Published in the Proceedings of PACLP99
The Practical Application of Constraint Technologies and Logic Programming, London, April 1999

Constraint Programming in Java with JSolver

Andy Hon Wai Chun[1]

City University of Hong Kong
Department of Electronic Engineering
Tat Chee Avenue
Kowloon, Hong Kong
Tel: (852)-2788-7194 Fax: (852)-2784-4242
eehwchun@cityu.edu.hk

Abstract

This paper describes our progress in designing and developing a Java constraint programming class library called JSolver[2]. JSolver extends the object-oriented programming paradigm of Java with constraint-based declarative programming. Other constraint-based Java tools act as pre-processors to Java. JSolver, on the other hand, is implemented as a set of pure Java classes and allows direct development of constraint-based applications within a Java environment. Constraint programming tools like JSolver allow server-side development of scheduling and resource management systems in Java and deployment of these systems over the Web. JSolver permits interactive and dynamic constraint-based applications to be built, which is not possible with previous pre-processor approaches. This paper uses an extended example of a simplified airport bay allocation system – Micro-BAS, to illustrate the facilities provided by JSolver.

1. INTRODUCTION

Constraint programming has been a successful paradigm in recent years to implement algorithms to solve constraint-satisfaction problems (CSP) [6]. Many different types of real-life scheduling, resource allocation and configuration problems can be modelled as CSP and solved using constraint-programming techniques. Unfortunately, constraint technology was previously only available as extensions of Prolog, such as CLP [1, 4], CHIP [7], and DecisionPower [2], or extensions of Lisp, such as Screamer [5], or extensions of C++, such as ILOG Solver [3]. With a growing number of Java server-side development tools, there is an increasing advantage in using Java to implement application servers for scheduling systems. Application servers will normally contain an in-memory cache of Java business objects together with the business logic. JSolver allows the business logic to be implemented directly in Java as constraints and

[1] Dr. Andy Chun is also founder and Managing Director of Advanced Object Technologies Ltd., a high-tech company specialising in resource optimisation systems.
[2] JSolver is available for download from http://www.aotl.com

embedded with the business objects, thus simplifying the design; no native calls to C++ or CLP will be needed.

2. OVERVIEW OF JSOLVER

JSolver is our experiment in developing a tool to support constraint-programming paradigm within an object-oriented Java environment. We designed JSolver to have a small but sufficient footprint. It provides all the essential classes needed to support constraint programming with Boolean and integer constrained variables, which is a common representation for CSP algorithms used for resource allocation and scheduling. JSolver occupies only roughly 100KB of disk space zipped.

JSolver was designed as a class library that can be embedded into an application server to perform run-time operations. This is a different design philosophy compared with Declarative Java (DJ) [8] that provides syntax-extensions to Java to support constraint programming. DJ code is compiled into Java using B-Prolog. B-Prolog performs a once-only compile-time solution of the constraint problem. Therefore, DJ is not suited for use in a scheduling system that may require continuous re-solving due to changes in problem definition. DJ is suited for solving a static hard-coded problem while JSolver is suited for solving dynamic problems that involves user interactions and run-time events. In DJ, B-Prolog provides the constraint-solving mechanism, while in JSolver this mechanism is built using Java.

Even though there are highly efficient constraint solvers implemented in C++, JSolver still has its advantages. By having an application implemented in pure Java, without native calls to C++ or CLP, application deployment can be simplified. Furthermore, resource allocation and scheduling applications requires the use of a large number of business objects. Passing these objects from Java to C++ or CLP will cause a substantial overhead. From a software design point–of-view, it is more elegant to embed the business logic, implemented as constraints, with the business objects in the same Java environment. Furthermore, a pure Java constraint solver, like JSolver, allows Java application servers and Enterprise Java Beans to be built that are also constraint-smart.

```
public class Example {
  public static void main(String argv[]) throws FailException {
    Var a = JSolver.var(0, 20, "a");
    Var b = JSolver.var(0, 20, "b");
    JSolver.post(a.diff(b).gt(10));
    VarVector vars = JSolver.varVector(a, b);
    JSolver.solve(JSolver.generate(vars));
    System.out.println(vars);
  }
}
```

