User guide

From Choco

Contents

An introduction to 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 (like optimization problems), 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 foundational issues, with implementation aspects and with new applications of constraint programming.

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 other object, say triangle, and introduce another constraint, say *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*, typicaly 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.

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 sudoku and constraint programming)
```

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 his problem without being interested in the way the problem is solved.

Modelling a Constraint Satisfaction Problem

Constraint programming allows to solve combinatorial problems modelized by a constraint satisfaction problem (CSP). Formally, a CSP is defined by a triplet (X,D,C) such as:

1. **Variables**: $X = \{X_1,X_2,...,X_n\}$ is the set of variables of the problem.
2. **Domain**: D is a function which associates to each variable X_i its domain $D(X_i)$, i.e., the possible values that can be affected to X_i.
3. **Constraints**: $C = \{C_1,C_2,...,C_k\}$ is the set of constraints. Each constraint C_j is a relation between a subset of variables which restricts the domain of each one.

A constraint is satisfied if the tuple of the values of its variables belongs to the relation describing the constraint. Thus, solving a CSP consists in finding a tuple on the set of variables such that each constraint is satisfied. Moreover, several types of variables can be distinguished:

1. **Integer variables** are variables described on domains containing integer values.
2. **Set variables** are variables described on domains containing sets of values.
3. **Real variables** are variables described on continous domains and generaly use intervals to represent values.

Examples

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

Example 1: the n-Queen's 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-Queen's 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-Queen's problem can be modelized by a CSP in the following way:

1. **Variables**: $X = \{X_i \mid i \text{ is an integer in } [1,n]\}$.
2. **Domain**: for all X_i in X, $D(X_i) = \{j \mid j \text{ is an integer in } [1,n]\}$.
3. **Constraints**: the set of constraints is defined by the union of the three following constraints,
   - **queens have to be on distinct lines**:
     - $C_{lines} = \{X_i \neq X_j \mid i \text{ and } j \text{ are two distincts integer in } [1,n]\}$.
   - **queens have to be on distinct diagonals**:
     - $C_{diag1} = \{X_i \neq X_j+j-i \mid i \text{ and } j \text{ are two distincts integer in } [1,n]\}$.
     - $C_{diag2} = \{X_i \neq X_j+i-j \mid i \text{ and } j \text{ are two distincts integer in } [1,n]\}$.

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 different triplets are different.

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

1. let t = n*(n-1)/6.
2. **Variables**: $X = \{X_i \mid i \text{ an integer in } [1,t]\}$.
3. **Domain**: for all X_i in X, $D(X_i) = \{1,...,n\}$.
4. **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], $|\text{intersection}(X_i,X_j)| <= 1$.

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:

1. **Variables**: x,y and z.
2. **Domain**:]-oo;+oo[.
3. **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.

Problem Modeling with Choco

Creating a problem

The central element of a choco program is the Problem object.

```
AbstractProblem myPb = new Problem();
```

The class Problem is a factory where all API to create variables and post constraints is available. Variables and constraints are therefore related to the Problem object as well as the access to the solution found.

Variables and domains

Actually, three main kind of variables exist :

1. **IntDomainVar** : It describes discrete domains where values are integers.
   - **EnumIntVar** : It corresponds to enumerated domains and should be used when discrete and quite small domains are needed.
   - **BoundIntVar** : Such domains are represented by their lower and upper bounds (propagation is only performed on the bounds). One can use them when large domains are needed.
2. **RealVar** : It describes continous domain and use intervals to represent values.
3. **SetVar** : It describes discrete set domains where a value of a variable is a set. Set vars are encoded with two classical bounds : the union of the all set of possible values called the envelope and the intersection of all set of possible values called the kernel.

Once the Problem has been created, variables are created through factories available on the Problem instead of the classical java constructor. The use of factories allows to redefine them in a specific Problem (as done for the PalmProblem) and ensures that constraints and variables types will remain compatible. The following example of code show how to create a finite domain variable:

```
IntDomainVar v1 = myPb.makeEnumIntVar("var1", 1, 10);
```

`v1` is an enumerated variable which is called `var1` and has a discrete domain from 1 to 10. It has been created for the problem `myPb`.

Integer Variables

1. `makeBoundIntVar(String s, int lb, int ub)` : creates a finite domain variable whose domain is approximated by bounds `(lb .. ub)`, with name `s`.
2. `makeEnumIntVar(String s, int lb, int ub)` : creates a finite domain variable with domain `(lb .. ub)`, with name `s`.

BoundIntVar 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 the update of the bounds. Value removals between the bounds are therefore ignored. The level of consistency achieved by most constraints on these variables is called bound consistency. On the contrary, the domain of an EnumIntVar is explicitly represented and every value is considered while pruning. Basic constraints are therefore often able to achieve arc-consistency on EnumIntVar (except for NP global constraint such as the cumulative constraint). Remember that switching from an EnumIntVar to a BoundIntVar decrease the level of propagation achieved by the system.

The state of an IntDomainVar can be accessed through the main following public methods on the IntVar class:

- `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 val)`: checks if the value val is contained in the domain of the variable.
- `getDomainSize()`: returns the current size of the domain.

The domain of an IntVar can be modified through the main following public methods: such operations are subject to the backtrack mechanism.

- `setInf()`: set the lower bound of the variable.
- `setSup()`: set the upper bound of the variable.
- `setVal()`: set the value of the variable.

Set Variables

Set variables are still under development but a bit of api is still available.

1. `makeSetVar(String name, int l, int u)` : creates a set domain variable with name s where l correponds to the lower bound of the inital enveloppe and b the upper bound.

Set variables are high level modelisation tool. It allows to represent variables whose values are sets. 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 (as the BoundIntVar) corresponding to the intersection of all possible sets (called the kernel) and the union of all possible sets (called the enveloppe) which are the possible candidats value for the variable. The consistency achivied on set variables is therefore a kind of bound consistency.

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

- `isInDomainKernel(int x)` : checks if a value x is contained in the current kernel.
- `isInDomainEnveloppe(int x)` : checks if a value x 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 integer int[] containing the current domain.

The domain of a SetVar can be modified through the main following public methods(ContradictionException can be thrown in case of an inconsistent change) :

- `setValIn(int x)`: set a value inside the kernel.
- `setValOut(int x)`: set a value outside the kernel.
- `setVal(int[] val)`: set the value of the variable.

Real Variables

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

1. `makeRealVar(String s, double lb, double ub)` : creates a continous domain variable whose domain is considered as an interval [lb,ub], with name s.

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

- `getInf()`: returns the lower bound of the variable (a double).
- `getSup()`: returns the upper bound of the variable (a 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.

Constraints

Constraints are represented by dedicated objects organized in a class hierarchy. It 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 modification.

A constraint is stated to a problem by using the method post available on the Problem object : post(Constraint c). Creating a constraint and adding it to the constraint network can be done using the Problem api. For example, adding a constraint of difference between two variables is simply written as follow:

```
myPb.post(myPb.neq(vars1, vars2));
```

Constraints on Integer variables

The constraints available with choco are arithmetic constraints (equality, difference, comparisons and linear combination), user-defined binary constraints (AC4,AC3, ...), boolean operators (or, and, implies, ...) among constraints as well as some global constraints.

Basic constraints

The simplest constraints are comparisons which are defined over expressions of variables such as linear combinations. The following comparison constraints can be accessed through the Problem API:

- `neq(IntExp v1, IntExp v2)` : v1 != v2.
- `eq(IntExp v1, IntExp v2)` : v1 = v2.
- `leq(IntExp v1, IntExp v2)` : v1 <= v2.
- `lt(IntExp v1, IntExp v2)` : v1 < v2.

To construct complex expressions of variables, simple operators can be used:

- `minus(IntExp exp1, IntExp exp2)` : exp1 - exp2.
- `plus(IntExp exp1, IntExp exp2)` : exp1 + exp2.
- `mult(int coef, IntExp exp)` : coef * exp.
- `scalar(int[] coef, IntVar[] vars)`: coef[1]*vars[1] + ... + coef[n]*vars[n].
- `sum(IntVar[] vars)`: vars[1] + ... + vars[n].

Arbitrary constraints (in extension)

Choco supports the statement of constraints defined by arbitrary relations. It offers the possibility of stating binary constraints with several AC Algorithm and also n-ary constraints with a weaker form of propagation.

Binary constraints

The relation defines feasible or infeasible pairs of values for the two variables involved in the constraint.

Relations may be defined by two means:

1. `Tables` : specifying those pairs of value for which the constraints is satisfied/unsatisfied.
2. `Predicates` : specifying the method to be called in order to check whether a pair of value is feasible or not.

On the one hand constraints based on tables of values may become rather memory consuming in case of large domains, although relation tables may be shared by different constraints. On the other hand, predicate constraints require little memory as they do not cache thruth 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. The creation of a constraint for a relation can be done through the following Problem API :

1. `makePairAC(IntVar v1, IntVar v2, ArrayList pairs, boolean feas, int ac).`
2. `makePairAC(IntVar v1, IntVar v2, boolean[][] pairs, boolean feas, int ac).`
3. `relationPairAC(IntVar v1, IntVar v2, BinRelation pairs, int ac).`

Parameter feas indicates whether the relation models feasible pairs of values or infeasible one (default is infeasible). Parameters pairs contains the definition of the relation. A list of int[] of size 2 in the first case, a boolean[][] in the second case or as a BinRelation in the last case. Finally parameter ac selects the algorithm for enforcing arc-consistency (default ac = 2001). Supported values for this parameter are :

1. 3 for AC3 algorithm (searching from scratch for supports on all values)
2. 4 for AC4 algorithm (maintaining a count of supports for each value)
3. 2001 for the AC2001 algorithm (maintaining the current support of each value)

The definition of a binary relation based on a predicate can be done by inheriting from the CouplesTest class. Have a look on the following example :

```
public class MyInequality extends CouplesTest{
  public boolean checkCouple(int x, int y) {
    return x != y;
  }
}
```

You can then state a constraint as the following :

