MCC Technical Report Number ACA-HI-406-88

AN INTRODUCTION TO HITS: HUMAN INTERFACE TOOL SUITE

James Hollan, Elaine Rich,
William Hill, David Wroblewski, Wayne Wilner,
Kent Wittenburg, Jonathan Grudin, and Members of the
Human Interface Laboratory

MCC Nonconfidential
December 1988

MCC
TECHNICAL
REPORT

MCC Technical Report Number ACA-HI-406-88

AN INTRODUCTION TO HITS: HUMAN INTERFACE TOOL SUITE

James Hollan, Elaine Rich,
William Hill, David Wroblewski, Wayne Wilner,
Kent Wittenburg, Jonathan Grudin, and Members of the
Human Interface Laboratory

MCC Nonconfidential
December 1988

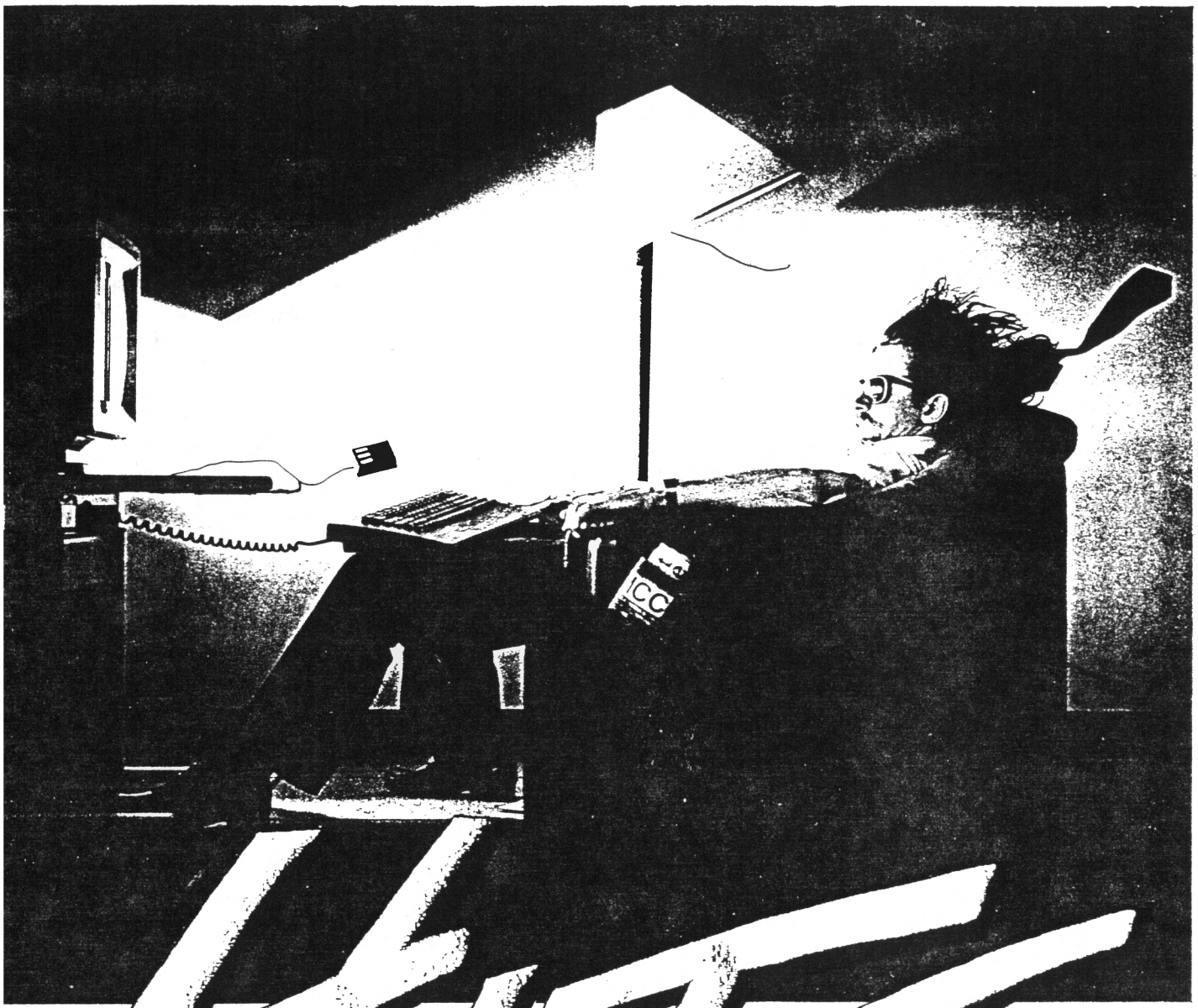
Abstract

Computers are the most plastic medium yet invented for the representation and propagation of information. They can mimic the behaviors of other information media and manifest behaviors not possible in any other medium. They provide us with interactive representational media of potentially revolutionary consequence. To realize the communicative potential of these new computationally-based media we must understand how to employ them to construct collaborative multimedia interfaces to high-functionality systems.

In this report, we introduce the *Human Interface Tool Suite*, an integrated set of tools for the construction of collaborative multimedia interfaces. Rather than documenting HITS as a user or reference manual would, we explain the ideas that motivate the HITS research programme.

Microelectronics and Computer Technology Corporation
Advanced Computer Architecture
Human Interface Laboratory
3500 West Balcones Center Drive
Austin, Texas 78720
(512) 343-0978

INTERFACE BY MCC



HITS

AN INTRODUCTION TO HITS: HUMAN INTERFACE TOOL SUITE

**James Hollan, Elaine Rich,
William Hill, David Wroblewski, Wayne Wilner,
Kent Wittenburg, Jonathan Grudin, and Members of the
Human Interface Laboratory**

Table of Contents

AN INTRODUCTION TO HITS: HUMAN INTERFACE TOOL SUITE

**James Hollan, Elaine Rich,
William Hill, David Wroblewski, Wayne Wilner,
Kent Wittenburg, Jonathan Grudin, and Members of the
Human Interface Laboratory**

A tool is something that constrains some of the degrees of freedom of a medium in order to manipulate the other ones better, and the computer is no tool in that sense; it is the ultimate medium, because its content is other media. *Alan Kay, 1982, House Subcommittee on Science, Research, and Technology.*

INTRODUCTION

Computers are the most plastic medium yet invented for the representation and propagation of information. They can mimic the behaviors of other information media and manifest behaviors not possible in any other medium. They provide us with interactive representational media of potentially revolutionary consequence. To realize the communicative potential of these new computationally-based representational systems we must understand how to fully exploit this *ultimate medium* to provide collaborative support for the solution of complex problems. This requires understanding how to construct collaborative multimedia interfaces to high functionality systems.

