Year 2014–2015

Constraint programming

Practical work handbook

Mireille Ducassé : mireille.ducasse@insa-rennes.fr
José Ángel Galindo Duarte : jagalindo@us.es
Pascal Garcia : Pascal.Garcia@insa-rennes.fr

4th year
Informatics departement
INSA Rennes

Sommaire

Foreword

This workbook to present various concrete problems combinatorial research and optimization. The purpose of this lab is to teach you how to model these problems by using programming by constraints. A strong point of constraint programming is the resolution of a problem is independent of its modeling and that, therefore, we can use this type of programming without know the underlying resolution algorithms (propagation, branch and bound, ldots). However, you will quickly has an understanding of these algorithms is needed to understand what is happening and, more concretely, debug your programs.

Thanks

Ce cahier de TP a été initialement conçu par Edouard Monnier avec l'aide de Christiane Hespel. Il a été ensuite régulièrement remanié par les intervenants successifs, en particulier, Matthieu Carlier, Florence Charreteur, Tristan Denmat, Coralie Haese, Vincent Montfort, Harold Mouchère, Matthieu Petit et Benoit Ronflette.

Introduction to ECLiPSe Prolog

All the exercises within this book will rely on the use of ECLiPSe[1]. Please note that you will need to use Prolog, mainly you should be able to work with the **unification** and **backtracking** algorithms (see chapter 2). You will use two extensions. First, the arrays and second, iterators. Prolog allows to resolve constraint satisfaction problems. However, prolog poses different problems when coping with complex problems. Therefore, we are going to use two different libraries

- *ic* to solve problems using integer variables.
- *ic_symbolic* to solve problems with enumerations. This is, those variables where a valid value is part of a set of finite values. E.g. $\{th\acute{e}, caf\acute{e}, eau, lait, jus\}$ (see. TP 2).

You will find Eclipse installed on the machines available for students in the INSA classroom. To start it open a terminal and write the following:

```
$ eclipsep
ECLiPSe Constraint Logic Programming System [kernel]
...
Version 6.0 #162 (i386_linux), Mon Sep 20 06:09 2010
[eclipse 1]:
```

Eclipse files ends with the extension .ecl or pl. You can open the file tp1.ecl with the command [tp1]. (standard command in Prolog). Please note that you cannot start the name of a file in capital letters because those are reserved for variables.

1 The matrices

In eclipse there are matrices. You can use them intead of lists when you need direct access to the elements within it. Also, you can have multidimensional matrices (eg : matrices $N \times M$).

A matrix is defined by the function `[]/1`. Also, the predicate `dim/2` allows to link a matrix with its dimension. E.g:

[1]You can download ECLiPSe from : `http://eclipseclp.org`

```
[Eclipse 1]:  Tab_1 = [](1,2,3),
              dim(Tab_1,Dim).

Tab_1 = [](1, 2, 3)
Dim = [3]

[Eclipse 2]:  Tab_2 = []([](1,2,3),[](4,5,6)),
              dim(Tab_2,Dim).

Tab_2 = []([](1,2,3),[](4,5,6))
Dim = [2,3]

[Eclipse 3]: dim(Tab,[3,3]).

Tab = []([](_183, _184, _185), [](_179, _180, _181), [](_175, _176, _177))
```

Note that in the last example we do utilize a variable not instanciated. This implies the creation of a matrix with dimension 3×3 without variables.

The access to the matrix elements is done with the predicate `is/2`. **You CANNOT use the unification directly.** E.g:

```
[Eclipse 4]: Tab_1 = [](1, 2, 3),
             Var = Tab_1[2].

Var = [](1, 2, 3)[2]

[Eclipse 5]: Tab_1 = [](1, 2, 3),
             Var is Tab_1[2].

Var = 2

[Eclipse 6]: Tab_2 = []([](1, 2, 3),[](4, 5, 6)),
             Var is Tab_2[1,3].

Var = 3.
```

However, you can use T[i] within the *ic* expresions:

```
[Eclipse 7]: Tab_1 = [](1, 2, 3),
             Tab_1[2] + Tab_1[3] #= Var

Var = 5.
```

2 The iterators

ECLiPSe allows the use of iterators in order to introduce control elements in Prolog. Those iterators are no more than sugar sintax that is later transformed on Prolog when compiling (see the preparation chapter). It's advised to remember that the compilation errors are linked to the transformed program instead of the written one. Also, ECLiPSe provides the traceability between the transformation and the code.

The general form is

```
(Itérateur1, ..., ItérateurN do Actions)
```

On each iteration the iterators from 1 to n evolve. Each pass the predicates are called. Note that this is not the case for nested iterations. However, you can still perform nested iterations within `Actions`.

There are only three different iterator flavors.

2.1 For each element of : `foreach(Elem,List)`

In the iteration i, the logic variable *Elem* is unified with the ith list element *List*. **Alert, the variable** *Elem* **is visible only inside the iterator** . E.g.:

```
:- ( foreach(Elem,[1, 2, 3])
   do
     write(Elem)
   ).

123

:- ( foreach(Elem, [1, 2, 3]),
     foreach(Elem2, List)
   do
     Elem2 is Elem + 5
   ).

List = [6, 7, 8]
```

2.2 For each integer between : `for(Indice,Min,Max)`

`for` is a subtype of `foreach` for the integer variables. It assign the variable *Indice* from Min to Max. e.g.

```
:- ( for(Indice, 1, 3)
   do
      write(Indice)
```

```
   ).

123
```

2.3 Accumulator : `fromto(Debut,In,Out,Fin)`

fromto is an accumulator where the initial value is `Debut`, the current value is entered `In`, the out value is assigned at `Out` and the final value is `Fin`. The calculus to pass from out to in has to be explicit within the predicates `Actions` associated to the fromto. in the first pass of the iteration , `In` is `Debut`, in the following pass i $\texttt{In}_i$ vaut $\texttt{Out}_{i-1}$, in the last pass `Out` is `Fin`. Ex:

```
:- ( fromto(1, In, Out, 3)
   do
      Out is In + 1,
      write(In)
   ).

123
```

from to is often used with another operator. The following example illustrates the power of fromto:

```
inverse(List,LInverse):-
    ( foreach(Elem, List),
      fromto([], In, Out, LInverse)
    do
       Out = [Elem | In]
    ).

:- inverse([1, 2, 3], List).

List = [3, 2, 1]
```

Scope of iterators

As detailed in the annex of the course, the variables used in *Actions* aren't visible for that iterator. For example, the following predicate wont work :

```
afficheKO(Tab):-
    dim(Tab, [Dim]),
    ( for(Indice, 1, Dim),
    do
      Var is Tab[Indice],
```

```
      write(Var)
    ).
```

In fact, the call `Var is Tab[Indice]` will fail because `Tab` is considered as a local variable, thus, is not instantiated.

