A C++ implementation of CLP

Jean-Francois Puget

ILOG SA,
2 avenue Gallieni, BP 85
94253 Gentilly Cedex, FRANCE
puget@ilog.fr

Abstract

We have implemented a C++ library, called ILOG SOLVER, that embodies Constraint Logic Programming (CLP) concepts such as logical variables, incremental constraint satisfaction and backtracking. This library combines Object Oriented Programming (OOP) with CLP. This has two advantages. First of all, everything is an object in SOLVER: variables, constraints and search algorithms (goals). Thus, SOLVER is easily extendable by defining new classes. Second, objects can be used for modeling the real problem that has to be solved, which is a great software engineering advantage. In particular, SOLVER provides for the definition of *class constraints*, that are inherited by all the objects of that class.

1 Introduction

Recent years have seen a growing interest in Constraint Logic Programming for finite domains, as witnessed by the implementation of several languages, such as CHIP [VH 89], CLP(fd) [DC 93] and CLP(BNR) [OV 93]. These languages are obtained by integrating a constraint solver over finite domains into Prolog, by extending unification. Prolog is a good candidate for implementing such a language since it already offers support for logical variables and efficient backtracking search. Prolog however has some limitations both in terms of integration with other software written in C for instance, and in term of modularity. Moreover, CLP languages are difficult to extend. If a new kind of constraint is needed, it is usually impossible to encode it efficiently in Prolog. In other words, the built-in constraints in a CLP language are not written in Prolog, but in a lower level language such as C or assembly language. This limitation is so severe that it begins to be recognised in the Prolog community [CCD 94].

Our motivation for using an OOP language is that the definition of new classes is a simple yet powerful mean for extending a software. Thus we asked ourselves if CLP techniques could be implemented in an OOP language. Following this remark, we implemented a Lisp-based Constraint Logic Programming package called PECOS[Pug 93]. Since PECOS was implemented with object classes, it seemed possible to implement a similar library in C++, leading to the development of ILOGSOLVER(SOLVER in the rest of this paper). This paper presents how the CLP concepts can be implemented using object classes, and how the use of objects enables a better extensibility, and modeling. Next section presents CLP concepts. Then section 3 describes SOLVER. Section 4 describes the use of an object oriented model. Then in section 6, we relate SOLVER with other work on constraint programming languages.

2 Constraint Logic Programming concepts

We briefly describe here the execution model we chose for ILOG SOLVER. This model is quite simple, and is similar to other CLP languages. A *store* contains all the logical variables and the constraints. When the value of a logical variable is not known, a *domain* is associated to it. This domain represents, either explicitly or implicitly, the values that the variable can take. An instruction in one of the following: creating a new logical variable; adding a constraint to the store.

In the latter case, the constraint system is incrementally simplified with a constraint propagation algorithm the removes inconsistent values form the domains of the variables. This is called *constraint propagation*, and

can lead to one of several possible outcomes: the system is solved (each logical variable is assigned a value); A contradiction is detected; None of these.

Constraint propagation is *incomplete*, i.e, it cannot detect all inconsistencies. This is the price to pay in order to have a polynomial runtime efficiency. To cope with incompleteness, CLP relies on nondeterministic search. The basic way to search for the solution of a constrained problem is to repeatedly do the following: set a choice point, add a constraint to the problem (usually of the form $x = a$, where x is a variable, and a a constant, i.e. assign a value a to a variable x), and apply the constraint propagation algorithm. If this leads to a contradiction, backtrack, i.e. undo the setting of $x = a$, and all its consequences, and try with another constraint.

3 Implementation in SOLVER

Prolog offers some constructs not present in C++: logical variables, memory mangement as a stack, delay mechanism used to encode constraint propagation, and non determinism. This section describes how these constructs can be encoded in C++.

3.1 Logical variables

In SOLVER, a logical variable is a C++ object. This object has several hidden slots. These slots include the value of the variable when it is known, the list of the constraints posted on that variable and a representation of the domain when the value is unknown. The static typing of C++ is used as follows: each type of variable is implemented by a class. For instance, the class for integer variables (`CtIntVar`) is different from the class for floating point variables `CtFloatVar`). SOLVER provides several classes of variables: integer variables, whose domain is either an interval, or an enumeration of integer; floating points, whose domain is an interval; boolean, whose domain is the set $\{0, 1\}$; sets.

