A CLP framework in Java

Lígia Ferreira and Salvador Abreu

Departmento de Informática
Universidade de Évora
PORTUGAL
{lsf,spa}@di.uevora.pt

Abstract

This paper discusses some basic ideas about the implementation of *GC*, a constraint propagation system, with three different implementations of finite variables domains. As it has been our wish to develop a system that uses the OO approach, it is implemented in Java, although other application frameworks are also available for C++.

The constraint propagation and variables domains are explained at the application and implementation level and some examples from the litterature are presented in order to clarify the construction of applications with *GC*.

1 Introduction

Many real life problems (scheduling, planning, etc) found in constraint programming, and particularly in constraint logic programming, a very convenient formulation. This is due to the expressiveness of this paradigm and its suitability to efficiently handle highly combinatorial problems.

Combining a programming methodology with a particular existing language, can be helpful because the result will inherit the host language's qualities but, in the case of CLP, one of Prolog's most significant handicaps is also passed on to the final programming language: it's lack of widespread diffusion. This aspect severely limits the ease with which a technology which is known to be useful (CLP) may be demonstrated and actually put to use in non-specialized environments.

The Java language is touted as having absolute machine-independence, thereby providing an appealing platform for the development of applications. This has been slightly hindered by the efficiency of the available Java implementations, but this situation is changing with the emergence of better

run-time support systems, for instance by way of the "just-in-time compilers", such as TYA.

The inefficiency of the implementations of JVM, the Java virtual machine, may may seem to be a deterrent to using it to implement other programming paradigms, especially ones which focus on (relative) efficiency. However, in the case of CLP, the appeal of having a widespread binary-compatible platform seems to address the difficulty that such systems have and was previously mentioned: scarcity of a widely used (and easy to install) implementation. Just consider that most Web browsers have a built-in Java runtime system, regardless of the hardware/software platform they run under.

This situation has been recognized and dealt with, albeit with a different language (C++), through the constraint programming package Ilog:Solver [6]. Our goal is to take this approach one step further and construct a similar but improved system in Java. At this stage raw efficiency (as per benchmark results) is not our purpose, we'd rather design and implement a clean framework for CLP in a Java embedding.

Another feature of the Java language which is paramount to the possibility of success of this choice is its inclusion of advanced graphical toolkits which share the language's most hailed benefit: portability. These toolkits, especially the Java Foundation Classes or *Swing*, will enable us to build sophisticated programming environments and extend our research in the direction of Visual Programming Languages.

This article outlines the *GC* in section 2. In section 3 we discuss different representations for variables. Section 4 presents the way in which *GC* implements constraints and section 5 discusses the mechanisms for constraint propagation. Section 6 describes *iterators*, which are used to artificially reduce some variable's domain. In section 7 we present a few examples from the literature and finally, in section 8 we discuss some of our short- and long-term plans for *GC*.

2 The *GC*

In order to obtain a dialect for an OO language that allows constraint programming methodology, there are two independent, but parallel, class hierarchies: the Variable and Constraint classes. Communication and synchronization between both is obtained by reference relations. Constraints contain Variables (in the environment class attribute) and Variables refer the set of constraints they occur in (by dependencies class attribute).

Figure 1 shows a summarized class hierarchy for *GC*.

For instance, if we have three variables X, Y and Z and the constraints C_1 and C_2, given by:

Fig. 1. Class hierarchy for GC

$$C_1 : X + Y = Z$$

$$C_2 : X \neq Y$$

The environment of C_1 will contain variables X, Y and Z and the environment of C_2, will contain X and Y. On the other hand, variable X will have the following dependencies: $(C_1, Y), (C_1, Z), (C_2, Y)$ and Y will have $(C_1, X), (C_1, Z), (C_2, X)$.

Applications will have instances of subclasses of Variable, Constraint and Iterator (to impose single value solutions). The architecture of `GC`, comprises three levels: the Kernel level, Constraints level and Application level. At the Kernel level, all the basic framework for constraints and variable domains is implemented. The Constraints level implements specific domains (e.g. FD, Booleans etc) and basic constraints over these domains ($<, \leq, =, \neq$, etc). Finally the Application level, which uses the previous two and may specify new constraint classes, specific to the problem in hand.

