Building Java Applets by Using DJ – A Java-based Constraint Language

Neng-Fa Zhou
Department of Computer and Information Science
Brooklyn College
The City University of New York
2900 Bedford Avenue, Brooklyn, New York 11210-2889
zhou@sci.brooklyn.cuny.edu

Abstract

DJ (Declarative Java) is an extension of Java that supports constraint programming. On the one hand, DJ can serve as a high-level specification language for Java applets. To construct a Graphic User Interface (GUI) with DJ, the users only need to specify the components that compose the GUI and the relationships among the components by using constraints. On the other hand, DJ, as a constraint language, improves the current constraint languages in that problems and solutions can be described in the same language. In addition, since DJ is a compiling language that uses Java as the object language for compilation, solutions can be distributed on the WWW as Java applets. In this paper, we present DJ by examples, aiming at illustrating the power and programming methodology of DJ.

1 Introduction

The World Wide Web has become a great network for disseminating various kinds of documents. Web documents used to be static and to contain only text, but now many Web documents are dynamic, and contain not only text but also images, video, audio, graphics and animation. Java is a powerful language for programming interactive, dynamic, and content-rich Web documents as Java applets. Nevertheless, developing GUIs including applets in Java is a time-consuming process. The users have to, among other things, choose appropriate layout managers and sometimes have to determine the sizes and positions of graphic components. There are many visual tools, such as Java Studio^{TM}, for creating Java programs without the need to write any code. The users can manipulate graphic components directly by using mice in these tools. However, it becomes hard to achieve satisfactory accuracy when the number of components becomes large and the constraints on the layout become complicated.

DJ is a language that amalgamates Java and Constraint Programming. DJ, as an extension of Java, significantly simplifies the process of constructing GUIs and Java applets. The users only need to specify the components that compose a GUI and the relationships among the components by using constraints. The layout for the components is automatically determined by the system. As a constraint programming language, DJ improves the current constraint languages in that problems and solutions can be described in the same language. And most importantly, since DJ is a compiling language that adopts Java as the object language, results can be distributed on the World Wide Web as Java applets and/or included in other larger applications.

A compiler for DJ has been implemented in B-Prolog [8], a constraint logic programming system. For a DJ program, the compiler first extracts a constraint satisfaction problem (CSP) from the program, and then invokes the constraint solver to solve the problem and determines the attribute values for the components. It finally generates a Java program and an HTML file from the original DJ program and the solution obtained by the constraint solver.

In this paper, we describe the DJ language through examples. The readers are assumed to be familiar with Java [1] and constraint programming concepts [4, 5, 6]. The formal specification of the language and more example programs are available from DJ's Web page [9].

2 A Brief Introduction to DJ

DJ is an extension of Java that supports constraint programming. It retains the object-oriented feature of Java. A DJ program consists of several classes and optionally several constraint definitions. In addition to field and method declarations, a class can also include *dj-field declarations*, *constraints*, and *actions*.

```
ClassMemberDeclaration::-
    FieldDeclaration
    MethodDeclaration
    DJFieldDeclaration
    Constraint
    Action
```

Dj-field declarations declare the graphic components that compose the class and the attributes of the class. A component is an instance of some base class or user-defined class. *Constraints* are relations among components or component attributes. Unlike member variables in Java that can be updated, dj-fields are single-assignment variables whose values are determined automatically by the system based on the constraints among them. An *action* associated with a component specifies the action to take when some event happens to the component.

The extension to Java's syntax is kept as small as possible. A constraint has similar syntax to that of a conditional expression in Java, and a constraint definition is like a method declaration except that the body is a sequence of constraints. Because of this similarity, users who know Java and constraint programming can master the new language in a very short time. The users can never get confused about constraints and statements because they appear in different contexts.

The DJ system is an interpreter that takes a DJ program and the name of the main class in the program. No class in the program needs to be declared as the main class when the program is written. The interpreter creates an instance of the main class and determines the attribute values of the components that satisfy the constraints. It finally generates a Java applet that displays the components.

