The HITS Icon Editor

The Specification of Graphic Behavior Without Coding

Mark Rosenstein
rosenstein@mcc.com

Louis Weitzman
weitzman@mcc.com

Human Interface Laboratory, MCC
3500 West Balcones Center Drive
Austin, TX 78759

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.

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. The Icon Editor, a HITS tool, is a platform for exploring the construction of dynamic graphical icons and the techniques for relating those entities to an application.

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* [11], 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.

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.

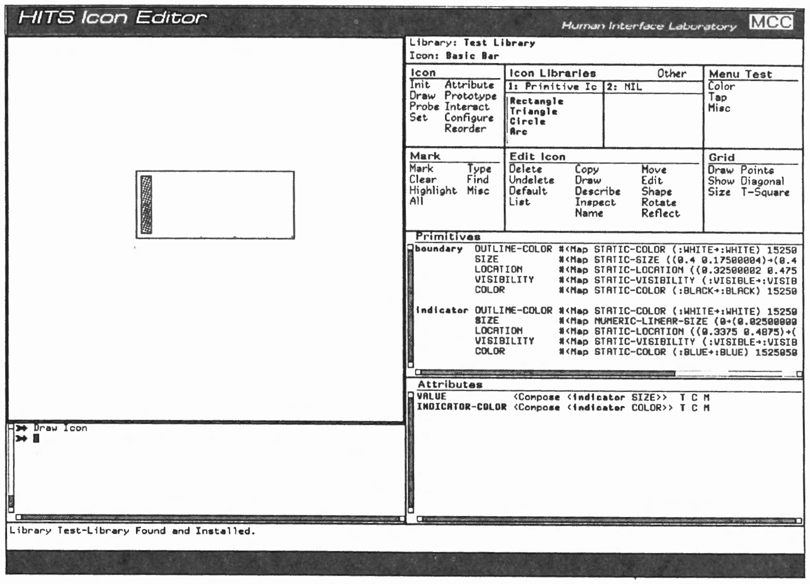
Interfaces constructed with HITS are targeted toward knowledge based applications. As computer systems move from mere calculation machines to true collaborators, both applications and interfaces must maintain representations of their structure and actions. Our tools aim to build next generation interfaces 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.

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



```
Library: Test Library
Icon: Basic Bar
```

```
boundary  OUTLINE-COLOR #<Map STATIC-COLOR (:WHITE→:WHITE) 15250
          SIZE          #<Map STATIC-SIZE ((0.4 0.17500004)→(0.4
          LOCATION      #<Map STATIC-LOCATION ((0.32500002 0.475
          VISIBILITY    #<Map STATIC-VISIBILITY (:VISIBLE→:VISIB
          COLOR         #<Map STATIC-COLOR (:BLACK→:BLACK) 15250

indicator OUTLINE-COLOR #<Map STATIC-COLOR (:WHITE→:WHITE) 15250
          SIZE          #<Map NUMERIC-LINEAR-SIZE (0→(0.02500000
          LOCATION      #<Map STATIC-LOCATION ((0.3375 0.4875)→(
          VISIBILITY    #<Map STATIC-VISIBILITY (:VISIBLE→:VISIB
          COLOR         #<Map STATIC-COLOR (:BLUE→:BLUE) 1525050
```

```
VALUE           <Compose <indicator SIZE>>  T C M
INDICATOR-COLOR <Compose <indicator COLOR>> T C M
```

```
➜ Draw Icon
➜ ▮
```

```
Library Test-Library Found and Installed.
```

Figure 1. The HITS Icon Editor

in future interfaces. For instance, the overall graphical style or natural language 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 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 an 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.

This paper discusses the specification of 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. Considered here are related efforts, the theoretical basis of our work, the current implementation, some examples of constructing icons, and finally, our future directions.

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 these 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 systems that use these techniques is provided in [8].

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 interface design (Myers' Peridot [9]) and education (Laura Gould's Programming by Rehearsal [3]).

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 a user interface which includes menus, scroll-bars, and light buttons.

