in [1, n] \\
& x_{ij} \neq x_{kl}, && \forall i, j, k, l \in [1, n], i \neq k, j \neq l \\
& \sum_{j=1}^{n} x_{ij} = n^2, && \forall i \in [1, n] \\
& \sum_{i=1}^{n} x_{ij} = n^2, && \forall j \in [1, n] \\
& \sum_{i=1}^{n} x_{ii} = n^2 \\
& \sum_{i=n}^{1} x_{i(n-i)} = n^2
\end{aligned}
$$

We have all the required information to model the problem with Choco.

> For the moment, we just talk about *model translation* from a mathematical representation to Choco. Choco can be used as a *black box*, that means we just need to define the problem without knowing the way it will be solved. We can therefore focus on the modeling not on the solving.

1.3.3 To Choco...

First, we need to know some of the basic Choco objects:

- The **model** (object `Model` in Choco) is one of the central elements of a Choco program. Variables and constraints are associated to it.

- The **variables** (objects `IntegerVariable`, `SetVariable`, and `RealVariable` in Choco) are the *unknown* of the problem. Values of variables are taken from a **domain** which is defined by a set of values or quite often simply by a lower bound and an upper bound of the allowed values. The domain is given when creating the variable.

> Do not forget that we manipulate **variables** in the mathematical sense (as opposed to classical computer science). Their effective value will be known only once the problem has been solved.

- The **constraints** define relations to be satisfied between variables and constants. In our first model, we only use the following constraints provided by Choco:

  - `eq(var1, var2)` which ensures that `var1` equals `var2`.
  - `neq(var1, var2)` which ensures that `var1` is not equal to `var2`.
  - `sum(var[])` which returns expression `var[0]+var[1]+...+var[n]`.

1.3.4 The program

After having created your java class file, import the Choco class to use the API:

```java
import choco.Choco;
```

First of all, let's create a Model:

```java
// Constant declaration
int n = 3; // Order of the magic square
int magicSum = n * (n * n + 1) / 2; // Magic sum
// Build the model
CPModel m = new CPModel();
```

We create an instance of `CPModel()` for **C**onstraint **P**rogramming Model. Do not forget to add the following imports:

```java
import choco.cp.model.CPModel;
```

Then we declare the variables of the problem:

```java
// Creation of an array of variables
IntegerVariable[][] var = new IntegerVariable[n][n];
// For each variable, we define its name and the boundaries of its domain.
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        var[i][j] = Choco.makeIntVar("var_" + i + "_" + j, 1, n * n);
        // Associate the variable to the model.
        m.addVariable(var[i][j]);
    }
}
```

Add the import:

```java
import choco.kernel.model.variables.integer.IntegerVariable;
```

We have defined the variable using the `makeIntVar` method which creates an enumerated domain: all the values are stored in the java object (beware, it is usually not necessary to store all the values and it is less efficient than to create a bounded domain).

Now, we are going to state a constraint ensuring that all variables must have a different value:

```
// All cells of the matrix must be different
for (int i = 0; i < n * n; i++) {
    for (int j = i + 1; j < n * n; j++) {
        Constraint c = (Choco.neq(var[i / n][i % n], var[j / n][j % n]));
        m.addConstraint(c);
    }
}
```

Add the import:

```
import choco.kernel.model.constraints.Constraint;
```

Then, we add the constraint ensuring that the magic sum is respected:

```
// All rows must be equal to the magic sum
for (int i = 0; i < n; i++) {
    m.addConstraint(Choco.eq(Choco.sum(var[i]), magicSum));
}
```

Then we define the constraint ensuring that each column is equal to the magic sum. Actually, `var` just denotes the rows of the square. So we have to declare a temporary array of variables that defines the columns.

```
IntegerVariable[][] varCol = new IntegerVariable[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // Copy of var in the column order
        varCol[i][j] = var[j][i];
    }
    // Each column?s sum is equal to the magic sum
    m.addConstraint(Choco.eq(Choco.sum(varCol[i]), magicSum));
}
```

It is sometimes useful to define some temporary variables to keep the model simple or to reorder array of variables. That is why we also define two other temporary arrays for diagonals.

```
IntegerVariable[] varDiag1 = new IntegerVariable[n];
IntegerVariable[] varDiag2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    varDiag1[i] = var[i][i]; // Copy of var in varDiag1
    varDiag2[i] = var[(n - 1) - i][i]; // Copy of var in varDiag2
}
// Every diagonal?s sum has to be equal to the magic sum
m.addConstraint(Choco.eq(Choco.sum(varDiag1), magicSum));
m.addConstraint(Choco.eq(Choco.sum(varDiag2), magicSum));
```

Now, we have defined the model. The next step is to solve it. For that, we build a Solver object:

```
// Build the solver
CPSolver s = new CPSolver();
```

with the imports:

```
import choco.cp.solver.CPSolver;
```

We create an instance of `CPSolver()` for Constraint Programming Solver. Then, the solver reads (translates) the model and solves it:

```
// Read the model
```

```
        s.read(m);
        // Solve the model
        s.solve();
        // Print the solution
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(MessageFormat.format("{0}␣", s.getVar(var[i][j]).getVal()));
            }
            System.out.println();
        }
```

The only variables that need to be printed are the ones in `var` (all the others are only references to these ones).

> We have to use the Solver to get the value of each variable of the model.
> The Model only declares the objects, the Solver finds their value.

We are done, we have created our first Choco program. The complete source code can be found here: ExMagicSquare.zip

1.3.5 In summary

- A Choco Model is defined by a set of Variables with a given domain and a set of Constraints that link Variables: it is necessary to add both Variables and Constraints to the Model.

- temporary Variables are useful to keep the Model readable, or necessary when reordering arrays.

- The value of a Variable can be known only once the Solver has found a solution.

- To keep the code readable, you can avoid the calls to the static methods of the Choco classes, by importing the static classes, i.e. instead of:

```
import choco.Choco;
...
IntegerVariable v = Choco.makeIntVar("v", 1, 10);
...
Constraint c = Choco.eq(v, 5);
```

you can use:

```
import static choco.Choco.*;
...
IntegerVariable v = makeIntVar("v", 1, 10);
...
Constraint c = eq(v, 5);
```

1.4 Complete examples

We provide now the complete Choco model for the three examples previously described.

1.4.1 Example 1: the n-queens problem with Choco

This first model for the n-queens problem only involves binary constraints of differences between integer variables. One can immediately recognize the 4 main elements of any Choco code. First of all, create the model object. Then create the variables by using the Choco API (One variable per queen giving the row (or the column) where the queen will be placed). Finally, add the constraints and solve the problem.

```
        int nbQueen = 8;
        //1- Create the model
        CPModel m = new CPModel();
        //2- Create the variables
        IntegerVariable[] queens = Choco.makeIntVarArray("Q", nbQueen, 1, nbQueen);
        //3- Post constraints
        for (int i = 0; i < nbQueen; i++) {
            for (int j = i + 1; j < nbQueen; j++) {
                int k = j - i;
                m.addConstraint(Choco.neq(queens[i], queens[j]));
                m.addConstraint(Choco.neq(queens[i], Choco.plus(queens[j], k))); // diagonal
                        constraints
                m.addConstraint(Choco.neq(queens[i], Choco.minus(queens[j], k))); // diagonal
                        constraints
            }
        }
        //4- Create the solver
        CPSolver s = new CPSolver();
        s.read(m);
        s.solveAll();
        //5- Print the number of solutions found
        System.out.println("Number of solutions found:"+s.getSolutionCount());
```

1.4.2 Example 2: the ternary Steiner problem with Choco

The ternary Steiner problem is entirely modeled using set variables and set constraints.

```
        //1- Create the problem
        CPModel mod = new CPModel();
        int m = 7;
        int n = m * (m - 1) / 6;

        //2- Create Variables
        SetVariable[] vars = new SetVariable[n]; // A variable for each set
        SetVariable[] intersect = new SetVariable[n * n]; // A variable for each pair of sets
        for (int i = 0; i < n; i++)
            vars[i] = Choco.makeSetVar("set " + i, 1, n);
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                intersect[i * n + j] = Choco.makeSetVar("interSet " + i + " " + j, 1, n);

        //3- Post constraints
        for (int i = 0; i < n; i++)
            mod.addConstraint(Choco.eqCard(vars[i], 3));
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++) {
                // the cardinality of the intersection of each pair is equal to one
                mod.addConstraint(Choco.setInter(vars[i], vars[j], intersect[i * n + j]));
                mod.addConstraint(Choco.leqCard(intersect[i * n + j], 1));
            }

        //4- Search for a solution
        CPSolver s = new CPSolver();
        s.read(mod);
        s.setVarSetSelector(new MinDomSet(s, s.getVar(vars)));
        s.setValSetSelector(new MinEnv());
        s.solve();

        //5- Print the solution found
        for(SetVariable var: vars){
```

```
        System.out.println(s.getVar(var).pretty());
    }
```

1.4.3 Example 3: the CycloHexane problem with Choco

Real variables are illustrated on the problem of finding the 3D configuration of a cyclohexane molecule.

```
//1- Create the problem

CPModel pb = new CPModel();
pb.setPrecision(1e-8);

//2- Create the variable
RealVariable x = Choco.makeRealVar("x", -1.0e8, 1.0e8);
RealVariable y = Choco.makeRealVar("y", -1.0e8, 1.0e8);
RealVariable z = Choco.makeRealVar("z", -1.0e8, 1.0e8);

//3- Create and post the constraints
RealExpressionVariable exp1 = Choco.plus(Choco.mult(Choco.power(y, 2), Choco.plus(1,
    Choco.power(z, 2))),
        Choco.mult(z, Choco.minus(z, Choco.mult(24, y))));

RealExpressionVariable exp2 = Choco.plus(Choco.mult(Choco.power(z, 2), Choco.plus(1,
    Choco.power(x, 2))),
        Choco.mult(x, Choco.minus(x, Choco.mult(24, z))));

RealExpressionVariable exp3 = Choco.plus(Choco.mult(Choco.power(x, 2), Choco.plus(1,
    Choco.power(y, 2))),
        Choco.mult(y, Choco.minus(y, Choco.mult(24, x))));

Constraint eq1 = Choco.eq(exp1, -13);
Constraint eq2 = Choco.eq(exp2, -13);
Constraint eq3 = Choco.eq(exp3, -13);

pb.addConstraint(eq1);
pb.addConstraint(eq2);
pb.addConstraint(eq3);

//4- Search for all solution
CPSolver s = new CPSolver();
s.read(pb);
s.setVarRealSelector(new CyclicRealVarSelector(s));
s.setValRealIterator(new RealIncreasingDomain());
s.solve();
//5- print the solution found
System.out.println("x " + s.getVar(x).getValue());
System.out.println("y " + s.getVar(y).getValue());
System.out.println("z " + s.getVar(z).getValue());
```

Chapter 2

The model

The `Model`, along with the `Solver`, is one of the two key elements of any Choco program. The Choco `Model` allows to describe a problem in an easy and declarative way: it simply records the variables and the constraints defining the problem.

This section describes the large API provided by Choco to create different types of variables and constraints.

Note that a static import is required to use the Choco API:

```
import static choco.Choco.*;
```

First of all, a `Model` object is created as follows:

```
Model model = new CPModel();
```

In that specific case, a Constraint Programming (CP) `Model` object has been created.

2.1   Variables

A Variable is defined by a type (integer, real, or set variable), a name, and the values of its domain. When creating a simple variable, some options can be set to specify its domain representation (ex: enumerated or bounded) within the `Solver`.

> The choice of the domain should be considered. The efficiency of the solver often depends on judicious choice of the domain type.

Variables can be combined as expression variables using operators.

One or more variables can be added to the model using the following methods of the `Model` class:

```
        model.addVariable(var1);
        model.addVariables(var2, var3);
```

> Explictly addition of variables is not mandatory. See `Constraint` for more details.

Specific role of variables *var* can be defined with *options*: non-decision variables or objective variable;

```
        model.addVariable(Options.V_OBJECTIVE, var4);
        model.addVariables(Options.V_NO_DECISION, var5, var6);
```

2.1.1   Simple Variables

See Section Variables for details:

```
    IntegerVariable, SetVariable, RealVariable
```

2.1.2   Constants

A constant is a variable with a fixed domain. `IntegerVariable` declared with a unique value are automatically set as constant. A constant declared twice or more is only stored once in a model.

```
        IntegerConstantVariable c10 = Choco.constant(10);
        RealConstantVariable c0dot0 = Choco.constant(0.0);
        SetConstantVariable c0_12 = Choco.constant(new int[]{0, 12});
        SetConstantVariable cEmpty = Choco.emptySet();
```

2.1.3   Expression variables and operators

Expression variables represent the result of combinations between variables of the same type made by operators. Two types of expression variables exist :

```
    IntegerExpressionVariable and RealExpressionVariable.
```

One can define a buffered expression variable to make a constraint easy to read, for example:

```
        IntegerVariable v1 = Choco.makeIntVar("v1", 1, 3);
        IntegerVariable v2 = Choco.makeIntVar("v2", 1, 3);
        IntegerExpressionVariable v1Andv2 = Choco.plus(v1, v2);
```

To construct expressions of variables, simple operators can be used. Each returns a `ExpressionVariable` object:

```
    abs, cos, distEq, distGt, distLt, distNeq, div, ifThenElse, max, min, minus, mod,
    mult, neg, plus, power, scalar, sin, sum.
```

Note that these operators are not considered as constraints: they do not return a `Constraint` objet but a `Variable` object.

2.1.4   MultipleVariables

These are syntaxic sugar. To make their declaration easier, `tree`, `geost` and scheduling constraints require to use `MultipleVariables`, like `TreeParametersObject`, `GeostObject` or `TaskVariable`.
    link to an example of user-defined MultipleVariables

2.1.5   Decision/non-decision variables

By default, each variable added to a model is a decision variable, *i.e.* is included in the default search strategy. A variable can be stated as a non decision one if its value can be computed by side-effect. To specify non decision variables, one can

- exclude them from its search strategies (see search strategy for more details);

- specify non-decision variables (adding `Options.V_NO_DECISION` to their options) and keep the default search strategy.

```
        IntegerVariable vNoDec = Choco.makeIntVar("vNoDec", 1, 3, Options.V_NO_DECISION);
```

Each of these options can also be set within a single instruction for a group of variables, as follows:

```
        IntegerVariable vNoDec1 = Choco.makeIntVar("vNoDec1", 1, 3);
        IntegerVariable vNoDec2 = Choco.makeIntVar("vNoDec2", 1, 3);
        model.addOptions(Options.V_NO_DECISION, vNoDec1, vNoDec2);
```

- The declaration of a search strategy will erase setting
  `Options.V_NO_DECISION`.

more precise: user-defined/pre-defined, variable and/or value heuristics ?

2.1.6 Objective variable

You can define an objective variable directly within the model, by using option `Options.V_OBJECTIVE`:

```
        IntegerVariable x = Choco.makeIntVar("x", 1, 1000, Options.V_OBJECTIVE);
        IntegerVariable y = Choco.makeIntVar("y", 20, 50);
        model.addConstraint(Choco.eq(x, Choco.mult(y, 20)));
        solver.read(model);
        solver.minimize(true);
```

Only one variable can be defined as an objective. If more than one objective variable is declared, then only the last one will be taken into account.

Note that optimization problems can be declared without defining an objective variable within the model (see the optimization example.)

2.2 Constraints

Choco provides a large number of simple and global constraints and allows the user to easily define its own new constraint. A constraint deals with one or more variables of the model and specify conditions to be held on these variables. A constraint is stated into the model by using the following methods available from the `Model` API:

```
        model.addConstraint(c1);
        model.addConstraints(c2,c3);
```

Adding a constraint automatically adds its variables to the model (explicit declaration of variables addition is not mandatory).

Example:

adding a difference (disequality) constraint between two variables of the model

```
        model.addConstraint(Choco.neq(var1, var2));
```

Available *options* depend on the kind of constraint c to add: they allow, for example, to choose the filtering algorithm to run during propagation. See Section options ans settings for more details, specific APIs exist for declaring options constraints.

This section presents the constraints available in the Choco API sorted by type or by domain. Related sections:

- a detailed description (with options, examples, references) of each constraint is given in Section constraints

- Section applications shows how to apply some specific global constraints

- Section user-defined constraint explains how to create its own constraint.

2.2.1   Binary constraints

Constraints involving two integer variables

- `eq`, `geq`, `gt`, `leq`, `lt`, `neq`

- `abs`, `oppositeSign`, `sameSign`

2.2.2   Ternary constraints

Constraints involving three integer variables

- `distanceEQ`, `distanceNEQ`, `distanceGT`, `distanceLT`

- `intDiv`, `mod`, `times`

2.2.3   Constraints involving real variables

Constraints involving two real variables

- `eq`, `geq`, `leq`

2.2.4   Constraints involving set variables

Set constraints are illustrated on the ternary Steiner problem.

- `eqCard`, `geqCard`, `leqCard`

- `member`, `notMember`

- `isIncluded`, `isNotIncluded`, `setDisjoint`

- `setInter`, `setUnion`

- `max`, `min`

- `pack`

2.2.5 Channeling constraints

The use of a redundant model is a frequent technique to strengthen propagation or to get more freedom to design dedicated search heuristics. The following constraints allow to ensure integrity of different models:

- `inverseChanneling`, `boolChanneling`, `domainConstraint`

More complex channeling can be done using reified constraints (see Section reification) although they are less efficient. For example, to ensure that two variables are equal or not, one can reify the equality into a boolean variables :

```
IntegerVariable reifiedB = Choco.makeIntVar("bvar", 0, 1);
IntegerVariable x = Choco.makeIntVar("a", 0, 10);
IntegerVariable y = Choco.makeIntVar("b", 0, 10);
model.addConstraint(Choco.reifiedConstraint(reifiedB, Choco.eq(x, y)));
```

2.2.6 Constraints in extension and relations

Choco supports the statement of constraints defining arbitrary relations over two or more variables. Such a relation may be defined by three means:

- **feasible table:** the list of allowed tuples of values (that belong to the relation),

- **infeasible table:** the list of forbidden tuples of values (that do not belong to the relation),

- **predicate:** a method to be called in order to check whether a tuple of values belongs or not to the relation.

On the one hand, constraints based on tables may be rather memory consuming in case of large domains, although one relation table may be shared by several constraints. On the other hand, predicate constraints require little memory as they do not cache truth values, but imply some run-time overhead for calling the feasibility test. Table constraints are thus well suited for constraints over small domains; while predicate constraints are well suited for situations with large domains.

Different levels of consistency can be enforce on constraints in extension:

- several arc-consistency (AC) algorithms for binary relations

- two AC algorithms for n-ary relations dedicated either to positive or to negative tables (relation defined by the allowed or forbidden tuples)

- a weaker forward-checking (FC) algorithm for n-ary relations.

The Choco API for creating constraints in extension are as follows:

- `feasPairAC`, `infeasPairAC`, `relationPairAC`

- `feasTupleAC`, `infeasTupleAC`, `relationTupleAC`

- `feasTupleFC`, `infeasTupleFC`, `relationTupleFC`

Relations.

A same relation might be shared among several constraints, in this case it is highly recommended to create it first and then use the relationPairAC, relationTupleAC, or relationTupleFC API on the same relation for each constraint.

For binary relations, the following Choco API is provided: `makeBinRelation(int[] min, int[] max,` `List<int[]>pairs, boolean feas)`

It returns a `BinRelation` giving a list of compatible (`feas=true`) or incompatible (`feas=false`) pairs of values. This relation can be applied to any pair of variables (x_1, x_2) whose domains are included in the `min/max` intervals, i.e. such that:

$$\mathtt{min}[i] \leq x_i.\mathtt{getInf}() \leq x_i.\mathtt{getSup}() \leq \mathtt{max}[i], \quad \forall i.$$

Bounds `min/max` are mandatory in order to allow to compute the opposite of the relation if needed.

For n-ary relations, the corresponding Choco API is: `makeLargeRelation(int[] min, int[] max,` `List<int[]> tuples, boolean feas)`; It returns a `LargeRelation`. If `feas=true`, the returned relation matches also the `IterLargeRelation` interface which provides constant time iteration abilities over tuples (for compatibility with the GAC algorithm used over feasible tuples).

```
LargeRelation r = Choco.makeLargeRelation(min, max, tuples, true);
model.addConstraint(Choco.relationTupleAC(vars, r));
```

Lastly, some specific relations can be defined without storing the tuples, as in the following example (`TuplesTest` extends `LargeRelation`):

```
public static class NotAllEqual extends TuplesTest {
  public boolean checkTuple(int[] tuple) {
    for (int i = 1; i < tuple.length; i++) {
      if (tuple[i - 1] != tuple[i]) return true;
    }
    return false;
  }
}
```

Then, a *NotAllEqual* constraint can be stated within the problem by:

```
CPModel model = new CPModel();
IntegerVariable[] vars = Choco.makeIntVarArray("v", 3, 1, 3);
model.addConstraint(Choco.relationTupleFC(vars, new NotAllEqual()));
```

2.2.7 Reified constraints

Constraints involved in another constraint are usually called reified constraints. Typical examples of reified constraints are constraints combined with logical operators, such as $(x \neq y) \vee (z \leq 9)$.

Choco provides a generic constraint to reify any constraints on integer variables or set variables into a boolean variable expressing its truth value:

- `reifiedConstraint`, `reifiedAnd`, `reifiedLeftImp`, `reifiedNot`, `reifiedOr`, `reifiedRightImp`, `reifiedXnor`, `reifiedXor`

This mechanism can be used for example to model MaxCSP problems where the number of satisfied constraints has to be maximized. It is also intended to give the freedom to the user to build complex reified constraints. However, Choco provides a more simple and direct API to build complex expressions using boolean operators:

- `and`, `or`, `implies`, `ifOnlyIf`, `ifThenElse`, `not`

Such an expression is represented as a tree of operators. The leaves of this tree are made of variables, constants or even traditional constraints. Variables and constants can be combined as `ExpressionVariable` using operators (e.g, `mult(10,abs(w))`), or using simple constraints (e.g., `leq(z,9)`), or even using global constraints (e.g, `alldifferent(vars)`). The language available on expressions is therefore slightly richer and matches the language used in the Constraint Solver Competition 2008 of the CPAI workshop.

For example, the following expression

$$((x = 10 * |y|) \lor (z \le 9)) \iff \texttt{alldifferent}(a, b, c)$$

could be represented by :

```
Constraint exp = ifOnlyIf( or( eq(x, mult(10, abs(y))), leq(z, 9) ),
                     alldifferent(new IntegerVariable[]{a,b,c}) );
```

Handling complex expressions.

Expressions offer a more powerful modeling language than the one available via standard constraints. However, they can not be handled as efficiently as the standard constraints that embed a dedicated propagation algorithm. We therefore recommend you to carefully check that you can not model the expression using the intensional constraints of Choco before using expressions. Inside the solver, expressions can be represented in two different ways that can be decided at the modeling level, using the following `Model` API:

```
setDefaultExpressionDecomposition(boolean decomp);
```

or the option `Options.E_DECOMP`.

- The first way (`decomp=false`) is to handle them as constraints in extension. The expression is then used to check a tuple in a dynamic way just like a n-ary relation that is defined without listing all the possible tuples. The expression is then propagated using the GAC3rm algorithm. This is very powerful as arc-consistency is obtained on the corresponding constraints.

- The second way (`decomp=true`) is to decompose the expression automatically by introducing intermediate variables and eventually the generic `reifiedIntConstraint`. By doing so, the level of pruning decreases but expressions of larger arity involving large domains can be represented.

Once the default representation is chosen, one can also make exception for a particular expression using options on `addConstraint`. For example, the following code tells the solver to decompose e1 and not e2 :

```
model.setDefaultExpressionDecomposition(false);
IntegerVariable x = makeIntVar("x", 1, 3, Options.V_BOUND);
IntegerVariable y = makeIntVar("y", 1, 3, Options.V_BOUND);
IntegerVariable z = makeIntVar("z", 1, 3, Options.V_BOUND);

Constraint e1 = or(lt(x, y), lt(y, x));
model.addConstraint(Options.E_DECOMP, e1);

Constraint e2 = or(lt(y, z), lt(z, y));
model.addConstraint(e2);
```

When and how should I use expressions ?

An expression (represented in extension) should be used in the case of a complex logical relationship that involves **few different variables**, each of **small domain**, and if **arc consistency** is desired on those variables. In such a case, an expression can even be more powerful than a model using intermediate variables and intensional constraints. Imagine the following "crazy" example :

```
or( and( eq( abs(sub(div(x,50),div(y,50))),1), eq( abs(sub(mod(x,50),mod(y,50))),2)),
    and( eq( abs(sub(div(x,50),div(y,50))),2), eq( abs(sub(mod(x,50),mod(y,50))),1)))
```

This expression has a small arity: it involves only two variables x and y. Let assume that their domains has no more than 300 values, then such an expression should typically not be decomposed. Indeed, arc consistency will create many holes in the domains and filter much more than if the relation was decomposed.

Conversely, an expression should be decomposed as soon as it involves a large number of variables, or at least one variable with a large domain.

2.2.8 Global constraints

Choco includes several global constraints, such as:

```
allDifferent, globalCardinality, atMostNValue, cumulative, lex, regular, tree,
geost, etc.
```

Those constraints offer dedicated filtering algorithms which are able to make deductions where a decomposed model would not. For instance, constraint $\texttt{alldifferent}(a, b, c, d)$ with $a, b \in [1, 4]$ and $c, d \in [3, 4]$ allows to deduce that a and b cannot be instantiated to 3 or 4; such rule cannot be inferred by simple binary constraints.

The up-to-date list of global constraints available in Choco can be found within the Javadoc API. Most of these global constraints are also described in Section Constraints.

2.2.9 Scheduling constraints

See also scheduling application.

```
cumulative, disjunctive, geost, pack, preceding, precedenceDisjoint,
precedenceImplied, precedenceReified.
```

2.2.10 Sequencing constraints

```
multiCostRegular, regular, stretchCyclic, stretchPath,
```

Chapter 3

The solver

To create a `Solver`, one just needs to create a new object as follow:

```
Solver solver = new CPSolver();
```

By this, a Constraint Programming (CP) `Solver` object is created.

The solver gives an API to read a model. The reading of a model is compulsory and must be done after the entire definition of the model.

```
solver.read(model);
```

The reading is divided in 2 parts: variables reading and constraints reading.

3.1 Variables reading

The solver iterates over the variables of the Model to create solver-specific variables and domains (as defined in the model). Thus, three types of variables can be created: integer variables, real variables and set variables. Depending on the constructor, the correct domain is created (like bounded domain or enumerated domain for integer variables).

> **Bound variables** are related to large domains which are only represented by their lower and upper bounds. The domain is encoded in a space efficient way and propagation events only concern bound updates. Value removals between the bounds are therefore ignored (*holes* are not considered). The level of consistency achieved by most constraints on these variables is called *bound-consistency*.
> On the contrary, the domain of an **enumerated variable** is explicitly represented and every value is considered while pruning. Basic constraints are therefore often able to achieve *arc-consistency* on enumerated variables (except for NP global constraint such as the cumulative constraint). Remember that switching from an enumerated variable to a bounded variables decrease the level of propagation achieved by the system.

Model variables and Solver variables are distinct. Solver variables are solver representation of the model variables. One can't access to variable value directly from the model variable. To access to a model variable thanks to the solver, use the following `Solver` API: `getVar(Variable v);`

3.1.1 Solver and IntegerVariables

A model integer variable can be accessed by the method `getVar(IntegerVariable v)` which returns a `IntDomainVar` object:

```
IntegerVariable x = makeEnumIntVar("x", 1, 100); // model variable
IntDomainVar xOnSolver = solver.getVar(x); // solver variable
```

The state of an `IntDomainVar` can be accessed through the main following public methods :

IntDomainVar API	description
hasEnumeratedDomain()	checks if the variable is an enumerated or a bound one
getInf()	returns the lower bound of the variable
getSup()	returns the upper bound of the variable
getVal()	returns the value if it is instantiated
isInstantiated()	checks if the domain is reduced to a singleton
canBeInstantiatedTo(int v)	checks if the value v is contained in the domain of the variable
getDomainSize()	returns the current size of the domain

For more informations on advanced uses of such `IntDomainVar`, see advanced uses.