Consider how to construct an applet that displays a `Hello World` button. The following shows the code:

```
class HelloWorld {
    dj Button bt{text=="Hello World!"};
}
```

The program consists of only one class called `HelloWorld`. The line starting with `dj` is called a *dj-field declaration.* It declares a component called `bt`, an instance of the base class `Button`. The expression in the brackets after `bt` is called a *constraint* that constrains the value of the `text` attribute of `bt` to be `"Hello World"`.

Dj-fields are single-assignment variables. Each dj-field is the name for some particular object that is created automatically when an instance of the encapsulating class that owns the field is created. Unlike a member variable in Java, a dj-field cannot be updated and the object it refers to does not need to be created by calling a constructor.

`Button` is a base class. All base classes are extended classes of `DJComponent`, which has the following attributes: `x`, `y`, `width`, `height`, and `visible`. The attributes `x` and `y` specify the position, i.e., the xy-coordinate of the left-upper corner, and the attributes `width` and `height` specify the size of the component. The attribute `visible` indicates whether the component is visible or not. The default value for this attribute is true. In addition to the attributes inherited from `DJComponent`, `Button` has the following attributes of its own: `text`, `font`, and `color`. In our example, nothing is said about the color and font of the text, and the size and position of the button in the main panel. we can but do not have to give the values to these attributes. The system will determine the attribute values for us either by using the constraints we provided or by using the default values.

Each user-defined class has four default attributes, namely, `x`, `y`, `width`, and `height`, that specify the panel or layout area for the components in the class. No component can be laid outside this area.

We can separate the constraint on `bt` from the component declaration in our first `HelloWorld` class and write it as a member of the class as follows:

```
class HelloWorld {
    dj Button bt;
    bt.text=="Hello World!";
}
```

As the constraint is not inside the scope of the component declaration, to refer to the `text` attribute of `bt`, we have to write `bt.text`.

DJ retains the object-oriented feature of Java. By using inheritance, we can rewrite our `HelloWorld` class as follows:

```
class HelloWorld extends Button {
    text=="Hello World!";
}
```

The attribute `text` is inherited from the super class `Button`. Although the `HelloWorld` defined here has the same state and behavior as our original class, the two definitions are different. Let `hw` be a component of `HelloWorld` declared in some class. If the original definition is used, we refer to the size of the button as `hw.bt.size`. In contrast, if the class defined by using inheritance is used, we refer to the size of the button as `hw.size`. The attribute `size` in `HelloWorld` is inherited from its super class `Button`.

3 Drawing binary trees

This example illustrates how small components can be combined to build larger ones. The following shows the code for drawing the tree as depicted in Figure 1 (a):

```
class Tree {
    dj Circle root, lc, rc;
    dj Line l1,l2;

    sameSize({root,lc,rc});
    positionNodes(root,lc,rc);
    l1.point1==root.center;
    l1.point2==lc.center;
    l2.point1==root.center;
    l2.point2==rc.center;
}

constraint positionNodes(Circle root,
                         Circle lc, Circle rc){
    root.centerX == (lc.centerX + rc.centerX)/2;
                               // symmetric
    left(lc,rc);
    lc.centerY == rc.centerY;
    root.centerY == rc.centerY-50-rc.diameter;
}
```

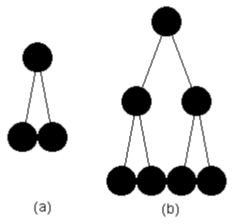

Fig.1 Binary trees.

The tree is composed of three circles and two lines. The constraint `sameSize` is a built-in constraint that takes an array of components and ensures the components are of the same size. The argument of the constraint `{root,lc,rc}` is an *anonymous array*. Anonymous arrays are very useful for grouping components and imposing some constraints on them.

The constraint `positionNodes` is a user-defined constraint. A constraint definition is similar in syntax to a method declaration, but it starts with the keyword `constraint` and its body is a sequence of constraints. The constraint `positionNodes` ensures that the root is located horizontally at the center of the two children and vertically 50 pixels above the two children.