A multimedia interface is one that supports more than one medium through which users and computers communicate. Such an interface might support gestures, graphics, menus, natural language, sketching, speech, touch, and video. A collaborative interface is one that exploits knowledge about tasks, applications, interfaces, and users in ways that help users accomplish tasks effectively. Collaborative interfaces interpret ambiguous inputs correctly in context, phrase outputs in ways sensitive to users' situations, and provide advice on efficient ways to accomplish users' goals. To act collaboratively an interface must be integrated. Events and objects in one part of the interface must be accessible in the other parts so that tasks can be split across interface components as appropriate and still function with users in a collaborative and integrated fashion.

In this paper, we introduce *HITS* (Human Interface Tool Suite), an integrated set of tools now under development in the Human Interface Laboratory at MCC. HITS supports both the construction and run-time execution of collaborative multimedia interfaces. Rather than documenting HITS as a user or reference manual would, this paper explains the ideas that motivate the HITS research programme. The first section shows how our research programme leads us to concern ourselves with (1) tools to support the complete interface design cycle, (2) the role knowledge plays in the development of such tools and their integration, (3) a flexible

run-time execution scheme that supports multimodal interaction, and (4) a new metaphor, the notion of a *tool chain*, that mediates the way we think about interfaces and the tools used to construct them. The next section summarizes a set of research questions that we are addressing with HITS. The third section, Building Interfaces with HITS, presents the suite's anchor tools, characterizing in turn, tools for graphics, gestures, natural language, and collaborative aspects of interface design. The fourth section, An Example Application, exemplifies integration of HITS interface technologies in a knowledge editor application that includes knowledge-based display, graphical views, collaborative angels, and natural language. The fifth section, the HITS Blackboard, presents the innovative run-time architecture that makes possible the hallmark integration characteristic of HITS designed interfaces. The final section summarizes the report.