3.1.2 Solver and SetVariables

A model set variable can be access by the method `getVar(SetVariable v)` which returns a `SetVar` object:

```
SetVariable x = makeBoundSetVar("x", 1, 40); // model variable
SetVar xOnSolver = solver.getVar(x); // solver variable
```

A set variable on integer values between $[1, n]$ has 2^n values (every possible subsets of $\{1..n\}$). This makes an exponential number of values and the domain is represented with two bounds corresponding to the intersection of all possible sets (called the kernel) and the union of all possible sets (called the envelope) which are the possible candidate values for the variable.

The state of a `SetVar` can be accessed through the main following public methods on the SetVar class:

SetVar API	description
getCard()	returns the `IntDomainVar` representing the cardinality of the set variable
isInDomainKernel(int v)	checks if value v is contained in the current kernel
isInDomainEnveloppe(int v)	checks if value v is contained in the current envelope
getDomain()	returns the domain of the variable as a `SetDomain`. Iterators on envelope or kernel can than be called
getKernelDomainSize()	returns the size of the kernel
getEnveloppeDomainSize()	returns the size of the envelope
getEnveloppeInf()	returns the first available value of the envelope
getEnveloppeSup()	returns the last available value of the envelope
getKernelInf()	returns the first available value of the kernel
getKernelSup()	returns the last available value of the kernel
getValue()	returns a table of integers `int[]` containing the current domain

For more informations on advanced uses of such `SetVar`, see advanced uses.

3.1.3 Solver and RealVariables

> *Real variables are still under development but can be used to solve toy problems such as small systems of equations.*

A model real variable can be access by the method `getVar(RealVariable v)` which returns a `RealVar` object:

```
RealVariable x = makeRealVar("x", 1.0, 3.0); // model variable
RealVar xOnSolver = s.getVar(x); // solver variable
```

Continuous variables are useful for non linear equation systems which are encountered in physics for example.

`RealVar` API	description
`getInf()`	returns the lower bound of the variable (`double`)
`getSup()`	returns the upper bound of the variable (`double`)
`isInstantiated()`	checks if the domain of a variable is reduced to a canonical interval. A canonical interval indicates that the domain has reached the precision given by the user or the solver

For more informations on advanced uses of such `RealVar`, see advanced uses.

3.2 Constraints reading

After variables, the Solver iterates over the constraints added to the Model. It creates Solver constraints that encapsulates a filtering algorithm which are called when a propagation step occur or when external events happen on the variables belonging to the constraint, such as value removals or bounds modifications. And it add it to the constraint network.

3.3 Search Strategy

A key ingredient of any constraint approach is a clever branching strategy. The construction of the search tree is done according to a series of Branching objects (that plays the role of achieving intermediate goals in logic programming). The user may specify the sequence of branching objects to be used to build the search tree. A common way to branch in CP is by assigning variables to values (such a branching is called `AssignVar` in choco). We will present in this section how to define your branching strategies with existing variables and values selectors.

3.3.1 Why is it important to define a search strategy ?

Once a fix point is reached, the Solver needs to select a variable and its value to continue the search. The way variables and values are chosen has a **real impact on the resolution step efficient**.

> *The search strategy should not be overlooked!!* An adapted search strategy can reduce: the execution time, the number of node expanded, the number of backtrack done.

Let see that small example:

```
Model m = new CPModel();
int n = 1000;
IntegerVariable var = makeBoundIntVar("var", 0, 2);
IntegerVariable[] bi = makeEnumIntVarArray("b", n, 0, 1);
m.addConstraint(eq(var, sum(bi)));

Solver badStrat = new CPSolver();
badStrat.read(m);
badStrat.setVarIntSelector(new MinDomain(badStrat));
badStrat.setValIntIterator(new IncreasingDomain());
badStrat.solve();
badStrat.printRuntimeSatistics();

Solver goodStrat = new CPSolver();
goodStrat.read(m);
goodStrat.setVarIntSelector(new MinDomain(goodStrat,
                              goodStrat.getVar(new IntegerVariable[]{var})));
```

```
    goodStrat.setValIntIterator(new DecreasingDomain());
    goodStrat.solve();
    goodStrat.printRuntimeSatistics();
```

This model ensures that $var = b_0 + b_1 + \ldots + b_{1000}$ where var has a small domain and b_i is a binary variable. The propagation has no effect on any domain and a fix point is reached at the beginning of the search. So, a decision has to be done choosing a variable and its value. As the default variable selector is `MinDomain` (see below), the solver will iterate over the variables, starting by the 1000 binary variables and ending with var, and 1001 nodes will be created.

3.3.2   Variable and value selection

A common way to explore the search tree in CP is by assigning values to variables. The branching class of Choco dedicated to this kind of search is **AssignVar**. More complex branching schemes can be performed in Choco but this section lists the default strategies available for exploring the search tree by assigning variables and that can be used within an **AssignVar** branching. These strategies are called *variable and value selection heuristics*.

The heuristics available in Choco and the API for selecting a given heuristic depend on the type of the considered variables. As instance, for integer variables, the default branching heuristic used by Choco is to choose the variable with current minimum domain size first (`MinDomain`) and to take its values in increasing order (`IncreasingDomain`). Customizing the value and variable heuristics on the integer variables of the solver can be done (before calling the `solve()` method) using the `Solver` API, as shown in the following example:

```
// select the next branching variable randomly
solver.setVarIntSelector(new RandomIntVarSelector(solver));
// select the values in increasing order
solver.setValIntIterator(new DecreasingDomain());
// *OR* select the next value randomly
solver.setValIntSelector(new RandomIntValSelector());
```

Variable selector.

It defines the way to choose the non instantiated variable on which the next decision will be made. A variable selector can be set using the following API:

`Solver` API	Variable	Default strategy
`setVarIntSelector(VarSelector)`	Integer	MinDomain
`setVarRealSelector(RealVarSelector)`	Real	CyclicRealVarSelector
`setVarSetSelector(SetVarSelector)`	Set	MinDomSet

The variable selectors currently available in Choco are the following: to complete

Integer Variable Selector	description
`StaticVarOrder(IntDomainVar[])`	A heuristic selecting the first non instantiated variable in the given static order
`MinDomain(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest domain
`DomOverDeg(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest ration (domainSize / degree), the *degree* of a variable is the number of constraints linked to it.
`DomOverDynDeg(Solver, IntDomainVar[])`	A heuristic selecting the variable with smallest degree, the *degree* of a variable is the number of constraints linked to it that is not completely instanciated.
`DomOverWDeg(Solver, IntDomainVar[])`	see example.
`MostConstrained(final Solver, final IntDomainVar[])`	A heuristic selecting the variable with the maximum degree
`RandomIntVarSelector(Solver, IntDomainVar[], long)`	A heuristic selecting randomly the non instantiated variable
`CompositeIntVarSelector(ConstraintSelector, HeuristicIntVarSelector)`	Composes two heuristics for selecting a variable: a first heuristic is applied for selecting a constraint. From that constraint a second heuristic is applied for selecting the variable
`LexIntVarSelector(HeuristicIntVarSelector, HeuristicIntVarSelector)`	applies two heuristics lexicographically for selecting a variable: a first heuristic is applied finding the best constraint, ties are broken with the second heuristic

Set Variable Selector	description
`MinDomSet(Solver)`	A heuristic selecting the variable with the smallest domain
`RandomSetVarSelector(Solver, SetVar[], long)`	A heuristic selecting randomly the variable
`StaticSetVarOrder(SetVar[])`	A heuristic selecting the first non instantiated variable in the given static order

Real Variable Selector	description
`CyclicRealVarSelector(Solver solver)`	Since a dichotomy algorithm is used, cyclic assiging is nedded for instantiate a real interval variable. A variable is selected several times to split its domain until it reaches the desired precision

> Solver variables have to be specified (not Model variables).

Value iterator

Once the variable has been choose, the Solver has to compute its value. The first way to do it is to schedule the value once and give an iterator to the solver. It can be done using the following API:

Solver API	Variable	Default strategy
`setValIntIterator(ValIterator)`	Integer	IncreasingDomain
`setValRealIterator(RealValIterator)`	Real	RealIncreasingDomain
`setValSetIterator(ValIterator)`	Set	MinEnv

The value iterators currently available in Choco are the following: to complete

Integer Value Iterator	description
`DecreasingDomain()`	A heuristic selecting value from the upper bound to the lower bound
`IncreasingDomain()`	A heuristic selecting value from the lower bound to the upper bound

Real Value Iterator	description
`RealIncreasingDomain()`	A heuristic selecting value from the lower bound to the upper bound

Value selector

The second way to do it is to compute the following value at each call. It can be done using the following API:

Solver API	Variable	Default strategy
`setValIntSelector(ValSelector)`	Integer	(none: see *value iterator*)
`setValRealSelector(ValSelector)`	Real	(none: see *value iterator*)
`setValSetSelector(SetValSelector)`	Set	(none: see *value iterator*)

The value selectors currently available in Choco are the following: to complete

Integer Value Selector	description
`MaxVal()`	Selecting the highest value in the domain
`MidVal()`	Selecting the middle value in the domain
`MinVal()`	Selecting the lowest value in the domain
`RandomIntValSelector()`	Selecting randomly the value in the domain

Set Value Selector	description
`MinEnv(Solver)`	Selecting the lowest value in the envelope and not in the kernel of the domain

3.3.3 Building a sequence of branching object

You might want to apply different heuristics to different set of variables of the problem. In that case, the search is viewed as a sequence of branching objects (or goals). Up to now, we only had one branching or one goal including all the variables of the problem but several goals can be used.

Adding a new goal is made through the solver with the **addGoal(AbstractBranching b)** method. As for the addition of your own limit, dont call the **solve()** method, but instead: build the solver by yourself, add your sequence of branching, and call the **launch()** method of the solver.

The following example add three branching objects on integer variables *vars1*, *vars2* and set variables *svars* to solver *s*. The first two branchings are both **AssignVar** but use two different variable/values selection strategies:

```
s.attachGoal(new AssignVar(new MinDomain(s,s.getVar(vars1)), new IncreasingDomain()));
s.addGoal(new AssignVar(new DomOverDeg(s,s.getVar(vars2)),new DecreasingDomain());
s.addGoal(new AssignSetVar(new MinDomSet(s,s.getVar(svars)), new MinEnv(s)));
s.generateSearchStrategy();
s.launch();
```

An example of how to set the search solver in case of optimization in given in the tutorial on cumulative.

3.3.4 Dom/WDeg

to introduce

```
Solver s = new CPSolver();
s.read(model);

s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
```

```
s.setFirstSolution(true);
s.generateSearchStrategy();
```

The decision variables can be set using :

```
DomOverWDegBranching dwdeg = new DomOverWDegBranching(s, new IncreasingDomain());
dwdeg.setBranchingVars(vars);
```

3.3.5 Impacts

to introduce

```
CPSolver s = new CPSolver();
s.read(model);

//create the branching on the decision variables vars.
ImpactBasedBranching ibb = new ImpactBasedBranching(s, vars);

//initialize the impacts with a time limit of 10s
ibb.getImpactStrategy().initImpacts(10000);

s.generateSearchStrategy();
s.attachGoal(ibb);
s.setFirstSolution(true);
s.launch();
```

3.3.6 Restarts

You can set geometric restarts by using the following API available on the solver:

```
setGeometricRestart(int base, double grow);
setGeometricRestart(int base, double grow, int restartLimit);
```

It performs a search with restarts regarding the number of backtrack. An initial allowed number of backtrack is given (parameter base) and once this limit is reached a restart is performed and the new limit imposed to the search is increased by multiplying the previous limit with the parameter grow. restartLimit parameter states the maximum number of restarts. Restart strategies makes really sense with strategies that make choices based on the past experience of the search : `DomOverWdeg` or Impact based search. It could also be used with a random heuristic

```
CPSolver s = new CPSolver();
s.read(model);

s.setGeometricRestart(14, 1.5d);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
s.launch();
```

You can also set Luby restarts by using the following API available on the solver:

```
setLubyRestart(int base);
setLubyRestart(int base, int grow);
setLubyRestart(int base, int grow, int restartLimit);
```

it performs a search with restarts regarding the number of backtracks. One way to describe this strategy is to say that all run lengths are power of two, and that each time a pair of runs of a given length has been completed, a run of twice that length is immediatly executed. The limit is equals to *length*base*.

- **example with growing factor of 2 : [1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, 1,...]**

- **example with growing factor of 3 : [1, 1, 1, 3, 1, 1, 1, 3, 9,...]**

```
CPSolver s = new CPSolver();
s.read(model);

s.setLubyRestart(50, 2, 100);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranching(s, new IncreasingDomain()));
s.launch();
```

3.4   Limiting Search Space

The Solver class provides some limits on the search strategy that you can fix or just monitor. Limits may be imposed on the search algorithm to avoid spending too much time in the exploration. The limits are updated and checked each time a new node is created. It has to be specified before the resolution. After having created the solver, you can specify whether or not you want to fix a limit:

time limit State a time limit on tree search. When the execution time is equal to the time limit, the search stops whatever a solution is found or not. You can define a time limit with the following API : `setTimeLimit(int timeLimit)` where unit is millisecond. Or just monitor (or not) the search time with the API : `monitorTimeLimit(boolean b)`. The default value is set to `true`. Finally, you can get the time limit, once the solve method has been called, with the API: `getTimeCount()`

node limit State a node limit on tree search. When the number of nodes explored is equal to the node limit, the search stops whatever a solution is found or not. You can define a node limit with the following API: `setNodeLimit(int nodeLimit)` where unit is the number of nodes. Or just monitor (or not) the number of nodes explored with the API: `monitorNodeLimit(boolean b)`. The default value is set to `true`. Finally, you can get the node limit, once the solve method has been called, with the API: `getNodeCount()`

backtrack limit State a backtrack limit on tree search. When the number of backtracks done is equal to the backtrack limit, the search stops whatever a solution is found or not. You can define a backtrack limit with the following API: `setBackTrackLimit(int backtrackLimit)` where unit is the number of backtracks. Or just monitor (or not) the number of backtrack done with the API: `monitorBackTrackLimit(boolean b)`. The default value is set to `false`. Finally, you can get the backtrack limit, once the solve method has been called, with the API: `getBackTrackCount()`

fail limit State a fail limit on tree search. When the number of failure is equal to the fail limit, the search stops whatever a solution is found or not. You can define a fail limit with the following API : `setFailLimit(int failLimit)` where unit is the number of failure. Or just monitor (or not) the number of failure encountered with the API : `monitorFailLimit(boolean b)`. The default value is set to `false`. Finally, you can get the fail limit, once the solve method has been called, with the API : `getFailCount()`

CPU time limit State a CPU limit on tree search. When the CPU time (user + system) is equal to the CPU time limit, the search stops whatever a solution is found or not. You can define a CPU time limit with the following API: `setCpuTimeLimit(int timeLimit)` where unit is millisecond. Or just monitor (or not) the search time with the API: `monitorCpuTimeLimit(boolean b)`. The default value is set to `false`. Finally, you can get the CPU time limit, once the solve method has been called, with the API: `getCpuTimeCount()`

add example

3.5   Solve a problem

As Solver is the second element of a Choco program, the control of the search process without using predefined tools is made on the Solver.

Solver API	description
solve()	Compute the first solution of the Model, if the Model is feasible.
solve(boolean all)	If *all* is set to true, computes all solutions of the Model, if the Model is feasible.
solveAll()	Computes all the solution of the Model, if the Model is feasible.
propagate()	Computes initial propagation of the Model, and reachs the first Fix Point. It reduces variables Domain through constraints linked and other variables domain. Can throw a ContradictionException if the Solver detects a contradiction in the Model.
maximize(Var obj, boolean restart)	Allows user to find a solution that maximizing the objective varible *obj*. The optimization finds a first solution then finds a new solution that improves *obj* and so on till no other solution can be found that improves *obj*. Parameter *restart* is a boolean indicating whether the Solver will restart the search after each solution found (if set to true) or if it will keep backtracking from the leaf of the last solution found. See example. **Beware**: the variable *obj* expected must be a Solver variable and not a Model variable.
minimize(Var obj, boolean restart)	Allows user to find a solution that minimizing the objective varible *obj*. The optimization finds a first solution then finds a new solution that improves *obj* and so on till no other solution can be found that improves *obj*. Parameter *restart* is a boolean indicating whether the Solver will restart the search after each solution found (if set to true) or if it will keep backtracking from the leaf of the last solution found. See example. **Beware**: the variable *obj* expected must be a Solver variable and not a Model variable.
nextSolution()	Allows the Solver to find the next solution, if one or more solution have already been find with solve() or nextSolution().
isFeasible()	Indicates whether or not the Model has at least one solution.

3.5.1 Solver settings

Logs

The solver class is instrumented in order to produce trace statements throughout search. The verbosity level of the solver can be set, by the following static method

```
CPSolver.setVerbosity(CPSolver.SEARCH);
// And after solver.solve()
CPSolver.flushLogs();
```

The code above ensure that messages are printed in order to describe the construction of the search tree.

Five verbosity levels are available:

Level	prints...
CPSolver.SILENT	nothing
CPSolver.SOLUTION	messages whenever a solution is reached
CPSolver.SEARCH	a message at each choice point
CPSolver.PROPAGATION	messages to trace propagation
CPSolver.FINEST	high level messages to trace propagation

Note, that in the case of a verbosity set to CPSolver.SEARCH, trace statements are printed up to a maximal depth in the search tree. By default, only the 5 first levels are traced, but you can change the value of this threshold, say to 10, with the following setter method:

```
solver.setLoggingMaxDepth(10);
```

3.5.2   Optimization

to introduce

```
Model m = new CPModel();
IntegerVariable obj1 = makeEnumIntVar("obj1", 0, 7);
IntegerVariable obj2 = makeEnumIntVar("obj1", 0, 5);
IntegerVariable obj3 = makeEnumIntVar("obj1", 0, 3);
IntegerVariable cost = makeBoundIntVar("cout", 0, 1000000);
int capacity = 34;
int[] volumes = new int[]{7, 5, 3};
int[] energy = new int[]{6, 4, 2};
// capacity constraint
m.addConstraint(leq(scalar(volumes, new IntegerVariable[]{obj1, obj2, obj3}), capacity));

// objective function
m.addConstraint(eq(scalar(energy, new IntegerVariable[]{obj1, obj2, obj3}), cost));

Solver s = new CPSolver();
s.read(m);

s.maximize(s.getVar(cost), false);
```

Chapter 4

Advanced uses of Choco

4.1 Environment

Environment is a central object of the backtracking system. It defines the notion of *world*. A world contains values of storable objects or operations that permit to *backtrack* to its state. The environment *pushes* and *pops* worlds.

There are *primitive* data types (`IstateBitSet, IStateBool, IStateDouble, IStateInt, IStateLong`) and *objects* data types (`IStateBinarytree, IStateIntInterval, IStateIntProcedure, IStateIntVector, IStateObject, IStateVector`).

There are two different environments: *EnvironmentTrailing* and *EnvironmentCopying*.

4.1.1 Copying

In that environment, each data type is defined by a value (primitive or object) and a timestamp. Every time a world is pushed, each value is copied in an array (one array per data type), with finite indice. When a world is popped, every value is restored.

4.1.2 Trailing

In that environment, data types are defined by its value. Every operation applied to a data type is pushed in a *trailer*. When a world is pushed, the indice of the last operation is stored. When a world is popped, these operations are popped and *unapplied* until reaching the last operation of the previous world.
Default one in CPSolver

4.2 Define your own limit search space

To define your own limits/statistics (notice that a limit object can be used only to get statistics about the search), you can create a limit object by extending the **AbstractGlobalSearchLimit** class or implementing directly the interface **IGlobalSearchLimit**. Limits are managed at each node of the tree search and are updated each time a node is open or closed. Notice that limits are therefore time consuming. Implementing its own limit need only to specify to the following interface :

```
/**
 * The interface of objects limiting the global search exploration
 */
public interface GlobalSearchLimit {

  /**
   * resets the limit (the counter run from now on)
   * @param first true for the very first initialization, false for subsequent ones
   */
  public void reset(boolean first);
```

```
   /**
    * notify the limit object whenever a new node is created in the search tree
    * @param solver the controller of the search exploration, managing the limit
    * @return true if the limit accepts the creation of the new node, false otherwise
    */
   public boolean newNode(AbstractGlobalSearchSolver solver);

   /**
    * notify the limit object whenever the search closes a node in the search tree
    * @param solver the controller of the search exploration, managing the limit
    * @return true if the limit accepts the death of the new node, false otherwise
    */
   public boolean endNode(AbstractGlobalSearchSolver solver);
}
```

Look at the following example to see a concrete implementation of the previous interface. We define here a limit on the depth of the search (which is not found by default in choco). The `getWorldIndex()` is used to get the current world, i.e the current depth of the search or the number of choices which have been done from baseWorld.

```
public class DepthLimit extends AbstractGlobalSearchLimit {

  public DepthLimit(AbstractGlobalSearchSolver theSolver, int theLimit) {
    super(theSolver,theLimit);
    unit = "deep";
  }

  public boolean newNode(AbstractGlobalSearchSolver solver) {
    nb = Math.max(nb, this.getProblem().getWorldIndex()
    this.getProblem().getSolver().getSearchSolver().baseWorld);
    return (nb < nbMax);
  }

  public boolean endNode(AbstractGlobalSearchSolver solver) {
    return true;
  }

  public void reset(boolean first) {
   if (first) {
    nbTot = 0;
   } else {
    nbTot = Math.max(nbTot, nb);
   }
   nb = 0;
  }
```

Once you have implemented your own limit, you need to tell the search solver to take it into account. Instead of using a call to the `solve()` method, you have to create the search solver by yourself and add the limit to its limits list such as in the following code :

```
  Solver s = new CPSolver();
  s.read(model);
  s.setFirstSolution(true);
  s.generateSearchStrategy();
  s.getSearchStrategy().limits.add(new DepthLimit(s.getSearchStrategy(),10));
  s.launch();
```

4.3   Define your own search strategy

Section Search strategy presented the default branching strategies available in Choco and showed how to post them or to compose them as goals. In this section, we will start with a very simple and common way to branch by choosing values for variables and specially how to define its own variable/value selection strategy. We will then focus on more complex branching such as dichotomic or n-ary choices. Finally we will show how to control the search space in more details with well known strategy such as LDS (Limited discrepancy search).

For integer variables, the variable and value selection strategy objects are based on the following interfaces:

- `IntVarSelector` : Interface for the (integer) variable selection

- `ValIterator` : Interface to describes an iteration scheme on the domain of a variable

- `ValSelector` : Interface for a value selection

Concrete examples of these interfaces are respectively `DomOverDeg`, `IncreasingDomain`, `MinVal`. The default branchings currently supported by Choco are available in packages `src.choco.cpsolver.search.integer` for integer variables, `src.choco.cpsolver.search.set` for set variables, `src.choco.cpsolver.search.real` for real variables.

4.3.1   Define your own variable selection

You may extend this small library of branching schemes and heuristics by defining your own concrete classes of `AbstractIntVarSelector`. We give here an example of an `IntVarSelector` with the implementation of a static variable ordering :

```java
public class StaticVarOrder extends AbstractIntVarSelector {

    // the sequence of variables that need be instantiated
    protected IntDomainVar[] vars;

    public StaticVarOrder(IntDomainVar[] vars) {
        this.vars = vars;
    }

    public IntDomainVar selectIntVar() {
        for (int i = 0; i < vars.length; i++)
            if (!vars[i].isInstantiated())
                return vars[i];
        return null;
    }
}
```

Notice on this example that you only need to implement method `selectIntVar()` which belongs to the contract of `IntVarSelector`. This method should return a non instantiated variable or `null`. Once the branching is finished, the next branching (if one exists) is taken by the search algorithm to continue the search, otherwise, the search stops as all variable are instantiated. To avoid the loop over the variables of the branching, a backtrackable integer (`StoredInt`) could be used to remember the last instantiated variable and to directly select the next one in the table. Notice that backtrackable structures could be used in any of the code presented in this chapter to speedup the computation of dynamic choices.

You can add your variable selector to the solver as common variable selector, using the `Solver` API:

```java
solver.setVarIntSelector(new MyVarSelector(...));
```

4.3.2   Define your own value selection

You may also define your own concrete classes of `ValIterator` or `ValSelector`.

Value selector

We give here an example of an `IntValSelector` with the implementation of a minimum value selecting:

```java
public class MinVal extends AbstractSearchHeuristic implements ValSelector {
  /**
    * selecting the lowest value in the domain
    * @param x the variable under consideration
    * @return what seems the most interesting value for branching
    */
  public int getBestVal(IntDomainVar x) {
      return x.getInf();
  }
}
```

Only `getBestVal()` method must be implemented, returning the best value *in the domain* according to the heuristic.

You can add your value selector to the solver as common variable selector, using the `Solver` API:

```java
solver.setValIntSelector(new MyValSelector(...));
```

> Using a value selector with bounded domain variable is strongly inadvised, except if it pick up bounds value. If the value selector pick up a value that is not a bound, when it goes up in the tree search, that value could be not removed and picked twice (or more)!

Values iterator

We give here an example of an `ValIterator` with the implementation of an increasing domain iterator:

```java
public final class IncreasingDomain implements ValIterator {
  /**
    * testing whether more branches can be considered after branch i,
    * on the alternative associated to variable x
    * @param x the variable under scrutiny
    * @param i the index of the last branch explored
    * @return true if more branches can be expanded after branch i
    */
  public boolean hasNextVal(Var x, int i) {
      return (i < ((IntDomainVar) x).getSup());
  }

  /**
    * accessing the index of the first branch for variable x
    * @param x the variable under scrutiny
    * @return the index of the first branch: first value to be assigned to x
    */
  public int getFirstVal(Var x) {
      return ((IntDomainVar) x).getInf();
  }

  /**
    * generates the index of the next branch after branch i,
    * on the alternative associated to variable x
    * @param x the variable under scrutiny
    * @param i the index of the last branch explored
    * @return the index of the next branch to be expanded after branch i
    */
```

```java
    public int getNextVal(Var x, int i) {
        return ((IntDomainVar) x).getNextDomainValue(i);
    }
}
```

You can add your value iterator to the solver as common variable selector, using the `Solver` API:

```java
s.setValIntIterator(new MyValIterator(..));
```

4.3.3   How does a search loop work ?

The seach loop is created when a `solve()` method is called. It goes down and up in the branches in order to cover the tree search.

Algorithm of the search loop in Choco

```
next_move = new node
WHILE no solution AND in search limit
     IF next_move is new node
    THEN
        create a new node : variable/value selection ;
        IF node exists
        THEN
            next_move <-- go down branch ;
        ELSE
            next_move <-- go up branch ;
            solution is found ;

    ELSE IF next_move is go down branch
        propagate ;
        IF no contradiction
        THEN
            next_move <-- new node ;
        ELSE
            next_move <-- go up branch ;

    ELSE IF next_move is go up branch
        find next branch ;
        propagate ;
        IF has next branch AND no contradiction
        THEN
            next_move <-- go down branch ;
        ELSE
            next_move <-- go up branch ;

    END IF

END WHILE
```

4.3.4   How to define your own Branching object

Beyond Variable/value selection...
    under development See old version

4.4   Define your own constraint

This section describes how to add you own constraint, with specific propagation algorithms. Note that this section is only useful in case you want to express a constraint for which the basic propagation

algorithms (using tables of tuples, or boolean predicates) are not efficient enough to propagate the constraint.

The general process consists in defining a new constraint class and implementing the various propagation methods. We recommend the user to follow the examples of existing constraint classes (for instance, such as `GreaterOrEqualXYC` for a binary inequality)

4.4.1 The constraint hierarchy

Each new constraint must be represented by an object implementing the **SConstraint** interface (`S` for solver constraint). To help the user defining new constraint classes, several abstract classes defining `SConstraint` have been implemented. These abstract classes provide the user with a management of the constraint network and the propagation engineering. They should be used as much as possible.

For constraints on integer variables, the easiest way to implement your own constraint is to inherit from one of the following classes, depending of the number of solver integer variables (`IntDomainVar`) involved:

Default class to implement	number of solver integer variables
`AbstractUnIntSConstraint`	**one** variable
`AbstractBinIntSConstraint`	**two** variables
`AbstractTernIntSConstraint`	**three** variables
`AbstractLargeIntSConstraint`	any number of variables.