In order to pass a program variable in the scope of an iterator, you must explicitly declare that this variable is used by the iterator via the predicate `param` (d'arité variable).
Ex:

```
afficheOK(Tab) :-
    dim(Tab, [Dim]),
    ( for(Indice, 1, Dim),
      param(Tab)
    do
       Var is Tab[Indice],
       write(Var)
    ).
```

3 The finite domains

As detailed in the course, a constraint problem with finite fields consists of a set of constraints and a set of finite fields in which the variables can take their value. Solving such a problem is to find an instantiation of the variables in their respective values such as all constraints are satisfied. Solving a problem is based on two principles:

- the constraint **propagation**. Each constraint is analyzsd individually to eliminate the values from the domains that cannot be assigned. For example if we consider the constraint $X < Y$ with the domains $D_x = D_y = 0..10$, the propagation algorithm will reduce the domains to $D_x = 0..9$ and $D_y = 1..10$. Later, if we add the constraint $X > Y$, the propagation algorithm will resolve that $D_x = 2..9$ and $D_y = 1..8$, then to re-traverse the first constraint. This will happen until the domains are $D_x = D_y = \emptyset$.
- the **enumeration**. When the propagation is not enought fot finding a solution, the enumeration is used. The enumeration consist on fixing randomly a value to a variable.

Note that these two algorithms are called numerous times during the constraint solving. A phase delay is performed, then a enumeration and a propagation phase taking into account the enumeration performed, etc.

To load the library for integer domains:

```
:- lib(ic).
```

The principal constraints in *ic* are numeric constraints: equations (`#=`), in-equations (`#<` , `#>`, `#=<`, `#>=`), dis-equations (`#\=`) wheter the members are the termes relatives to the finite domain variables.

The domain of an integer variable is defined thanks to the predicate `#::/2` :

- `V #:: D` constraint the variable V from the domain D ;
- `Lv #:: D` constraint each variable from the list `Lv` starting with the domain D.

The domain D is specified by the set of values contained by it, either it is an interval or a set of numeric values *min..max*, etc. (see documentation).

The propagation algorithm is launched when the first constraint is posed to ECLiPSe. However, the enumeration algorithm should be called explicitly using the predicates `labeling/1`, `search/6`, or a custom version of the labeling (see chapter 4). `labeling(Lv)` successfully assign for each variable L_v a value within the domain depicted by the constraints.

Is it possible to do optimization withing the finites domains using an algorithm **branch and bound**. You'll find in the library the code `branch_and_bound` of *ic*, the preicate `minimize(Pred,Var)` looks for a solution of `Pred` that minimize the value of `Var`.

Symbolic domains

We call symbolic domain a domain that consist on a finite sets of non numerical values. In this case, the field of a variable is a finite set of terminals, e.g. `: Day &:: week`, with `week` the domain representing the symbolic domain
`{mo, tu, we, th, fr, sa, su}`.

The way to load the library of symbolic domains is :

```
:- lib(ic_symbolic).
```

We find the same operators as those depicted for the numeric finites domains with the diference that those start with `&` instead of `#`.

To define a domain *couleur* locally we can do as follows:

```
:- local domain(couleur(rouge,vert,bleu)).
```

The area is defined by a predicate functor which is the name of field and the arguments are the atoms that represent the values Possible domain. The arity of the predicate corresponds to the arity field (here 3 colors).

4 Enumeration : `labeling` and `search`

Both predicates labeling and search are used to to guide the list of possible values for variables when the spread is not sufficient to find a solution.

Warning: as seen in detail in Chapter 4 of the course, these two predicates do not cover all the ways to guide enumeration. In some cases they are not sufficiently effective. In TP 6 you will propose enumeration strategies best suited to the problem by customizing the predicate labeling.

4.1 Labeling

The labeling predicate chose a variable v from the problem and a single value a within the concrete domain of that variable, later it adds the constraint $v = a$. if there is no solution wheter that constraint is satisfied , $v = a$ is removed from the problem, a is removed from the domain v the a backtrack is performed. Here is the base implementation of this predicate in ECLiPSe:

```
labeling([]).
labeling([Var|List]) :-
   indomain(Var),
   labeling(List).
```

`indomain(Var)` is a predicate from *ic* that instantiate the variable Var enumerating the values of its domain in ascending order. For example, if we have $Var \in 0..10$, then `indomain(Var)`is going to start by posing $Var = 0$. If it ends in failure, then $Var = 1$ is going to be posed so on until a solution is found or all domain values have been tried.

In the standard version of the labeling, the enumeration is done by instantiating the variables in the order they appear in the list of variables.

4.2 Search

The search predicate is an implementation of labeling offering different enumeration strategies widely used in the community of constraint programming. It has a parameter different axes. A careful reading of the documentation is Recommended to understand the predicate and the various opportunities.

5 Some utilities

5.1 Previous commands

The command `h.` allows t know the previous commands executed in the ECLiPSe session. Ex :

```
[Eclipse 6]: h.
1 Tab_1 = [](1, 2, 3).
2 Tab_1 = [](1, 2, 3), dim(Tab_1).
3 Tab_1 = [](1, 2, 3), dim(Tab_1, Dim).
4 lib(ic).
5 ["mesTps/bal"].
```

To re-execute a previous command its enough to press the number of that command followed by a point. E,g :

```
 [Eclipse 6]: 4.
lib(ic).

Yes (0.00s cpu)
```

5.2 Exit ECLiPSe

To exit ECLiPSe, you can use the command `halt` or the shortcut `Ctrl+D`.

Conducting our practical work

Homework

The practical work will be done in pairs and each TP will be a report that will be scored. **These reports should be delivered at worst before the next TP, printed and uploaded to Moodle.**

At least they should include:

1. The final code for your solution. Note: The scores will take into account the quality of the product code and comments.
2. **The tests** you have validated your code. Each test will be described by: :
   - The data used (if these are not those of the text TP)
   - The aim executed
   - The first responses of ECLiPSe
   - The time to execute it
   - Some indications about the number of solutions
3. A short response but applicable to all comprehension questions. Do not hesitate of using the schemas from the examples when you consider its pertinent.

   The last two points will be included in a general comment. The dropped file Moodle should be executable by the professor.