3 Domain Variables

These represent the values associated with the variables and contain a set of constraints in which this variable occurs. At the present moment we have implemented three domain Variable classes, which represent variables over finite domains. `FD` variables, which assume that the values for the variable are represented by a unique interval of positive integers, so its representation relies

on minimum and maximum values. `FDD` variables are `FD` variables in which a bitmap is added to represent the values in the domain and `FDIU` variables, also `FD` variables, in which domains are represented by a disjoint union of intervals.

In the hierarchy of domain variables the generic work is done in `gc.variable`. It implements the work related to the dependencies and lets the work specific of the domain to be done in subclasses. These must respond to messages like: whether a domain is ground, empty, copy variables and clone them with no dependencies, give the first element in the domain, give the next value given the previous one and set a single range.

3.1 Working with FD *Variables*

They are implemented in `gc.fd.Variable` and are a subclass of `gc.Variable`. Since we assume that the possible range for the variable is a compact domain, `FD` variables work fine. Dealing with this kind of variables implies the handling of domain extremes only. Constraints over these variables are restricted to evaluate the maximum and minimum values of the domain and to `update` them conveniently.

3.2 Working with FDD *Variables*

As we frequently have holes in the domain, this representation seems to work out better. It's a subclass of `gc.fd.Variable`, to which a bitmap is added to represent the values in the domain. Commonly we use `clear` and `put` methods, to add and remove values from the domain and constraints over these variables make large use of these methods.

3.3 Working with FDIU *Variables*

¿From the discussed so far, it's clear that when we have a large, but compact domain, we must use `FD` variables and when we have to affect values in the middle of the domain, we use `FDD` variables. So, what about large domains with many/few holes?

Using a bitmap to represent the domain values could be very inappropriate, when we have large domains or a variable that is initialized from $0..\infty$. A condensed form for representing series of contiguous integers is of course an interval, since we only need to store the minimum and the maximum values. Unions of intervals are the natural way to represent, in a condensed form, large and sparse sets of values. `FDIU` variables use this representation. The package `gc.fdiu.domain` defines a domain as a list of intervals and exports methods to add and remove intervals from a domain. Some basic operations for intervals (and domains) such as, disjunction, union, sum, difference, etc, are also available.

4 Constraints over Variables

No doubt constraints and the propagation mechanism, are the core of `GC`. Constraints are relations between variables and/or entities external to the constraint propagation system (e.g. constants). The addition of a constraint to the store will create the dependencies between the variables involved in it. The imposition of a constraint will frequently narrow the variable's domain. Once that happens all the dependent variables' domains will be analyzed and updated too, if necessary.

Constraints classes follow an inheritance hierarchy. New constraints subclasses could be added with the definition of appropriate `localUpdate(n)`, method. `localUpdate` is responsible for the actualization of the n^{th} variable domain for that constraint and it's called by `update` method (see section 5), so at this level no propagation takes place and there's no need to check for any kind of changes. The definition of a constraint becomes simple and the single work to do, is to fit the constraint into the proper place into the hierarchy and to define the method for updating all variables' domains of the constraint.

5 Constraint Propagation

The root of constraint class hierarchy defines the `update(n)` method, which is responsible for the propagation mechanism and is triggered by the installation of the constraint. It's main responsibility is the updating of the n^{th} variable of the constraint. If there's a change in the corresponding domain, all the constraints involving that variable will be re-evaluated, until a fixpoint is reached.

6 Iterators

The purpose of Iterators is to restrict a specified set of variables so that these may become a singleton . The variables that will be iterated on, are the ones added to the iterator. The initialization of the iterator is required, in order to prepare it for the first solution. Next solutions in sequence are obtained by, iterating through all possible values (and thereby triggering the propagation mechanism). This is achieved by sending a message to the Iterator instance.

Domain classes must provide methods to select the first value in a domain and the next value, given the previous one, since they are domain-dependent.

The search strategy used is a prolog-like depth-first, left-to-right, but others may be implemented (breath-first, first-fail, etc) as subclasses of Iterator. Since iterators backtrack, a trail data structure is used. This is done with dependency-free variables which store the previous domain value.