Programming by Rehearsal does not use inferencing but allows 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 interactions of the performers in a production are defined mostly, but not exclusively, by having the system *watch* the designer perform the actions. 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** [1] 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** [2]. Instead of specifically laying out the behaviorial constraints of a new icon or stepping through an example interaction, this approach provides a mechanism to modify the specific attributes of the primitives. These attributes change as they reflect an underlying dynamic process. This external process drives the appearance of the interface. It is the constraints on the state of the variables within the external process 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 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.

Basis of the Icon Editor

A fundamental tenet 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 predefined attributes, 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, to identify their input type. 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 existing icon's attributes. Upon completion, instances of the new icon can be immediately used for testing, 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 of an icon 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.

Primitives

The basic mechanism for creating a new icon starts with the composition of icons from previously defined icon libraries. The working set of libraries is specified by the user and includes predefined libraries and libraries created through previous use of the Icon Editor. The most basic library is a set of hand-coded icons currently composed of rectangles, triangles, circles, arcs, lines, splines, text, and bitmap images. User defined libraries from the Icon Editor are generally organized by type, eg., button icons, or by project. The icon under construction is itself placed in a library indicated by the status line at the top of the Icon Editor frame.

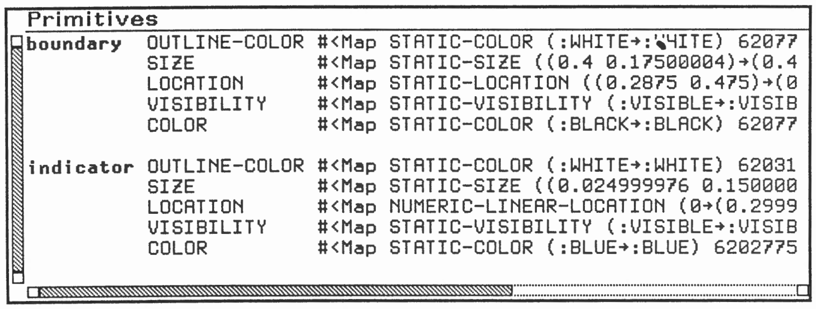
Composition of these components provides the structure for the new icon. An icon selected from a library is positioned and sized on the display screen. The appearance of the new icon is then the sum of these component icons.

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*.

In Figure 2, 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. As we shall see, it is the maps that provide the mechanism for dynamic change.

By default, icons created in the Icon Editor 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 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 via tapping, only



```
Primitives
boundary   OUTLINE-COLOR #<Map STATIC-COLOR (:WHITE→:WHITE) 62077
           SIZE          #<Map STATIC-SIZE ((0.4 0.17500004)→(0.4
           LOCATION      #<Map STATIC-LOCATION ((0.2875 0.475)→(0
           VISIBILITY    #<Map STATIC-VISIBILITY (:VISIBLE→:VISIB
           COLOR         #<Map STATIC-COLOR (:BLACK→:BLACK) 62077

indicator  OUTLINE-COLOR #<Map STATIC-COLOR (:WHITE→:WHITE) 62031
           SIZE          #<Map STATIC-SIZE ((0.024999976 0.150000
           LOCATION      #<Map NUMERIC-LINEAR-LOCATION (0→(0.2999
           VISIBILITY    #<Map STATIC-VISIBILITY (:VISIBLE→:VISIB
           COLOR         #<Map STATIC-COLOR (:BLUE→:BLUE) 6202775
```

Figure 2. The primitive inspector pane showing attributes that can be dynamically controlled

these attributes will be accessible 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.

Transformation Maps

Attributes parameterize graphical behavior. For example, a color attribute's value is used to determine the color of a part of an icon using a vocabulary of color values, like *red*, or *green*. Applications contain a separate language of numeric or symbolic values. In order to perform dynamic behaviors, an icon maintains a 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 into a size value.

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 numeric input moves between the minimum and maximum value, the size smoothly varies between its two sizes.

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.

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, a thermometer icon might 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], standard graphical mappings can be utilized to provide uniformity throughout a class of interfaces. 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.

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.

Consider the following example of a 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 propagate its value to the text-string attribute of the text icon.