Most of the JSolver facility is accessible through a *JSolver* utility class. Most constraints are created through methods provided by the JSolver constrained variable classes – *Var* and *VarVector*. The above simple example illustrates the typical declarative programming style when using JSolver. In this example, we create two constrained

integer variables, both with a domain ranging from 0 to 20 and a name. We then post the constraint "a-b>10." The variables are then placed into a vector to be used as parameter to *JSolver.generate(),* which is a goal to instantiate each variable. The *JSolver.solve()* method performs the non-deterministic constraint-based search. (All JSolver class and methods are highlighted in the sample source codes.)

2.1 JSolver Variables

Constrained integer variables are created simply by providing the minimum and maximum domain values and an optional name. Constrained Boolean variables have just an optional name parameter. The JSolver provided constrained variable classes could be subclassed if needed to store additional application-specific attributes.

```
Var x = JSolver.var(0, 10, "x"); // integer variable x[0..10]
Var y = JSolver.boolVar("y"); // Boolean variable y[true, false]
```

Constrained variables may be stored in a vector data-structure using the *VarVector* class, which has the same interface as the standard Java Vector plus methods to define constraints. A vector is used to simplify passing arguments of variables. In addition, there are constraints that can be applied to a whole vector, such as the summation constraint. Variable vectors may be created in several ways:

```
JSolver.varVector(x, y);
JSolver.varVector(2, 0, 10); // two variables with domain [0..10]
JSolver.varVector(); // un-initialised vector
```

2.2 JSolver Constraints

Once variables and variable vectors are created, constraints may be posted. JSolver stores posted constraints in a constraint-network that allows constraint propagation to be performed efficiently without further runtime search. JSolver provides a set of standard unary and binary constraints. Application developers can easily extend the JSolver constraint facility by defining additional constraint classes simply by subclassing the JSolver *Constraint* class and overriding a few abstract methods. The following shows how an inequality constraint can be posted in JSolver:

```
JSolver.post(x.neq(5));
JSolver.post(x.neq(y));
```

JSolver also provides constraints on a vector of variables such as the "all different" constraint, cardinality constraints, and summation constraints. The "all different" constraint ensures that all variables within a vector take on different values. Cardinality constraints are used to define cardinality relationships for variables within a vector, such as defining the cardinality to be greater than, less than, or equal to a given value. The summation constraints define relationships on the total value of the variables within a vector. For example, total variable values must be less than a given value.

2.3 JSolver Revertible Classes

All JSolver constrained variables and constraints are automatically "reverted" or undone during backtracking. User-defined variables seldom need to be reverted. However, JSolver also provides this capability if needed for non-constrained variables. JSolver provides a set of classes to represent values that might need to be reverted to previous values as a result of backtracking. For example, JSolver supports revertible integers (*RevInteger*), revertible floats (*RevFloat*), and revertible Boolean (*RevBoolean).* These classes are analogous to Java wrapper classes with the additional capability to undo value assignments during backtracking. Normally, only constrained variables need to be reverted.

2.4 JSolver Search Algorithm

The following simplified flow-chart illustrates the CSP search algorithm provided by the JSolver class library. Normally, before the search, constrained variables of the problem are created and constraints are posted. Additional variables and constraints can be created dynamically during the search if needed.

