

TECHNICAL RESEARCH REPORT

Using Interaction Object Graphs to Specify and Develop Graphical Widgets

*by D.A. Carr, N. Jog, H.P. Kumar,
M. Teittinen, and C. Ahlberg*

T.R. 94-69



*Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry*

CAR-TR-734
ISR-TR-94-69
CS-TR-3344

Sept 1994

USING INTERACTION OBJECT GRAPHS TO SPECIFY AND DEVELOP GRAPHICAL WIDGETS

David A. Carr¹, Ninad Jog², Harsha Prem Kumar, Marko Teittinen,
and Christopher Ahlberg³
Human-Computer Interaction Laboratory,
Department of Computer Science,
& Institute for Systems Research
University of Maryland, College Park, MD 20742

© Copyright 1994, David A. Carr, Ninad Jog, Harsha Prem Kumar,
Marko Teittinen, and Christopher Alberg.
All rights reserved.

email: davecarr@cs.umd.edu, ninad@cs.umd.edu,
harsha@src.umd.edu, marko@cs.umd.edu, &
ahlberg@cs.chalmers.se

- ¹ also RMS, Inc., Code 520.9, NASA Goddard Space Flight Center, Greenbelt, MD 20771.
- ² Dept. of Electrical Engineering, University of Maryland
- ³ Dept. of Computer Science, Chalmers Univ. of Technology, S-412 96 Göteborg, Sweden

ABSTRACT

This document describes five widgets that have been developed at the Human-Computer Interaction Laboratory of the University of Maryland. These widgets are: a range selection slider, a two-level alpha-slider, a secure switch, a tree viewer, and a treemap viewer. The last two use the same tree representation and can be used as alternate visualizations of the same hierarchy. In addition, a system for widget specification is introduced and each widget is specified using this system.

Note: The application program interface (API) is defined in an appendix to this report. This appendix is not normally included with the report and must be requested.



The 11-year old Human-Computer Interaction Laboratory (HCIL) is an interdisciplinary effort within the Center for Automation Research. The main participants are faculty, staff, and students from the Department of Computer Science, Department of Psychology, and College of Library and Information Services at the University of Maryland, College Park, MD.

For single copies
or a list of all HCIL
technical reports
please write to:
Teresa Casey
Human-Computer
Interaction Laboratory
A.V. Williams Building
University of Maryland
College Park MD 20742
email: tcasey@cs.umd.edu

USING INTERACTION OBJECT GRAPHS TO SPECIFY AND DEVELOP GRAPHICAL WIDGETS

David A. Carr
University of Maryland Department of Computer Science,
University of Maryland Human-Computer Interaction Laboratory, and
RMS, Inc., Code 520.9
NASA Goddard Space Flight Center
Greenbelt, MD 20771
davecarr@cs.umd.edu

Ninad Jog
Department of Electrical Engineering &
Human-Computer Interaction Laboratory
University of Maryland
ninad@cs.umd.edu

Harsha Prem Kumar
Institute for Systems Research &
Human-Computer Interaction Laboratory
University of Maryland
harsha@src.umd.edu

Marko Teittinen
Department of Computer Science,
Human-Computer Interaction Laboratory &
Institute for Systems Research
University of Maryland
marko@cs.umd.edu

Christopher Ahlberg
Department of Computer Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden
ahlberg@cs.chalmers.se

ABSTRACT

This paper describes a new system for specifying widgets and the University of Maryland Widget Library™. For each widget there is a brief description of its operation and a specification. Each of the five widgets has been developed at the Human-Computer Interaction Laboratory of the University of Maryland and implemented using the Galaxy™ platform independent User Interface Development System. These widgets are: a range selection slider, the AlphaSlider, a tree viewer, a treemap viewer, and a secure switch.

1 INTRODUCTION

The University of Maryland Human-Computer Interaction Laboratory (HCIL) has designed and developed a number of new widgets. These widgets were generally developed as part of a prototype application. However, recent developments in platform independent User Interface Development Systems (UIDS) has made it possible to build these widgets for all major computer systems. Therefore, we decided to implement our more recent widgets using one such system, Galaxy™. These widgets include:

1. A range selection slider that allows the user to set a range of values in the display space needed for a conventional slider.
2. An AlphaSlider that allows the user to select one item from a list of thousands with minimal display space.
3. A node-link tree viewer that along with the treemap viewer presents two different views of the same data and share a common data structure.
4. A treemap viewer that uses a space filling algorithm to reduce the display space required to view a hierarchy.
5. A secure toggle switch that requires the user to drag the switch between positions to reduce the chance of accidental operation.

An immediate need of this project was a method to communicate the new widget design. We wanted a system which was compact and formal. While not immediately required, we desired that this method could be the basis for rapid prototyping. In order to fulfill these needs the Interaction Object Graph was developed.

2 THE IOG SPECIFICATION METHOD

Specification of user interfaces has been used to aid in the design of user-computer dialog and software. This work has led to the development of User Interface Management Systems or UIMSs. These systems significantly reduce the work required to design and specify a user-computer dialog. They also allow non-programmers to prototype and design complex user interfaces. However, current UIMSs assume that a set of or widgets exist and manipulate the presentation of these widgets. If none of the interaction objects are quite what the designer wants, then either the designer must compromise and redesign the dialog with the interaction objects provided, or the desired dialog must be coded in a programming language and integrated into the UIMS. This section presents the Interaction Object Graph (IOG) as an approach to this widget building problem. IOGs can be used to specify interaction objects at a higher level than programming languages. In succeeding sections IOGs have been used to specify the University of Maryland widgets.

2.1 Previous Research

Over the years a number of methods have been used to specify user interfaces. These include grammars, algebraic specifications, task description languages, transition diagrams, statecharts, interface representation graphs, rule-based systems and by demonstration.

Shneiderman's [1982] multiparty grammars are an example of a grammar based specification. A multiparty grammar divides non-terminals into three classes: user-input, computer, and mixed. User-input and computer non-terminals represent user actions and computer responses, respectively. Mixed non-terminals represent sequences in the human-computer dialogs. Multiparty grammars are good for modeling keyboard-based command language interactions, but are very awkward for direct-manipulation interfaces. Multiparty grammars do not model the inherently non-sequential nature of direct manipulation well. For example, consider a direct-manipulation interface with two forms. The user can, and probably will, interleave completing the forms. In order to specify such an interface, the notation must support some way of describing

interleaved action. While this is possible with a grammar the number of productions tends to increase as a product of the number of widgets. Thus, the number of productions quickly becomes unmanageable.

Algebraic specification of window systems was introduced by Guttag and Horning [1980]. They proposed the design of a windowing system based on axiomatic specification of abstract data types. This method permits formally proving properties of the user interface. However, algebraic specifications have serious drawbacks. They are very difficult to read and require considerable time and training to understand. They are even more difficult to write. This makes them unsuited for communicating interface behavior.

Task description languages concentrate on describing user actions. They were originally developed to model user performance and most do not have any provision for describing system actions. Siochi and Hartson's User Action Notation (UAN) [1989; Hartson and Gray, 1992] is one language which also makes a contribution by specifying computer feedback and interface internal state. Another paper [Hartson et al., 1990] corrected a major shortcoming by adding a link to the application computation. However, UAN concentrates heavily on describing user actions and is not well adapted to describing software state. In addition, UAN suffers from its task description heritage. Traditionally, these languages have concentrated on specifying error-free behavior. Specifying system responses to unexpected user actions is awkward. Often the designer must explicitly consider every possible user input sequence and write an explicit system response. Unlike multiparty grammars, UAN does not have a symbol which matches all unspecified sequences. Finally, UAN's tabular notation does not readily show relationships between tasks.

Another approach to modeling user interfaces is the transition diagram [Wasserman, 1985]. In this approach the transitions represent user inputs and the nodes represent states of the interface. Computer outputs are specified as either annotations to the state or transitions. However, transition diagrams suffer from a combinatorial explosion in the number of states and transitions as system complexity increases. Jacob [1986] solved part of this problem by allowing concurrent states to coexist as parallel machines or co-routines. Co-routines did not completely solve the transition complexity problems for specifying interfaces with modal dialog boxes.

Harel's [1988; Wellner, 1989] statecharts were designed as a formal solution to the combinatorial problems with transition diagrams. The statechart adds the concept of a meta-state. Meta-states group together sets of states with common transitions that are inherited by all states enclosed in the meta-state. Since meta-states may enclose other meta-states, a complete inheritance hierarchy is supported. A special history state is supported to return the meta-state to its previous status on return transitions from events such as invoking help. Meta-states are divided into two types: parallel or AND-states and sequential or XOR-states. Meta-states enclosed within AND-states may execute in parallel and fulfill the function of co-routines. As originally defined, statecharts do not incorporate data flow or abstraction.

Interface Representation Graphs (IRGs) were used by Rouff [1991; Rouff and Horowitz, 1991] as the underlying representation for their Rapid Program Prototyper. IRGs extend the statechart to represent dialog. This extensions are: IRG nodes represent a physical or logical component of the interface as well as a state. Data flow as well as control flow can be specified in an IRG. IRGs support inheritance of interface objects, data flow, control flow, and attributes. Constraints on data and control flow are supported. Finally to support UIMS functionality, IRGs permit specification of semantic feedback between the application and the user interface.

However, the IRG specifies interaction between "widgets" and is not designed to specify new ones.

The User Interface Design Environment (UIDE) uses pre and post conditions to control user interface dialog [Gieskens and Foley, 1992]. Each widget in the user interface has two preconditions associated with it. One specifies when it is visible and the other specifies when it is active. Each functionally separate action of the widget has a post condition associated with it. As the user operates the interface the post conditions modify a blackboard representation of the user interface state. While expressed in a different style than state diagrams, the pre-condition/post-condition notation can be mapped into a state diagram. (Take the power set of all possible conditions which may be posted to the blackboard and use the pre-condition/post-condition pairs as the transition function.) The UIDE system does not provide for definition of new interaction objects. Both Olsen's PPS [1990; 1992] and Hill's Sassafra [1986] use rule based systems which are similar to the pre-condition/post-condition method.

A final technique used for specifying human-computer dialog is to do so by demonstration. Using this technique the designer places widgets on the screen and the system makes inferences about the intended design. Druid [Singh et al., 1990] and Peridot [Myers, 1988] are two systems which build user interfaces by demonstration. Druid provides for interactive layout and dialog definition. However, its dialog model does not include constraints between interface objects or the modification of interface objects. Peridot is a similar system which includes constraints and modification by redrawing. Myers states that Peridot has two limitations which would require more research. First, it does not support text input. Second, specifying operation sequences requires more research.

