# Programmation par Contraintes Modélisation et Résolution (CM3)

4A STI

B. Nguyen

D’après : F. Laburthe, S. Person

# PPC vs Méthodes de RO

- Ce qu'on peut exprimer en PPC
  - Contraintes définies en énumérant les valeurs possibles (définition en **extention**)
  - Contraintes arithmétiques sur des expressions (<, >, =, …)
  - Contraintes logiques (ET, OU, …)
  - Contraintes dont la sémantique est définie (Valeurs_toutes_différentes)
- Pour un problème de RO, on ne sait pas forcément quel algorithmes utiliser pour le résoudre.
- Algorithmes utilisés (à voir dans les prochains cours)
  - Generate and Test + Backtrack
  - Propagation de contraintes

# Implémentation des solvers

- De nombreux exemples de solvers existent.
- Dans ce cours, nous utiliserons la version 1 du solver « Choco » (Laburthe *et al.*, première version en 1996)
  - http://sourceforge.net/projects/choco/

  → Très simple à utiliser en Java
- Actuellement à la version 3.3
  - http://choco-solver.org/

# Modélisation d'un problème

- Classe « choco.Problem »
  - Variables et contraintes sont liées à cette classe
- Types de variables : Int, Real, et Set
  - EnumIntVar : ensemble énuméré
  - BoundsIntVar : bornes d'un intervalle
  - RealVar : idem
  - SetVar : la variable représente un ensemble fini de valeurs. Cette variable est définie par deux valeurs : L et U, respectivement les bornes minimales et maximales de l'enveloppe. L'enveloppe représente l'union de tous les ensembles, le noyau représente l'intersection de tous les ensembles. NB : un variable représetant un ensemble parmis un univers de N valeurs possède $2^N$ valeurs différentes !

**On va considérer des valeurs entières surtout !**

# Quelques éléments du *user guide*

http://choco.sourceforge.net/userguide.html

# IntVar

- Crées avec :
  - makeBoundIntVar(String s, int lb, int ub) : creates a finite domain variable with domain (lb .. ub), with name s.
  - makeEnumIntVar(String s, int lb, int ub) : creates a finite domain variable whose domain is approximated by bounds (lb .. ub), with name s.
- L'état peut être observé au travers des méthodes suivantes
  - 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.
  - getValue(): 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.

# RealVar

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

# SetVar

- Peuvent être crées avec :
  - 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.
- L'état peut être accédé avec:
  - 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 kernel.
  - 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.
- Le domaine peut être modifié avec : (ContradictionException possible en cas de changement inconsistant) :
  - 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.

# Contraintes

- Les contraintes les plus simples sont des comparaisons définies sur des expressions de variables, telles que des combinaisons linéaires. Elles sont accessibles via l'API de la classe Problem
  - 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.
- Pour construire des expressions complexes de variables, des opérateurs simples peuvent être utilisés :
  - 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].

- NB : la racine d'un IntExp est une IntVar …

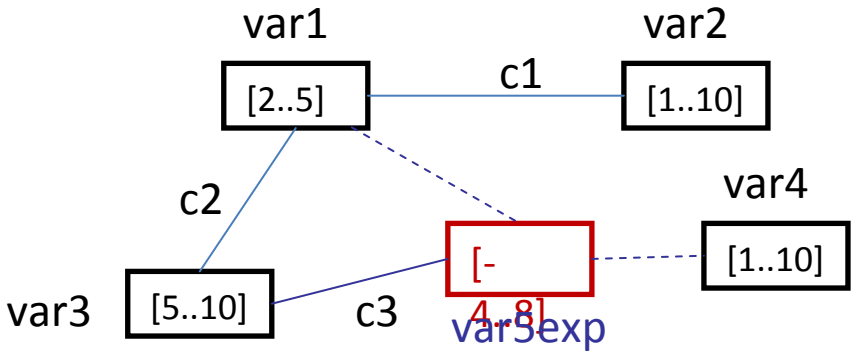
# Contraintes

- Une contrainte est ensuite ajoutée au problème.

```
Constraint C1 = P.eq(var, 5);
P.post(C1);
```