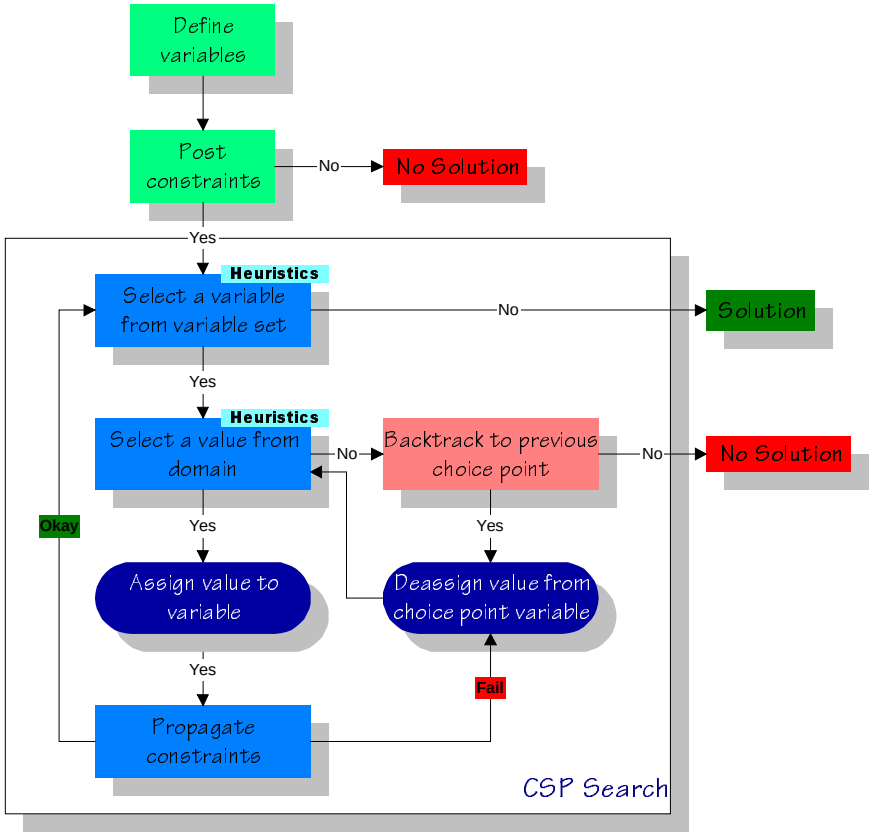

Figure 1. Simplified flow-chart of CSP algorithm implemented by JSolver.

The JSolver search algorithm selects one variable at a time from a given variable set to instantiate. By default, variables are selected in the order they are stored. JSolver provides a set of pre-defined strategies. However, additional strategies can be

implemented by subclassing the JSolver *ChooseVarHeuristic* class. Once all variables are instantiated a *solution* is found. Values from variable domains are selected using the minimum value first. Again JSolver provides a set of pre-defined strategies and additional strategies can be implemented by subclassing the JSolver *SelectValueHeuristic* class. Constraint propagation will occur after a value is assigned to a variable. Constraint propagation may cause domain reduction in associated constrained variables. If any domain becomes null, a *failure* is signalled. Once this happens, another value will be tried. If no other values are available, the JSolver algorithm will *backtrack* to a previous choice-point that does have alternative values and continue the search from there. If no more choice-points are left, the program throws a *FailException*.

3. EXTENDED EXAMPLE: MICRO-BAS

This Section illustrates the facilities offered by JSolver and the JSolver constraint programming syntax using a simplified airport stand/bay allocation system as an example. We will call this program Micro-BAS. Although the example is simplified, it still illustrates the key constraint programming components required by a full-scaled scheduling system.

The task of bay allocation is simply to assign a parking bay to each arrival aircraft. In our example, we will use a simplified airport as shown in Fig. 2. In this airport, like many others around the world, there are two main types of bays – remote bays and inner bays. Inner bays are attached to the airport terminal building. These bays are more desirable since passengers can walk directly into the terminal from the aircraft. Remote bays are remote parking locations that require buses to transport passengers to and from the terminal building.