Constraints over integers must implement the following methods (grouped in the `IntSConstraint` interface):

Method to implement	description
`pretty()`	Returns a pretty print of the constraint
`propagate()`	The main propagation method (propagation from scratch). Propagating the constraint until local consistency is reached.
`awake()`	Propagating the constraint for the very first time until local consistency is reached. The awake is meant to initialize the data structures contrary to the propagate. Specially, it is important to avoid initializing the data structures in the constructor.
`awakeOnInst(int x)`	Default propagation on instantiation: full constraint re-propagation.
`awakeOnBounds(int x)`	Default propagation on improved bounds: propagation on domain revision.
`awakeOnRemovals(int x, IntIterator v)`	Default propagation on mutliple values removal: propagation on domain revision. The iterator allow to iterate over the values that have been removed.

Methods `awakeOnBounds` and `awakeOnRemovals` can be replaced by more fine grained methods:	
`awakeOnInf(int x)`	Default propagation on improved lower bound: propagation on domain revision.
`awakeOnSup(int x)`	Default propagation on improved upper bound: propagation on domain revision.
`awakeOnRem(int x, int v)`	Default propagation on one value removal: propagation on domain revision.

To use the constraint in expressions or reification, the following minimum API is mandatory:	
`isSatisfied(int[] x)`	Tests if the constraint is satisfied when the variables are instantiated.
`isEntailed()`	Checks if the constraint must be checked or must fail. It returns true if the constraint is known to be satisfied whatever happend on the variable from now on, false if it is violated.
`opposite()`	It returns an AbstractSConstraint that is the opposite of the current constraint.

In the same way, a **set constraint** can inherit from `AbstractUnSetSConstraint`, `AbstractBinSetSConstraint`, `AbstractTernSetSConstraint` or `AbstractLargeSetSConstraint`.

A **real constraint** can inherit from `AbstractUnRealSConstraint`, `AbstractBinRealSConstraint` or `AbstractLargeRealSConstraint`.

A mixed constraint between **set and integer variables** can inherit from `AbstractBinSetIntSConstraint` or `AbstractLargeSetIntSConstraint`.

A simple way to implement its own constraint is to:

- create an empty constraint with only `propagate()` method implemented and every `awakeOnXxx()` ones set to `this.constAwake(false);`

- when the propagation filter is sure, separate it into the `awakeOnXxx()` methods in order to have finer granularity

- finally, if necessary, use backtrackables objects to improve the efficient of your constraint

---

How do I add my constraint to the Model ?

Adding your constraint to the model requires you to definite a specific constraint manager (that can be a inner class of your Constraint). This manager need to implement:

```
makeConstraint(Solver s, Variable[] vars, Object params, HashSet<String> options)
```

This method allows the Solver to create an instance of your constraint, with your parameters and Solver objects.

> If you create your constraint manager as an inner class, you must declare this class as **public and static**. If you don't, the solver can't instantiate your manager.

Once this manager has been implemented, you simply add your constraint to the model using the `addConstraint()` API with a `ComponentConstraint` object:

```
model.addConstraint( new ComponentConstraint(MyConstraintManager.class, params, vars) );
// OR
model.addConstraint( new ComponentConstraint("package.of.MyConstraint", params, vars) );
```

Where *params* is whatever you want (`Object[]`, `int`, `String`,...) and *vars* is an array of Model Variables (or more specific) objects.

4.4.2 Example: implement and add the `IsOdd` constraint

One creates the constraint by implementing the `AbstractUnIntSConstraint` (one integer variable) class:

```java
public class IsOdd extends AbstractUnIntSConstraint {

    @Override
    public int getFilteredEventMask(int idx) {
        return IntVarEvent.INSTINTbitvector;
    }

    public IsOdd(IntDomainVar v0) {
        super(v0);
    }

    /**
     * Default initial propagation: full constraint re-propagation.
     */

    public void awake() throws ContradictionException {
        DisposableIntIterator it = v0.getDomain().getIterator();
        try{
            while(it.hasNext()){
                int val = it.next();
                if(val%2==0){
                    v0.removeVal(val, cIdx0);
                }
            }
        }finally {
            it.dispose();
        }
    }

    /**
     * <i>Propagation:</i>
```

```
        * Propagating the constraint until local consistency is reached.
        *
        * @throws ContradictionException
        * contradiction exception
        */

    public void propagate() throws ContradictionException {
        if(v0.isInstantiated()){
            if(v0.getVal()%2==0){
                fail();
            }
        }
    }


}
```

To add the constraint to the model, one creates the following class (or inner class):

```
public class IsOddManager extends IntConstraintManager {
    public SConstraint makeConstraint(Solver solver, IntegerVariable[] variables, Object
        parameters, Set<String> options) {
        if (solver instanceof CPSolver) {
            return new IsOdd(solver.getVar(variables[0]));
        }
        return null;
    }
}
```

It calls the constructor of the constraint, with every *vars*, *params* and *options* needed.
    Then, the constraint can be added to a model as follows:

```
    // Creation of the model
    Model m = new CPModel();

    // Declaration of the variable
    IntegerVariable aVar = Choco.makeIntVar("a_variable", 0, 10);

    // Adding the constraint to the model, 1st solution:
    m.addConstraint(new ComponentConstraint(IsOddManager.class, null, new IntegerVariable[]{aVar
        }));
    // OR 2nd solution:
    m.addConstraint(new ComponentConstraint("myPackage.Constraint.IsOddManager", null, new
        IntegerVariable[]{aVar}));

    Solver s = new CPSolver();
    s.read(m);
    s.solve();
```

And that's it!!

4.4.3 Example of an empty constraint

See the complete code: ConstraintPattern.zip

```
  public class ConstraintPattern extends AbstractLargeIntSConstraint {

    public ConstraintPattern(IntDomainVar[] vars) {
        super(vars);
    }

    /**
```

```java
    * pretty print. The String is not constant and may depend on the context.
    * @return a readable string representation of the object
    */
   public String pretty() {
       return null;
   }


   /**
    * check whether the tuple satisfies the constraint
    * @param tuple values
    * @return true if satisfied
    */
   public boolean isSatisfied(int[] tuple) {
       return false;
   }


   /**
    * propagate until local consistency is reached
    */
   public void propagate() throws ContradictionException {
       // elementary method to implement
   }


   /**
    * propagate for the very first time until local consistency is reached.
    */
   public void awake() throws ContradictionException {
       constAwake(false); // change if necessary
   }



   /**
    * default propagation on instantiation: full constraint re-propagation
    * @param var index of the variable to reduce
    */
   public void awakeOnInst(int var) throws ContradictionException {
       constAwake(false); // change if necessary
   }


   /**
    * default propagation on improved lower bound: propagation on domain revision
    * @param var index of the variable to reduce
    */
   public void awakeOnInf(int var) throws ContradictionException {
       constAwake(false); // change if necessary
   }



   /**
    * default propagation on improved upper bound: propagation on domain revision
    * @param var index of the variable to reduce
    */
   public void awakeOnSup(int var) throws ContradictionException {
       constAwake(false); // change if necessary
   }


   /**
    * default propagation on improve bounds: propagation on domain revision
    * @param var index of the variable to reduce
    */
   public void awakeOnBounds(int var) throws ContradictionException {
```

```
        constAwake(false); // change if necessary
    }

    /**
     * default propagation on one value removal: propagation on domain revision
     * @param var index of the variable to reduce
     * @param val the removed value
     */
    public void awakeOnRem(int var, int val) throws ContradictionException {
        constAwake(false); // change if necessary
    }

    /**
     * default propagation on one value removal: propagation on domain revision
     * @param var index of the variable to reduce
     * @param delta iterator over remove values
     */
    public void awakeOnRemovals(int var, IntIterator delta) throws ContradictionException {
        constAwake(false); // change if necessary
    }
}
```

The first step to create a constraint in Choco is to implement all `awakeOn...` methods with `constAwake(false)` and to put your propagation algorithm in the `propagate()` method.

A constraint can choose not to react to fine grained events such as the removal of a value of a given variable but instead delay its propagation at the end of the fix point reached by "fine grained events" and fast constraints that deal with them incrementally (that's the purpose of the constraints events queue).

To do that, you can use `constAwake(false)` that tells the solver that you want this constraint to be called only once the variables events queue is empty. This is done so that heavy propagators can delay their action after the fast one to avoid doing a heavy processing at each single little modification of domains.

4.5   Define your own operator

to complete

4.6   Define your own variable

to complete

4.7   Backtrackable structures

to complete

4.8   Logging System

Choco logging system is based on the `java.util.logging` package and located in the package `common.logging`. Most Choco abstract classes or interfaces propose a static field `LOGGER`. The following figures present the architecture of the logging system with the default verbosity.

The shape of the node depicts the type of logger:

- The *house* loggers represent private loggers. Do not use directly these loggers because their level are low and all messages would always be displayed.

- The *octagon* loggers represent critical loggers. These loggers are provided in the variables, constraints and search classes and could have a huge impact on the global performances.
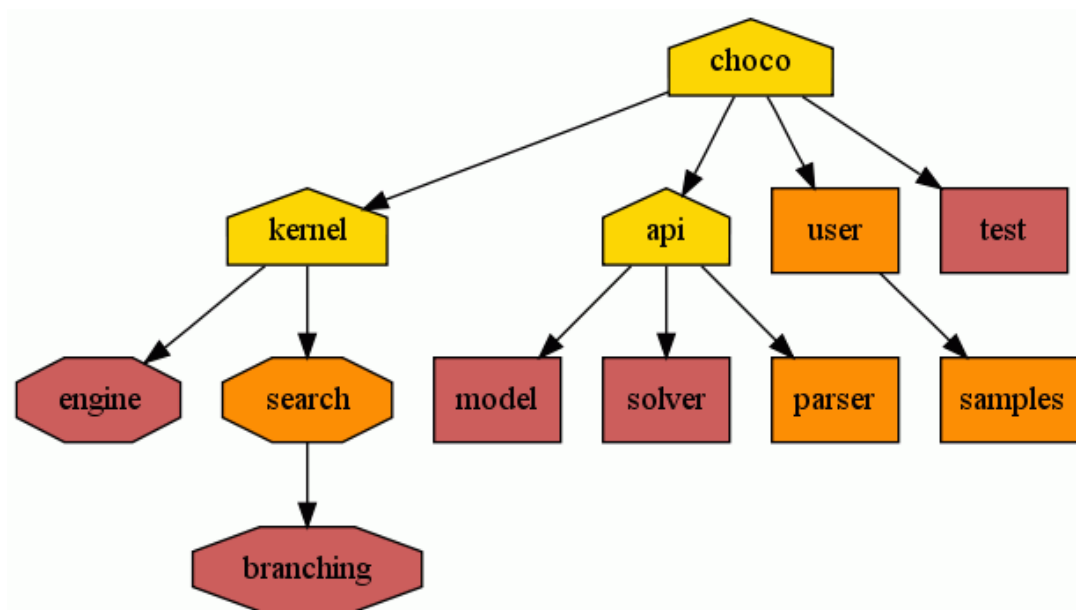
Figure 4.1: Logger Tree with the default verbosity

- The *box* loggers are provided for dev and users.

The color of the node gives its logging level with DEFAULT verbosity: `Level.FINEST` (gold), `Level.INFO` (orange), `Level.WARNING` (red).

Verbosity and messages.

The following table summarizes the verbosities available in choco:

- **OFF – level 0:** Disable logging.

- **SILENT – level 1:** Display only severe messages.

- **DEFAULT – level 2:** Display informations on final search state.

    - ON START

    ```
    ** CHOCO : Constraint Programming Solver
    ** CHOCO v2.1.1 (April, 2009), Copyleft (c) 1999-2010
    ```

    - ON COMPLETE SEARCH:

    ```
    - Search completed -
     [Maximize      : {0},]
     [Minimize      : {1},]
      Solutions     : {2},
      Times (ms)    : {3},
      Nodes         : {4},
      Backtracks    : {5},
      Restarts      : {6}.
    ```

    brackets [*line*] indicate *line* is optionnal,
    `Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective value in *maximize()* – –resp. *minimize()*– strategy.

    - ON COMPLETE SEARCH WITHOUT SOLUTIONS :

```
- Search completed - No solutions
 [Maximize     : {0},]
 [Minimize     : {1},]
  Solutions    : {2},
  Times (ms)   : {3},
  Nodes        : {4},
  Backtracks   : {5},
  Restarts     : {6}.
```

brackets [*line*] indicate *line* is optionnal,
`Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective
value in *maximize()* – –resp. *minimize()*– strategy.

– ON INCOMPLETE SEARCH:

```
- Search incompleted - Exiting on limit reached
  Limit        : {0},
 [Maximize     : {1},]
 [Minimize     : {2},]
  Solutions    : {3},
  Times (ms)   : {4},
  Nodes        : {5},
  Backtracks   : {6},
  Restarts     : {7}.
```

brackets [*line*] indicate *line* is optionnal,
`Maximize` –resp. `Minimize`– indicates the best known value before exiting of the objective
value in *maximize()* – –resp. *minimize()*– strategy.

- **VERBOSE – level 3:** Display informations on search state.

  – EVERY X (default=1000) NODES:

  ```
  - Partial search - [Objective : {0}, ]{1} solutions, {2} Time (ms), {3} Nodes, {4}
      Backtracks, {5} Restarts.
  ```

  `Objective` indicates the best known value.

  – ON RESTART :

  ```
  - Restarting search - {0} Restarts.
  ```

- **SOLUTION – level 4:** display all solutions.

  – AT EACH SOLUTION:

  ```
  - Solution #{0} found. [Objective: {0}, ]{1} Solutions, {2} Time (ms), {3} Nodes, {4}
      Backtracks, {5} Restarts.
    X_1:v1, x_2:v2...
  ```

- **SEARCH – level 5:** Display the search tree.

  – AT EACH NODE, ON DOWN BRANCH:

  ```
  ...[w] down branch X==v branch b
  ```

  where `w` is the current world index, `X` the branching variable, `v` the branching value and `b` the
  branch index. This message can be adapted on variable type and search strategy.

  – AT EACH NODE, ON UP BRANCH:

  ```
  ...[w] up branch X==v branch b
  ```

where `w` is the current world index, `X` the branching variable, `v` the branching value and `b` the branch index. This message can be adapted on variable type and search strategy.

- **FINEST** – **level 6:** display all logs.

More precisely, if the verbosity level is greater than DEFAULT, then the verbosity levels of the model and of the solver are increased to INFO, and the verbosity levels of the search and of the branching are slightly modified to display the solution(s) and search messages.

The verbosity level can be changed as follows:

```
ChocoLogging.setVerbosity(Verbosity.VERBOSE);
```

How to write logging statements ?

- Critical Loggers are provided to display error or warning. Displaying too much message really **impacts the performances**.

- Check the logging level before creating arrays or strings.

- Avoid multiple calls to `Logger` functions. Prefer to build a `StringBuilder` then call the `Logger` function.

- Use the `Logger.log` function instead of building string in `Logger.info()`.

Handlers.

Logs are displayed on `System.out` but warnings and severe messages are also displayed on `System.err`. `ChocoLogging.java` also provides utility functions to easily change handlers:

- Functions `set...Handler` remove current handlers and replace them by a new handler.

- Functions `add...Handler` add new handlers but do not touch existing handlers.

Define your own logger.

```
ChocoLogging.makeUserLogger(String suffix);
```

Part II

Elements of Choco

Chapter 5

Variables (Model)

This section describes the three kinds of variables that can be used within a Choco Model, and an object-variable.

5.1 Integer variables

`IntegerVariable` is a variable whose associated domain is made of integer values.

constructors:

Choco method	return type
`makeIntVar(String name, int lowB, int uppB, String... options)`	`IntegerVariable`
`makeIntVar(String name, List<Integer> values, String... options)`	`IntegerVariable`
`makeIntVar(String name, int[] values, String... options)`	`IntegerVariable`
`makeBooleanVar(String name, String... options)`	`IntegerVariable`
`makeIntVarArray(String name, int dim, int lowB, int uppB, String... options)`	`IntegerVariable[]`
`makeIntVarArray(String name, int dim, int[] values, String... options)`	`IntegerVariable[]`
`makeBooleanVarArray(String name, int dim, String... options)`	`IntegerVariable[]`
`makeIntVarArray(String name, int dim1, int dim2, int lowB, int uppB, String ... options)`	`IntegerVariable[][]`
`makeIntVarArray(String name, int dim1, int dim2, int[] values, String... options)`	`IntegerVariable[][]`

options:

- *no option* : equivalent to option `Options.V_ENUM`

- `Options.V_ENUM` : to force Solver to create enumerated domain for the variable.

- `Options.V_BOUND` : to force Solver to create bounded domain for the variable.

- `Options.V_LINK` : to force Solver to create linked list domain for the variable.

- `Options.V_BTREE` : to force Solver to create binary tree domain for the variable.

- `Options.V_BLIST` : to force Solver to create bipartite list domain for the variable.

- `Options.V_MAKEPSAN` : declare the current variable as makespan.

- `Options.V_NO_DECISION` : to force variable to be removed from the pool of decisional variables.

- `Options.V_OBJECTIVE` : to define the variable to be the one to optimize.

methods:

- `removeVal(int val)`: remove value *val* from the domain of the current variable

A variable with $\{0, 1\}$ domain is automatically considered as boolean domain.

Example:

```
    IntegerVariable ivar1 = makeIntVar("ivar1", -10, 10);
    IntegerVariable ivar2 = makeIntVar("ivar2", 0, 10000, Options.V_BOUND, Options.
        V_NO_DECISION);
    IntegerVariable bool = makeBooleanVar("bool");
```

Integer variables are illustrated on the n-Queens problem.

5.2 Real variables

`RealVariable` is a variable whose associated domain is made of real values. Only enumerated domain is available for real variables.

Such domain are memory consuming. In order to minimize the memory use and to have the precision you need, the model offers a way to set a precision (default value is 1.0e-6):

```
    Model m = new CPModel();
    m.setPrecision(0.01);
```

constructor:

Choco method	return type
`makeRealVar(String name, double lowB, double uppB, String... options)`	`RealVariable`

options:

- *no option* : no particular choice on decision or objective.

- `Options.V_NO_DECISION` : to force variable to be removed from the pool of decisional variables.

- `Options.V_OBJECTIVE` : to define the variable to be the one to optimize.

Example:

```
    RealVariable rvar1 = makeRealVar("rvar1", -10.0, 10.0);
    RealVariable rvar2 = makeRealVar("rvar2", 0.0, 100.0, Options.V_NO_DECISION, Options.
        V_OBJECTIVE);
```

Real variables are illustrated on the CycloHexan problem.

5.3 Set variables

`SetVariable` is high level modeling tool. It allows to represent variable whose values are sets. A SetVariable on integer values between $[1, n]$ has $2 * n$ values (every possible subsets of $\{1..n\}$). This makes an exponential number of values and the domain is represented with two bounds corresponding to the intersection of all possible sets (called the kernel) and the union of all possible sets (called the envelope) which are the possible candidate values for the variable. The consistency achieved on SetVariables is therefore a kind of bound consistency.

constructors:

Choco method	return type
makeSetVar(String name, int lowB, int uppB, String... options)	SetVariable
makeSetVarArray(String name, int dim, int lowB, int uppB, String... options)	SetVariable[]

options:

- *no option* : equivalent to option Options.V_ENUM

- Options.V_ENUM : to force Solver to create SetVariable with enumerated domain for the cardinality variable.

- Options.V_BOUND : to force Solver to create SetVariable with bounded cardinality.

- Options.V_NO_DECISION : to force variable to be removed from the pool of decisional variables.

- Options.V_OBJECTIVE : to define the variable to be the one to optimize.

The variable representing the cardinality can be accessed and constrained using method getCard() that returns an IntegerVariable object.

Example:

```
SetVariable svar1 = makeSetVar("svar1", -10, 10);
SetVariable svar2 = makeSetVar("svar2", 0, 10000, Options.V_BOUND, Options.V_NO_DECISION
    );
```

Set variables are illustrated on the ternary Steiner problem.

5.4 Task variables

TaskVariable is an object-variable composed of three IntegerVariable: a starting time integer variable *start*, an ending time integer variable *end* and a duration integer variable *duration*. To create a TaskVariable, one can creates the *start*, *end* and *duration* before, or indicates the earliest starting time (*int*), the latest completion time (*int*) and the duration(int or IntegerVariable).

constructors:

Choco method	return type
makeTaskVar(String name, IntegerVariable start, IntegerVariable end, IntegerVariable duration, String... options)	TaskVariable
makeTaskVar(String name, IntegerVariable start, IntegerVariable duration, String... options)	TaskVariable
makeTaskVar(String name, int binf, int bsup, IntegerVariable duration, String... options)	TaskVariable
makeTaskVar(String name, int binf, int bsup, int duration, String... options)	TaskVariable
makeTaskVar(String name, int bsup, IntegerVariable duration, String... options)	TaskVariable
makeTaskVar(String name, int bsup, int duration, String... options)	TaskVariable
makeTaskVarArray(String prefix, IntegerVariable[] starts, IntegerVariable[] ends, IntegerVariable[] durations, String... options)	TaskVariable[]
makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[] durations, String... options)	TaskVariable[]
makeTaskVarArray(String name, int binf, int bsup, int[] durations, String... options)	TaskVariable[][]
makeTaskVarArray(String name, int binf, int bsup, IntegerVariable[][] durations, String... options)	TaskVariable[][]
makeTaskVarArray(String name, int binf, int bsup, int[][] durations, String... options)	TaskVariable[][]

options:

Options are for the three `IntegerVariable`. See `IntegerVariable` for more details about options.

Example:

```
TaskVariable tvar1 = makeTaskVar("tvar1", 0, 123, 18, Options.V_ENUM);
IntegerVariable start = makeIntVar("start", 0, 30);
IntegerVariable end = makeIntVar("end", 10, 60);
IntegerVariable duration = makeIntVar("duration", 7, 13);
TaskVariable tvar2 = makeTaskVar("tvar2", start, end, duration);
```

Chapter 6

Operators (Model)

This section lists and details the operators that can be used within a Choco Model to combine variables in expressions.

6.1 abs (operator)

Returns an expression variable that represents the absolute value of the argument ($|n|$).

- **API** : `abs(IntegerExpressionVariable n)`

- **return type** : IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : unknown

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5, Options.V_ENUM);
IntegerVariable y = makeIntVar("y", -5, 5, Options.V_ENUM);
m.addConstraint(eq(abs(x), y));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.2 cos (operator)

Returns an expression variable corresponding to the cosinus value of the argument ($cos(x)$).

- **API** : `cos(RealExpressionVariable exp)`

- **return type** : RealExpressionVariable

- **options** : n/a

- **favorite domain** : real

Example:

```
Model m = new CPModel();
RealVariable x = makeRealVar("x", -Math.PI/2, Math.PI);
m.addConstraint(eq(cos(x), 2/3));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.3 distEq (operator)

To complete

6.4 distGt (operator)

To complete

6.5 distLt (operator)

To complete

6.6 distNeq (operator)

To complete

6.7 div (operator)

Returns an expression variable that represents the *integer quotient* of the division of the first argument variable by the second one (n_1/n_2).

- **API** :
  - div(IntegerExpressionVariable n1, IntegerExpressionVariable n2)
  - div(IntegerExpressionVariable n1, int n2)
  - div(int n1, IntegerExpressionVariable n2)
- **return type** : IntegerExpressionVariable
- **options** : n/a
- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
IntegerVariable w = makeIntVar("w", 22, 44);
IntegerVariable z = makeIntVar("z", 12, 21);
m.addConstraint(eq(z, div(w, x)));
s.read(m);
s.solve();
```

6.8 ifThenElse (operator)

ifThenElse(c, v_1, v_2) states that if the constraint c is satisfied, it returns the second parameter v_1, otherwise it returns the third one v_2.

- **API** : ifThenElse(Constraint c, IntegerExpressionVariable v1, IntegerExpressionVariable v2)
- **return type** : IntegerExpressionVariable
- **options** : n/a
- **favorite domain** : unknown

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 0, 10);
m.addConstraint(eq(y, ifThenElse(gt(x,2), mult(x,x), x)));
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

6.9 max (operator)

Returns an expression variable equals to the greater value of the argument $(max(x_1, x_2, ..., x_n))$.

- **API** :

  - max(IntegerExpressionVariable x1, IntegerExpressionVariable x2)

  - max(int x1, IntegerExpressionVariable x2)

  - max(IntegerExpressionVariable x1, int x2)

  - max(IntegerExpressionVariable[] x)

- **return type**: IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
m.setDefaultExpressionDecomposition(true);
IntegerVariable[] v = makeIntVarArray("v", 3, -3, 3);
IntegerVariable maxv = makeIntVar("max", -3, 3);
Constraint c = eq(maxv, max(v));
m.addConstraint(c);
Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

6.10 min (operator)

Returns an expression variable equals to the smaller value of the argument $(min(x_1, x_2, ..., x_n))$.

- **API** :

  - min(IntegerExpressionVariable x1, IntegerExpressionVariable x2)

  - min(int x1, IntegerExpressionVariable x2)

  - min(IntegerExpressionVariable x1, int x2)

  - min(IntegerExpressionVariable[] x)

- **return type**: IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        m.setDefaultExpressionDecomposition(true);
        IntegerVariable[] v = makeIntVarArray("v", 3, -3, 3);
        IntegerVariable minv = makeIntVar("min", -3, 3);
        Constraint c = eq(minv, min(v));
        m.addConstraint(c);
        Solver s = new CPSolver();
        s.read(m);
        s.solveAll();
```

6.11   minus (operator)

Returns an expression variable that corresponding to the difference between the two arguments $(x - y)$.

- **API** :

  – minus(IntegerExpressionVariable x, IntegerExpressionVariable y)

  – minus(IntegerExpressionVariable x, int y)

  – minus(int x, IntegerExpressionVariable y)

  – minus(RealExpressionVariable x, RealExpressionVariable y)

  – minus(RealExpressionVariable x, double y)

  – minus(double x, RealExpressionVariable y)

- **return type** :

  – IntegerExpressionVariable, if parameters are IntegerExpressionVariable

  – RealExpressionVariable, if parameters are RealExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable a = makeIntVar("a", 0, 4);
        m.addConstraint(eq(minus(a, 1), 2));
        s.read(m);
        s.solve();
```

6.12   mod (operator)

Returns an expression variable that represents the integer remainder of the division of the first argument variable by the second one $(x_1 \% x_2)$.

- **API**:

  – mod(IntegerExpressionVariable x1, IntegerExpressionVariable x2)

  – mod(int x1, IntegerExpressionVariable x2)

  – mod(IntegerExpressionVariable x1, int x2)

- **return type** : IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
IntegerVariable w = makeIntVar("w", 22, 44);
m.addConstraint(eq(1, mod(w, x)));
s.read(m);
s.solve();
```

6.13   mult (operator)

Returns an expression variable that corresponding to the product of variables in argument $(x * y)$.

- **API** :
    - mult(IntegerExpressionVariable x, IntegerExpressionVariable y)
    - mult(IntegerExpressionVariable x, int y)
    - mult(int x, IntegerExpressionVariable y)
    - mult(RealExpressionVariable x, RealExpressionVariable y)
    - mult(RealExpressionVariable x, double y)
    - mult(double x, RealExpressionVariable y)

- **return type** :
    - IntegerExpressionVariable, if parameters are IntegerExpressionVariable
    - RealExpressionVariable, if parameters are RealExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example

```
CPModel m = new CPModel();
IntegerVariable x = makeIntVar("x", -10, 10);
IntegerVariable z = makeIntVar("z", -10, 10);
IntegerVariable w = makeIntVar("w", -10, 10);
m.addVariables(x, z, w);
CPSolver s = new CPSolver();
// x >= z * w
Constraint exp = geq(x, mult(z, w));
m.setDefaultExpressionDecomposition(true);
m.addConstraint(exp);
s.read(m);
s.solveAll();
```

6.14   neg (operator)

Returns an expression variable that is the opposite of the expression integer variable in argument $(-x)$.

- **API** : neg(IntegerExpressionVariable x)

- **return type** : IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", -10, 10);
IntegerVariable w = makeIntVar("w", -10, 10);
// -x = w - 20
m.addConstraint(eq(neg(x), minus(w, 20)));
s.read(m);
s.solve();
```

6.15   plus (operator)

Returns an expression variable that corresponding to the sum of the two arguments $(x + y)$.