The treatment of integer variables is similar to what is described in[VH 89]. The treatment of floating point variables is done by interval propagation, and owes much to [Lho 93]. It is also related to CLP(BNR)[OV 93]. The treatment of booleans is also done by propagation, as in CLP(BNR). The last class of variables is specific to SOLVER[Pug 93]. The value of a finite set variable is a finite set. When the variable is not bound, the domain is represented by its two bounds: the greatest lower bound (i.e. the intersection) of the possible values of the variable, and the least upper bound (i.e. the union) of the possible values of the variable. It is also possible to constrain the cardinality of such a variable. Different types of variables can be used in the same application as the propagation algorithm we use treats in a uniform way all the variables. This algorithm fits into the framework of AC-5[DeVH 91] although AC-5 is presented on integers only.

3.2 Memory management

Variables are objects, and can thus be allocated and created as usual in C++: on the C++ stack, or on the heap, with the `new` operator. Once allocated, an object is initialized by a `constructor`. The management of objects allocated in the heap is let to the user, who is responsible for explicitely deallocating the memory used when an object is no longer useful, with the `delete` operator. However, the most common use of SOLVER uses a non deterministic search which amounts to a depth first search. An allocator is provided by SOLVER. This allocator automatically reclaims the memory used by an object created in a branch of the search tree when this branch is abandoned. For instance, the next line creates an integer variable using SOLVER memory management:

```
CtIntVar* z = new (CtHeap()) CtIntVar(-10, 10);
```

3.3 Constraints

A constraint is a C++ object. Stating a constraint amount to the creation of a constraint object and to add this object in the constraints lists of the appropriate variables. This is encapsulated by a simple function call for built-in constraints. For instance, the following line expresses an addition constraint:

```
CtTell(x == (y + z));
```

Basic constraints include the following: $=$, $\leq$, $\geq$, $<$, $>$, $+$, $-$, $*$, $/$, subset, superset, union, intersection, member, boolean or, boolean and, boolean not, boolean xor. Some generic versions of these constraints are available, such as `CtAllNeq` which states that all the variables in a given array are different, or `CtArraySum` that returns a variable equals to the sum of the variables in a given array. Such generic constraints are useful since only one constraint is allocated, whatever the number of variable is. Some more elaborate constraints are also available, including `CtElement`[VH 89], a counting constraint[Pug 93], and piecewise linear constraints.

SOLVERoffers a comprehensive range of constraints, as shown above. However, from our experience on applications of constraints to real world problems, it seems very useful to be able to introduce domain specific constraints. This was not possible in traditional CLP languages such as CHIP, but some recent work study how this could be efficiently done[CCD 94]. We had implemented a `defctconstraint` primitive in PECOS, using Lisp macros. This approach is not feasible in C++ since macros are too simple in this language. We have chosen to define new constraints by composing predefined constraints with logical operators *or*, *and* *xor* and *not*. These operators are implemented using C++ boolean operators: `||`, `&&`, `!=`, `!`. SOLVER then offers two primitives to assert such constraints: `CtTell` that asserts a given boolean formula, and `CtIfThen` that asserts a logical implication between two boolean formulas.

For instance the following line states that at least one of the variables `x` and `y` must be equal to `0`:

```
CtTell ( ( x == 0)  || ( y  == 0));
```

When the composition is not sufficient, it is possible to define a new class of constraint, whose methods describe how the constraint has to be propagated.

3.4 Search

The last CLP concept is the ability to design algorithms as choices between different possibilities, without knowing in advance which one is the right one. This is very different from standard C++ programming, and is close to Prolog. However, Prolog compilation techniques are not directly applicable here since we want to stick to standard C++.

We chose to use objects here too. A non deterministic algorithm is define with objects called *goals*. A goal is an object with a particular method that defines how to execute it. A goal *fails* if an inconsistency is detected during its execution. The `CtAnd` function creates a goal which is a sequence of other goals. The `CtOr` function creates a goal which is a choice between several other goals. The execution of such a goal creates a choice point, then starts the execution of the goals until one succeeds. In order to simplify the syntax, a goal class can be defined by a macro called `CTGOALn`, with a name. and a list of parameters as arguments, `n` being the number of parameters. This macro is followed by the body of the method that executes the goal.

For instance the following goal selects a value for a variable. It first selects a value, then sets a choice between, either assigning the value to the variable, or removing the value from the domain of the variable, and calling itself recursively in order to try the remaining possible values.

```
CTGOAL1(CtInstantiate, CtIntVar*, x){
    CtInt a = x->chooseValue();
    CtOr(Constraint(x == a),
         CtAnd(Constraint(x != a),
               CtInstantiate(x)));
}
```

An equivalent Prolog code could be written as follows. Choice points are implicit in Prolog, and are created by defining two or more clauses with the same name.

```
Instantiate(x):- ChooseValue(x,a), InstantiateAux(x,a).

InstantiateAux(x, a):- Constraint(x = a).
InstantiateAux(x, a):- Constraint(x != a), Instantiate(x).
```

This shows one of the interest of SOLVER: one does not have to worry about nondeterminism when it is not needed. For instance, the `chooseValue` function is implemented directly as a C++ function.