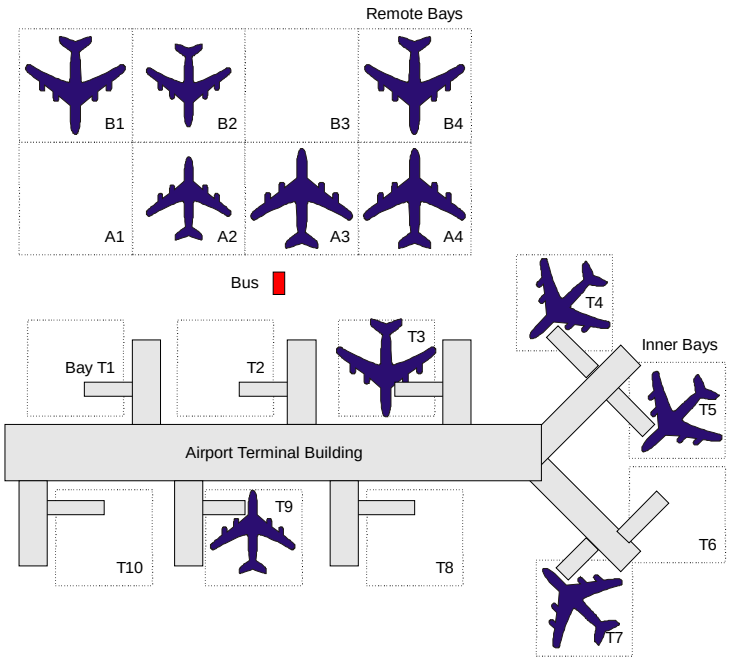

Figure 2. The simplified airport used by Micro-BAS.

Instead of considering each individual type of aircraft, we will simply assume that aircraft come in three sizes – small, medium, and large. The input to our Micro-BAS consists of two ASCII files – a daily schedule file and an airport configuration file. The daily schedule file contains the airline code, flight number, arrival and departure times, and the aircraft size. The airport configuration file contains the layout of the airport in terms of bay name, neighbouring bays, and distance from the terminal. The key bay allocation criteria considered by Micro-BAS are:

- Aircraft should not be assigned a bay that already has another aircraft.
- Only medium or large size aircraft can park at the inner bay.
- Two large aircraft cannot park next to each other.
- Larger aircraft have higher priority and should be allocated first.
- Prefer inner bays to remote bays.
- If remote bays are used, use bays closer to the terminal.

3.1 The OO Design

One of the key advantages of using object-oriented constraint programming is that we can combine benefits from Object Technology with Constraint Technology. An object-oriented (OO) methodology can thus be used to support constraint-based system design and development. For example, the modelling of the airport bay allocation problem can be integrated with traditional object-oriented analysis and design. For Micro-BAS, we start with the design of the airport domain classes:

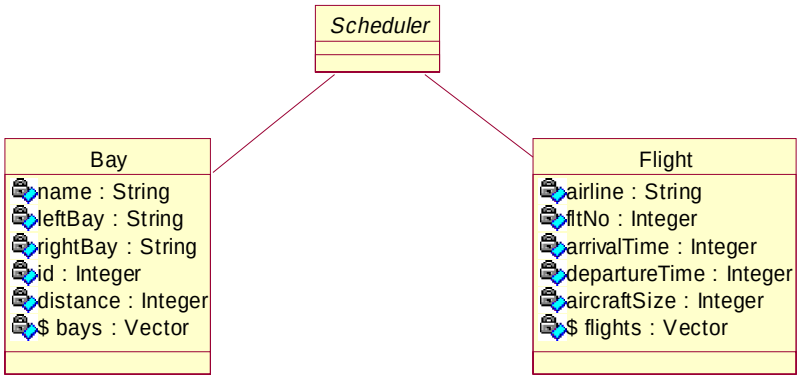

Figure 3. The initial UML Class Diagram of domain classes.

