Integrating a Constraint Mechanism With the JavaBeans Model

William M. Shapiro

WUCS-98-12

May 1998

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Integrating a Constraint Mechanism With the JavaBeans Model

Undergraduate Thesis

Washington University

Department of Computer Science

By

William M. Shapiro

May, 1998

Thesis Advisor: Dr. Kenneth J. Goldman

Integrating a Constraint Mechanism With the JavaBeans Model

William M. Shapiro
Washington University
Dept. of Computer Science
wms1@cs.wustl.edu

Abstract

The JavaBeans component model allows users to plug together software components to create Java applications by specifying simple relationships between component events and properties. This paper describes work on augmenting the simple JavaBeans model with a multi-way constraint mechanism that allows users to graphically specify more complex multi-way constraints, resolve cyclical constraints between bean properties and graphically layout bean components. We also discuss weaknesses in the JavaBeans model and Java Abstract Windowing Toolkit (AWT) that were discovered while integrating a constraint mechanism with JavaBeans.

1. Introduction

The primary goal of the JavaBeans component model [6] is to allow users to plug together existing software components to create Java applications, requiring little or no programming. Existing development environments (e.g., JBuilder [8], Java Studio [7]) allow developers to graphically create simple Java applications by creating event channels, where events flow from sources to listeners, and by creating simple relationships between properties of different beans.

However, many applications require more complex and dynamic relationships between their constituent components. Several types of applications benefit from these additional features including animations, design tools, and prototyping tools. For example, Figure 1 shows how constraints can be used to rapidly prototype applications. The application depicted is a simple graphing application for three stocks that continually updates the graph to reflect the current stock prices. The prototype was created in less than 10 minutes from pre-existing beans using the constraint mechanisms discussed in this paper.

Another example application is an interior design kit. The kit would provide different layout components that the user could dynamically add and remove from the design such as windows, carpeting, walls, and furniture. Constraints simplify the user interface by allowing the user to easily snap

together components. For example, by equating the positions of different corners of walls the user is able to join two walls with a single mouse movement. Additionally, the user may use constraints to specify, for example, that all walls have the same color by creating constraints between their color properties. The user would then be able to change the color of all walls in a single operation.

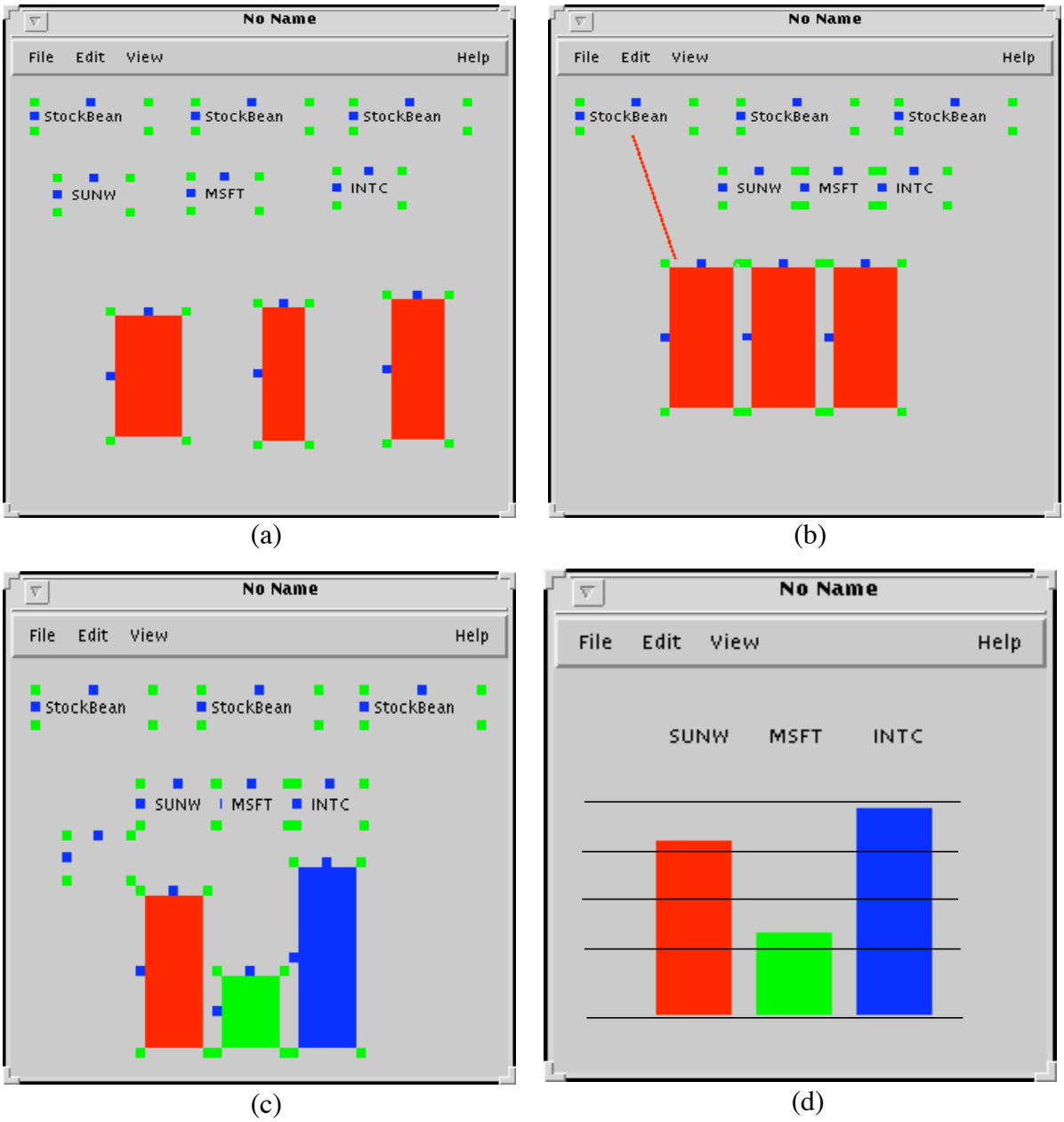


Figure 1: Rapid Prototyping of an Interactive Graph

This paper presents our experience with integrating a user constraint mechanism that allows users to specify multi-way constraints between bean properties with the JavaBeans model. The primary

advantages of a constraint mechanism are:

- simple graphical layout of components,
- concise specification of complex, dynamic relationships between bean properties, and
- automated maintenance of such relationships.