The four equality constraints in the end of the class `Tree` ensure that line `l1` connects the root and the left child, and line `l2` connects the root and the right child.

Now we are ready to combine three small trees to build the bigger tree as depicted in Figure 1 (b). The following shows the code:

```
class BigTree {
    dj Tree t0,t1,t2;

    t0.lc == t1.root;
    t0.rc == t2.root;
    left(t1,t2);
}
```

This big tree is made up of three small trees, `t0`, `t1`, `t2`. The left child of `t0` is the same as the root of `t1`, and the right child of `t0` is the same as the root of `t2`. We see from this example that we can not only constrain attributes, but also unify components. Two components are unifiable if they belong to the same class and each attribute in one component is unifiable with its counterpart in the other component. The last constraint `left(t1,t2)` is necessary because otherwise `t1` and `t2` may overlap each other.

4 N-queen problem

DJ can be used not only to draw graphics and build graphical user interfaces but also to solve constraint satisfaction problems in general. In this example, we consider how to describe the N-queen problem and display the solutions graphically.

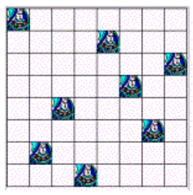

Fig.2 A solution to the 8-queen problem.

The problem is described as follows: There are an `N` by `N` chessboard and `N` queens. One is required to place the queens on the chessboard such that no two queens attack each other, i.e., no two queens can be placed on the same row, the same column, and the same diagonal.

Figure 2 shows a solution to the 8-queen problem. The following shows the code:

```
class Queens {
    static public final int N = 8;

    dj Square board[N][N]{fill==false};
    dj Image queens[N]{name=="queen.gif";
                       size==board[0][0].size};
    dj int pos[N] in 0..N-1;

    for (i in 0..N-1)
        samePosition(queens[i],board[i][pos[i]]);
    notattack(pos,N);
    grid(board);
}

constraint notattack(int[] pos,int N){
    for (i in 0..N-2,j in i+1..N-1){
        pos[i] != pos[j];
        pos[i] != pos[j]+j-i;
        pos[i] != pos[j]+i-j;
    }
}
```

N is defined as a constant. The chessboard is represented as a two-dimensional array of unfilled squares and the N queens are represented as an array of images. Every image has the same size as a grid square. We also use an attribute array, called pos, to indicate the positions of the queens. For each queen i, the element pos[i] indicates the number of the row on which the queen is placed.

The for constraint constrains the positions of the images. In general, a for constraint takes the following form:

```
for (Enumerator, ... ,Enumerator) Constraint;
```

where Enumerator is either a domain constraint in the form of V in D or a relational constraint. Let Vars be the set of variables appearing in all the enumerators. The for constraint means that for each tuple of values for Vars that satisfies the enumerators, Constraint must be satisfied.

In our example, the for constraint has the following meaning: For each queen i, the image queen[i] and the square board[i][pos[i]] must have the same position. Notice that indexes of arrays can be variables. This is a powerful feature. Without this feature, we would have to write the constraint as follows:

```
for (i in 0..N-1, j in 0..N-1, pos[i]==j)
    samePosition(queens[i],board[i][j]);
```

which is hard to read and inefficient since it generates N square samePosition constraints.

The notattack constraint, which is a user-defined one, ensures that no two queens attack each other. The grid constraint constrains the layout for the board. The signature for grid is as follows:

```
grid(DJComponent[][] comps)
```

It takes a two-dimensional array of components and ensures that the components are placed in a grid board.

5 Magic square

Magic square is another typical constraint satisfaction problem. There is an N by N grid board. One is required to assign each grid a different digit from 1 to N such that all the rows, all the columns, and both of the diagonals have the same total. This example illustrates the usage of *aggregation* constraints.