# Programme Exemple1

# Programme Exemple2




```
IntDomainVar var1 = prob.makeEnumIntVar(“var1”, 2, 5);
IntDomainVar var2 = prob.makeEnumIntVar(“var2”, 1, 10);
IntDomainVar var3 = prob.makeEnumIntVar(“var3”, 5, 10);
IntDomainVar var4 = prob.makeEnumIntVar(“var4”, 1, 10);

Constraint c1 = prob.gt(var2,var1);
Constraint c2 = prob.leq(var1,var3);

IntExp var5Exp = prob.minus(var4, var1);

Constraint c3 = prob.eq(var5Exp,var3);
prob.post(c1); prob.post(c2); prob.post(c3);
```

# Plus de contraintes…

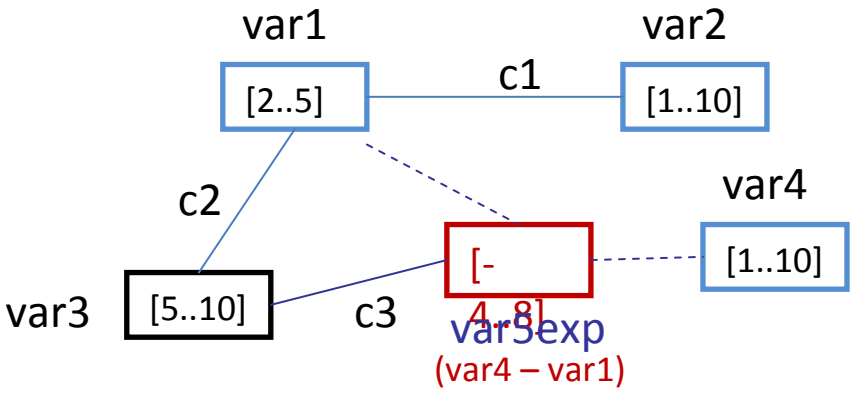
- Globales
  - allDiff
  - globalCardinality
  - occurrence
  - cumulative
  - nth
  - lex
  - atMostValue
  - regular

# allDiff / globalCardinality

- *allDiff:* les variables sont distinctes deux à deux
- *globalCardinality:* le nombre de variables prenant cette valeur doit être dans un intervalle

vars=[var1,var2,var4]
low[4]=1
high[4]=2

c1:var1 < var2
c2: var1 <= var3
c3: var5exp<>var3
c4:gc(vars,low,high)



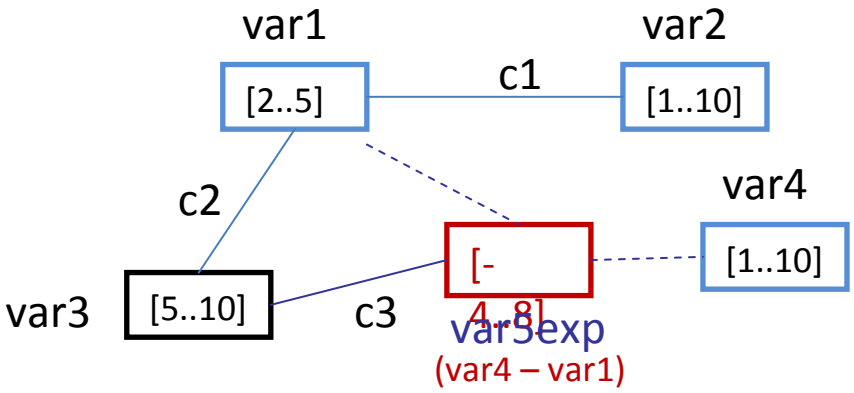

Satisfying?
var1 = 5
var2 = 8
var3 = 5
var4 = 5

# Occurence

- *occurrence:* met dans la variable (ici *var3*) le nombre de fois qu'une valeur apparait dans un ensemble de variables (ici *vars*)

vars=[var1,var2,var4]

c1:var1 < var2
c2: var1 <= var3
c3: var5exp<>var3
c4: occur(vars, 5, var3)