Before beginning a discussion of the HITS research programme, let's explore an example of the kind of interface that HITS is intended to produce. Consider the scenario depicted in Figure 1. In the first panel the user, a designer of copier interfaces, sketches a prototype copier control panel with a stylus on a flat interactive worksurface incorporated into the designer's desk. The system recognizes the sketch and turns it into a collection of dynamic icons. During the evolution of the control panel design, the system offers access to comments and suggestions concerning graphic design principles relevant to the developing interface. The user employs natural language phrases to describe and associate the components of the iconic interface with an underlying simulation model. Then by touching the the control panel the user specifies the number of copies to be made and activates the simulation by pressing the recently created *Start Button*. This scenario currently runs on an experimental interactive worksurface[1] in our laboratory. Figure 2 lists the HITS tools used to construct this demonstration[2], which makes use of many of the interface components of HITS described in this paper. These include neural nets constructed using our Gesture Editor, icons designed with our Icon Behavior Editor and configured with our Graphics Editor, collaborative advice constructed using our Advisory System and made available via Advice Angels, linguistic mappings made using Luke, and natural language understanding employing Lucy our natural language system.

HITS RESEARCH PROGRAMME

HITS is a research project being pursued in the Human Interface Laboratory at MCC. The lab's purpose is to develop the scientific and technological foundations for principled and efficient construction of collaborative user-centered interfaces. This mission requires balance and interaction between the laboratory's scientific and technological efforts. Science done in isolation can lead to irrelevance, to the development of toy systems, and to clever ideas that have no practical impact. Technological applications in the absence of sound theory can lead to clever gadgets, perhaps important for a particular job, but unlikely to generalize to new situations. A central tenet for the Human Interface Laboratory is the crucial importance of coupling scientific and technological efforts. Productive tension between the two contributes to a balanced research portfolio and advances our long range human interface research programme.

We coordinate research within the laboratory around the construction of an integrated interface design environment to leverage our efforts and to focus the laboratory on a scale of project appropriate to MCC. We envision the tools we are building as evolving from an integrated set of human interface tools (HITS) toward a general user interface design environment (GUIDE) with increasing amounts of intelligent support for the overall process of interface design. HITS and its evolution into GUIDE are experimental vehicles for grounding, motivating, and coordinating our scientific and technological efforts. They are intended to serve as prototypes supporting the rapid implementation, exploration, and demonstration of new human interface concepts.

---

[1]The interactive worksurface is a horizontally mounted plasma display bonded with a high-resolution transparent digitizer and incorporating infrared touch sensitivity.

[2]A video tape of this demonstration is available.

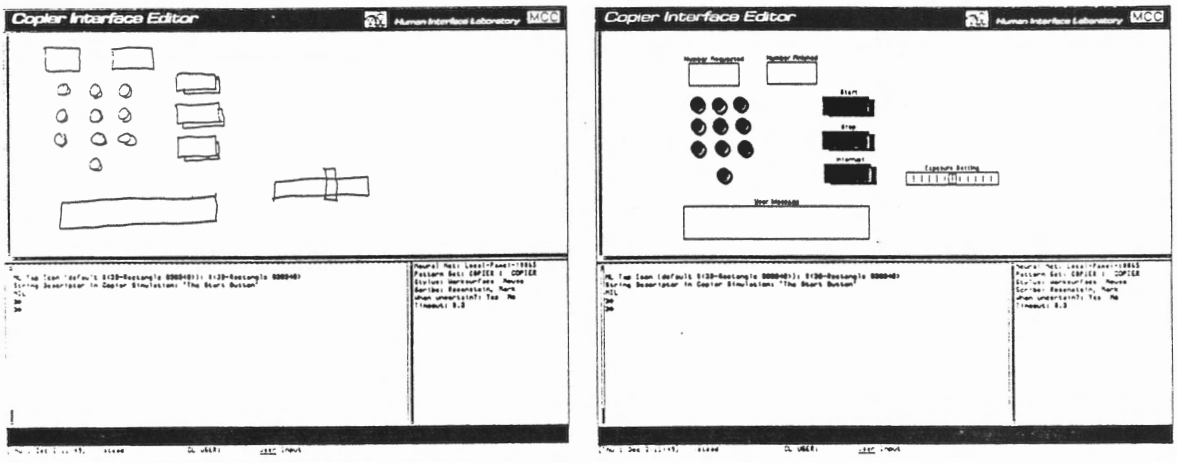


Figure 1: Copier Control Panel Sketch Interface Scenario.