The above specification methods concentrate on describing interface behavior. However, interface layout and spatial relations between objects are also important. One approach to layout is simply to draw the interface. This method does not do very well when run-time re-sizing is allowed. Recent research has settled on constraint grammars [Vander Zanden, 1989] to solve this problem. Hudson's [1989] Apogee UIMS has a particularly clever method for setting layout constraints graphically. GalaxyTM [Visix, 1994] uses a springs and struts model for layout constraints.

2.2 Interaction Object Graphs

Interaction Object Graphs (IOGs) [Carr, 1994] are designed to add widget specification to Interface Representation Graphs. They combine the data flow and constraint specifications of IRGs with the statechart transition diagram execution model. This expands the statechart to show data relationships as well as control flow. It also permits specification of low level interaction objects which cannot be specified by Interface Representation Graphs. Below is a brief description of the IOG state diagram and a transition description language used to specify transition conditions.

The IOG state diagram traces its lineage from UAN, statecharts, and IRGs. Statecharts added four new state types to the traditional state diagram. These states are used in IOGs. They are: the XOR meta-state, the AND meta-state, and two types of history state.

The meta-states can contain both normal states and other meta-states. Transitions from meta-states are inherited by all contained states. This helps reduce the problem of arc explosion. The XOR meta-state contains a sequential transition network. Only one state inside of an XOR meta-

state can be active at one time. On the other hand, an AND meta-state contains more than one transition network. Each of these networks executes in parallel.

A history state can only be contained in an XOR meta-state. Whenever a transition transfers control from a meta-state, the history state remembers which state was active immediately before the transition. If a later transition returns control to the history state, the meta-state is returned to the remembered status. History states help control state explosion. To see this, consider a specification of a help system which is independent of the user interface. An ordinary transition network would require replicating the help system specification once for every state in the user interface. Otherwise, there would be no way to return to the user interface state that was active before help was requested. A statechart history state could receive the return transition from the help system and only one copy would be required. There are two types of history states. They differ in how they treat a return when the last active state was a meta-state. The **H** state restarts meta-states at their start state and provides one level of history. On the other hand, the **H*** state restarts meta-states at their history state when they have one, thus, allowing multilevel history. Figure 1 shows the representation of the new states.

IOGs add two additional state types to the statechart, data objects and display states. Data objects were present in IRGs. However, their meaning is slightly different in IOGs. Data objects are represented as parallelograms. (Figure 1.) In IOGs they represent the storage of a data item and control is never passed to them. They can only be destinations for the constraint and data arcs discussed below. Display states are control states that have a change in the display associated with them. In IOG diagrams a picture of the display change is used whenever possible instead of a program-like statement such as "draw(ActiveON)".

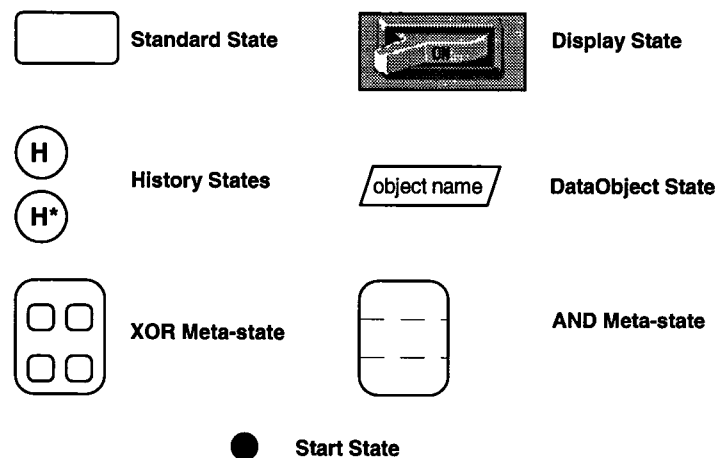


Figure 1 -- IOG state symbols.

IOGs also add two special arc types: the event arc and the data arc. Events allow the designer to define "messages" which may be lacking in the underlying specification model. For example when specifying the trash can in the Macintosh interface, one needs to know when a file is being dragged over it as opposed to when the pointer is being dragged over it. One way to do this would be for the file icon to generate a "dragging started" event and a "dropped" event. The trash can would then be highlighted whenever the pointer was over it between a "dragging started" event and a "dropped" event. An event is represented by a special transition passing through an **E** in a diamond. (Figure 2).

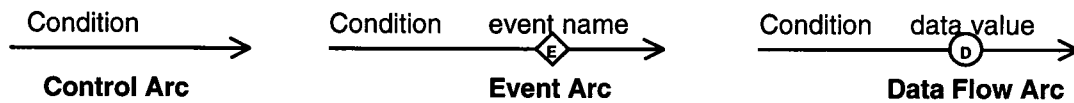


Figure 2 -- IOG arc symbols.

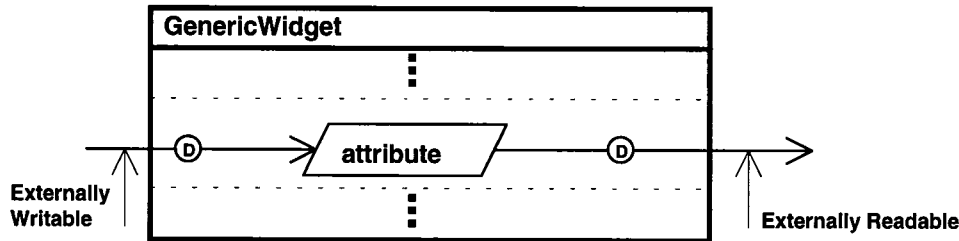


Figure 3 -- A readable and writable widget attribute in an IOG.

Data flow is represented in a manner similar to events – an arc passing through a **D** in a circle (Figure 2). A data flow arc may have any state as a source and can only terminate at a data object or have an unspecified termination. In addition, at least one end must be attached to a data object. Data flow arcs with data objects as a source, whose destination arrow is unspecified, and whose destination is outside of the containing interaction object, indicate exported data (Figure 3). This data may be used by the application or attached to other user interface components as a more complete specification is constructed. Data flow arcs with data objects as destinations represent updating the data object. If the arc's source is a control state, it represents a change in the value when the arc conditional is satisfied. In this case, the data flow arc is labeled with the new object value. An arc without a source represents a possible external update of the object.

Constraints are useful in specifying one attribute of the user interface in terms of others. With constraints it is simple to restrict an icon to be contained within a window or to map the values of a slider to a specific range. IOG data arcs support a form of one-way constraints by expressing the data value as an equation in terms of other attributes. Together with the condition on the data arc, these equations provide a means to constrain one attribute in terms of another with a Boolean guard.

2.3 IOG Transition Descriptions

In order to describe the transitions between states, an abstract model of the user interface and a description language for that model are required. IOGs abstract the interface into the following objects: Booleans, numbers, strings, points, regions, icons, view ports, windows, and user inputs. A brief description of these objects follows.

Booleans, numbers, and strings (BNS) are the usual abstractions with the usual operations. It should be noted that numbers contain both the real and integer data types. In addition, any of them may be converted into an icon representation by the operator **icon(BNS, point)** or **icon(BNS, region)**. Both operators convert the boolean, number, or string **BNS** to a text representation and convert it into a picture. If specified with a point, the resulting icon is as big as it needs to be to hold **BNS**. If a region is specified, the icon is the size of the region.

Points are an ordered pair of two numbers (x,y). Points have the algebraic operators which are normally associated with them. A point may be assigned a value by writing **p=(x,y)**. In addition, **p.x** and **p.y** represent the x and y coordinates from the point **p**.

A region is a set of display points defined relative to an origin called the **location**. The location of the region is always the point (minx,miny) where minx and miny are the smallest x and y coordinates in the region. Regions have a **size** operator which returns a point giving the height and width of the smallest rectangle which covers the region. Regions also have an **in** operator which tests if a point is in the region. This is written **Region.in(pt)** and returns a Boolean value. Although regions are not restricted to be rectangular, rectangles are the most commonly used. Note, a region cannot be visible on the display. There is no drawing operation associated with a region.

Icons are regions with pictures. That is some points in the region have a color number attached to them and are shown on the display. Icons add the operations **draw** and **erase**. In addition, if the origin of the icon is changed, there is an implicit **erase-draw** operation sequence. Unless otherwise specified the region associated with an icon is a rectangle.

A view port is a region with an associated mapping function for some underlying application data. The mapping would be in two parts: conversion to a world coordinate system and graphics representation; and projection onto the display. For example, text would first be converted from ASCII to a font representation and a location on a page. The page would then be projected onto the display. The mapping is controlled by a projection function (**proj**), a translation point (**translate**), and a scale change point (**scale**). If **convert** is the conversion function for some object in some view port, then the function **translate + proj(scale, (convert(object)))** would be the view port mapping. Parts of objects projected to points not in the region are not displayed and objects in view ports are addressed relative to the view port location.

Windows group the above objects together. They add a level attribute which determines window stacking relative to other windows. They can be viewed as view ports containing only objects already mapped to display coordinates. A window with a lower level obscures an overlapping window with a higher level.

Objects are addressed in the specification using a dot notation. For example, "win.icon1.location.x" would be the x coordinate of the location of icon "icon1" in window "win".

User inputs are mapped to numbers, points, and Boolean variables. Keyboard input events are represented by quoted strings ("quit-**J**" when the word quit is typed and followed by a carriage return) or key events similar to those in UAN (**LShiftv** for left shift key pressed). The mouse is mapped into a point for location (**M@**), a point for relative change (**MΔ**), a Boolean indicating it moved (**ΔM**), button change events (**Mv**, **M^**, **M2v**, ...), and button status variables (**Mdn**, **Mup**, **M2dn**, ...). Since the value of the mouse location is tested frequently, **in[Region]** is written as a shorthand for **Region.in(M@)**. The special notations **~[Region]** and **[Region]~** mean the event of the mouse entering and leaving the **Region**.

2.4 Example Widget Specification (Dragable* Icon)

In order to get a better idea how an IOG specification is constructed and interpreted, a simple basic widget, a draggable icon, will be specified and explained. The draggable icon is a basic building block for later widgets such as the AlphaSlider and the range selector.

