CAR-TR-687 Sept. 1993
CS-TR-3142

Specification of Interface Interaction Objects

David A. Carr

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

KEYWORDS

User Interface Specification, User Interface Design

ABSTRACT

User Interface Management Systems have significantly reduced the effort required to build a user interface. However, current systems assume a set of standard "widgets" and make no provisions for defining new ones. This forces user interface designers to either do without or laboriously build new widgets with code. The Interface Object Graph is presented as a method for specifying and communicating the design of interaction objects or widgets. Two sample specifications are presented, one for a secure switch and the other for a two dimensional graphical browser.

INTRODUCTION

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 primitives (interaction objects or widgets) exist and manipulate the presentation of these objects. 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. This paper presents the Interaction Object Graph as an approach to this widget building problem. IOGs can be used to specify interaction objects at a higher level than programming languages. As such they could be the basis for extending UIMSs to allow design and prototyping of new interaction objects.

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, and interface representation graphs.

Shneiderman's multiparty grammars[8] 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.

Algebraic specification of window systems was introduced by Guttag and Horning[1]. 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 provide any provision for describing system actions. Siochi and Hartson's User Action Notation (UAN) is one language which also makes a contribution by specifying computer feedback and interface internal state[2,10]. However, UAN concentrates heavily on describing user actions and is not well adapted to describing software state. Specifying system responses to unexpected user actions is awkward. In addition, UAN's tabular notation does not readily show relationships between tasks.

Another approach to modeling user interfaces is the transition diagram[12]. 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 solved part of this problem by allowing concurrent states to coexist as parallel machines or co-routines[5]. Co-routines did not completely solve the transition complexity problems for specifying some systems (notably those with context sensitive help).

Harel's statecharts[3,13] 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) used by Rouff[7] as the underlying representation for his Rapid Program Prototyper (RPP) extend the statechart to represent dialog. Extensions to statecharts 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 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. However, this method does not do very well when run-time re-sizing is allowed. Recent research has settled on constraint grammars[11] to solve this problem. Hudson's Apogee UIMS[4] has a particularly clever method for setting layout constraints graphically.

INTERACTION OBJECT GRAPHS

Interaction Object Graphs (IOGs) 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.

Interaction Object Graph State Diagrams

The IOG state diagram traces its lineage from traditional state diagrams, Harel's statecharts, and Rouff's 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. Both of these state types were present in IRGs. However, their meaning is slightly different in IOGs. Data objects are represented as parallelograms. (See 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)".

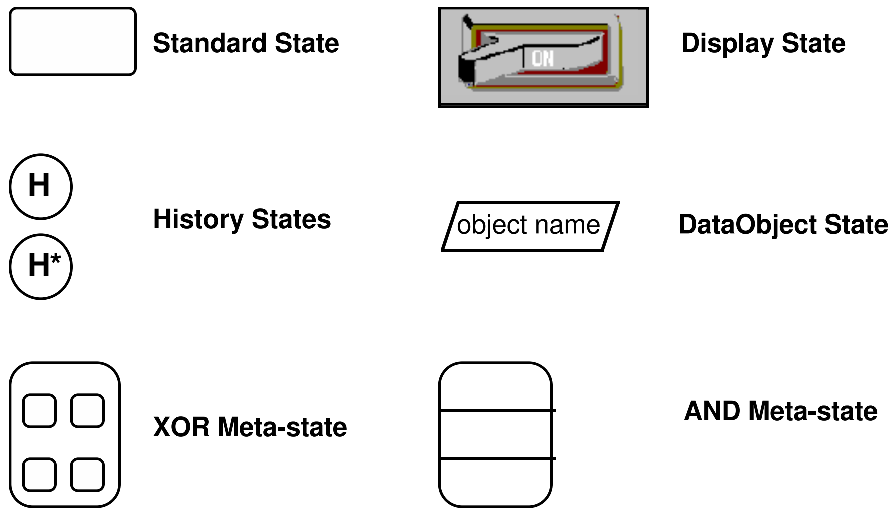

Figure 1 -- IOG state symbols.

IOGs also add three new transition arc types. These are: the event arc, data arc, and constraint 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 highlight 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. (See Figure 2.) The user defined event is not present in either IRGs or statecharts, but was used in Jacob's transition diagrams. The purpose of an event transition is to create new symbols for use when describing transitions.

Data flow is represented in a manner similar to events – an arc passing through a **D** in a diamond. (See 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 interactor object, are considered to be exported. The 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. IRGs have a similar mechanism, but data arcs only represent data interchange with the application.