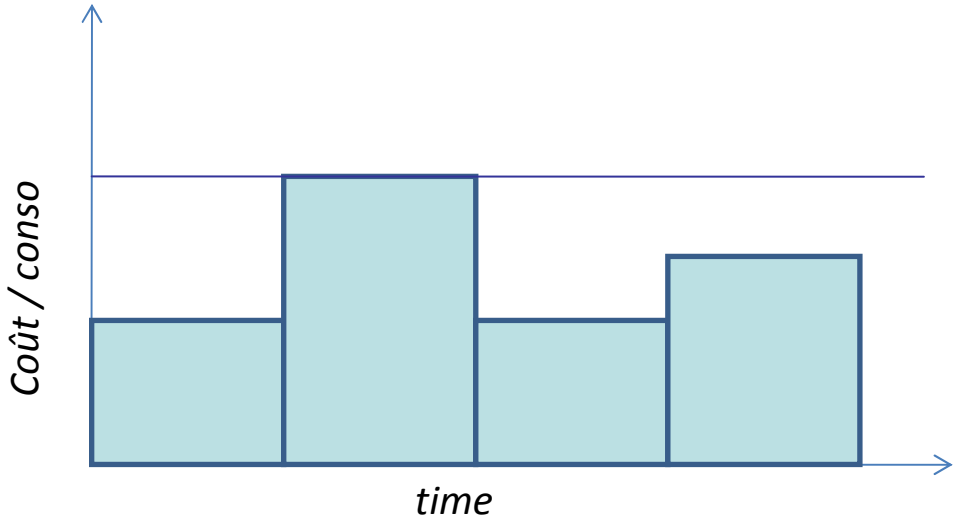



Satisfying?
var1 = 5
var2 = 8
var3 = 2
var4 = 5

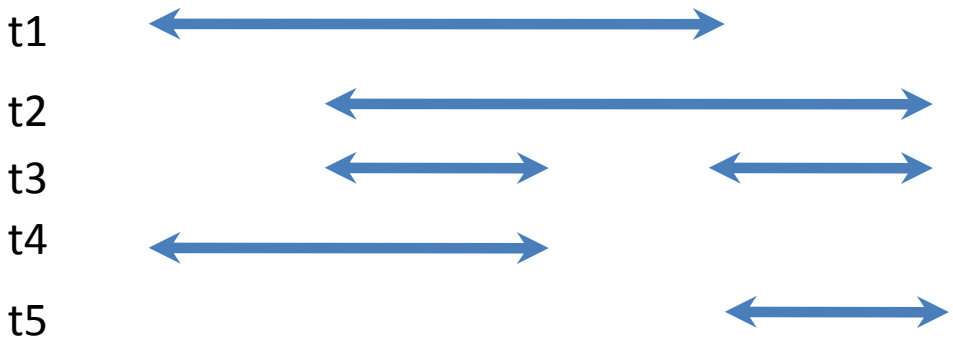
# Cumulative

- *cumulative:* Etant donné un ensemble de tâches définies par :
  - leur date de début, fin, durée, consommation/hauteur
  - *Cumulative* assure qu'à n'importe quel instant *t* la somme des hauteurs des tâches exécutées à cette instant est inférieur à une certaine borne.

# Cumulative




*C* = capacité (cumulative)




Dans cette exemple, chaque tâche à une consommation = 1 unité

# Nth / lex

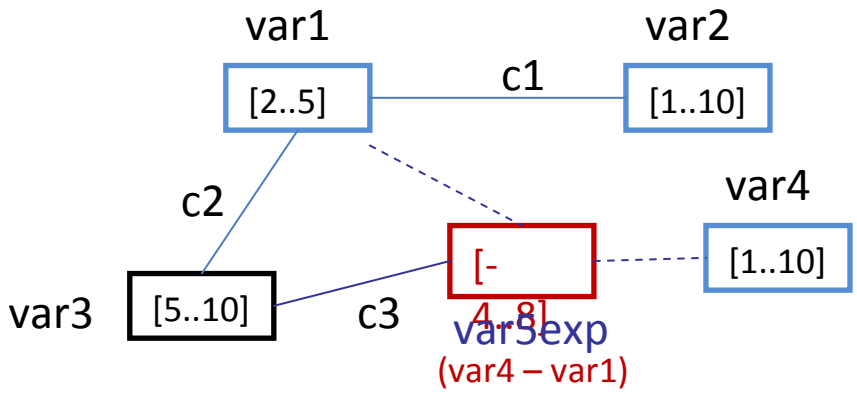
- *nth:* permet de spécifier une contrainte où les valeurs sont variables
- *lex:* ordre lexicographique sur deux vecteurs de valeurs entières.