6.1 Example

If we define $Xin[2..4] \cup [6..20]$ and $Yin[1..7]$ and the constraint $X < Y$, we'll have $Xin[2..4] \cup \{6\}, Yin[3..7]$. The following table shows the results of iterating over the different variables.

Iterate Over	*Number Solutions*	*All Solutions*
X	4	$[X = 2, Y = [3..7]]$ $[X = 3, Y = [4..7]]$ $[X = 4, Y = [5..7]]$ $[X = 6, Y = \{7\}]$
Y	5	$[X = \{2\}, Y = 3]$ $[X = [2..3], Y = 4]$ $[X = [2..4], Y = 5]$ $[X = [2..4], Y = 6]$ $[X = [2..4] \cup \{6\}, Y = 7]$
X, Y	13	$[X = 2, Y = 3]$ $[X = 2, Y = 4]$ $[X = 2, Y = 5]$ $[X = 2, Y = 6]$ $[X = 2, Y = 7]$ $[X = 3, Y = 4]$ $[X = 3, Y = 5]$ $[X = 3, Y = 6]$ $[X = 3, Y = 7]$ $[X = 4, Y = 5]$ $[X = 4, Y = 6]$ $[X = 4, Y = 7]$ $[X = 6, Y = 7]$

7 Examples

We present three well known examples and the approach used to solve them, in order to clarify some issues about constructing an application with `GC`. The implementation of these solutions is mostly taken from [2].

7.1 The Five Houses Puzzle

- The problem:
  There are five persons who live side by side in the same street. They have different nationalities, jobs, favourite drinks and favourite pets. Based on the constraints given by the problem, find who's who.
- The variables:
  We have 25 variables, to know, the five names, the five nationalities, the five jobs, the five drinks and the five pets. It's better to group the names, nationalities,... Let's define an array of variables
  `N`=$[N_0, N_1, N_2, N_3, N_4]$ for names,
  `C`=$[C_0, C_1, C_2, C_3, C_4]$ for nationalities,
  `P`=$[P_0, P_1, P_2, P_3, P_4]$ for jobs,
  `A`=$[A_0, A_1, A_2, A3, A_4]$ for pets and
  `D`=$[D_0, D_1, D_2, D_3, D_4]$ for drinks.
  All the variables take values from 1 to 5, so they must be created by a

constructor of the type `Variable X=new Variable(1,5)`.

- The constraints:
  Constraints for Five Houses Puzzle, are in the following table.

2 Variables	$N_5 = 1$; $D_5 = 3$; $N_1 = C_2$; $N_2 = A_1$ $D_5 = 3$; $N_3 = P_1$; $N_4 = D_3$; $P_3 = D_1$ $C_1 = D_4$; $P_5 = A_4$; $P_2 = C_3$
3 Variables	$C_1 = C_5 + 1$
Lists	allDifferent(N);allDifferent(C) ;allDifferent(P); allDifferent(A);llDifferent(D)
Disjunctive	$A_3 = P_4 + 1 \vee A_3 = P_4 - 1$ $A_5 = P_2 + 1 \vee A_5 = P_2 - 1$ $A_5 = P_2 + 1 \vee A_5 = P_2 - 1$

  Constraints of the first line are installed by
  `new EQ(N5,new Variable(1)).tell()` for the first constraint and
  `new EQ(P2,C3).tell()` for the last one.
  The 3 variables constraint is installed by
  `new XplusYeqZ(C5,new Variable(1),C1).tell()`
  Establishing that the variables in the arrays must be all different could be done by the following two ways:

(i) Since `n` variables are all different if they are different two by two and $\neq$ is a basic constraint, we can make a method that install all the different constraints.

```
...
for (int i=0;i<5;i++)
   for (int j=i+1;j<5;j++)
     new NE(X[i],X[j]).tell();
...
```

(ii) We can also make use of the `allDifferent` constraint for lists. This constraint expects in the argument an array of domain variables and install the non equal (`NE`) constraints for the pairs in the array. In fact, it uses the same mechanism described above. In this case, we just make `new allDifferent(X).tell()`.

```
public class allDifferent extends gc.fdiu.List.Constraint{
     public allDifferent (Variable List[]) {
         super(List);
     }
     public void localUpdate (int n) {
```

```
            int l=env.length;
            for (int i=0; i<l;i++){
                if (i!=n && env[i].ground()){
                    env[n].clear (env[i].min);
                    env[n].updateMinMax();
                }
            }
        }
    }
```

The remaining constraints are the disjunctive ones.