Structure de vos programmes

The code for your TP will have the following structure :

- main predicate responsible for solving the problem posed (this predicate will be enriched through the TP).
- predicates in charge of posing the data

- predicates definition of variables and their areas
- predicates posing constraints
- utilities predicates

Code quality and coding conventions

You must pay attention to code quality, especially :

- Use meaningful variable names, not p(X,Y), ...)
- **Indent the code**
- Do not list **just a single predicate by line,** even, and especially, if the predicate is short. It may seem a waste of space, but it will make you a lot of time during the development. It also will allow teachers to help you more easily.
- Do not interlard its code comments : **group all comments header in a predicate.** Comment within a predicate is often a sign that we should cut in auxiliary predicates whose name could convey the essence of what we wanted to in the commentary.
- Do not add hard constraints to the problem
- Try to modularize the code. (Do not copy-paste!)
- Write reusable code when relevant

Methodology

The constraints to debug programs is not clear. Also, it is very important to test your programs as you. The process of solving a problem will be incremental:

1. Definition of data for the development (possibly smaller than the data of the problem that really seeks to solve).
2. Definition of variables and their field
3. Obtaining a solution and checking the consistency of this solution with the constraints already placed
4. Adding a constraint
5. Obtaining a solution and verification
6. Iterate over the two previous points as long as there are constraints to pose

7. Obtaining a solution taking into account all the constraints
8. Iterate with the data of the statement, if the original data had been limited.

It will always start with the smallest possible test data and move on to substantial values that once the point 7 is reached.

Documentation

The ECLiPSe documentation is available locally under `/usr/local/stow/eclipse6.0_136/doc/bips`.This is the documentation that must be considered first. In case the information would come to miss, see also the online documentation to the url : `http://eclipseclp.org/doc/bips`.

Add Bookmarks during the first TP on my pages and always have an open reference manual at TP (see appendix of the course).

Tuning

Finally, here are some points that can help you find errors in your programs :

- **Make prompted systematically remove compiler warnings**. Indeed, most of the time warnings reassembled by the compiler are error symptoms.

- **beware of "delayed goals"**. It is quite normal to have goals suspended until you are doing that pose constraints. However, when ECLiPSe said to have found a solution to these constraints, if there are goals on hold, it is likely that all constraints were not taken into account. We can not trust the result, especially if there was the " Branch and Bound".

- **Use the debugger**. This allows you to see exactly what happens at runtime.

  The course schedule includes concrete indications for the use of tracer ECLiPSe. This can be very useful to understand, for example, runtime errors or poorly nested iterators. It is crucial to trace executions where the test data have been reduced as much as possible to reduce the size of the trace.

TP 1

Discovering the library for constraint programing.

1 From Prolog to Prolog+ic

1.1 Constraints over trees

A rich and refined vehicle buyer wishes to purchase a car and a boat of the same color. The car he has chosen can be find in red, light green, gray or white (single color will be represented by an atom, for example " red " and an existing color in various shades by a term such as " green (light) '). The boat model that interests may be available in green, black or white.

Question 1.1 *Write in Prolog pure, the sentence* `choixCouleur(?CouleurBateau, ?CouleurVoiture)` *which is true iff the colors chosen for the boat and the car are the same and are part of existing choices.*

Question 1.2 *Explain why Prolog can be considered as a constraint solver on the domain tree.*

1.2 Prolog do not manage Maths (but it's an awesome calculator)

To produce its flagship product, a company needs to capacitors and resistors. Electrical equipment provider can provide between 5,000 and 10,000 resistors and capacitors between 9000 and 20000. For the next order, the company plans to order more than resistors capacitors.

Question 1.3 *Write the sentence* `isBetween(?Var,+Min,+Max)` *which sets a value for* `Var` *and it's true iff* `Var` *has one value in between* `Min` *and* `Max`*.*

Question 1.4 *Use the sentence* `isBetween` *and* `>=` *to define* `commande(-NbResistance, -NbCondensateur)` *which sets the number of resistors and capacitors in order to meet the problem statement.*

Question 1.5 *rely on ECLiPSe's debugger to design the Prolog search tree when searching the solution* `commande(-NbResistance, -NbCondensateur)`.
(NB: we do not ask for the trace in the report, only the tree.)

Question 1.6 *Justify the title of the year (tracks: what will happen if we put the predicate* `>=` *before the calls to* `isBetween` *? Why we use and call it test-driven development?)*

1.3 The solver `ic` to the rescue

Question 1.7 *Change the sentence* `isBetween` *by the constraint* `ic Var #:: Min..Max` *and* `>=` *by* `#>=`. *See what happens in ECLiPSe. Why is this happening?*

Question 1.8 *Use the* `labeling` *predicate to find solutions to problem (see section labeling page 9) and draw the new search tree Prolog, keep on using the tracer.*

2 Zoologie

Consider `Chatss` cats and `Pies` pies. These `Chats` Cats cats and `Pies` pies total `Pattes` legs and `Tetes` heads. The problem that concerns us therefore includes four variables that are "linked" with numerical constraints.

These variables belong to a finite domain: a sub-set of integers natural. Indeed the negative head numbers or 2/3 means anything for us!

Define a sentence `chapie/4` where `chapie(-Chats,-Pies,-Pattes,-Tetes)` establishes constraint linking the four variables.

Use this predicate to answer the following questions:

Question 1.9 *How many pies and legs does it take to total five heads and two cats?*

Question 1.10 *How much to cats and pies for three times as many legs as heads?*

3 Le "OU" en contraintes

When programming with `Prolog + ic`, the " or " logic can be implemented in two ways:

- With a choice point Prolog.
- With the disjunction operator `or` from *ic*.

Question 1.11 *En utilisant successivement ces deux méthodes, définissez le prédicat* `vabs/2`, *where* `vabs(?Val,?AbsVal)` *impose the constraint :* `AbsVal` *is the absolute value of an integer* `AbsVal`. *Test different versions of this predicate in varying arguments.*

Question 1.12 *Run the query* `X #:: -10..10, vabs(X,Y)` *with both versions of* `vabs`. *Compare and discuss the results.*

Following is defined by:

$$X_{i+2} = |X_{i+1}| - X_i$$

Question 1.13 *Define the predicate* `faitListe(?ListVar, ?Taille, +Min, +Max)` *forcing ListVar to be a size of Size list whose elements are variable* `ic` *so the domain is* `Min..Max` *.*

Question 1.14 *Define the prédicate* `suite(?ListVar)` *which takes a list and constraint the elements of that list is such a way that they are consecutive.*

Question 1.15 *Ask a query to verify that this sequence is periodic of period 9.*

4 Compte-rendu de TP

4.1 À rendre

1. Source code ECLiPSe and queries ECLiPSe used and the system responses.
2. The written answers questions 1.2, 1.6, 1.7 et 1.12, and the search tree for 1.5 et 1.8.

TP 2

Contraintes logiques

In a general way, a constraint is expresed by an elementary operator such as X #> Y. This is a contraint is composed of multiple primitices linked with logic connectors: `and`, `or`, `=>`, `#=` et `neg`. We also call them logic constrints.

We call symbolic constraints those refering to variables within an integer domain.In that case, the domain is defined by multiple terms such as `: Day &:: week`, with `week` the symbolic domain represent the set of `{mo, tu, we, th, fr, sa, su}`.

