MCC Technical Report Number ACT-HI-135-89

The HITS Icon Editor

Mark Rosenstein and Louis M. Weitzman

MCC Nonconfidential

June, 1989

MCC TECHNICAL REPORT

MCC Technical Report Number ACT-HI-135-89

The HITS Icon Editor

Mark Rosenstein and Louis M. Weitzman

MCC Nonconfidential

June, 1989

Abstract

The Icon Editor is a platform for exploring the construction of dynamic graphical icons and the techniques for relating those entities to an application. The research goals of this work are to discover new techniques for the graphical specification of behavior and to develop a foundation for the connection of an application to an interface. The broader vision is a framework supporting a series of knowledge-based tools to aid an interface designer in building interfaces without coding.

A significant portion of the effort expended in the construction of a computer project is in the design of its graphical interface. The appearance and functionality of this interface can critically affect the useability and the overall aesthetics of the program. These issues are being addressed by *The Human Interface Tool Suite (HITS)*, an integrated design environment being developed in the Human Interface Laboratory of MCC. (CONTINUED next page.)

Microelectronics and Computer Technology Corporation
Advanced Computing Technology
Human Interface Laboratory
3500 West Balcones Center Drive
Austin, TX 78759
(512) 343-0978

MCC Technical Report Number ACT-HI-135-89

At a given moment, an interface provides a window into and a means to manipulate an application. The total interface, then, is a series of these windows providing multiple perspectives on the application. One technique to achieve a highly interactive system is to build the interface from dynamic components. We refer to these components as *icons*, which represent a state or states within the application. An icon presents a state through some aspect of its graphical appearance.

Microelectronics and Computer Technology Corporation

Advanced Computing Technology

Human Interface Laboratory

3500 West Balcones Center Drive
Austin, TX 78759
(512) 343-0978

Table of Contents

List of Figures

The HITS Icon Editor

The Specification of Graphic Behavior Without Coding

by
Mark Rosenstein
Louis Weitzman

1. Abstract

The Icon Editor is a platform for exploring the construction of dynamic graphical icons and the techniques for relating those entities to an application. The research goals of this work are to discover new techniques for the graphical specification of behavior and to develop a foundation for the connection of an application to an interface. The broader vision is a framework supporting a series of knowledge-based tools to aid an interface designer in building interfaces without coding.

2. Introduction

A significant portion of the effort expended in the construction of a computer system is in the design of its graphical interface. The appearance and functionality of this interface can critically affect the useablity and the overall aesthetics of the program. These issues are being addressed by *The Human Interface Tool Suite (HITS)*, an integrated design environment being developed in the Human Interface Laboratory of MCC. Interface design information is represented in a common knowledge base and utilized by a wide variety of tools.

At a given moment, an interface provides a window into and a means to manipulate an application. The total interface, then, is a series of these windows providing multiple perspectives on the application. One technique to achieve a highly interactive system is to build the interface from dynamic components. We refer to these components as *icons*, which represent a state or states within the application. An icon presents a state through some aspect of its graphical appearance. Time paths are represented by graphical behavior. These behaviors range from positioning a needle in a dial to flipping the switch in an image of a wall light switch. Input dynamics involve interactions by which the icon is modified, such as rotating the position of the needle or toggling the switch. These actions change the appropriate state.

HITS contains a cascade of facilities concerned with the graphics incorporated within the interface. The suite of graphic programs includes a tool, the *Graphics Editor* [Weitzman, Rosenstein, and Winkler, 1989], to allow domain experts to build iconic interfaces. These interfaces are the views that monitor and control the application. Another tool, the *HITS Icon Editor*, allows the design and creation of

new icons to be used in these interfaces. Integrated with the Icon Editor and Graphics Editor is *Designer*, a set of tools and design knowledge which interactively analyzes, advises, and supports the selection of alternatives to the graphical presentations generated by these tools.

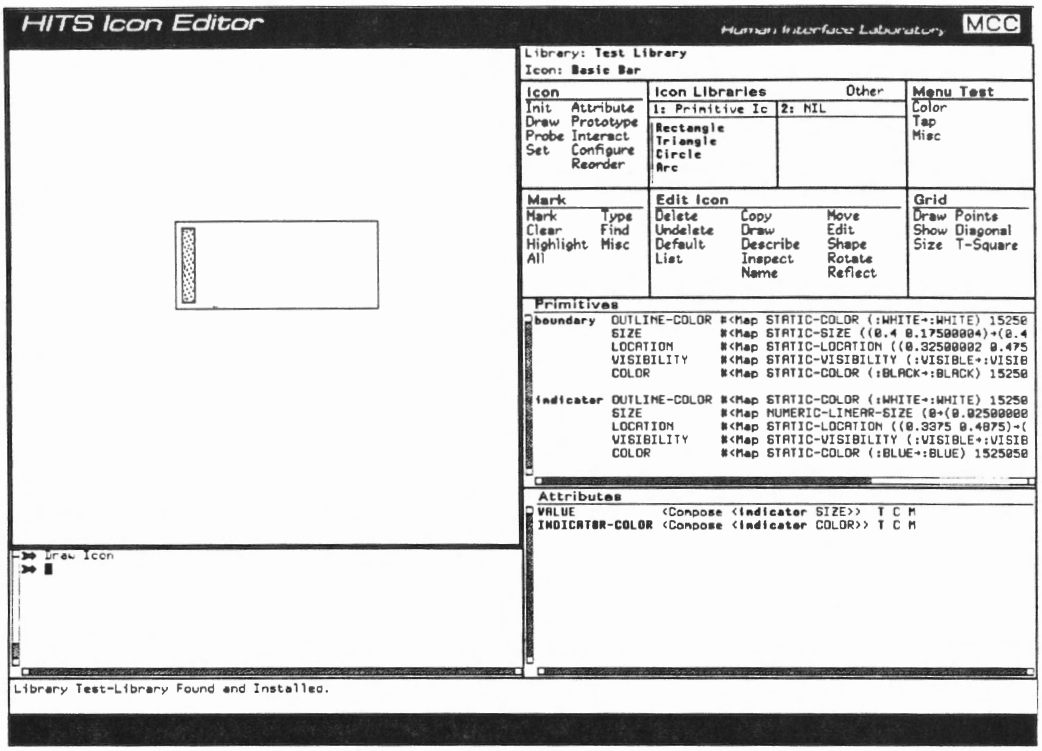

Figure 1. The HITS Icon Editor

A major motivation of the Icon Editor was to expand the class of domains and users for HITS. We concluded that no existing set of icons would be sufficient for all applications. We also concluded that hand coding of icons restrained the domain of end users to too great an extent. What was needed was the ability to create the interface objects interactively without coding. We adopted the Graphics Editor model of providing the user with a set of primitives and the ability to combine, incrementally refine, and critique these primitives.

This paper discusses the specification of these dynamic icons for use in an interface. Considered here are both a conceptualization for building appearance and behavior and the realization of these ideas in a prototype. In order to specify the behaviors of these new icons, this work generalizes the notion of *tapping*, the process of relating an interface object to its application. The tapping mechanism supports a two-way communication between the interface and the application. Not only do the icons monitor their application state, but they can also modify it via mouse input. This representation often involves a transformation of values between application and graphical states.

The remainder of this paper will discuss the presuppositions behind our work, a look at related efforts, the theoretical basis of our work, the current implementation, some examples of constructing icons, and finally, our future directions.

3. Presuppositions

Knowledge Based Tools

From both a research methodology and practical perspective, tools provide greater leverage than one of a kind implementations. Therefore, our effort is focused on the development of tools to further the work within our lab. Tools allow a community of users to develop and share in expanding resources, e.g., libraries of icons, that can be used across multiple projects and domains. While embodying the interface constraints and knowledge of their initial design, tools are also an enabling media allowing the creation of artifacts beyond those imagined by the tools' designers.