Our work is motivate by the belief that the advancement of interface design necessitates understanding how to build collaborative interfaces. Collaborative interfaces increase people's productivity in computer-supported tasks by allowing them to work closer to their conceptions of tasks, freeing them from irrelevant computer-oriented details. Advanced forms of collaborative assistance will increase access to computer-mediated applications by a heterogeneous set of users and provide interfaces that allow richer exploitation of the powerful computational platforms of the future. For interfaces to be cooperative and adaptive, they must have representations of users' task, the languages of interaction, applications, and users. Thus, a substantial portion of interfaces to future knowledge-based and other high-functionality systems will themselves be knowledge-based.

A complete set of tools for building collaborative interfaces must support all phases of the interface design and construction process: design, implementation, run-time execution, and evaluation. Figure 3 shows these stages and suggests examples of tools that might be provided at each stage. Like many software design activities, interface design and construction involve iterative processes that rarely proceed in the linear fashion depicted in Figure 3. The delineation of phases is intended primarily as an aid to exposition and description of HITS.

Tools Throughout the Interface Building Process

Tools that address the first stage, design, require information about how people conceptualize the tasks supported by the interface, and how particular interfaces address those conceptualizations. Norman [1] refers to these two sides of the interface as the *User's Model* and the *System Image.* One way of understanding the human side of interfaces is to conduct protocol studies of people performing the tasks involved. Unfortunately, although such protocols can provide valuable insights, collecting the data and analyzing them is very difficult. Thus, protocol analysis tools are