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. Constraints are restricted to interactions between two data objects. A constraint is shown with an arc passing through a **C** in a diamond. (See Figure 2.) The arc indicates that the destination data object is dependent on the source data object. Constraint arcs come from IRGs and are not present in statecharts.

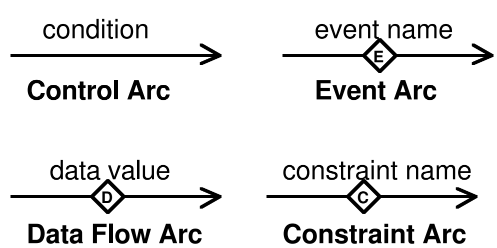

Figure 2 -- IOG arc symbols.

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, points, regions, icons, view ports, windows, and user inputs. A brief description of these objects follows.

Booleans and numbers are the usual abstractions with the usual operations. It should be noted that numbers contain both the real and integer data types.

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 projection is controlled by a point named **translate** and its scale change is controlled by a point name **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 and add a level 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↵" 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 (**M**), a Boolean indicating it moved (**M**), 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@)**.

EXAMPLE SPECIFICATIONS

Below are two examples of how IOGs would be used to specify components of a user interface. The first example is a special form of a toggle switch which models the behavior of a rocker switch. The second example is the specification of a two dimensional graphical browser and shows how the IOGs might be used to build a visual description language for browsers. Both of these examples concentrate on the behavior of the components. Many details such as colors, fonts, and actual locations are not given. These details would only serve to make the examples more complex and less illustrative. In addition, these details would normally be determined when the widget is incorporated into a complete user interface. These properties are also very simple to specify with a property list, while behavior cannot be so specified.

Secure Toggle Switch

The first example will be a "secure" switch based on the designs by Plaisant[6]. This switch is designed to prevent accidental manipulation of a device under computer control. For example, this switch could be use for the master power switch for factory machinery. 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 is similar. 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 and five icons. The regions relate the mouse position to the corresponding side of the switch image. (See figure 3.) The icons provide feedback to the user about the switch value and about its operation. Figure 4 shows the specification and the icons.

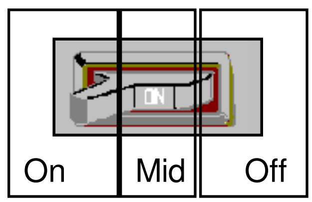

Figure 3 -- Regions used in specification of a secure switch

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, then 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.

It is interesting to note that specifying the secure switch with UAN takes about a page. A transition diagram requires 9 states and 20 transitions, and a statechart is about the same as figure 4. However, none of these methods give the reader a clear idea of the switches appearance or the dynamic changes in its appearance.