The paper proceeds as follows. Section 2 provides an overview of the JavaBeans model. Section 3 introduces user interface constraints, discussing the constraint algorithm that is used in the project and related work in multi-way constraints. Section 4 explains how user interface constraints are integrated with the JavaBeans model. Section 5 discusses several limitations in the JavaBeans model and the Java AWT that were discovered during this project. Finally, Section 6 provides some concluding remarks and possible future extensions and enhancements to the project.

2. The JavaBeans Model

JavaBeans is a component architecture that allows independent developers to write components called "beans" that can later be composed together by a third party to create Java applications. The JavaBeans specification [4] defines a bean as "a reusable software component that can be manipulated visually in a builder tool." The JavaBeans model includes a set of APIs that allow developers to specify bean properties, bean events and interactions between beans. This section briefly describes the aspects of the JavaBeans model that are relevant to this paper. A comprehensive description of JavaBeans can be found in [6].

2.1 Bean Properties

A property is a named attribute of a bean. Properties are defined by getter and/or setter methods on the bean. For example, if the user specifies a method *getFoo* and a method *setFoo* for a bean, **foo** would be assumed to be a property of that bean. Thus, the JavaBeans API uses a simple design pattern to resolve properties in a bean. Using introspection, the JavaBeans API is able to determine which methods a bean supports. A property can be read-only, write-only, read/write, bound and/or constrained. If a property is bound, it means that whenever the property is changed, any object that is registered as a listener for that property is notified. If a property is constrained, whenever the setter method for that property is called, all objects that are listeners for that property have a chance to veto the change, preventing it from occurring. It is important to note that a "constrained proper-

ty" is a JavaBeans term and is different from the notion of a constraint between properties, which will be explained later in the paper.

The JavaBeans API uses simple conventions to determine what a property is. For example, if a bean only has a getter method for a specific property, the property is read-only. Bound and constrained properties are more complicated. A bean may support bound properties if it has an *addPropertyChangeListener* method and a *removePropertyChangeListener* method. However, it is not possible to know if a particular property within a bean is "bound" without further information because the writer of the bean must explicitly add code to the setter method for the property to specify that it notify all listeners of the change.

A small example will help to clarify this issue. Below is a simple bean that has two properties, "dollars" and "cents":

```
public class Money {
   public Money(int dollars, int cents) {
      this.dollars = dollars;
      this.cents = cents;
   }

   public int getDollars() {
      return dollars;
   }

   public void setDollars(int dollars) { //bound property
      int oldDollars = this.dollars;
      this.dollars = dollars;
      changes.firePropertyChange("dollars", new Integer(oldDollars),
                                 new Integer(dollars));
   }

   public int getCents() {
      return cents;
   }

   public void setCents(int cents) { //not bound
      this.cents = cents;
   }

   public void addPropertyChangeListener(PropertyChangeListener l) {
      changes.addPropertyChangeListener(l);
   }

   public void removePropertyChangeListener(PropertyChangeListener l) {
      changes.removePropertyChangeListener(l);
   }

   private int cents;
   private int dollars;
   private PropertyChangeSupport changes =
```

```
            new PropertyChangeSupport(this);
    }
```

First, notice that there is no magic in this code. The user must explicitly create methods for adding and removing listeners as well as notify listeners of a property change. However, the *PropertyChangeSupport* class does help to simplify the process by handling all of the book keeping. Note that it is not possible to determine which properties are actually bound without looking at the body of the setter method for that property.

Because there is potentially much information that cannot be determined by performing introspection on a bean, the JavaBeans API includes a *BeanInfo* interface that a class can support to provide more information about a bean. This class can be either the bean itself, or a separate class that is bundled with the bean. The developer of the bean can then specify which bean properties are bound as well as other information.

A property may be constrained in the same way that it can be bound, except that the *VetoChageSupport* class is used in the place of the *PropertyChangeSupport* class. Additionally, a property that is constrained has a setter method that throws a *VetoableChangeException*. Therefore, it is possible to determine if a property is constrained by examining the exceptions that it throws.

If several beans are listeners for a constrained property and a change is vetoed, the listener beans are not notified of this change. Therefore, constrained properties are often also bound. In this way, all listeners are first given the chance to veto a change to a property. If the change is not vetoed, all listeners receive a property change event, which signifies that the change was not vetoed.

2.2 The Java 1.1 Event Model

Events are the mechanism through which JavaBeans communicate. In the Java 1.1 event model, event objects consist of *source* objects and *listener* objects. A source object produces events and a listener object may subscribe to receive events produced by a source. Examples of event classes include Graphical User Interface (GUI) events (e.g., MouseEvent) and a change in a bean property (PropertyChangeEvent). Figure 2 provides a simple graphical overview of the event model.

For an object to be an source object, it should implement addXListener and removeXListener methods, where “X” is the name of the event. Additionally, the object must notify all listeners that have been added when the given event or events occur. The JavaBeans API provides *Property-*

ChangeSupport and *VetoableChangeSupport* classes, as mentioned in the previous section, that facilitate the creation of source objects by keeping track of registered listeners.

For an object to be a listener, it must implement a Listener interface (an interface that inherits from java.util.EventListener). The listener interface is generally named after the type of event it is listening for (e.g., MouseListener for mouse events). Again, the JavaBeans API provides *PropertyChangeListener* and *VetoableChangeListener* interfaces for creating listener objects.