Interfaces constructed with HITS are targeted toward knowledge based applications. As computer systems move from mere calculation machines to true collaborators, both applications and interfaces must maintain representations of their structure and actions. Our tools aim to build next generation interfaces having effective graphical presentations, advising, and natural language interactions. All of these are doomed from the onset unless there are representations over which they can perform computations. The implication is that the dynamic icons must have explicit representations. Currently icons and behaviors use an object oriented paradigm with full multiple inheritance, differentiating classes and instances. An ongoing effort is to move these objects to a frame based system, where we can take advantage of constraints as a more expressive media for the relations among these entities.

Classes of Users

We have developed a perspective that categorizes the task of interface design into three levels. One characterization of these levels is that they distinguish the generality of the task being executed.

The least generality is afforded the end user performing the domain task. For this user, the interface must help solve the problems of the application. This might be a point of sale terminal, a library reference request system, the operating console of a power plant, or interfaces to systems we have yet to imagine. This interface must support the task by taking advantage of the skills of its operators and compensating for the operators' shortcomings. This is the output of HITS tools like the Graphics Editor.

The next level involves the design of the end user interface. This design must include all the perspectives needed to support the task. The designer knows the task and how the end user will likely perform it. From this user centered view the interface designer must build the interface from a set of pre-existing components and connect the interface to the application. The Graphics Editor is targeted at this level of user.

The last level provides the most general control by building and specifying the components available to the interface builder. The component designer must anticipate the need for the icons and construct them so that they can be placed in future interfaces. For instance, the overall graphical style or lexicon used as input for a product line would be specified at this level. This is the level of abstraction a user of the Icon Editor must consider.

It is possible that one person may fill these multiple roles, but different classes of his or her knowledge must be utilized to successfully complete each task. These different levels provide discrete points on a continuum of control and specification. One can easily generate tools that fall between these levels. By placing more constraints on the interface designer's tool, the flexibility of that tool decreases. These constraints can be imposed in a number of ways: by limiting the choice of icons available, enforcing relationships between elements, and constraining the actions of the tool. For instance, the Graphics Editor allows the construction of any interface from available libraries of icons. This may provide too much freedom for a company that wishes to maintain consistency throughout its product line. The company could specialize the Graphics Editor to become a product interface design tool. This tool would provide the standard components that the company builds into its machines and embody the company's style and design rules.

What We Are Not

It is important to note that the goal of the Icon Editor and Graphics Editor is not to provide support for building window systems. We have neither the manpower nor the desire to duplicate efforts that address these issues. The dynamic entities that we create are developed within an existing system, and this system provides facilities that we use as basic resources, e.g., window frames and pop-up menus. In the much longer time frame, we may be able to generalize this work to the building of window configurations, but that is not on the current research agenda.

4. Related Work

We have seen the increased commercial availability of tools that allow a user to construct the appearance of an icon [e.g., Macintosh interfaces and Steamer graphics]. What has been lacking in all but a few cases has been the ability to specify the behavior for these icons. Mentioned here are a number of related research efforts that allow users to specify interface behavior without programming. Whether the technique be *programming by example*, *visual programming*, or some other technique, the common goal is to provide a tool that frees the designer from having to write code. One factor that distinguishes the various approaches is the domain in which the system is used. Each domain places strict requirements on the types of behaviors that are needed. A taxonomy of various systems that use some of these techniques is provided in [Myers, 1986].

In *programming by example* the final result is specified through examples presented to the system. The system infers what actions to take from these examples. This technique has been applied in a variety of domains, including education (Laura Gould's Programming by Rehearsal [Gould and Finzer, 1984]) and interface design (Myers' Peridot [Myers and Buxton, 1986]).

Peridot creates user interfaces by having the designer demonstrate to the system how the interface should look and feel. This interaction generates code that can then be executed. The technique to generate this code is automatic inferencing, simple condition-action rules that help the system *guess* what the designer's intentions are. These rules are used in the specification of the behaviors that the elements take and in their presentation on the screen. The system supports behaviors for building the user interface which includes menus, scroll-bars, and light buttons.

Programming by Rehearsal does not use inferencing but does allow the user to specify behavior by example. Through the use of the theater metaphor, *productions* are created with *performers* carrying out specific tasks. These performers rely heavily on predefined actions. Each action is associated with a specific predefined *cue*. The *writing* of the interactions of the performers in a production is done mostly, but not exclusively, by having the system *watch* the designer perform the actions once. A nice feature of systems that use programming by example is that everything is visible, and the designer of the interface is always thinking concretely.

Visual Programming allows the specification of programs through the use of graphics. Borning's further work with **Thinglab** [Borning, 1986] is a very good example of using graphics to *visually program* constraints. Constraints are specified by *hooking* up objects and then running them to get their new combined behaviors. These objects are the building blocks of the system. For detailed networks, however, the use of graphics becomes questionable because of the complexity it introduces in the representation with which the designer must interact.

A different approach, one more similar to ours, is taken in Foley's **Process Visualization System** [Foley, 1986]. Instead of specifically laying out the constraints of how a new icon will behave or stepping through an example of what the icon might do, this approach provides a mechanism to modify the specific attributes that exist in the primitives. These attributes change as they reflect an underlying dynamic process. This external process drives the appearance of the interface. The constraints at work are the constraints within the external process, and it is the state of the variables that affects the behavior of the new icon. Binding to the process occurs by connecting a portion of an item in the view and a process variable from a process library or *data dictionary*. The data dictionary contains the variables that represent the values that can be monitored. A major difference with this system and ours is that we restrict the ability to modify attributes in the final presentation. The icon builder specifies what can be modified, while it is the view builder, not necessarily the same person, who connects those attributes in the specific instance. By encapsulating this new behavior, we build an icon that will have a consistent presentation for future views.

5. Basis of the Icon Editor

A fundamental idea of our system is that aspects of the application need to be viewed and manipulated. The graphics tools of HITS provide for this interaction by converting application state into graphic behaviors. An icon provides a language for this conversion by allowing an interface designer to specify displayable and modifiable aspects of the application. These characteristics are communicated by graphical characteristics controlled by an icon's attributes. A crucial task is to specify these attributes. With a foundation of primitive icons containing attributes of *size*, *location*, *visibility*, and *color*, new icons with new attributes can be created to be recombined into more complex icons.

The creation of an icon consists of four major steps. The first step specifies the appearance of the icon. This involves *placing* and *shaping* instances of existing icons. In the second step, attributes of these instances that will display dynamic values are modified. As part of this process the type of input to the attribute is specified. In the third step, new attributes for the new icon are created. The final step completes the process by *specifying* how the new attributes control the characteristics of the attributes identified in the second step. Upon completion, instances of the new icon can be immediately used in building views or as components of other icons.

In the remainder of this section, we will discuss the Icon Editor's support for each of these steps: 1) *primitive icons*, the components used in creating a new icon, 2) *attributes*, the displayable characteristics that can be modified dynamically, 3) *transformation maps*, the objects that convert application values and user input to graphic values, 4) *constraints*, the methodology to propagate input to existing primitives' attributes, and 5) *typing*, the use of types to assist in the construction of the new icons.

5.1. Primitives

The basic mechanism for creating a new icon starts with the composition of icons from previously defined icon libraries. This provides the structure for the new icons. A selected icon is positioned and sized on the display screen. Both color and black-and-white screens are supported. In the case of black-and-white screens, the system represents color intensities in shades of gray. The icon under construction is placed in the current library indicated by the status line at the top of the Icon Editor frame.

Figure 2 illustrates the icon library menus which provide access to the individual icons. These libraries are generally organized by type, e.g., button icons, or by project, e.g., the Copier Interface Editor. The working set of libraries from which primitive icons are chosen is dynamic and can change to allow for primitive components from many different sources.

