Electronic Notes in Theoretical Computer Science 48 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume48.html pp. 1 – 12

Design for AJACS, yet another Java Constraint Programming framework [1]

Lígia Ferreira [2] Salvador Abreu [3]

Departmento de Informática
Universidade de Évora
Évora, Portugal

Abstract

This article introduces AJACS (Another Java Constraint Programming System), a toolkit for Concurrent Constraint programming implemented in the Java language. It comes as a successor to our previous work in implementing Constraint Programming idioms in Java, GC [5], in that it represents an attempt to deal with some of GC's inadequacies in terms of performance whilst providing a setting which is adequate to express problems in a way that can be easily solved in a parallel execution environment, as provided by a concurrent programming setting.

We claim that AJACS provides a very flexible toolkit for use in general applications that may benefit from constraint programming techniques. We also claim that AJACS allows for the coding of CSP problems whose solution is amenable to a practically effortless parallelization.

Key words: Constraint Logic Programming, Concurrent Constraints, Java.

1 Introduction

Combining a programming methodology with a particular existing language can be helpful because the result will inherit the host language's qualities. In the case of CLP [7], 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

[1] The authors would like to thank the anonymous referees for their constructive review of an earlier version of this article. Universidade de Évora, CENTRIA and Fundação da Ciência e Tecnologia (under contract PRAXIS P/EEI/10191/98 "OAR") are acknowledged for their support of the work described herein.

[2] Email:lsf@di.uevora.pt
[3] Email:spa@di.uevora.pt

useful (CLP) may be demonstrated and actually put to use in non-specialized environments.

The Java language is touted as having a high degree of machine-independence at all both the source and compiled object level, thereby providing an appealing platform for the development of applications. This aspect has been slightly hindered by the efficiency of the available Java implementations, but this situation is changing rapidly with the emergence of better compilers and run-time support systems.

The inefficiency of the implementations of JVM, the Java virtual machine, 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 [9]. 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 Constraint Logic Programming 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 more easily support our research in the direction of Visual Programming Languages.

In our previous work [5] we proposed a toolkit for Constraint programming in Java, much along the lines of the classical CLP implementations such as Ilog:Solver [9], CHIP [4] or CLP(FD) [2]. While being relatively easy to program with, GC was plagued with the heavy burden of reproducing the CLP search process in a Java setting, in particular:

- It trailed variables' values which led to a very inefficient execution as was demonstrated by the poor performance results, which were even further aggravated by the inefficiency of the Java compiler and run-time implementation.
- The data structures used in GC were not really amenable to parallelization, cutting short our attempts at making a concurrent and distributed implementation.
- GC implemented several representations for variables' domains but provided no automatic way to switch between these representations.

Aware of these shortcomings, we decided to start anew and work on a CC-

like [10] approach to provide a Java-based toolkit for concurrent constraint programming. Some of our main goals were to:

- Automate the migration of a value's representation in order to always choose the most appropriate one, without requiring the programmer to be aware of this issue.
- Provide a setting which would be more easily programmable in a concurrent and parallel execution setting.
- Avoid trailing entirely. This goal can be thought of as part of the previous one (enabling parallel execution), as the purpose of trailing was to overwrite variables' values with previous ones, which is an undesirable approach in a parallel setting.
- Provide a framework whereby different search strategies could easily be specified by the programmer, while providing reasonable implementations for the most common search procedures.

The architecture of AJACS revolves around a few key concepts which will be described in the following section.

The remainder of this article is structured as follows: section 2 gives a formal description of each of the main concepts in AJACS. In section 3 we describe the structure of the Java implementation. Section 4 discusses in more detail how we deal with the subject of exploring a search space. Finally, section 5 makes a comparative balance between AJACS and other approaches and in section 6 we conclude and discuss a few planned developments.

2 Concepts

Notational convention: single objects will be denoted by lower-case letters, sets of objects of the same type will be denoted by the corresponding capital letter.

2.1 Value

A *value* represents a subset of a variable's domain, ie. a set of elements which are designated *singular values*. A value is said to be *ground* if it contains exactly one singular value.

We shall use the letter u to designate a *value* while a collection of values will be denoted by U.