- **API** :

  − plus(IntegerExpressionVariable x, IntegerExpressionVariable y)

  − plus(IntegerExpressionVariable x, int y)

  − plus(int x, IntegerExpressionVariable y)

  − plus(RealExpressionVariable x, RealExpressionVariable y)

  − plus(RealExpressionVariable x, double y)

  − plus(double x, RealExpressionVariable y)

- **return type** :

  − IntegerExpressionVariable, if parameters are IntegerExpressionVariable

  − RealExpressionVariable, if parameters are RealExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable a = makeIntVar("a", 0, 4);
// a + 1 = 2
m.addConstraint(eq(plus(a, 1), 2));
s.read(m);
s.solve();
```

6.16   power (operator)

Returns an expression variable that represents the first argument raised to the power of the second argument (x^y).

- **API** :

  − power(IntegerExpressionVariable x, IntegerExpressionVariable y)

  − power(int x, IntegerExpressionVariable y)

  − power(IntegerExpressionVariable x, int y)

  − power(RealExpressionVariable x, int y)

- **return type**:
    - IntegerExpressionVariable, if parameters are IntegerExpressionVariable
    - RealExpressionVariable, if parameters are RealExpressionVariable

- **option** : *n/a*

- **favorite domain** : *to complete*

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 0, 10);
IntegerVariable y = makeIntVar("y", 2, 4);
IntegerVariable z = makeIntVar("z", 28, 80);
m.addConstraint(eq(z, power(x, y)));
s.read(m);
s.solve();
```

6.17 scalar (operator)

Return an integer expression that corresponds to the scalar product of coefficients array and variables array ($c_1 * x_1 + c_2 * x_2 + ... + c_n * x_n$).

- **API** :
    - scalar(int[] c, IntegerVariable[] x)
    - scalar(IntegerVariable[] x, int[] c)

- **return type** : IntegerExpressionVariable

- **options** : *n/a*

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable[] vars = makeIntVarArray("C", 9, 1, 10);
int[] coefficients = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9};
m.addConstraint(eq(165, scalar(coefficients, vars)));

s.read(m);
s.solve();
System.out.print("165 = (" + coefficients[0] + "*" + s.getVar(vars[0]).getVal()+")");
for (int i = 1; i < vars.length; i++) {
    System.out.print(" + (" + coefficients[i] + "*" + s.getVar(vars[i]).getVal()+")");
}
System.out.println();
```

6.18 sin (operator)

Returns a real variable that corresponding to the sinus value of the argument ($sin(x)$).

- **API** : sin(RealExpressionVariable exp)

- **return type** : RealExpressionVariable

- **options** : n/a

- **favorite domain** : real

Example:

```
Model m = new CPModel();
RealVariable x = makeRealVar("x", 0, Math.PI);
m.addConstraint(eq(sin(x), 1));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

6.19   sum (operator)

Return an integer expression that corresponds to the sum of the variables given in argument ($x_1 + x_2 + ... + x_n$).

- **API**: `sum(IntegerVariable... lv)`

- **return type** : IntegerExpressionVariable

- **options** : n/a

- **favorite domain** : *to complete*

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();

IntegerVariable[] vars = makeIntVarArray("C", 10, 1, 10);
m.addConstraint(eq(99, sum(vars)));

s.read(m);
s.solve();
if(s.isFeasible()){
    System.out.print("99 = " + s.getVar(vars[0]).getVal());
    for (int i = 1; i < vars.length; i++) {
        System.out.print(" + "+s.getVar(vars[i]).getVal());
    }
    System.out.println();
}
```

Chapter 7

Constraints (Model)

This section lists and details the constraints currently available in Choco.

7.1   abs (constraint)

$\mathtt{abs}(x, y)$ states that x is the absolute value of y:

$$x = |y|$$

- **API** : `abs(IntegerVariable x, IntegerVariable y)`
- **return type** : `Constraint`
- **options** : n/a
- **favorite domain** : enumerated

Example:

```
Model m = new CPModel();
IntegerVariable x = makeIntVar("x", 1, 5, Options.V_ENUM);
IntegerVariable y = makeIntVar("y", -5, 5, Options.V_ENUM);
m.addConstraint(abs(x, y));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

7.2   allDifferent (constraint)

$\mathtt{allDifferent}(x_1, \ldots, x_n)$ states that the arguments have pairwise distinct values:

$$x_i \neq x_j, \quad \forall\, i \neq j$$

This constraint is useful for some matching problems. Notice that the filtering algorithm used will depend on the nature (enumerated or bounded) of the variables: when *enumerated*, the constraint refers to the alldifferent of [Régin, 1994]; when *bounded*, a dedicated algorithm for bound propagation is used [López-Ortiz et al., 2003].

- **API** :

  - `allDifferent(IntegerVariable... x)`

  - `allDifferent(String options, IntegerVariable... x)`

- **return type** : `Constraint`

- **options** :

  - *no option* clever choice made on domains of given variables

  - `Options.C_ALLDIFFERENT_AC` for [Régin, 1994] implementation of arc consistency

  - `Options.C_ALLDIFFERENT_BC` for [López-Ortiz et al., 2003] implementation of bound consistency

  - `Options.C_ALLDIFFERENT_CLIQUE` for propagating the clique of differences

- **favorite domain** : depending of options.

- **references** :

  - [Régin, 1994]: *A filtering algorithm for constraints of difference in CSPs*

  - [López-Ortiz et al., 2003]: *A fast and simple algorithm for bounds consistency of the alldifferent constraint*

  - global constraint catalog: `alldifferent`

Example:

```
int n = 8;
CPModel m = new CPModel();
IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] diag1 = new IntegerVariable[n];
IntegerVariable[] diag2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n);
    diag1[i] = makeIntVar("D1" + i, 1, 2 * n);
    diag2[i] = makeIntVar("D2" + i, -n + 1, n);
}
m.addConstraint(allDifferent(queens));
for (int i = 0; i < n; i++) {
    m.addConstraint(eq(diag1[i], plus(queens[i], i)));
    m.addConstraint(eq(diag2[i], minus(queens[i], i)));
}
m.addConstraint(Options.C_ALLDIFFERENT_CLIQUE, allDifferent(diag1));
m.addConstraint(Options.C_ALLDIFFERENT_CLIQUE, allDifferent(diag2));
// diagonal constraints
CPSolver s = new CPSolver();
s.read(m);
long tps = System.currentTimeMillis();
s.solveAll();
System.out.println("tps nreines1 " + (System.currentTimeMillis() - tps) + " nbNode " + s
    .
        getNodeCount());
```

7.3 among (constraint)

7.3.1 among values

among(x, V) states that the variable x takes its value in V:

$$x \subseteq V$$

- **API**: `among(IntegerVariable x, int[] v)`

- **return type**: `Constraint`

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable var = makeIntVar("v1", 0, 100, Options.V_BOUND);
int[] values = new int[]{0, 25, 50, 75, 100};
m.addConstraint(among(var, values));
s.read(m);
s.solve();
```

7.3.2 among values with counter

among(z, x, V) states that z is the number of variables x that take their value in V:

$$z = |\{i | x_i \in V\}|$$

- **API**: `among(IntegerVariable z, IntegerVariable[] x, int[] v)`

- **return type**: `Constraint`

- [Bessière et al., 2005a]: *Among, common and disjoint Constraints*

- [Bessière et al., 2006]: *Among, common and disjoint Constraints*

- global constraint catalog: among

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable nvar = makeIntVar("v1", 1, 2);
IntegerVariable[] vars = Choco.makeIntVarArray("var", 10, 0, 10);
int[] values = new int[]{2, 3, 5};
m.addConstraint(among(nvar, vars, values));
s.read(m);
s.solve();
```

7.3.3   among values with counter and set

among(z, x, s) states that z is the number of variables x that take their value in the set variable s:

$$z = |\{i | x_i \in s\}|$$

- **API**: `among(IntegerVariable z, IntegerVariable[] x, SetVariable s)`

- **return type**: `Constraint`

- [Bessière et al., 2005a]: *Among, common and disjoint Constraints*

- [Bessière et al., 2006]: *Among, common and disjoint Constraints*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable nvar = makeIntVar("v1", 1, 2);
IntegerVariable[] vars = Choco.makeIntVarArray("var", 10, 0, 10);
SetVariable values = Choco.makeSetVar("s", 2, 6);
m.addConstraint(among(nvar, vars, values));
s.read(m);
s.solve();
```

7.4   and (constraint)

and($c_1, \ldots, c_n$) states that every constraints in arguments are satisfied:

$$c_1 \wedge c_2 \wedge \ldots \wedge c_n$$

and($b_1, \ldots, b_n$) states that every 0-1 variables in arguments are true (equal to 1):

$$b_1 = 1 \wedge b_2 = 1 \wedge \ldots \wedge b_n = 1$$

- **API** :

  - `and(Constraint... c)`
  - `and(IntegerVariable... b)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

- **references** :
  global constraint catalog: `and`

Examples:

- example1:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable v1 = makeIntVar("v1", 0, 1);
        IntegerVariable v2 = makeIntVar("v2", 0, 1);
        m.addConstraint(and(eq(v1, 1), eq(v2, 1)));
        s.read(m);
        s.solve();
```

- example2

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable[] vars = makeBooleanVarArray("b", 10);
        m.addConstraint(and(vars));
        s.read(m);
        s.solve();
```

7.5 atMostNValue (constraint)

atMostNValue(x, z) states that the number of different values occurring in the array of variables x is at most z:

$$z \geq |\{x_1, \ldots, x_n\}|$$

- **API** : `atMostNValue(IntegerVariable[] x, IntegerVariable z)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

- **references** :

  - [Bessière et al., 2005b] *Filtering algorithms for the NValue constraint*

  - global constraint catalog: `atmost_nvalue`

Example:

```
        Model m = new CPModel();
        CPSolver s = new CPSolver();
        IntegerVariable v1 = makeIntVar("v1", 1, 1);
        IntegerVariable v2 = makeIntVar("v2", 2, 2);
        IntegerVariable v3 = makeIntVar("v3", 3, 3);
        IntegerVariable v4 = makeIntVar("v4", 3, 4);
        IntegerVariable n = makeIntVar("n", 3, 3);
        Constraint c = atMostNValue(new IntegerVariable[]{v1, v2, v3, v4}, n);
        m.addConstraint(c);
        s.read(m);
        s.solve();
```

7.6 boolChanneling (constraint)

boolChanneling(b, x, v) states that b is true if and only if x is equal to v:

$$ b \quad \Longleftrightarrow \quad (x = v) $$

It acts as an observer of value v. Imagine a bin packing problem where variable x tells you on which a given bin object is placed. By stating the boolean channeling, b is true if and only if the object is placed on bin v, the knapsack constraint for bin v can then be easily stated as a scalar of the boolean variables.

- **API** : `boolChanneling(IntegerVariable b, IntegerVariable x, int v)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : enumerated for x

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable bool = makeIntVar("bool", 0, 1);
IntegerVariable x = makeIntVar("x", 0, 5);
m.addConstraint(boolChanneling(bool, x, 4));
s.read(m);
s.solveAll();
```

7.7 clause (constraint)

clause(px, nx) enforces :

$$ (px_1 \lor px_2 \lor \cdots \lor px_n) \lor (\neg nx_1 \lor \neg nx_2 \lor \cdots \lor \neg nx_n) $$

- **API** :
  - `clause(IntegerVariable... x)`
  - `clause(String options, IntegerVariable... x)`

- **return type** : `Constraint`

- **options** :
  - *no option* default filtering
  - `Options.C_CLAUSES_ENTAIL` ensures quick entailment tests

- **favorite domain** : n/a.

- **references** :global constraint catalog: `clause_or`

Example:

```
        CPModel mod = new CPModel();
        CPSolver s = new CPSolver();
        IntegerVariable[] vars = makeBooleanVarArray("b", 8);

        IntegerVariable[] plits1 = new IntegerVariable[]{vars[0], vars[3], vars[4]};
        IntegerVariable[] nlits1 = new IntegerVariable[]{vars[1], vars[2], vars[6]};
        mod.addConstraint(clause(plits1, nlits1));

        IntegerVariable[] plits2 = new IntegerVariable[]{vars[5], vars[3]};
        IntegerVariable[] nlits2 = new IntegerVariable[]{vars[1], vars[4], vars[7]};
        mod.addConstraint(clause(plits2, nlits2));

        s.read(mod);
        s.solveAll();
```

7.8   costRegular (constraint)

$costRegular(x, cx, d, c)$ ensures that the sequence of variables values of x will follow a pattern defined by a DFA d and that this sequence has a cost defined by

$$\sum_{i=1}^{n} c_i^{x_i} = cx$$

.

- **API** :
    - costRegular(IntegerVariable[] vars, IntegerVariable cvar, FiniteAutomaton auto, int[][] costs)
    - costRegular(IntegerVariable[] vars, IntegerVariable cvar, FiniteAutomaton auto, int[][][] costs)
    - costRegular(IntegerVariable[] vars, IntegerVariable cvar, DirectedMultigraph<Node, Arc> graph, Node source)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a

- **references** : [Demassey et al., 2006]: *A `Cost-Regular` based hybrid column generation approach*

Example:

```
        int n = 10;
        IntegerVariable[] vars = makeIntVarArray("x",n,0,2, Options.V_ENUM);
        IntegerVariable z = makeIntVar("z",3,4, Options.V_BOUND);

        FiniteAutomaton auto = new FiniteAutomaton();
        int start = auto.addState();
        int end = auto.addState();
        auto.setInitialState(start);
        auto.setFinal(start);
        auto.setFinal(end);
```

```
      auto.addTransition(start,start, 0,1);
      auto.addTransition(start,end,2);

      auto.addTransition(end,start,2);
      auto.addTransition(end,start, 0,1);

      int[][][] costs = new int[n][3][2];
      for (int i = 0 ; i < costs.length ; i++)
      {
          costs[i][0][1] = 1;
          costs[i][1][1] = 1;
      }

      CPModel m = new CPModel();
      m.addConstraint(costRegular(vars,z,auto,costs));

      CPSolver s= new CPSolver();
      s.read(m);
      s.solveAll();
```

7.9 cumulative (constraint)

`cumulative(start,duration,height,capacity)` states that a set of tasks (defined by their starting times, finishing dates, durations and heights (or consumptions)) are executed on a cumulative resource of limited capacity. That is, the total height of the tasks which are executed at any time t does not exceed the capacity of the resource:

$$\sum_{\{i \ | \ \texttt{start}[i] \leq t < \texttt{start}[i] + \texttt{duration}[i]\}} \texttt{height}[i] \leq \texttt{capacity}, \quad (\forall \text{ time } t)$$

The notion of task does not exist yet in Choco. The `cumulative` takes therefore as input, several arrays of integer variables (of same size n) denoting the starting, duration, and height of each task. When the array of finishing times is also specified, the constraint ensures that `start[i] + duration[i]` `= end[i]` for all task i. As usual, a task is executed in the interval `[start,end-1]`.

A tutorial on the use of this constraint is available here

- **API** :

    - `cumulative(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration,` `IntegerVariable[] height, IntegerVariable capa, String... options)`
    - `cumulative(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration,` `int[] height, int capa, String... options)`
    - `cumulative(IntegerVariable[] start, IntegerVariable[] duration, IntegerVariable[] height` `, IntegerVariable capa, String... options)`

- **return type** : `Constraint`

- **options** :

    - *no option*
    - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
    - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
    - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier

- SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)

- **favorite domain** : n/a

- **references** :

  - [Beldiceanu and Carlsson, 2002] *A new multi-resource cumulatives constraint with negative heights*

  - global constraint catalog: cumulative

Example:

```java
CPModel m = new CPModel();
// data
int n = 11 + 3; //number of tasks (include the three fake tasks)
int[] heights_data = new int[]{2, 1, 4, 2, 3, 1, 5, 6, 2, 1, 3, 1, 1, 2};
int[] durations_data = new int[]{1, 1, 1, 2, 1, 3, 1, 1, 3, 4, 2, 3, 1, 1};
// variables
IntegerVariable capa = constant(7);
IntegerVariable[] starts = makeIntVarArray("start", n, 0, 5, Options.V_BOUND);
IntegerVariable[] ends = makeIntVarArray("end", n, 0, 6, Options.V_BOUND);
IntegerVariable[] duration = new IntegerVariable[n];
IntegerVariable[] height = new IntegerVariable[n];
for (int i = 0; i < height.length; i++) {
    duration[i] = constant(durations_data[i]);
    height[i] = makeIntVar("height " + i, new int[]{0, heights_data[i]});
}
TaskVariable[] tasks = Choco.makeTaskVarArray("Task", starts, ends, duration);

IntegerVariable[] bool = makeIntVarArray("taskIn?", n, 0, 1);
IntegerVariable obj = makeIntVar("obj", 0, n, Options.V_BOUND, Options.V_OBJECTIVE);
//post the cumulative
m.addConstraint(cumulative("cumulative", tasks, height, constant(0), capa,
        SettingType.TASK_INTERVAL.getOptionName()));
//post the channeling to know if the task is scheduled or not
for (int i = 0; i < n; i++) {
    m.addConstraint(boolChanneling(bool[i], height[i], heights_data[i]));
}
//state the objective function
m.addConstraint(eq(sum(bool), obj));
CPSolver s = new CPSolver();
s.read(m);
//set the fake tasks to establish the profile capacity of the ressource
try {
    s.getVar(starts[0]).setVal(1);
    s.getVar(ends[0]).setVal(2);
    s.getVar(height[0]).setVal(2);
    s.getVar(starts[1]).setVal(2);
    s.getVar(ends[1]).setVal(3);
    s.getVar(height[1]).setVal(1);
    s.getVar(starts[2]).setVal(3);
    s.getVar(ends[2]).setVal(4);
    s.getVar(height[2]).setVal(4);
} catch (ContradictionException e) {
    System.out.println("error, no contradiction expected at this stage");
}
// maximize the number of tasks placed in this profile
s.maximize(s.getVar(obj), false);
System.out.println("Objective : " + (s.getVar(obj).getVal() - 3));
for (int i = 3; i < starts.length; i++) {
    if (s.getVar(height[i]).getVal() != 0)
```

```
            System.out.println("[" + s.getVar(starts[i]).getVal() + " - "
                    + (s.getVar(ends[i]).getVal() - 1) + "]:"
                    + s.getVar(height[i]).getVal());
    }
```

7.10   cumulativeMax (constraint)

Specific case of Cumulative, where the **capacity** is infinite.

- **API** :
  - cumulativeMax(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable capacity, String... options)
  - cumulativeMax(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable capacity, String... options)
  - cumulativeMax(TaskVariable[] tasks, int[] heights, int capacity, String... options)

- **return type** : Constraint

- **options** :
  - *no option*
  - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
  - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
  - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)

- **favorite domain** : *n/a*

7.11   cumulativeMin (constraint)

Specific case of Cumulative, where the **consumption** is equal to 0.

- **API** :
  - cumulativeMin(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, String... options)
  - cumulativeMin(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, String... options)
  - cumulativeMin(TaskVariable[] tasks, int[] heights, int consumption, String... options)

- **return type** : Constraint

- **options** :
  - *no option*
  - SettingType.TASK_INTERVAL.getOptionName() for fast task intervals
  - SettingType.SLOW_TASK_INTERVAL.getOptionName() for slow task intervals
  - SettingType.VILIM_CEF_ALGO.getOptionName() for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - SettingType.VHM_CEF_ALGO_N2K.getOptionName() for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)

- **favorite domain** : *n/a*

7.12 disjoint (constraint)

7.12.1 disjoint values

disjoint(x, V) states that the variable x takes its value out of V:

$$x \cup V = \emptyset$$

- **API**: `disjoint(IntegerVariable x, int[] v)`

- **return type**: `Constraint`

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable var = makeIntVar("v1", 0, 100, Options.V_BOUND);
int[] values = new int[]{10,20,30,40,50,60,70,80,90};
m.addConstraint(disjoint(var, values));
s.read(m);
s.solve();
```

7.12.2 disjoint tasks

disjoint$(t1, t2)$ states that each tasks t_1 should not overlap anu tasks t_2:

$$x \cup V = \emptyset$$

CHOCO only provides a decomposition with reified precedences because the coloured cumulative is not available.

- **API**: `disjoint(TaskVariable[] t1, TaskVariable[] t2)`

- **return type**: `Constraint[]`

- **favorite domain** : n/a.

- **references** :

    – global constraint catalog: disjoint_tasks

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
TaskVariable[] tasks1 = Choco.makeTaskVarArray("Task1", 0, 10, new int[]{2,5});
TaskVariable[] tasks2 = Choco.makeTaskVarArray("Task2", 0, 10, new int[]{3,4});
m.addConstraints(disjoint(tasks1, tasks2));
s.read(m);
s.solve();
```

7.13 disjunctive (constraint)

disjunctive(`start`,`duration`) states that a set of tasks (defined by their starting times and durations) are executed on a ddisjunctive resource, i.e. they do not overlap in time:

$$|\{i \mid \texttt{start}[i] \leq t < \texttt{start}[i] + \texttt{duration}[i]\}| \leq 1, \quad (\forall \text{ time } t)$$

The notion of task does not exist yet in Choco. The `disjunctive` takes therefore as input arrays of integer variables (of same size n) denoting the starting and duration of each task. When the array of finishing times is also specified, the constraint ensures that `start[i] + duration[i] = end[i]` for all task i. As usual, a task is executed in the interval `[start,end-1]`.

- **API** :

  - disjunctive(IntegerVariable[] start, int[] duration, String...options)

  - disjunctive(IntegerVariable[] start, IntegerVariable[] duration, String... options)

  - disjunctive(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration, String... options)

  - disjunctive(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration, IntegerVariable uppBound, String... options)

- **return type** : Constraint

- **options** :

  - *no option*

  - SettingType.OVERLOAD_CHECKING.getOptionName() overload checking rule (O(n*log(n)), Vilim), also known as task interval

  - SettingType.NF_NL.getOptionName() NotFirst/NotLast rule (O(n*log(n)), Vilim) (recommended)

  - SettingType.DETECTABLE_PRECEDENCE.getOptionName() Detectable Precedence rule (O(n*log(n)), Vilim)

  - SettingType.EDGE_FINDING_D.getOptionName() disjunctive Edge Finding rule (O(n*log(n)), Vilim) (recommended)

  - SettingType.DEFAULT_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint

  - SettingType.VILIM_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint

  - SettingType.SINGLE_RULE_FILTERING.getOptionName() use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint. A single filtering rule (debug only).

- **favorite domain** : *n/a*

- **references** :
  global constraint catalog: disjunctive

Example: //TODO: complete

7.14 distanceEQ (constraint)

distanceEQ(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is equal to the distance between x_1 and x_2:
$$x_3 + c = |x_1 - x_2|$$

- **API** :
    - distanceEQ(IntegerVariable x1, IntegerVariable x2, int x3)
    - distanceEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)
    - distanceEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)

- **return type**: Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: all_min_dist (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceEQ(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

7.15 distanceGT (constraint)

distanceGT(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is strictly greater than the distance between x_1 and x_2:
$$x_3 + c > |x_1 - x_2|$$

- **API** :
    - distanceGT(IntegerVariable x1, IntegerVariable x2, int x3)
    - distanceGT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)
    - distanceGT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)

- **return type**: Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: `all_min_dist` (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceGT(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

7.16 distanceLT (constraint)

`distanceLT`(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is strictly smaller than the distance between x_1 and x_2:

$$x_3 + c < |x_1 - x_2|$$

- **API** :

  - `distanceLT(IntegerVariable x1, IntegerVariable x2, int x3)`

  - `distanceLT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`

  - `distanceLT(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`

- **return type**: `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
IntegerVariable v2 = makeIntVar("v2", 0, 5);
m.addConstraint(distanceLT(v0, v1, v2, 0));
s.read(m);
s.solveAll();
```

7.17 distanceNEQ (constraint)

`distanceNEQ`(x_1, x_2, x_3, c) states that x_3 plus an offset c (by default $c = 0$) is not equal to the distance between x_1 and x_2:

$$x_3 + c \neq |x_1 - x_2|$$

- **API** :

     – `distanceNEQ(IntegerVariable x1, IntegerVariable x2, int x3)`

     – `distanceNEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`

     – `distanceNEQ(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3, int c)`

- **return type**: `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: all_min_dist (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v0 = makeIntVar("v0", 0, 5);
IntegerVariable v1 = makeIntVar("v1", 0, 5);
m.addConstraint(distanceNEQ(v0, v1, 0));
s.read(m);
s.solveAll();
```

7.18   domainConstraint (constraint)

`domainConstraint`$(bVar, values)$ states that $values[i]$ is equal to 1 if and only if $bVar$ is equal to i (0 otherwise):

$$values[i] = 1 \quad \Longleftrightarrow \quad (bVar = i)$$

It makes the link between a domain variable $bVar$ and those 0-1 variables that are associated with each potential value of $bVar$: the 0-1 variable associated with the value that is taken by variable $bVar$ is equal to 1, while the remaining 0-1 variables are all equal to 0.

- **API** : `domainConstraint(IntegerVariable bVar, IntegerVariable[] values)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : enumerated for $bVar$

- **references** :
  global constraint catalog: domainConstraint

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable bVar = makeIntVar("bVar", 0, 10);
IntegerVariable[] values = makeBooleanVarArray("value", 10);
m.addConstraint(domainConstraint(bVar, values));
s.read(m);
s.solveAll();
```

7.19   element (constraint)

See nth.

7.20   endsAfter (constraint)

endsAfter(t, c) states that the task variable t ends after c:

$$e_t \geq c$$

- **API** : endsAfter(TaskVariable t, int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.21   endsAfterBegin (constraint)

endsAfterBegin(t_1, t_2, c) states that the task variable t_1 ends after the beginning of t_2, minus c:

$$e_{t_1} \geq s_{t_2} - c$$

- **API** : endsAfterBegin(TaskVariable t1, TaskVariable t2, int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.22   endsAfterEnd (constraint)

endsAfterEnd(t_1, t_2, c) states that the task variable t_1 ends after the end of t_2, minus c:

$$e_{t_1} \geq e_{t_2} - c$$

- **API** : endsAfterEnd(TaskVariable t1, TaskVariable t2, int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.23 endsBefore (constraint)

endsBefore(t, c) states that the task variable t ends before c:

$$e_t \leq c$$

- **API** :endsBefore(`final TaskVariable t, final int c`)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.24 endsBeforeBegin (constraint)

endsBeforeBegin(t_1, t_2, c) states that the task variable t_1 ends before the beginning of t_2, minus c:

$$e_{t_1} \leq s_{t_2} - c$$

- **API** :endsBeforeBegin(`TaskVariable t1, TaskVariable t2, int c`)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.25 endsBeforeEnd (constraint)

endsBeforeEnd(t_1, t_2, c) states that the task variable t_1 ends before the end of t_2, minus c:

$$e_{t_1} \leq e_{t_2} - c$$

- **API** :endsBeforeEnd(`TaskVariable t1, TaskVariable t2, int c`)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.26 endsBetween (constraint)

endsBetween(t, c_1, c_2) states that the task variable t ends between c_1 and c_2:

$$c_1 \leq e_t \leq c_2$$

- **API** : endsBetween(TaskVariable t, int min, int max)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a.

Examples: *to complete*

7.27 eq (constraint)

eq(x, y) states that the two arguments are equal:

$$x = y$$

- **API** :

    - eq(IntegerExpressionVariable x, IntegerExpressionVariable y)

    - eq(IntegerExpressionVariable x, int y)

    - eq(int x, IntegerExpressionVariable y)

    - eq(SetVariable x, SetVariable y)

    - eq(RealExpressionVariable x, RealExpressionVariable y)

    - eq(RealExpressionVariable x, double y)

    - eq(double x, RealExpressionVariable y)

    - eq(IntegerVariable x, RealVariable y)

    - eq(RealVariable x, IntegerVariable y)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*.

- **references** :
  global constraint catalog: eq (on domain variables) and eq_set (on set variables).

Examples:

- example1:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        int c = 1;
        IntegerVariable v = makeIntVar("v", 0, 2);
        m.addConstraint(eq(v, c));
        s.read(m);
        s.solve();
```

- example2

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable v1 = makeIntVar("v1", 0, 2);
        IntegerVariable v2 = makeIntVar("v2", 0, 2);
        IntegerExpressionVariable w1 = plus(v1, 1);
        IntegerExpressionVariable w2 = minus(v2, 1);
        m.addConstraint(eq(w1, w2));
        s.read(m);
        s.solve();
```

7.28 eqCard (constraint)

eqCard(s, x) states that the cardinality of set s is equal to x:

$$|s| = x$$

- **API** :

    - eqCard(SetVariable s, IntegerVariable x)

    - eqCard(SetVariable s, int x)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        SetVariable set = makeSetVar("s", 1, 5);
        IntegerVariable card = makeIntVar("card", 2, 3);
        m.addConstraint(member(set, 3));
        m.addConstraint(eqCard(set, card));
        s.read(m);
        s.solve();
```

7.29 equation (constraint)

equation(x, c, z) states a linear equation:

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n = z$$

It enforces GAC using regular to state a *knapsack* constraint.