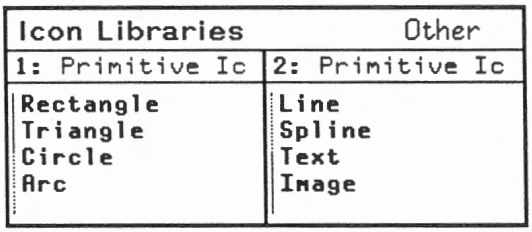

Figure 2. The icon library menu pane

The libraries from which the designer may choose icons include a default primitive library and user defined libraries built within the Icon Editor. The default library currently consists of a set of hand-coded icons composed of rectangles, triangles, circles, arcs, lines, splines, text, and bitmap images. An ongoing experiment is to see how well we can bootstrap the system providing a wide variety of icons from this limited set of initial primitives.

5.2. Attributes of Primitives

Attributes are the controllable characteristics of an icon. The modification of these characteristics create the interactive and dynamic behavior of the icons. Each icon maintains the graphic attributes necessary to present itself. As a minimum, every icon includes the attributes of *color, location,* and *visibility*. The color attribute affects the color in which the icon will be drawn, the location attribute affects where the primitive will be drawn, and the visibility attribute affects whether a primitive will be drawn at all. Icons that enclose an area (e.g., rectangles, circles) also include the *size* and *outline-color* attributes. Primitives like *text* include specialized attributes, such as *horizontal-justification* and *font*.

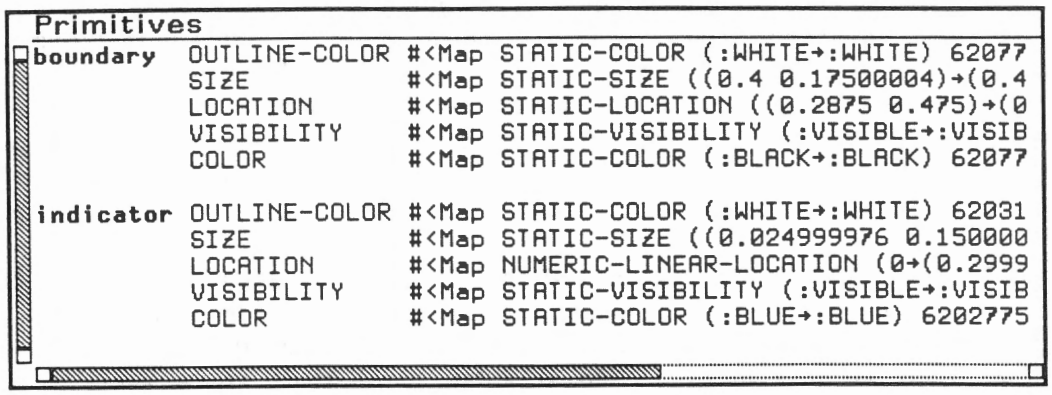

Figure 3. The primitive inspector pane showing attributes that can be dynamically controlled

In Figure 3, we see the primitive inspector. The first column contains the name of the icon. The second column contains the names of attributes associated with the icon. The last column contains a complex representation of the map that is on the attribute. It is the maps that provide the mechanism for dynamic change.

By default, newly created icons only include the attributes of *color*, *location*, and *visibility*. Any additional attributes are icon specific and must be added by the icon designer. These new attributes can then be used to constrain the behavior of this icon as it is being used in an interface. Together, the default and user defined attributes provide the control points from the Graphics Editor into the new icon. When relating this new icon to the application, only these attributes will be accessible via tapping to the interface builder. Similarly, when this new icon is used as a component of another icon, only the default and newly defined attributes will appear in the primitive inspector.

5.3. Transformation Maps

Attributes parameterize graphical behavior. A color attribute's value is used to determine the color of a part of an icon. The color attribute uses a vocabulary of color values, like red. Applications contain a separate language of numeric or symbolic values. In order to perform dynamic behaviors, each icon maintains a transformation map for each of its graphical attributes.

These maps accept an input value from an input device or the application and modify it via a transformation. The result is a *mapped value* that is used by the icon to determine how to display the attribute. Maps are implemented as objects with defined input and output types. The convention for map names is to concatenate the input type, the transformation, and the output type. For example, a *numeric-linear-size* map takes a number and linearly transforms it to a size.

The system contains a hierarchy of maps that utilize inheritance. The numeric-linear-size map is based on a more general *continuous-map*. Maps of this class use minimum and maximum inputs to compute outputs. In this case, the outputs are minimum and maximum size specifications, so as the input moves between the minimum and maximum value, the size smoothly moves between the two specified size values. See Appendix B for a more complete list of current map types.

Newly created attributes initially contain static maps, whose input type and output type are the same and perform no transformation of their input value. Static maps maintain a fixed state independent of changes in the application. They can also be used in cases where their input value is of the same type as their required displayable value, for example, in an application whose states are already color values.

An example of a specialized map is the *continuous-color-map*. With this map a user sets minimum and maximum values and the colors to be associated with these endpoint values. As the numeric input goes from the minimum to the maximum, the attribute to which this map is attached will continuously change from the color at the minimum to the color at the maximum. For instance, one could build a thermometer icon and assign the inside color attribute to have a continuous color map. The color extremes could then change from an ice blue to brilliant red as the application value varies.

In an application with standard sets of values such as [OFF, ON], fixed graphical mappings can be utilized to provide uniformity throughout the interfaces being built. For instance, if colors are used to represent state, an OFF component could always be displayed as red, while an ON component is displayed as green. The *discrete* class of mapping takes fixed input states and transforms them into fixed output states.

Built into the system are a number of macro definitions that help support the creation of new maps. Here is the specification of the *off-on-color* map which takes the values [OFF, ON] and maps them to the color values [RED, GREEN]:

```
(defdiscrete-map OFF-ON-COLOR
                 :type-of-value 'off-on
                 :type-of-mapped-value 'color
                 :default-value :off
                 :mapping-list '((:off . :red)
                                 (:on  . :green)))
```

With these macros on-the-fly creation of discrete maps is very simple. Since all this macro specifies is the map name and the transformation from application names to graphical properties, a later release will provide an interface to make this specification.

5.4. Constraints

When building an icon, we allow the designer great latitude in determining the flexibility of the new icon. There is no requirement that the new icon be dynamic, and for some applications a set of icons that are placed merely for structure may be built. The more interesting case is the icon designer giving the interface designer varying degrees of modifiability by adding new behaviors to the icon. The icon designer provides this by creating new attributes and constraining them to the appropriate attributes of the component icons. We have investigated some simple constraint mechanisms, including *composition* and *state visibility*.

Consider the following example of a composition constraint. As part of building a bar graph icon, the designer wishes to allow a view builder the ability to label the x-axis. A simple way to do this is to place a text primitive centered beneath the x-axis. The text icon has an attribute, *text-string*, which the icon designer wishes to make available for the graph. An attribute of the graph is created called *x-axis-label*. This attribute is edited to be of type *text* and to compose the text-string attribute of the text icon.

Composition is the simplest form of constraint provided. This constraint has a type, in this case *text*, and a series of one or more attributes that are constrained to have the same value. In this case, we use the Icon Editor interface to make the attribute *text-string* be the attribute that *x-axis-label* constrains. When the view designer adds the graph and edits its attributes, one of the attributes will be *x-label-axis*. By default, this attribute will have a static text map. The view designer can put in an appropriate text string, like *IP Packets per Hour*, for the bar graph label.