SOLVERoffers two more control primitives `CtSolve` and `CtMinimize`. The first one nondeterministically searches for a solution of a goal, the second one searches for a solution that minimizes the value of a

given logical variable. The algorithm used is a depth-first branch and bound, similar to the one described in[VH 89]. It is also possible to label a choice point, and to backtrack to a labeled choice point. This enables to implement dependency directed backtracking search. [LPMD 94] describes an application that uses this feature. Last, backtracking undoes the effects of constraint propagation. To achieve this, a trailing mechanism is used: changes to variables and the constraint store are stored, so that that can be undone. It is also possible to save the state of *any* C++ object so that it is restored when backtracking.

4 Objects and constraints

Objects are used for the implementation of SOLVER since they allows a simple yet powerful extensibility. Objects can also be used when building a SOLVER application. Indeed, we can: define new variable classes, define object slots with constraints, define class constraints, and define class dependent search algorithms, as shown by the next example.

4.1 A configuration example

The general formulation is this: given a supply of components and bins of given types, determine all assignments of components to bins satisfying specified assignment constraints subject to an optimization criterion.

In the following example there are 5 types of components: glass, plastic, steel, wood, copper. There are three types of bins: red, blue, green. Their capacity constraints are: red has capacity 3, blue has capacity 1, green has capacity 4. The containment constraints are: red can contain glass, wood, copper, blue can contain glass, steel, copper, green can contain plastic, wood, copper. Requirement constraints are (for all bin types): wood requires plastic. Certain component types cannot coexist: glass excludes copper, copper excludes plastic. Certain bin types have capacity constraint for certain components: red contains at most 1 of wood, green contains at most 2 of wood. Given an initial supply of: 1 of glass, 2 of plastic, 1 of steel, 3 of wood, 2 of copper, what is the minimum total number of bins required to contain the components?

Let us first define the data of the problem. We use C++ enumerations for associating meaningful names to integers:

```
enum Component{glass,plastic,steel,wood,copper};
enum Type  {red, blue, green};
```

4.2 Classes of variables

Logical variables are objects. Thus, it is possible to define new classes that have special properties. For instance, we define the class `Number` that represent constrained integer variables between 0 and 4:

```
class Number : public CtIntVar{
  public:  Number() : CtIntVar(0, 4){}
};
```

4.3 Constrained slots

A class `bin` seems a good design choice. Its slots are: the type (red, blue or green), the capacity, and the number of each component. In a classical object oriented language, the slots have known values. here,we *have to* find these values while respecting some constraints. SOLVER provides a very simple way for doing this: a logical variable is used. The value of the slot is then the value of the variable when the variable is known.

Then we define the class `Bin` with logical integer variables: Nearly all constraints are posted in the constructor of the class

```
class Bin {
  public:
    CtIntVar type;
    Number capacity;
    Number n[NComponents];
```

```
    Bin ();
    void labeling();
};
```

4.4 class constraints

One of the interset of OOP is that informations shared by a set of object can be expressed in a single place: the class of these objects. With SOLVER it is possible to express constraints in the class. These constraints will then be satisfied by all the instances of that class. It is obvious that such class constraints cannot be expressed in Prolog.

A class constraint is easy to state in SOLVER: one has to state it in the class constructor:

```
Bin::Bin () : type(red, green){
  CtElement(type,capacity,nTypes,Capacity);
  CtTell(capacity >= n[glass]+n[plastic] + n[steel] + n[wood] + n[copper]);
  CtIfThen( type==red,     n[plastic]==0 && n[steel]==0);
  CtIfThen (type==blue,    n[plastic]==0 && n[wood]==0);
  CtIfThen (type == green, n[glass]==0 && n[steel]==0);
  CtIfThen (n[wood] > 0,   n[plastic] > 0);
  CtIfThen (type==red,     n[wood] <= 1);
  CtIfThen (type==green,   n[wood] <= 2);
  CtTell   (n[glass]==0  || n[copper]==0);
  CtTell   (n[copper]==0 || n[plastic]==0);
}
```

The `CtElement` constraint sets the functional dependency between the type and the capacity of the bin. The next constraints states that the capacity must be greater than the sum of the component numbers. Then the various exclusions are expressed with composite constraints.

4.5 Search algorithms

A last advantage of OOP is to define search algorithms that depends upon the object classes. One has merely to express how to assign a value to each logical variables of an object:

```
void Bin::labeling(){
    for (int i = 0 ; i < NComponents; i++) CtInstantiate(n[i]);
    CtInstantiate(type);
}
```

The rest of this example (20 lines) include the demand constraints and the minimization procedure, anmd is given in [Pug 94].