This is not a basic constraint, so, we must implement it explicitly. As we said before, the implementation of a constraint requires that we define a new class, in the proper place of the constraint hierarchy and define the `localUpadte` method for that constraint. `localUpadte` must update properly the constraint's variable's domains, depending of course on the constraint we are dealing with.

In the `EQ` case for instance, and for `FDIU` variables, if both domains must share the same values, we only need to compute their intersection. An update of the minimum and maximum values of the variables' domains is required because `FDIU` variables are (a specialization of) `FD` variables.

To implement the disjunctive constraint `plusOrMinus`, it's necessary to define a subclass of `gc.fdiu.VV.Constraint`. This subclass defines a new instance variable, `value`, used in `localUpadte` to create a singleton-valued variable. `localUpadte` just updates the corresponding domain variable in the following manner:

$domain(X) = domain(X) \cap [[domain(Y) - \{value\}] \cup [domain(Y) + \{value\}]$

```
public class plusOrMinus extends gc.fdiu.VV.Constraint{
    private int value;
    public plusOrMinus (Variable VX, Variable VY, int V) {
        super(VX, VY);
        value=V;
    }
    public void localUpdate (int n) {
        int m=1-n;    // env[n]=env[m]-i  V env[n]=env[m]+i
        Domain Dn=env[n].domain;
        Domain Dm=env[m].domain;
        Domain Di=new Domain(value);
        Domain Ddiff=Dm.diff(Di);
        Domain Dsum=Dm.sum(Di);
        env[n].domain=Dn.intercept(Ddiff.union(Dsum));
        env[n].updateMinMax();
```

```
    }
}
```

7.2 N Queens

- The problem:
  Lay down `N` queens on a $N \times N$ chess-board so that there is no couple of queens threatening each other.
- The variables:
  We have `N` variables that take values from 1 to `N`. For instance, to $N = 4$, the solution `[3,1,4,2]` means that, in row 1 the queen must be placed at column 3, in row 2 the queen must be placed at column 1, and so on.
- The constraints:
  To avoid the queens from threatening each other we will implement a constraint. If C_i and C_j are queens placed in columns i and j respectively, to avoid them to attack each other we must have

$$\forall j > i \begin{cases} C_j \neq C_i \\ C_j \neq C_{i+(j-i)} \\ C_j \neq C_{i-(j-i)} \end{cases}$$

  The constraint `NoAttack`, for 2 variables, implements this restrictions.

```
public class NoAttack extends gc.fdiu.VV.Constraint{
    private int c;
    public NoAttack(Variable VX, Variable VY, int V) {
        super(VX,VY);
        c=V;
    };
    public void localUpdate(int n) {
        int m=1-n;
        if (env[m].ground ()) {
            env[n].clear (env[m].min);
            env[n].clear (env[m].min + c);
            env[n].clear (env[m].min - c);
        }
        env[n].updateMinMax ();
    }
}
```

  Since this constraint is for two variables, a method to apply it at all variables in the list of queens, is required.

```
static void safe (Variable queen[], int n) {
    for (int i=0; i<n; ++i)
```

```
        for (int j=i+1; j<n; ++j)
            new NoAttack(queen[i], queen[j], j-i).tell();
}
```

7.3 Send More Money

- The problem:
  Find the digits corresponding to the letters, in order to validate the operation

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

- The variables:
  We have 8 variables, `S`, `E`, `N`, `D`, `O`, `R` and `Y`, they all take values from 0 to 9, except `M` which varies over 1 to 9. All these variables must be different. We need of course other variables, and their number depends on the approach used to achieve the solution.
- The constraints:
  There are two different approaches to building the solution of this problem:

(i) The solution can be found solving the equation:
$1000S + 100E + 10N + D + 1000M + 100O + 10R + E =$
$10000M + 1000O + 100N + 10E + Y$
To express this constraint, we can use the constraints `XtimesYeqZ` ($X \times Y = Z$), `XplusYeqZ` ($X + Y = Z$) and auxiliary variables. It requires a lot of variables, calls to constraints `XtimesYeqZ` and `XplusYeqZ` and a lot of time too. It possible, but very inefficient.