The *Flight* class represents a flight to the airport. For simplicity, we have represented both the arrival and departure flights as one object. In reality, these two entities are usually represented separately as *flight legs.* In addition, instead of representing the aircraft as a class, we have simply stored the size of the aircraft as an attribute of the *Flight* class. Arrival and departure times are represented as integers instead of date objects. The *flights* class attribute contains a Vector of all the flight objects known to the system.

```
public class Flight {
  private String airline;
  private int fltNo, arrTime, depTime, size;

  public final static int SMALL = 0, MEDIUM = 1, LARGE = 2;
  final static Vector flights = new Vector();
}
```

The *Bay* class represents both inner and outer parking bays. Each bay also knows its left and right adjacent bays. Since bays closer to the airport terminal building are more desirable, each bay also has a *distance* attribute, which is zero for inner bays. The *bays* class attribute contains a Vector of all the bays in the airport. The *counter* class attribute counts the total number of bays that have been created and is used to determine the upper bound for our constrained variables.

```
public class Bay {
  private String name, leftBay, rightBay;
  private int dist, id;

  private static int counter = 0;
  final static Vector bays = new Vector();
}
```

A *Scheduler* utility class is used to co-ordinate the scheduling task.

```
public abstract class Scheduler {
  public static void main(String argv[])
    throws IOException, FailException
  {
    Bay.load("bays.csv");
    Flight.load("flights.csv");
    // . . .
  }
}
```

3.2 Extending the OO Design with CP

Up to now, the OO design and Java implementation do not contain any constraint programming features. The next step is to augment the design with constrained variables. In Micro-BAS there is only one type of unknown – the bay to park an aircraft, and hence only one constrained variable. The constrained variable will represent the ID number of the bay to be assigned to an aircraft. We can model this as an attribute of the *Flight* class:

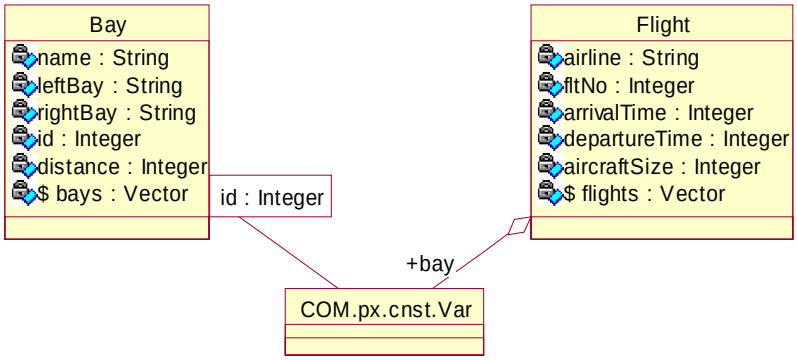

Figure 4. The class diagram extended with constrained variable.

The following are extensions of our previous implementation to support the constrained variable. The variable is initialised later after all the bays have been created in order to determine the upper bound of the domain. The *Bay* class provides a *find(int)* class method to link the bay ID with the actual bay object.

```
public class Flight {
  private Var bay;
  // . . .
}
public class Bay {
  public static Bay find(int id) {
  // . . .
}
```

3.3 Modelling and Implementing the Constraints

Restrictions on how constrained variable may be assigned values are represented as constraints and heuristics in JSolver. Within an object-oriented model, we can classify constraints as *inherent*, *intrinsic*, or *relational*. Inherent constraints are constraints that apply to all instances of a class. Inherent constraints are usually posted by the constructor. Intrinsic constraints are constraints that only apply to a particular instance. These constraints are posted right after an instance has been created. Relational constraints are those that represent relationship between two or more instances. These constraints are either posted before search or during the search using *value-action* demons. Value-action demons are programs that get executed whenever a constrained variable is assigned a value. Actions performed by these demons on constrained variables are undone during backtracking.