The XEvent, XEventListener class pair specifies how event notification takes places for a specific

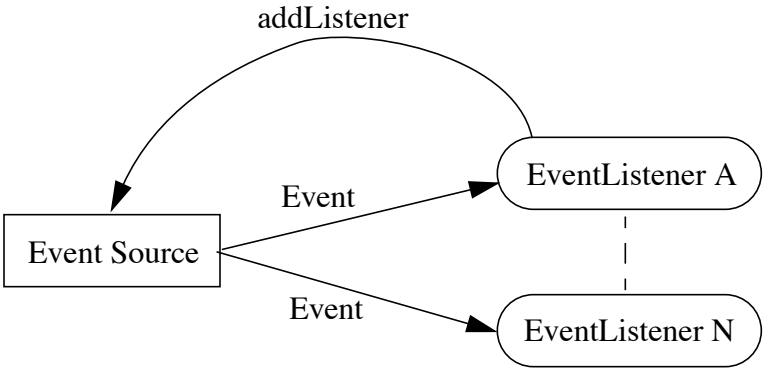


Figure 2: Simple Overview of the Java 1.1 Event Model

event or set of events. The eventListener interface specifies the method(s) the listener must implement to receive event notification. The event class, in turn, knows which method to call on the event listener and what arguments to provide to notify the listener of a particular event.

3. User Interface Constraints

Constraints are used to specify relationships between properties. For example, the constraint: "left + width = right," in a rectangle enforces the relationship that the position of the left side of the rectangle plus the width of the rectangle is equal to the position of the right side of the rectangle. One or more *methods* are associated with each constraint, which are used to satisfy an equation. A method consists of code that computes the value of an output variable based on the values of input variables.

User interface constraints are useful because they allow users, through the use of interactive constraint declarations, to specify the graphical layout of components, relationships between graphical

component properties and how components should respond to external events. The *constraint solver* is then responsible for solving these constraint relationships.

3.1 One-way Versus Multi-way Constraints

A constraint may be *one-way*, in which case the constraint is only on one direction. Assignment is an example of a one-way constraint (e.g., x →y), where the right side of the assignment statement is always updated to equal the left side, but not the opposite. A *multi-way* constraint specifies a constraint in multiple directions. For example, the equality relationship, "x=y," is a multi-way (in this case, two-way) constraint that specifies that y should change to reflect a change in the value of x and that x should change to reflect a change in the value of y.

One-way constraints have a single method that satisfies them and a single output variable -- the variable that is modified when the constraint is solved, whereas multi-way constraints have several satisfying methods and a single output variable. A one-way constraint system is generally more predictable and satisfiable than a multi-way constraint system, although it is not guaranteed to be satisfiable.

A multi-way constraint system is more likely to be unsatisfied and unpredictable because there may be multiple ways to satisfy each constraint (if it is under-constrained) or one or more constraints may not be satisfiable at all (in the case that it is over-constrained). If, for example, a constraint system consists solely of the constraint "A=B+C", and the value of A is changed, the system can be satisfied by changing either B or C. This is an example of an under-constrained system, where a method must arbitrarily be chosen to satisfy a constraint. On the other hand, if the system also included the constraints "A=5" and "B=7" and "C=3", the system would be over-constrained and unsatisfiable.

However, multi-way constraints are also more powerful because they allow users to specify more complicated constraint relationships. Returning to our first example: "left + width = height," we want this equality to hold regardless of which variable we change. If we change the width, the left and/or height should change to maintain the equality.

3.2 Constraint Hierarchies

One of the primary problems with constraints systems, as mentioned above, is their unpredictabil-

ity. A common solution to this problem is to assign different strengths to different constraints, creating a hierarchy that determines which constraints the constraint solver should enforce. Generally, some constraints are labelled as *required*, meaning that they must be enforced. For a rectangle object, for example, there would be a requirement that the location of the left side plus the width must equal the location of the right side. Constraints that are not required are then enforced according to their relative strengths.

3.3 UltraBlue

The constraint solver I have chosen for this project is UltraBlue[10]. UltraBlue is an incremental multi-way constraint solver, meaning that it does not try to solve all constraints simultaneously (as linear equations), but rather, incrementally tries to enforce constraints based on their relative strengths.

UltraBlue uses a cycle avoidance heuristic to resolve cyclic constraint relationships. Figure 3 shows how UltraBlue avoids a simple constraint cycle. The initial configuration is shown in (a). There is a weak *stay* constraint specifying that the value of A should remain constant and a constraint specifying that "C=A+B." When a new constraint ("B=C") is added in (b), UltraBlue changes the propagation direction of the constraint "C=A+B" so that A is the new output variable, preventing a cycle from forming. Because the constraint "A=C-B" has a higher strength, the weaker constraint, "A=5" is unenforced.

UltraBlue is able to resolve cycles created by a new constraint in $O(DN^2)$ time, where N is the number of constraints and D is the number of output variables. However, UltraBlue is not guaranteed to satisfy all constraints, even if an acyclic solution exists (i.e., all constraints can be enforced).

At least one more recent algorithm (see Section 3.4) is guaranteed to provide an acyclic solution

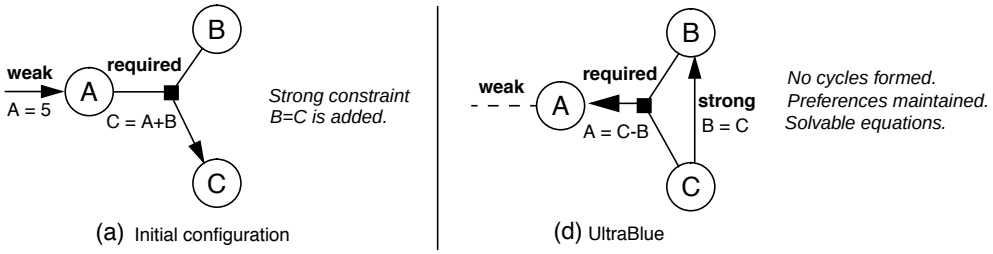


Figure 3: An example of how UltraBlue avoids cycles in a constraint graph. from [10].

to a hierarchy of multi-way constraints if an acyclic solution exists, and with the same time com-

plexity as UltraBlue. Nonetheless, I chose to use UltraBlue for several reasons. First, the C++ code for UltraBlue was available, therefore making it very easy to port to Java, which satisfied important time-constraint considerations for this project. Second, UltraBlue is a relatively simple algorithm, making it easy to implement and debug, and the developer of the algorithm was accessible. Finally, UltraBlue has been used in a large application (EUPHORIA [9]), and works fairly well in practice.