Even from this simple design additional flexibility is available to the interface designer. In the application, the designer might provide the ability to control the period of the x-axis, i.e., days, months, or years. We could then tap our label into this period variable to generate meaningful labels as the period changed. The view designer chooses a *format-text* map for the x-axis-label, which uses a format string and variable in the tap to map to text. If the format string was "IP packets per ~a" and the map was tapped to the appropriate variable, the x-axis-label would vary between "IP packets per day," "IP packets per month," and "IP packets per year" depending on the data being displayed. To provide more flexibility, the icon designer could make available other characteristics of the text such as typeface and justification.

Another type of constraint is *state visibility*. Many applications require icons to vary their display among several discrete states. When this constraint is used on a new attribute, it identifies which primitives within an icon are to be made visible in the separate states. One interesting side effect of this type of constraint is that a new map is dynamically created that encodes the transformations necessary to describe the state transitions. More automatic creation of maps is a feature that we would like to support in future versions of our system.

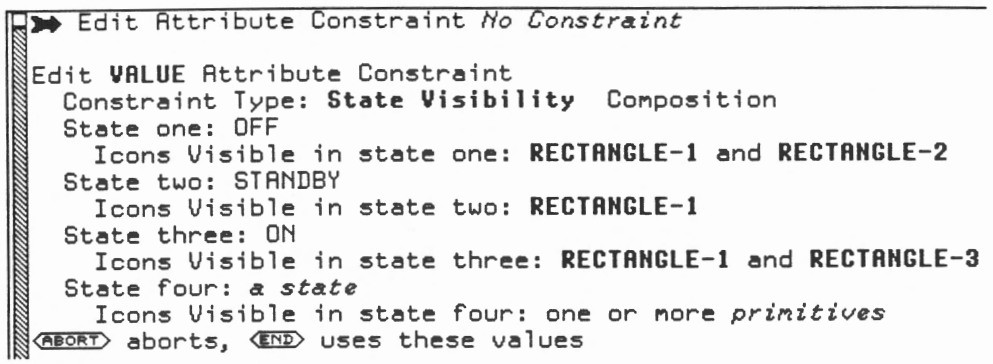

Figure 4. Editing the state visibility constraint for a new attribute

As seen in Figure 4, the state visibility constraint is described by the state names and the icons visible in each state. The input to this map will be one of the application states (e.g., [OFF, STANDBY, ON]), and the result will be to make the specified icons visible.

5.5. Typing

Every attribute is defined to be of a specific type. In addition, input and output values of maps are also typed. By convention, the type of a map is the type of its output value. The system uses this information to guide the users into making semantically correct choices for the selection of maps for attributes and input values for those maps. Each map placed in an attribute must have an output type which corresponds to the attribute type. For instance, only color maps may be placed within a color attribute. In addition, each map knows what type of input value it expects. For example, only [OFF, STANDBY, ON] are acceptable as input to an *off-standby-on-xxx* map (where xxx is an unspecified transformation and output type), and only colors will be generated by a *xxx-color* map (where xxx is an unspecified input type and transformation). With this knowledge specific help can be provided when the designer of the icon is unsure of the possible choices for attributes, maps, or values. This information is also used to guarantee an icon's attributes are tapped into legal values. Refer to Appendix A for the current attribute types.

For user input, a mouse type is defined that indicates a mouse click is necessary as an input value to a map. This stays within the general paradigm of the mappings used throughout the system and has proven effective as a general input technique.

6. Use of the Icon Editor

To demonstrate these ideas we will use two separate Icon Editor examples. The first example will highlight the construction of a family of icons to display a continuous range of values. Each icon will be a bar icon with a different technique for the presentation of a value within a set range. The second example will step through the process of using the Icon Editor to solve a typical problem of visualizing a new domain, that of monitoring the simulation of queuing in a multiple processor environment. In order to understand the implications of design tradeoffs in this simulation, a set of new icons was created to present the problem effectively.

6.1. Family of Bar Icons

The family of bar icons includes the *basic bar*, the *force bar*, and the *threshold bar*. Each of these icons only needs two primitives, a bounding rectangle which is the extent of the icon and an indicator rectangle which will change its size, position, and color depending on the desired presentation. The *basic bar* icon displays a value by changing the size of the indicating bar to reflect its value. The *force bar* presents its value by moving a fixed size indicator from left to right. The final bar icon, a *threshold bar*, enhances the force bar's functionality by modifying the indicator's color as the icon's value exceeds a threshold. This behavior is similar to dials used in process control applications which include redlines to indicate when an application parameter has exceeded a safe value.

6.1.1. Basic Bar

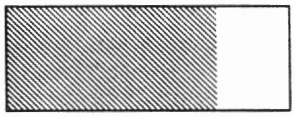

Figure 5. A basic bar icon

In the Icon Editor frame, we select the rectangle primitive. This rectangle is placed and sized on the display screen to become the exterior boundary of the bar icon. See *Demonstrations of HITS 1.0* for a more detailed tutorial on the step-by-step construction sequence. A second rectangle is added which will become the indicator bar and is renamed to be *indicator*. By clicking on the size attribute of the indicator, we will edit this attribute enabling it to transform numeric input from a minimum-maximum range to a minimum-maximum size. An editing menu is invoked to modify the map. The final edited menu is shown in Figure 6. We first alter the default static map to be of the new type, *numeric-linear-size* map. As we change the map, the parameters that are unique to the map (i.e., minimum and maximum values and minimum and maximum sizes) are presented with their

default values. The size values default to the existing size of the indicator rectangle. When modifying the maximum size parameter, the cursor is taken to the display pane where we graphically indicate what the indicator should look like when presenting its maximum value. In this case the maximum size will completely fill the boundary rectangle.

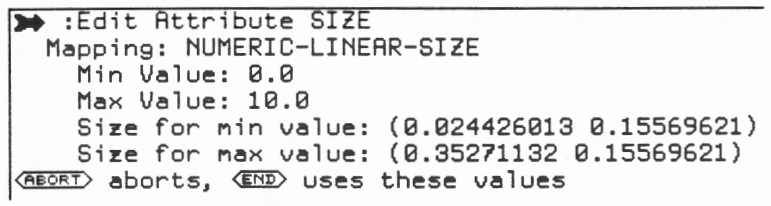

Figure 6. Editing the map for the attribute **SIZE** of the indicator primitive

Next the designer must create a new attribute that will be available to users of this icon in the Graphics Editor. This new attribute we will label *VALUE*, and it will constrain the indicator's size. When editing the constraint for this attribute, the designer selects the attribute type to be numeric, so that the maps on the existing attributes involved with this constraint must accept numeric values as input.

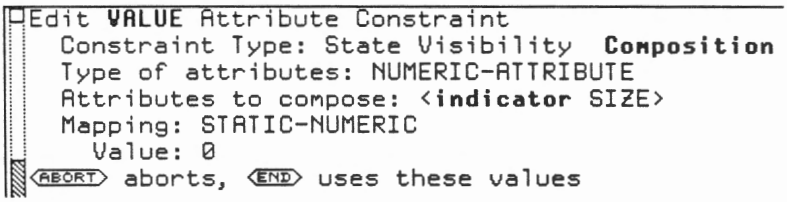

Figure 7. Editing the constraint for the new attribute **VALUE** on the basic bar

Then through a series of mouse actions, the designer indicates that this new attribute is equivalent to the size attribute of the indicator rectangle. The final attribute constraint definition menu is shown in Figure 7. In this way, the designer has specified that the new attribute *VALUE* will constrain the numeric size attribute of the indicator rectangle.

We can test this constraint by setting the value of *VALUE* between 0 and 10 and see the indicator change size proportionally between the minimum and the maximum specified sizes.

6.1.2. Force Bar

Figure 8. A force bar icon

In this variation of the bar icon, the value presented is via modification of the indicator's location, not its size. The specification of this behavior is very similar to the basic bar. The icon designer modifies the indicator's location attribute instead of the size attribute. As shown in the menu in Figure 9, values for the minimum and maximum locations can be modified. These default to the primitive's current location. When modifying the location for the maximum value, the user is taken to the display pane to position the indicator primitive. Intermediate values for positions are determined by a linear interpretation between the minimum and maximum locations.

