POOC - a Platform for Object-Oriented Constraint Programming

- work-in-progress paper -

Hans Schlenker and Georg Ringwelski

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
{hans.schlenker,georg.ringwelski}@first.fraunhofer.de

Abstract. In this paper, we describe an implementation-independent object-oriented interface for commercial and academic Constraint Solvers. This serves as a basis for evaluating different Constraint Solvers and for developing solver-independent applications. We show, how applications can use the interface, which solvers are already integrated into the framework and how additional solvers can be woven in. Furtermore, we provide to the community the described system as real Java packages via Internet, that even includes a basic but powerful Constraint Solver.

1 Motivation

Constraint Programming (CP, [7, 6, 5]) has emerged an adequate means to implement and solve many problems known from Artificial Intelligence. Constraint Satisfaction Problems (CSP) can be solved by stating the problem itself instead of defining an algorithm that is to find a solution. The constraints are posted in a domain specific solver that infers a solution of the CSP from the constraints and the possible values of constraint variables. Thanks to this declarativity, constraint based methods are used in many academic applications and in some professional software systems. Another reason for the success of CP is its efficiency. Many combinatorial NP-hard problems can be solved with constraint based methods.

From the application point of view, CP always needs a Constraint Solver (CS). Thus, one is usually faced with the problem to find the solver that best fits some specific needs. In a recent project, we compared the available solvers, performance and functional capabilities. We also wanted to be somehow independent from an actual implementation or product. This could allow us to develop a system, using some constraint solver, while being able to exchange the solver by another one in case we need new features or the former product becomes unavailable for some reason. Recapitulating on the above our requirements were:

- comparison and classification of existing solvers
- keep benchmark programs and maybe even applications solver-independent

This was our motivation to create POOC: a Platform for Object-Oriented Constraint Programming.

2 Approach

Our approach to supply our needs is to create a solver-independent object-oriented platform for generic constraint programming. This platform is designed to be a wrapper around existing constraint solvers. We use Java as implementation language: POOC is a Java package that provides wrappers, it is not a solver by itself. Figure 1 shows exemplary how to use it in the well-known Send-More-Money example.

The wrapper provides some abstraction from concrete implementation details: similar constraints in different solvers (like the `disjoint` constraint in SICStus Prolog [9] and `diffn` in CHIP [2]) are called through one interface and similar structures are mapped into one class tree. For example, arithmetic expressions are built recursively from arithmetic operations, variables and constants. The wrapper is designed as Object Factory[1]: new objects like variables or expressions are created by the solver at runtime, instead of explicitely being linked against some solver at compile time. We achieved this by defining Interfaces that every concrete solver (or its wrapper, resp.) must implement. In Java, an *Interface* is a special kind of an abstract class and therefore especially well suited for defining abstract (here: implementation-independent) entities.

Figure 2 shows the interfaces, their methods and how the interfaces are related to each other. We use the popular UML [1] notation of a class diagram, where every class is described by a rectangle that contains its name, its attributes and its methods. The classes are connected by specific arrows, described below.

Each `Solver` provides methods for the creation of concrete variables, domains, expressions (represented by the interface `X`) and labelings. This allows an application to be compiled independently from the concrete solver which is linked at runtime. Additionally, each `Solver` implements the basic FD[2] constraints, like `alldifferent` [10] and arithmetic in-/equalities, and modification of a variable's domain (method `in`). A `Domain` object describes such a domain which is given at least by its minimum and maximum (if a `Domain` is defined without specifying min and max, the globally defined values `INFIMUM` and `SUPREMUM` are used instead, resp.). A `Var` wraps a constraint variable whose current domain and, in case it is instantiated, its value can be queried. Each `X` stands for an arithmetic expression. Expressions can be recursively built by variables and constants and expressions joined by operators. A `Labeling` object is used for assigning values to the allocated variables: new solutions can be generated, an actual assignment can be commited (accepted) or undone. We use a labeling *object* and not — what seems more intuitive — a *method* since we need references on existing labelings to generate successive solutions. Each labeling, domain, expression, and variable is related to some `Solver`. This means, it contains a reference to its solver, which

[1] *Object Factory* is a term that is often used in Object-Oriented Software Development, while there is no formal definition. The canonical application case can be seen as follows: *In general, a factory implementation is useful when you need one object to control the creation of and/or access to other objects.*[13]