3.4 Related Work

This section provides a brief discussion of related work in constraints, based largely on [10]. Many systems have utilized one-way constraints to solve similar problems. However, this section is only concerned with the work in multi-way constraints.

The SketchPad system [5] was the first graphical system to use constraints. SketchPad allowed users to specify constraint relationships between graphical objects, which were maintained in real-time.

DeltaBlue [2] was one of the first algorithms to provide support for solving hierarchies of multi-way constraints. DeltaBlue uses a comparator known as "locally-predicate-better" to choose which constraint to enforce. DeltaBlue, however, does not attempt to solve or avoid cyclical constraints. When it finds a cycle, it arbitrarily unenforces a constraint, regardless of its strength. UltraBlue improves on DeltaBlue by using a heuristic function to try to avoid cycles.

A recent algorithm, QuickPlan [3], further improves on DeltaBlue. QuickPlan is guaranteed to solve a hierarchy of multi-way constraints if an acyclic solution exists in $O(N^2)$ time in the worst case. The only requirement imposed by the algorithm is that all variables used in a constraint must be either input or output variables in the constraint.

QuickPlan is based on the concept of "propagation of degrees of freedom". In short, the constraint solver works as follows. It finds a variable that is attached to a single constraint and is an output variable for that constraint. The corresponding method used to satisfy the constraint is selected and the variable and its edges are removed from the graph. The process then repeats on the resulting subgraph until all variables have been removed or all contain more than one constraint. In the later case, no acyclic solution exists and the algorithm uses constraint hierarchies to determine which constraint(s) to retract.

Another recent algorithm, Ultraviolet [1], stems from the realization that there is no fully general means known to satisfy constraints. The algorithm is, therefore, a hybrid algorithm that uses different techniques for solving constraint hierarchies based on the desired properties of the solver. For example, the solver may choose whether to allow cycles, inequality relationships, multi-way or one-way constraints.

To allow such flexibility, Ultraviolet supports several cooperating subsolvers including:

- A local propagation solver for functional constraints
- A local propagation solver for constraints on the reals, including inequalities.
- A solver for simultaneous linear equations.
- A solver for simultaneous linear equations and inequalities.

A constraint network is represented as a bipartite graph and the graph is partitioned into subgraphs, each of which can use a separate subsolver. Communication between the separate subsolvers then takes place through the variables they have in common.

4. Adding A Constraint Mechanism to a Simple Java BeanBox

This section describes the architecture of the constraint-enhanced BeanBox described in this paper. Sun's reference BeanBox, which is described in Section 4.1, was used as a starting point for this project. Section 4.2 describes the two constraint mechanisms that were added to Sun's reference BeanBox.

4.1 Sun's Reference BeanBox

Sun's reference implementation consists of a list of sample beans, the actual BeanBox that beans can be dropped into, and a property sheet for the highlighted bean. Figure 4 shows a BeanBox containing three beans. The "blue button" bean is highlighted and its property sheet, which allows the user to customize bean attributes (e.g., background, name), is shown on the right. The BeanBox supports both a design-time mode, where beans can be added, customized and removed, and a run-time mode, where all configuration information is hidden.

Architecturally, the BeanBox consists of the *BeanBoxFrame* class, which is the window itself. Embedded in the window is a *BeanBox* class, which is considered a bean. Each bean is contained in a *Wrapper* class that, among other things, draws the border around the selected bean and allows