2.2 Variable

There is no explicit concept of variable in AJACS. These are abstractly thought of as the set of values located at the same index in a set of stores (see the next section for the definition of a store).

2.3 Store

A *store* is an indexed collection of *values*. The goal is that, in solving a CSP, several similar stores (w.r.t the number of values) will be created in which the set of values given by the same index across all stores represents a variable.

Since a store is obtained from another one by the application of a constraint propagation step (see below), a store must contain a reference to its ancestor. In the special case of the initial store, the ancestor is undefined. Each line [4] of the store is related with the values of a specific variable.

Moreover, and since a store is obtained from an ancestor store by restricting a variable's value, each store must mention *which of its lines* has been restricted in order to obtain the successor stores.

The definition of a store s is:

$$s \equiv (U, i, s_p)$$

Where:

- $U = (u_1, u_2, \ldots, u_n)$, in which u_i is the value (in the present store) associated with store line i.
- i is the index into s for the variable whose domain has been restricted in s's successor stores (i.e. all stores $s' = (U', i', s'_p)/s'_p = s$.
- s_p is s's ancestor store.

2.4 Constraint

Constraints are relations over the variables which occur in a problem. The AJACS concept of *constraint* expects these to be the mechanism responsible for propagating changes made to one variable onto the remaining variables occurring in the store. A constraint c can be defined as the pair:

$$c \equiv (f, (i_1, i_2 \cdots i_n))$$

Where $f = \lambda x_1 x_2 \ldots x_n \cdot e$ is a boolean-valued function ranging over the set of variables in the store. This function is expected to map $(i_1, i_2 \cdots i_n)$ to *true* whenever the constraint holds for the values indicated by the store lines $i_1, i_2 \cdots i_n$ and to *false* whenever the resulting store would be inconsistent.

The tuple $(i_1, i_2 \cdots i_n)$ is called the constraint's *environment*.

2.5 Problem

A CSP is modeled by the *problem* concept, which is defined by a set of variables with an associated initial domain (i.e. a store) together with a set of constraints over these variables.

The purpose for which a problem is formulated is to obtain *solutions* to it, i.e. sets of ground values for the problem's variables that are consistent with the set of constraints.

[4] We shall use the expression "store line" as a synonym for "index into the store".

In AJACS, solutions are obtained from a store, so stores belong to our formulation of the problem. A problem p is defined as:

$$p \equiv (s_{init}, C, C_v)$$

Where:

- $s_{init} = (U, -, -)$ is the *initial store*.
- $C = \{c_1, c_2 \cdots c_k\}$ is the set of constraints which we want to use to solve the CSP. These refer to store lines within s_{init}.
- $C_v = \{(i, C'_i) \ / \ \forall i \leq \#U,\ C'_i = \{(c_j, n) \ / \ c_j = (f, X) \in C \wedge v = X_n\}\}$. Informally, C_v is the set of pairs made up of a store line (i.e. that which designates a problem variable) and a set of constraints where it occurs. The latter set is in fact made up of pairs in which we have a specific constraint and the index into the constraint's arguments which is the occurrence of the first variable.

  The purpose of this component is to indicate the set of constraints in which a variable occurs.

The initial store for the problem is obtained by the initial values of the variables. Note that all stores in a problem have the same structure.

2.6 Search and Search Strategy

The concept of *search* embodies the procedure which, given a *problem*, finds *solutions* (i.e. all-ground *stores*) for that problem.

A *search procedure* may be thought of as the repetitive application of a *search step*, until a solution is found or the search space is exhausted. In keeping with our objective of trying to be as flexible as possible, search procedures may be either sequential or parallel, independently of the nature of the search step.

A "search step" can be defined as the concrete action stipulated by a *search strategy* or *strategy* for short. A strategy applies to a state of the computation, i.e. a *store*, and specifies its successor state in the search process. Finding the successor store entails deciding:

- Which of the remaining non-ground [5] variables is to be selected, and:
- For the selected variable, how is its domain going to be reduced. Usually this will mean deciding what singular value it will take. This value must always be a subset of the variable's original domain.