[2] Our current focus is on finite domain solvers. The approach, however, is not limited to that constraint system. Other arithmetical systems can be integrated as well.

```
public class Send {
    public static void main(String[] args) throws Exception {
        // Create solver from class name given as first program argument.
        Solver solver = (Solver) Class.forName(args[0]).newInstance();
        // Create variables with initial domains.
        Var s = solver.newVar(1,9);   Var m = solver.newVar(1,9);
        Var e = solver.newVar(0,9);   Var o = solver.newVar(0,9);
        Var n = solver.newVar(0,9);   Var r = solver.newVar(0,9);
        Var d = solver.newVar(0,9);   Var y = solver.newVar(0,9);
        // State constraints.
        solver.alldifferent(new Var[]{s,e,n,d,m,o,r,y});
        solver.eq(      s.times(solver.newX(1000))
                  .plus(e.times(solver.newX(100)))
                  .plus(n.times(solver.newX(10)))
                  .plus(d)
                  .plus(m.times(solver.newX(1000)))
                  .plus(o.times(solver.newX(100)))
                  .plus(r.times(solver.newX(10)))
                  .plus(e) ,
                        m.times(solver.newX(10000))
                  .plus(o.times(solver.newX(1000)))
                  .plus(n.times(solver.newX(100)))
                  .plus(e.times(solver.newX(10)))
                  .plus(y)
                  );
        // Create labeling (reference). No solution is computed here.
        Labeling l = solver.newLabeling(new Var[]{s,e,n,d,m,o,r,y});
        // Generate successive solutions and write them to stdout.
        while (l.nextSolution()) {
            System.out.println("   "+s+e+n+d);
            System.out.println("+  "+m+o+r+e);
            System.out.println("=======");
            System.out.println("= "+m+o+n+e+y+"\n");
        }
    }
}
```

Fig. 1. Java+POOC implementation of the Send-More-Money puzzle.

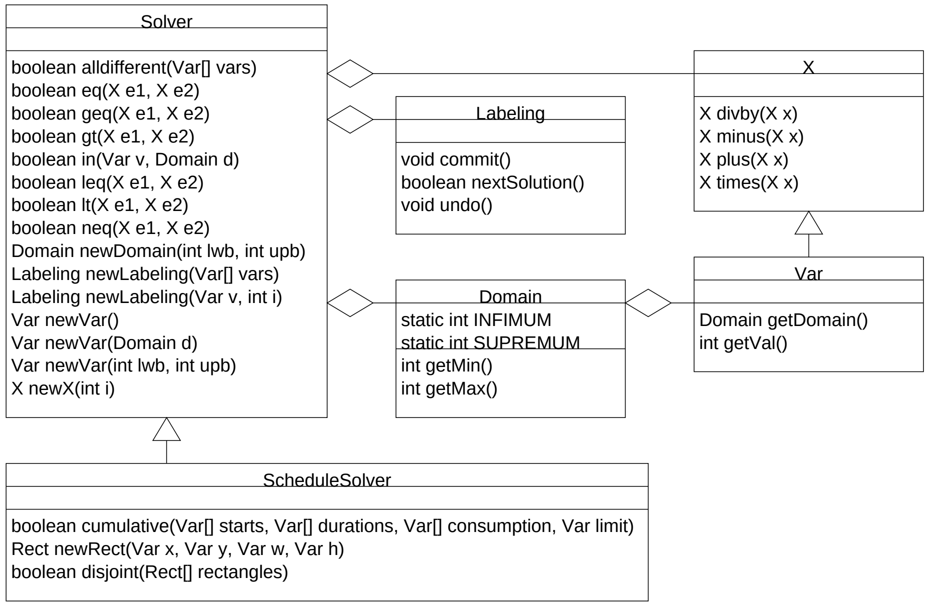

Fig. 2. POOC's classes

is depicted by the rhombic arrow. Each variable is related to its domain. The interface `Var` is a specialization (depicted by the up-arrow) of the interface `X`. This means that every variable is an expression. A `ScheduleSolver` is a `Solver` that implements additional combinatorial constraints: `cumulative` (e.g. [4]) and `disjoint` (e.g. [9]). `cumulative` ensures that a given number of tasks (defined by a starting time and a duration) that consume a certain amount of one common resource, never consume (in sum) more than the given limit. `disjoint` ensures that the given rectangles do not overlap. The latter constraint is kind of a generalization of the former. Both constraints are very well suited for solving resource allocation or scheduling problems. A `ScheduleSolver` serves also as a factory for rectangles.