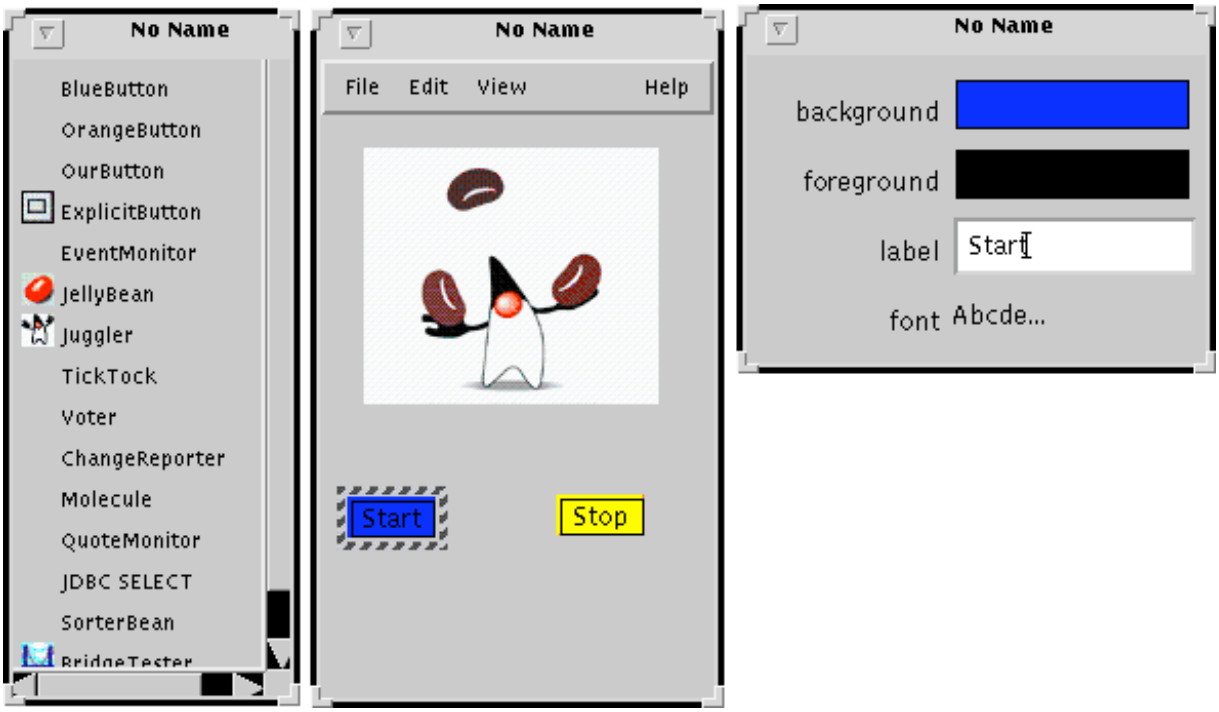


Figure 4: Sun's Reference BeanBox

the bean to be resized and moved.

Sun's beanbox provides support for both binding properties and hooking up events. Binding two properties, A to B, specifies that changes in the value of A are propagated to B, but not vice versa. Therefore, a bound property is similar to a one-way constraint, but only simple constraints are supported. We will see in the next section how the constraint mechanism presented in this paper improves upon the simple binding mechanism. An event hookup causes a method to be called on the *target* bean when an event occurs on the *source* bean. To create the event hookup, a class is automatically generated that is a listener for the source event and invokes the target method when the given event occurs.

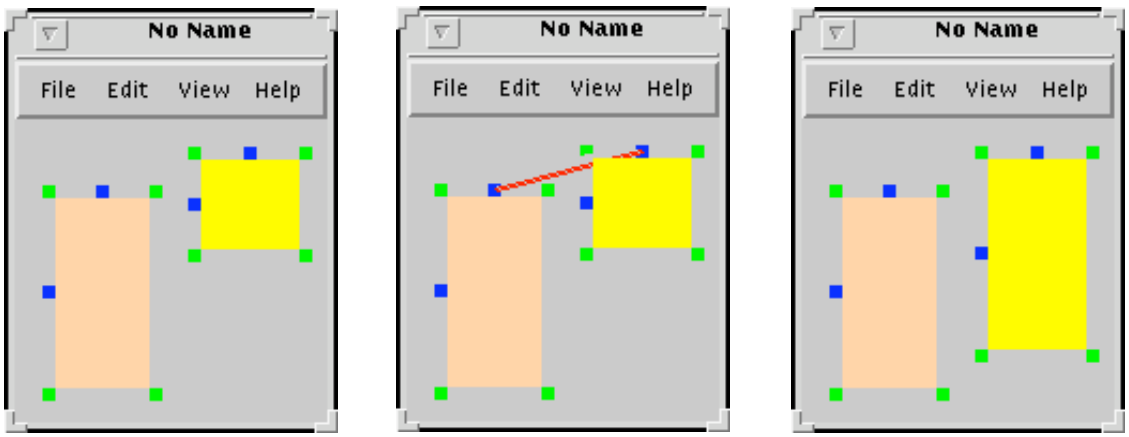
4.2 BeanBox Enhancements

Two separate constraint mechanisms were added to the BeanBox. One mechanism allows users to graphically constrain general bean attributes (i.e., attributes that all graphical beans share) such as width, height and position through the use of graphical handles. The second mechanism is more general, allowing bean properties to be constrained using menus to specify which properties should be constrained.

Ideally, a single constraint mechanism would support all desired operations. However, two separate constraint mechanisms were used for several reasons. Graphical handles are used to manipulate attributes that all beans have in common (e.g., size, location), and are generally completely under the control of the BeanBox (the bean may specify a minimum or preferred size, but rarely places other limits on its size, location, etc.). Therefore, the BeanBox can dictate what values these attributes should have and how they can change. On the other hand, general bean properties (as described in Section 2.1) are entirely dependent on each bean. Additionally, not all properties can be changed and some changes may be vetoed. Finally, the BeanBox may not simply change property values, but must use the setter method which may perform an arbitrary number of additional operations on the bean. This requires performing introspection on the bean to find and invoke the method and incurs significant overhead.

4.2.1 Handle Constraints

Every visual wrapper that is dropped into our BeanBox is instantiated with surrounding handles that allow the bean to be resized and moved. Figure 5 shows how a user can constrain the heights of two beans to be equal by drawing a line with the mouse between their height handles. Each green handle (at each corner) represents a corner of the bean. The two blue handles (at the middle of the left and top side) represent the width and height of the bean. By clicking on a handle with the left mouse button and dragging the mouse, the user is able to resize the bean, subject to the constraints that have been specified for the bean. By clicking on a handle with the right mouse button and dragging the mouse to another handle, the user is able to create an equality constraint between two handles (as was done in Figure 5).



(a) (b) (c)

Figure 5: Using handles to constrain the height of two beans

Figure 6 shows the Object Modeling Technique (OMT) diagram for the constraint handle portion of the BeanBox[1]. Each Wrapper contains a *ConstraintHandles* class that maintains a rectangle constraint graph for the given bean. When a handle is graphically manipulated, the changes are input into the constraint graph and the constraint graph propagates them and redraws the bean. The rectangle constraint graph is contained in the *ConstraintRect* class which allows a graphics object to be associated with the constraint graph and updates the graphics object to reflect changes in the constraint graph.