This form of constraint is called composition. 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. When the designer adds this type of graph to a view and attaches its attributes to an application, one choice will be *x-label-axis*. The view designer can put in an appropriate text string, like *Neutrons per Hour*, for the bar graph label.

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-x* map (where x is an unspecified transformation and output type), and only colors will be generated by a *y-color* map (where y 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.

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.

Use of the Icon Editor

We currently have a working prototype of the Icon Editor. To demonstrate these ideas we will use two separate Icon Editor examples. The first example will highlight the construction of an icon to display a continuous range of values. 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.

Basic Bar

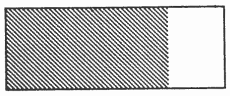
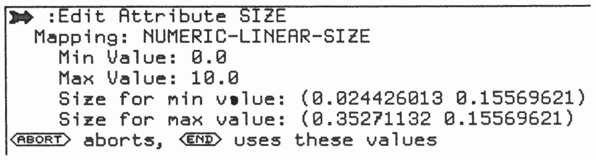


Figure 3. A basic bar icon

The *basic bar* icon displays a value by changing the size of its indicating bar to reflect its value. This icon only needs two primitives, a bounding rectangle which is the extent of the icon and an indicator rectangle which will change its size to indicate a value.

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. 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.

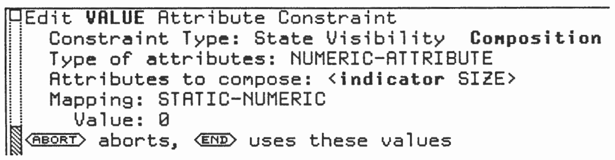
We first alter the default static map to be of the new type, *numeric-linear-size* map. As we change the map, the parameters that are unique to the map (i.e., minimum and maximum values and minimum and maximum sizes) are presented with their default values. The size values default to the existing size 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. The final edited menu is shown in Figure 4.



```
:Edit Attribute SIZE
 Mapping: NUMERIC-LINEAR-SIZE
   Min Value: 0.0
   Max Value: 10.0
   Size for min value: (0.024426013 0.15569621)
   Size for max value: (0.35271132 0.15569621)
ABORT aborts, END uses these values
```

Figure 4. 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.



```
Edit VALUE Attribute Constraint
  Constraint Type: State Visibility  Composition
  Type of attributes: NUMERIC-ATTRIBUTE
  Attributes to compose: <indicator SIZE>
  Mapping: STATIC-NUMERIC
    Value: 0
ABORT aborts, END uses these values
```

Figure 5. 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 5. 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.

Queuing Example

This example illustrates the capability of the Icon Editor to work in new domains. A technique to notice here is modular icon development by reuse of incrementally built 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.

The final view used with this data is shown in Figure 6. Each box represents a processor, and each processor contains three queues shown as a row of six circles. As the simulation advances, each processor and included queues maintain the correct presentation of its state.

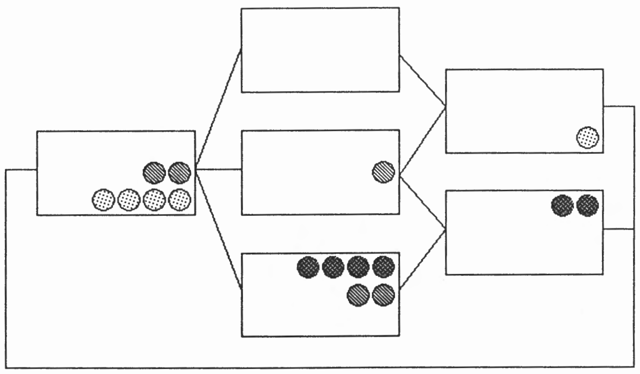


Figure 6. An application view showing 6 icons created with the Icon Editor

Basic Queue

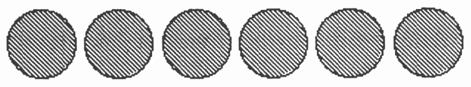
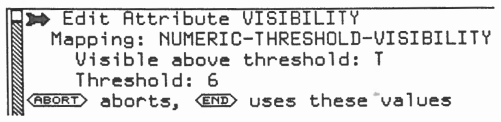


Figure 7. A basic queue icon

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.