```
pb.post(pb.relationPairAC(v1, v2,new MyInequality()));
```

The complete Problem API allow to easily create binary constraint :

1. infeasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
2. feasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
3. relationPairAC(IntVar v1, IntVar v2, BinRelation binR)
4. ...

Nary constraints

The situations for binary constraints is extended to the case of relations involving more than two variables, upto a significant difference from the propagation point of view: The propagation engine maintains arc-consistency for binary constraints throughout the solving process, while for n-ary constraints, it uses a weaker propagation mechanism with a forward checking algorithm. The API for creating such constraints is the

following ones:

1. makeTupleFC(IntVar[] vs, ArrayList tuples, boolean feas)
2. relationTuple(IntVar[] vs, LargeRelation rela)

Defining a specific n-ary relation without storing the tuples can be done as on the following example :

```
public 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;
       }

   }
```

Otherwise, the tuples are given in an ArrayList as int[] table given the compatible/incompatible values. One can then state the constraint to the problem :

```
pb.post(pb.relationTuple(new IntVar[]{x, y, z}, new NotAllEqual()))
```

Finally, the structure of the Consistency relations can be seen in more details on the following picture :

Advanced constraints

Choco includes several global constraints. Those constraints allows to filter efficiently some inconsistant values. For instance, if several variables should be affected to different values, using a global constraint can offer some additionnal filtering rules (for instance a should be in [1,4], b in [1,4], c in [3,4] and d in [3,4], then one can deduce that a and b cannot be instantiated to 3 or 4; such rule cannot be infered by simple binary constraints). Here are described some of those constraints :

1. **pb.allDifferent(IntVar[] vars)** creates a constraint ensuring that all pairs of variable have distinct values (which is useful for some matching problems); Notice that the filtering algorithm used will depend on the nature (EnumIntVar or BoundIntVar) of the table of variable `vars`. In case of EnumIntVar, the constraint refers to the alldifferent of régin AAAI94 : "A filtering algorithm for constraints of difference in CSPs". In case of BoundIntVar, a dedicated algorithm for bound propagation is used (see Lopez-Ortiz'03 :" A fast and simple algorithm for bounds consistency of the alldifferent constraint"). Moreover, it is possible to avoid the use of a global filtering algorithm by using a slight different api with a boolean `global` to false : **pb.allDifferent(IntVar[] vars, boolean global)** so by default this boolean is set to true.
2. **pb.occurrence(IntVar[] vars, int value, IntVar occurrence)** creates a constraint to ensure that *occurrence* will be instantiated to the number of occurrences of value in the list of variables vars; this is a specialization of the following constraint;
3. **pb.globalCardinality(IntVar[] vars, int[] low, int[] up)** creates a constraint ensuring that the number of occurrences of the value 1 in all the variables vars is between low[0] and up[0], and generallay the number of occurrences of the value i in vars is between low[i-1] and up[i-1].
4. **pb.nth(IntVar index, int[] values, IntVar val)** allow to state the well known Element constraint. This constraint ensures that values[index] = val where `index` and `val` are variables.
5. **pb.nth(IntVar index, IntVar[] values, IntVar val)** allow to state an Element constraint where the values are variables.
6. **pb.cumulative(IntVar[] starts, IntVar[] ends, IntVar[] durations, int[] heights, int Capa)** : Given a set of tasks defined by their starting dates, ending dates, durations and consumptions/heights, the cumulative ensures that at any time t, the sum of the heights of the tasks which are executed at time t does not exceed a given limit C (the capacity of the ressource). The notion of task does not exist yet in choco. The cumulative takes therefore as input three arrays of integer variables (of same size n)

denoting the starting, ending, and duration of each task. The heights of the tasks are considered constant and given via an array of size n of positive integers. The last parameter Capa denotes the Capacity of the cumulative (of the ressource). The implementation is based on the paper of Bediceanu and al : "A new multi-resource cumulatives constraint with negative heights" in CP02. WARNING : Keep in mind that the task is active in the interval [start,end-1] or [start,end[because we always have start + duree = end. This is the usual definition of start, end, and duration in scheduling.

7. **pb.lex(IntVar[] x, IntVar[] y)** : enforces a strict lexicographic ordering on two vectors of integer variables x <_lex y with x = <x_0, ..., x_n>, and y = <y_0, ..., y_n> : x $<_{lex}$ y. Implementation refers to "Global Constraints for Lexicographic Orderings" (Frisch and al) of CP'02. **lexeq** denotes the relation x $<=_{lex}$ y.

Some other global constraints can be added to Choco in future releases. One can find all the API and constraints available on the Javadoc API.

Boolean composition of constraints

Constraints on Set variables

Choco supports the statement of constraints among sets :