The advantage of classifying constraints in this way is to help determine how and where constraints should be implemented. According to this classification, Micro-BAS has one inherent constraint, no intrinsic constraint, and two relational constraints:

- **Inherent Constraints**
  [C1] Only medium or large size aircraft can park at the inner bay (posted in the Flight constructor)

- **Relational Constraints**
  [C2] Aircraft should not be assigned a bay that already has another aircraft (posted before search after all flights have been created)
  [C3] Two large aircraft cannot park next to each other (implemented as a value-action)

Constraint [C1] can be implemented using JSolver as shown below. A *FailException* will be thrown if constraint posting causes some variables to have a null domain. The *JSolver.var()* method creates a constrained variable with the given lower and upper domain bounds. The *JSolver.post()* method posts a constraint. The *Var.neq()* method creates an inequality constraint.

```
public class Flight {
  public Flight(String airline, int fltNo, int arrTime,
                int depTime, int size) throws FailException
  {
    // . . .
    bay = JSolver.var(0, Bay.getNumBays());
    postInnerBayConstraint();
  }
  void postInnerBayConstraint() throws FailException {
    if (size==SMALL) {
      for (int i = 0; i<Bay.getNumBays(); i++) {
        if (Bay.find(i).isInnerBay())
          JSolver.post(bay.neq(i));
      }
    }
  }
}
```

Constraint [C2] is posted before the search and can be implemented as shown below. Again, the *Var.neq()* method is used to create inequality constraints.

```
public class Flight {
  // . . .
  public static void postNoOverlapConstraint () throws FailException
  {
    for (int i = 0; i!=flights.size(); i++) {
      for (int j = i+1; j!=flights.size(); j++) {
        Flight iflt = (Flight)flights.elementAt(i);
        Flight jflt = (Flight)flights.elementAt(j);
        if (iflt.isTimeOverlap(jflt))
          JSolver.post(iflt.bay.neq(jflt.bay));
      }
    }
  }
}
```

Constraint [C3] is implemented as a value-action demon. Value-action demons are usually created in the constructor. We subclassed the JSolver provided *Demon* class and overrode the *Demon.executeDemon()* abstract method to represent constraint [C3]. The *Flight.checkAdjacencyConstraint()* method will throw a *FailException* if the constraint has been violated using the *JSolver.fail()* method. This will trigger the JSolver backtracking mechanism to backtrack to a previous choice-point and continue the non-

deterministic search from that point onwards.

```
public class Flight {
  public Flight(String airline, int fltNo, int arrTime,
                int depTime, int size) throws FailException
  {
    // . . .
    bay.addValueAction(new AdjacencyDemon(this));
  }
  void checkAdjacencyConstraint() throws FailException {
    if (size==LARGE) {
      Bay thisBay = Bay.find(bay.getValue());
      Bay leftBay = thisBay.getLeftBay();
      Bay rightBay = thisBay.getRightBay();
      for (int i = 0; i<flights.size(); i++) {
        Flight flt = (Flight)flights.elementAt(i);
        if (flt!=this && flt.size==LARGE
            && isTimeOverlap(flt) && flt.bay.isBound()) {
          Bay assignBay = Bay.find(flt.bay.getValue());
          if (assignBay==leftBay || assignBay==rightBay) {
            JSolver.fail();
          }
        }
      }
    }
  }
}
final class AdjacencyDemon extends Demon {
  public AdjacencyDemon(Flight flt) { this.flt = flt; }
  public void executeDemon() throws FailException {
    flt.checkAdjacencyConstraint();
  }
  private Flight flt;
}
```

3.4 Defining the Heuristics

Restrictions and guideline on how constrained variables may be assigned values can either be represented as CSP constraints as shown above or as heuristics to guide the CSP search. JSolver provides two types of search heuristics – heuristics on which variable to solve first and heuristics on which value to try first. These heuristics are implemented by subclassing the JSolver provided *ChooseVarHeuristic* and *SelectValueHeuristic* classes and then provide implementation to their abstract methods. In Micro-BAS, we have the following heuristics:

- **Choose Variable Heuristics**
  [H1] Larger aircraft have higher priority and should be allocated first.