- **API** :

    – equation(IntegerVariable[] x, int[] c, int z)

- **return type** : `Constraint`

- **options** : *n/a*

- **favorite domain** : *to complete*

Example:

```
CPModel m = new CPModel();
CPSolver s = new CPSolver();
int n = 10;
IntegerVariable[] bvars = makeIntVarArray("b", n, 0, 10, Options.V_ENUM);
int[] coefs = new int[n];

int charge = 10;
Random rand = new Random();
for (int i = 0; i < coefs.length; i++) {
    coefs[i] = rand.nextInt(10);
}
Constraint knapsack = equation(bvars, coefs, charge);
m.addConstraint(knapsack);
s.read(m);
s.solveAll();
```

7.30 exactly (constraint)

To complete

7.31 FALSE (constraint)

FALSE always returns *false*.

7.32 feasPairAC (constraint)

feasPairAC($x, y, feasTuples$) states an extensional binary constraint on (x, y) defined by the table $feasTuples$ of compatible pairs of values, and then enforces arc consistency. Two APIs are available to define the compatible pairs:

- if $feasTuples$ is encoded as a list of pairs `List<int[2]>`, then:

$$\exists \text{ tuple } i \mid \quad (x, y) = feasTuples[i]$$

- if $feasTuples$ is encoded as a boolean matrix `boolean[][]`, let $\underline{x}$ and $\underline{y}$ be the initial minimum values of x and y, then:

$$\exists (u, v) \mid \quad (x, y) = (u + \underline{x}, v + \underline{y}) \ \wedge \ feasTuples[u][v]$$

The two APIs are duplicated to allow definition of options.

- **API** :

  - feasPairAC(IntegerVariable x, IntegerVariable y, List<int[]> feasTuples)
  - feasPairAC(String options, IntegerVariable x, IntegerVariable y, List<int[]> feasTuples)
  - feasPairAC(IntegerVariable x, IntegerVariable y, boolean[][] feasTuples)
  - feasPairAC(String options, IntegerVariable x, IntegerVariable y, boolean[][] feasTuples)

- **return type** : Constraint

- **options** :

  - *no option*: use AC3 (default arc consistency)
  - Options.C_EXT_AC3: to get AC3 algorithm (searching from scratch for supports on all values)
  - Options.C_EXT_AC2001: to get AC2001 algorithm (maintaining the current support of each value)
  - Options.C_EXT_AC32: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - Options.C_EXT_AC322: to get AC3 with the used of BitSet to know if a support still exists

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: in_relation

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
ArrayList<int[]> couples2 = new ArrayList<int[]>();
couples2.add(new int[]{1, 2});
couples2.add(new int[]{1, 3});
couples2.add(new int[]{2, 1});
couples2.add(new int[]{3, 1});
couples2.add(new int[]{4, 1});
IntegerVariable v1 = makeIntVar("v1", 1, 4);
IntegerVariable v2 = makeIntVar("v2", 1, 4);
```

```
    m.addConstraint(feasPairAC(Options.C_EXT_AC32, v1, v2, couples2));
    s.read(m);
    s.solveAll();
```

7.33   feasTupleAC (constraint)

feasTupleAC(x, $feasTuples$) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $feasTuples$ of compatible tuples of values, and then enforces arc consistency:

$$\exists \text{ tuple } i \mid \quad (x_1, \ldots, x_n) = feasTuples[i]$$

The API is duplicated to define options.

- **API** :
  - feasTupleAC(List<int[]> feasTuples, IntegerVariable... x)
  - feasTupleAC(String options, List<int[]> feasTuples, IntegerVariable... x)

- **return type**: Constraint

- **options** :
  - *no option*: use AC32 (default arc consistency)
  - Options.C_EXT_AC32: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - Options.C_EXT_AC2001: to get AC2001 algorithm (maintaining the current support of each value)
  - Options.C_EXT_AC2008: to get AC2008 algorithm (maintained by STR)

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: in_relation

Example:

```
    Model m = new CPModel();
    Solver s = new CPSolver();
    IntegerVariable v1 = makeIntVar("v1", 0, 2);
    IntegerVariable v2 = makeIntVar("v2", 0, 4);
    ArrayList<int[]> feasTuple = new ArrayList<int[]>();
    feasTuple.add(new int[]{1, 1}); // x*y = 1
    feasTuple.add(new int[]{2, 4}); // x*y = 1
    m.addConstraint(feasTupleAC(Options.C_EXT_AC2001, feasTuple, v1, v2));
    s.read(m);
    s.solve();
```

7.34   feasTupleFC (constraint)

feasTupleFC(x, $feasTuples$) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $feasTuples$ of compatible tuples of values, and then performs Forward Checking:

$$\exists \text{ tuple } i \mid \quad (x_1, \ldots, x_n) = feasTuples[i]$$

- **API** : feasTupleFC(List<int[]> tuples, IntegerVariable... x)

- **return type**: Constraint

- **options** : *n/a*

- **favorite domain**: *to complete*

- **references** :
  global constraint catalog: in_relation

Example:

```java
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 4);
ArrayList<int[]> feasTuple = new ArrayList<int[]>();
feasTuple.add(new int[]{1, 1}); // x*y = 1
feasTuple.add(new int[]{2, 4}); // x*y = 1
m.addConstraint(feasTupleFC(feasTuple, v1, v2));
s.read(m);
s.solve();
```

7.35 forbiddenInterval (constraint)

forbiddenInterval(t) applies additionnal search tree reduction based on time intervals in which no operation can start or end in an optimal solution. *The tasks must all belong to one disjunctive resource and have fixed durations.*

- **API** :
  - forbiddenInterval(String name, TaskVariable[] tasks)
  - forbiddenInterval(TaskVariable[] tasks)

- **return type** : Constraint

- **options** : *n/a*

- **favorite domain** : *n/a*

Example: *to complete*

7.36 geost (constraint)

geost is a global constraint that generically handles a variety of geometrical placement problems. It handles geometrical constraints (non-overlapping, distance, etc.) between polymorphic objects (ex: polymorphism can be used for representing rotation) in any dimension. The parameters of geost($dim, objects, shiftedBoxes, eCtrs$) are respectively: the space dimension, the list of geometrical objects, the set of boxes that compose the shapes of the objects, the set of geometrical constraints. The greedy mode should be used without external constraints to have safe results, because it excludes external constraints from its exploration and look for instanciation of variables involved in geost which respect the geost constraint.

- **API** :
  ```
  geost(int dim, Vector<GeostObject> objects, Vector<ShiftedBox> shiftedBoxes, Vector<ExternalConstraint
  > eCtrs)
  geost(int dim, Vector<GeostObject> objects, Vector<ShiftedBox> shiftedBoxes, Vector<ExternalConstraint
  > eCtrs, Vector<int[]> ctrlVs)
  ```

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: geost

The geost constraint requires the creation of different objects:

parameter	type	description
objects	`Vector<GeostObject>`	geometrical objects
shiftedBoxes	`Vector<ShiftedBox>`	boxes that compose the object shapes
eCtrs	`Vector<ExternalConstraint>`	geometrical constraints
ctrlVs	`Vector<int[]>`	controlling vectors (for greedy mode)

Where a **GeostObject** is defined by:

attribute	type	description
dim	`int`	dimension
objectId	`int`	object id
shapeId	`IntegerVariable`	shape id
coordinates	`IntegerVariable[dim]`	coordinates of the origin
startTime	`IntegerVariable`	starting time
durationTime	`IntegerVariable`	duration
endTime	`IntegerVariable`	finishing time

Where a **ShiftedBox** is a *dim*-box defined by the shape it belongs to, its origin (the coordinates of the lower left corner of the box) and its lengths in every dimensions:

attribute	type	description
sid	`int`	shape id
offset	`int[dim]`	coordinates of the offset (lower left corner)
size	`int[dim]`	lengths in every dimensions

Where an **ExternalConstraint** contains informations and functionality common to all external constraints and is defined by:

attribute	type	description
ectrID	`int`	constraint id
dimensions	`int[]`	list of dimensions that the external constraint is active for
objectIdentifiers	`int[]`	list of object ids that this external constraint affects.

For further informations, visit the following page.

Example:

```
Model m = new CPModel();
int dim = 3;
```

```
    int lengths[] = {5, 3, 2};
    int widths[] = {2, 2, 1};
    int heights[] = {1, 1, 1};
    int nbOfObj = 3;
    long seed = 0;
    //Create the Objects
    Vector<GeostObject> obj = new Vector<GeostObject>();
    for (int i = 0; i < nbOfObj; i++) {
        IntegerVariable shapeId = Choco.makeIntVar("sid", i, i);
        IntegerVariable coords[] = new IntegerVariable[dim];
        for (int j = 0; j < coords.length; j++) {
            coords[j] = Choco.makeIntVar("x" + j, 0, 2);
        }
        IntegerVariable start = Choco.makeIntVar("start", 1, 1);
        IntegerVariable duration = Choco.makeIntVar("duration", 1, 1);
        IntegerVariable end = Choco.makeIntVar("end", 1, 1);
        obj.add(new GeostObject(dim, i, shapeId, coords, start, duration, end));
    }
    //Create the ShiftedBoxes and add them to corresponding shapes
    Vector<ShiftedBox> sb = new Vector<ShiftedBox>();
    int[] t = {0, 0, 0};
    for (int d = 0; d < nbOfObj; d++) {
        int[] l = {lengths[d], heights[d], widths[d]};
        sb.add(new ShiftedBox(d, t, l));
    }
    //Create the external constraints vector
    Vector<IExternalConstraint> ectr = new Vector<IExternalConstraint>();
    //create the list of dimensions for the external constraint
    int[] ectrDim = new int[dim];
    for (int d = 0; d < dim; d++)
        ectrDim[d] = d;
    //create the list of object ids for the external constraint
    int[] objOfEctr = new int[nbOfObj];
    for (int d = 0; d < nbOfObj; d++) {
        objOfEctr[d] = obj.elementAt(d).getObjectId();
    }
    //create and add one external constraint of type non overlapping
    NonOverlappingModel n = new NonOverlappingModel(Constants.NON_OVERLAPPING, ectrDim,
        objOfEctr);
    ectr.add(n);
    //create and post the geost constraint
    Constraint geost = Choco.geost(dim, obj, sb, ectr);
    m.addConstraint(geost);
    Solver s = new CPSolver();
    s.read(m);
    s.setValIntSelector(new RandomIntValSelector(seed));
    s.setVarIntSelector(new RandomIntVarSelector(s, seed));
    s.solveAll();
```

7.37   geq (constraint)

$\mathbf{geq}(x, y)$ states that x is greater than or equal to y:

$$x \geq y$$

- **API** :

    - geq(IntegerExpressionVariable x, IntegerExpressionVariable y)

    - geq(IntegerExpressionVariable x, int y)

    - geq(int x, IntegerExpressionVariable y)

    - geq(RealExpressionVariable x, RealExpressionVariable y)

    - geq(RealExpressionVariable x, double y)

    - geq(double x, RealExpressionVariable y)

- **return type** : Constraint

- **options** : *n/a*

- **favorite domain** : *to complete.*

- **references** :
  global constraint catalog: geq

Examples:

- example1:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(eq(v, c));
s.read(m);
s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(eq(w1, w2));
s.read(m);
s.solve();
```

7.38 geqCard (constraint)

geqCard(s, x) states that the cardinality of set s is greater than or equal to x:

$$|s| \geq x$$

- **API** :

    - geqCard(SetVariable s, IntegerVariable x)

    - geqCard(SetVariable s, int x)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable i = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(geqCard(set, i));
s.read(m);
s.solve();
```

7.39 globalCardinality (constraint)

globalCardinality(x, low, up) states bounds on the occurrence numbers of any value v in x (here, offset min is the minimum value over all variables in x) :

$$low[v - min] \leq |\{i \mid x_i = v\}| \leq up[v - min], \quad \forall \text{ value } v$$

Mulitple APIs exist:

- Given an array of variables x and an $offset$, the constraint ensures that the number of occurrences of the value v in x is between $low[v - offset]$ and $up[v - offset]$. Use the propagator of [Quimper et al., 2003].

- *variable cardinalities:* Given an array of variables x, an array of variables *card* to represent the cardinalities and an $offset$, the constraint ensures that the number of occurrences of the value v among the variables is equal to $card[v - offset]$. This constraint:

    - enforces Bound Consistency over x regarding the lower and upper bounds of *card*,
    - maintains the upper bounds of *card* by counting the number of variables in which each value can occur,
    - maintains the lower bounds of *card* by counting the number of variables instantiated to each value,
    - enforces $card[0] + \cdots + card[m] = n$, where n is the number of variables and m the number of values.

    Use the propagator of [Quimper et al., 2003].

The APIs are duplicated to define options.

- **API** :

    - globalCardinality(IntegerVariable[] x, int[] low, int[] up, int offset)
    - globalCardinality(String options, IntegerVariable[] x, int[] low, int[] up, int offset
)
    - globalCardinality(IntegerVariable[] x, IntegerVariable[] card, int offset)

- **return type** : Constraint

- **options**:

- *no option* :
- `Options.C_GCC_AC` : for [Régin, 1996] implementation of arc consistency
- `Options.C_GCC_BC` : for [Quimper et al., 2003] implementation of bound consistency

- **favorite domain** : *to complete*

- **references** :

- [Régin, 1996]: *Generalized arc consistency for global cardinality constraint,*
- [Quimper et al., 2003]: *An efficient bounds consistency algorithm for the global cardinality constraint*
- global constraint catalog: global_cardinality

Examples:

- example1:

```
int n = 5;
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    vars[i] = makeIntVar("var " + i, 1, n);
}
int[] LB2 = {0, 1, 1, 0, 3};
int[] UB2 = {0, 1, 1, 0, 3};
m.addConstraint(Options.C_GCC_BC, globalCardinality(vars, LB2, UB2, 1));
s.read(m);
s.solve();
```

- example2

```
int n = 5;
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = makeIntVarArray("vars", n, 1, n);
IntegerVariable[] cards = makeIntVarArray("cards", n, 0, 1);


m.addConstraint(Options.C_GCC_BC, globalCardinality(vars, cards, 1));
s.read(m);
s.solve();
```

7.40   gt (constraint)

$gt(x, y)$ states that x is strictly greater than y:

$$x > y$$

- **API** :

- gt(IntegerExpressionVariable x, IntegerExpressionVariable y)

  – `gt(IntegerExpressionVariable x, int y)`

  – `gt(int x, IntegerExpressionVariable y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete.*

- **references** :
  global constraint catalog: gt

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(gt(v, c));
s.read(m);
s.solve();
```

7.41 ifOnlyIf (constraint)

`ifOnlyIf`(c_1, c_2) states that c_1 holds if and only if c_2 holds:

$$c_1 \iff c_2$$

- **API** : `ifOnlyIf(Constraint c1, Constraint c2)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 3);
IntegerVariable y = makeIntVar("y", 1, 3);
IntegerVariable z = makeIntVar("z", 1, 3);
m.addVariables(Options.V_BOUND,x ,y, z);
m.addConstraint(ifOnlyIf(lt(x, y), lt(y, z)));
s.read(m);
s.solveAll();
```

7.42 ifThenElse (constraint)

`ifThenElse`(c_1, c_2, c_3) states that if c_1 holds then c_2 holds, otherwise c_3 holds:

$$(c_1 \wedge c_2) \vee (\neg c_1 \wedge c_3)$$

- **API** :ifThenElse(Constraint c1, Constraint c2, Constraint c3)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 3);
IntegerVariable y = makeIntVar("y", 1, 3);
IntegerVariable z = makeIntVar("z", 1, 3);
// use API ifThenElse(Constraint, Constraint, Constraint)
m.addConstraint(ifThenElse(lt((x), (y)), gt((y), (z)), FALSE));
 // and ifThenElse(Constraint, IntegerExpressionVariable, IntegerExpressionVariable)
m.addConstraint(leq(z, ifThenElse(lt(x, y), constant(1), plus(x,y))));
s.read(m);
s.solveAll();
```

7.43   implies (constraint)

implies(c_1, c_2) states that if c_1 holds then c_2 holds:

$$c_1 \implies c_2$$

- **API** : implies(Constraint c1, Constraint c2)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 2);
IntegerVariable y = makeIntVar("y", 1, 2);
IntegerVariable z = makeIntVar("z", 1, 2);
m.addVariables(Options.V_BOUND,x ,y, z);
Constraint e1 = implies(leq(x, y), leq(x, z));
m.addConstraint(e1);
s.read(m);
s.solveAll();
```

7.44   increasingnvalue (constraint)

increasing_nvalue($nval, variables$) states that $variables$ are increasing. In addition, $nval$ is the number of distinct values taken by $variables$.

- **API** : `increasing_nvalue(IntegerVariable nval, IntegerVariable[] variables)`

- **return type** : `Constraint`

- **options** :

  - *no option* filter on lower bound and on lower bound
  - `Options.C_INCREASING_NVALUE_ATLEAST` filter on lower bound only
  - `Options.C_INCREASING_NVALUE_ATMOST` filter on upper bound only
  - `Options.C_INCREASING_NVALUE_BOTH` –*default value*– filter on lower bound and on upper bound

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: increasing_nvalue

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable nval = makeIntVar("nval", 1, 3);
IntegerVariable[] variables = makeIntVarArray("vars", 6, 1, 4);
m.addConstraint(increasing_nvalue(Options.C_INCREASING_NVALUE_BOTH, nval, variables));
s.read(m);
s.solveAll();
```

7.45 infeasPairAC (constraint)

`infeasPairAC`$(x, y, infeasTuples)$ states an extensional binary constraint on (x, y) defined by the table *infeasTuples* of forbidden pairs of values, and then enforces arc consistency. Two APIs are available to define the forbidden pairs:

- if *infeasTuples* is encoded as a list of pairs `List<int[2]>`, then:

$$\forall \text{ tuple } i \mid \quad (x, y) \neq infeasTuples[i]$$

- if *infeasTuples* is encoded as a boolean matrix `boolean[][]`, let $\underline{x}$ and $\underline{y}$ be the initial minimum values of x and y, then:

$$\forall (u, v) \mid \quad (x, y) = (u + \underline{x}, v + \underline{y}) \ \vee \ \neg infeasTuples[u][v]$$

The two APIs are duplicated to allow definition of options.

- **API** :

  - `infeasPairAC(IntegerVariable x, IntegerVariable y, List<int[]> infeasTuples)`
  - `infeasPairAC(String options, IntegerVariable x, IntegerVariable y, List<int[]> infeasTuples)`
  - `infeasPairAC(IntegerVariable x, IntegerVariable y, boolean[][] infeasTuples)`
  - `infeasPairAC(String options, IntegerVariable x, IntegerVariable y, boolean[][] infeasTuples)`

- **return type** : `Constraint`

- **options** :

  - *no option*: use AC3 (default arc consistency)
  - `Options.C_EXT_AC3`: to get AC3 algorithm (searching from scratch for supports on all values)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_AC322`: to get AC3 with the used of `BitSet` to know if a support still exists

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
boolean[][] matrice2 = new boolean[][]{
            {false, true, true, false},
            {true, false, false, false},
            {false, false, true, false},
            {false, true, false, false}};
IntegerVariable v1 = makeIntVar("v1", 1, 4);
IntegerVariable v2 = makeIntVar("v2", 1, 4);
m.addConstraint(feasPairAC(Options.C_EXT_AC32,v1, v2, matrice2));
s.read(m);
s.solveAll();
```

7.46 infeasTupleAC (constraint)

`infeasTupleAC`$(x, feasTuples)$ states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table *infeasTuples* of compatible tuples of values, and then enforces arc consistency:

$$\forall \text{ tuple } i \mid \quad (x_1, \ldots, x_n) \neq infeasTuples[i]$$

The API is duplicated to define options.

- **API** :

  - `infeasTupleAC(List<int[]> infeasTuples, IntegerVariable... x)`
  - `infeasTupleAC(String options, List<int[]> infeasTuples, IntegerVariable... x)`

- **return type**: `Constraint`

- **options** :

  - *no option*: use AC32 (default arc consistency)
  - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
  - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
  - `Options.C_EXT_AC2008`: to get AC2008 algorithm (maintained by STR)

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
ArrayList<int[]> forbiddenTuples = new ArrayList<int[]>();
forbiddenTuples.add(new int[]{1, 1, 1});
forbiddenTuples.add(new int[]{2, 2, 2});
forbiddenTuples.add(new int[]{2, 5, 3});
m.addConstraint(infeasTupleAC(forbiddenTuples, x, y, z));
s.read(m);
s.solveAll();
```

7.47   infeasTupleFC (constraint)

`infeasTupleFC`$(x, feasTuples)$ states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the table $infeasTuples$ of compatible tuples of values, and then performs Forward Checking:

$$\forall \text{ tuple } i \mid \quad (x_1, \ldots, x_n) \neq infeasTuples[i]$$

- **API** : `infeasTupleFC(List<int[]> infeasTuples, IntegerVariable... x)`

- **return type**: `Constraint`

- **options** : n/a

- **favorite domain**: *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
ArrayList<int[]> forbiddenTuples = new ArrayList<int[]>();
forbiddenTuples.add(new int[]{1, 1, 1});
forbiddenTuples.add(new int[]{2, 2, 2});
forbiddenTuples.add(new int[]{2, 5, 3});
m.addConstraint(infeasTupleFC(forbiddenTuples, x, y, z));
s.read(m);
s.solveAll();
```

7.48   intDiv (constraint)

`intDiv`(x, y, z) states that the z is equal to the integer quotient of x by y:

$$z = \lfloor x/y \rfloor$$

- **API**: `intDiv(IntegerVariable x, IntegerVariable y, IntegerVariable z)`

- **return type** : `Constraint`

- **option** : n/a

- **favorite domain**: bound

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
long seed = 0;
IntegerVariable x = makeIntVar("x", 3, 5);
IntegerVariable y = makeIntVar("y", 1, 2);
IntegerVariable z = makeIntVar("z", 0, 5);
m.addConstraint(intDiv(x, y, z));
s.setVarIntSelector(new RandomIntVarSelector(s, seed));
s.setValIntSelector(new RandomIntValSelector(seed + 1));
s.read(m);
s.solve();
```

7.49   inverseChanneling (constraint)

`inverseChanneling`(x, y) states a channeling between two arrays x and y of integer variables with the same domain. It enforces that if the i-th element of x is equal to j then the j-th element of y is equal to i and conversely:
$$x_i = j \iff y_j = i$$

- **API** : `inverseChanneling(IntegerVariable[] x, IntegerVariable[] y)`

- **return type** : `Constraint`

- **options** : *no options*

- **favorite domain** : enumerated for x

- **references** :
  global constraint catalog: inverse

Example:

```
int n = 8;
Model m = new CPModel();
IntegerVariable[] queens = new IntegerVariable[n];
IntegerVariable[] queensdual = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    queens[i] = makeIntVar("Q" + i, 1, n);
    queensdual[i] = makeIntVar("QD" + i, 1, n);
}
for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
        int k = j - i;
        m.addConstraint(neq(queens[i], queens[j]));
        m.addConstraint(neq(queens[i], plus(queens[j], k)));  // diagonal constraints
        m.addConstraint(neq(queens[i], minus(queens[j], k))); // diagonal constraints
    }
}
```

```
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            int k = j - i;
            m.addConstraint(neq(queensdual[i], queensdual[j]));
            m.addConstraint(neq(queensdual[i], plus(queensdual[j], k))); // diagonal
                constraints
            m.addConstraint(neq(queensdual[i], minus(queensdual[j], k))); // diagonal
                constraints
        }
    }
    m.addConstraint(inverseChanneling(queens, queensdual));
    Solver s = new CPSolver();
    s.read(m);
    s.solveAll();
```

7.50 inverseset (constraint)

inverseset(iv, sv) states a channeling between an array iv of integer variables and an array sv of set variables. It enforces that value j belongs to $sv[i]$ if and only if $iv[j]$ is equal to i and conversely:

$$sv_i = j \quad \Longleftrightarrow \quad iv_i = j$$

- **API** : inverseSet(IntegerVariable[] iv, SetVariable[] sv)

- **return type** : Constraint

- **options** : *no options*

- **favorite domain** : enumerated for iv

- **references** :
  global constraint catalog: inverse_set

Example:

```
    int i = 4;
    int j = 2;
    Model m = new CPModel();
    IntegerVariable[] iv = makeIntVarArray("iv", i, 0, j);
    SetVariable[] sv = makeSetVarArray("sv", j, 0, i);

    m.addConstraint(inverseSet(iv, sv));
    Solver s = new CPSolver();
    s.read(m);
    s.solveAll();
```

7.51 isIncluded (constraint)

isIncluded(x, y) states that the second set y contains the first set x:

$$x \subseteq y$$

- **API** : `isIncluded(SetVariable x, SetVariable y)`

- **return type** : `Constraint`

- **options** : *n/a*

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable v1 = makeSetVar("v1", 3, 4);
SetVariable v2 = makeSetVar("v2", 3, 8);
m.addConstraint(isIncluded(v1, v2));
s.read(m);
s.solveAll();
```

7.52   isNotIncluded (constraint)

`isNotIncluded`(x, y) states that the second set y does not contain the first set x:

$$x \nsubseteq y$$

- **API** : `isNotIncluded(SetVariable x, SetVariable y)`

- **return type** : `Constraint`

- **options** : *n/a*

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable v1 = makeSetVar("v1", 3, 4);
SetVariable v2 = makeSetVar("v2", 3, 8);
m.addConstraint(isNotIncluded(v1, v2));
s.read(m);
s.solveAll();
```

7.53   knapsackProblem (constraint)

`knapsackProblem`(x, W, V, w, v) states that :

$$\sum_{i=1}^{n} x_i w_i = W \wedge \sum_{i=1}^{n} x_i v_i = V$$

x_i is the number of copies of the item i, each kind of item i has a value v_i and a weight w_i.

It constructs a knapsack problem constraint with an underlying costregular constraint, where it simulates M. Triks dynamic programming approach.