Now, let's begin the draggable icon example. The draggable icon widget moves in both the horizontal and vertical directions. It returns a value which is related to its location. To operate it the user places the mouse over the icon and presses the button. The icon then follows the mouse as long as the mouse stays within a predefined region, called **FreeArea** (Figure 5). Once the mouse leaves **FreeArea** the icon stops following the mouse. If the mouse is moved outside of a second larger region, called **ActiveArea**, the icon jumps back to its original position when the mouse button is released. In order that the user may distinguish between the tracking and non-tracking states of the widget there are two icons **Active** and **Idle** associated with the widget (Figure 4).



Figure 4 -- Example icons for a draggable icon widget.

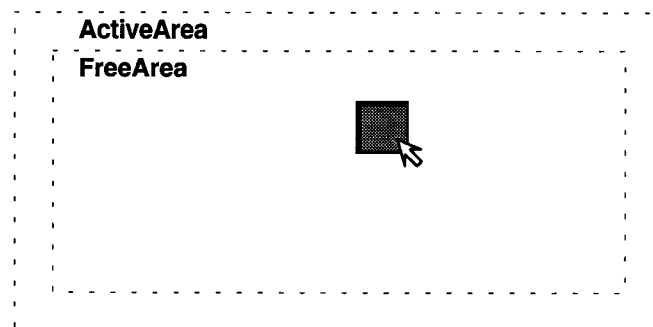


Figure 5 -- Example layout for **ActiveArea** and **FreeArea**.

The behavior of a draggable icon depends on the value of four attributes. Two of these, **location** and **old_value**, are only accessible within the widget. One, **selected**, is read only and the last, **value**, is both readable and writable. The attribute **location** is a point and represents the display coordinates of the icon. The programmer is not allowed to set this attribute because **value** determines the location and it is writable. The **old_value** attribute stores **value** so that it may be restored if the user cancels the action by lifting the mouse button outside of **ActiveArea**. The **selected** attribute indicates whether or not the user is operating the draggable icon.

The IOG specification for a draggable icon is shown in Figure 6. The widget specification starts with an AND meta-state named DragIcon which holds the four data attributes and an XOR meta-state which contains the behavior. The behavior state should be interpreted as follows. When started the widget displays the **Idle** icon. Moving the mouse into the region defined by the **Idle**

* A word coined to mean an icon that follows the mouse whenever the cursor is within the icon and then the left button is pressed.

icon and pressing the mouse button causes **value** to be saved in **old_value**, sets **selected** TRUE, and displays the **Active** icon (via the "Mv && in[Idle]" arc). While in this state and in the **FreeArea** region, the **cDrag** constraint is active. (This constraint relates the mouse position to the **value** of the draggable icon.) The **cChg** constraint between **value** and **location** is always active. Releasing the mouse button will cause **selected** to be set to FALSE and redisplay the **Idle** icon (via the "M^" arc). Moving the mouse outside the **ActiveArea** region will transfer to the "// cancel?" state (via the "[ActiveArea]~" arc). Releasing the mouse button in this state will return the icon to its original position, update **selected** to FALSE, and display the **Idle** icon (via the "M^" arc). Moving back into **ActiveArea** re-activates the icon tracking the mouse (via the "~[ActiveArea]" arc).

All that remains is to specify the values of the two constraints **cDrag** and **cChg**. These values are application specific. If there was to be a one-to-one correspondence between pixels in **FreeArea** and **value** with **value** ranging between (0,0) and **FreeArea.size** less the icon size, then the following relationships would hold.

cDrag: { value = min(max(value + MΔ - FreeArea.location, point(0,0)), FreeArea.size - Idle.size); }

cChg: { location = value + FreeArea.location; }

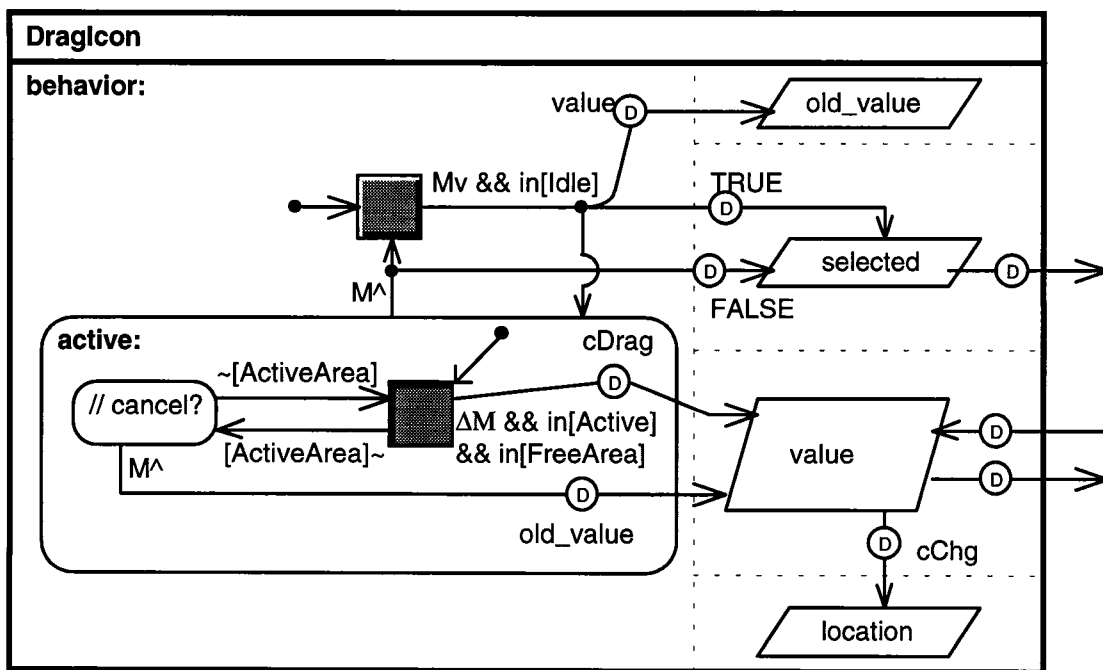


Figure 6 -- Specification of a draggable icon.

It should be noted that both a horizontal and a vertical slider indicator are simple modifications of the draggable icon. In order to convert the draggable icon into a horizontal slider indicator one must make the following changes:

1. The type of **value** changes from Point to Number.
2. Two new attribute numbers are added: **min** and **max**, represent the range of slider values. For convenience an internal attribute **range** is added and constrained to be (**max** - **min**).

3. **cDrag** is rewritten to be dependent on the x-value of the mouse and the slider range. So, **cDrag** becomes: $\text{value} = \text{min} + \text{range} * (\text{M.x} - \text{FreeArea.location.x}) / \text{FreeArea.size.x}$
4. The **cChg** constraint changes so that only the x-coordinate of the location is variable.
5. The **FreeArea** and **ActiveArea** are defined in relation to the slider boundary. (Figure 7.)

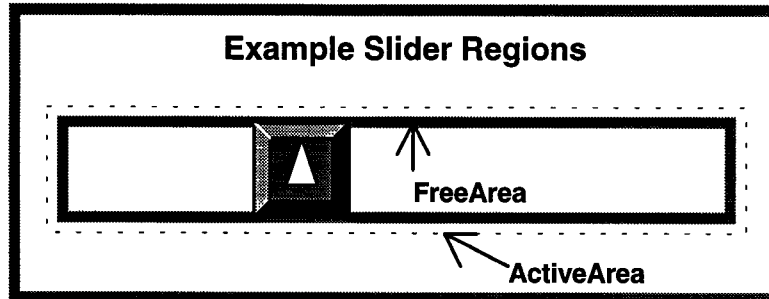


Figure 7 -- Example definition of **FreeArea** and **ActiveArea** for horizontal slider.

3 THE UNIVERSITY OF MARYLAND WIDGET LIBRARY™

The following sections describe each widget in the University of Maryland Widget Library™. An IOG specification and operational description is presented for each widget.

3.1 The Range Selection Slider

Most existing sliders let the user set a single value. This makes it impossible to specify a range of values. One solution would be to use two sliders - one each for setting the lower and upper boundaries. This approach has problems, the lower boundary can be erroneously set to a higher value than the upper boundary. The range selection slider (Figure 8) prevents this. It uses two indicators that can be moved independently of each other, letting the user select a range by fixing both its lower and upper bounds [Becker et. al., 1989 & 1991]. Thus, the range selection slider allows selection of a range of values in the same space as a conventional slider. Our range selection slider was first used in the HomeFinder [Williamson and Shneiderman, 1992].

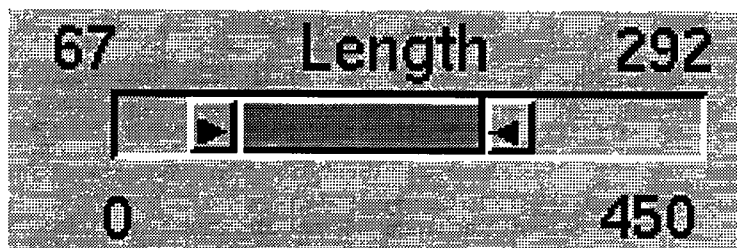


Figure 8 -- The range selection slider

Each indicator in the range selection slider operates like a single horizontal slider indicator. The only difference is that the other indicator is the boundary of the motion on one side. An indicator is activated by the user pointing at it and pressing the mouse button. The indicator will

then track the mouse as long as the mouse stays in the corresponding **FreeArea**. As long as the user doesn't leave the **ActiveArea**, the new value of the indicator will be set when the user releases the mouse button. If the user does leave the **ActiveArea**, then the indicator will return to the starting point when the user releases the mouse button (Figure 9). The white dot-dash-dot line corresponds to **FreeArea** for the high indicator and the black dashed line corresponds to **FreeArea** for the low indicator. Unlike the regions for the horizontal slider above these regions are dynamic -- moving one indicator alters the region for the other.

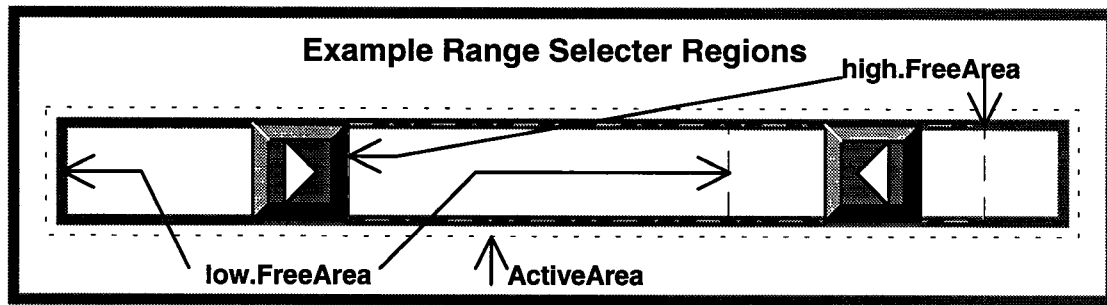


Figure 9 -- Sample definitions for **FreeArea(s)** and **ActiveArea** for the range selection slider.

