DJ: A Java-based Constraint Language and System

Neng-Fa Zhou, Sousuke Kaneko, and Kouji Yamauchi
{zhou,kaneko,yamauchi}@cad.mse.kyutech.ac.jp
Faculty of Computer Science and Systems Engineering
Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, 820 Fukuoka

Abstract

DJ (Declarative Java) is an extension of Java that supports constraint programming. With DJ, users do not need to learn the complicated class hierarchy of the AWT package or Java's layout managers. To construct a GUI, users only need to specify the components that compose the GUI and the relationship among the components by using constraints. As a constraint language, DJ can be used not only to solve Constraint Satisfaction Problems (CSPs) but also to specify how solutions are displayed graphically. DJ is a compiling language that uses Java as the object language. Because of this, solutions of CSPs can be disseminated on the Internet easily.

1 Introduction

Constraint technology has proved effective for constructing GUIs [1, 3, 4]. Recently, as the popularity of Java grows, many people begin to introduce constraint solving into Java [2]. Most projects aim at integrating existing systems with Java and make resources in Java available to the systems. We designed and implemented a new constraint language, called DJ. This project has a two-fold objective. First, we want to significantly simplify the process of writing Java applets. And second, we want a constraint language that can be used not only to describe and solve problems, but also to display solutions graphically.

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. Besides variable and method declarations, a class can also include *component declarations*, *attribute declarations*, *constraints*, and *actions*. Component declarations declare the graphical components that make up 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, attribute values are determined by the system based on the constraints. An action associated with a component specifies the action to take when the component is clicked.

A compiler for DJ, which is about 11000 lines long including comments, has been implemented in B-Prolog [5, 6, 7], a constraint logic programming system. For a DJ program, the compiler first extracts the 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 as well as an HTML file from the original DJ program and the solution obtained by the constraint solver.

DJ can be used for two purposes: Firstly, DJ can be used for specifying Java applets. Unlike in Java where the users have to choose appropriate layout managers and sometimes have to determine the sizes and positions of graphical components, users of DJ only need to specify the components that compose a window and the relationship among the components by using constraints. The geometric attribute values of the components are all determined by the system.

Secondly, DJ can be used to solve CSPs. For a CSP, users can describe not only the variables and the constraints in the problem, but also how to display solutions graphically. Unlike in many constraint languages where users are required to describe problems and solutions in different languages, DJ is a language for describing both problems and solutions. Another important advantage of DJ is that solutions are Java applets, and thus can be disseminated on the Internet easily.

In this paper, we describe the syntax and semantics of DJ. The description is informal and based on examples. Consult DJ's web page [8] for details of the language and more examples.

2 An Informal Introduction to DJ

2.1 Component declarations

Let us start with the following example program:

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

```
}
```

This program defines a class called `HelloWorld`. The line starting with the keyword `component` is called a *component declaration*. It declares a component called `bt`. 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`. Notice that nothing is said about the color and size of the button, the font of the text, and the position of the button in the window. 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.

2.2 Attribute declarations

A class may also have several attributes. Each attribute has a type, a name, and possibly a constraint that restricts the domain of values for the attribute. Consider, for example, the following class:

```
class HelloWorld {
    component Button bt{text=="Hello World!"};
    attribute int centerX == bt.x+bt.width/2,
                  centerY == bt.y+bt.height/2;
}
```

The line starting with the keyword `attribute` is called an *attribute declaration*. For each component, whether it is of a base or of a user-defined class, there are four default attributes, namely, `x`, `y`, `width`, and `height`, that indicate its position and size. In our example, the attributes `centerX` and `centerY` are constrained to be the coordinates of the center of `bt`.