- **API** :

- `knapsackProblem(IntegerVariable[] vars, IntegerVariable cVar1, IntegerVariable cVar2, int[] costs1, int[] costs2)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

- **references** :

  - [Demassey et al., 2006]: *A `Cost-Regular` based hybrid column generation approach*
  - [Trick, 2003]: *A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints*
  - costRegular

Example:

```
CPModel m = new CPModel();
IntegerVariable[] items = new IntegerVariable[3];
items[0] = makeIntVar("item_1", 0, 5);
items[1] = makeIntVar("item_2", 0, 7);
items[2] = makeIntVar("item_3", 0, 10);

IntegerVariable sumWeight = makeIntVar("sumWeight", 0, 40, Options.V_BOUND);
IntegerVariable sumValue = makeIntVar("sumValue", 0, 34, Options.V_OBJECTIVE);

int[] weights = new int[]{7, 5, 3};
int[] values = new int[]{6, 4, 2};

Constraint knapsack = Choco.knapsackProblem(items, sumWeight, sumValue, weights, values)
    ;
m.addConstraint(knapsack);

Solver s = new CPSolver();
s.read(m);
s.maximize(true);
```

7.54 leq (constraint)

`leq(x, y)` states that x is less than or equal to y:

$$x \leq y$$

- **API** :

  - `leq(IntegerExpressionVariable x, IntegerExpressionVariable y)`
  - `leq(IntegerExpressionVariable x, int y)`
  - `leq(int x, IntegerExpressionVariable y)`
  - `leq(RealExpressionVariable x, RealExpressionVariable y)`
  - `leq(RealExpressionVariable x, double y)`
  - `leq(double x, RealExpressionVariable y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete.*

- **references** :
  global constraint catalog: leq

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(leq(v, c));
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(leq(w1, w2));
s.read(m);
s.solve();
```

7.55 leqCard (constraint)

`leqCard(s, x)` states that the cardinality of set s is less than or equal to x:

$$|s| \leq x$$

- **API** :

  – `leqCard(SetVariable s, IntegerVariable x)`

  – `leqCard(SetVariable s, int x)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable i = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(leqCard(set, i));
s.read(m);
s.solve();
```

7.56   lex (constraint)

lex(x, y) enforces a strict lexicographic ordering $x <_{lex} y$ between two arrays of same size n:

$$\exists\, j \in \{1, \ldots, n\} \mid \qquad x_j < y_j \quad \wedge \quad x_i = y_i \,(\forall\, i < j)$$

- **API** : `lex(IntegerVariable[] x, IntegerVariable[] y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

  - [Frisch et al., 2002]: *Global Constraints for Lexicographic Orderings*
  - global constraint catalog: lex_less

Example:

```java
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    vs1[i] = makeIntVar("" + i, 0, k);
    vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lex(vs1, vs2));
s.read(m);
s.solve();
```

7.57   lexChain (constraint)

lexChain$(x^1, x^2, x^3, \ldots)$ enforces a strict lexicographic ordering on a chain of integer vectors:

$$x^1 <_{lex} x^2 <_{lex} x^3 <_{lex} \cdots$$

- **API** : `lexChain(IntegerVariable[]... arrayOfVectors)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

  - [Carlsson and Beldiceanu, 2002] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*

– global constraint catalog: lex_chain_less

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexChain(vs1, vs2));
s.read(m);
s.solve();
```

7.58   lexChainEq (constraint)

`lexChainEq`$(x^1, x^2, x^3, \ldots)$ enforces a lexicographic ordering on a chain of integer vectors:

$$x^1 \leq_{lex} x^2 \leq_{lex} x^3 \leq_{lex} \cdots$$

- **API** : `lexChainEq(IntegerVariable[]... arrayOfVectors)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

  – [Carlsson and Beldiceanu, 2002] *Arc-Consistency for a chain of Lexicographic Ordering Constraints*

  – global constraint catalog: lex_chain_lesseq

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
   vs1[i] = makeIntVar("" + i, 0, k);
   vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexChainEq(vs1, vs2));
s.read(m);
s.solve();
```

7.59   lexeq (constraint)

lexeq(x, y) enforces a lexicographic ordering $x \leq_{lex} y$ between two arrays of same size n:

$$\exists \, j \in \{1, \ldots, n\} \mid \quad x_j \leq y_j \quad \wedge \quad x_i = y_i \; (\forall \; i < j)$$

- **API** : lexeq(IntegerVariable[] x, IntegerVariable[] y)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

    - [Frisch et al., 2002]: *Global Constraints for Lexicographic Orderings*
    - global constraint catalog: lex_lesseq

Example: **Example**:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    vs1[i] = makeIntVar("" + i, 0, k);
    vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lexeq(vs1, vs2));
s.read(m);
s.solve();
```

7.60   leximin (constraint)

TODO: verify the specifications of the implemented version.

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ be two vectors of n integers, and let x' and y' be respectively permutations of vectors x and y sorted by increasing order of the components. Constraint leximin(x, y) holds if and only if $x' <_{lex} y'$:

$$\exists \, j \in \{1, \ldots, n\} \mid \quad x'_j < y'_j \quad \wedge \quad x'_i = y'_i \; (\forall \; i < j)$$

- **API** :

    - leximin(IntegerVariable[] x, IntegerVariable[] y)
    - leximin(int[] x, IntegerVariable[] y)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

  – [Frisch et al., 2003]: *Multiset ordering constraints*

  – global constraint catalog: lex_lesseq_allperm (variant)

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] u = makeIntVarArray("u", 3, 2, 5);
IntegerVariable[] v = makeIntVarArray("v", 3, 2, 4);
m.addConstraint(leximin(u, v));
m.addConstraint(allDifferent(v));
s.read(m);
s.solve();
```

7.61   lt (constraint)

$\mathtt{lt}(x, y)$ states that x is strictly smaller than y:

$$x < y$$

- **API** :

  – lt(IntegerExpressionVariable x, IntegerExpressionVariable y)

  – lt(IntegerExpressionVariable x, int y)

  – lt(int x, IntegerExpressionVariable y)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete.*

- **references** :
  global constraint catalog: lt

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int c = 1;
IntegerVariable v = makeIntVar("v", 0, 2);
m.addConstraint(lt(v, c));
s.read(m);
s.solve();
```

7.62 max (constraint)

7.62.1 max of a list

$\max(x, z)$ states that z is equal to the greater element of vector x:

$$z = \max(x_1, x_2, ..., x_n)$$

- **API**:
    - max(IntegerVariable[] x, IntegerVariable z)
    - max(IntegerVariable x1, IntegerVariable x2, IntegerVariable z)
    - max(int x1, IntegerVariable x2, IntegerVariable z)
    - max(IntegerVariable x1, int x2, IntegerVariable z)

- **return type**: Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: maximum

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
m.addVariables(Options.V_BOUND, x, y, z);
m.addConstraint(max(y, z, x));
s.read(m);
s.solve();
```

7.62.2 max of a set

$\max(s, x, z)$ states that z is equal to the greater element of vector x whose index is in set s:

$$z = \max_{i \in s}(x_i)$$

- **API**:
    - max(SetVariable s, IntegerVariable[] x, IntegerVariable z)

- **return type**: Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable[] x = constantArray(new int[]{5, 7, 9, 10, 12, 3, 2});
        IntegerVariable max = makeIntVar("max", 1, 100);
        SetVariable set = makeSetVar("set", 0, x.length - 1);
        m.addConstraints(max(set, x, max), leqCard(set, constant(5)));
        s.read(m);
        s.solve();
```

7.63   member (constraint)

$\mathtt{member}(x, s)$ states that integer x is contained in set s:

$$x \in s.$$

- **API** :

    - member(int x, SetVariable s)

    - member(SetVariable s, int x)

    - member(SetVariable s, IntegerVariable x)

    - member(IntegerVariable x, SetVariable s)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: in_set

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        int x = 3;
        int card = 2;
        SetVariable y = makeSetVar("y", 2, 4);
        m.addConstraint(member(y, x));
        m.addConstraint(eqCard(y, card));
        s.read(m);
        s.solveAll();
```

7.64   min (constraint)

7.64.1   min of a list

$\mathtt{mix}(x, z)$ states that z is equal to the smaller element of vector x:

$$z = \min(x_1, x_2, ..., x_n).$$

- **API**:

    – `min(IntegerVariable[] x, IntegerVariable z)`

    – `min(IntegerVariable x1, IntegerVariable x2, IntegerVariable z)`

    – `min(int x1, IntegerVariable x2, IntegerVariable z)`

    – `min(IntegerVariable x1, int x2, IntegerVariable z)`

- **return type**: `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: minimum

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
m.addVariables(Options.V_BOUND, x, y, z);
m.addConstraint(min(y, z, x));
s.read(m);
s.solve();
```

7.64.2   min of a set

$min(s, x, z)$ states that z is equal to the smaller element of vector x whose index is in set s:

$$z = \min_{i \in s}(x_i).$$

- **API**:

    – `min(SetVariable s, IntegerVariable[] x, IntegerVariable z)`

- **return type**: `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] x = constantArray(new int[]{5, 7, 9, 10, 12, 3, 2});
IntegerVariable min = makeIntVar("min", 1, 100);
SetVariable set = makeSetVar("set", 0, x.length - 1);
m.addConstraints(min(set, x, min), leqCard(set, constant(5)));
s.read(m);
s.solve();
```

7.65   mod (constraint)

$\texttt{mod}(x_1, x_2, x_3)$ states that x_1 is congruent to x_2 modulo x_3:

$$x_1 \equiv x_2 \mod x_3$$

- **API** : `mod(IntegerVariable x1, IntegerVariable x2, int x3)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int n = 4;
int k = 2;
IntegerVariable[] vs1 = new IntegerVariable[n];
IntegerVariable[] vs2 = new IntegerVariable[n];
for (int i = 0; i < n; i++) {
    vs1[i] = makeIntVar("" + i, 0, k);
    vs2[i] = makeIntVar("" + i, 0, k);
}
m.addConstraint(lex(vs1, vs2));
s.read(m);
s.solve();
```

7.66   multiCostRegular (constraint)

$\texttt{multiCostRegular}(x, z, \mathcal{L}(\Pi), c)$ states that sequence x is a word belonging to the regular language $\mathcal{L}(\Pi)$,

$$(x_1, \ldots, x_n) \in \mathcal{L}(\Pi)$$

and that the bounded vector z is equal to the costs of x according to the assigment cost matrix c:

$$\sum_{i=1}^{n} c[r][i][x_i] = z[r], \quad \forall r \in \{0, \ldots, R\}$$

`multiCostRegular` is a conjunction of a `regular` constraint with $R+1$ cost functions. It may be used in the context of personnel scheduling problems, handling complex work regulations by the mean of regular expressions, together with cardinality or financial constraints by the mean of cost functions. The filtering algorithm associated with `multiCostRegular` is based on lagrangian relaxation and computations of shortest/longest paths in a layered digraph [Menana and Demassey, 2009]. It typically performs more filtering than the conjunction of `costRegular` and `globalCardinality` or than multiple `costRegular`.

The accepting language is specified by a deterministic finite automaton (DFA): Automaton Π is defined on a given *alphabet* $\Sigma \subseteq \mathbb{Z}$ by a set $Q = \{0, \ldots, m\}$ of *states*, a subset $A \subseteq Q$ of *final* or *accepting states* and a table $\Delta \subseteq Q \times \Sigma \times Q$ of *transitions* between states. Π is encoded as an object of class `Automaton`, whose API contains:

```
Automaton();
int addState();
```

```
void setStartingState(int state);
void setAcceptingState(int state);
void addTransition(int state1, int state2, int label);
int getNbStates();
```

The cost functions are encoded as one matrix `int cost[nTime][nAct][auto.getNbStates()][nRes]` such that `cost[i][j][s][r]` is the cost of assigning variable x_i to activity j at state s on dimension $r + 1$.

- **API** : `multiCostRegular(IntegerVariable[] x, IntegerVariable[] z, Automaton P, int[][][][] c)`

- **return type** : `Constraint`

- **options** :

  - `MultiCostRegular.DATA_STRUCT` is `MultiCostRegular.BITSET` or `MultiCostRegular.LIST`: a parameter stating which backtrable data structure to use for storing the outgoing arcs of the layered digraph. The observed behaviour is until 1000 arcs the bipartite list is much more efficient, afterwards the memory efficiency of the bitset representation allow faster operations.
  - `MultiCostRegular.U0`, `MultiCostRegular.R0`, `MultiCostRegular.MAXNONIMPROVEITER`, and `MultiCostRegular.MAXBOUNDITER` are value parameters of the subgradient algorithm used for solving the lagrangean relaxation.
  - `MultiCostRegular.D_PREC` is a double parameter stating the precision of float computation. It is set by default to 10^{-5}.

- **favorite domain** : *to complete*

- **references** :
  [Menana and Demassey, 2009]: *Sequencing and Counting with the* `multicost-regular` *Constraint*

Example:

```
import choco.kernel.model.constraints.automaton.FA.Automaton;
```

```
//0- declare parameters
int DAY = 0;
int NIGHT = 1;
int REST = 2;
//1- create the model
Model m = new CPModel();
int nTime = 14; // 2 weeks: 14 days
int nAct = 3; // 3 activities: DAY, NIGHT, REST
int nRes = 4; // 4 resources: cost (0), #DAY (1), #NIGHT (2), #WORK (3)
//2- Create the schedule variables: the activity processed at each time slot
IntegerVariable[] sequence = makeIntVarArray("x", nTime, 0, nAct - 1, Options.V_ENUM);
// - create the cost variables (one for each resource)
IntegerVariable[] bounds = new IntegerVariable[4];
bounds[0] = makeIntVar("z_0", 30, 80, Options.V_BOUND); // 30 <= cost <= 80
bounds[1] = makeIntVar("day", 0, 7, Options.V_BOUND); // 0 <= #DAY <= 7
bounds[2] = makeIntVar("night", 3, 7, Options.V_BOUND); // 3 <= #NIGHT <= 7
bounds[3] = makeIntVar("work", 7, 9, Options.V_BOUND); // 7 <= #WORK <= 9
//3- Create the automaton
FiniteAutomaton auto = new FiniteAutomaton();
// state 0: starting and accepting state
int start = auto.addState();
auto.setInitialState(start);
auto.setFinal(start);
// state 1 and a transition (0,DAY,1)
int first = auto.addState();
auto.addTransition(start, first, DAY);
```

```
    // state 2 and transitions (1,DAY,2), (1,NIGHT,2), (2,REST,0), (0,NIGHT,2)
    int second = auto.addState();
    auto.addTransition(first, second, DAY, NIGHT);
    auto.addTransition(second, start, REST);
    auto.addTransition(start, second, NIGHT);
    //4- Declare the assignment/transition costs:
    // csts[i][j][s][r]: cost on resource r of assigning Xi to activity j at state s
    int[][][][] csts = new int[nTime][nAct][auto.getNbStates()][nRes];
    for (int i = 0; i < csts.length; i++) {
        csts[i][DAY][0] = new int[]{3, 1, 0, 1}; // costs of transition (0,DAY,1)
        csts[i][NIGHT][0] = new int[]{8, 0, 1, 1}; // costs of transition (0,NIGHT,2)
        csts[i][DAY][1] = new int[]{5, 1, 0, 1}; // costs of transition (1,DAY,2)
        csts[i][NIGHT][1] = new int[]{9, 0, 1, 1}; // costs of transition (1,NIGHT,2)
        csts[i][REST][2] = new int[]{2, 0, 0, 0}; // costs of transition (2,REST,0)
    }

    //5- add the constraint
    m.addConstraint(multiCostRegular(sequence, bounds, auto, csts));
    //6- create the solver, read the model and solve it
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
```

7.67   neq (constraint)

neq states that the two arguments are different:

$$x \neq y.$$

- **API** :

    - neq(IntegerExpressionVariable x, IntegerExpressionVariable y)

    - neq(IntegerExpressionVariable x, int y)

    - neq(int x, IntegerExpressionVariable y)

- **return type** : Constraint

- **options** : *n/a*

- **favorite domain** : *to complete.*

- **references** :
  global constraint catalog: neq

Examples:

- example1:

```
    Model m = new CPModel();
    Solver s = new CPSolver();
    int c = 1;
    IntegerVariable v = makeIntVar("v", 0, 2);
    m.addConstraint(neq(v, c));
    s.read(m);
    s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 2);
IntegerVariable v2 = makeIntVar("v2", 0, 2);
IntegerExpressionVariable w1 = plus(v1, 1);
IntegerExpressionVariable w2 = minus(v2, 1);
m.addConstraint(neq(w1, w2));
s.read(m);
s.solve();
```

7.68   neqCard (constraint)

neqCard(s, x) states that the cardinality of set s is not equal to x:

$$|s|! = x$$

- **API** :
    - neqCard(SetVariable s, IntegerVariable x)
    - neqCard(SetVariable s, int x)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable set = makeSetVar("s", 1, 5);
IntegerVariable card = makeIntVar("card", 2, 3);
m.addConstraint(member(set, 3));
m.addConstraint(neqCard(set, card));
s.read(m);
s.solve();
```

7.69   not (constraint)

not(c) holds if and only if constraint c does not hold:

$$\neg c$$

- **API** : not(Constraint c)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a

Example :

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", 1, 10);
m.addConstraint(not(geq(x, 3)));
s.read(m);
s.solve();
```

7.70   notMember (constraint)

notMember(x, s) states that integer x is not contained in set s:

$$x \notin s$$

- **API** :

  – notMember(int x, SetVariable s)

  – notMember(SetVariable s, int x)

  – notMember(SetVariable s, IntegerVariable x)

  – notMember(IntegerVariable x, SetVariable s)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int x = 3;
int card = 2;
SetVariable y = makeSetVar("y", 2, 4);
m.addConstraint(notMember(y, x));
m.addConstraint(eqCard(y, card));
s.read(m);
s.solveAll();
```

7.71   nth (constraint)

nth is the well known *element* constraint. Several APIs are available:

- nth(i, x, y) ensures that $x[i] = y$

- nth(i, x, y, o) ensures that $x[i + o] = y$ (o is an *offset* for shifting values)

- nth(i, j, x, y) ensures that $x[i][j] = y$

- **API** :

    - nth(IntegerVariable i, int[] x, IntegerVariable y)

    - nth(String option, IntegerVariable i, int[] x, IntegerVariable y)

    - nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y)

    - nth(IntegerVariable i, int[] x, IntegerVariable y, int offset)

    - nth(String option, IntegerVariable i, int[] x, IntegerVariable y, int offset)

    - nth(IntegerVariable i, IntegerVariable[] x, IntegerVariable y, int offset)

    - nth(String option, IntegerVariable i, IntegerVariable[] x, IntegerVariable y, int offset
)

    - nth(IntegerVariable i, IntegerVariable j, int[][] x, IntegerVariable y)

- **return type** : Constraint

- **options** :

    - *no option*

    - Options.C_NTH_G for global consistency

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: element

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
int[][] values = new int[][]{
    {1, 2, 0, 4, -323},
    {2, 1, 0, 3, 42},
    {6, 1, -7, 4, -40},
    {-1, 0, 6, 2, -33},
    {2, 3, 0, -1, 49}};
IntegerVariable index1 = makeIntVar("index1", -3, 10);
IntegerVariable index2 = makeIntVar("index2", -3, 10);
IntegerVariable var = makeIntVar("value", -20, 20);
m.addConstraint(nth(index1, index2, values, var));
s.read(m);
s.solveAll();
```

7.72   occurrence (constraint)

occurrence(v, z, x) states that z is equal to the number of elements in x with value v:

$$z = |\{i \mid x_i = v\}|$$

This is a specialization of the globalCardinality constraint.

- **API**: occurrence(int v, IntegerVariable z, IntegerVariable... x)

- **return type** : Constraint

- **options** :n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);
m.addConstraint(occurrence(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}));
s.read(m);
s.solve();
```

7.73 occurrenceMax (constraint)

occurrenceMax(v, z, x) states that z is at least equal to the number of elements in x with value v:

$$z \geq |\{i \mid x_i = v\}|$$

This is a specialization of the `globalCardinality` constraint.

- **API**: `occurrenceMax(int v, IntegerVariable z, IntegerVariable... x)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);
m.addConstraint(occurrenceMax(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}))
    ;
s.read(m);
s.solve();
```

7.74 occurrenceMin (constraint)

occurrenceMin(v, z, x) states that z is at most equal to the number of elements in x with value v:

$$z \leq |\{i \mid x_i = v\}|$$

This is a specialization of the `globalCardinality` constraint.

- **API**: `occurrenceMin(int v, IntegerVariable z, IntegerVariable... x)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
  global constraint catalog: count

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x1 = makeIntVar("X1", 0, 10);
IntegerVariable x2 = makeIntVar("X2", 0, 10);
IntegerVariable x3 = makeIntVar("X3", 0, 10);
IntegerVariable x4 = makeIntVar("X4", 0, 10);
IntegerVariable x5 = makeIntVar("X5", 0, 10);
IntegerVariable x6 = makeIntVar("X6", 0, 10);
IntegerVariable x7 = makeIntVar("X7", 0, 10);
IntegerVariable y1 = makeIntVar("Y1", 0, 10);
m.addConstraint(occurrenceMin(3, y1, new IntegerVariable[]{x1, x2, x3, x4, x5, x6, x7}))
    ;
s.read(m);
s.solve();
```

7.75 oppositeSign (constraint)

oppositeSign(x, y) states that the two arguments have opposite signs:

$$xy \leq 0$$

0 is considered as both sign, if one argument is equal to 0, the constraint is not satisfied.

- **API** : `oppositeSign(IntegerExpressionVariable x, IntegerExpressionVariable y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable x = makeIntVar("x", -1, 1);
IntegerVariable y = makeIntVar("y", -1, 1);
IntegerVariable z = makeIntVar("z", 0, 1000);
m.addConstraint(oppositeSign(x,y));
m.addConstraint(eq(z, plus(mult(x, -425), mult(y, 391))));
s.read(m);
s.solve();
```

7.76   or (constraint)

$\mathtt{or}(c_1, \ldots, c_n)$ states that one or more of the constraints in arguments are satisfied:

$$c_1 \vee c_2 \vee \ldots \vee c_n$$

$\mathtt{or}(b_1, \ldots, b_n)$ states that one or more of the 0-1 variables in arguments are true (equal to 1):

$$b_1 = 1 \vee b_2 = 1 \vee \ldots \vee b_n = 1$$

- **API** :
    - `or(Constraint... c)`
    - `or(IntegerVariable... b)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

Examples:

- example1:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 1);
IntegerVariable v2 = makeIntVar("v2", 0, 1);
m.addConstraint(or(eq(v1, 1), eq(v2, 1)));
s.read(m);
s.solve();
```

- example2

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable[] vars = makeBooleanVarArray("b", 10);
m.addConstraint(or(vars));
s.read(m);
s.solve();
```

7.77   pack (constraint)

`pack(items, load, bin, size)` states that a collection of items is packed into different bins, such that the total size of the items in each bin does not exceed the bin capacity:

$$\texttt{load}[b] = \sum_{i \in \texttt{items}[b]} \texttt{size}[i], \quad \forall \text{ bin } b$$

$$i \in \texttt{items}[b] \iff \texttt{bin}[i] = b, \quad \forall \text{ bin } b, \ \forall \text{ item } i$$

`pack` is a bin packing constraint based on [Shaw, 2004].

- **API** :
  - `pack(SetVariable[] items, IntegerVariable[] load, IntegerVariable[] bin, IntegerConstantVariable[] size, String... options)`
  - `pack(PackModeler modeler, String... options)`: PackModeler is a high-level modeling object.
  - `pack(int[] sizes, int nbBins, int capacity, String... options)`: build instance with Pack-Modeler.

- **Variables**:
  - `SetVariable[] items`:   items[b] is the set of items packed into bin b.
  - `IntegerVariable[] load`:   load[b] is the total size of the items packed into bin b.
  - `IntegerVariable[] bin`:   bin[i] is the bin where item i is packed into.
  - `IntegerConstantVariable[] size`:   size[i] is the size of item i.

- **return type** : `Constraint`

- **options** :
  - SettingType.ADDITIONAL_RULES.getOptionName(): additional filtering rules *recommended*
  - SettingType.DYNAMIC_LB.getOptionName(): feasibility tests based on dynamic lower bounds for 1D-bin packing
  - SettingType.FILL_BIN.getOptionName(): dominance rule: fill a bin when an item fit into pertfectly equal-sized items and bins must be equivalent
  - SettingType.LAST_BINS_EMPTY.getOptionName(): empty bins are the last ones

- **favorite domain** : *to complete*

- **references** :
  - [Shaw, 2004]: *A constraint for bin packing*
  - global constraint catalog: bin_packing (variant)

Example:
Take a look at *samples.pack* to see advanced use of the constraint.

```
import choco.cp.solver.SettingType;
import choco.cp.solver.search.integer.varselector.StaticVarOrder;
```

```
Model m = new CPModel();
m.addConstraint(pack(new int[]{5,3,2,6,8,5},5,10, SettingType.ADDITIONAL_RULES.
    getOptionName()));
Solver s = new CPSolver();
s.read(m);
s.solve();
```

7.78   precedenceReified (constraint)

`precedenceReified`(x_1, d, x_2, b) states that x_1 plus duration d is less than or equal to x_2 requires boolean b to be true:

$$b \quad \Longleftrightarrow \quad x_1 + d \le x_2$$

- **API** : `precedenceReified(IntegerVariable x1, int d, IntegerVariable x2, IntegerVariable b)`

- **return type** : `Constraint`

- **options** : *n/a*

- **favorite domain** : *to complete*

Example:

```
    int k1 = 5;
    Model m = new CPModel();
 IntegerVariable x = makeIntVar("x", 1, 10);
 IntegerVariable y = makeIntVar("y", 1, 10);
 m.addVariables(Options.V_BOUND, x, y);
 IntegerVariable z = makeIntVar("z", 0, 1);
 m.addConstraint(precedenceReified(x,k1,y,z));
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
```

7.79   precedenceimplied (constraint)

To complete

7.80   precedence (constraint)

To complete

7.81   precedencedisjoint (constraint)

To complete

7.82   regular (constraint)

`regular`$(x, \mathcal{L}(\Pi))$ states that sequence x is a word belonging to the regular language $\mathcal{L}(\Pi)$:

$$(x_1, \ldots, x_n) \in \mathcal{L}(\Pi)$$

The accepting language can be specified either by a deterministic finite automaton (DFA), a list of feasible or infeasible tuples, or a regular expression:

DFA: Automaton Π is defined on a given *alphabet* $\Sigma \subseteq \mathbb{Z}$ by a set $Q = \{0, \ldots, m\}$ of *states*, a subset $A \subseteq Q$ of *final* or *accepting states* and a table $\Delta \subseteq Q \times \Sigma \times Q$ of *transitions* between states. Δ is encoded as `List<Transition>` where a Transition object $\delta = $ `new Transition`(q_i, σ, q_j) is made of three integers expressing the ingoing state q_i, the label σ, and the outgoing state q_j. Automaton Π is a DFA if Δ is finite and if it has only one initial state (here, state 0 is considered as the unique initial state) and no two transitions sharing the same ingoing state and the same label.

feasible tuples: *regular* can be used as an extensional constraint. Given the list of *feasible* tuples for sequence x, this API builds a DFA from the list, and then enforces GAC on the constraint. Using `regular` can be more efficient than a standard GAC algorithm on tables of tuples if the tuples are structured so that the resulting DFA is compact. The DFA is built from the list of tuples by computing incrementally the minimal DFA after each addition of tuple.

infeasible tuples: An another API allows to specify the list of *infeasible* tuples and then builds the corresponding feasible DFA. This operation requires to know the entire alphabet, hence this API has two mandatory table fields *min* and *max* defining the minimum and maximum values of each variable x_i.

regular expression: Finally, the `regular` constraint can be based on a regular expression, such as `String regexp = "(12)(3*)";`— This expression recognizes any (possibly empty) sequences of 3 preceded by at least one 1 or one 2.

- **API** :
    - `regular(DFA pi, IntegerVariable[] x)`
    - `regular(IntegerVariable[] x, List<int[]> feasTuples)`
    - `regular(IntegerVariable[] x, List<int[]> infeasTuples, int[] min, int[] max)`
    - `regular(String regexp, IntegerVariable[] x)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :
    [Pesant, 2004]: *A regular language membership constraint*

Examples:

- example 1 with DFA:

```
import choco.kernel.model.constraints.automaton.DFA;
import choco.kernel.model.constraints.automaton.Transition;
import choco.kernel.model.constraints.Constraint;
```

```
//1- Create the model
Model m = new CPModel();
int n = 6;
IntegerVariable[] vars = new IntegerVariable[n];
for (int i = 0; i < vars.length; i++) {
    vars[i] = makeIntVar("v" + i, 0, 5);
}
//2- Build the list of transitions of the DFA
List<Transition> t = new LinkedList<Transition>();
t.add(new Transition(0, 1, 1));
t.add(new Transition(1, 1, 2));
// transition with label 1 from state 2 to state 3
t.add(new Transition(2, 1, 3));
t.add(new Transition(3, 3, 0));
```