```
pb = new Problem();
pb.post(mod.eqCard(vars[i],3));
```

The following set constraints are available :

1. member(SetVar sv1, int val): states that the variable sv1 contains value val.
2. notMember(SetVar sv1, int val): states that the variable sv1 does not contain value val.
3. setDisjoint(SetVar sv1, SetVar sv2): states that sv1 and sv2 are disjoint sets ; e.g. that sv1 and sv2 contain no common values.
4. setInter(SetVar sv1, SetVar sv2, SetVar inter): states that the inter set variable is the intersection of set variables sv1 and sv2 ; e.g. states that inter contains exactly those values contained in both sets sv1 and sv2.
5. eqCard(SetVar sv, int val): states that the cardinality of the set variable is equal to value val.
6. geqCard(SetVar sv, int val): states that the cardinality of the set variable is greater or equal equal than value val.
7. leqCard(SetVar sv, int val): states that the cardinality of the set variable is equal than value val.

To deal with integer variables, the following mixed constraint are available :

1. member(SetVar sv1, IntVar var).
2. notMember(SetVar sv1, IntVar var).
3. eqCard(SetVar sv, IntVar iv).
4. geqCard(SetVar sv, IntVar iv).
5. leqCard(SetVar sv, IntVar iv).

Constraints on Real variables

Examples

We provide now the complete choco model for the three examples described in section 1.

Example 1: the n-Queen's problem with Choco

This first model for the nqueen problem only involves binary constraints of differences between integer

variables. One can immediatly recognize the 4 main elements of any choco code. First of all, create the problem object. Then create the variables by using the factory methods of the problem (One variable per queen giving the row (or the column) where the queen will be placed). Finally, post the constraints and solve the problem.

```
//1- Create the problem
AbstractProblem pb = new Problem();

//2- Create the variables
IntVar[] queens = new IntVar[n];
for (int i = 0; i < n; i++) {
   queens[i] = pb.makeEnumIntVar("Q" + i, 1, n);
}

//3- Post constraints
for (int i = 0; i < n; i++) {
   for (int j = i + 1; j < n; j++) {
     int k = j - i;
     pb.post(pb.neq(queens[i], queens[j]));
     pb.post(pb.neq(queens[i], pb.plus(queens[j], k)));  // diagonal constraints
     pb.post(pb.neq(queens[i], pb.minus(queens[j], k))); // diagonal constraints
   }
}

//4- Search for all solutions
pb.solveAll();

//5- Print the number of solution found
System.out.println("NbSol: " + pb.getSolver().getNbSolutions());
```

Example 2: the ternary Steiner problem with Choco

Set variables are illustrated on the ternary steiner system problem. Let's recall that a ternary Steiner system of order n is a set of triplets of distinct elements taking their values between 1 and n, such that all the pairs included in two different triplets are different. See http://mathworld.wolfram.com/SteinerTripleSystem.html for details. The problem is entirely modelled using set variables and set constraints. There is only one difference between the previous code for integer variables : the search for solutions. The search can be simply controlled from the problem object via a call to solve, solveAll, nextSolution or via the solver object to control it in a finer way. The use of the Solver object will be explained in details in the next section about Search and Branching.

```
//1- Create the problem
AbstractProblem pb = new Problem();
int m = 7;
int n = m * (m - 1) / 6;

//2- Create Variables
SetVar[] vars = new SetVar[n]; // A variable for each set
SetVar[] intersect = new SetVar[n * n]; // A variable for each pair of sets

for (int i = 0; i < n; i++)
    vars[i] = pb.makeSetVar("set " + i, 1, n);
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    intersect[i * n + j] = pb.makeSetVar("interSet " + i + " " + j, 1, n);

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

//4- Search for a solution
pb.getSolver().setFirstSolution(true);
```

```
pb.getSolver().generateSearchSolver(pb);
pb.getSolver().addGoal(new AssignSetVar(new MinDomSet(pb, vars), new MinEnv(pb)));
pb.getSolver().launch();

//5- print the solution
for (int i = 0; i < n; i++) {
  System.out.println(vars[i].pretty());
}
```

Example 3: the CycloHexane problem with Choco

Real variables are illustrated on the problem of finding the 3D configuration of a cyclohexane molecule. It is described with a system of three non linear equations :

```
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
```

It has been taken from the Elisa project (LINA) examples.

```
//1- Create the problem
AbstractProblem pb = new Problem();
pb.setPrecision(1e-8);

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

//3- Create and post the constraints
RealExp exp1 = pb.plus(pb.mult(pb.power(y, 2), pb.plus(pb.cst(1.0), pb.power(z, 2))),
               pb.mult(z, pb.minus(z, pb.mult(pb.cst(24), y))));

RealExp exp2 = pb.plus(pb.mult(pb.power(z, 2), pb.plus(pb.cst(1.0), pb.power(x, 2))),
               pb.mult(x, pb.minus(x, pb.mult(pb.cst(24), z))));

RealExp exp3 = pb.plus(pb.mult(pb.power(x, 2), pb.plus(pb.cst(1.0), pb.power(y, 2))),
               pb.mult(y, pb.minus(y, pb.mult(pb.cst(24), x))));

Equation eq1 = (Equation) pb.eq(exp1, pb.cst(-13));
eq1.addBoxedVar(y);
eq1.addBoxedVar(z);

Equation eq2 = (Equation) pb.eq(exp2, pb.cst(-13));
eq2.addBoxedVar(x);
eq2.addBoxedVar(z);