In order to achieve these goals, a strategy e can be defined as the pair:

$$e \equiv (f_v, f_u)$$

Where f_v is the function that selects the variable from s which is to be modified and f_u is the function that selects the particular value that the aforementioned

[5] Ground variables already have singleton domains and are therefore not susceptible of having that domain subdivided.

variable will take in the new store.

These functions are specific to each strategy and must be typed in the following manner:

$$f_v: \qquad s \mapsto i$$

$$f_u: (s, i, j) \mapsto u$$

Where i is (the index of) a variable in the store s, v is a singular value and j is a pass count whose interpretation depends on the particular instances of the strategy (ie. f_v and f_u).

For example, a search strategy $e_0 = (f_v, f_u)$, for use in depth-first and breadth-first searches, which selects any non-ground variable in the store and iterates through its singular values could be given by:

$$f_v(s) = i \ / \ U = \text{vars}(s) \wedge \#U_i > 1$$

$$f_u(s, i, j) = u \ / \ u = \{x\} \wedge x = \text{nth}(U_i, j)$$

Where vars is the variables-extraction function and $\text{nth}(u, j)$ is the j^{th} single value from the set specified by the domain u. For instance, $u = \{4, 5, 6, 7\}, \text{nth}(u, 3) = 6$.

For example, an algorithm for a sequential search procedure which returns the first solution is given by algorithm $sequential\_search\_first(s, C)$ where s is a store and C is a set of constraints. This algorithm is illustrated in figure 1 on page 6. The loop applies to A parallel version of this search procedure can

```
funct sequential_search_first(s, C) ≡
  if < s is ground >
    then return s
    else let U = vars(s);
         i = f_v(s);
         foreach j ∈ U_i do
           s' = f_u(s, i, j);
           propagate(s', C);
           if < s' is consistent >
             then return sequential_search_first(s', C)
         end.
```

Fig. 1. Algorithm for a sequential search procedure

easily be obtained by replacing the **foreach** loop construct by a parallelizing cycle in which the results may be collected.

The end result of applying a search procedure to a problem is a set of *solutions* to the problem. A solution is simply a store with the same structure as the original one, except that it only has ground variables.

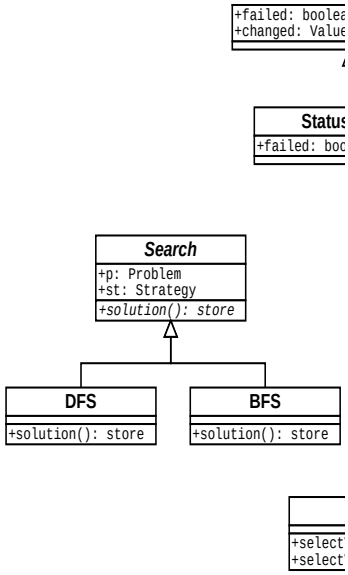

Fig. 2. Class System for AJACS

3 Java Classes

3.1 Class `Value`

Values represent the set of integers that is possible for a variable to assume. They have three internal representations, a compact one, and two non-compact. Class *Value* implements compact intervals, class *FddValue* implements non-compact intervals as bit sets and class *FdiuValue* implements non-compact intervals as a disjoint union of compact intervals.

The migration between the different representations is done automatically by the system. Constraints affect the values by doing some operations over them. Removing an integer from a `value`, intercept `values`, unify them, sum, etc., are some of the possibilities of the actions performed by a constraint to generate new values. The appropriate representation of a value is then decided by the system, which is to say that the result of an operation under one or more objects of one of the *Value* classes could return an object of another *Value* class. A "proper" *Value* could always be represented by an *FddValue* or by an *FdiuValue*. The reverse is not always possible.

The Appropriate Representation of a Value
Whenever it is possible compact representations are always preferred. Figure 3 shows all the possible migrations over the different representations. The operation represented by the arrow is removing an integer from the value. Transitions are performed in the following manner: a *Value* remains in it's

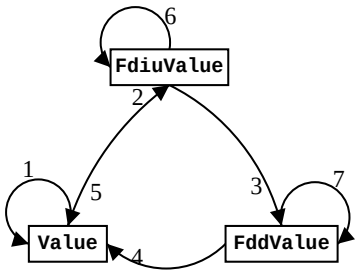

Fig. 3. Class Transitions in *Value*

class if the object removed is one of the extremes of the interval (1), otherwise it will be transformed in a *FdiuValue* (2). An *FdiuValue* will be transformed into a *Value* if the interval union results in only one interval (5) and will be transformed into an *FddValue* if all the intervals of the union have only one element (3). In all the other cases, the *FdiuValue* stays an *FdiuValue* (6). An *FddValue* will be transformed into a *Value* if all the elements of the set are contiguous (4), in all the other cases it will remain an *FddValue* (7).