This design has the following properties:

- It keeps the application program independent from a concrete solver (since the application can use abstract interfaces and object factories instead of concrete classes).
- It allows for the integration of additional constraint solvers (since it is rather abstract and hides implementation details like representation of domains or rectangles).
- It allows the use of more than one solver in one program (since each entity is related to its own solver).

The last property eases runtime comparison of constraint solvers.

3 Application

A typical Constraint Program consists of three major parts: definition of decision variables, statement of constraints, and search for a solution. Using POOC, all three parts are rather easy to implement. Figure 1 shows an exemplary implementation of the famous Send-More-Money puzzle.

The first step is to load a Constraint Solver. In Java, this can be done dynamically, i.e. the runtime implementation of the solver to be used can be given at runtime (here by passing the name of an existing class to the main program). The only requirement is, that the loaded solver implements a given interface: the `Solver`. This solver is a factory that serves for creating new variables, etc.

Therefore, new decision variables can simply be created by calling the solver's factory-methods `newVar()`. The application can provide an initial domain (using `newVar(Domain d)`) or lower and upper bounds directly (using `newVar(int lwb, int upb)`).

Constraints are then stated simply by using the solver's methods. Since in our example, all decision variables have to be assigned different values, we use the `alldifferent` constraint. To express the requirement that $send + more = money$, we have to build some arithmetic expressions: all decision variables will get a one-digit number, the puzzle's requirement can thus be stated as follows: `1000*s + 100*e + 10*n + d + 1000*m + 100*o + 10*r + e = 10000*m + + 1000*o + 100*n + 10*e + y`. The according expressions are built recursively using the variables, new number-expressions, and the given arithmetic operators[3].

Basic labeling (search for an appropriate variable assignment, i.e. solution) is simple: each solver provides an according method `newLabeling(Var[])`. The returned reference to a `Labeling` object serves as a handle for the generation of new solutions and commitment or disposal of old ones. The use is rather straight forward: `nextSolution()` generates a new variable assignment and returns `true` in case there is one and `false` if not. The basic labeling procedure that each solver has to provide, usually comes from a native implementation and sometimes allows the specification of options. These options are implementation-specific and therefore not directly reflected by the interface. It does, however, provide technical support for passing optional parameters from external resources.

Labelings can also be implemented directly in Java: using `newLabeling( Variable v,int i)`, a single decision variable can be assigned a concrete value, which can be taken back using `undo()`. As an example, Figure 3 shows an implementation of a simple depth first search: first variable first, lowest value first.

Found solutions can then be processed by the application: the variable's values can directly be queried using `getVal()` which throws an exception, if the variable is not yet instantiated. The standard method `Var.toString()` is overloaded such that variable's values can directly be printed.

[3] Since Java does — in contrast to C++ or Prolog — not allow to overload symbolic operators, we have to use real method-names. This makes our *arithmetic* expressions a bit clumsy, unfortunately.

```
static void label(Solver solver, Var[] vars) {
    label(solver,vars,0);
}

static void label(Solver solver, Var[] vars, int index) {
    if (index >= vars.length) { // solution complete
        for (int i=0; i<vars.length; i++) // print result
            System.out.print(vars[i]+" ");
        System.out.println();
        return;
    }
    Var v = vars[index]; // instantiate v
    for (int i=v.getMin(); i<=v.getMax(); i++) {
        Labeling l = solver.newLabeling(v,i);
        if (l.nextSolution())
            label(solver,vars,i+1); // instantiate next var
        l.undo();
    }
}
```

Fig. 3. User defined labeling procedure.

The presented usage and example show some of the advantages, our object-oriented design has:

- Rather abstract entities represent expressions, variables and more complex structures like rectangles for example.
- These entities can be related to each other using high-level constraints such as `alldifferent`, arithmetic constraints or rectangle-disjointness.
- The given classes can be aggregated to even more complex entities such as Tasks, Routes etc., or constraints like Multi-Resource or Round-Trip for the Traveling Salesman Problem.

The current implementation of POOC includes an implementation for SICStus Prolog [9]. SICStus (at least the new 3.8 versions) actually provides already a Java interface. This one is a rather low-level interface with which one can stick together Prolog terms, call them, and read resulting variable bindings. POOC is more abstract from Prolog and more concrete towards Constraint Programming.

In addition to the SICStus Prolog interface, POOC provides an interface to our own constraint solver `firstcs`.