(ii) It's better to implement a constraint that makes the sum of three digits and generate the result digit and the carry. It's a constraint over five variables, and it's called `FullAdder`.The corresponding arguments are respectively, the three digits, the result and the carry. With this constraint it's sufficient the following system of constraints to solve the problem:

```
new FullAdder(theVar(0),D,E,Y,C1);
new FullAdder(C1,N,R,E,C2);
new FullAdder(C2,E,O,N,C3);
new FullAdder(C3,S,M,O,C4);
new Eq(C4,M);
```

7.4 Experimental Evaluation

In the following table are the comparative running times (in milliseconds) for these examples. The GC times are followed by the slowdown relative to

GNU-Prolog [3], shown in parentheses.

Program	GPROLOG	GC (FDIU)	GC (FDD)	GC(FD)
Five	51(1)	322(6.3)	473(9.3)	390(7.6)
Queens 4	50(1)	37(0.7)	33(0.7)	18(0.4)
Queens 8	53(1)	386(7.3)	316(6.0)	212(4.0)
Queens 16	762(1)	77957(102.3)	59492(78.1)	60949(80.0)
Send	50(1)	828(16.6)	854(17.1)	2297(45.9)
Average		(26.6)	(22.2)	(27.6)

In program Send the second approach was used. The times for program Queens, are taken when the *first solution* was found. Programs Five and Send use the `allDifferent` constraint for lists. These times correspond to an AMD K6/300 with Linux and standard JDK 1.1.7, i.e. no "just in time" compiler was used. An option for Java, that disables assynchronous garbage collection was used. Java run-time initialization times are significant and therefore they're being subtracted from the GC times, which explains the speedups obtained in the Queens 4 benchmark.

The resulting times are similar for the three domains except in program Queens. `NoAttack` is the only constraint that is implemented in the same way for domain representations `FDIU` and `FDIU`. We are interested in removing single values from the domain, so, no particular method to manipulate a `FDIU` domain is used. Notice that none of the examples match the requirements that motivated the implementation of `FDIU` variables, as all the variables of the problems have very small-range domains. Defining constraints over `FD` variables could only be done, in most cases, when the domains are ground. The `FD` representation is not fit for cases where the domains are very large.

8 Future Work

New variable domains classes are being implemented. We are currently working on real numbers and intervals over these, because they constitute an appropriate representation for 2D geometry.

We'll also consider implementing FD variables whose domains are obtained by querying a relational database, as this is essential to another work-in-progress, the University of Évora's integrated information system: `GC` will provide the engine that will allow the implementation, in Java, of a Constraint Logic Programming system.

We also plan on designing an application for modeling Information Systems using this CLP system. This language will have a visual syntax.

Along the way we shall work on improving GC's overall efficiency, trying to make use of whatever techniques may be appropriate; this includes resorting to other Java compilers and run-time systems such as `gcj`.

References

[1] Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, June 1996.

[2] Daniel Diaz. *Étude de la compilation des Langages Logiques de Programmation par Contraintes sur les Domaines Finis: le Système CLP(FD).* PhD thesis, Université d'Orléans, 1995.

[3] Daniel Diaz and Philippe Codognet. GNU Prolog: Beyond Compiling to C. In *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000).*

[4] Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19(20):503–581, 1994.

[5] Ken Kahn. ToonTalk: An Animated Programming Environment for Children. *Journal of Visual Languages and Computing*, 7(2):197–217, June 1996.

[6] Jean-Francois Puget and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Logic Programming, Proceedings of the 1995 International Symposium*, pages 513–527. MIT Press, 1995. ISBN 0-262-62099-5.

[7] David Canfield Smith, Allen Cypher, and Jim Spohrer. KIDSIM: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7):55–67, 1994.