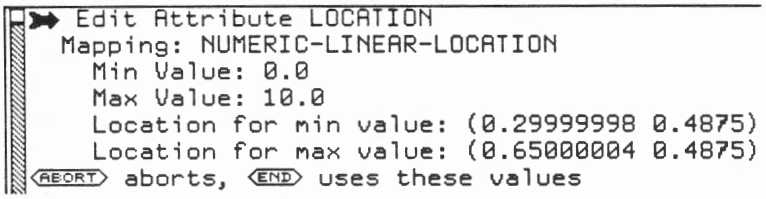

Figure 9. Editing the map for the attribute **LOCATION** of the indicator primitive

A new attribute will be created for the force bar similar to the basic bar's *VALUE* attribute. This attribute would be based on the indicator's location and will include a *numeric-linear-location* map. Again, the designer can test this prototype by changing the value of the attribute and seeing the position of the indicator change.

6.1.3. Threshold Bar

Figure 10. A threshold bar icon

The final icon in this series extends the capabilities of the force bar by adding a behavior to the indicator's color attribute. By changing the map on the color attribute, it, along with the *location* attribute, can be constrained by the new *VALUE* attribute. This map must also accept numeric input but should generate a color value used to draw the indicator.

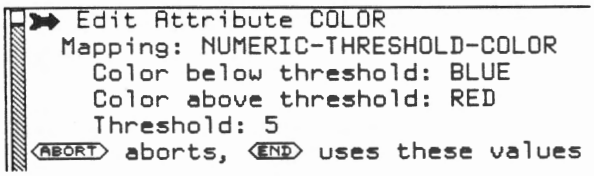

Figure 11. Editing the map for the attribute **COLOR** of the indicator color

The *numeric-threshold-color* map, shown in Figure 11, changes the color it generates when the input value exceeds a threshold value. As illustrated here, when the value exceeds the threshold of 5, the indicator will change from blue, its default, to red.

Once this map has been placed on the indicator's color attribute, it can be added to the attribute *VALUE*'s constraint, as shown in the editing menu in Figure 12.

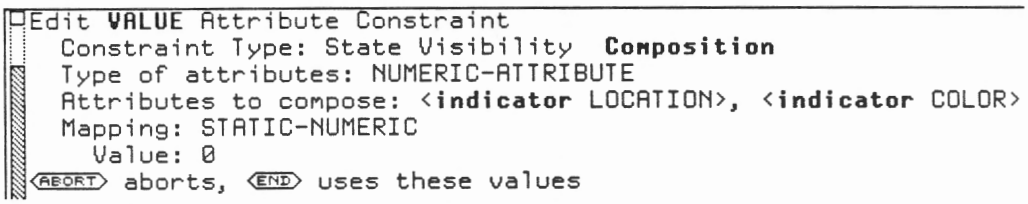

Figure 12. Editing the constraint for the new attribute **VALUE** on the threshold bar

This illustrates how multiple attributes of included primitives can be constrained. It is not necessary that these attributes be from the same primitive. Only that they accept the same type of input. As can be seen by these examples, it is very easy to build up a library of similar icons. By specifying which attributes will be modified and using the standard set of mappings, it is easy to specify the desired behaviors.

6.2. Queuing Example

This example helps to illustrate the capability of the Icon Editor to work in new domains. It illustrates the technique of modular icon development by incrementally building reusable components. This modularity allows these components to be included in libraries for future icon construction. This example was taken from an application developed by the Systems Technology Lab at MCC. A multiprocessor simulation generated data which needed graphic support for its analysis. An important condition to identify in the simulation is convoys. Convoys occurred when the multiple queues were being used inefficiently. The important point is not the application but the process of creating the supporting tools to help visualize this new domain.

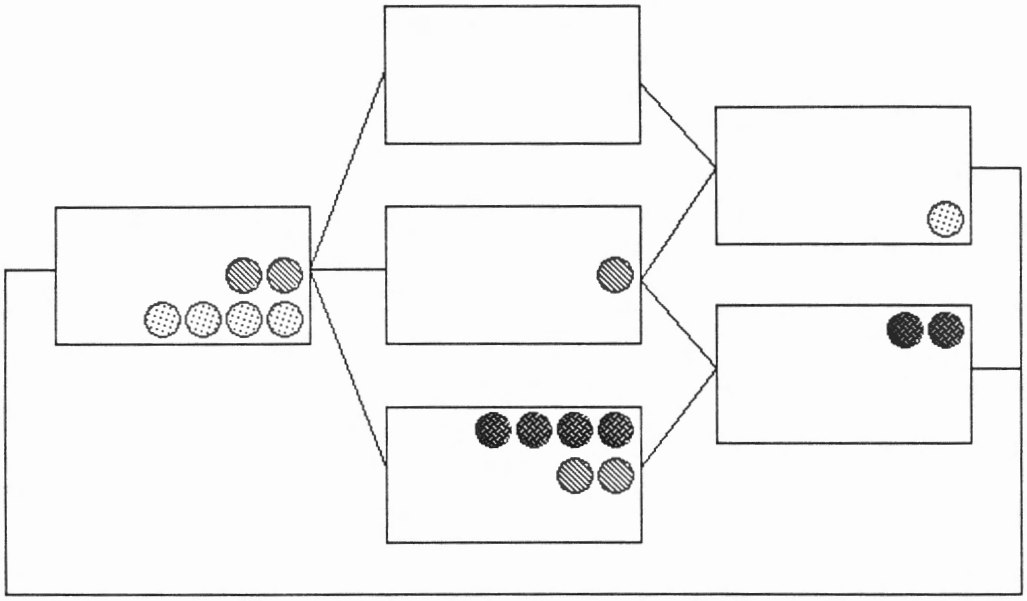

Figure 13. An application view showing 6 three-queue icons

The final view used with this data is shown in Figure 13. Each box is a processor, and each processor contains three queues represented as a row of six circles. As the simulation advances, each processor and included queues maintain the correct presentation of its state.

6.2.1. Basic Queue

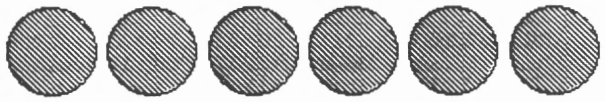

Figure 14. A basic queue icon

We first created a basic queue icon that represents the queue as a linear sequence of circles. When the queue is empty, no circles are visible. As the queue fills up, more circles are displayed.

After adding the six circle primitives to the new icon, we edited the maps on all of the circles' visibility attributes. By placing a *numeric-threshold-visibility* map on each attribute, a new attribute could control all the circles' visibility. By incrementing the threshold on each successive map, the number of circles displayed would grow. Figure 15 illustrates one of the circle's visibility maps. This map indicates that when the input value is 6 or more this primitive will be visible.

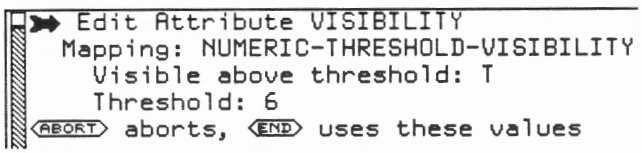

Figure 15. Editing the attribute **VISIBILITY** of one of the basic queue's circles

There are two important new attributes for this icon. The attribute *queue-value* will be used to constrain the visibility of the circles to represent the queue length, as described above. After modifying each of the visibility maps, we can then edit the attribute constraint for this new attribute as shown in Figure 16. All visibility attributes (now with an input type of *numeric*) are added to the list of attributes to compose.