1 Logic puzzle

To solve this puzzle you need two libraries.

- the library *ic_symbolic* for symbolic constraints (which operators start with `&`)
- lthe library *ic* for logical connectors and constraints on integer variables (whose operators begin with `#`)

A predicate `alldifferent(+List)` exists in both libraries and implements global constraint which forces variables in the list `List` to have a different value in the same field. The list must contain only variables same field.

NB: When a predicate with the same name and is in the same arity two libraries, use the syntax `bibliothèque:prédicat` to specify which one to use.

1.1 Enoncé

A street nearby houses contains 5 different colors. the residents of each house are of different nationalities, possess different animals, different cars and all a different favorite drink. In addition, the houses are numbered January to May in the order they are in the street.

We know more about these houses:

(a) The Englishman lives in the red house
(b) The Spaniard owns a dog
(c) The person living in the green house drinks coffee
(d) The Ukrainian drinks tea
(e) The green house is located just to the right of the white house
(f) BMW's driver has snakes
(g) The inhabitant of the house has a yellow Toyota
(h) Milk is drunk in the middle house
(i) The Norwegian lives in the leftmost house
(j) The driver of the Ford lives next to the person who owns a fox
(k) The person driving a Toyota lives next to the house where there is a horse
(l) Honda driver drinking orange juice
(m) The Japanese drove a Datsun
(n) The Norwegian lives next to the blue house

We want to know who owns a zebra and who drinks water.

1.2 Modélisation

A house is represented by a term `m(Pays,Couleur,Boisson,Voiture,Animal,Numero)` or `Pays`, `Couleur`, `Boisson`, `Voiture`, `Animal` and `Numero` are six variables representing the houses characteristics. The variables of the problem are represented by a five-member list where each element is a term `m(...)`. The first element of the list is the home leftmost and is number 1, the second element is the second house from the left and carries the number 2, etc.

1.3 Questions

Question 2.1 *Define the different symbolic domains through libraries from ECLiPSe (cf. documentation of* ic_symbolic*).*

Question 2.2 *Define a predicate* `domaines_maison(m(...))` *which forced the field of variables that make up a home.*

Question 2.3 *Define the predicate* `rue(?Street)` *that unifies* `Rue` *to the list of houses and pose constraints field. NB: this predicate must set the value of variables* `Numero` *of each home.*

Question 2.4 *Write the predicate* `ecrit_maisons(?Rue)` *defines iterator that retrieves each element of* `Rue` *and writes means of the predicate* `writeln/1`*. You will need this type iterator in the following questions.*

Question 2.5 *Define the predicate* `getVarList(?Rue, ?Liste)` *allowing retrieve the list of variables of the problem.*

Then set a predicate labeling `labeling_symbolic(+Liste)` *using the predicate* `indomain/1` *de* ic_symbolic *(cf. page 9).*

Question 2.6 *Define the predicate* `resoudre(?Rue)` *using predicates previous to find a solution within the constraints of field. Check that the solutions given by ECLiPSe are consistent.*

Question 2.7 *Ask the constraints corresponding to the information of (a) to (n) adding to as the predicate* `resoudre` *and verify that the proposed solutions are correct.*

Question 2.8 *Answer the question posed in the statement : has a zebra and drinking water ?*

2 Compte-rendu

2.1 Questions de compréhension

1. In the question 2.3, you set the value of variables representing the number of a house on the street. Would it have been possible not to set these values? What would have been the impact on the search for solutions?

2.2 À rendre

1. The code ECLiPSe *commented*. Give queries ECLiPSe and the responses of the system and the test data when the data are not the problem.

2. the answer to the question 2.8.

TP 3

Task scheduling on two machines

1 Problem statement

Consider a plant with two machines and to produce parts. Each piece is the result of a sequence of tasks takes place on one of the two machines. NB: machines are specialized and can not be used interchangeably.

Each is described by its duration, the machine on which it must be running and the list of preliminary tasks to be completed before it starts. A task can at most run on the same machine at a given time.

The problem is presented in the Figure 3.1. Each node represents one task. The values linked with the nodes are the task id and its duration. The tasks executed by the 1st machine are represented by a double circle. The others represent the tasks executed by the 2nd machine. The links represent the dependencies in between different tasks. A tack can only starts if all its previous dependencies have finished. The job is done when all tasks are done.

This is to find start times for each task, in accordance with these constraints.

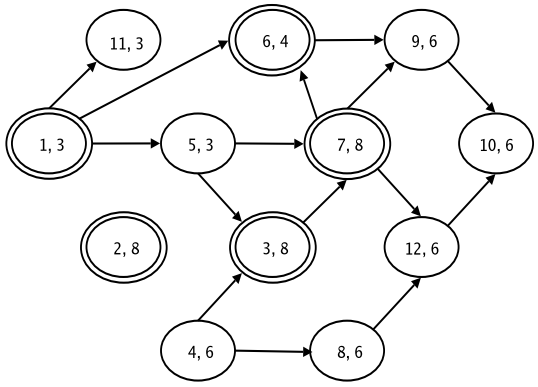


Figure 3.1: Dépendances entre les tâches

2 Modeling knowledge

We define a symbolic area for machine consists of two values `m1` and `m2` (This allows the use of symbolic constraints between variables representing machines). It represents a task by a term of the form:

`tache(Duree, Noms, Machine, Debut)`,

where `Duree` is an integer giving the duration of the task in minutes `Noms` is a list of clues preliminary tasks in the table tasks, `Machine` is the name of the machine and `Debut` is a variable representing the start date. It represents the data and variables of the problem by an array of such

words:

```
[](tache(3, [],     m1, _),
   tache(8, [],     m1, _),
   tache(8, [4,5],  m1, _),
   tache(6, [],     m2, _),
   tache(3, [1],    m2, _),
   tache(4, [1,7],  m1, _),
   tache(8, [3,5],  m1, _),
   tache(6, [4],    m2, _),
   tache(6, [6,7],  m2, _),
   tache(6, [9,12], m2, _),
   tache(3, [1],    m2, _),
   tache(6, [7,8],  m2, _))
```

Note that in this problem, the machine that runs each task is known.

3 A first ECLiPSe solution

3.1 A partial solution

Question 3.1 *Define a predicate* `taches(?Taches)` *that unify* `Taches` *with a task table.*

Question 3.2 *Set the iterator that retrieves each element and the poster. You will need this type of iterator for following questions.*

Question 3.3 *Define a predicate* `domaines(+Taches, ?Fin)` *which requires that any task* `Taches` *starts after time 0 and finish before the end of (additional variable corresponding to the moment when all tasks are completed).*

Question 3.4 *Define a predicate* `getVarList(+Taches, ?Fin, ?List)` *which allows to retrieve the list of variables from the problem.*

Question 3.5 *Define the predicate* `solve(?Taches, ?Fin)` *which uses the previous three predicates to find a schedule that respects the constraints of areas. Check that the solutions given by ECLiPSe are coherent.*

3.2 Posing scheduling constraints

NB: For the following two questions, test your response using simple data.