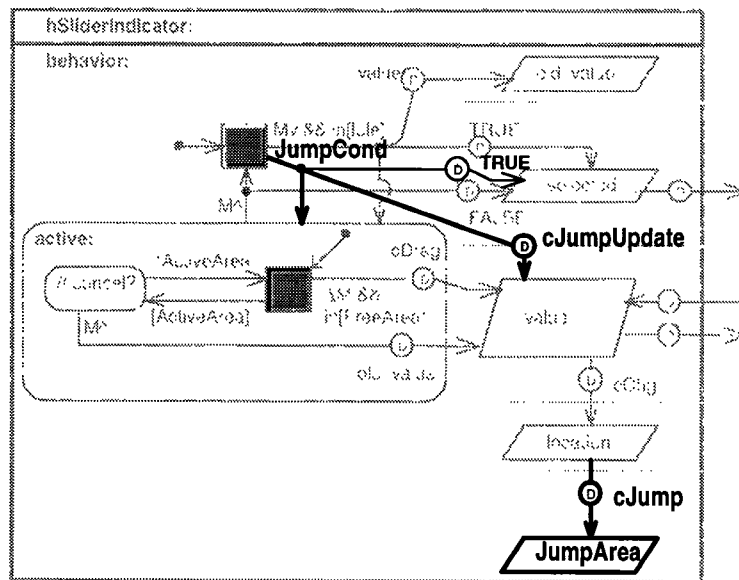


Figure 10 -- Adding "jump to value" behavior to a horizontal slider.

The range selection slider also implements three positioning short cuts. Clicking to the left of the low range indicator and in the low **FreeArea** causes that indicator to jump to the indicated value. Clicking to the right of the high range indicator and in the high **FreeArea** causes the high indicator to jump. Adding this jump behavior requires a slight modification to the horizontal slider widget already defined (Figure 10). The jump condition would be **Mv && in[FreeArea] && !in[Idle]** for a single indicator slider. However, the indicators in the range slider are more complex. The jump condition is modified to **Mv && in[JumpArea]**. Where **JumpArea** is a

rectangular region dynamically constrained by **cJump**. For the low range indicator the rectangle would be:

```
JumpArea.location = low.FreeArea.location
JumpArea.size.y = low.FreeArea.size.y
JumpArea.size.x = low.location.x - JumpArea.location.x
```

The high **JumpArea** would be similarly constrained to be to the right of the high indicator. The **cJumpUpdate** constraint would be a function relating the location of the mouse press to the slider value. This function would be different for each slider and would be an inverse of the **cChg** constraint function.

The final short cut applies to drag motions between the two indicators. This causes both indicators to move while retaining the same relative range size. The region between the two indicators is defined by the high indicator **FreeArea.location** as an upper left corner, a vertical size equal to that of the **FreeArea**, and a horizontal size equal to the difference between the x-location of the high indicator and its **FreeArea.location.x**. In the range slider this area is represented as a button. When specifying it, only the constraints on the widget are shown. A state based system like IOGs is not particularly efficient for specifying arbitrary graphs. These are hidden in the view port abstraction. The **cMidLoc** and **cMidSize** constraints in Figure 11 dynamically control the button size and location.

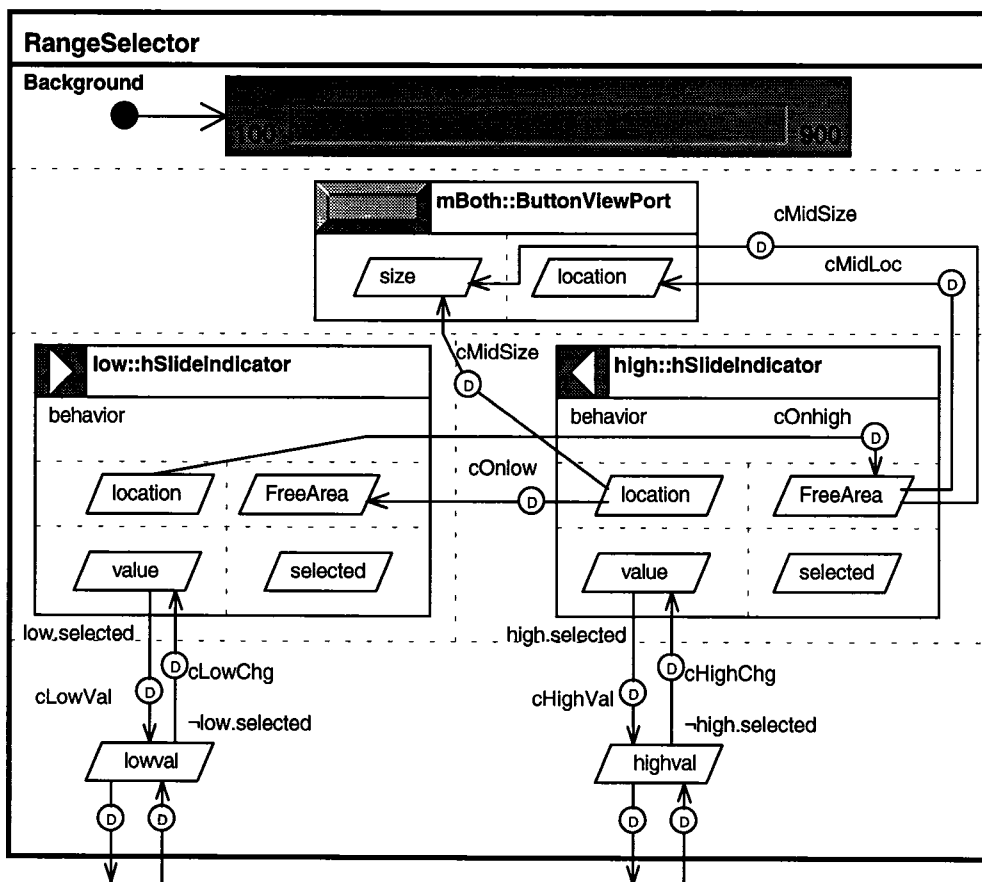


Figure 11 -- Specification of the range selection slider.

The range selection slider specification shows how earlier IOGs can be used to build a definition for a new widget. The range selector is an example of a widget built from two copies of a previously defined one. Therefore, its IOG includes two copies of the modified **hSliderIndicator** IOG. The copies are not shown in full detail to reduce the clutter in the IOG diagram. Only those parts of the **hSliderIndicator** which are relevant to the new behavior are shown. In order to adjust the control regions of each indicator, two new constraints must be introduced. These constraints, **cOnlow** and **cOnhigh** are on the **FreeArea** regions. These regions were not shown in the original specification as they did not affect behavior. One can see that composing a new widget from old ones requires specification of the relationships and coordination between them.

3.2 The AlphaSlider

The AlphaSlider (Figure 12) is a descendant of the single level alpha-slider designed by Osada, Liao, and Shneiderman [1993]. The idea is to provide rapid access to a single text item in a list. This item should be small, usually a line or two. The user slides the indicator and the system displays a text item such as a hotel name and telephone number. The axis of the slider is calibrated from A-Z instead of numerically. Osada's original design works well as long as there are not more items in the list than pixels available in the slider. However with thousands of items, there are not enough pixels in the screen. The two-level AlphaSlider was designed to overcome this limitation. It provides a coarse and fine level of selection. The coarse level provides a mapping of many items to one pixel. (e. g., for a 10,000 item list, a 300 pixel slider would provide a 33 to 1 mapping.) This lets the user rapidly select the approximate point in the slider. The fine level maps one mouse increment to one item. This lets the user select the exact item. The AlphaSlider was used in the FilmFinder [Ahlberg and Shneiderman, 1994a&b] to display the contents of five different lists of names in an area of 7 cm. by 12.5 cm.

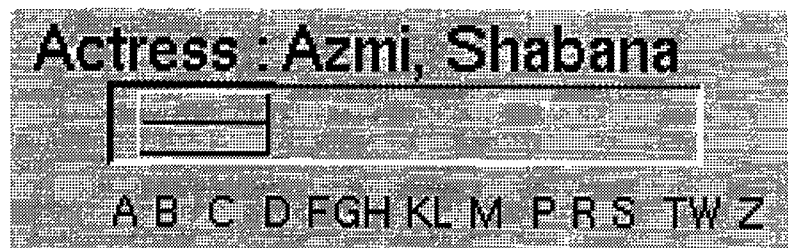


Figure 12 -- The AlphaSlider

The AlphaSlider is operated in three different ways. First, the user may make a coarse selection by clicking in the slide area. This causes the slider indicator to jump to the location of the click. Second, the user may point at and drag in the upper half of the slider indicator. This causes the AlphaSlider to enter the coarse adjustment mode. In coarse adjustment mode, each pixel movement of the slider indicator causes the list to jump in proportion to the ratio between the length of the slider and the number of items in the list. The third method of operation is fine positioning. To activate fine positioning the user points to and drags the bottom half of the slider indicator. This causes the slider to change one list item for each pixel of mouse movement.

The specification for the AlphaSlider begins with a refinement of the **hSliderIndicator** specified in Figure 10. This refinement adds two granularities to the horizontal slider indicator. We need two "active" icons to give the user feedback about which granularity is being used (Figure 13). To select each granularity we need two different activation regions to replace the region associated with the **Idle** icon in the draggable icon. These regions, **FineArea** and **CoarseArea**, determine whether fine or coarse granularity is used (Figure 14).



Figure 13 -- Example icons for a horizontal AlphaSlider.