3.1 `firstcs`

`firstcs`[4] is our new plain-Java constraint solver. It is designed as a light-weight special purpose solver. We intend to use it as a basis for new propagation al-

[4] This acronym is derived from its package name `de.fhg.first.cs`.

gorithms and for application development. The philosophy behind it is *right-sizing*, i.e. building the perfect solver for our problems, while eventually losing generality. Currently, basic and some higher-level FD-constraints are implemented, using state-of-the-art propagation techniques. For some mid-size real world multi-resource problems, we even outperformed commercial (non-Java!) constraint solvers.

The package we provide to the community (see 5) contains the current implementation of `firstcs`.

4 Extension

POOC's design allows the integration of an arbitrary number of constraint solver implementations. The effort of adding a new one depends on whether a Java-interface already exists and on the solver's execution model. First steps have been made plugging-in ILOG [11] and GNU Prolog [3]. Since it is a CLP-solver similar to the already plugged-in SICStus, the integration of the latter turns out to be not very problematic. We use Java's native interface JNI [12] to invoke GNU Prolog predicates via its C-interface. The integration of ILOG Solver v5.0, however, is very different in many respects. POOC generaly allows for applications mixing the statement of constraints and the search for a solution. ILOG needs these things be seperated, i.e. first the model being built and finally the solution being searched. Additionally, the definition of a Labeling is very special in ILOG. This is a bit a challenge for its integration into our framework, though not being impossible.

Solvers that provide special functionality to be used in POOC applications can be integrated with extensions of the `Solver`-interface. For example solvers that provide the `disjoint` resp. `diffn` constraint are subsumed in the `Scheduling-Solver` interface as shown in Figure 2. The same could be done for ILOG and other solvers, like JCL [8] for example, that strictly separate the modelling and solving of CSPs.

5 Download!

We realized the described interface as a Java package: POOC, which we are able to make open to the public! Please visit the dedicated web page at
`http://www.first.fhg.de/~hs/pooc` .

The package offered there for free download contains the POOC interface definitions, documentation, application examples, and implementations for SICStus Prolog and our solver `firstcs`. The latter solver is even contained in the package as well! Therefore you can get there a complete easy-to-use Java-Constraint-Solver.

The software is packaged using `jar`, the Java packager. This tool is part of every Java installation. Just downlaod `pooc.jar` from our website and call `jar xf pooc.jar`. This will result in a directory `POOC` that contains everything you need. Inside this directory, you can see the package documentation under `doc`.

The documentation contains descriptions on how to run the examples. For a first start, try

```
java de.fhg.first.fd.example.Order de.fhg.first.fd.firstcs.Solver 10
```

This loads the `Order` example (a number of variables, ordered by the Less-Than constraint `lt(X,X)`), with the Solver `firstcs` to be used, and the problem containing 10 variables. The program computes a solution and prints the result.

6 Conclusion

POOC gives a nice framework for the development of object-oriented Java-based Constraint Programs and the evaluation of existing Constraint-Solver-implementations. It

– is easy to use.
– allows for specifying underlying solvers at runtime.
– allows for multiple solvers even from different vendors within one program.
– is extendible.

The system is realized and provided to the public community. Please visit `http://www.first.fhg.de/~hs/pooc` .

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language Semantics and Notation Guide 1.3.* Rational Software Cooperation, San Jose, CA, 2001.

[2] Cosytec. Chip 5, 2001. `http://www.cosytec.com`.

[3] Daniel Diaz. GNU Prolog, 2001. `http://gnu-prolog.inria.fr`.

[4] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS88)*, 1988.

[5] Thom Frühwirth and Slim Abdenadher. *Constraint-Programmierung.* Springer, Berlin Heidelberg, 1997. In German.

[6] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[7] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87*, pages 111–119, 1987.

[8] EPFL Lausanne. Java constraint library. `http://liawww.epfl.ch/~akira/JCL/`.

[9] Swedish Institute of Computer Science. SICStus Prolog v3, 2001. `http://www.sics.se/sicstus.html`.

[10] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[11] ILOG S.A. ILOG internet pages, 2001. `http://www.ilog.fr`.

[12] SUN Microsystems, `http://java.sun.com/j2se/1.3/docs/guide/jni/`. *Java Native Interface (JNI).*

[13] SUN Microsystems, `http://java.sun.com/j2se/1.3/docs/guide/rmi/`. *Java Remote Method Invocation (RMI).*