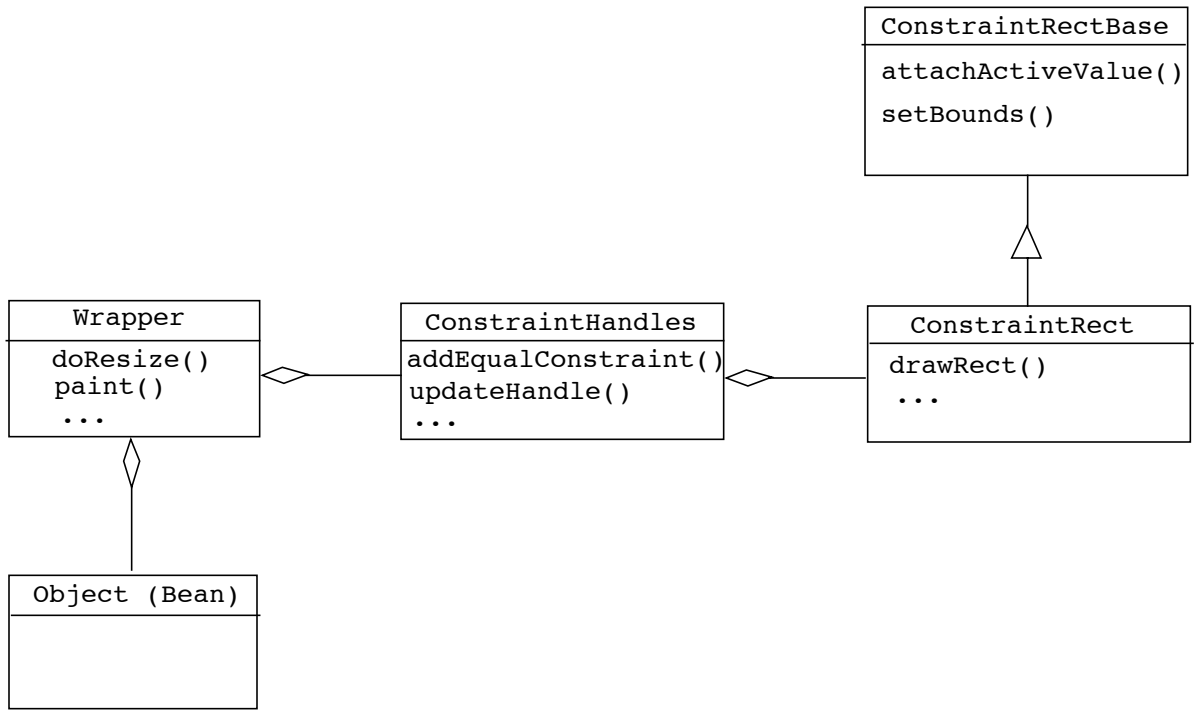


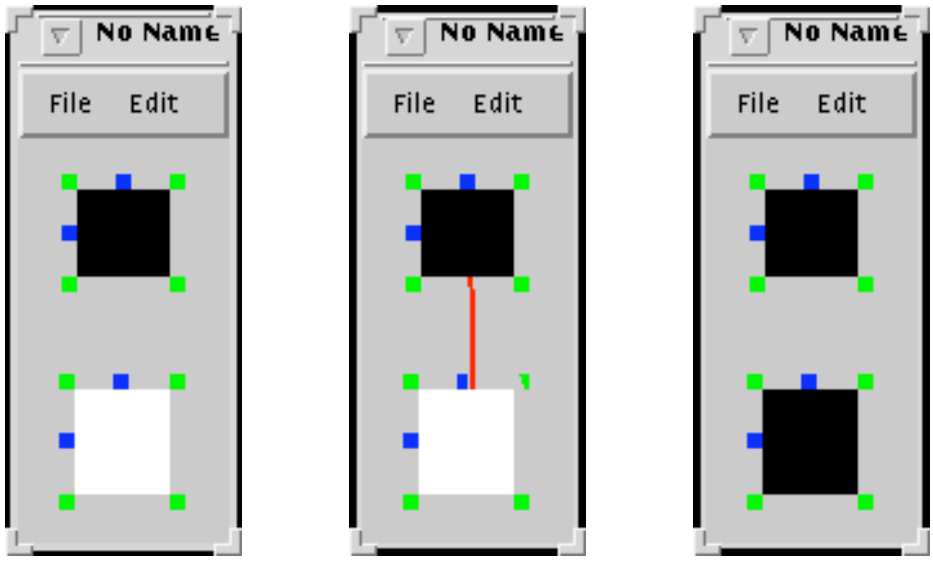
Figure 6: OMT Diagram of the Handles Portion of the BeanBox

4.2.2 Property Constraints

The addition of property constraints allow users to graphically specify constraint relationships between bean properties. To specify a property constraint, the user selects the source object and clicks on the menu item "constrain property." A dialog box pops up that allows the user to specify the property to be constrained. A line then connects the cursor and the source bean. The user drags the mouse over the target bean and clicks the mouse button. Another dialog box pops up that allows the user to specify the target property to be constrained. This process creates an equality con-

[1] In OMT notation, a line with a diamond at the base represents a "has a" relationship. In this example, a Wrapper has an instance of a ConstraintHandles class. A triangular arrow represents an inheritance relationship between two classes, where the arrow points up the inheritance tree.

straint between the two properties. Figure 7 shows how a user can constrain a "background color" property for two beans. Of course, bean properties need not be graphical. The BeanBox only allows properties of the same type (e.g., ints, Squares, etc.) to be constrained by only presenting choices for the second property that are of the same type as the first.



(a) (b) (c)

Figure 7: Example of Constraining Bean Properties

The property constraint mechanism does have one important limitation. Only bound properties (see Section 2.1) may be constrained because the underlying constraint graph must have some way of being notified of a change to a property. Therefore, if a property is not bound, our BeanBox will not allow the property to be used in a constraint. In practice, it is not always possible to know if a particular property in a bean is constrained (as was mentioned in Section 2.1). Therefore, if some, but not all properties of a bean are bound, it is the responsibility of the user to only create constraints between bound properties. Ideally, this should not be the concern of the user, but there is no guaranteed means of automatically determining this information. This problem is discussed in more detail later.