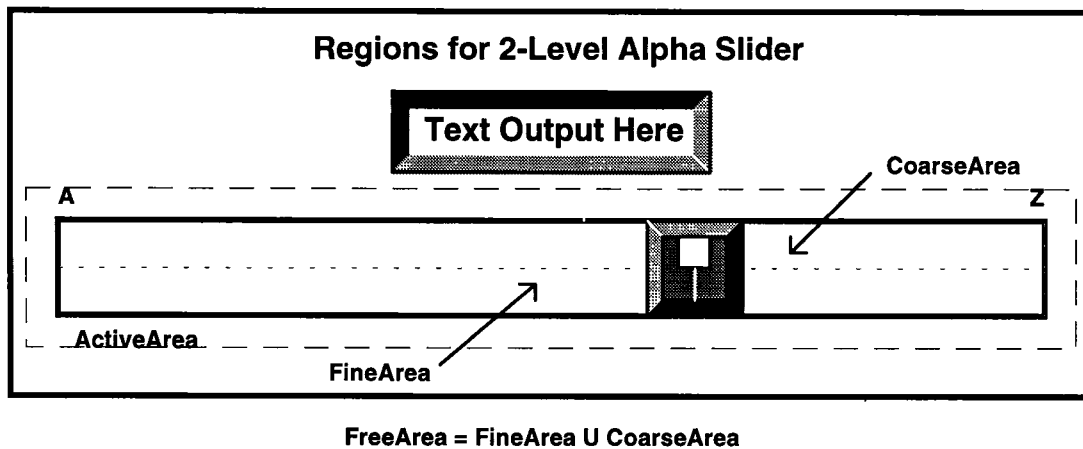


Figure 14 -- Example definition of regions for a horizontal AlphaSlider.

To complete the specification for the two granularities we need to modify how the slider behaves. We replace the single active state and the **cDrag** constraint with a XOR meta-state and two constraints. By comparing Figure 6 with Figure 15, we can see that the single **Active** display state of the draggable icon has been replaced with an XOR meta-state. This meta-state contains two display states, one for fine-grain adjustment and one for coarse-grain adjustment. The active state depends on where the mouse was pointing when the user pressed the mouse button. From each state there is a constraint (**cCoarse** and **cFine**) which specifies the relationship between mouse movement and the indicator's value. The meta state also contains a history state so that the indicator will return to the proper granularity from the "cancel?" state.

The final step in specifying the AlphaSlider requires showing the display output. Figure 16 shows the this specification. The value of the indicator is mapped into the value of the slider through the expression "**psval[int(Indi.value)]**". This expression assumes an array of strings, **psval**, which is the list of possible AlphaSlider values. The indicator value is converted to an index for this array and that value is displayed in a text output view port.

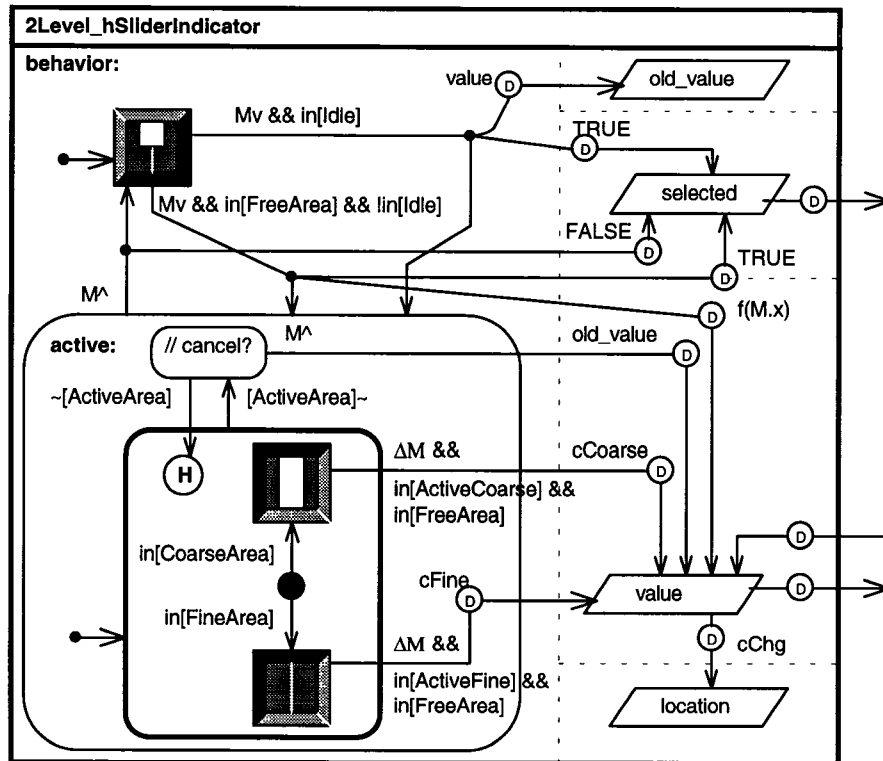


Figure 15 -- Behavior of 2-level horizontal slider indicator.

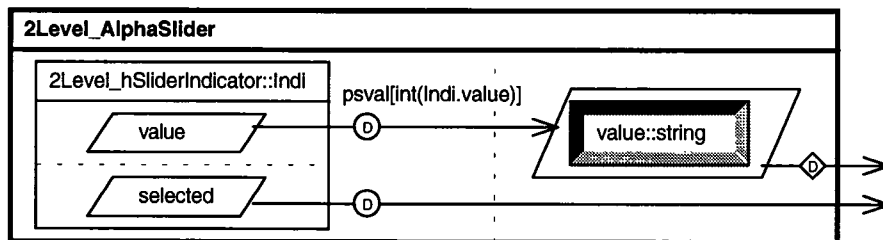


Figure 16 -- Specification of 2-level AlphaSlider.

3.3 The Treeview Widget

The treeview widget provides a node-link visualization of any tree structure. It was developed for use in configuring satellite networks [Kumar et al., 1994]. Treeviews have traditionally been used for a number of applications, for example:

1. Hierarchy Visualization Applications: Treeviews have been used to visualize hierarchical information such as geographical data, computer file systems, satellite communications networks and organizational structures. Various other tools such as Treemaps [Shneiderman, 1992] and Cone-Trees [Robertson et al., 1992] are also used.

2. Non-Hierarchy Visualization Applications: Treeviews are widely used in the areas of probability and set theory, reliability engineering and systems engineering. Event trees and fault trees are two examples.

The treeview widget and the treemap widget make use of the same tree data structure. They may be thought of as two different view port mappings of the same data. The design of the tree and treeview data structures was done in such a way as to be flexible, so that it could be extended depending on the specific application. For example, a tree can have multiple treeviews.

The treeview widget takes a tree as input, generates the layout, and draws the treeview. Any arbitrary tree (m-ary tree) can be visualized. The treeview is laid out horizontally, and the user can choose the orientation (left-to-right or right-to-left). The treeview can be browsed in X and Y directions using the horizontal and vertical scrollbars, respectively (Figure 18).

This widget has been used in the Tree-browser application (Figure 18). This application is a visualization tool for hierarchical data that makes use of dynamic queries [Williamson and Shneiderman, 1992] and pruning. The Tree-browser uses two coordinated or tightly-coupled (Figure 19) treeviews of the same tree, one a detailed view and the other a miniature view or overview. The user can select attributes (both numerical and textual) for querying nodes at each level in the hierarchy. Nodes at higher levels in the hierarchy that do not match the query cause their subtrees to be pruned out of the visualization. Thus, one can reduce a huge data set (with thousands of nodes) to a much smaller set, from which a good selection can be made.

The treeview widget operates as standard window which scrolls the node-link representation of the tree. Thus, the specification (Figure 17) consists of a view port with two scroll bars. Since scroll bars are supplied by the window system tool kit, they are represented by a "black box" in our specification system. (One can also build them from the horizontal slider already defined.) The view itself is visualized as a view port over a canvas which contains the entire tree already drawn. View ports have associated with them a mapping function which is controlled by a translation point (**translate**) and scale (magnification). Since the treeview uses a constant scale, **translate** is specified by two constraints, one from the horizontal scroll bar and one from the vertical scroll bar.

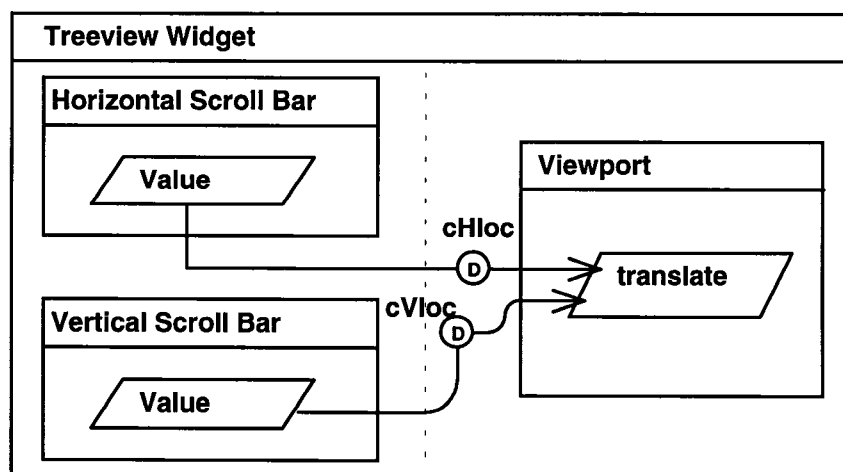


Figure 17 -- Treeview widget specification.