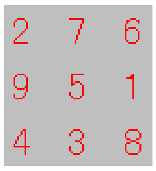

Fig.3 A solution to the 3×3 *problem.*

Figure 3 shows a solution of the 3×3 magic square problem. The following shows the code:

```
class Cell {
    dj Label lb;
    dj int value in 1..Magic.N*Magic.N;
    lb.text == String(value);
}

class Magic {
    public static final int N = 3;
    dj Cell board[N][N];
    dj int total;

    for (i in 0..N-1)   //columns
      sum(board[i][j].value, j in 0..N-1)==total;
    for (j in 0..N-1)   //rows
      sum(board[i][j].value,i in 0..N-1)==total;
    sum(board[i][i].value, i in 0..N-1)==total;
                        //left-upper-down diagonal
    sum(board[i][j].value, i in 0..N-1,
        j in 0..N-1, i+j==N-1)== total;
                        // left-bottom-up diagonal
    alldifferent(array(board[i][j].value,
                 i in 0..N-1,j in 0..N-1));
    grid(board);
}
```

Each grid square is a component of the class Cell, which has a label, called lb, and an integer attribute, called value, whose domain is declared to be 1..N^2. The constraint lb.text==String(value) ensures that value is the same as lb.text after it is converted to a string.

The board is represented as a two-dimensional array of cells. We use an integer attribute, called total, to represent the sum.

The expressions sum(...) in the constraints are called *aggregation* expressions. In general, an aggregation expression has the following form:

```
xxx(Exp,Enumerator,...,Enumerator)
```

where XXX can be substituted by sum, min, or max. Every variable in Exp must occur at least once in the enumerators. For each tuple of values for the variables that satisfy the enumerators, the Exp got a value. The aggregation expression stands for the sum, minimum, or maximum of all such values.

The two for constraints say that all the columns and rows have the same sum total. The two aggregation constraints say that the two diagonals have the same sum.

The alldifferent constraint takes an array of variables of some type and ensures that all the variables take different values. The argument is an anonymous array built by using an array expression. In general, an array expression has the following form:

```
array(Exp,Enumerator,...,Enumerator)
```

where Exp is an expression. For each tuple of values for the variables in the enumerators that satisfies the enumerators, Exp has a value. This expression represents an array of all such values. The array expression in our example converts a two-dimensional array of cells into a one-dimensional array of integers.

6 Building a calculator

Up till now, all the applets we have built are static in the sense that they do not respond to any events. We now consider how to build a working calculator, as depicted in Figure 4. This applet is dynamic and handles presses of buttons.

The following shows the program:

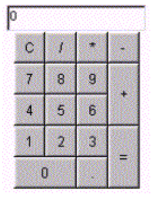

Fig.4 A calculator.

```
class Board {
    dj Button b[9]; // Buttons "0",...,"9"
    dj Button bc {text == "C"};
    ... // Buttons "+","-","*","/",".","="

    for (i in 0..9) b[i].text == String(i);
    grid({{bc,bdiv,bmul,bsub},
          {b[7],b[8],  b[9],  badd},
          {b[4],b[5],  b[6],  badd},
          {b[1],b[2],  b[3],  beq },
          {b[0],b[0],  bpoint,beq}});
    }
}

class Calculator {
    dj TextField tf{text=="0";columns==20};
    dj Board bd;
    ...

    // constraints
    above(tf,bd);
    sameCenterX(tf,bd);

    // actions
    for (i in 0..9) command(bd.b[i],inputDigit(char(i)));
    command(bd.bc,clear());
    command(bd.badd,inputOp('+'));
    ...
}
```

The class Board is composed of several buttons, each button for a key. The for constraint determines the values for the text attributes of the digit buttons. The expression String(i) converts an integer i into a string. The grid constraint constrains the positions for the buttons. This example illustrates a nice feature of the grid constraint: A component may occur multiple times in the array argument. In this case, the component will use multiple grid squares as its layout area. In our example, the buttons for 0, +, and = are twice as big as other buttons.

The Calculator class is composed of two components: a display and a board. The display is simply a text field which is already available as a base class in DJ. The text field must be located above the board and the centers must be aligned along the y-axis.