The use of the JavaBeans event registration model (event sources and listeners) further complicates the constraint mechanism. An example best illustrates this problem. Consider again the task of constraining the "background color" property of two beans (A and B) as depicted in Figure 7. Figure 8 shows how the change would occur in the constraint graph. Circles represent variables, lines represent constraints and arrows represent the propagation direction for the graph. (a) shows the

initial configuration where variables A and B are constrained to be equal. When a change to the value of A occurs, the variable A is set editable so that changes propagate away from A, as shown in (b). However, when the change from A propagates to B, B's setter method is called, which results in a property change event being generated. The property change event results in B being set editable, as shown in (c). However, we now have a conflict because A is already editable and we receive an error because B cannot be changed.

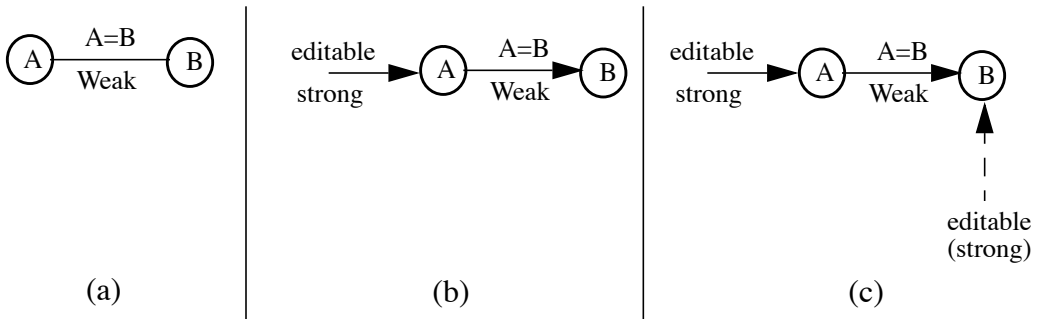


Figure 8: Problems with Propagating Value Changes in a Constraint Graph

The problem is further complicated by the possibility of multiple threads simultaneously changing properties, which could result in multiple threads accessing a constraint graph at the same time.

Luckily, the solution to the problem is relatively simple. We have added a thread-safe queue to the BeanBox that queues property change events. Instead of propagating changes to properties in a depth-first manner, we queue each property change. A separate consumer thread then reads each event out of the event queue and propagates any changes in the constraint graph. This producer-consumer model clearly solves the thread synchronization problem by only allowing a single thread (the event consumer thread) to access the constraint graph at a time. The event queue also only permits one change to be propagated through the constraint graph at a time.

Figure 9 shows the OMT diagram for the property constraint support. Each wrapper has a *ConstraintSupport* class that allows constraints to be added to each of the bean's properties. The ConstraintSupport class maintains a hash table containing constraint information for each bean property. The class is a listener for the bean's properties and if a bean property changes, the ConstraintSupport class checks its hash table to see if the property has any associated constraints. If it does, the change is propagated through the constraint graph.

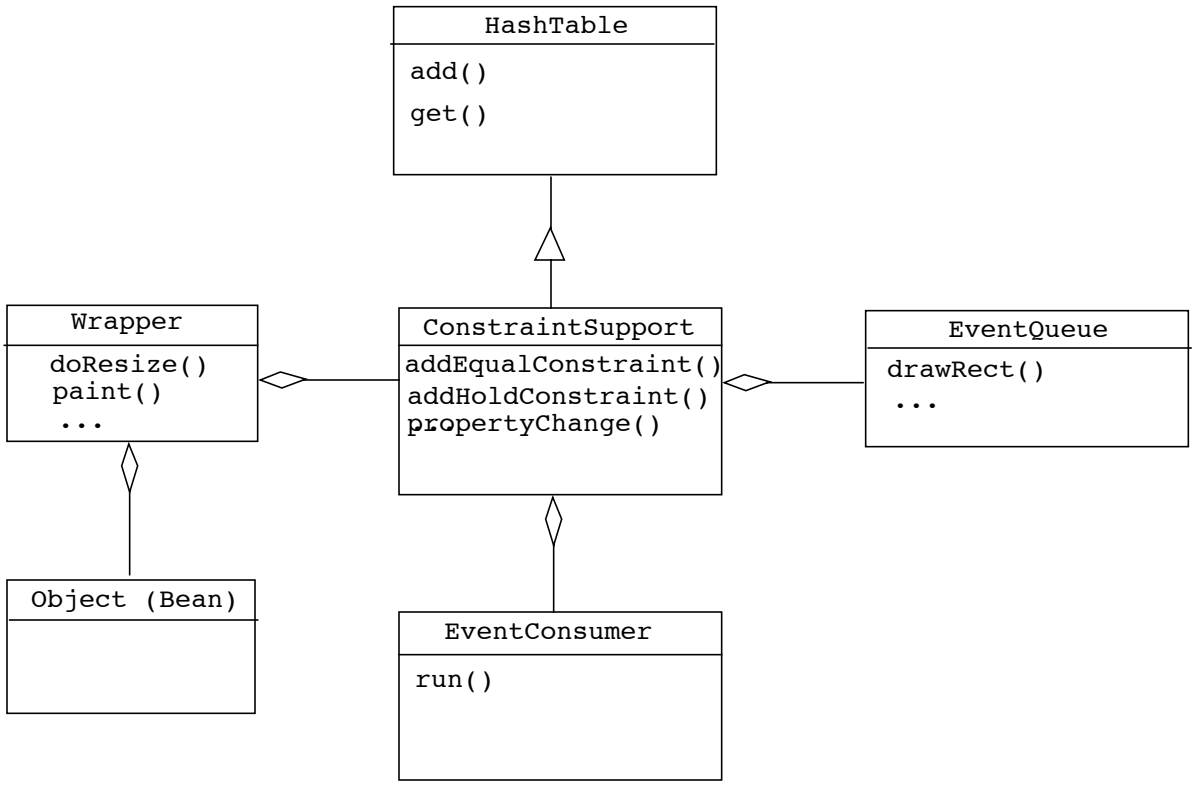


Figure 9: OMT Diagram for Property Constraint Portion of BeanBox

Currently, the property constraint mechanism supports both equality constraints and hold constraints -- constraints that specify that a property's value should remain constant. However, other types of constraints can easily be added to the existing framework.