We mention removing integers from a value, because this is the basic operation performed by the constraints. Other operations follow the same scheme of transitions, for instance, if an operation over an *FdiuValue* changed all the intervals into singleton intervals, then the returned *Value* is an *FddValue*.

3.2 Class `Store`

The Java implementation of class *Store*, follows it's formal description. A constructor is used to create a store, given an array of values. The *setVar* method permits to select the *var* instance variable that will be modified by *Search*. *setValue* is used to impose a new value for a variable. *setAncestor* makes the link between two stores.

3.3 Class `Constraint`

The *Constraint* class has an instance variable *env*, that stores the environment of the constraints. This environment is filled with the constructor method of the class. Method *update(s,i)*, will try to update all the variables $env[k], for\ k \neq i$. The returned object is of type *Status*. It can then be a

fail or a Value. In that case, the *Value* has the index of the variables that changed. See figure 2.

3.4 Class `Problem`

The `Problem` class is implemented with three instance variables, an initial store *initStore*, a list of constraints *C*, and a list *Cv*, that is constitute by lists of *ConstVar*. The constructor *new*, is used to create a new problem, witch has an initial store defined by the *Values*, and two empty lists, *C* and *Cv*. Method *add* is used to add constraints to the problem, updating both lists. The *update* method will be used to propagate the effects of affect a new value to the i^{th} variable of the store s. The propagation is done using all pairs (c, n) of C_{v_i}, and calling *c.update(s,n)*.

3.5 Classes `Search` *and* `Strategy`

The *Search* class is abstract: its method `solution` is also abstract. All subclasses of *Search*, represented in figure by the examples *DFS* and *BFS*, must therefore redefine this method. The `solution` method has the responsibility of generating the sequence of stores that leads to a solution or a failure. For that purpose, it uses the strategy specified by `st`.

Class `Strategy` is also abstract. All implemented strategies are subclasses of this one. In figure 2 are two examples of strategies, `StFirst` and `StDivide`. `StFirst` is the strategy defined in 2.6 and `StDivide` is a strategy that split a Value in two equal subparts.

3.6 An Example

We will give an example of how to work with this classes system, with the classical N-Queens. Consider N=4, for simplicity.

We have 4 `Values`: `u0`, `u1`, `u2` and `u3`, defined by:

```
u0 = u1 = u2 = u3 = new Value(1,4).
```

We must implement a constraint, lets call it *NoAttack*. NoAttack is a subclass of *Constraint*, and assures that two queens do not threatening each other. The problem, and its initial store is defined by:

```
p = new Problem([u0,u1,u2,u3]).
```

In order to update the the lists *C* and *Cv*, we add the constraints to the problem.

```
for (i=0; i<=2; ++i)
   for (j=i+1; j<=3; ++j)
        p.add (new NoAttack (i,j,j-i))
```

Let `C1` be the constraint `NoAttack(0,1)`, `C2` be the constraint `NoAttack(0,2)`, etc. `C1.env=[0,1]`, `C2.env=[0,2]`, etc. Adding the constraints to the prob-

lem turns List C into {C1, C2, C3, C4, C5, C6} and List Cv into Cv = {Cv0={(C1,0), (C2,0), (C3,0)}, Cv1={(C1,1), (C4,0), (C5,0)}, Cv2={(C2,1), (C4,1), (C6,0)}, Cv3={(C3,1), (C5,1), (C6,1)}}.

Now we could apply a search, lets define it: `s = new Search(p, st=new StFirst())`. The solution is given by `s.solution()`. Figure 4 shows the sequence of stores generated by `solution`.