The second important new attribute describes the color of the queue. A designer using the Graphics Editor should easily be able to specify the colors for all of the circles in the queue. After adding the new attribute *queue-color*, the designer edits its constraint as illustrated in Figure 17. The icon designer first makes the attribute type to be *color-attribute*. Then all of the circles' colors are constrained by this attribute. Now when the interface designer specifies the color via queue-color, it will be propagated to all of the composed primitives.

This completes the specification of the basic queue. We will now show how this new icon is incorporated into a more complex icon.

```
Edit Attribute Constraint <Compose <CIRCLE-1 VISIBILITY>...>

Edit QUEUE-VALUE Attribute Constraint
  Constraint Type: State Visibility  Composition
  Type of attributes: NUMERIC-ATTRIBUTE
  Attributes to compose: <CIRCLE-1 VISIBILITY>, <CIRCLE-2 VISIBI..
  Mapping: STATIC-NUMERIC
    Value: 6
<ABORT> aborts, <END> uses these values
```

Figure 16. Editing the constraint for the new attribute **QUEUE-VALUE** on the basic queue

```
Edit Attribute Constraint <Compose <CIRCLE-1 COLOR>...>

Edit QUEUE-COLOR Attribute Constraint
  Constraint Type: State Visibility  Composition
  Type of attributes: COLOR-ATTRIBUTE
  Attributes to compose: <CIRCLE-1 COLOR>, <CIRCLE-2 COLOR>...
  Mapping: STATIC-COLOR
    Value: RED
<ABORT> aborts, <END> uses these values
```

Figure 17. Editing the constraint for the new attribute **QUEUE-COLOR** on the basic queue

6.2.2. Three Queue

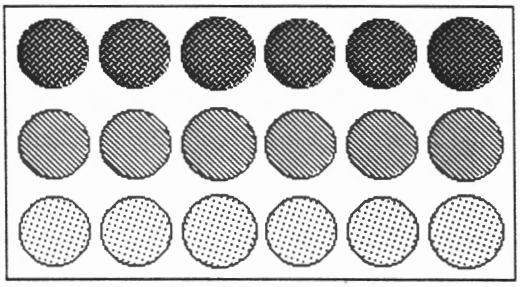

Figure 18. A three-queue icon

The previous steps define the basic queue icon. We can now use it as a primitive in a *three queue* icon. In this icon we include three basic queues as primitives to represent the top, middle, and bottom queues. We must now specify the new

attributes that will provide access to the values and colors for each separate queue.

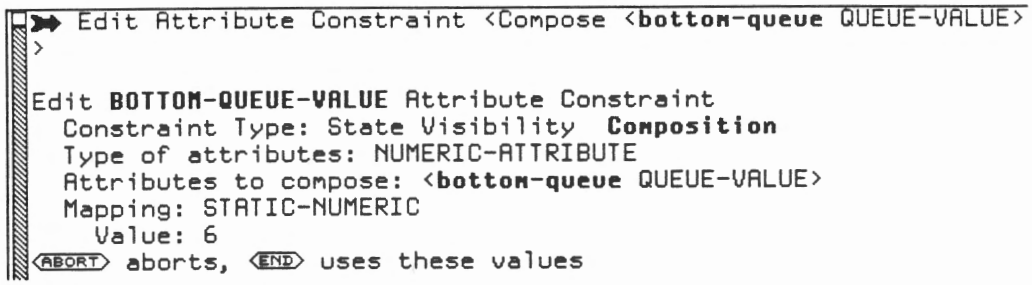

Figure 19. Editing the constraint for the new attribute **BOTTOM-QUEUE-VALUE** on the three queue icon

Editing the value for the bottom queue is shown in Figure 19. The attribute type is NUMERIC-ATTRIBUTE, and the attribute constrained is the *queue-value* of the *bottom-queue*. The new attribute that will constrain color of the *bottom-queue* is shown in Figure 20. It is a *color-attribute* and constrains the bottom queue's colors. We repeat this for each of the basic queues of this new icon.

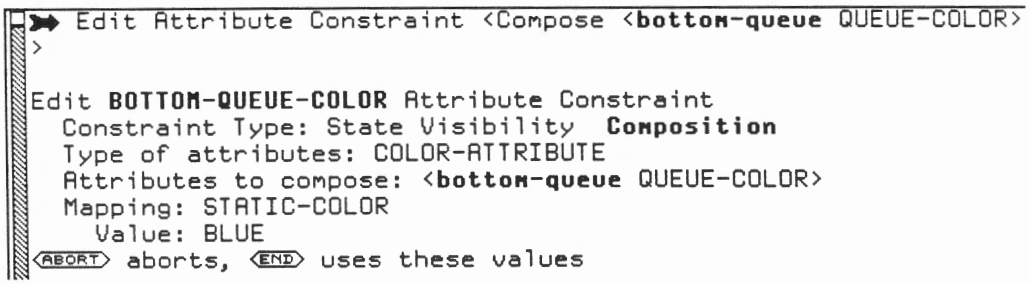

Figure 20. Editing the constraint for the new attribute **BOTTOM-QUEUE-COLOR** on the three queue icon

We add a line around the outside extent of the three queue icon to finish the graphic display of this icon. This icon is now available within the Graphics Editor, and we can build a view with many multi-queue icons to visualize our domain task. We tap the queue-values to the appropriate variables in our simulation and watch as the various queues at each node reflect the behavior in the simulation. It is interesting to note that when the developers of the simulation came to discuss the interface, their attention was immediately drawn to the application and how the queues were performing in the simulation. The interface fell away, and they were able to pursue their task.

6.3. Current Implementation

We currently have a working prototype of the Icon Editor. We are just beginning to understand the task of icon construction, and as our understanding develops we are evolving our interface to better satisfy the needs of this task. The current interface allows its user access to almost all the power of the ideas discussed in the previous sections, but in some cases too much of the underlying implementation is visible. However, our design philosophy includes providing access to the underlying substrate.

The current implementation runs on the Symbolics family of processors and is written in Symbolics Common Lisp using the Flavors object-oriented programming facility. Both color and black-and-white interfaces are supported.

The use of an object-oriented programming paradigm has greatly facilitated the development of this system. The power of the object-oriented approach allows the combination of inherited behaviors and characteristics for the icons, their attributes, and maps. For example, icons are implemented as objects. A particular class of icon, say circle, inherits most of its properties from basic icon classes which deal with many housekeeping functions. Each new class of icon provides methods for the generic functions: *draw, to-show,* and *probe.* The icon's draw method controls the icons appearance on the screen, while the to-show method details the input characteristics of the icon, and the probe method details the interaction of the icon and the application.

However, the use of object-oriented programming is not sufficient as the applications and interface become more sophisticated. We are moving toward a knowledge-based approach to the representation of icons and the task of interface construction itself. This facilitates integration with other components within the HITS environment and enables assistance in the creation of iconic interfaces.

7. Future of the Icon Editor

The Icon Editor provides a means for exploring the specification of dynamic graphical behaviors. As more interfaces are built with the system our methodology will be strained, helping us to better understand the limits and coverage of this approach. In this section we discuss the short and long range plans for the Icon Editor, evolving to a better understanding of interface creation and relating an application to an interface.

7.1. Improvements

A number of issues need to be addressed concerning the day-to-day problems of the designer. These issues range from substrate insufficiencies to improving tool interfaces.

View and Icon Independence

Currently there are two paradigms of interaction between an application and its graphical interface. In one the view controls the icons. In this approach commands address the view to manipulate its icons. In the other approach each icon supports its own editing. Here the application must manage the set of icons that is its interface. Merging these two techniques will not only provide the functionality of views for new applications but also allow the ease of direct interaction. Detailed issues here include: the storage and retrieval of these icons, what process controls the underlying application, updating the icons visual presentation, how that process knows about the icons, and switching between editing and interacting based on context.