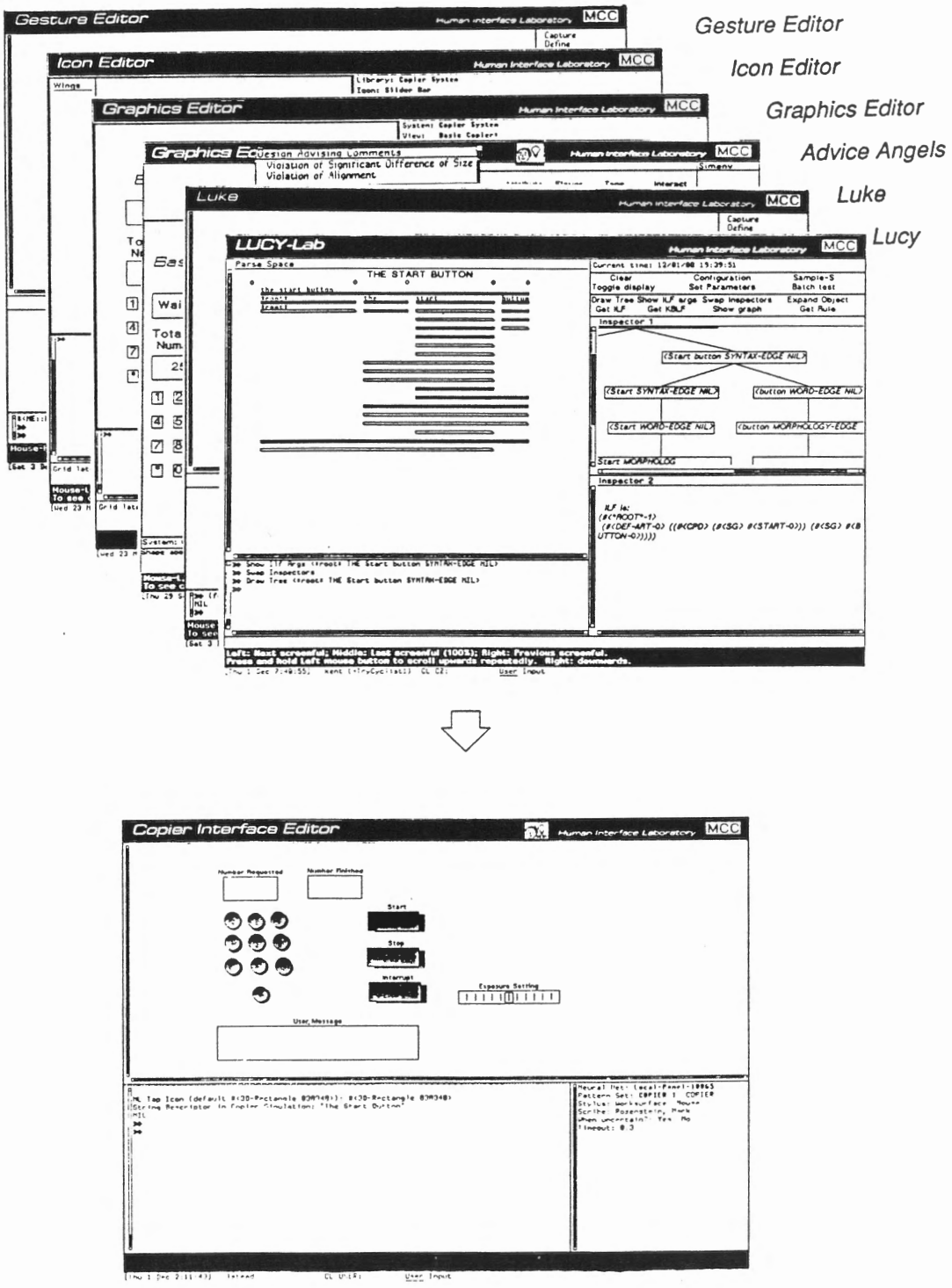


Figure 2: HITS Tools Used in Copier Control Panel Sketch Demo.

appropriate components of HITS.[3]

The implementation stage requires a set of tools that corresponds to the various facets of the

[3]Protocol analysis tools also have the potential of providing important support for the competitive argumentation [17] that underlies productive understandings of user tasks.