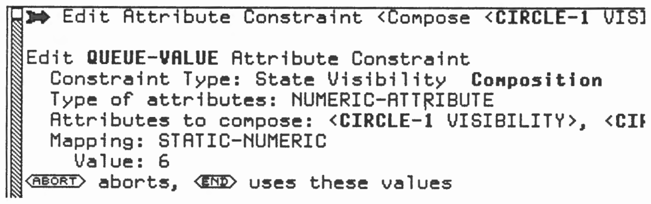
After adding the six circle primitives to the new icon, we edited the maps on all of the circles' visibility attributes. By placing a *numeric-threshold-visibility* map on each attribute, a single new attribute could control all the circles' visibility. By incrementing the threshold on each successive map, the number of circles displayed would grow. Figure 8 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.



```
Edit Attribute VISIBILITY
  Mapping: NUMERIC-THRESHOLD-VISIBILITY
    Visible above threshold: T
    Threshold: 6
ABORT aborts, END uses these values
```

Figure 8. 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 9. All visibility attributes (now with an input type of *numeric*) are added to the list of attributes to compose.

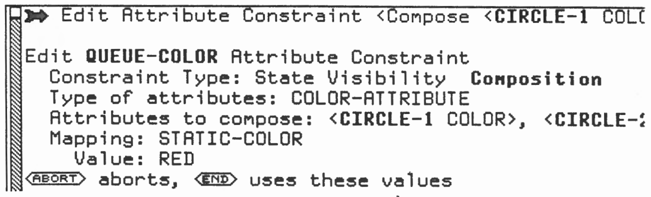


```
Edit Attribute Constraint <Compose <CIRCLE-1 VIS

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

Figure 9. Editing the constraint for the new attribute **QUEUE-VALUE** on the basic queue

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 another new attribute *queue-color*, the designer edits its constraint as illustrated in Figure 10. 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.



```
Edit Attribute Constraint <Compose <CIRCLE-1 COL

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

Figure 10. Editing the constraint for the new attribute **QUEUE-COLOR** on the basic queue

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

Three Queue

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.

Editing the value for the bottom queue is shown in Figure 12. 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 constructed similarly. We repeat this for each of the three basic queues of this new icon.

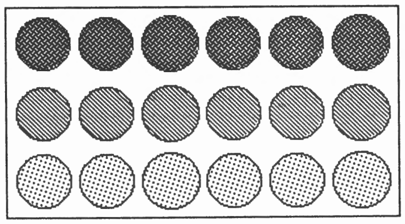
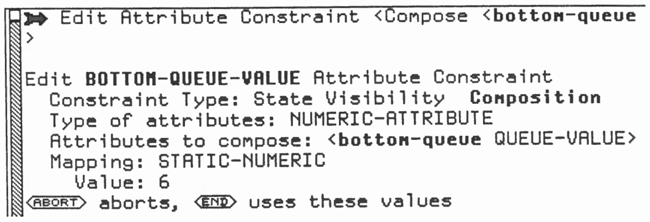


Figure 11. A three-queue icon



```
➽ Edit Attribute Constraint <Compose <bottom-queue
>

Edit BOTTOM-QUEUE-VALUE Attribute Constraint
  Constraint Type: State Visibility  Composition
  Type of attributes: NUMERIC-ATTRIBUTE
  Attributes to compose: <bottom-queue QUEUE-VALUE>
  Mapping: STATIC-NUMERIC
    Value: 6
ABORT aborts, END uses these values
```

Figure 12. Editing the constraint for the new attribute **BOTTOM-QUEUE-VALUE** 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. After invoking the command to define this icon, it is available within the Graphics Editor. We can now 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.

Future of the Icon Editor

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 Editor, the Graphics Editor, 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 a common knowledge base language used throughout HITS. By sharing this substrate, we can incorporate 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 provides 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.

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.

Bibliography

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

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

[3] 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).

[4] 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).

[5] 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.

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

[7] 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.

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

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

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

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