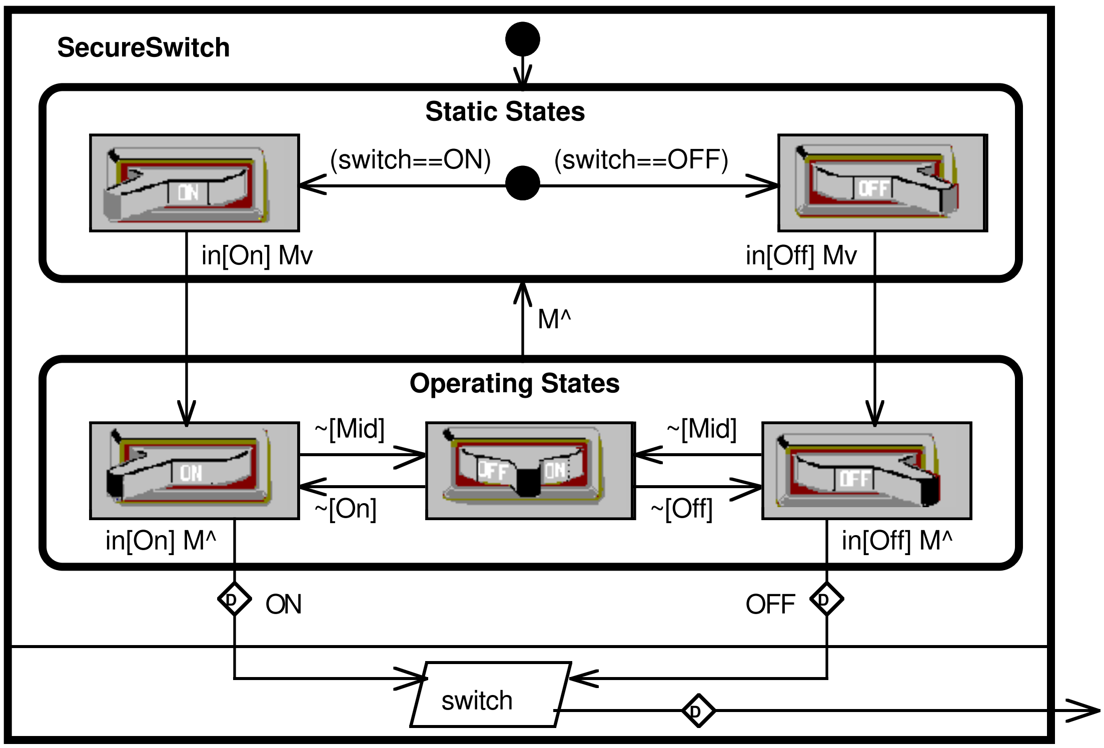

Figure 4 -- Specification of a secure switch

Two Window Graphical Browser

In this example we wish to specify a two-window graphical browser with coordination between the two windows. Figure 5 shows the coordination in a diagram patterned after DM Sketch[9]. This browser consists of two parts. The window on the left is called the overview. It shows the entire graphical image. (i.e., If the application were to browse a US road map, then the entire US would be in the overview.) The right hand window is called the detailed view it shows a magnified view of some portion of the overview. (i.e., The detailed view might contain a rectangle enclosing Maryland.)

The relationships between the detailed view and the overview are indicated by the gray symbols and the browser elements are shown in black. The springy arrow represents movement constraints on the object at its tail. In particular it means that object is free to move in the context of the object at its head. The crossed box in the overview means that it is a view box which is used to mark the view in another window. The view box is tied to the detailed view by three symbols. The first is the "%=" line between the view box's horizontal movement indicator and the right window's slider movement indicator. The second is a similar line between the vertical slider and the vertical movement indicator. The meaning of these symbols is that the two objects are constrained to move together proportionately. That is if the view box is moved to 50% of its free area, then the sliders move 50% of their travel and visa versa. The final symbol is the "P" arrow. This is the graphical projection symbol and means that the items in the view box are magnified and displayed in the target window.

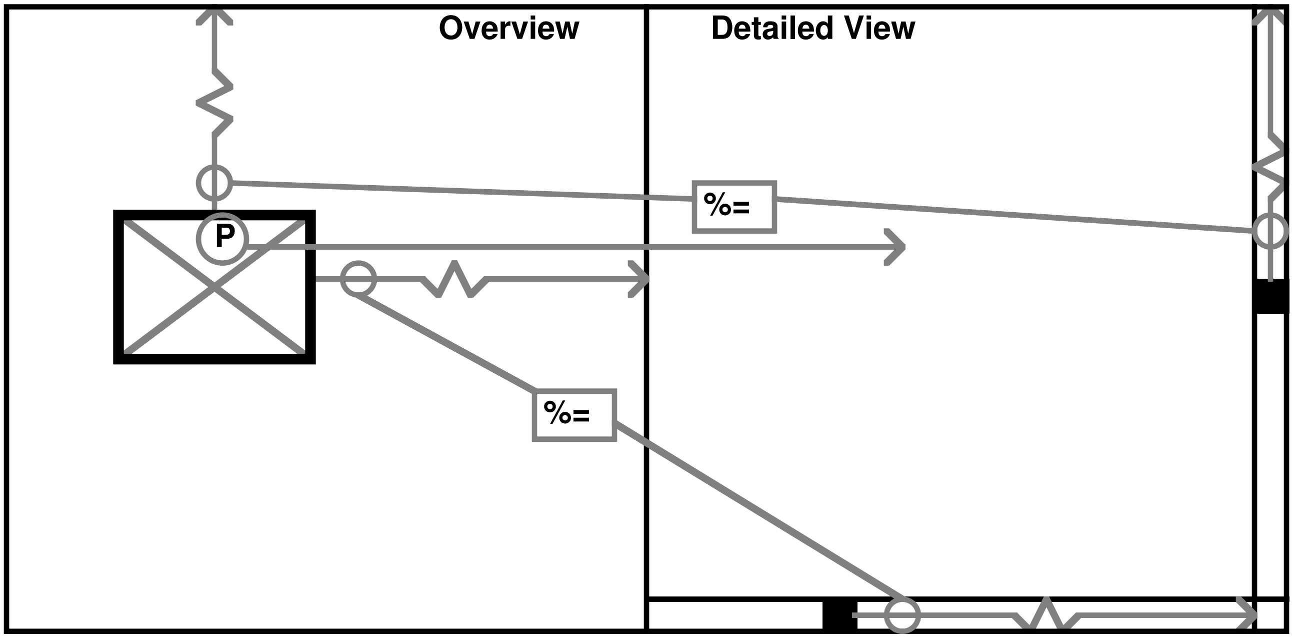

Figure 5 -- Coordination between windows in a two window graphical browser.