```
    t.add(new Transition(0, 3, 0));
    //3- Two final states: 0, 3
    List<Integer> fs = new LinkedList<Integer>();
    fs.add(0); fs.add(3);
    //4- Build the DFA
    DFA auto = new DFA(t, fs, n);
    //5- add the constraint
    m.addConstraint(regular(auto, vars));
    //6- create the solver, read the model and solve it
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
    do {
        for (int i = 0; i < n; i++)
        System.out.print(s.getVar(vars[i]).getVal());
        System.out.println("");
    } while (s.nextSolution());
    //7- Print the number of solution found
    System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

• example 2 with feasible tuples:

```
    //1- Create the model
    Model m = new CPModel();
    IntegerVariable v1 = makeIntVar("v1", 1, 4);
    IntegerVariable v2 = makeIntVar("v2", 1, 4);
    IntegerVariable v3 = makeIntVar("v3", 1, 4);
    //2- add some allowed tuples (here, the tuples define a all_equal constraint)
    List<int[]> tuples = new LinkedList<int[]>();
    tuples.add(new int[]{1, 1, 1});
    tuples.add(new int[]{2, 2, 2});
    tuples.add(new int[]{3, 3, 3});
    tuples.add(new int[]{4, 4, 4});
    //3- add the constraint
    m.addConstraint(regular(new IntegerVariable[]{v1, v2, v3}, tuples));
    //4- Create the solver, read the model and solve it
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
    do {
        System.out.println("("+s.getVar(v1)+","+s.getVar(v2)+","+s.getVar(v3)+")");
    } while (s.nextSolution());
    //5- Print the number of solution found
    System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

• example 3 with regular expression:

```
    //1- Create the model
    Model m = new CPModel();
    int n = 6;
    IntegerVariable[] vars = makeIntVarArray("v", n, 0, 5);
    //2- add the constraint
    String regexp = "(1|2)(3*)(4|5)";
    m.addConstraint(regular(regexp, vars));
    //3- Create the solver, read the model and solve it
    Solver s = new CPSolver();
    s.read(m);
    s.solve();
    do {
```

```
            for (int i = 0; i < n; i++)
                System.out.print(s.getVar(vars[i]).getVal());
            System.out.println("");
        } while (s.nextSolution());
        //4- Print the number of solution found
        System.out.println("Nb_sol␣:␣" + s.getNbSolutions());
```

7.83 reifiedAnd (constraint)

To complete

7.84 reifiedConstraint (constraint)

- `reifiedConstraint`(b, c) states that boolean b is true if and only if constraint c holds:

$$b \iff c$$

- `reifiedConstraint`(b, c_1, c_2) states that boolean b is true if and only if c_1 holds, and b is false if and only if c_2 holds (c_2 must be the opposite constraint of c_1):

$$(b \land c_1) \lor (\neg b \land c_2)$$

- **API** :

    – `reifiedConstraint(IntegerVariable b, Constraint c)`

    – `reifiedConstraint(IntegerVariable b, Constraint c1, Constraint c2)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

Parameter b is a boolean variable (enumerated domain with two values $\{0, 1\}$) and c is a constraint over Integer variables or Set variables.

The constraint c to reify has to provide its opposite (the opposite is needed for propagation). Most basic constraints of Choco provides their opposite by default, and can then be reified using the first API. The second API attends to reify user-defined constraints as it allows the user to directly specify the **opposite** constraint.

`reifiedConstraint` filter algorithm:

1. if b is instantiated to 1 (resp. to 0), then c (resp. $\neg c$) is propagated

2. otehrwise

    (a) if c is entailed, b is set to 1

    (b) else b is set to 0.

Example:

```
        CPModel m = new CPModel();
        CPSolver s = new CPSolver();
        IntegerVariable b = makeIntVar("b", 0, 1);
        IntegerVariable x = makeIntVar("x", 0, 10);
```

```
        IntegerVariable y = makeIntVar("y", 0, 10);
         // reified constraint (x<=y)
         m.addConstraint(reifiedConstraint(b, leq(x, y)));
        s.read(m);
        s.solveAll();
```

7.85   reifiedLeftImp (constraint)

To complete

7.86   reifiedNot (constraint)

To complete

7.87   reifiedOr (constraint)

To complete

7.88   reifiedRightImp (constraint)

To complete

7.89   reifiedXnor (constraint)

To complete

7.90   reifiedXor (constraint)

To complete

7.91   relationPairAC (constraint)

`relationPairAC`(x, y, rel) states an extensional binary constraint on (x, y) defined by the binary relation rel:

$$(x, y) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory.

The API is duplicated to allow definition of options.

- **API** :

  – `relationPairAC(IntegerVariable x, IntegerVariable y, BinRelation rel)`

  – `relationPairAC(String options, IntegerVariable x, IntegerVariable y, BinRelation rel)`

- **return type** : `Constraint`

- **options** :

  – *no option* : use AC3 (default arc consistency)

  – `Options.C_EXT_AC3`: to get AC3 algorithm (searching from scratch for supports on all values)

- `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)

- `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)

- `Options.C_EXT_AC322`: to get AC3 with the used of `BitSet` to know if a support still exists

- **favorite domain** : *to complete*

Example:

```
import choco.kernel.solver.constraints.integer.extension.CouplesTest;
import choco.kernel.solver.constraints.integer.extension.TuplesTest;
```

```java
    public static class MyEquality extends CouplesTest {

        public boolean checkCouple(int x, int y) {
            return x == y;
        }
    }
```

```java
        Model m = new CPModel();
        Solver s = new CPSolver();
        IntegerVariable v1 = makeIntVar("v1", 1, 4);
        IntegerVariable v2 = makeIntVar("v2", 1, 4);
        IntegerVariable v3 = makeIntVar("v3", 3, 6);
        m.addConstraint(relationPairAC(Options.C_EXT_AC32, v1, v2, new MyEquality()));
        m.addConstraint(relationPairAC(Options.C_EXT_AC32, v2, v3, new MyEquality()));
        s.read(m);
        s.solveAll();
```

7.92   relationTupleAC (constraint)

`relationTupleAC(x, rel)` states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the n-ary relation rel, and then enforces arc consistency:

$$(x_1, \ldots, x_n) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory. The API is duplicated to define options.

- **API**:

    - `relationTupleAC(IntegerVariable[] x, LargeRelation rel)`
    - `relationTupleAC(String options, IntegerVariable[] x, LargeRelation rel)`

- **return type**: `Constraint`

- **options** :

    - *no option*: use AC32 (default arc consistency)
    - `Options.C_EXT_AC32`: to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)
    - `Options.C_EXT_AC2001`: to get AC2001 algorithm (maintaining the current support of each value)
    - `Options.C_EXT_AC2008`: to get AC2008 algorithm (maintained by STR)

- **favorite domain** : *to complete*

Example :

```java
public static class NotAllEqual extends TuplesTest {

  public boolean checkTuple(int[] tuple) {
    for (int i = 1; i < tuple.length; i++) {
      if (tuple[i - 1] != tuple[i]) return true;
    }
    return false;
  }
}
```

```java
    Model m = new CPModel();
    IntegerVariable x = makeIntVar("x", 1, 5);
    IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
m.addConstraint(relationTupleAC(new IntegerVariable[]{x, y, z}, new NotAllEqual()));
    Solver s = new CPSolver();
s.read(m);
s.solveAll();
```

7.93   relationTupleFC (constraint)

`relationTupleFC`(x, rel) states an extensional constraint on $(x_1, \ldots, x_n)$ defined by the n-ary relation rel, and then enforces forward checking:

$$(x_1, \ldots, x_n) \in rel$$

Many constraints of the same kind often appear in a model. Relations can therefore often be shared among many constraints to spare memory.

- **API**: `relationTupleFC(IntegerVariable[] x, LargeRelation rel)`

- **return type**: `Constraint`

- **options** : *n/a*

- **favorite domain** : *to complete*

Example :

```java
public static class NotAllEqual extends TuplesTest {

  public boolean checkTuple(int[] tuple) {
    for (int i = 1; i < tuple.length; i++) {
      if (tuple[i - 1] != tuple[i]) return true;
    }
    return false;
  }
}
```

```java
    Model m = new CPModel();
    IntegerVariable x = makeIntVar("x", 1, 5);
IntegerVariable y = makeIntVar("y", 1, 5);
IntegerVariable z = makeIntVar("z", 1, 5);
    m.addConstraint(relationTupleFC(new IntegerVariable[]{x, y, z}, new NotAllEqual()));
```

```
    Solver s = new CPSolver();
      s.read(m);
    s.solveAll();
```

7.94 sameSign (constraint)

> `sameSign`(x, y) states that the two arguments have the same sign:
>
> $$xy \geq 0$$

- **API** : `sameSign(IntegerExpressionVariable x, IntegerExpressionVariable y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
    Model m = new CPModel();
    Solver s = new CPSolver();
    IntegerVariable x = makeIntVar("x", -1, 1);
    IntegerVariable y = makeIntVar("y", -1, 1);
    IntegerVariable z = makeIntVar("z", 0, 1000);
    m.addConstraint(oppositeSign(x,y));
    m.addConstraint(eq(z, plus(mult(x, -425), mult(y, 391))));
    s.read(m);
    s.solve();
    System.out.println(s.getVar(z).getVal());
```

7.95 setDisjoint (constraint)

> `setDisjoint`$(s_1, \ldots, s_n)$ states that the arguments are pairwise disjoint:
>
> $$s_i \cap s_j = \emptyset, \quad \forall \, i \neq j$$

- **API** : `setDisjoint(SetVariable[] sv)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        SetVariable x = makeSetVar("X", 1, 3);
        SetVariable y = makeSetVar("Y", 1, 3);
        SetVariable z = makeSetVar("Z", 1, 3);
        Constraint c1 = setDisjoint(x, y, z);
        m.addConstraint(c1);
        s.read(m);
        s.solveAll();
```

7.96   setInter (constraint)

setInter(s_1, s_2, s_3) states that the third set s_3 is exactly the intersection of the two first sets:

$$s_1 \cap s_2 = s_3$$

- **API** : setInter(SetVariable s1, SetVariable s2, SetVariable s3)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
        Model m = new CPModel();
        Solver s = new CPSolver();
        SetVariable x = makeSetVar("X", 1, 3);
        SetVariable y = makeSetVar("Y", 1, 3);
        SetVariable z = makeSetVar("Z", 2, 3);
        Constraint c1 = setInter(x, y, z);
        m.addConstraint(c1);
        s.read(m);
        s.solveAll();
```

7.97   setUnion (constraint)

setUnion(sv, s_{union}) states that the s_{union} set is exactly the union of the sets sv:

$$sv_1 \cup sv_2 \cup \ldots sv_i \cup sv_{i+1} \ldots \cup sv_n = s_{union}$$

- **API** :
  - setUnion(SetVariable s1, SetVariable s2, SetVariable union)
  - setUnion(SetVariable[] sv, SetVariable union)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : *to complete*

Example:

```
Model m = new CPModel();
Solver s = new CPSolver();
SetVariable x = makeSetVar("X", 1, 3);
SetVariable y = makeSetVar("Y", 3, 5);
SetVariable z = makeSetVar("Z", 0, 6);
Constraint c1 = setUnion(x, y, z);
m.addConstraint(c1);
s.read(m);
s.solveAll();
```

7.98 sorting (constraint)

sorting(x, y) holds on the set of variables being either in x or in y, and is satisfied by v if and only if v(y) is the sorted version of v(x) in increasing order.

$$y = x_s orted$$

- **API**: `sorting(IntegerVariable[] x, IntegerVariable[] y)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

  – [Bleuzen-Guernalec and Colmerauer, 1997]: *Narrowing a block of sortings in quadratic time*
  – [Mehlhorn and Thiel, 2000]: *Faster algorithms for bound-consistency of the Sortedness and the Alldifferent constraint*
  – global constraint catalog: count

Example:

```
CPModel m = new CPModel();
int n = 3;
IntegerVariable[] x = makeIntVarArray("x", n, 0, n);
IntegerVariable[] y = makeIntVarArray("y", n, 0, n);
m.addConstraint(sorting(x, y));
m.addConstraint(allDifferent(x));
CPSolver s = new CPSolver();
s.read(m);
s.solveAll();
```

7.99 startsAfter (constraint)

startsAfter(t, c) states that the task variable t starts after c:

$$s_t \geq c$$

- **API** :startsAfter(final TaskVariable t, final int c)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a.

Examples: *to complete*

7.100 startsAfterBegin (constraint)

startsAfterBegin(t_1, t_2, c) states that the task variable t_1 starts after the beginning of t_2, minus c:

$$s_{t_1} \geq s_{t_2} - c$$

- **API** :startsAfterBegin(TaskVariable t1, TaskVariable t2, int c)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a.

Examples: *to complete*

7.101 startsAfterEnd (constraint)

startsAfterEnd(t_1, t_2, c) states that the task variable t_1 starts after the end of t_2 minus c:

$$s_{t_1} \leq e_{t_2} - c$$

- **API** :startsAfterEnd(TaskVariable t1, TaskVariable t2, int c)

- **return type** : Constraint

- **options** : n/a

- **favorite domain** : n/a.

Examples: *to complete*

7.102 startsBefore (constraint)

startsBefore(t, c) states that the task variable t starts before c:

$$s_t \leq c$$

- **API** : startsBefore(final TaskVariable t, final int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.103 startsBeforeBegin (constraint)

startsBeforeBegin(t_1, t_2, c) states that the task variable t_1 starts before the beginning of t_2, minus c:

$$s_{t_1} \leq s_{t_2} - c$$

- **API** : startsBeforeBegin(TaskVariable t1, TaskVariable t2, int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.104 startsBeforeEnd (constraint)

startsBeforeEnd(t_1, t_2, c) states that the task variable t_1 starts before the end of t_2, minus c:

$$s_{t_1} \leq e_{t_2} - c$$

- **API** : startsBeforeEnd(TaskVariable t1, TaskVariable t2, int c)
- **return type** : Constraint
- **options** : n/a
- **favorite domain** : n/a.

Examples: *to complete*

7.105 startsBetween (constraint)

startsBetween(t, c_1, c_2) states that the task variable t starts between c_1 and c_2:

$$c_1 \leq s_t \leq c_2$$

- **API** :startsBetween(`TaskVariable t, int min, int max`)

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a.

Examples: *to complete*

7.106 stretchPath (constraint)

A *stretch* in a sequence x is a maximum subsequence of (consecutive) identical values. stretchPath$(param, x)$ enforces the minimal and maximal length of the stretches in sequence x of any values given in *param*: Consider the sequence x as a concatenation of stretches $x^1.x^2 \ldots x^k$ with v^i and l^i being respectively the value and the length of stretch x^i,

$$\forall i \in \{1, \ldots, k\}, \ \forall j, \quad param[j][0] = v^i \quad \Longrightarrow \quad param[j][1] \leq l^i \leq param[j][2]$$

Useful for Rostering Problems. `stretchPath` is implemented by a `regular` constraint that performs GAC. The bounds on the stretch lengths are defined by *param* a list of triples of integers: [*value, min, max*] specifying the minimal and maximal lengths of any stretch of the corresponding value.

This API requires a Java library on automaton available on http://www.brics.dk/automaton/. (It is contained in the Choco jar file.)

- **API** : stretchPath(`List<int[]> param, IntegerVariable... x`)

- **return type** : Constraint

- **options** :n/a

- **favorite domain** : *to complete*

- **references** :

    - [Pesant, 2004]: *A regular language membership constraint*
    - global constraint catalog: stretch_path

Example:

```
Model m = new CPModel();
int n = 7;
IntegerVariable[] vars = makeIntVarArray("v", n, 0, 2);
//define the stretches
ArrayList<int[]> lgt = new ArrayList<int[]>();
lgt.add(new int[]{0, 2, 2}); // stretches of value 0 are of length 2
lgt.add(new int[]{1, 2, 3}); // stretches of value 1 are of length 2 or 3
```

```
        lgt.add(new int[]{2, 2, 2}); // stretches of value 2 are of length 2
        m.addConstraint(stretchPath(lgt, vars));
        Solver s = new CPSolver();
        s.read(m);
        s.solve();
```

7.107 times (constraint)

`times`(x_1, x_2, x_3) states that the third argument is equal to the product of the two arguments:

$$x_3 = x_1 \times x_2.$$

- **API**:

    - `times(IntegerVariable x1, IntegerVariable x2, IntegerVariable x3)`

    - `times(int x1, IntegerVariable x2, IntegerVariable x3)`

    - `times(IntegerVariable x1, int x2, IntegerVariable x3)`

- **return type** : `Constraint`

- **option** : n/a

- **favorite domain**: bound

Example:

```
        Model m = new CPModel();
        IntegerVariable x = makeIntVar("x", 1, 2);
        IntegerVariable y = makeIntVar("y", 3, 5);
        IntegerVariable z = makeIntVar("z", 3, 10);
        m.addConstraint(times(x, y, z));
        Solver s = new CPSolver();
        s.read(m);
        s.solve();
```

7.108 tree (constraint)

Let $G = (V, A)$ be a digraph on $V = \{1, \ldots, n\}$. G can be modeled by a sequence of domain variables $x = (x_1, \ldots, x_n) \in V^n$ – the *successors* variables – whose respective domains are given by $D_i = \{j \in V \mid (i, j) \in A\}$. Conversely, when instantiated, x defines a subgraph $G_x = (V, A_x)$ of G with $A_x = \{(i, x_i) \mid i \in V\} \subseteq A$. Such a subgraph has one particularity: any connected component of G_x contains either no loop – and then it contains a cycle – or exactly one loop $x_i = i$ and then it is a *tree* of root i (literally, it is an anti-arborescence as there exists a path from each node to i and i has a loop).

> $\texttt{tree}(x, restrictions)$ is a vertex-disjoint graph partitioning constraint. It states that G_x is a forest (its connected components are trees) that satisfies some conditions specified by $restrictions$. $\texttt{tree}$ deals with several kinds of graph restrictions on:
>
> - the number of trees
>
> - the number of proper trees (a tree is proper if it contains more than 2 nodes)
>
> - the weigth of the partition: the sum of the weights of the edges
>
> - incomparability: some nodes in pairs have to belong to distinct trees
>
> - precedence: some nodes in pairs have to belong to the same tree in a given order
>
> - conditional precedence: some nodes in pairs have to respect a given order if they belong to the same tree
>
> - the in-degree of the nodes
>
> - the time windows on nodes (given travelling times on arcs)

Many applications require to partition a graph such that each component contains exactly one *resource* node and several *task* nodes. A typical example is a routing problem where vehicle routes are paths (a path is a special case of tree) starting from a depot and delivering goods to several clients. Another example is a local network where each computer has to be connected to one shared printer. Last, one can cite the problem of reconstructing plylogeny trees. The constraint $\texttt{tree}$ can handle these kinds of problems with many additional constraints on the structure of the partition.

- **API** : `tree(TreeParametersObject param)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : *to complete*

- **references** :

    - [Beldiceanu et al., 2008]: *Combining tree partitioning, precedence, and incomparability constraints*

    - global constraint catalog: proper_forest (variant)

The tree constraint API requires a particular Model object, named `TreeParametersObject`. It can be created with the following parameters:

parameter	type	description
n	`int`	number of nodes in the initial graph G
$nTree$	`IntegerVariable`	number of trees in the resulting forest G_x
$nProper$	`IntegerVariable`	number of proper trees in G_x
$objective$	`IntegerVariable`	(bounded) total cost of G_x
$graphs$	`List<BitSet[]>`	graphs encoded as successor lists,
		`graphs[0]` the initial graph G,
		`graphs[1]` a precedence graph,
		`graphs[2]` a conditional precedence graph,
		`graphs[3]` an incomparability graph
$matrix$	`List<int[][]>`	`matrix[0]` the indegree of each node, and
		`matrix[1]` the starting time from each node
$travel$	`int[][]`	the travel time of each arc

Example:

```
import choco.kernel.model.variables.tree.TreeParametersObject;
```

```
    Model m = new CPModel();
    int nbNodes = 7;
    //1- create the variables involved in the partitioning problem
    IntegerVariable ntree = makeIntVar("ntree",1,5);
    IntegerVariable nproper = makeIntVar("nproper",1,1);
    IntegerVariable objective = makeIntVar("objective",1,100);
    //2- create the different graphs modeling restrictions
    List<BitSet[]> graphs = new ArrayList<BitSet[]>();
    BitSet[] succ = new BitSet[nbNodes];
    BitSet[] prec = new BitSet[nbNodes];
    BitSet[] condPrecs = new BitSet[nbNodes];
    BitSet[] inc = new BitSet[nbNodes];
    for (int i = 0; i < nbNodes; i++) {
        succ[i] = new BitSet(nbNodes);
        prec[i] = new BitSet(nbNodes);
        condPrecs[i] = new BitSet(nbNodes);
        inc[i] = new BitSet(nbNodes);
    }
     // initial graph (encoded as successors variables)
    succ[0].set(0,true); succ[0].set(2,true); succ[0].set(4,true);
    succ[1].set(0,true); succ[1].set(1,true); succ[1].set(3,true);
    succ[2].set(0,true); succ[2].set(1,true); succ[2].set(3,true); succ[2].set(4,true);
    succ[3].set(2,true); succ[3].set(4,true); // successor of 3 is either 2 or 4
    succ[4].set(2,true); succ[4].set(3,true);
    succ[5].set(4,true); succ[5].set(5,true); succ[5].set(6,true);
    succ[6].set(3,true); succ[6].set(4,true); succ[6].set(5,true);
     // restriction on precedences
    prec[0].set(4,true); // 0 has to precede 4
    prec[4].set(3,true); prec[4].set(2,true);
    prec[6].set(4,true);
     // restriction on conditional precedences
    condPrecs[5].set(1,true); // 5 has to precede 1 if they belong to the same tree
     // restriction on incomparability:
    inc[0].set(6,true); inc[6].set(0,true); // 0 and 6 have to belong to distinct trees
    graphs.add(succ);
    graphs.add(prec);
    graphs.add(condPrecs);
    graphs.add(inc);
    //3- create the different matrix modeling restrictions
    List<int[][]> matrix = new ArrayList<int[][]>();
     // restriction on bounds on the indegree of each node
    int[][] degree = new int[nbNodes][2];
    for (int i = 0; i < nbNodes; i++) {
        degree[i][0] = 0; degree[i][1] = 2; // 0 <= indegree[i] <= 2
    }
    matrix.add(degree);
     // restriction on bounds on the starting time at each node
    int[][] tw = new int[nbNodes][2];
    for (int i = 0; i < nbNodes; i++) {
        tw[i][0] = 0; tw[i][1] = 100; // 0 <= start[i] <= 100
    }
    tw[0][1] = 15; // 0 <= start[0] <= 15
    tw[2][0] = 35; tw[2][1] = 40; // 35 <= start[2] <= 45
    tw[6][1] = 5; // 0 <= start[6] <= 5
    matrix.add(tw);
    //4- matrix for the travel time between each pair of nodes
    int[][] travel = new int[nbNodes][nbNodes];
    for (int i = 0; i < nbNodes; i++) {
```

```
        for (int j = 0; j < nbNodes; j++) travel[i][j] = 100000;
    }
    travel[0][0] = 0; travel[0][2] = 10; travel[0][4] = 20;
    travel[1][0] = 20; travel[1][1] = 0; travel[1][3] = 20;
    travel[2][0] = 10; travel[2][1] = 10; travel[2][3] = 5; travel[2][4] = 5;
    travel[3][2] = 5; travel[3][4] = 2;
    travel[4][2] = 5; travel[4][3] = 2;
    travel[5][4] = 15; travel[5][5] = 0; travel[5][6] = 10;
    travel[6][3] = 5; travel[6][4] = 20; travel[6][5] = 10;
    //5- create the input structure and the tree constraint
    TreeParametersObject parameters = new TreeParametersObject(nbNodes, ntree, nproper,
        objective
        , graphs, matrix, travel);
    Constraint c = Choco.tree(parameters);
    m.addConstraint(c);
    Solver s = new CPSolver();
    s.read(m);
    //6- heuristic: choose successor variables as the only decision variables
    s.setVarIntSelector(new StaticVarOrder(s, s.getVar(parameters.getSuccVars())));
    s.solveAll();
```

7.109 TRUE (constraint)

TRUE always returns *true*.

7.110 xnor (constraint)

xnor(b_1, b_2) states that the 0-1 variables in arguments take same value:

$$(b_1 = 1 \land b_2 = 1) \lor (b_1 = 0 \land b_2 = 0)$$

- **API** : xnor(IntegerVariable b1, IntegerVariable b2)

- **return type** : Constraint

- **options** : *n/a*

- **favorite domain** : *n/a*

Examples:

```
    Model m = new CPModel();
    Solver s = new CPSolver();
    IntegerVariable v1 = makeIntVar("v1", 0, 1);
    IntegerVariable v2 = makeIntVar("v2", 0, 1);
    m.addConstraint(xnor(v1,v2));
    s.read(m);
    s.solve();
```

7.111 xor (constraint)

xor(b_1, b_2) states that the 0-1 variables in arguments take distinct value:

$$(b_1 = 1 \land b_2 = 0) \lor (b_1 = 0 \land b_2 = 1)$$

- **API** : `xor(IntegerVariable... b)`

- **return type** : `Constraint`

- **options** : n/a

- **favorite domain** : n/a

Examples:

```
Model m = new CPModel();
Solver s = new CPSolver();
IntegerVariable v1 = makeIntVar("v1", 0, 1);
IntegerVariable v2 = makeIntVar("v2", 0, 1);
m.addConstraint(xor(v1,v2));
s.read(m);
s.solve();
```

Chapter 8

Options (Model)

This section lists and details the options that can be declared on variables or constraints within a Choco Model .

8.1 Options and settings

The variables and some constraints allow the declaration of options. Options and settings are defined in the classes `Options` and `SettingType`. Default options are specified. Most of the time, options parameters are *varargs*.

8.1.1 Options for variables:

- `Options.NO_OPTION`, `""`, or *empty argument*

- `Options.V_BOUND` or `"cp:bound"`

    - **goal** : force the solver to create bounded domain variable. It is a domain where only bound propagation can be done (no holes). It is very well suited when constraints performing only Bound Consistency are added on the corresponding variables. It must be used when large domains are needed. Implemented by two integers.

    - **scope** :

        * IntegerVariable
        * SetVariable
        * TaskVariable

- `Options.V_ENUM` or `"cp:enum"`

    - **goal** : force the solver to create enumerated domain variable *(default option)*. It is a domain in which holes can be created by the solver. It should be used when discrete and quite small domains are needed and when constraints performing Arc Consistency are added on the corresponding variables. Implemented by a `BitSet` object.

