# JaCoP Solver

The following information has been extracted from the manual:
http://jacopguide.osolpro.com/guideJaCoP.html

JaCoP library provides constraint programming paradigm implemented in Java. It provides primitives to define finite domain (FD) variables, constraints and search methods. The user should be familiar with constraint (logic) programming (CLP) to be able to use this library.

JaCoP library provides most commonly used *primitive constraints*, such as equality, inequality as well as *logical*, *reified* and *conditional constraints*. It contains also number of *global constraints*.

```
javac -classpath .:JaCoP.jar Main.java  *(generates Main.class)
java -classpath .:JaCoP.jar Main        *(runs Main.class)
```

In Java application which uses JaCoP it is required to specify import statements for all used classes from JaCoP library. An example of the import statements that import the whole subpackages of JaCoP at once is shown below.

```
import org.jacop.core.*;
import org.jacop.constraints.*;
import org.jacop.search.*;
```

Obviously, different Java IDE (Eclipse, NetBeans, etc.) and pure Java build tools (e.g., Ant) can be used for JaCoP based application development.

Other libraries useful for SAT:

```
import org.jacop.jasat.utils.structures.*
```

```
import org.jacop.jasat.utils.structures.*
```

The problem is specified with the help of variables (FDVs) and constraints over these variables. JaCoP support both finite domain variables (integer variables) and set variables. Both variables and constraints are stored in the store (`Store`). The store has to be created before variables and constraints.

```
Store store = new Store();
```

The store has large number of responsibilities. In short, it knits together all components required to model and solve the problem using JaCoP (CP methodology).

One often abused functionality is printing method. The store has redefined the method `toString()`, but use it with care as printing large stores can be a very slow/memory consuming process.

Variable *X* :: 1..100 is specified in JaCoP using the following general statement (assuming that we have defined store with name `store`).

```
IntVar x = new IntVar(store, "X", 1,100);

BooleanVar bv = new BooleanVar(s, "bv");
```

Primitive constraints and global constraints can be imposed using `impose` method. An example that imposes a primitive constraint `XneqY` is defined below. Again, in order to impose a constraint a store object must be available.

```
store.impose( new XeqY(x1, x2));
```

Alternatively, one can define first a constraint and then impose it, as shown below.

```
PrimitiveConstraint c = new XeqY(x1, x2);
   c.impose(store);
```

The depth-first-search method requires the following information:

- how to assign values for each FDV from its domain; this is defined by `IndomainMin` class that starts assignments from the minimal value in the domain first and later assigns successive values.
- how to select FDV for an assignment from the array of FDVs (`vars`); this is decided explicitly here by `InputOrderSelect` class that selects FDVs using the specified order present in `vars`.
- how to perform labeling; this is specified by `DepthFirstSearch` class that is an ordinary depth-first-search.

Different classes can be used to implement `SelectChoicePoint` interface. They are summarized in Appendix B. The following example uses `SimpleSelect` that selects variables using the size of their domains, i.e., variable with the smallest domain is selected first.

```
IntVar[] vars;
   ...
   Search<IntVar> label = new DepthFirstSearch<IntVar>();
   SelectChoicePoint<IntVar> select =
   new SimpleSelect<IntVar>(vars,
```

```
    new SmallestDomain<IntVar>(),
    new IndomainMin<IntVar>());
    boolean result = label.labeling(store, select);
```

A solution satisfying all constraints can be found using a depth first search algorithm. This algorithm searches for a possible solution by organizing the search space as a search tree. In every node of this tree a value is assigned to a domain variable and a decision whether the node will be extended or the search will be cut in this node is made. The search is cut if the assignment to the selected domain variable does not fulfill all constraints. Since assignment of a value to a variable triggers the constraint propagation and possible adjustment of the domain variable representing the cost function, the decision can easily be made to continue or to cut the search at this node of the search tree.

For extracting the value in a single solution of certain variable:

```
value = var.value();
```

### SAT solver

JaCoP has a SAT solver that can be used together with JaCoP solver. It has been developed by Simon Cruanes and Radosław Szymanek. The SAT solver can be integrated in JaCoP as a SAT constraint and used with JaCoP search methods. This functionality is offered by `org.jacop.satwrapper.SatWrapper` class. Typical translation of Boolean constraints, used in flatzinc compilations are defined in `org.jacop.satwrapper.SatTranslation`.

## 7.2 FDV encoding and constraint translations

`SatWrapper` offers a general way to define literals and clauses. The process is divided into two parts. First, each FDV and a inequality relation on it is bound to a literal. Then a clause is defined on the literals. We illustrate this process below.

Assume that we want to define a constraint $(a \leq 10) \rightarrow (b = 3 \wedge c > 7)$ with IntVar a, b and c. The literals for this constraint are defined as $a \leq 10$, $b = 3$ and $c \leq 7$ (negation of this literal is $a > 7$). This is encoded in JaCoP as follows.

```
    IntVar a = new IntVar(store, "a", 0, 100);
    IntVar b = new IntVar(store, "b", 0, 100);
    IntVar c = new IntVar(store, "c", 0, 100);
```

```
*consider the use of loops for defining variables for big instances
   SatWrapper wrapper = new SatWrapper();
   store.impose(wrapper);

   wrapper.register(a);
   wrapper.register(b);
   wrapper.register(c);

   int aLiteral = wrapper.cpVarToBoolVar(a, 10, false);
   int bLiteral = wrapper.cpVarToBoolVar(b, 3, true);
   int cLiteral = wrapper.cpVarToBoolVar(c, 7, false);
```

In the above method `cpVarToBoolVar` the first parameter is the FDV variable, the second is the rhs value and the third parameter defines type if relation (false for ≤ and true for =).

When literals are defined we can generate clauses using CNF format, as presented below. First, we translate the clause to CNF form as ($a \leq 10 \vee b = 3$) ∧ ($a \leq 10 \vee c \leq 7$). Then we encode this into JaCoP SAT solver.

```
IntVec clause = new IntVec(wrapper.pool);
 // first clause
   clause.add(- aLiteral);
   clause.add(bLiteral);
* the 2 previous lines are equivalent to: bLiteral ∨ (not) aLiteral
   wrapper.addModelClause(clause.toArray()); * clause ∧ previous clauses

   clause = new IntVec(wrapper.pool);

   // second clause
   clause.add(- aLiteral);
   clause.add(- cLiteral);
   wrapper.addModelClause(clause.toArray());
```

In the above code minus in front of the literal in statement `clause.add` means that the negated literal is added to the clause.

A possible solution generated by JaCoP is `{0, 3, 8 }`.