Question 3.6 *Define a predicate* `precedences(+Taches)` *qui contraint chaque tâche à démarrer après la fin de ses tâches préliminaires. Modifiez* `solve` *pour prendre en compte ces contraintes et vérifiez que les nouvelles solutions proposées par ECLiPSe sont correctes.*

Question 3.7 *Define a predicate* `conflits(+Taches)` *qui impose que, sur une machine, deux tâches ne se déroulent pas en même temps. Modifiez* `solve` *pour obtenir une solution du problème prenant en compte cette dernière contrainte.*

3.3 The best solution ?

In this problem, there is a notion of relative quality of solutions: if a solution S_1 is such that the value of `Fin` is less than that of the solution S_2, then S_1 is better than S_2. We can be interested in the research the best solution of the problem or the best solutions.

Question 3.8 *Do you think the solution found by eclipse is the best solution? If so, explain why. Otherwise modify your code to find an optimal solution.*

4 Homework

4.1 To deliver

1. The code ECLiPSe *commenté*. Give eclipse requests used to test each question and test data when the data are not the problem. For each request, also give the system responses.

2. The detailed answer to the question 3.8

TP 4

The races

1 Problem Statement

The organizers of a regatta can have 3 yachts each to host a number of teammates. The accommodation capacities for the yachts are:

7	6	5

4 teams responded to the invitation of the organizers. The size of these invited teams is given by the following table:

5	5	2	1

The puzzle of the organizers then is to try to make a schedule with the following requirements:

- The regatta takes place in 3 successive confrontations in which 3 sailboats compete;
- invited teams are spread over the 3 boats so that :
    1. members of the same team stay together,
    2. sailboats of accommodation capacities are met,
    3. during successive confrontations no team returns two time on the same boat,
    4. during successive confrontations no team finds with the same partner team.

The goal is to establish a schedule two entries for any confrontation and for any guest team indicates the number of host sailboat. the schedule below is a possible answer. It states that during the confrontation 3 Team No. 4 is on the boat 2; it is then Team Partner 2.

Équipe \ Confrontation	1	2	3
1	1	2	3
2	3	1	2
3	2	3	1
4	1	3	2

2 Looking for a solution using ECLiPSe

2.1 A first partial solution

The problem data are in the form of two vectors: the vector of terminal facilities and the vector of the sizes of invited teams. The variables of the problem are the number of `NbEquipes * NbConfrontations`. These variables, representing a host sailboat, have a field `1..NbBateaux`. In the following, the variables are contained in an array with `NbEquipes` rows and `NbConfrontations` columns.

Question 4.1 *Define the predicate* `getData(?TailleEquipes, ?NbEquipes, ?CapaBateaux, ?NbBateaux, ?NbConf)` *that unifies the variables as parameters with the data of problems.*

Question 4.2 *Define the predicate* `defineVars(?T, +NbEquipes, +NbConf, +NbBateaux)` *which unifies* `T` *the array of variables described above and forced the field variables.*

Question 4.3 *Define the predicate* `getVarList(+T,?L)` *which constructs the list* `L` *with the variables in the table*`T`*. The variable list must contain the variables of the first column followed by those in the second column, etc. (a test that shows that the order is correct).*

Question 4.4 *Define the predicate* `solve(?T)` *qui résoud le problème des régates où seules les contraintes de domaines sont posées. Vérifiez que les réponses rendues par eclipse vous semblent correctes.*

2.2 Taking into account problem constraints

Question 4.5 *Define the predicate* `pasMemeBateaux(+T, +NbEquipes, +NbConf)` *which requires that the same team will not return twice on the same boat. Modify the predicate* `solve` *to take into account this new constraint and check that the answers given by eclipse are consistent.*

Question 4.6 *Define the predicate* `pasMemePartenaires(+T, +NbEquipes, +NbConf)` *which requires that the same team is not found twice with the same team. Modify the predicate* `solve` *to take into account this new constraint and check that the answers given by eclipse are consistent.*

Question 4.7 *Define the predicate* `capaBateaux(+T, +TailleEquipes, +NbEquipes, +CapaBateaux, +NbBateaux, +NbConf)` *which verifies that the capacity of ships are met at every confrontation. Modify the predicate* `solve` *to take into account this new constraint and check that the answers given by eclipse are consistent.*

3 let's go ahead with a real-world size problem

The data used so far are only test data; it would be quite possible to solve the problem without using a computer. In this section, we consider data more representative of the puzzle that can be organizing a regatta:

We have 13 sailboats whose capacities are listed below:

10	10	9	8	8	8	8	8	8	7	6	4	4

The number of teams going to 29. The size of each team:

7	6	5	5	5	4	4	4	4	4	4	4	4	3	3

2	2	2	2	2	2	2	2	2	2	2	2	2	2

The regatta takes place in 7 confrontations. First try with less confrontation and gradually try to 7 confrontations.

Finding a solution to this new problem can be very long! he you should program your own labeling. Idea to dig: when you have the list of 29 variables of a confrontation without doubt it is preferable to get faster a first solution, reorder the list so as to alternate small and large team.

Question 4.8 *Implement* `getVarListAlt(+T,?L)` *to solve this problem in a reasonable time.*

The following configuration is a solution of the problem:

Équipe \ Confrontation	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	1	4	3	6	5	8
3	3	4	1	2	8	7	9
4	4	3	5	1	2	9	10
5	5	6	2	7	1	3	4
6	6	5	7	2	1	4	3
7	6	7	8	5	2	1	11
8	7	5	6	8	9	1	2
9	8	9	7	10	4	11	5
10	8	10	9	6	11	3	2
11	9	8	12	13	7	2	6
12	10	9	8	11	13	12	1
13	12	13	11	9	10	8	1
14	11	10	13	3	8	9	4
15	11	12	10	7	3	13	9
16	13	11	10	9	6	1	5
17	13	8	11	12	4	10	3
18	10	11	9	8	12	2	3
19	9	11	6	12	10	5	13
20	9	7	10	11	12	8	2
21	7	8	9	10	3	4	1
22	7	6	5	9	11	2	8
23	5	7	1	8	3	10	13
24	4	1	6	5	3	2	12
25	3	2	4	1	7	11	12
26	3	1	2	6	9	10	11
27	2	4	3	1	9	8	6
28	2	3	1	6	7	4	5
29	1	3	2	5	4	7	6

4 homework

4.1 To deliver

1. The answer to the problem.
2. The code ECLiPSe *with comments*. Give eclipse requests and the system responses.

TP 5

Constraining thus researching

Generally the constraints programs include two stages, the first is to express the problem and the second is to find a variable assignment that satisfies constraints, ie a solution to the problem. We may want find any solution, all solutions or solution optimal (or 'good 'according to a given criterion). In this problem we intéresserons the latter type of solution.