    - **scope** :

        * IntegerVariable
        * SetVariable
        * TaskVariable

- `Options.V_BTREE` or `"cp:btree"`

    - **goal** : force the solver to create binary tree domain variable. *Under development*

    - **scope** : IntegerVariable

- `Options.V_BLIST` or `"cp:blist"`

- **goal** : force the solver to create bipartite list domain variable. It is a domain where unavailable values are placed in the left part of the list, the other one on the right one.
- **scope** : IntegerVariable

- Options.V_LINK or "cp:link"

  - **goal** : force the solver to create linked list domain variable. It is an enumerated domain where holes can be done and every values has a link to the previous value and to the next value. It is built by giving its name and its bounds: lower bound and upper bound. It must be used when the very small domains are needed, because although linked list domain consumes more memory than the BitSet implementation, it can provide good performance as iteration over the domain is made in constant time. Implemented by a LinkedList object.
  - **scope** : IntegerVariable

- Options.V_MAKESPAN or "cp:makespan"

  - **goal** : declare the current variable as makespan
  - **scope** : IntegerVariable

- Options.V_NO_DECISION or "cp:no_decision"

  - **goal** : force variable to be removed from the pool of decisionnal variables of the default search strategy
  - **scope** :
    * IntegerVariable
    * SetVariable
    * RealVariable
    * TaskVariable

- Options.V_OBJECTIVE or "cp:objective"

  - **goal** : declare the objective variable
  - **scope** :
    * IntegerVariable
    * SetVariable
    * RealVariable

8.1.2 Options for expressions:

- Options.NO_OPTION, "", or *empty argument*

- Options.E_DECOMP or "cp:decomp"

  - **goal** : force decomposition of the scoped expression.
  - **scope** : IntegerExpressionVariable

8.1.3 Options and settings for constraints:

- Options.NO_OPTION, "", or *empty argument*

- Options.C_EXT_AC3 or "cp:ac3"

  - **goal** : to get AC3 algorithm (searching from scratch for supports on all values)
  - **scope** :
    * feasPairAC(String, IntegerVariable, IntegerVariable, List)
    * infeasPairAC(String, IntegerVariable, IntegerVariable, List)

* infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
* relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)

- `Options.C_EXT_AC32` or `"cp:ac32"`

  – **goal** : to get AC3rm algorithm (maintaining the current support of each value in a non backtrackable way)

  – **scope** :

  * feasPairAC(String, IntegerVariable, IntegerVariable, List)
  * feasTupleAC(String, List, IntegerVariable[])
  * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
  * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
  * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)
  * relationTupleAC(String, IntegerVariable[], LargeRelation)

- `Options.C_EXT_AC322` or `"cp:ac322"`

  – **goal** : to get AC3 with the used of `BitSet` to know if a support still exists

  – **scope** :

  * feasPairAC(String, IntegerVariable, IntegerVariable, List)
  * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
  * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
  * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)

- `Options.C_EXT_AC2001` or `"cp:ac2001"`

  – **goal** : to get AC2001 algorithm (maintaining the current support of each value)

  – **scope** :

  * feasPairAC(String, IntegerVariable, IntegerVariable, List)
  * feasTupleAC(String, List, IntegerVariable[])
  * infeasPairAC(String, IntegerVariable, IntegerVariable, List)
  * infeasPairAC(String, IntegerVariable, IntegerVariable, boolean[][])
  * relationPairAC(String, IntegerVariable, IntegerVariable, BinRelation)
  * relationTupleAC(String, IntegerVariable[], LargeRelation)

- `Options.C_EXT_AC2008` or `"cp:ac2008"`

  – **goal** : to get AC2008 algorithm (maintained by STR)

  – **scope** :

  * feasTupleAC(String, List, IntegerVariable[])
  * infeasTupleAC(String, List, IntegerVariable[])
  * relationTupleAC(String, IntegerVariable[], LargeRelation)

- `Options.C_EXT_FC` or `"cp:fc"`

  – **goal** : set filter policy to forward checking

  – **scope** :

  * feasTupleAC(String, List, IntegerVariable[])
  * infeasTupleAC(String, List, IntegerVariable[])
  * relationTupleAC(String, IntegerVariable[], LargeRelation)

- `Options.C_ALLDIFFERENT_AC` or `"cp:ac"`

  – **goal** : for Regin implementation

– **scope** : allDifferent(String, IntegerVariable[])

- `Options.C_ALLDIFFERENT_BC` or `"cp:bc"`

    – **goal** : for bound all different using the propagator of A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. IJCAI-2003

    – **scope** : allDifferent(String, IntegerVariable[])

- `Options.C_ALLDIFFERENT_CLIQUE` or `"cp:clique"`

    – **goal** : propagate on the clique of differences

    – **scope** : allDifferent(String, IntegerVariable[])

- `Options.C_GCC_AC` or `"cp:ac"`

    – **goal** : for Regin implementation

    – **scope** : globalCardinality(String, IntegerVariable[], int[], int[], int)

- `Options.C_GCC_BC` or `"cp:bc"`

    – **goal** : for Quimper implementation

    – **scope** : globalCardinality(String, IntegerVariable[], int[], int[], int)

- `Options.C_INCREASING_NVALUE_ATLEAST` or `"cp:atleast"`

    – **goal** : set filtering policy to filter on lower bound only

    – **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])

- `Options.C_INCREASING_NVALUE_ATMOST` or `"cp:atmost"`

    – **goal** : set filtering policy to filter on upper bound only

    – **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])

- `Options.C_INCREASING_NVALUE_BOTH` or `"cp:both"`

    – **goal** : set filtering policy to filter on lower and upper bound only

    – **scope** : increasing_nvalue(String, IntegerVariable, IntegerVariable[])

- `Options.C_NTH_G` or `"cp:G"`

    – **goal** : global consistency

    – **scope** :

        * nth(String options, IntegerVariable index, int[] values, IntegerVariable val)
        * nth(String option, IntegerVariable index, IntegerVariable[] varArray, IntegerVariable val)
        * nth(String options, IntegerVariable index, IntegerVariable[] varArray, IntegerVariable val, int offset)

- `Options.C_CLAUSES_ENTAIL` or `"cp:entail"`

    – **goal** : ensures quick entailment tests

    – **scope** : clause(IntegerVariable[],IntegerVariable[])

- `Options.C_POST_PONED` or `"cp:postponed"`

    – **goal** : postponed a constraint

    – **scope** : Constraint

- `SettingType.ADDITIONAL_RULES` or `"cp:pack:ar"`

- **goal** : more filtering rules (recommended)
- **scope** :
  * pack(int[] sizes, int nbBins, int capacity, String... options)
  * pack(PackModeler modeler,String... options)
  * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
  * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- SettingType.DYNAMIC_LB or "cp:pack:dlb"

  - **goal** : feasibility test based on a dynamic lower bound
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- SettingType.FILL_BIN or "cp:pack:fill"

  - **goal** : dominance rule: fill a bin when an item fit into pertfectly equal-sized items and bins must be equivalent
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- SettingType.LAST_BINS_EMPTY or "cp:pack:lbe"

  - **goal** : empty bins are the last ones
  - **scope** :
    * pack(int[] sizes, int nbBins, int capacity, String... options)
    * pack(PackModeler modeler,String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes, String... options)
    * pack(SetVariable[] itemSets, IntegerVariable[] loads, IntegerVariable[] bins, IntegerConstantVariable[] sizes,IntegerVariable nbNonEmpty, String... options)

- SettingType.TASK_INTERVAL or "cp:cumul:ti"

  - **goal** : for fast task intervals
  - **scope** :
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)

- `SettingType.`SLOW_TASK_INTERVAL or `"cp:cumul:sti"`

  - **goal** : for slow task intervals
  - **scope** :
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)

- `SettingType.`VILIM_CEF_ALGO or `"cp:cumul:cef"`

  - **goal** : for Vilim theta lambda tree + lazy computation of the inner maximization of the edge finding rule of Van hentenrick and Mercier
  - **scope** :
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)

- `SettingType.`VHM_CEF_ALGO_N2K or `"cp:cumul:scef"`

  - **goal** : for Simple $n^2 * k$ algorithm (lazy for R) (CalcEF – Van Hentenrick)
  - **scope** :
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, IntegerVariable uppBound, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable[] usages, IntegerVariable consumption, IntegerVariable capacity, String... options)
    * cumulative(String name, TaskVariable[] tasks, IntegerVariable[] heights, IntegerVariable consumption, IntegerVariable capacity, String... options)

- `SettingType.`OVERLOAD_CHECKING or `"cp:unary:oc"`

  - **goal** : overload checking rule (O(n*log(n)), Vilim), also known as task interval
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- `SettingType.`NF_NL or `"cp:unary:nfnl"`

  - **goal** : NotFirst/NotLast rule (O(n*log(n)), Vilim) (recommended).
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)

* disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
* disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
* disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- SettingType.`DETECTABLE_PRECEDENCE` or `"cp:unary:dp"`

  - **goal** : Detectable Precedence rule (O(n*log(n)), Vilim).
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- SettingType.`EDGE_FINDING_D` or `"cp:unary:ef"`

  - **goal** : disjunctive Edge Finding rule (O(n*log(n)), Vilim) (recommended).
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- SettingType.`DEFAULT_FILTERING` or `"cp:unary:df"`

  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint.
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- SettingType.`VILIM_FILTERING` or `"cp:unary:df"`

  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint.
  - **scope** :
    * disjunctive(TaskVariable[] tasks, String... options)
    * disjunctive(String name, TaskVariable[] tasks, String... options)
    * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

- SettingType.`SINGLE_RULE_FILTERING` or `"cp:unary:srf"`

  - **goal** : use filtering algorithm proposed by Vilim. nested loop, each rule is applied until it reach it fixpoint. A single filtering rule (debug only).

- **scope** :
    - * disjunctive(TaskVariable[] tasks, String... options)
    - * disjunctive(String name, TaskVariable[] tasks, String... options)
    - * disjunctive(TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, String... options)
    - * disjunctive(String name, TaskVariable[] tasks,IntegerVariable[] usages, IntegerVariable uppBound, String... options)

8.1.4   Options for solvers:

- `Options.`NO_OPTION, `""`, or *empty argument*

- `Options.`S_MULTIPLE_READINGS or `"cp:multiple_readings"`

    - **goal** : Allow a solver to read a model more than one time. *In that case, the redundant constraints for scheduling must be posted explicitly.*
    - **scope** : `CPSolver`

Part III

Extras

Chapter 9

Choco and Visu

9.1 Why?

Since few months, it has seemed more and more evident for us that CHOCO needed a way to visualize dynamically the resolution of a problem. We wanted that visualization to be open, easy to use and not static. Now, you will find a new package on **Choco 2.0.1** (the actual beta version) named *visu*.

9.2 The visu package

The *visu* package contains objects to define a visualization of the resolution, domain reduction, constraints propagation, etc.

Figures 9.1 depicts the class diagram of the visu package (*powered by BOUML*):

9.3 Steps to use the Visu

Only one Visu can be linked to one Solver.

We are going to see a short example of Visu use, based on Sudoku problem. In our modeling, variables are cells of a sudoku grid, represented by the matrix *rows*. We want to define a standard visualization where a variable is displayed on a line. Its name is written, and the domain is viewed as an array of colored square. That representation is known in CHOCO as a *FULLDOMAIN* representation.

9.3.1 Visu creation

The first step is to create the Visu object, which is basically a frame with components. We use the static constructor defined in `Visu.java`:

- `Visu.createFullVisu()`: build a Visu object with default minimum size (width 480 px and heigth 640 px), with *next*, *play,pause* buttons and the break length slider.

- `Visu.createFullVisu(int width, int height)`: build a Visu object with user defined minimum size (width *width* px and heigth *height* px), with *next*, *play,pause* buttons and the break length slider.

- `Visu.createVisu(VisuButton... buttons)`: build a Visu object with default minimum size (width 480 px and heigth 640 px), with *buttons* buttons and the break length slider.

- `Visu.createVisu(int width, int height, final VisuButton... buttons)`: build a Visu object with user defined minimum size (width *width* px and heigth *height* px), with *buttons* buttons and the break length slider if necessary (at least, if there.is one button).

Parameter *buttons* is an array of VisuButton that can take one of the following values: *NEXT*, *PLAY*. *NEXT* add the *next* button to the frame and the slider, *PLAY* add the *play* and *pause* buttons and the slider.

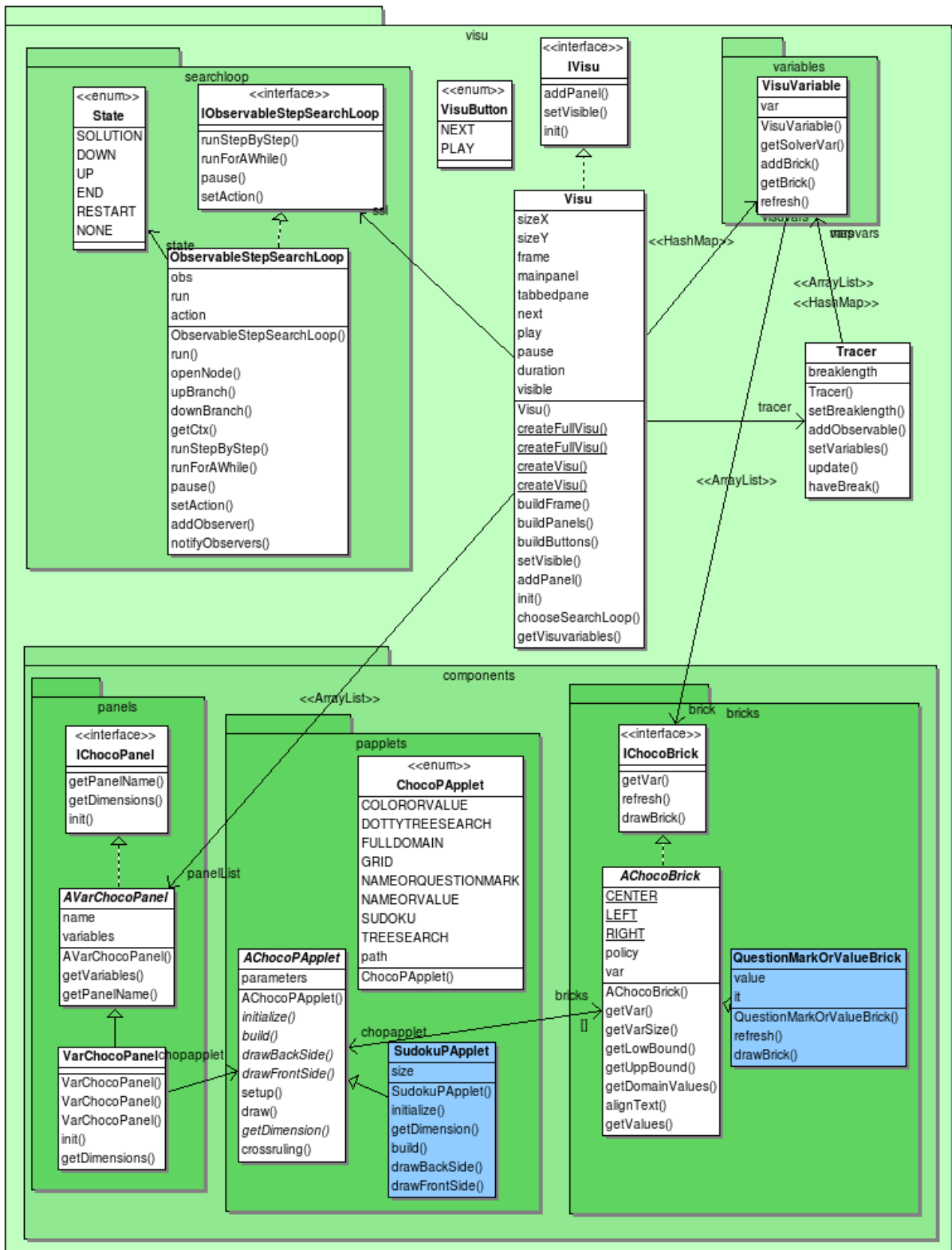We want to create a simple full Visu:

Figure 9.1: Visu classes diagram. The blue classes are examples of implementation and inheritence.

```
Visu v = Visu.createVisu();
```

9.3.2 Adding panel

Now the frame is defined, we have to add a component: a `VarChocoPanel`. It is a specified panel, added to a `TabbedPane`, where one visualization (a `ChocoPApplet`) can be put. A `ChocoPApplet` can be defined in two ways: an existing one, or a user defined one. Constructors of `VarChocoPanel` are:

- `VarChocoPanel(final String name, final Variable[] x, final ChocoPApplet applet, final Object params)`: to add a predefined `ChocoPApplet`. *params* can be null, except for *applet=DOTTYTREESEARCH* (see below).

- `VarChocoPanel(final String name, final Variable[] x, final Class appletclass, Object params)`: like previous, but `ChocoPApplet` is replaced by *class* which is the class name of the user's `ChocoPApplet`. Recommanded for use of user's ChocoPApplet.

- `VarChocoPanel(final String name, final Variable[] x, final String appletpath, Object params)`: like previous, but `ChocoPApplet` is replaced by *path* which is the path of the user's `ChocoPApplet` in the project.

Existing ChocoPApplet

Few ChocoPApplet are defined in Choco:

- **COLORORVALUE** : draw an applet where variables are in columns and where their value is displayed with a colored square (blue: not instantiated, green: instantiated),

- **DOTTYTREESEARCH** : specific applet, which do not display anything, but a *screensaver*. It builds a dot file (name given in parameters) with nodes of the tree search, to represent the tree search. The paramaters are :
  - *filename* (`String`) : output file name
  - *nbMaxNode* (`int`): size limit of the tree seach. If there is more than *nbMaxNode* nodes, the dot file will not be printed. The number of nodes has an impact on the file size
  - *watch* (`Var`) : the variable to optimize. Can be `null` if no optimization is performed.
  - *maximize* (`Boolean`) : indicating wether the optimization is a maximization (if set to `true`) or a minimization (if set to `false`). Can be `null` if no optimization is performed.
  - *restart* (`Boolean`) : indicating wether the search can restart (is set to `true`) or not (if set to `false`). Can be `null` if no optimization is performed.

- **FULLDOMAIN** : draw an applet where variables are in columns. Each line is build with a variable name and a set of colored square (blue: not instantiated, green: instantiated) representing each value of the domain.

- **GRID** : draw an applet with a simple grid, where each cells contains the value of a variable (question mark or value).

- **NAMEORQUESTIONMARK** : draw an applet where a variables are displayed on columns, by a question mark (if not instanciated) or its value (if instanciated).

- **NAMEORVALUE** : draw an applet where a variables are displayed on columns, by its name (if not instanciated) or its value (if instanciated).

- **SUDOKU** : specific applet, draw a sudoku grid where each cell represents the value of a variable or a question mark.

- **TREESEARH** : draw the dynamique construction of the tree search.

To add a panel where one of that ChocoPApplet will be drawn, use the following code:

```
Visu v = Visu.createVisu(
v.addPanel(new VarChocoPanel("Grid", vars, GRID, null));
v.addPanel(new VarChocoPanel("TreeSearch", vars, TREESEARCH, null));
v.addPanel(new VarChocoPanel("Dotty", vars, DOTTYTREESEARCH,
           new Object[]{"/home/choco/treesearch.dot", 100, null, null, null}));
```

User ChocoPApplet

UNDER DEVELOPMENT

9.4   Examples

UNDER DEVELOPMENT

Chapter 10

Sudoku and Constraint Programming

10.1 Sudoku ?!?

Figure 10.1: A sudoku grid

Everybody knows those grids that appeared last year in the subway, in wating lounges, on colleague's desks, etc. In Japanese *su* means digit and *doku*, unique. But this game has been discovered by an American ! The first grids appeared in the USA in 1979 (they were hand crafted). Wikipedia tells us that they were designed by Howard Garns a retired architect. He died in 1989 well before the success story of sudoku initiated by Wayne Gould, a retired judge from Hong-Kong. The rules are really simple: a 81 cells square grid is divided in 9 smaller blocks of 9 cells (3 x 3). Some of the 81 are filled with one digit. The aim of the puzzle is to fill in the other cells, using digits except 0, such as each digit appears once and only once in each row, each column and each smaller block. The solution is unique.
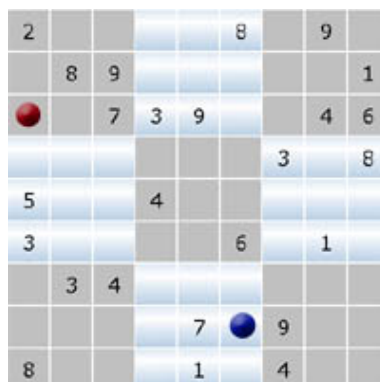
10.1.1 Solving sudokus

Many computer techniques exist to quickly solve a sudoku puzzle. Mainly, they are based on backtracking algorithms. The idea is the following: give a free cell a value and continue as long as choices remain consistent. As soon as an inconsistency is detected, the computer program backtracks to its earliest past choice et tries another value. If no more value is available, the program keeps backtracking until it can go forward again. This systematic technique make it sure to solve a sudoku grid. However, no human player plays this way: this needs too much memory !

> see Wikipedia for a panel of solving techniques.

10.2   Sudoku and Artificial Intelligence

Many techniques and rules have been designed and discovered to solve sudoku grids. Some are really simple, some need to use some useful tools: pencil and eraser.

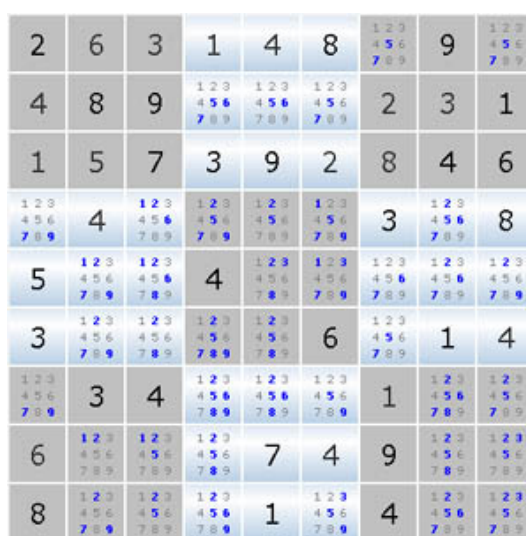### 10.2.1   Simple rules: single candidate and single position

Figure 10.2: Simple rules: single candidates and single position

Let consider the grid on Figure 10.2 and the cell with the red dot. In the same line, we find: 3, 4, 6, 7, and 9. In the same column: 2, 3, 5, and 8. In the same block: 2, 7, 8, and 9. There remain only one possibility: **1**. This is the **single candidate** rule. This cell should be filled in with **1**.

Now let consider a given digit: let's say 4. In the block with a blue dot, there is no 4. Where can it be ? The 4's in the surrounding blocks heavily constrain the problem. There is a **single position** possible: the blue dot. This another simple rule to apply.

Alternatively using these two rules allows a player to fill in many cells and even solve the simplest grids. But, limits are easily reached. More subtle approaches are needed: but an important tool is now needed ... an eraser !

10.2.2   Human reasoning principles

Figure 10.3: Introducing marks

Many techniques do exist but a vaste amount of them rely on simple principles. The first one is: do not try to find the value of a cell but instead focus on values that **will never be assigned** to it. The

space of possibility is then reduced. This is where the eraser comes handy. Many players marks the remaining possibilities as in the grid on the left.

Using this information, rather subtle reasoning is possible. For example, consider the seventh column on the grid on the left. Two cells contain as possible values the two values 5 and 7. This means that those two values cannot appear elsewhere in that very same column. Therefore, the other unassigned cell on the column can only contain a 6. We have *deduced* something.

This was an easy to spot inference. This is not always the case. Consider the part of the grid on the right. Let us consider the third column. For cells 4 and 5, only two values are available: 4 and 8. Those values cannot be assigned to any other cell in that column. Therefore, in cell 6 we have a 3, and thus and 7 in cell 2 and finally a 1 in cell 3. This can be a very powerful rule.

Such a reasoning (sometimes called *Naked Pairs*) is easily generalized to any number of cells (always in the same region: row, column or block) presenting this same configuration. This local reasoning can be applied to any region of the grid. It is important to notice that the inferred information can (and should) be used from a region to another.

The following principles of *human* reasoning can be listed:

- reasoning on *possible* values for a cell (by erasing impossible ones)

- systematically applying an evolved local reasoning (such as the *Naked Pairs* rule)

- transmitting inferred information from a region to another related through a given a set of cells

10.2.3 Towards Constraint Programming

Those three principles are at the core of **constraint programming** a recent technique coming from both *artificial intelligence* and *operations research*.

- The first principle is called **domain reduction** or *filtering*

- The second considers its region as a **constraint** (a relation to be verified by the solution of the problem): here we consider an *all different* constraint (all the values must be different in a given region). Constraints are considered **locally** for reasoning

- The third principle is called **propagation**: constraints (regions) communicate with one another through the available values in variables (cells)

Constraint programming is able to solve this problem as a human would do. Moreover, a large majority of the rules and techniques described on the Internet amount to a well-known problem: the alldifferent problem. A **constraint solver** (as **Choco**) is therefore able to reason on this problem allowing the solving of sudoku grid as a human would do although it has not be specifically designed to.

Ideally, iterating local reasoning will lead to a solution. However, for exceptionnaly hard grids, an enumerating phase (all constraint solvers provide tools for that) relying on backtracking may be necessary.

10.3   See also

- SudokuHelper a sudoku solver and helper applet developed with *Choco*.

- PalmSudoku a rather complete list of rules and tips for solving sudokus

Bibliography

[Beldiceanu and Carlsson, 2002] Beldiceanu, N. and Carlsson, M. (2002). A new multi-resource `cumulatives` constraint with negative heights. In *International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 63–79. Springer-Verlag.

[Beldiceanu et al., 2008] Beldiceanu, N., Flener, P., and Lorca, X. (2008). Combining tree partitioning, precedence, and incomparability constraints. *Constraints*, 13(4):459–489.

[Bessière et al., 2005a] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2005a). *Among*, *common* and *disjoint* Constraints. In Carlsson, M., Fages, F., Hnich, B., and Rossi, F., editors, *Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2005)*, pages 223–235.

[Bessière et al., 2005b] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2005b). Filtering algorithms for the `nvalue` constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'05)*, volume 3524 of *LNCS*, pages 79–93. Springer-Verlag.

[Bessière et al., 2006] Bessière, C., Hebrard, E., Hnich, B., Kızıltan, Z., and Walsh, T. (2006). *Among*, *common* and *disjoint* Constraints. In Hnich, B., Carlsson, M., Fages, F., and Rossi, F., editors, *Recent Advances in Constraints, Joint ERCIM/Colognet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP05; Uppsala, Sweden, June 2005; Revised Selected and Invited Papers*, volume 3978 of *LNAI*, pages 28–43. Springer-Verlag.

[Bleuzen-Guernalec and Colmerauer, 1997] Bleuzen-Guernalec, N. and Colmerauer, A. (1997). Narrowing a block of sortings in quadratic time. In *International Conference on Principles and Practice of Constraint Programming (CP'97)*, volume ? of *LNCS*, pages ?–? Springer-Verlag.

[Carlsson and Beldiceanu, 2002] Carlsson, M. and Beldiceanu, N. (2002). Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, Swedish Institute of Computer Science.

[Demassey et al., 2006] Demassey, S., Pesant, G., and Rousseau, L.-M. (2006). A `Cost-Regular` based hybrid column generation approach. *Constraints*, 11(4):315–333. Special issue following CPAIOR'05.

[Frisch et al., 2002] Frisch, A. M., Hnich, B., Kızıltan, Z., Miguel, I., and Walsh, T. (2002). Global constraints for lexicographic orderings. In *International Conference on Principles and Practice of Constraint Programming (CP'2002)*, volume 2470 of *LNCS*, pages 93–108. Springer-Verlag.

[Frisch et al., 2003] Frisch, A. M., Hnich, B., Kızıltan, Z., Miguel, I., and Walsh, T. (2003). Multiset ordering constraints. In *18th International Joint Conference on Artificial Intelligence (IJCAI'03)*.

[López-Ortiz et al., 2003] López-Ortiz, A., Quimper, C.-G., Tromp, J., and van Beek, P. (2003). A fast and simple algorithm for bounds consistency of the `alldifferent` constraint. In *18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 245–250.

[Mehlhorn and Thiel, 2000] Mehlhorn, K. and Thiel, S. (2000). Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *International Conference on Principles and Practice of Constraint Programming (CP'00)*, volume ? of *LNCS*, pages ?–? Springer-Verlag.

[Menana and Demassey, 2009] Menana, J. and Demassey, S. (2009). Sequencing and counting with the `multicost-regular` constraint. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *LNCS*, pages 178–192. Springer-Verlag.

[Pesant, 2004] Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 482–495. Springer-Verlag.

[Quimper et al., 2003] Quimper, C.-G., van Beek, P., López-Ortiz, A., Golynski, A., and Sadjad, S. B. (2003). An efficient bounds consistency algorithm for the *global-cardinality* constraint. In *International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, pages 600–614. Springer-Verlag.

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in CSP. In *12th National Conference on Artificial Intelligence (AAAI'94)*, pages 362–367.

[Régin, 1996] Régin, J.-C. (1996). Generalized arc consistency for `global-cardinality` constraint. In *14th National Conference on Artificial Intelligence (AAAI'96)*, pages 209–215.

[Shaw, 2004] Shaw, P. (2004). A constraint for bin packing. In *International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 648–662. Springer-Verlag.

[Trick, 2003] Trick, M. A. (2003). A dynamic programming approach for consistency and propagation for knapsack constraints. In Netherlands, S., editor, *Annals of Operations Research*, volume Volume 118, Numbers 1-4 / feb. 2003, pages 73–84. Springer Netherlands.

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. `http://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access

to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties?for example, statements of peer review or that the text has

been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include

the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.3
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.