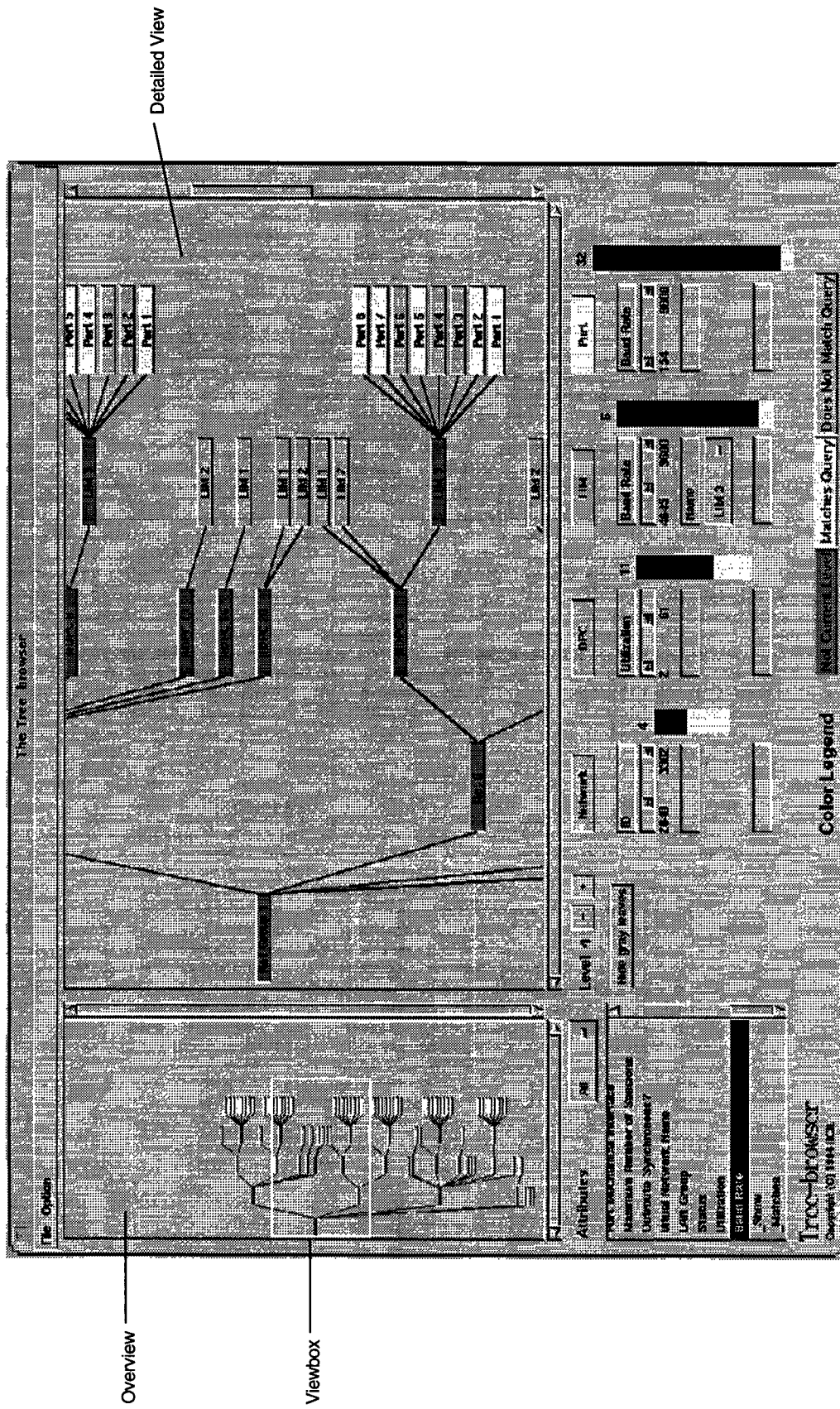


Figure 18 -- Two treeview widgets used in the Tree-browser application.

The coordination between the two viewers may also be specified using IOGs. The left-hand treeview widget is the overview. The viewbox in the overview is just a draggable icon. It is constrained to be located within the overview by assigning the **FreeArea** and **ActiveArea** inside of the overview. The values of the viewbox use the same range as the scroll bars for the right-hand treeview widget or detailed view. Finally, constraints are assigned between the viewbox value and the detailed view scroll bars (Figure 19). Note, the data arcs assigning these constraints have Boolean conditions which limit updating to the widgets not being manipulated.

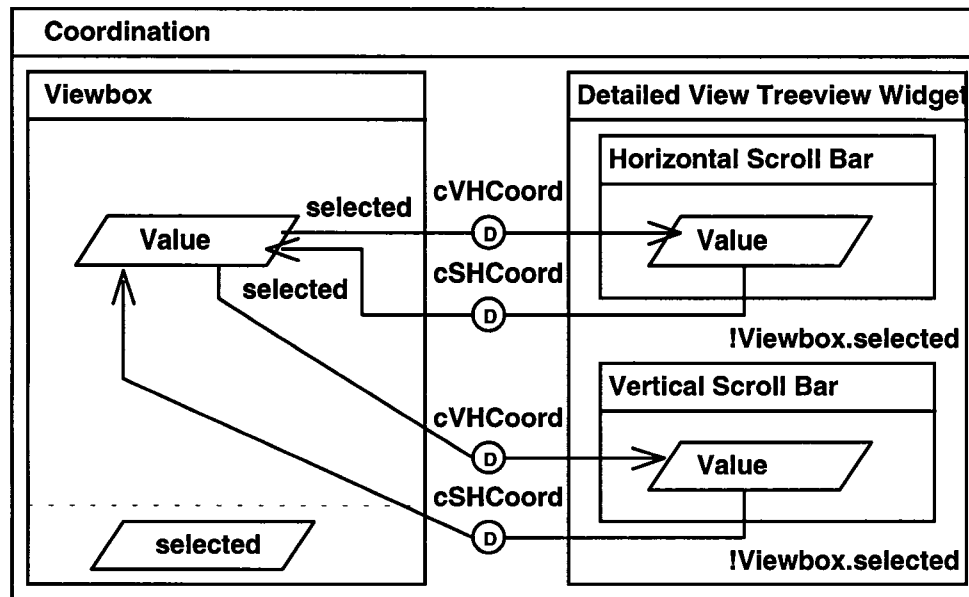


Figure 19 -- Specification of treeview coordination in the Tree-browser application.

3.4 The Treemap Widget

The treemap viewer uses a space filling algorithm to reduce the display space required to view a hierarchy [Shneiderman 1992]. The treemap has been used to visualize file directories [Johnson and Shneiderman, 1991], budgets, stock portfolios [Jungmeister and Turo, 1992], satellite network links [Kumar et al., 1994], and other tree structured hierarchies. Many applications handle vast amounts of data. It is often difficult to present this data to the user in a way that would give the user a good overview of the data and at the same time allow easy access to the details of the information. The Treemap is a data visualization tool developed for this purpose.

Treemaps (Figure 20) are best suited for viewing hierarchical data, and when necessary, a hierarchy can be created for almost any kind of data. Instead of displaying a conventional tree structure where most of the display space remains unused (Figure 18), treemaps take the space-filling approach of Venn diagrams and divide the parent node area among its child nodes (Figure 21). The area of a treemap is divided among the nodes corresponding to their weights, so a treemap displays more information than a conventional tree diagram. Therefore, treemaps display hierarchically structured information in relatively small screen space.

The basic attributes of a treemap are the tree, root node, selected node, child node offset, orientation in which the nodes are divided, number of levels displayed, node name display, size

attribute, and color attribute. The size attribute values can be filtered, which is very useful if the variation in node sizes is small. The size attribute can also be inverted, so that nodes with small values appear large on the screen and nodes with large values appear small. The color palette used for coloring the nodes can also be changed.

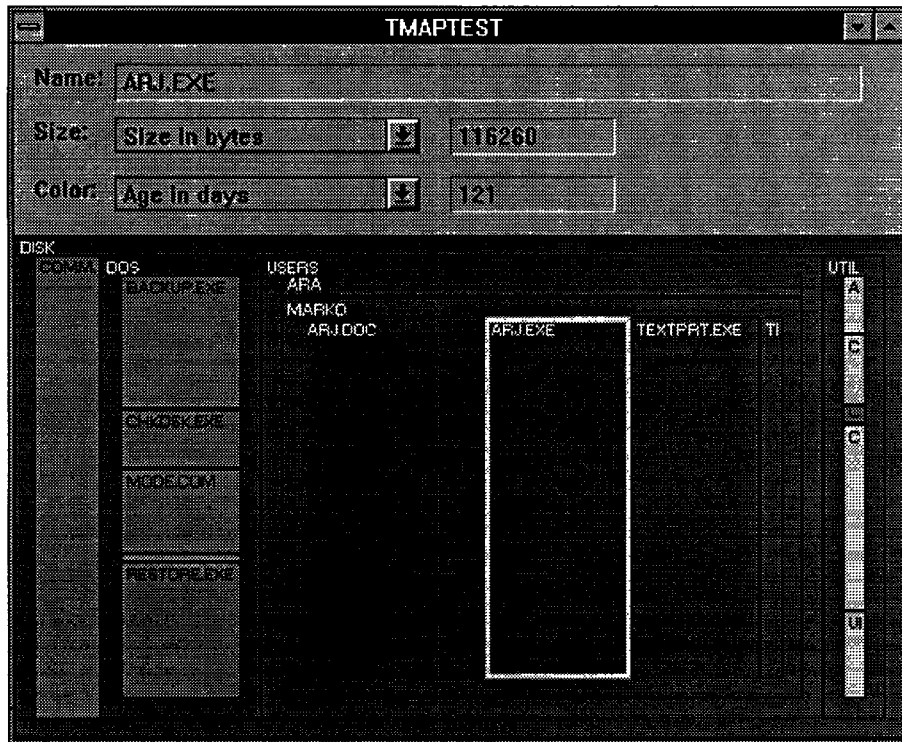


Figure 20 -- Screen snapshot of a treemap.

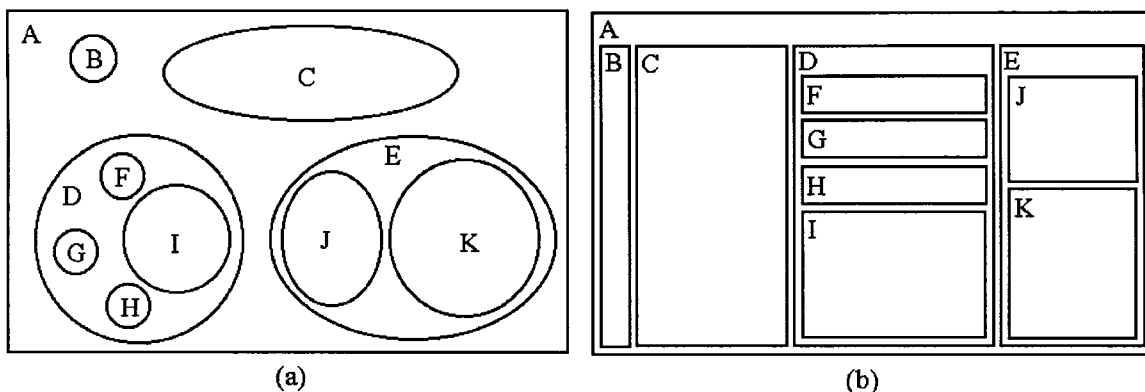


Figure 21 -- a) Venn diagram b) Corresponding treemap

A tree map is constructed by first selecting a display root node that can be any node in the tree. The root node covers the entire treemap display area. Changing the display root node is a good way to zoom into the hierarchy. The area of the display root node is divided between its children according to their size attribute value. The division is done either horizontally or vertically

depending on the orientation attribute. The direction of division is alternated on successive levels of the tree. If the treemap offset attribute is greater than zero, a border of *offset* pixels is left around the divided node area. The node name can be displayed in the upper left corner of the node if needed. The deeper levels of the hierarchy can be left out to display higher levels more clearly. The selected node is highlighted with a (colored) wide border.