Now the browser needs to be specified with IOGs. First, let's start with a horizontal slider. The slider consists of an indicator which has two icons **Idle** and **Active**. It has a two regions associated with the indicator **FreeArea** and **ActiveArea**. **FreeArea** is that region in which the indicator will follow the mouse. **ActiveArea** is a region that includes **FreeArea**. Going outside of **ActiveArea** and releasing the mouse button will reset the slider to its previous value. (Thus, **FreeArea** allows the user to drift out of the slider and still operate it.) Figure 6 specifies the behavior of the indicator. The value of the slider is related to the indicator through the three constraints **cDrag**, **cChg**, and **cMove**. **cDrag** relates the mouse location to the indicator location and is only valid when the indicator is active and the mouse is in FreeArea. The actual constraint would be "location.x = location.x+M.x". **cChg** is valid only when the indicator is selected and propagates the location changes to the slider value (not shown). Assuming the slider had a base value of "BASE" and a range of "RANGE" then **cChg** would be:

SCALE = (location.x - FreeArea.location.x) /
FreeArea.size.x
value = BASE + RANGE * SCALE

cMove is the inverse of **cChg** and moves the indicator when the value of the slider is changed by the application program.

It is obvious that by changing any '.x' to '.y' in all of the constraints and laying out the regions appropriately we can convert this specification to be a vertical slider indicator. What is not so obvious is that this same specification can be modified and used for the view box as well.

Consider what happens when we define the value to be a point. Now, the **FreeArea** is defined to be the part of the left window that can contain the view box location while the view box is visible in the window. Furthermore, let the **ActiveArea** be the window itself. Now, let **cDrag** be "location = location + M", **cChg** be "value = location", and **cMove** be "location = value". If the icons are defined to be a rectangle filled transparently, then one has a draggable rectangle whose value is its location. (It should be noted that the specification just described can be used for a general 2D draggable icon which selects two values.)