Additional Primitives, Attributes, and Behaviors

We have on the drawing board additional primitives, attributes, and behaviors that will greatly enhance the effectiveness of an interface. A number of these ideas will expand the icons' range into additional modalities. An audio icon, for example, could be used as part of an alarm icon. With the use of a numeric-threshold-audio map an audio signal could be generated when input exceeded a specified threshold. Another area of active investigation is the integration of still video and video segments. Dynamics, such as flow, can be represented using color table animation. We are working to provide icon builders with a palette of animating colors to make this behavior available. More prosaic additions include primitives to build ticklines and rotary indicators, like needles.

Repetition and Conditionalization

Systems that deal with abstract issues through concrete representations require effective concrete realizations. Repetition is a notion that is required in interfaces, and simple implementation provide tedious interfaces. An analysis of the use of repetition with our systems has revealed that repetition usually occurs from attempting to represent an application's regular data structure. We have implemented a *multi-icon,* which solves this portion of complex data structure representation. A user lays out a multi-icon in the standard manner. In critiquing the icon, the user can specify the type of icon to represent each element of the data structure and a layout in one or two dimensions. When tapping, the icon provides alternative mappings between the data structure and the elements of the icon.

Redisplay optimization

In complex scenes automatic optimal redisplay of changing graphical properties is a difficult problem. The system must keep track of complex interactions between and internal to icons. The current system uses a combination of user specified partial ordering of component drawings and a set of heuristics to compute redraw candidates. Additional work is needed to fully handle all cases. Another approach is to use a high-powered graphics workstation that can update the display in real time based on the declarative state of the icons.

A Map Editor

As we move to a more knowledge based approach, it is increasingly more important to avoid mixing application and interface knowledge. The risk of this confluence is greatest in maps. To see this, consider a user wanting to show in the interface a complex combination of a number of characteristics in the application. A user could write a complex map to accomplish this, but in most cases this would be wrong. The correct approach is to define this new combination in the application, since it almost certainly derives its semantics there. Secondly, by putting it in the application's knowledge base, knowledge base sources, such as advising, have access to it as well as allowing views from additional perspectives.

Our interest is in providing a large coverage without encouraging the wrong level of abstraction. For the special case of discrete maps, we understand how to build the Lisp instructions from a higher level specification but still at the map level. In some cases it is possible to supply an interface at a much higher level. For instance, the specification of the state visibility constraint in designing new attributes actually builds maps, though the user is unaware of the process. We need more general tools and conceptualizations to support this higher level view.

Improved Interface

Our primary focus has been to provide as much functionality to the icon designer as possible. To this end the interface itself has been neglected. We envision a much better interface to our tool, one that focuses more on the task and less on the underlying implementation.

Portability

Lately, portability has been a topic of concern. We have looked at moving toward a more industry standard platform. This would mean Common Lisp, CLOS, and an X window substrate. If our code ran in these standards, we would be closer to running in many different workstation environments.

7.2. Directions

We believe that systems in the future will not only be reactive but will actively aid in the task being jointly solved by the computer and its user. In order to support the user, these systems will rely heavily on knowledge bases. These knowledge bases will contain information on the application, the user, and the interface itself. Our next major effort is to provide an underlying substrate of knowledge about the Icon and Graphics Editors and interface design in order to support the user in building more effective interfaces.

We intend to represent the graphical knowledge that will classify icons and relationships between the icons as they are utilized within an interface. Currently, the Icon Editor is being integrated with CYC, a knowledge base language used throughout HITS By sharing this common knowledge base substrate, we can incorporate the other technologies developed in the lab.

Being part of a larger group, we are in a unique position to create an environment that will support the user. We can draw on the tools from within HI to provide advisory capabilities, natural language support, and sketch recognition. This integration will provide more leverage than merely having graphical knowledge available. Likewise, we will be able to support others with the specific knowledge about the graphics used in the interface.

One direction we are taking is the integration of our efforts with the neural net sketch recognition capabilities. A new class of interface building tools can be built by providing the user with sketching as a way of specifying the icons to be added, editing procedures to be invoked, and assisting in library lookup for unknown icon types.

As well as drawing on knowledge, the graphics tools can augment the knowledge base. As an interface is specified either by sketching or from mouse interactions, a description of the interface can be constructed by the system to be used by other HITS tools. Knowledge of new icons can be acquired easily and incrementally added to the knowledge-base while using the Icon Editor.

Knowledge incorporated in graphics tools can assist in the design process. With graphic design knowledge encoded, a user can be supported with design critiques. Advice can be given to suggest alternative layouts or more effective designs, allowing the user to see specific instantiations of solutions moving more rapidly toward the most appropriate design. This set of design assistance tools will interactively process the graphic interfaces generated by the Icon and Graphics Editors and advise on and demonstrate, in context, more effective graphical presentations.

8. Conclusion

We have shown how behaviors can be specified for new icons and have presented an implementation of a prototype that enables users to create a new icon and use it. We intend to continue to explore interface issues in our evolving platform for HITS. We plan to increase the functionality of our current interface for building icons by making available additional graphic behaviors and building additional tools to ease the construction of new icons.

As we increase the functionality of the Icon Editor, it will provide the graphics necessary to support users of the HITS environment. In addition, these tools will cooperate in the construction of a common knowledge base about the interface itself. This common knowledge substrate will help drive the interface and support the user in an integrated suite of tools. These efforts will, we believe, raise interesting issues concerning the support of users in the interfaces of future systems.

9. Acknowledgments

We would like to thank the members of the technical staff for their support and comments in the creation of the Icon Editor as a prototypical tool in the Human Interface Tool Suite. We would especially like to thank Tim McCandless for his efforts in applying it to other components of HITS, making them more *inspectable*. Most of all, we would like to thank Jim Hollan for his vision of what HITS could be and for creating the kind of environment in which research and good ideas can flourish.

10. Bibliography

Borning, A., Defining Constraints Graphically, *Human Factors in Computing Systems* April 1986, 137-143.

Foley, J. D., McMath, C. F., Dynamic Process Visualization *IEEE Computer Graphics and Application* March 1986, 16-25.

Gould, L., Finzer, W., *Programming by Rehearsal*, Xerox Palo Alto Research Center Technical Report SCL-84-1. May, 1984 (excerpted in Byte 9(6) June, 1984).

Hollan, J. D., Hutchins, E. L., Weitzman, L., Steamer: An Interactive Inspectable Simulation-Based Training System *AI Magazine*, Vol. 5 No. 2, 1984, 15-28. (reprinted in *Artificial Intelligence and Instruction*, (Ed.) Greg Kearsley, Addison-Wesley, 1987).

Hollan, J. D., Hutchins, E. L., McCandless, T. P., Rosenstein, M., & Weitzman, L., Graphical Interfaces for Simulation *Advances in Man-Machine Systems Research*, Vol. 3, (Ed.) W.B. Rouse, Jai Press, 1987.

Martin, G., Avery, J., Pittman, J., personal conversations about the capabilities of neural net sketch recognition.

Members of the Human Interface Laboratory, *Demonstrations of HITS 1.0: The Human Interface Tool Suite*, MCC Technical Report ACT-HI-116-89-P, March 1989.

Myers, B. A., *Visual Programming, Programming by Example, and Program Visualization: A Taxonomy*, CHI'86 Conference Proceedings, April 1986.

Myers, B. A., Buxton, W., Creating Highly-Interactive and Graphical User Interfaces by Demonstration, *Computer Graphics* Vol. 20 No. 4 1986, 249-258.

Weitzman, L., *Designer: A Knowledge-Based Graphic Design Assistant*, Institute for Cognitive Science Report 8609, July 1986. (Reprinted in MCC Technical Report ACA-HI-017-88, January 1988).

Weitzman, L., Rosenstein, M., Winkler, A., *The HITS Simulation Environment*, MCC Technical Report ACT-HI-120-89, May 1989.