1 The problem

Consider the following problem: a production unit can produce various mobile phone products. The manufacture of the so S N mobilizes workers and produces mobile phones P a day, for each of these phones benefit is E euros. The unit has 22 technicians.

It has the following data:

Sorte	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
Nb. Tech	5	7	2	6	9	3	7	5	3
Qté jour	140	130	60	95	70	85	100	30	45
Bénef	4	5	8	5	6	4	7	10	11

We want to know what you have to start manufacturing for the benefit.

2 Model and constrain

Before coding it is imperative model, that is to say to the problem equations. To this we must look for a model general. The problem data are in the form of three values of vectors: `Techniciens`, `Quantite`, `Benefice`. The variables form a vector`Fabriquer` which the values will be determined by your program. These values are either 1 or 0, depending on whether or not the lance production.
The program must be composed as follows:

1. part data (facts, variables);
2. part « services predicates (scalar product, ...);
3. part constraints predicates (equations expressing the problem of properties);
4. part resolution constraints, a predicate that:
   - « recovery » data and variables of the problem,
   - asks the constraints on these variables.

Question 5.1 *Write predicates that define the three vectors values and the vector of variables.*

Question 5.2 *Define the predicates which express from defined vectors:*

- *the total number of necessary workers*
- *the vector of total profit by kind of telephone*
- *the total profit*

These equations represent predicates using vector operations.

Question 5.3 *Define the predicate:*
`pose_contraintes(?Fabriquer, ?NbTechniciensTotal, ?Profit)` which poses constraints, then call and list the solutions respect these constraints (there are many!).

3 Optimizing

3.1 Branch and bound within ECLiPSe

To make the optimization under the constraints, the algorithm *branch and bound* is used.

In ECLiPSe, the predicate `minimize/2` from the library *branch_and_bound* is an implementation of that algorithm. the first parameter is a goal. This object generates a search tree which performs the list of possible solutions. The second parameter is a variable representing the cost of the solution. It is this variable that should be minimized.

The predicate `minimize / 2` works on the following principle: as soon as a solution is found, it is stored and research continues with the additional constraint that the cost is less than stored. By repeating this process until no longer find solution, an optimal solution is obtained.

NB: `minimize / 2` requires that the object instantiates the variable cost but not the rest of the variables. In this case, it may there are no solutions to the problem to obtain the value optimum found. The following example illustrates this property:

```
[X,Y,Z,W] #:: [0..10], X #= Z+Y+2*W , X #\= Z+Y+W
```

Question 5.4 *Use `minimize/2` to find the smallest value `X` solution of this example. Try the labeling only `X` and finally `[X, Y, Z, W]`. Explain the two responses made by different ECLiPSe. What must always be in the purpose?*

3.2 Application to our problem

Question 5.5 *Call the predicate poses constraints and seek "optimal" solution (maximizing profit) and meets the constraints posed.*

Times change. The mobile market begins to saturate. shareholders worry. So the company policy fits. We want to produce less but earn at least 1 000 per day. We want to keep making mobile lines that ensure this goal but keeping the minimum workers!

Question 5.6 *Use the previous approach to address this new problem. There not much to do (same data, same services predicates, . . .) !*

4 Homework

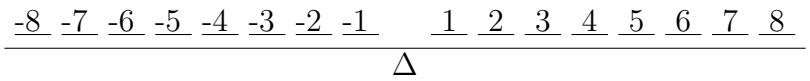
4.1 To do

1. The answer to the problem.
2. The commented code. Deliver the queries ECLiPSe and the system responses.
3. The detailed answer to the question 5.4.

TP 6

Using a swing

A family of 10 arrived at the park and wants to use a swing 16 seats in the schematic drawing below (seats are spaced 1 meter, except the 2 separate head offices 2 meters)



The weights of the family members are given in the table below :

24	39	85	60	165	6	32	123	7	14
ron	zoe	jim	lou	luc	dan	ted	tom	max	kim

The constraints of the problem are as follows :

1. Once installed our 10 people, **the swing must be balanced** (time left = right time)
2. Lou and Tom, the mom and dad of the siblings of 8 children, wish coach their children in order to monitor
3. Dan and Max, the two Young people are on two opposite sides, just in front of their mom or dad
4. there are five persons on each side

We are seeking a solution to this problem that minimizes forces moments standards.

Remember: each person X sits down in the swing introduce a force (their weight), note $\overrightarrow{P_X}$. If d_X is the distance between X and the swing center, then the norm of the moment exerted on the axis of rotation of the swing by $\overrightarrow{P_X}$ is equal to $\|\overrightarrow{P_X}\| \times d_X$

Here the data of the problem is in the form of a vector of values : Weight.

The variables (representing places) also form a vector whose *Places* values will be determined by your program. These values belong to domain $[-8..-1] \cup [1..8]$. In *Places* and *Poids* rank refers implicitly the person. Although you have to use the vector *Places* as such, you will still need to name some variables appear explicitly in constraints.

1 Find a solution to the problem

The program will be composed as follows :

1. part data
2. part « services predicates »
3. part defining constraints
4. predicate of calculating a solution using predicates other parts.

Note: as in previous works, it is essential to test code as His. So you-have to start by putting in place predicates That can ask ECLiPSe to find a solution. When adding an additional constraint, Then you can easily check the impact of this new constraint on the solutions Given by ECLiPSe.

Question 6.1 *Write the program that defines the data and sets the constraints of the problem. NB : ic provides various arithmetic constraints that you can use for this lab (abs/2, min/2, ...). Feel free to search the documentation of ECLiPSe!*

Question 6.2 *Ask ECLiPSe to find a solution.*

Question 6.3 *What symmetry may appear in the solutions to this problem? Remove this symmetry.*

What is the impact of this disposal on finding solutions?

2 Finding the best solution

Question 6.4 *Use the predicate* `minimize` *from the library* `branch_and_bound` *to find the best solution to the problem.*

The search for the best solution may be very long, so it is important to help the system to quickly find a good solution. For this, the predicate `search / 6` allows you to control the list in two ways:

- the order in which the variables are instantiated
- the order in which each value of the domain of a variable is tested

Version 1 *(see* `search/6` *in the doc) In this version the first variables considered are those involved in the more constraints (we try to fail as soon as possible to avoid developing unnecessary branches : « To succeed, try first where you are most likely to fail ! »)* and *for a given variable values are tried in ascending order.*

If wait too long before finding an optimal solution, interrupt !