2.3 Constraints

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

```
class HelloWorld {
    component 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`.

A constraint is either a *basic* one or a *composite* one. A basic constraint is a *domain*, an *arithmetic*, a *symbolic*, or a call to a *user-defined constraint*. A domain constraint has the form `A in D` where `A` is an attribute or a component, and `D` is a domain expression which is either an integer interval `l..u` or a list of values of some type in the form $\{a_1, a_2, \ldots, a_n\}$. Arithmetic constraints have the same form as conditional expressions in Java. Symbolic constraints make it possible to define relations among components without referencing their attributes directly. A composite constraint is made up of basic constraints and various connectives.

2.4 Using inheritance

DJ retains the object-oriented feature of Java. A class `A` can extend another class `B`. In this case, class `A` is called a *subclass* of `B`, and class `B` is called the *super class* of `A`. 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 appearance as our original class, the definitions are different. Let `hw` be a component of `HelloWorld` declared in some class:

```
    component HelloWorld hw;
```

If the original definition is used, we refer to the size of the button as `hw.bt.size`. However, 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`.

2.5 Defining constraints

Users can define their own constraints. Constraint definitions do not reside in any class, and the constraints defined can be used by any class. This is a source for debate because it is not compliant with the pure object-oriented design principle. Nevertheless, as a relation is not directional, putting it into any one of the involved classes would make description unnatural.

The following shows another declaration of the `HelloWorld` class.

```
class HelloWorld {
    component Button bt;
        sameString(text,"Hello World!");
}
constraint sameString(String s1,String s2){
    s1 == s2;
}
```

The constraint `sameString` just trivially redefines the equality constraint over strings.

2.6 Arrays

It is possible to declare arrays of components and attributes in DJ. The following shows an example that use an array.

```
class Hellos {
component Button hws[10]{
          {text == "Hellos"},
          {text == "Bonjour"},
          ...
          {text == "Ni Hao"}};
}
```

This class has ten buttons each of which has a different text on it. A sequence of constraints enclosed in {} is called a `constraint block`. If there is only one constraint block following an array name, then the constraints apply to all the array elements; otherwise, if there is a sequence of constraint blocks enclosed in {}, then the first block applies to the `0`th element, the second block applies to the `1`th element, and so on. So, in this example, the button `hws[0]` has the text `Hello` on it.

Arrays that do not have names and that do not need to be declared called *anonymous arrays*. An anonymous array is a list of elements in the form of $\{a_1, a_2, \ldots, a_n\}$ where each element can be of any type. Anonymous arrays are very useful for imposing constraints on a group of components.

2.7 Iterative constraints

An iterative constraint takes the following form:

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

where `Enumerator` is a domain constraint or an arithmetic constraint. Let `Vars` be the set of variables appearing in all the enumerators. The iterative constraint means that for each tuple of values for `vars` that satisfies the enumerators, `Constraint` is satisfied.

The following shows an example.

```
for (i in 0..a.length-2, j in i+1..a.length-1)
        a[i] != a[j];
```

This constraint ensures that the elements in the array `a` are pair-wisely different.

2.8 Layout constraints

The following shows part of the symbolic constraints that are available for placing components in some regular ways.

```
grid(Object[][])
table(Object[][])
```

The grid constraint takes a two-dimensional array of components and ensures that the components are placed in a grid board. Suppose the array is $m \times n$. Then, the grid board is $m \times n$ and each grid `(i,j)` is covered by the component at `[i][j]` in the array. The `table` constraint is similar to `grid`, but places the components in a tabular form.

2.9 Aggregate constraints

An aggregation expression, which takes one of the following form, represents some aggregation of arrays

```
sum(Expression,Enumerator,...,Enumerator)
minimum(Expression,Enumerator,...,Enumerator)
maximum(Expression,Enumerator...,Enumerator)
```

For each tuple of values that satisfy the enumerator constraints, the `Expression` has a value. The `sum` expression refers to the sum , `minimum` refers to the minimum, and `maximum` refers to the maximum of all such values

An aggregate constraint is an arithmetic constraint in which at least one aggregation expression appears. The following shows two examples:

```
sum(a[i],i in 0..a.length-1) > 100
sum(1,i in 0..a.length-1, a[i]==u) < 5
```

The first constraint ensures that the total sum of the elements in the attribute array `a` is greater than `100`, and the second constraint ensures that the total number of elements in `a` that are equal to the variable `u` be less than `5`:

2.10 Event handling

DJ can handle only one event, i.e., what to do when a component is clicked. The following statement is just for this purpose:

```
command(Component,Action)
```

where `Action` is a method invocation. Command statements can appear anywhere a constraint can occur.

The following actions are built-in and can be used without definition:

```
showDocument(String name)
playClip(String name)
```

The first action shows a document and the second action plays an audio clip.

3 Applications

More than twenty examples have been developed. Figure 1 shows four of them.

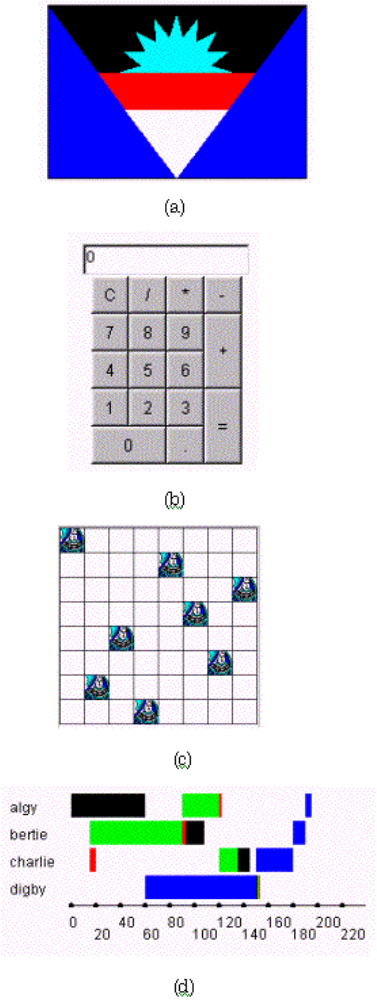

Figure 1: Applications.

Figure 1 (a) shows a national flag of Antigua and Barbuda. The flag is composed of a red rectangle, a black triangle, a yellow 16-star, a blue triangle and white triangle. The order in which the components are declared is very important. When two components overlap, the component declared later will overwrite the one declared earlier.

Figure 1 (b) shows a calculator designed by using DJ. Only one grid constraint is used for determining the layout of the buttons. For each button, there is a command that associates an action with the button.

Figure 1 (c) shows a solution to the 8-queen problem. There are `N`×`N` squares that form the chess grid board and `N` images. Every image has the same size as a grid. If the `i`th queen is placed in the `j`th column, then the `i`th image is placed at the grid `(i,j)`.

Figure 1 (d) shows a solution to a toy scheduling problem. There are four persons and four newspapers. Every person wants to read all the four newspapers. So, there are 16 tasks. The duration for each task is known. The problem is to find a schedule for the tasks such that the total duration is minimized.

4 Conclusion

DJ is a very simple constraint language. On the one hand, it extends Java by supplying it with a search engine. On the other hand, it improves the current constraint languages in that problems and solutions can be described in the same language, and the solutions can be shown on the Internet.

Several improvements and extensions are undergoing. One is to extend the base classes to make it easy to do drawing with DJ in various engineering areas. Another work is to implement a new placement algorithm.

Acknowledgement

This research is supported by AITEC during 1997-1999.

References

[1] A. Borning and R. Duisberg: "Constraint-Based Tools for Building User Interfaces", *ACM Transactions on Graphics*, Vol.5, No.4, pp.345-374, 1996.

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

[3] I. Cruz, K. Marroitt, and P.V. Hentenryck: *Special Issue on Constraints, Graphics, and Visualization*, Eds., *Constraints An International Journal*, Vol.3, No.1, 1998.

[4] A.B.A. Myers: *History of Human Computer Interaction Technology*, Tech. Rep. CMU-CS-96-163, Carnegie Mellon University, 1996.

[5] N.F. Zhou: "Parameter Passing and Control Stack Management in Prolog Implementation Revisited", *ACM Transactions on Programming Languages and Systems*, Vol18, No6, pp.752-779, November, 1996.

[6] N.F. Zhou: "A High-Level Intermediate Language and the Algorithms for Compiling Finite-Domain Constraints", *Proc. Joint International Conference and Symposium on Logic Programming*, pp.70-84, MIT Press, 1998.

[7] N.F. Zhou: *B-Prolog Users Manual, Version 3.0*, Kyushu Institute of Technology, 1998, http://www.cad.mse.kyutech.ac.jp /people/zhou/bprolog.html.

[8] N.F. Zhou: *DJ Users Manual, Version 0.3*, Kyushu Institute of Technology, 1998, http://www.cad.mse.kyutech.ac.jp /people/zhou/dj.html.