Appendix A
Attribute Types

The following is the list of attribute types that are currently available. The attribute types place restrictions on the type of values that an icon primitive can take. Each attribute type is listed with the values and/or restrictions on that value, the default value, and the maps that map to this value type. The map types are described in Appendix B.

Binary-Attribute Binary attributes have the values of 0 and 1. The default value is 0.

Maps that map to this type: STATIC-BINARY MOUSE-DISCRETE-BINARY

Color-Attribute Color attributes consist of the basic colors available. These colors include :RED :GREEN :YELLOW :BLUE :CYAN :MAGENTA :BLACK :DARK-GRAY :LIGHT-GRAY :WHITE. The default color is :WHITE.

Maps that map to this type: STATIC-COLOR NUMERIC-LINEAR-COLOR NUMERIC-THRESHOLD-COLOR SECURED-WARMUP-OPERATING-COLOR OFF-LOW-HIGH-COLOR LOGICAL-COLOR BINARY-COLOR OFF-ON-COLOR MOUSE-DISCRETE-COLOR

Horizontal-Justification-Attribute
Horizontal-Justification attributes are used for the placement of text. The legal values are :Left :Right :Center. The default is :Left.

Maps that map to this type: STATIC-HORIZONTAL-JUSTIFICATION MOUSE-DISCRETE-HORIZONTAL-JUSTIFICATION

Location-Attribute The location attribute is an x y coordinate pair. The default values are taken from the initial position of the icon.

Maps that map to this type: STATIC-LOCATION NUMERIC-LINEAR-LOCATION

Logical-Attribute Logical attributes include NIL and T. The default is NIL.

Maps that map to this type: STATIC-LOGICAL MOUSE-DISCRETE-LOGICAL

Numeric-Attribute This attribute is the corresponds to the Common Lisp type for number. The default is 0.

Maps that map to this type: STATIC-NUMERIC NUMERIC-AFFINE-NUMERIC MOUSE-NUMERIC

Off-Low-High-Attribute

This attribute is for the discrete class of values :OFF :LOW :HIGH. The default is :OFF.

Maps that map to this type: STATIC-OFF-LOW-HIGH MOUSE-DISCRETE-OFF-LOW-HIGH

Off-On-Attribute This attribute is for the discrete class of values :OFF :ON. The default is :OFF.

Maps that map to this type: STATIC-OFF-ON MOUSE-DISCRETE-OFF-ON

Overlay This attribute type is used with bitmap icons when one bitmap is overlaying another. Special attention must be addressed to the erasing and drawing sequences. The legal values for this attribute are NIL and T. The default is NIL.

Maps that map to this type: STATIC-OVERLAY MOUSE-DISCRETE-OVERLAY

Secured-Warmup-Operating-Attribute

This attribute defines the discrete class of values :SECURED :WARMUP :OPERATING. The default value is :SECURED.

Maps that map to this type: STATIC-SECURED-WARMUP-OPERATING MOUSE-DISCRETE-SECURED-WARMUP-OPERATING

Size-Attribute This attribute defines the size of the icon. The default size is .1 .1 but this is overridden when the user creates and sizes a new icon. The size is taken from the user input.

Maps that map to this type: STATIC-SIZE NUMERIC-LINEAR-SIZE

Text-Attribute This attribute is a text string that can be used in an icon. The default is the empty string (i.e., "").

Maps that map to this type: STATIC-TEXT NUMERIC-FORMAT-TEXT

Text-Font-Attribute This attribute is used to determine the font type. The legal values are :SMALL :MEDIUM :LARGE :MEDIUM-BOLD :MEDIUM-ITALIC :VERY-LARGE. The default is :MEDIUM.

Maps that map to this type: STATIC-TEXT-FONT MOUSE-DISCRETE-TEXT-FONT

Visibility-Attribute This attribute is for the visibility of the icon. The legal values are :VISIBLE and :INVISIBLE. The default is :VISIBLE.

Maps that map to this type: STATIC-VISIBILITY STATE-DISCRETE-VISIBILITY NUMERIC-THRESHOLD-VISIBILITY OFF-ON-VISIBILITY LOGICAL-VISIBILITY BINARY-VISIBILITY MOUSE-DISCRETE-VISIBILITY

Appendix B
Map Types

The following is the list of map types that are currently available in the system. Each map is presented with its input values, its output values, the transformation from input to output, and a default value when instantiated.

Color Maps

Binary-Color
Type of input value: Binary
Type of output value: Color
Mapping transformation: ((0 . :Red) (1 . :Green))
Default value: 0

Logical-Color
Type of input value: Logical
Type of output value: Color
Mapping transformation: ((Nil . :Red) (T . :Green))
Default value: Nil

Mouse-Discrete-Color
Type of input value: A mouse position
Type of output value: Color
Mapping transformation:
Default value:

Numeric-Linear-Color
Type of input value: Number
Type of output value: Color
Mapping transformation:
Default value:

Numeric-Threshold-Color
Type of input value: Number
Type of output value: Color
Mapping transformation:
Default value:

Off-Low-High-Color
Type of input value: Off-Low-High
Type of output value: Color
Mapping transformation: ((:Off . :Red) (:Low . :Yellow) (:High . :Green))
Default value: :Off

Off-On-Color
Type of input value: Off-On
Type of output value: Color
Mapping transformation: ((:Off . :Red) (:On . :Green))
Default value: :Off

Secured-Warmup-Operating-Color
Type of input value: Secured-Warmup-Operating
Type of output value: Color
Mapping transformation: ((:Secured . :Red) (:Operating . :Green) (:Warmup . :Yellow))
Default value: :Secured

Static-Color
Type of input value: Color
Type of output value: Color
Mapping transformation: None
Default value: White

Size Maps

Numeric-Linear-Size
Type of input value: Number
Type of output value: Size
Mapping transformation: Linear transformation between minimum (0) and maximum (10) values to minimum and maximum sizes.
Default value: 0

Static-Size
Type of input value: Size
Type of output value: Size
Mapping transformation: None
Default value: (0 0)

Visibility Maps

Binary-Visibility
Type of input value: Binary
Type of output value: Visibility
Mapping transformation: ((1 :Visible) (0 :Invisible))
Default value: 1

Logical-Visibility
Type of input value: Logical
Type of output value: Visibility
Mapping transformation: ((T :Visible) (Nil :Invisible))
Default value: T

Mouse-Discrete-Visibility
Type of input value: A mouse position
Type of output value: Visibility
Mapping transformation: Discrete state transformation
Default value: :Visible

Off-On-Visibility
Type of input value: Off-On
Type of output value: Visibility
Mapping transformation: ((Off :Invisible) (On :Visible))
Default value: On

Static-Visibility
Type of input value: Visibility
Type of output value: Visibility
Mapping transformation: None
Default value: :Visible

Location Maps

Numeric-Linear-Location
Type of input value: Number
Type of output value: Location
Mapping transformation: Linear transformation between minimum (0) and maximum (10) values to minimum and maximum location.
Default value: 0

Static-Location
Type of input value: Location
Type of output value: Location
Mapping transformation: None
Default value: (0 0)

Numeric Maps

Mouse-Numeric
Type of input value: A mouse position
Type of output value: Number
Mapping transformation: A linear transformation (in x) between mininum and maximum values.
Default value: 0

Numeric-Affine-Numeric
Type of input value: Number
Type of output value: Number
Mapping transformation: An affine tranformation that maps a range of numbers to a new range.
Default value: 0

Static-Numeric
Type of input value: Number
Type of output value: Number
Mapping transformation: None
Default value: 0

Index

MICROELECTRONICS AND
COMPUTER TECHNOLOGY
CORPORATION

3500 WEST BALCONES CENTER DR.
AUSTIN, TEXAS 78759
(512) 343-0978