# atMostValue

- *atMostValue:* assure que le nombre de valeurs différentes est au plus *n*

vars=[var1,var2,var4]

c1:var1 < var2
c2: var1 <= var3
c3: var5exp<>var3
c4:amv(vars, 2)

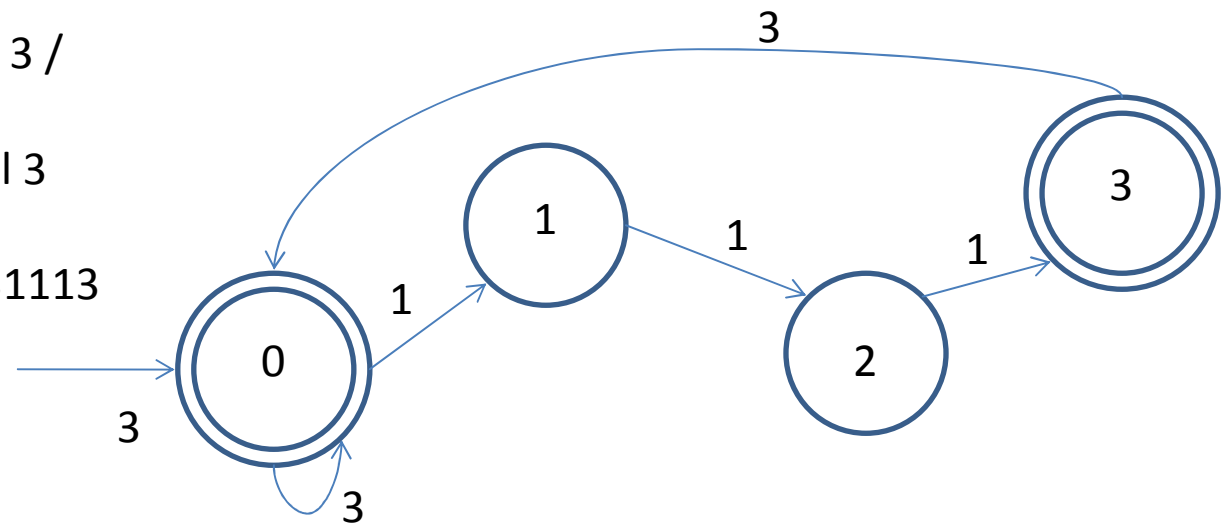



Satisfying?
var1 = 5
var2 = 8
var3 = 5
var4 = 5

# Regular

- *regular:* doivent être reconnues par un automate fini (langage régulier)

**Mots acceptés :** mots commençant par un nombre arbitraire de 3 / séquence 111 et se terminant par un seul 3
Exemple : 331113 ou 333111331113333113



# Regular

```
Problem pb = new Problem();
IntDomainVar[] vars = new IntDomainVar[6];
for (int i = 0; i < vars.length; i++)
   { vars[i] = pb.makeEnumIntVar("v" + i, 0, 5); }
// 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));
t.add(new Transition(2, 1, 3));
t.add(new Transition(3, 3, 0));
t.add(new Transition(0, 3, 0));
// Two final states: 0, 3
List<Integer> fs = new LinkedList<Integer>();
fs.add(0); fs.add(3);
DFA auto = new DFA(t, fs, 6); // Build the DFA
pb.post(pb.regular(vars , auto)); // post the constraint
```

# Integer Vars > Advanced Constraints > *regular*

- Can be used as a GAC algorithm for a list of feasible or infeasible tuples
  - Can be more efficient than a standard GAC algorithm if the DFA is compact

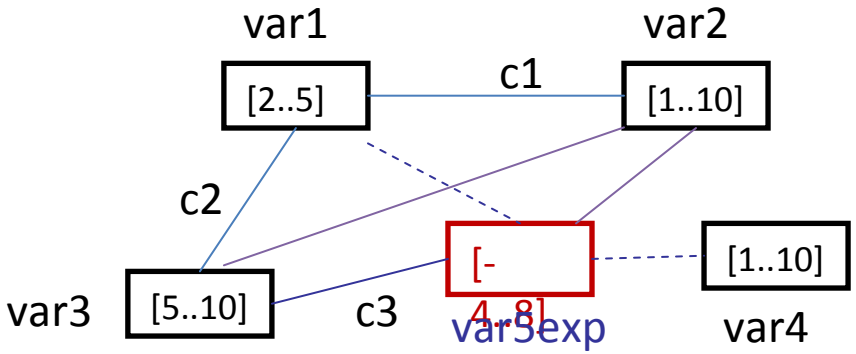
# Integer Vars > Advanced Constraints > *regular*