Equation eq3 = (Equation) pb.eq(exp3, pb.cst(-13));
eq3.addBoxedVar(x);
eq3.addBoxedVar(y);

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

//4- Search for all solution
Solver solver = pb.getSolver();
solver.setFirstSolution(true);
solver.generateSearchSolver(pb);
solver.addGoal(new AssignInterval(new CyclicRealVarSelector(pb), new RealIncreasingDomain()));
solver.launch();

//5- print the solution found
System.out.println("x " + x.getValue());
System.out.println("y " + y.getValue());
System.out.println("z " + z.getValue());
```

Search and branching

Search for one or all solution

Once the problem is modelled and created thanks to the API described in previous sections, one may want to solve it ! If only one solution is needed, this is quite easy. The following code solves the problem and displays the solution:

```
if (pb.solve() == Boolean.TRUE) {
 for(int i = 0; i < pb.getNbIntVars(); i++) {
  System.out.println(pb.getIntVar(i) + " = " + ((IntDomainVar) pb.getIntVar(i)).getVal());
 }
}
```

Notice that the solve() method returns a Boolean object instead of a primitive boolean because its value may be null meaning that a limit has been reached. If one wants several solutions, the incremental solve API can be used : nextSolution search for another solution in the search tree :

```
if (pb.solve() == Boolean.TRUE) {
  do {
    for(int i = 0; i < pb.getNbIntVars(); i ++) {
      System.out.println(pb.getIntVar(i) + " = " + ((IntDomainVar) pb.getIntVar(i)).getVal());
    }
  } while(pb.nextSolution() == Boolean.TRUE);
}
```

The Solver

The Problem is the central element of a choco model as it allows the creation of variables and constraints. But the control of the search process without using predefined tools is made on the Solver class. The following code will do exactly what you get by calling the solve() method but you may access this time to the solver once it has been generated to parameterize it in more details. We will use this piece of code in section 3 and 4 to show how the search space can be controlled in details.

```
Solver s = pb.getSolver();
s.setFirstSolution(true);
s.generateSearchSolver(pb);
// insert here the code to parameterize the solver in details (adding goals, new limits, //etc …)
s.launch();
Boolean isFeasible = pb.isFeasible();
```

Optimization

Optimization is done in Choco according to a variable denoting the objective value. The objective function is then expressed as a constraint over this variable and the rest of the problem. The API concerning optimization proposes to minimize/maximize this objective variable (instead of a call to pb.solve()) :

1. minimize(IntVar obj, boolean restart).
2. maximize(IntVar obj, boolean restart).

Parameter restart is a boolean indicating whether the solver will restart the search after each solution found or if it will keep backtracking from the leaf of the last solution found. Look at the following knapsack example where a scalar product over three variables is maximized :

```
AbstractProblem pb = new Problem();
IntDomainVar obj1 = pb.makeEnumIntVar("obj1",0,7);
IntDomainVar obj2 = pb.makeEnumIntVar("obj1",0,5);
IntDomainVar obj3 = pb.makeEnumIntVar("obj1",0,3);
IntDomainVar cost = pb.makeBoundIntVar("cout",0,1000000);
int capacite = 34;
int[] volumes = new int[]{7,5,3};
int[] energie = new int[]{6,4,2};
// capacity constraint
pb.post(pb.leq(pb.scalar(volumes,new IntVar[]{obj1,obj2,obj3}),capacite));

// objective function
pb.post(pb.eq(pb.scalar(energie,new IntVar[]{obj1,obj2,obj3}),cost));

pb.maximize(c,false);
```

Limiting the search space

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. The API is given on the Solver class:

1. setTimeLimit(int timeLimit): stops the search algorithm after timeLimit milliseconds have been spent searching.
2. setNodeLimit(int nodeLimit): stops the search algorithm after nodeLimit nodes have been expanded.

For example, to stop the search after 30 seconds, just add the following line ((before a call to pb.solve()):

```
pb.getSolver().setTimeLimit(30000);
```

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 IGlobalSearchLimit extends Entity {
  /**
   * 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 = pb.getSolver();
s.setFirstSolution(true);
s.generateSearchSolver(pb);
s.getSearchSolver().limits.add(new DepthLimit(s.getSearchSolver(),10));
s.launch();
```

Define your own tree search

A key ingredient of any constraint approach is a clever branching strategy. The construction of the search tree is done according to a serie 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. We will first present in this section how to define your own branching object and how to compose it with other goals. 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).