Version 2 *(see* `search/6` *and* `get_domain_as_list` *dans la doc) In this version the variables are considered in the order of the list,* and *for a given variable values are tested in a sequence suitable to the problem addressed. (In the order of values depends on the order of development of branches when seeking an optimal solution is often an advantage in quickly finding a "good" solution because it will allow a more efficient pruning The order of values depends on. the linear form to be optimized.)*

If wait too long before finding an optimal solution, interrupt !

Version 3 *Combine versions 1 and 2.*

Version 4 *Heuristic of choosing the order of the variables calculate a score for each variable according to a specific criterion (see doc* `search`*). When all variables have the same score, the order in the initial list is used. Thus, the initial order of the variables can be very important even when using search heuristics.*

Submit an initial variable order adapted to the problem and verify the impact of this order on the time to search for the optimal solution.

Note: To reduce the field at the earliest places of Tom and Lou it is possible to state redundant constraints.

3 Homework

3.1 Questions of understanding

1. At the end of the year you have been asked to write your own labeling process. What does the original labeling of ECLiPSe and why is it not effective for the problem addressed ?

3.2 To deliver

1. The answer to the question of understanding.
2. The code ECLiPSe. Give queries eclipse and the system responses.

3. The answer to the question 6.3.

4. A justification of the choices you have made for versions 2 and 4 of the enumeration strategies.

TP 7

Les mobiles de Choco

1 Des contraintes sans Prolog

Les problèmes concrets de l'industrie utilisant les contraintes ne sont pas souvent résolus entièrement en Prolog. Il existe alors plusieurs solutions pour utiliser vos connaissances en programmation par contraintes :

- utiliser une interface entre Prolog et le langage utilisé dans le reste du projet (par exemple ECLiPSe s'interface très bien avec JAVA et C++) ;
- utiliser un autre moteur de résolution des contraintes (i.e. un autre solveur) dans un autre langage.

C'est cette dernière solution que nous allons explorer dans ce chapitre. Il existe beaucoup de moteurs dans différents langages, pour n'en citer que quelques uns :

- en C++ : Disolver, gecode, ...
- en Java : JaCob, JCK, JCL, Choco, ...
- en Ocaml : FaCiLe, ...

Nous allons utiliser **Choco** pour résoudre un problème simple dans le domaine des entiers dans un programme Java. Vous trouverez en annexe A un descriptif des fonctionnalités de Choco pour poser et résoudre les problèmes. Vous pouvez aussi aller voir sur le site de Choco `http://www.emn.fr/z-info/choco-solver/index.html`.

Nous vous fournissons une grosse partie du code Java, il ne reste que la partie programmation des contraintes à réaliser. Ce code est fourni à l'emplacement `/home-info/commun/4info/Contraintes_choco` de votre machine.

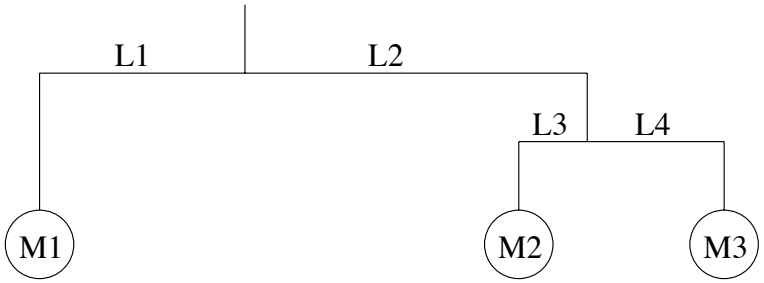


Figure 7.1: Le mobile à équilibrer

2 Le problème : équilibrer un mobile

Le problème que nous considérons est l'équilibrage d'un mobile de structure fixée décrite par la figure 7.1 (il n'est pas demandé de résoudre le problème dans le cas général, même si la programmation objet de Java le permettrait facilement).

Le problème comporte donc 7 variables entières (problème dans le domaine fini) :

- les longueurs des branches : L1 et L2 pour le premier étage et L3 et L4 pour le second ;
- les masses M1, M2 et M3 au bout des branches.

Le mobile est soumis à quelques contraintes (physiques). Le mobile doit pouvoir tourner (une contrainte entre les longueurs à valeurs entières) sachant que les masses sont de tailles négligeables. Le mobile doit être équilibré (ce qui correspond à deux contraintes entre les variables). Pour équilibrer le mobile on dispose de 20 masses différentes, dont les poids à valeur entière s'échantillonnent de 1 à 20g.

3 Les problèmes à résoudre

Question 7.1 *Écrire les programmes qui résolvent les deux problèmes qui suivent.*

1. L'utilisateur entre des valeurs pour les longueurs des branches, vous devez vérifier que ces longueurs sont cohérentes ;
2. Trouver les masses M1, M2 et M3 qu'il faut choisir parmi les poids disponibles pour équilibrer les mobiles (ce qui n'est pas toujours possible mais parfois il existe plusieurs solutions).

Il faut définir les variables et leurs domaines puis poser les différentes contraintes. Il vous sera indiqué en TP quelles méthodes Java compléter.

TP 8

Histoire de menteurs

1 Le puzzle

Parent1 et Parent2 forment un couple hétérosexuel, mais on ne sait pas qui est la femme ni qui est l'homme ! Ces deux personnes ont un enfant Enfant dont on ne connaît pas le sexe.

Le seul critère permettant de discerner femmes et hommes est le suivant :

1. Les femmes disent toujours la vérité.
2. Les hommes alternent systématiquement vérité et mensonge.

(Selon vos convictions, vous pouvez bien sûr adapter l'énoncé . . .)

On demande à Enfant s'il est un homme ou une femme :

Enfant affirme : Arrheu, arrheu !

On se tourne alors vers ses parents Parent1 et Parent2.

Parent1 affirme : Enfant vous dit qu'elle est une femme.
Parent2 affirme : Enfant est un homme puis . . .
Parent2 affirme : Enfant ment.

On cherche à savoir qui est le père, qui est la mère et quel est le sexe de l'enfant.

2 Modélisation

Modéliser ce puzzle logique en utilisant les contraintes logiques de la bibliothèque *ic*. Pour cela, identifier les variables et leur domaine.

Question 8.1 *« Les femmes disent toujours la vérité. ».*
Définir le prédicat `affirme/2` *tel que* `affirme(?S,?A)` *pose la contrainte : l'affirmation* `A` *est vraie si* `S` *est une femme.*

Question 8.2 *« Les hommes alternent systématiquement vérité et mensonge. »*
Définir le prédicat `affirme/3` *tel que* `affirme(?S,?A1,?A2)` *pose la contrainte : si* `S` *est un homme, les affirmations A_1 et A_2 sont l'une vraie, l'autre fausse.*

AffE est l'affirmation de l'enfant.
AffEselonP1 est l'affirmation de l'enfant selon Parent1.
AffP1 est l'affirmation de Parent1.
Aff1P2 est la première affirmation de Parent2.
Aff2P2 est la seconde affirmation de Parent2.