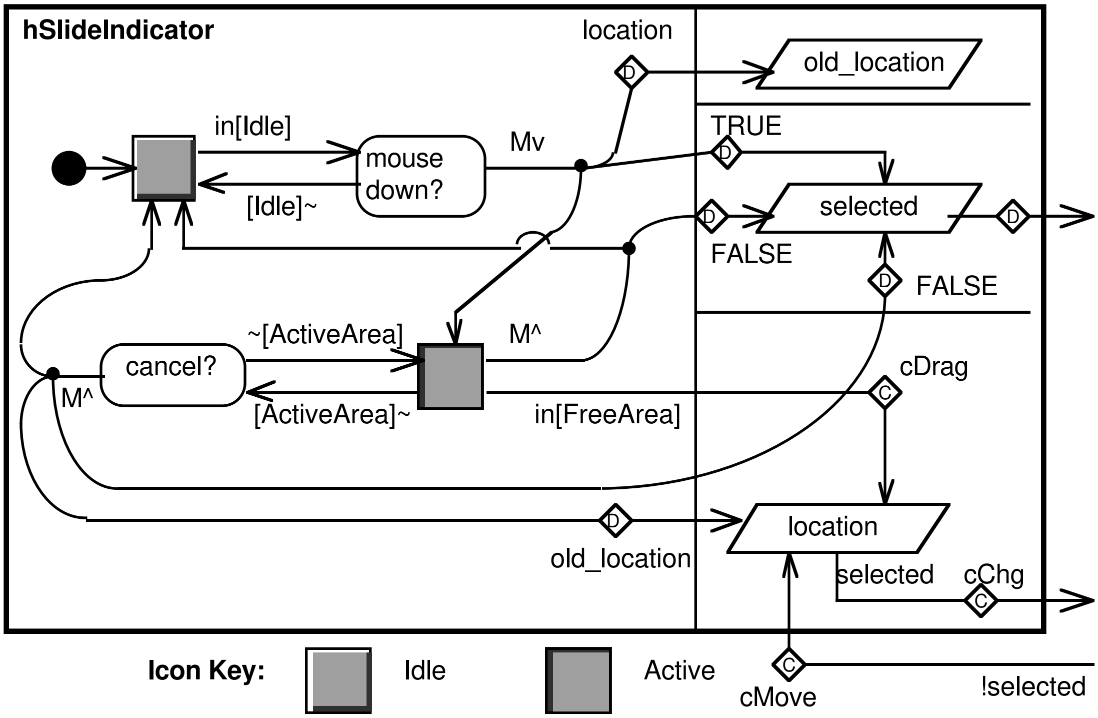

Figure 6 -- Specification of a horizontal slider indicator

Now that the view box and sliders have been defined, the coordination between the windows needs to be specified. The proportional constraints can be specified as conditional constraints on the view box value. Let vb be the view box, hs be the horizontal slider, and vs be the vertical slider. Then:

if (vb.selected) then hs.value = vb.value.x;
vs.value = vb.value.y;
if (¬vb.selected) then vb.value = (hs.value,vs.value);

specifies the necessary relationships. If rw is the right-hand window (detailed view) and lw is the left-hand window (overview), we can characterize the mapping function for its view port vp with:

rw.vp.translate = lw.vp.translate - vb.location;
rw.vp.scale = lw.vp.scale * lw.size / vb.size;

As long as both windows use the same conversion function, this completes the specification of the graphical browser.

The above example shows how IOGs may be used to succinctly specify the essential behavior of a two window browser. It shows the similarities between vertical sliders, horizontal sliders, and a two dimensional value selector such as the view box. Finally, it demonstrates the power of combining constraint equations with a hierarchical state diagram.

CONCLUSION

The IOG shows promise as a method to specify and communicate designs of new user interface interaction objects. It has been used at the Human Computer Interaction Laboratory to specify a library of new interaction objects. This effort has shown two shortcomings. First, the level of detail needed to completely specify an interaction object quickly swamps the specification. This makes it difficult to see both "the forest" and "the important trees". Flexible interaction objects can have several dozen data attributes and an IOG for one becomes dominated by a large number of parallelograms. The preliminary solution to this has been to only show those attributes which affect dynamic behavior in the IOG and to define the rest in a property list. The second problem is that we are still coding the new interaction objects. It would be very useful to have a tool which allowed one to edit an IOG and directly execute it. A project to do this has been started at the NASA Goddard Space Flight Center.

ACKNOWLEDGMENTS

For their encouragement and support, I wish to thank Sylvia Sheppard and Chris Rouff of NASA Goddard Space Flight Center, Ben Shneiderman, and the other members of the University of Maryland Human-Computer Interaction Laboratory. Special thanks go to Carl Rollo for proof reading and constructive criticism of early versions of this paper.

REFERENCES

[1] Guttag, John and J. J. Horning, "Formal Specification as a Design Tool," *Proceedings of the 7th Symposium on Programming Languages,* 1980, pp. 251-261.

[2] Hartson, H. Rex and Phillip D. Gray, "Temporal Aspects of Tasks in the User Action Notation", *Human-Computer Interaction,* Volume 7, 1992, Lawrence Erlbaum Associates, Inc. pp. 1-45.

[3] Harel, David, "On Visual Formalisms," *Communications of the ACM*, 31(5), May 1988, pp. 514-530.

[4] Hudson, Scott E., "Graphical Specification of Flexible User Interface Displays," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* 1989, pp. 105-114.

[5] Jacob, Robert J. K., "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transactions on Graphics*, vol. 5, no. 4, October 1986, pp. 283-317.

[6] Plaisant, Catherine, and Wallace D., *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), Nov. 1990.

[7] Rouff, Christopher, *Specification and Rapid Prototyping of User Interfaces,* University of Southern California Ph.D., 1991, 219 pages.

[8] Shneiderman, Ben, "Multiparty Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics,* 12(2):148-154, March/April 1982.

[9] Shneiderman, Ben, *Designing the User Interface, 2nd edition*, Addison-Wesley, 1992.

[10] Siochi, Antonio C. and H. Rex Hartson, "Task Oriented Representation of Asynchronous User Interfaces", *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems,* 1989, pp. 183-188.

[11] Vander Zanden, Bradley T., "Constraint Grammars -- A New Model for Specifying Graphical Applications," *Proceedings of ACM CHI'89 Conference on Human Factors in Computing Systems,* 1989, pp. 325-330.

[12] Wasserman, Anthony I., "Extending State Transition Diagrams for the Specification of Human-Computer Interaction", *IEEE Transactions on Software Engineering,* vol. SE-11(8), August 1985, pp. 699-713.

[13] Wellner, Pierre D., "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,* 1989, pp. 177-182.