5. Java Limitations Discovered

While integrating constraint support with Sun's BeanBox, we discovered several limitations inherent in both the Java Abstract Windowing Toolkit (AWT) and the JavaBeans API. This section discusses some of these problems and possible solutions.

5.1 Java AWT

The Java AWT provides API's that allow programmers to create simple graphical interfaces that work on all Java-supported systems. The basis of the Java AWT is the *Component* class, which dictates how graphical objects are manipulated (e.g., resized, moved, colored). In order to provide cross-platform support, the Component interface delegates to *Peer* classes that are mappings of

general operations onto windowing-system specific operations. For example, the AWT provides an interface for creating windows. However, the actual work of creating a window is delegated to the window-system specific peer. Therefore, a window created on a Microsoft Windows system will have the default appearance and behavior for that windowing system. A window on a Motif windowing system will have its default appearance and behavior, and so on.

Although the use of peer classes is useful in providing cross-platform support, we found that it also hindered our efforts by not allowing the programmer access to lower-level graphics operations. In particular, when propagating a value change through a constraint graph, we would like to first propagate the change and then synchronously update the screen so that all changes appear to occur simultaneously. This effect is particularly important when animating objects to give the changes a smooth appearance.

Unfortunately, the AWT does not allow the programmer to control when a particular component refreshes. For example, if we wish to set the background color of two objects at the same time, we cannot do so using the AWT, because the "setBackground()" call delegates to the peer's "setBackground()" call, which we have no control over.

The Java Foundation Classes (JFC) that will be included in the next official release of the Java Developer's kit promise to provide a much more flexible and customizable toolkit abstraction by eliminating the requirement of peer components. One of the primary features of the JFC is the addition of "pluggable look and feel," which allows users to specify their own appearance for windows, dialogs, etc., which are consistent across platforms.

5.2 Bound and Constrained Properties in the JavaBeans Model

The JavaBeans component model provides a very general and extensible way to monitor and hook-up property change events. However, generality is not without a price. In this project we discovered two shortcomings of the JavaBeans model.

We call the first problem the "echo" problem. If we are stranded in a cave and are shouting, hoping that someone will hear us, we don't want to mistake our own echo for another person and falsely conclude a rescuer is coming. In the JavaBeans model, a similar problem is present with the property change model. When a property change event is "fired" in the setter method of a bean, it is not possible to know the object that invoked the setter method. For example, if an object is a listener

for a property that changes on a particular bean and the object invokes a setter method on that bean, it receives the property change event that it causes, but has no way of knowing what object originated the change. The Java 1.1 Beans API provides a "PropagationId" field in the PropertyChangeEvent class, but it is currently not used.

A second problem with the JavaBeans model, which was mentioned in Section 2.1, is the inability to determine whether a given property in a bean is bound. The bean developer may choose to include a BeanInfo class along with each bean that describes which properties are bound, etc. However, there is no requirement that the developer do so. Therefore, a BeanBox can assume very little abound bean properties, making it very difficult to take advantage of bound properties. In the BeanBox presented here, we ignore this issue. If a bean contains "addPropertyChangeListener" and "removePropertyChangeListener" methods, it is assumed that all properties are bound.

6. Conclusion and Future Work

This paper presents a simple framework and implementation for integrating a constraint mechanism with the Java Beans and Java Event models. With the existing framework, numerous additional features can easily be added. This section briefly outlines potential topics for future work.

6.1 Constraint Adapter

Because the BeanBox uses two separate constraint mechanisms, it is not possible to create constraints between handles and bean properties. A constraint adapter would allow such interaction to take place by providing proxy properties for each handle that would act like bean properties, allowing the two mechanisms to communicate.

6.2 Layout Manager

A constraint-based layout manager would allow users to use simple constraint statements to specify complex, dynamic layouts. The *LayoutManager2* class in the Java AWT provides an interface that allows the user to specify constraint information when adding a component to the container. In a layout manager it is still not possible to control when the screen refreshes (see Section 5.1). However, the problem is not as important in the context of a layout manager because a container's layout generally does not change frequently.

Work Cited

[1] Alan Borning and Bjorn Freeman-Benson, “Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics”, CONSTRAINTS: An International Journal, Special Issue on Constraints, Graphics, and Visualization, Vol. 3 No. 1, April 1998, pages 9-32.

[2] Bjorn Freeman-Benson, John Maloney, Alan Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54-63, 1990.

[3] Brad Vander Zanden ``An Incremental Algorithm for Satisfying Hierarchies of Multi-way, Dataflow Constraints'', University of Tennessee Technical Report CS-95-282, March 1995.

[4] Graham Hamilton (ed). “The JavaBeans 1.0.1 Specification.” http://java.sun.com/beans/docs/spec.html. Sun Microsystems. July, 1997.

[5] I. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the Spring Joint Computer Conference*, pages 329-345, IFIPS 1963.

[6] “JavaBeans: The Only Component Architecture for Java.” http://java.sun.com/beans/. Sun Microsystems, 1997.

[7] “Java Studio.” http://www.sun.com/studio/. Sun Microsystems, 1998.

[8] “JBuilder Home Page.” http://www.inprise.com/jbuilder/. Inprise Corporation, 1998.

[9] T. Paul McCartney, Kenneth J. Goldman, David E. Saff. EUPHORIA: End-User Construction of Direct Manipulation User Interfaces for Distributed Applications. Software Concepts and Tools, 16(4):147-159, December 1995.

[10] T. Paul McCartney. User Interface Applications of a Multi-way Constraint Solver. Washington University Department of Computer Science WUCS-95-22, October 1995.