The sizes of the nodes are determined by any numerical attribute of the nodes. The nodes with large values are displayed large. If there are lots of the nodes, the smallest ones do not necessarily get any screen space. If many nodes have approximately the same size, the size attribute can be filtered through a function to exaggerate the size differences making it easier to find the largest node. If the hierarchy contains a huge (uninteresting) node that covers most of the screen space. This node can be closed and its area given to other nodes. The leaf nodes of the tree can be colored according to any numerical attribute they have. The number of colors and the colors themselves in the palette can be selected by the programmer.

The user can select any node on the treemap by clicking it with mouse button 1. The selected node is indicated by highlighted borders. If the user holds the mouse button down and moves the pointer around, the selection follows pointer movement. The user can zoom into the treemap by double clicking a node with mouse button 1. This causes the selected node to fill the entire treemap display space. The user can zoom out one level at a time by clicking mouse button 2. The parent node of the displayed root node becomes the new root for the treemap display filling the entire area.

In addition to the complex graphics, the treemap viewer presents an additional specification problem -- where is the line between the application and the interface? Clearly, the interface should handle events like pointing and clicking. But, what about closing nodes, zooming-in, zooming-out, and changing the attribute mapped to size? Each of these actions changes the treemap being viewed and is more properly part of the application. The specification presented here will assume that the application handles changes to the treemap and will concentrate on generating events which the application needs to change the view.

The treemap viewer consists of two component types: a treemap node and a treemap viewer. The node IOG would be created for each tree item by the application. For each node the application would also create an identifier. This identifier will be assumed to be an index into arrays for the three attributes of the treemap (Name, Size, and Color in the example in Figure 20). Now, whenever the user points to a node with the mouse button pressed, the node identifier is exported from the TreemapNode (Figure 22).

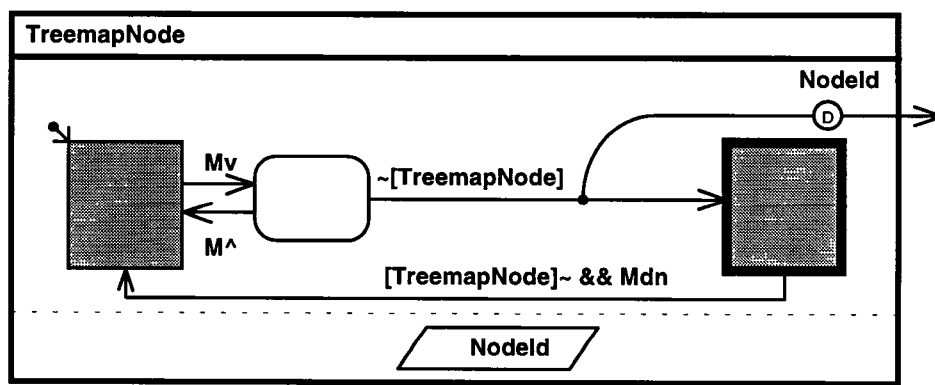


Figure 22 -- Specification of treemap node.

The treemap itself (Figure 23) consists of a group of TreemapNodes, a currently selected node attribute, and displays for the Name, Size, and Color mappings. The group of nodes is depicted as the stack of meta-states on the left. They all are connected to CurrentNode which in turn connects to the displays. The CurrentNode to display connection is an index into a string array initialized by the application. The displays are of the standard single height list display found in all window systems and have not been specified in order to save space.

The zooming operations have been attached to the entire treemap. A double click ($Mv^{\wedge}v^{\wedge}$) generates the **ZoomIn** event. Similarly, clicking the second (right) mouse button generates the **ZoomOut** event. It is assumed that an implementation would attach these events to semantic functions in the application.

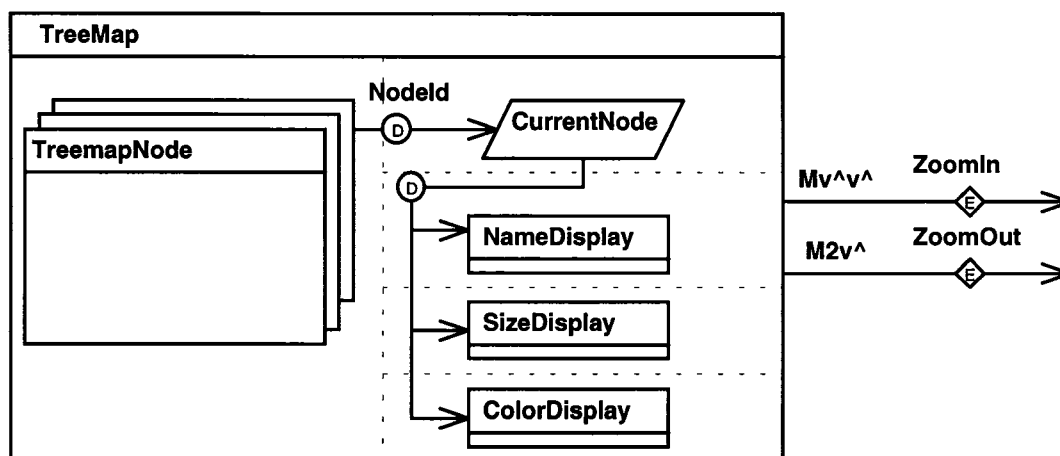


Figure 23 -- Specification of treemap viewer.

3.5 Secure Toggle Switch

Most existing buttons in computer applications give no indication of the current state - novice users get confused as to whether the label on a button portrays its current state or is an indication of the action it performs.

The lever toggle uses direct manipulation and visual feedback to perform the following functions:

1. Indicate the current state of the device in an unambiguous manner.
2. Make obvious how the user can change the state of the toggle.
3. Acknowledge the user's actions by providing appropriate visual feedback.

Additionally, a secure lever toggle (Figure 24) guards against an accidental change of state by requiring a deliberate user action.

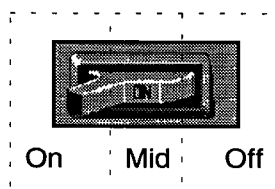


Figure 24 -- The secure switch (shown with specification regions).

Physical secure lever toggles find extensive use in heavy industry - for example in controls that should not be switched on or off by the accidental push of an operator, like aircraft auto pilot/manual settings. In computer applications the secure toggle can be used in machinery simulators, like an aircraft simulator, and in any application that requires a deliberate user action. They can also be used in computer-controlled machinery, like the control panel of various home appliances such as air conditioners, security systems, ovens, etc.

The "secure" switch is based on the designs by Plaisant and Wallace [1990 & 1992]. The switch requires that the operator point at its current state and drag the switch to its new state in order to operate it. This prevents inadvertently changing the switch position (and the controlled system). In order to turn the switch on, the operator would have to move the mouse to the "Off" region of the switch, press the mouse button, drag the mouse into an intermediate region, drag the mouse into the "On" region, and release the button. Releasing the mouse button in any region other than "On" will return the switch to the off state. Turning the switch off requires similar manipulation. Each step in the operation sequence corresponds to a similar state in the user interface.

In order to specify the user interface we need to define three regions (Figure 24) and five icons. The regions relate the mouse position to the corresponding side of the switch image. The icons provide feedback to the user about the switch value and about its operation. Figure 25 shows the specification and the icons.

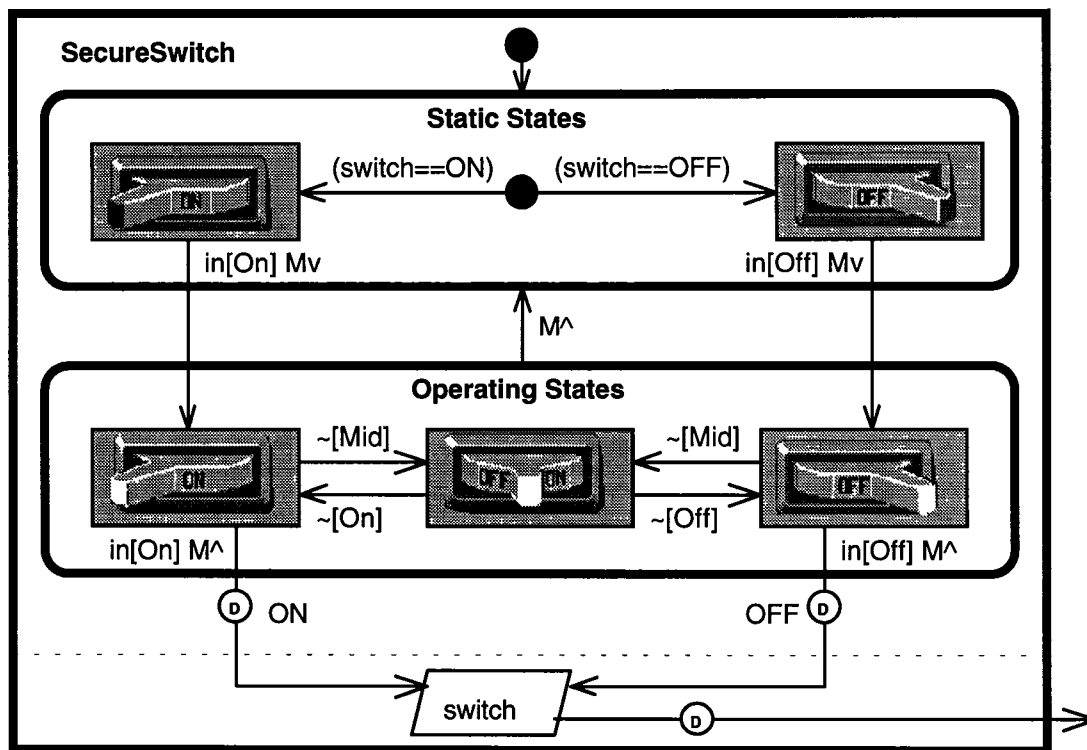


Figure 25 -- Specification of a secure switch.