- **Select Value Heuristics**
  [H2] Prefer inner bays over remote bays.
  [H3] If remote bays are used, use those closer to the terminal.

The [H1] heuristic can be implemented by overriding the *ChooseVarHeuristic.selectVariable()* abstract method:

```
final class ChooseFlight extends ChooseVarHeuristic {
  public int chooseVariable(VarVector vec) {
    for (int i=0 ; i<vec.size(); i++) {
      Var var = (Var)vec.elementAt(i);
      if (!var.isBound()) {
        Flight flt = (Flight)var.getObject();
        if (flt.getSize()==Flight.LARGE) return i;
      }
    }
    for (int i=0 ; i<vec.size(); i++) {
      Var var = (Var)vec.elementAt(i);
      if (!var.isBound()) return i;
    }
    return -1;
  }
}
```

The [H2] and [H3] heuristics are related and can be implemented by overriding the *SelectValueHeuristic.selectValue ()* abstract method:

```
final class SelectBay extends SelectValueHeuristic {
  public synchronized int selectValue(Var var) {
    for (VarEnumeration e = var.elements(); e.hasMoreElements(); ) {
      int bay = ((Integer)e.nextElement()).intValue();
      if (Bay.find(bay).isInnerBay()) return bay;
    }
    for (VarEnumeration e = var.elements(); e.hasMoreElements(); ) {
      int bay = ((Integer)e.nextElement()).intValue();
      if (Bay.find(bay).getDist()==1) return bay;
    }
    return var.getMin();
  }
}
```

3.5 Searching for a Solution

Instances of the heuristic classes defined above must be created and passed to the JSolver search algorithm before they can be used. The *Scheduler* utility class is used to initiate the CSP search using the *JSolver.solve()* method. The *JSolver.generate()* method creates a goal for *JSolver.solve()* to solve. *JSolver.generate()* simply instantiates each constrained variable in the vector provided by the first parameter. The second and third parameters to *JSolver.generate()* are just the heuristics described previously. The relational constraint [C2] *postNoOverlapConstraint()* is posted just before the search as mentioned before.

```
public abstract class Scheduler {
  public static void main(String argv[])
    throws IOException, FailException
  {
    Bay.load("bays.csv");
    Flight.load("flights.csv");
    Flight.postNoOverlapConstraint();

    if (JSolver.solve(JSolver.generate(Flight.getBayVars(),
                                       new ChooseFlight(),
                                       new SelectBay())))
      System.out.println(Flight.flights);
    else
      System.out.println("No solution!");
  }
}
```

4. CONCLUSION

This paper describes our JSolver implementation that enhances Java with constraint programming features. Through the extended Micro-BAS example, we illustrated the facilities and syntax of JSolver. There is a momentum in using Java for server-side development. We hope tools like JSolver can open up the possibility of embedding constraint technology into Java application servers or Enterprise Java Beans for scheduling and resource allocation.

Acknowledgements

The author would like to thank developers at Advanced Object Technologies Limited for their feedback in using initial versions of JSolver.

References

[1] J. Cohen, *Constraint Logic Programming,* Communications of the ACM, 33(7), pp.52-68, 1990.

[2] http://www.icl.com

[3] http://www.ilog.com

[4] J. Jaffar and J-L. Lassez, "Constraint Logic Programming," In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*, pp.111-119, 1987.

[5] J.M. Siskind and D.A. McAllester, *SCREAMER: A Portable Implementation of Nondeterministic Common Lisp,* Technical Report IRCS-93-03, University of Pennsylvania Institute for Research in Cognitive Science, 1993.

[6] G.L. Steele Jr., *The Definition and Implementation of a Computer Programming Language Based on Constraints,* Ph.D. Thesis, MIT, 1980.

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

[8] http://www.cad.mse.kyutech.ac.jp/~people/zhou