The command expressions specify the actions. In general, an action takes the following form:

```
command(Component,Event,MethodCall)
```

It attaches the action `MethodCall` to `Component` which will be taken when `Event` happens to the component. If `Event` is *right-clicked*, then it can be omitted. The following two methods are built-ins in DJ: `showDocument(file)` and `playClip(file)`. The former shows an HTML document and the later plays an audio clip.

The actions do calculations. They have no impact on the layout of the graphic components. Even if they change the layout of some graphical components, the constraints on the components will not be maintained dynamically. Constraints in the original program will be lost after an applet is generated. However, The methods `clear`, `inputOp`, etc. together with the variables they manipulate will remain.

7 Conclusion

Constraint programming languages and concepts have been gaining a wide acceptance recently. There are quite a few constraint languages around now [2, 3, 7, 5], and even some non-constraint languages also provide some constraint-like statements.

In this paper, we presented DJ, an extension of Java that supports constraint programming. We illustrated by examples that DJ can serve as a constraint programming language, in general, and serve as a powerful layout manager for graphic components, in particular.

Java's `GridBagLayout` layout manager allows the users to specify constraints that constrain the sizes and positions of components to be laid out in a grid board. Nevertheless, the constraints allowed are very limited. Constraints are all directional, and it is impossible to describe arbitrary layout of components.

Constraint logic programming (CLP) languages combine constraint solving and logic programming [5]. DJ is implemented in B-Prolog, but it is more than another syntactical sugar for CLP. DJ is object-oriented and provides arrays and many graphic classes. The lack of arrays and the object-oriented feature in CLP has long been considered a serious shortcoming. Compared with CLP languages, DJ is a domain-specific language. DJ is well suited for developing applets to be embedded in some Web documents and/or integrated into some larger applications.

There are constraint libraries for various languages, including very successful commercial products. The good thing of this approach to introducing constraint solving to a language is that everything is done in one language. The implementers do not need to implement and the users do not need to learn a new language. The bad thing is the lack of flexibility. The syntax for constraints is usually unnatural, especially for libraries in Java which does not support macro expansion and operator overloading. In addition, it is still hopeless now to implement a fast constraint solver in Java.

There are a lot of work remaining to be done to make DJ a nice tool for building Web documents and solving combinatorial optimization problems. Just like we develop various LaTex macros for different publishing purposes, we need to enrich the set of base classes in DJ to facilitate developing various Java applets. Also, we need to design classes for various combinatorial optimization problems such as scheduling, time tabling, and resource location.

8 Acknowledgements

This work was conducted while the author was with Kyushu Institute of Technology and was supported in part by AITEC under the grants *itaku-9-21* and *itaku-10-13* and by TAO under the grant *JSPS-RFTF97R16301.*

References

[1] K. Arnold and J. Gosling: *The Java Programming Language*, Second Edition, Addison-Wesley, 1998.

[2] K.R. Apt and A. Schaerf: Search and Imperative Programming, Proc. of 24th ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL '97), pp. 67-79.

[3] A. Borning, R. Lin, and K. Marriott: "Constraints and the Web", *Proceedings of the ACM Multimedia Conference*, pp.173-182, 1997.

[4] J. Cohen: Constraint Logic Programming Languages, *Communications of ACM*, Vol.33, No.7, pp.52-68.

[5] J. Jaffar & M.J. Maher: Constraint Logic Programming: A Survey, *J. Logic Programming*, Vols.19/20, pp.503-582, 1994.

[6] W. Leler: *Constraint Programming Languages, Their Specification and Generation*, Addison-Wesley Pub., 1988.

[7] P.V. Hentenryck: *The OPL Optimization Programming Language*, The MIT Press, 1999.

[8] N.F. Zhou: *B-Prolog Users Manual, Version 3.1*, Kyushu Institute of Technology, 1998, *http://www.cad.mse.kyutech.ac.jp /people/zhou/bprolog.html.*

[9] N.F. Zhou: *DJ Users Manual, Version 0.5*, Kyushu Institute of Technology, 1999, *http://www.cad.mse.kyutech.ac.jp /people/zhou/dj.html.*