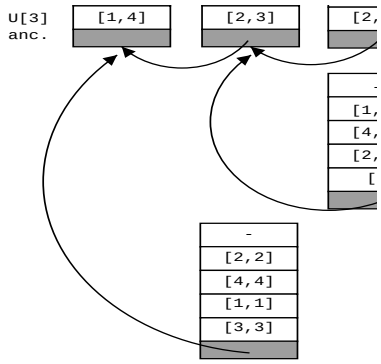

Fig. 4. DFS search applied to 4-Queens problem

4 Search

Creating a constraint solver that would be versatile from the search strategy point of view was the one of the goals in the design of AJACS. This versatility is achieved in three ways:

- At any point of search, we could see the state of variables. Its all in the store.
- There is a total freedom to define any kind of search technique, by implementing a *Search* subclass. This is extensible to *strategies*. The combination of an appropriate *strategy* and the *next* store generation allows us to implement a variety of search procedures. Depth-first, Breadth-first, Best-first are some examples. The most appropriate search strategy depends of the problem. It is possible that what is a solution under some search strategy is not ground.

5 Comparison with other Approaches

The AJACS system appears as a continuation to our previous work on GC [5], and positions itself largely in the same manner. It is currently being implemented so we cannot yet assess its performance. However, the design represents a significant departure from GC in that:

- We get rid of trailing, which is always an expensive operation when dealing with constraint variables.
- The organization of the system allows for a simple parallel execution scheme.

Many of these ideas can be found in one way or another in existing systems (for instance the recent implementations of the Oz language [11]), but none do so in a setting that may easily be exploited by an unaware user: this comes as a direct result of our option to implement AJACS as a Java package.

There are other efforts that use Java to implement Logic Programming languages, such as jProlog or Banbara and Tamura's work on generating Java from a Linear Logic programming language [1]. None of these are directly comparable to our approach of creating a constraint programming toolkit for Java. Other similar toolkits do exist, but for C++: such is the case of Ilog:Solver [9] or Figaro [6].

6 Conclusions

Because AJACS is currently being implemented, we have no relevant performance data with which we could quantitatively compare our work to reference Constraint Logic Programming or Concurrent Constraint systems.

The platform on which the implementation is being developed and tested is a Linux cluster made up of 8 dual-processor nodes connected by a System-area network. We expect this hardware architecture to be useful in assessing the usefulness of the present model, in its ability to exploit parallelism.

References

[1] Banbara, M. and N. Tamura, *Translating a linear logic programming lanaguage into java*, Elsevier Electronic Notes in Theoretical Computer Science **30** (2000).

[2] Codognet, P. and D. Diaz, *Compiling Constraints in clp(FD)*, Journal of Logic Programming **27** (1996), pp. 185–226.

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

[4] Dincbas, M., P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, *The Constraint Logic Programming Language CHIP*, in: *International Conference on Fifth Generation Computer Systems 1988* (1988), pp. 693–702.

[5] Ferreira, L. and S. P. Abreu, *A constaint logic programming framework in java*, Elsevier Electronic Notes in Theoretical Computer Science **30** (2000).
URL `http://www.elsevier.nl/gej-ng/31/29/23/55/27/show/Products/notes/index.htt`

[6] Henz, M., T. Mueller and N. K. Boon, *Figaro: Yet Another Constaint Programming Library*, Elsevier Electronic Notes in Theoretical Computer Science **30** (2000).

[7] Jaffar, J. and J.-L. Lassez, *Constraint logic programming*, in: *Proceedings Fourtheenth Annual ACM Symposium on Principles of Programming Languages* (1987), pp. 111–119.

[8] Jaffar, J. and M. J. Maher, *Constraint Logic Programming: A Survey*, Journal of Logic Programming **19** (1994), pp. 503–581.

[9] Puget, J.-F. and M. Leconte, *Beyond the Glass Box: Constraints as Objects*, in: *Logic Programming, Proceedings of the 1995 International Symposium* (1995), pp. 513–527.

[10] Saraswat, V. A., "Concurrent Constraint Programming Languages," Ph.D. thesis, Carnegie-Mellon University (1989), available as Technical Report CMU-CS-89-108.

[11] Smolka, G., *The Oz programming model*, in: J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science **1000**, Springer-Verlag, Berlin, 1995 pp. 324–343.