Building a sequence of branching object

Adding a new goal is made through the problem solver (Solver s = pb.getSolver()) with the addGoal (AbstractBranching b) method of the solver. As for the addition of your own limit, don’t 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. The first two branching are both AssignVar (see next section) but uses two different variable/values selection strategies:

```
Solver s = pb.getSolver();
pb.getSolver().generateSearchSolver(pb);
```

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

Variable/value selection for AssignVar branching

Choco provides means of composing search trees by specifying the heuristics used for selecting variables and values on integer variables in case of AssignVar branchings. An AssignVar branching takes as input the variable and value selection strategy which are based on the following interfaces:

1. IIntVarSelector : Interface for the variable selection
2. IValIterator : Interface that provide a way of describing an iteration scheme on the domain of a variable
3. IValSelector : Interface for a value selection

Default branching heuristics

The default branching heuristic used by Choco is to choose the variable with current minimum domain size first and to take its values in increasing order. The default branchings currently supported by choco are available in the packages choco.integer.search for integer variables (choco.set.search for set variables). Concrete examples of the previous interfaces are the classes MinDomain, MostConstrained, DomOverDeg, RandomIntVarSelector ... If you only want to use one single goal but with customized value and variable heuristics, you can use the API available on the Solver class (before calling the solve() method) as shown on the following example :

```
pb.getSolver().setVarSelector(new RandomIntVarSelector(pb));
```

Changing the values enumeration/selection can be done in the same way:

```
// select the value in increasing order
pb.getSolver().setValIterator(new DecreasingDomain());
// or select a random value
pb.getSolver().setValSelector(new RandomIntValSelector());
```

How to define it own variable selection You may extend this small library of branching schemes and heuristics by defining your own concrete classes of IIntVarSelector, IValIterator and IValSelector. We give here an example of an IntVarSelector with the implementation of a static variable ordering :

```
public class StaticVarOrder implements IIntVarSelector {
  /**
   * the sequence of variables that need be instantiated
  */
  protected IntDomainVar[] vars;

  public StaticVarOrder(IntVar[] 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 the method selectIntVar which belongs to the contract of IIntVarSelector. Once the branching is finished, it returns null and 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.

Beyond Variable/value selection, how to define its own Branching object

A branching object is based on the IntBranching interface where each alternative is labelled with an integer.

```
/**
 * IntBranching objects are specific branching objects where each branch is labelled with an integer.
 * This is typically useful for choice points in search trees
 */
public interface IntBranching extends Branching {

/**
   * selecting the object under scrutiny (that object on which an alternative will be set)
   * @return the object on which an alternative will be set (often  a variable)
   */
   public Object selectBranchingObject();

  /**
   * performs the action, so that we go down a branch from the current choice point
   * @param x the object on which the alternative is set
   * @param i the label of the branch that we want to go down
   */
  public void goDownBranch(Object x, int i) throws ContradictionException;

  /**
   * performs the action, so that we go down up the current branch to the father choice point
   * @param x the object on which the alternative has been set at the father choice point
   * @param i the label of the branch that has been travelled down from the father choice point
   */
  public void goUpBranch(Object x, int i) throws ContradictionException;

  /**
   * Computes the search index of the first branch of the choice point
   * @param x the object on which the alternative is set
   * @return the index of the first branch
   */
  public int getFirstBranch(Object x);

  /**
   * Computes the search index of the next branch of the choice point
   * @param x the object on which the alternative is set
   * @param i the index of the current branch
   * @return the index of the next branch
   */
  public int getNextBranch(Object x, int i);

  /**
   * Checks whether all branches have already been explored at the current choice point
   * @param x the object on which the alternative is set
   * @param i the index of the last branch
   * @return true if no more branches can be generated
   */
  public boolean finishedBranching(Object x, int i);
```

The AssignVar branching typically implements the computation of the branching object (selectBranchingObject()) by delegating it to its VarSelector as well as first and next branch computation are delegated to its Value Selector or Iterator. So in this case, the value chosen is used to label the branch. Finally the AssignVar branching simply implements the goDownBranch(Object x, i) by assigning the value i to variable x (cast in this case as an IntVar) and propagating this choice :

```
  public void goDownBranch(Object x, int i) throws ContradictionException {
    (IntDomainVar) x.setVal(i);
    manager.problem.propagate();
  }

  public void goUpBranch(Object x, int i) throws ContradictionException {
    ((IntDomainVar) x).remVal(i);
    manager.problem.propagate();
  }
```

Notice that in this implementation, we choose to propagate the removal of a value when this value has proved to lead to a failure. We could have chosen to leave the goUpBranch(Object x, int i) empty and to keep branching with the next values.

An example of a dichotomic branching

In the case of a dichotomic branching, we use a predefined AbstractBinIntBranching where there is only two alternatives at each choice point. So no value selection strategy is used and the branch selection is simply implemented as :

```
public abstract class AbstractBinIntBranching extends AbstractIntBranching {
  public int getFirstBranch(Object x) {
    return 1;
  }

  public int getNextBranch(Object x, int i) {
    return 2;
  }

  public boolean finishedBranching(Object x, int i) {
    return (i == 2);
  }
}
```

On this basis, goDownBranch(Object x, int i) is the single method that needs to be implemented. It computes the middle point of the domain and branches first on the right side by updating the lower bound of the variable and then on the left side by updating the upper bound.