- AllEqual constraint using the regular constraint

```
Problem pb = new Problem();
IntDomainVar v1 = pb.makeEnumIntVar("v1", 1, 4);
IntDomainVar v2 = pb.makeEnumIntVar("v2", 1, 4);
IntDomainVar v3 = pb.makeEnumIntVar("v3", 1, 4);
//add some allowed tuples (here, the tuples define an allEqual 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});
// post the constraint
pb.post(pb.regular(new IntDomainVar[]{v1, v2, v3},tuples));
```

# Composition de contraintes

- And
- Or
- Implies
- If Only If
- If Then




```
IntDomainVar var1 = prob.makeEnumIntVar(“var1”, 2, 5);
IntDomainVar var2 = prob.makeEnumIntVar(“var2”, 1, 10);
IntDomainVar var3 = prob.makeEnumIntVar(“var3”, 5, 10);
IntDomainVar var4 = prob.makeEnumIntVar(“var4”, 1, 10);

Constraint c1 = prob.gt(var2,var1);
Constraint c2 = prob.leq(var1,var3);

IntExp var5Exp = prob.minus(var4, var1);

Constraint c3 = prob.neq(var5Exp,var3);
Constraint c4 = prob.or(prob.lt(var2,var3),prob.gt(var5Exp,0));
prob.post(c1); prob.post(c2); prob.post(c3); prob.post(c4);
```

# Contraintes sur des Set

- member
- notMember
- setDisjoint
- eqCard
- …

# Variables Réelles

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

```
RealVar x = pb.makeRealVar("x");
RealVar y = pb.makeRealVar("y", -1.0e8, 1.0e8);
RealVar z = pb.makeRealVar("z", -1.0e8, 1.0e8);

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))));
…
Equation eq1 = (Equation) pb.eq(exp1, pb.cst(-13));
eq1.addBoxedVar(y); eq1.addBoxedVar(z);
…
pb.post(eq1); pb.post(eq2); pb.post(eq3);
```

# Contraintes Binaires

- Une relation définit des paires autorisées ou interdites pour les deux variables présentes dans la contrainte. Une relation peut être définie de deux manières :
  - Tables : en spécifiant *extensionnellement* toutes les valeurs
  - Prédicat : en codant une méthode pour vérifier si la contrainte est correcte ou pas
- Les contraintes de table sont utiles pour des domaines petits. Les contraintes de prédicat sont utiles pour des domaines grands.

# Contraintes Binaires

- La création d'une contrainte pour une relation peut être faite au travers de l'API Problem également :
  - makePairAC(IntVar v1, IntVar v2, ArrayList pairs, boolean feas, int ac) (pairs = liste d'int[] de taille 2)
  - relationPairAC(IntVar v1, IntVar v2, BinRelation pairs, int ac).
- Le paramètre *feas* indique si la relation correspond à des paires autorisées ou interdites. Le paramètre ac indique l'algorithme à utiliser pour *arc consistency* (par défaut ac=2001). Les autres valeurs sont :
  - 3 for AC3 algorithm (searching from scratch for supports on all values)
  - 4 for AC4 algorithm (maintaining a count of supports for each value)
  - 2001 for the AC2001 algorithm (maintaining the current support of each value)
- La définition d'une relation binaire basée sur un prédicat peut être faite en héritant de la classe CouplesTest, comme indiqué dans l'exemple suivant :

  ```
  public class MyInequality extends CouplesTest{ public boolean checkCouple(int x, int y) { return x != y; } }
  ```
- On peut définir la contrainte suivante :

  ```
  pb.post(pb.relationPairAC(v1, v2,new MyInequality()));
  ```
- On peut créer des contraintes binaires avec
  - infeasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
  - feasPairAC(IntVar v1, IntVar v2, ArrayList pairs)
  - relationPairAC(IntVar v1, IntVar v2, BinRelation binR)

# ExempleTuple

# Contraintes n-aires

- Pour les contraintes binaires → Arc Consistency
- Pour les contraintes n-aires → Forward Checking
- L'API pour créer les contraintes est :
  - makeTupleFC(IntVar[] vs, ArrayList tuples, boolean feas)
  - relationTuple(IntVar[] vs, LargeRelation rela)
- On peut définir une relation n-aire en intension aussi :

```
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; }
}
```

- Sinon, les tuples sont donnés dans une table int[] donnant les valeurs permises/interdites. On peut ensuite donner les contraintes sur le problème :

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