5 Comparison with Other Systems

The theoretical principles behind SOLVERare the same as other CLP systems such as CHIP[VH 89] or CLP(fd)[DC 93], except the treatment of sets. The interest of SOLVER lies in the combination of OOP with CLP. This has 3 main advantages: extensibility (new constraint types), class constraints and specific search algorithms. These 3 points have proved to be very important when building real world applications[1].

There are many other hybrid systems that mix objects, logic and constraints. The closest one is LAURE[CF 94] which deeply integrates constraints, objects and rules in a single architecture. THINKLAB [Bor 79] and its followers are languages designed to perform reactive resolution of simple constraints through propagation, thus they cannot be compared with backtracking CSP languages. The results given in [Pug 94] on classical benchmarks show that SOLVER is faster that state of the art CLP languages such as cc(fd) and CLP(fd).

[1] more than 400 SOLVER licenses are used in about 150 industrial projects

6 Conclusion

The aim of this paper was to show that it is possible to integrate CLP concepts in C++ without loosing the important properties of CLP, such as declarativity and efficiency. For a large part, the techniques and algorithms used in ILOG SOLVERare not unique, except the treatment of finite sets . The integration of all these techniques in a uniform C++ environment is unique. This integration seems very promising regarding the delivery of CLP applications in an industrial environment.

Moreover, the use of objects has 3 advantages: extensibility (new constraint types), class constraints and specific search algorithms.

One may wonder however, whether CLP is the good level of tools to be offered for constraint based applications, as opposed to more specialized constraint solvers: it seems that for some classes of problems, it is a good idea to provide a set of specialized constraints. For instance, in job shop scheduling, some constraints that arise naturaly, such as finite capacity constraints, are quite difficult to encode with general constraints such as the one described here. We think that the answer is that specialized constraint are needed. For scheduling, we have developed a library of specialized constraints called ILOG SCHEDULE. Due to lack of space, it cannot be described here, and we refer to [Lep 94] for more details. However, it is impossible to predefine all the specialized constraints needed by all the potential application domains. This is precisely why the mechanism used to implement the primitive constraints is available to the SOLVER programmer.

Acknowledgements The following people have participated in the implementation of SOLVER: Abdelhakim Erhili, Younes Alaoui and Claude Lepape. Without them this work would not have converged. Patrick Albert had stimulating remarks on the design of SOLVER. Bruno Levy has been an unlimited source of enthusiasm for the past two years. This work has benefited from discussions with Yves Caseau.

References

[Bor 79] A. H. Borning, "THINGLAB A Constraint Oriented Simulation Laboratory", *PhD thesis*, Stanford University, July 1979.

[CCD 94] B. Carlson, M. Carlsonn, D. Diaz, "Entailment of Finite Domain Constraints" *Proceedings of the 11th International Conference on Logic Programming*, 1994.

[DC 93] D. Diaz, P. Codognet, "A minimal extension of the WAM for clp(FD)", *Proceedings of the 10th International Conference on Logic Programming*, 1993.

[CF 94] Y. Caseau. F. Laburthe, "Improved CLP Scheduling with Task Intervals", *to appear in the proceedings of ICLP 94*

[DeVH 91] Y. Deville and P. Van Hentenryck "An Efficient Arc Consistency Algorithm for a Class of CSP Problems", *Proc. IJCAI 91*, pp 325-330.

[Lep 94] C. Le Pape. *Implementation of Resource Constraints in* ILOG SCHEDULE*: A Library for the Development of Constraint-Based Scheduling Systems.* Intelligent Systems Engineering, 3(2), 1994.

[LPMD 94] C. Lepape, J.F. Puget, C. Moreau and P. Darneaud. "PMFP : The Use of Constraint-Based Programming for Predictive Personnel Management", Proc ECAI'94, Amsterdam, August 94.

[Lho 93] O. Lhomme "Consistency Techniques for numeric CSPs" *Proc. IJCAI93*, Chambery, France, pp 232-238.

[OV 93] W.J. Older and A. Vellino. "Constraint Arithmetic on Real Intervals", *Constraint Logic Programming: Selected Research*, MIT Press, 1993 .

[Pug 93] J.-F. Puget "PECOS A High Level Constraint programming Language", *Proc Spicis 92*, Singapore, September 1992.

[Pug 94] J.-F. Puget "A C++ Implantation of CLP", *Ilog Solver Collected papers*, Ilog tech report, 1994.

[VH 89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*, MIT press, 1989.

[VSD 92] P. Van Hentenrryck, V. Saraswat, and Y. Deville, "Design, Implementation and Evaluation of cc(fd)" Technical report, Brown university, 1992.