```
/**
 * A dichotomic branching example
 */
public class DichotomicBranching extends AbstractBinIntBranching implements IntBranching {

    protected IVarSelector varSelector;

    public DichotomicBranching(IVarSelector varSel) {
        this.varSelector = varSel;
    }

    /**
     * delegates to the var selector the choice of the  branching variable
     */
    public Object selectBranchingObject() {
        return varSelector.selectVar();
    }


    public void goDownBranch(Object x, int i) throws ContradictionException {
        IntDomainVar var = (IntDomainVar) x;

        // we compute the bound to split
        int bound = (var.getSup() - var.getInf()) / 2 + var.getInf() + 1;

        switch (i) {
            case 1: {
                var.setInf(bound);
```

```
                manager.problem.propagate();break;
            }
            case 2: {
                var.setSup(bound - 1);
                manager.problem.propagate();break;
            }
        }
    }

}
```

Branching by posting constraints

You can easily implement complex branching by posting a constraint instead of acting on the domain of a variable as we only did at that time. It is indeed useful to implement n-ary branching (on more than one variable). Just for example, the previous code can be changed by the following where a “greater than equal” and “less than equal” constraint are posted instead of calling the updateInf and updateSup methods :

```
public void goDownBranch(Object x, int i) throws ContradictionException {
        IntDomainVar var = (IntDomainVar) x;
        int bound = (var.getSup() - var.getInf()) / 2 + var.getInf() + 1;

        switch (i) {
            case 1: {
                manager.problem.post(manager.problem.geq(var, bound));
                manager.problem.propagate();break;
            }
            case 2: {
                manager.problem.post(manager.problem.leq(var, bound-1));
                manager.problem.propagate();break ;
            }
        }
    }
```

LDS (Limited discrepancy search)

Harvey and Ginsberg describe an interesting tree search algorithm called limited discrepancy search (LDS) that exploits the existence of good heuristics. LDS is based on the assumption that a solution constructed according to a good value-ordering heuristic is unlikely to contain many mistakes. For a path in the search tree, the number of nodes where the chosen value differs from that specified by the value-ordering heuristic is called the discrepancy count. When LDS is forced to backtrack, it examines new paths in increasing order of the discrepancy count. A tree search with a fixed limited discrepancy according to a given branching can be implemented as a specific Branching which maintains the number of discrepancy of the current path using a backtrackable integer. All methods are delegated to the branching which is limited (attribute delegate) except the getNextBranch and finishedBranching which counts the discrepancy and check it does not violates the limit allowed. The LimitedSearch class constitutes a kind of meta-branching which applies a limited discrepancy search to its delegate branching.

```
public class LimitedSearch extends AbstractIntBranching implements IntBranching {

    IntBranching delegate;
    IStateInt discrepancyCount;
    int maxDiscrepancy;

    public LimitedSearch(Problem pb, IntBranching delegate, int nbViolsMax) {
        this.delegate = delegate;
        discrepancyCount = new StoredInt(pb.getEnvironment(), 0);
        maxDiscrepancy = nbViolsMax;
    }

    public int getNextBranch(Object x, int i) {
        discrepancyCount.add(1);
        return delegate.getNextBranch(x, i);
    }
```

```
    public boolean finishedBranching(Object x, int i) {
        if (discrepancyCount.get() < maxDiscrepancy)
return delegate.finishedBranching(x, i);
        return true;
    }

    public Object selectBranchingObject() {
        return delegate.selectBranchingObject();
    }

    public int getFirstBranch(Object x) {
        return delegate.getFirstBranch(x);
    }

    public void goDownBranch(Object x, int i) throws ContradictionException {
        delegate.goDownBranch(x, i);
    }

    public void goUpBranch(Object x, int i) throws ContradictionException {
        delegate.goUpBranch(x, i);
    }

}
```

Limited Depth first search

Another way to limit the tree search exploration is to limit the depth of the tree. Once the limited depth is reached, we want the solver to branch according to the heuristic without backtracking at all. Every methods of the IntBranching class is simply implemented by delegating it to its delegate attribut. The only method to change is the finishedBranching which returns true as soon as the limit of depth has been reached. Indeed, As for the LDS above, after this point, no alternatives will be tried as the branching always tells the solver it is finished without asking the delegate.

```
public class DepthLimitedSearch extends AbstractIntBranching implements IntBranching {
    IntBranching delegate;
    int maxDepth;

    public DepthLimitedSearch (IntBranching delegate, int maxDepth) {
        this.delegate = delegate;
        this.maxDepth = maxDepth;
    }

    public boolean finishedBranching(Object x, int i) {
          int startDepth = this.getProblem().getSolver().getSearchSolver().baseWorld;
        if (manager.problem.getWorldIndex() -  startDepth < maxDepth)
             return delegate.finishedBranching(x, i);
        return true;
    }

   public Object selectBranchingObject() {
        return delegate.selectBranchingObject();
    }

    public int getFirstBranch(Object x) {
        return delegate.getFirstBranch(x);
    }

    public int getNextBranch(Object x, int i) {
        return delegate.getNextBranch(x, i);
    }

    public void goDownBranch(Object x, int i) throws ContradictionException {
        delegate.goDownBranch(x, i);
    }