To get a better idea of how the specification is interpreted let's trace a user turning the switch on. At startup, "Static States" is entered at the start state. A test is made on the initial value of "switch" and either the "on" image or the "off" image is displayed. In this case, assume the

system starts with the value of switch as OFF. So, the IOG initially moves to the "Static States - Off" and displays the icon shown. The user now positions the mouse in the Off region and presses the mouse button. This satisfies the **in[Off] Mv** condition and the IOG moves to "Operating States - Off". This move causes the switch display to light up. Next, the user drags into the middle of the switch. This activates the **~[Mid]** event and causes the transition to "Operating States - Mid" to be taken. Again the display changes when this occurs. The user continues and drags into the On region, the **~[On]** event occurs, and the transition to "Operating States - On" occurs. At this point the user releases the mouse button. This enables two transitions: **in[On] M^** on the data arc to "switch" and **M^** from "Operating States" to "Static States". By convention all data arcs are evaluated first. So, the value of "switch" changes to ON. Since the **M^** transition is to the meta-state and not to a contained state, "Static States" is restarted at its start state. The transition to "Static States - On" is taken and the display updated to the "on" switch which is not lit up. If the user had moved the mouse out of the On region before releasing the mouse button, only the **M^** transition would have been enabled. Thus, when "Static States" was restarted the "switch" would have been OFF and the display would have returned to the off state.

3.6 Summary of Experiences

We found the specifications helpful in implementing the widgets. In the case of the secure switch, the programmer raced ahead before the specification was completed and programmed from a verbal description and a working prototype of the original toggle switch. However, the resulting switch was not quite correct. Reprogramming from the specification produced correct behavior and reduced the size of the code. The specifications were also helpful in answering questions like "What happens when the range selection slider indicators meet?" However, there were problems implementing from the specifications. Since we were using Galaxy for implementation and its abstractions are not the same as the IOG abstractions, there was a certain amount of translation required. We also found that we were doing more work to specify and then code as the specifications required revisions as well as the implementations.

We also found that for widgets like the secure switch or the AlphaSlider, the IOG method gave clear visualizations of the widget behavior. On the other hand, more complex widgets such as the tree viewer and tree map viewers were very abstract and required careful reading to extract meaning. There was also a trade-off between the level of detail required for an operating widget and the level of detail required for a "readable" specification. Real widgets have lots of attributes which don't affect the runtime behavior of the widget. A complete specification requires their presence. However, they clutter the diagrams with data objects and make the diagrams harder to read. We settled on the convention that the diagrams would only show data objects involved in behavior. Hierarchically organizing the widget specification and reducing some details to a "black box" helped readability.

Finally, adding data arcs and data object nodes to IOGs help tremendously in visualizing the relationships between the data and widget behavior. The one-way constraints were useful for expressing relationships between attributes. This was so, even though the implementation system did not directly support constraints. However, one drawback of constraints is that they are deceptively simple. The first constraint written usually results in sliders which can slide out of

their containers and values which are outside the desired ranges. We found that many constraint equations needed to be guarded by minimums and maximums.

4 FUTURE RESEARCH

One feature that was sorely missed was being able to execute the specification directly. This would have saved considerable time. Also, testing the specifications directly would reveal errors and usability problems before committing to code. Even a library which implements the IOG abstraction would be a help. This would eliminate one possible source of error, the translation from IOGs to the target development system. A system which allowed the designer to draw the widget IOG and then directly executed it would be a valuable tool for prototyping.

Another area which shows promise for future work is algorithms to detect common errors. While we feel it would be impossible to prove a widget error free, some common errors should be detectable. For example, many widgets have a direct mapping between a display state (display node in IOGs) and the value of a widget attribute. At least for those states which represent the widget when the user is not manipulating it, it would be desirable to verify that the attribute value could be guaranteed. Verifying the attribute value is closely related to the usability concept of visibility of underlying system state. Forgetting a data arc is the most common cause for this error. Other properties to check could include freedom from "sink states" and "dialog completion". A sink state is one which, once entered, cannot be left and is a violation of the reversibility principal of direct manipulation interfaces. Dialog completion would insure that the widget returned to "non-operating" states when the user stopped manipulating it. "Non-operating" states represent the widget when the user is not manipulating it. For example, in the secure switch releasing the mouse button should result in the widget entering one of the display states in the top group (Figure 25). Any other state would be an error and might give the user incorrect feedback. (e.g., displaying the switch in the on-active position when it was off and not being operated, might cause the user to conclude the switch was on.)

5 ACKNOWLEDGMENTS

We would like to acknowledge the advice of Richard Chimera, Ben Shneiderman, Catherine Plaisant, and the other HCIL members for both widget design and help in preparing this paper. Sylvia Sheppard and Chris Rouff of NASA Goddard Space Flight Center provided both encouragement and support. Finally, we would like to thank Carl Rollo for diligently proofreading the drafts of this paper.

6 REFERENCES

- Ahlberg, C. and Shneiderman, B. 1994a. The Alphaslider: a compact and rapid selector. *Proceedings of the ACM CHI'94 Conference on Human Factors in Computing Systems*, pp. 365-371.
- Ahlberg, C. and Shneiderman, B. 1994b. Visual information seeking: tight coupling of dynamic query filters with starfield displays. *Proceedings of the ACM CHI'94 Conference on Human Factors in Computing Systems*, pp. 313-317, 479, & 480.

- Becker, R., Eick, S., Miller, E., and Wilks, A., 1989. Dynamic graphical analysis of network data. in *ISI Conference Proceedings*, Paris, France.
- Becker, R., Eick, S., and Wilks, A. 1991. Basics of network visualization. *IEEE Computer Graphics and Applications*, Vol 11(3), pp. 12-14.
- Carr, D. 1994. Specification of interface interaction objects. *Proceedings of the ACM CHI'94 Conference on Human Factors in Computing Systems*, pp. 372-378.
- Gieskens, D. and Foley, J. 1992. Controlling user interface objects through pre- and postconditions. *Proceedings of the ACM CHI'92 Conference on Human Factors in Computing Systems*, pp. 189-194.
- Guttag, J. and Horning, J. J. 1980. Formal specification as a design tool. *Proceedings of the 7th Symposium on Programming Languages*, pp. 251-261.
- Hartson, H. R., Siochi, A. C., and Hix, D. 1990. The UAN: a user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, Vol. 8, No. 3, (July), pp. 181-203.
- Hartson, H. R. and Gray, P. D. 1992. Temporal aspects of tasks in the user action notation. *Human-Computer Interaction*, Volume 7, Lawrence Erlbaum Associates, Inc. pp. 1-45.
- Harel, D. 1988. On visual formalisms. *Communications of the ACM*, 31(5), (May), pp. 514-530.
- Hill, R. D. 1986. Supporting concurrency, communication, and synchronization in human-computer interaction - the Sassafras UIMS. *ACM Transactions on Graphics*, Vol. 5, No. 3, (July), pp. 179-210.
- Hudson, S. E. 1989. Graphical specification of flexible user interface displays. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 105-114.
- Jacob, R. J. K. 1986. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, vol. 5, no. 4, (October), pp. 283-317.
- Johnson, B. and Shneiderman, B. 1991. Tree maps: a space-filling approach to the visualization of hierarchical information structures. *Proceedings of the 2nd International IEEE Visualization Conference*, San Diego, (Oct.), pp. 284-291.
- Jungmeister, W.-A. and Turo, D. 1992. Adapting treemaps to stock portfolio visualization, University of Maryland, Center for Automation Research technical report CAR-TR-648 (also CS-TR-2996, SRC-TR-92-120).
- Kumar, H. P., Plaisant, C., Teittinen, M., and Shneiderman, B. 1994. Visual information management for network configuration, University of Maryland, Institute for Systems Research technical report, ISR-TR-94-45 (also CAR-TR-716, CS-TR-3288).
- Myers, B. A. 1988. *Creating User Interfaces by Demonstration*, Academic Press, Boston.
- Olsen, D. R. Jr., 1990. Propositional production systems for dialog description. *Proceedings of the ACM CHI'90 Conference on Human Factors in Computing Systems*, pp. 57-63.
- Olsen, D. R. Jr. 1992. *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann Publisher, San Mateo, California.
- Osada, M., Liao, H., and Shneiderman, B. 1993. Alphaslides: searching textual lists with sliders, University of Maryland, Center for Automation Research technical report CAR-TR-673 (also CS-TR-3078 & SRC-TR-93-52).
- Plaisant, C. and Wallace D. 1990. Touchscreen toggle switches: push or slide? design issues and usability study, University of Maryland, Center for Automation Research technical report CAR-TR-521 (also CS-TR-2557).

- Plaisant, C. and Wallace D. 1992. Touchscreen toggle design. *Proceedings of the ACM CHI'92 Conference on Human Factors in Computing Systems*, pp. 667-668 and the CHI'92 Formal Video Program.
- Robertson, G. G., Mackinlay, J. D., and Card, S. K. 1991. Cone trees: animated 3d visualizations of hierarchical information. *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pp. 189-194.
- Rouff, C., *Specification and Rapid Prototyping of User Interfaces*, University of Southern California Ph.D., 1991, 219 pages.
- Rouff, C. and Horowitz, E. 1991. A system for specifying and rapidly prototyping user interfaces. in *Taking Software Design Seriously*, J. Karat, editor, Academic Press, 1991, pp. 257-272.
- Shneiderman, B. 1982. Multiparty grammars and related features for defining interactive systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 12(2):148-154, (March/April).
- Shneiderman, B. 1992. Tree visualization with tree maps: a 2-d space-filling approach. *ACM Transactions on Graphics*, Vol. 11, No. 1, (January), pp. 92-99.
- Singh, G., Kok, C. H., and Ngan, T. Y. 1990. Druid: a system for demonstrational rapid user interface development. *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 167-177.
- Siochi, A. C., and Hartson, H. R. 1989. Task oriented representation of asynchronous user interfaces. *Proceedings of the ACM CHI'89 Conference on Human Factors in Computing Systems*, pp. 183-188.
- Vander Zanden, B. T. 1989. Constraint grammars -- a new model for specifying graphical applications. *Proceedings of the ACM CHI'89 Conference on Human Factors in Computing Systems*, pp. 325-330.
- Visix Software Inc. 1994. *Galaxy Application Environment Version 2.0*.
- Wasserman, A. I. 1985. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, vol. SE-11(8), (August), pp. 699-713.
- Wellner, P. D. 1989. Statemaster: a UIMS based on statecharts for prototyping and target implementation notation for specification. *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems*, pp. 177-182.
- Williamson, C. and Shneiderman, B. 1992. The dynamic HomeFinder: evaluating dynamic queries in a real-estate information exploration system. *Proceedings of the 1992 ACM SIGIR Conference*, Copenhagen, pp. 338-346.