Chacune de ces affirmations appartient au domaine booléen $\{0, 1\}$ (représenté par l'intervalle entier [0..1]) :

AffEselonP1 :	Enfant est une femme
AffP1 :	AffEselonP1 = AffE
Aff1P2 :	Enfant est un homme
Aff2P2 :	AffE = 0

Question 8.3 *Définir le domaine symbolique des variables* `Parent1`, `Parent2` *et* `Enfant`, *puis écrire le prédicat qui contraint le domaine de l'ensemble des variables.*

Question 8.4 *Poser les contraintes sur les variables et définir un prédicat de labeling pour les valeurs symboliques (comme pour le TP 2). Utilisez ce prédicat pour résoudre le problème.*

Appendix A

Utilisation du solveur Choco

Cette annexe présente les éléments utiles pour réaliser le TP sur les mobiles avec le solveur Choco, écrit en Java. Pour plus de renseignements, se référer au site:

`http://www.emn.fr/z-info/choco-solver/index.html`

Faire appel à la bibliothèque Choco

La bibliothèque Choco est disponible sous forme de .jar. Sous l'environnement de développement pour Java ECLiPSe, il suffit d'inclure le chemin vers la bibliothèque dans le classpath du projet (menu Project, Properties, Java Build Path, onglet Libraries, bouton Add External JARs ...).
Il faut importer en tête de fichier les classes de Choco qui seront utilisées. Dans notre cas:

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

Architecture générale choco

On identifie clairement deux parties différentes sur le schéma A.1:

1. modélisation du problème via la classe Model
2. résolution du problème via la classe Solver

Détaillons ces étapes.

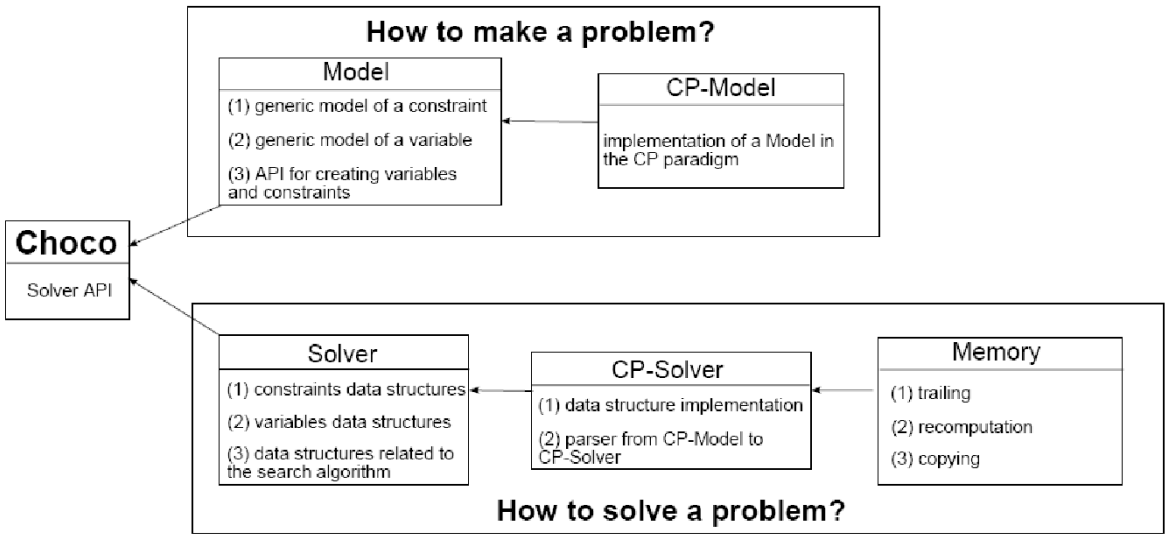


Figure A.1: Architecture choco

Modéliser un problème

En Choco, le problème à résoudre par programmation par contraintes est une instance de la classe `Model`.
Constructeur d'un objet de type `Model` : `CPModel()`

À un objet de type `Model`, on associe:

1. des variables. Dans notre cas, il s'agit de variables entières à domaine fini.
   - Déclaration:
     `IntegerVariable makeIntVar(String nom,int borne_min,int borne_max)`
     où `nom` est le nom de la variable, `borne_min` et `borne_max` sont les bornes inférieure et supérieure de son domaine de définition.
   - Association à l'objet `Model`:
     `void Model::addVariable(IntegerVariable var);`
2. des contraintes.
   - Déclaration: on dispose d'un large panel de contraintes. Les plus utiles sont listées dans le paragraphe ci-dessous.
   - Association à un objet `Model`:
     `void Model::addConstraint(Constraint c);`

Imports en tête de fichier : `import choco.cp.model.CPModel;`
`import choco.kernel.model.variables.integer.IntegerVariable;`

Exprimer une contrainte

Les contraintes sont exprimées sur des instances de la classe `IntegerExpressionVariable`. Cette classe permet de représenter des expressions linéaires entières (nous le notons par la suite IEV pour des raisons de lisibilité). La classe représentant les variables sur les domaines finis est `IntegerVariable`, elle hérite de `IntegerExpressionVariable`.

Opération d'addition et de multiplication :
`IEV plus(IEV t1, IEV t2)`
`IEV mult(int i, IEV t)`

Contrainte d'égalité :
`Constraint eq(IEV t1, IEV t2)`

Contrainte de stricte infériorité :
`Constraint lt(IEV t1, IEV t2)`

Contrainte allDifferent :
`Constraint allDifferent(IV[] tab)`
où `tab` contient les variables dont on veut contraindre les valeurs à être toutes différentes

Résoudre un problème

Afin de résoudre un problème, on va déclarer un solveur, instance de la classe `Solver`, et l'associer au modèle.

Constructeur d'un objet de type `Solver` :
`CPSolver()`

Lecture du modèle par le solveur:
`void Solver::read(Model model)`

Lancer la résolution du système de contraintes :
`java.lang.Boolean Solver::solve()`
rend vrai si une solution est trouvée. Les variables sous contraintes sont instanciées avec les valeurs formant cette solution.

Trouver une nouvelle solution :
`java.lang.Boolean Solver::nextSolution()`

rend vrai si une nouvelle solution est trouvée.

Il n'est pas possible d'accéder à la valeur ou au domaine d'une variable lorsque le processus de résolution a été appliqué par l'intermédiaire du modèle (l'instance `Model`). Il faut pour cela utiliser le solveur.

Récupérer le domaine de la variable :
`IntDomainVar Solver::getVar(IEV var)`

Accéder à la valeur si instanciée (valeur pour la solution courante):
`int IntDomainVar::getVal()`

Import en tête de fichier :
`import choco.cp.solver.CPSolver;`