    public void goUpBranch(Object x, int i) throws ContradictionException {
        delegate.goUpBranch(x, i);
    }
}
```

How to implement 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 predictates) 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 contraint classes (for instance, such as GreaterOrEqualXYC for a binary inequality)

The constraint hierarchy

Each new constraint must be represented by an object implementing the Constraint interface. To help the user defining new constraint classes, several abstract classes defining Constraint 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 :

1. AbstractUnIntConstraint, AbstractBinIntConstraint, AbstractTernIntConstraint : A default implementation for constraint involving one, two or three integer variables (IntDomainVar).
2. AbstractLargeIntConstraint : A default implementation for constraint involving any number of integer variables.

Constraints over integers must implement the following methods (grouped in the IntConstraint interface).

-
-

In the same way, SetConstraint can inherit from :

1. AbstractUnSetConstraint, AbstractBinSetConstraint, AbstractTernSetConstraint.
2. AbstractLargeSetConstraint.

Moreover, Constraints stating on integer and set variables can be written by inheriting from AbstractMixedConstraint.

Backtrackable structures

Updating domains of variable (indexes and links with propagation)

The event system handled by the propagation mechanism (constawake, asynchronous)

An Example : the occurrence constraint

Solver settings

The solver object is responsible for managing the search for solutions. It is accessed from the problem, via the getSolver method. This object may be parameterized before actually searching for a solution, in order to control its behavior. This is done by means of of setter methods on the Solver object.

The first possibility for Solver parameterization concern trace logs.

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

```
Solver.setVerbosity(Solver.SEARCH);
```

The code above ensure that messages are printed in order to describe the construction of the search tree. For instance, the following trace ....

Four verbosity levels are available:

- **Solver.SILENT** prints nothing
- **Solver.SOLUTION** prints messages whenever a solution is reached
- **Solver.SEARCH** prints a message at each choice point
- **Solver.PROPAGATION** prints messages to trace propagation

Note, that in the case of a verbosity set to **Solver.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 with the following setter method

```
pb.getSolver().setLoggingMaxDepth(10);
```

which changes this value to 10.

Inside choco

CHOCO CVS Source public access

The Java Choco source can be checked out from the SourceForge CVS repository directly. Here is a basic *Howto* under Eclipse IDE:

- 1. In the workspace of your choice, do "New Project" and choose "CVS / Checkout Projects from CVS"
- 2. Check "Create a new repository location" and fill the following info:

```
Host :                choco.cvs.sourceforge.net
Repository path :     /cvsroot/choco
User :                anonymous
(no password)
(keep pserver and default port)
```

- 3. "Use an existing module [...]" and select "java"
- 4. Follow the next screens as usual to create the project.

You will need **junit** jar to compile the project properly (JUnit.org (http://www.junit.org/) if you don't have it already in Eclipse). To generate the choco.jar, just export all files, excluding test* and chocosamples.

Packages description

Packages (Javadoc)	
choco	The root package for the Constraint Programming Kernel
choco.bool	A package devoted to propagation over Boolean combinations of constraints
choco.branch	A package devoted to control (branching schemes and heuristics) for branching in a search tree
choco.global	
choco.global.matching	
choco.integer	A package devoted to propagation over integer domain variables.
choco.integer.constraints	A package devoted to constraints over integers.
choco.integer.constraints.extension	
choco.integer.search	A package devoted to choice points and search heuristics specific to integer variables
choco.integer.var	A package devoted to the management of variables and domains for integers
choco.mem	A package devoted to backtrackable data structures.
choco.palm	A package devoted to e-tools - an explanation-based solver and all the needed tools.
choco.palm.benders	
choco.palm.benders.explain	
choco.palm.benders.search	
choco.palm.cbj	
choco.palm.cbj.explain	
choco.palm.cbj.integer	
choco.palm.cbj.search	
choco.palm.dbt	
choco.palm.dbt.explain	This package contains generic interfaces and classes for storing and managing explanations.
choco.palm.dbt.integer	
choco.palm.dbt.integer.explain	A package devoted to integer explanations, that is explanations for the different kinds of variables.
choco.palm.dbt.prop	A package devoted to an extension of Choco propagation tool to support explanation features.
choco.palm.dbt.search	A package devoted to explanation based search.
choco.palm.dbt.search.pathrepair	A package devoted to the decision-repair algorithm.
choco.palm.global.matching	
choco.palm.integer	This package contains classes for integer-based objects (integer constraints, variables, explanations).
choco.palm.integer.constraints	Package devoted for integer constraints.
choco.palm.real	This package contains classes for real-based objects (real constraints, variables, explanations).

choco.palm.real.constraints	
choco.palm.real.exp	
choco.palm.real.explain	
choco.palm.real.search	
choco.palm.search	
choco.prop	A package of classes devoted to the event model of constraint propagation
choco.real	A package devoted to continuous propagation based on interval arithmetic.
choco.real.constraint	A package devoted continuous constraints.
choco.real.exp	A package devoted real expression, that is composition of operators over real variables.
choco.real.search	A package devoted to serach tools based on real constraints and variables.
choco.real.var	A package devoted to contiinuous domains and variables.
choco.search	A package devoted to the the control of search algorithms
choco.set	
choco.set.constraint	
choco.set.search	
choco.set.var	
choco.util	A package devoted to non-backtrackable data structures

- This page was last modified 18:50, 3 October 2006.