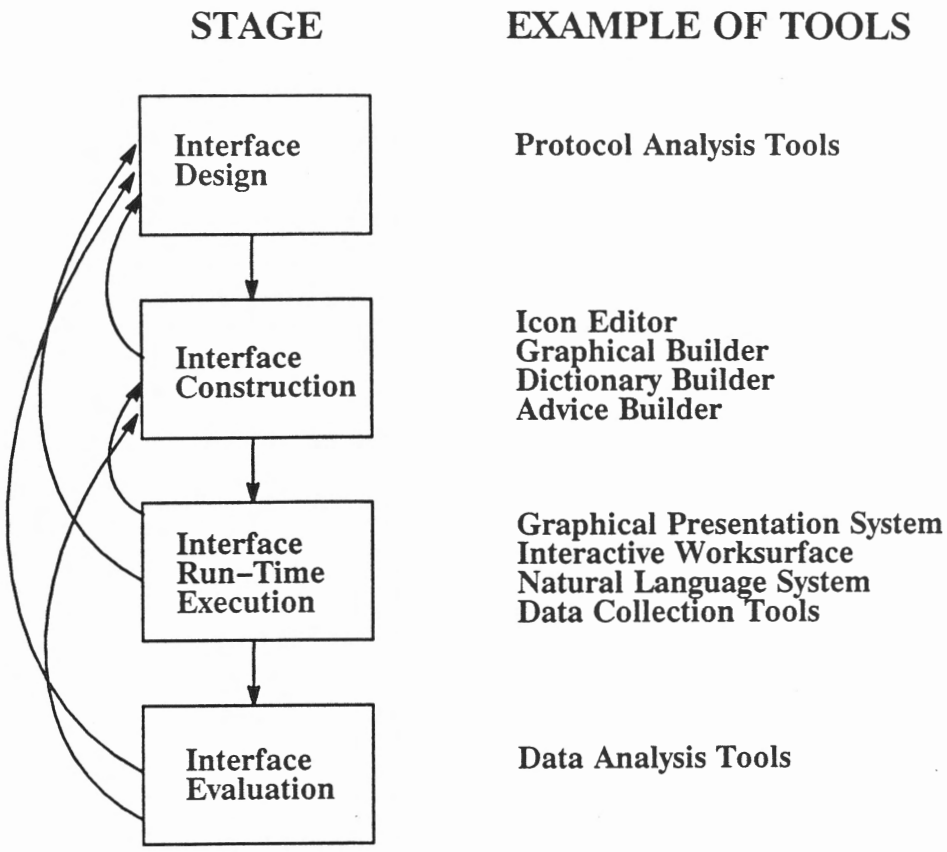


Figure 3: Tools Throughout the Interface Building Process

interface. For example, building the graphical parts of the interface requires tools both for the construction of graphical objects and their dynamic behaviors, for their association with underlying applications, and for their integration into a complete interface. Building the natural language part of the interface requires tools to support the construction of a dictionary that links words to representations of objects in applications as well as tools to facilitate construction of syntax, semantics, discourse, and pragmatics components. Building an advisory system requires tools that support the definition of advisory strategies, the construction of systems capable of planning in specific application domains, and methods for delivering advice and support to users.

The use of the completed interface requires run-time support for the various components of the interface. For example, the graphical part of the interface requires a presentation system that converts representations of graphical objects into device-level commands to draw objects and reflect appropriate dynamic state changes. If gestures are to be exploited as an input modality, then a system that recognizes and interprets them must be available at run-time. Natural language interaction requires a system that maps from natural language statements into an appropriate formal language (and back, if natural language generation is used). An advisory system requires systems that can execute the available advisory strategies to collaborate with users in accomplishing their tasks.

Interface evaluation requires data about the use of the interface. This stage can be facilitated by incorporating data collection facilities into the run-time system and by providing data analysis tools. In addition, the tools and interfaces that comprise HITS need to record the appropriate data about their use to permit evaluations of HITS itself.

The final significant aspect of these tools is their linkage to earlier stages in the interface definition process. Tools for interface construction should be able to check the implementor's actions against previously established design constraints. The actions taken during the use and evaluation of an interface can be similarly associated with the design and construction stages that make these actions possible. This eases the task of finding the parts of the design or implementation that need to be modified during redesign or reimplementation. These linkages, and the resulting power that they provide, are possible only if tools are developed as related parts of an integrated environment. HITS employs a state-of-the-art knowledge representation language (CYCL [2]) to provide the basis for integrated development and run-time environments.

THE ROLE OF KNOWLEDGE IN HITS

The need for an underlying knowledge representation system follows directly from three important premises underlying the design of HITS:

- **Integrated Multimedia Interfaces:** If we want an integrated multimedia interface, as opposed to an interface that happens to allow multiple forms of input and output, then the various modalities in the interface must be able to communicate with each other. For example, to support the scenario we described above, it must be possible for the icons that were created with the graphical interface to be accessible to the natural language system. This will happen naturally if all the interface components share a single formalism in which all accessible objects are represented in a unified way. We believe that a knowledge representation language is an appropriate formalism for this purpose, for reasons that will become clear as we continue.
- **Collaborative Interfaces:** If we want a collaborative interface, then we must provide the relevant knowledge to support reasoning about user actions. For example, if we want to give effective advice to users, then the advisory system must have access to knowledge about how to produce effective advice. If we want to be able to interpret ambiguous input such as, for example, English sentences, then we need knowledge about what it makes sense to say in a given context. Thus, in addition to the dynamic knowledge about interface objects, a collaborative interface must exploit a more static knowledge base about interfaces in general.
- **Interfaces to Knowledge-Based and High-Functionality Applications:** The most important use for the kind of interface we are describing is to support application programs that have a great enough range of functionality that simpler interface structures are inadequate to provide effective access to the complete system. We expect next generation systems to have precisely this property. Thus, we are focusing on knowledge-based and other high-functionality applications. This means that, in addition to the interface-specific knowledge that we have just described, a HITS-based interface can expect to access some useful domain knowledge, shared with the application program with which it is communicating.

The theme that emerges from these premises is the key role that knowledge plays, both in the tools that compose HITS and in the interfaces that the tools are used to construct. To be effective, this knowledge must be both integrated, to support communication among interface components and between the interface and the application, and modular, to make it easy to add specific information to support particular applications and their interfaces. Let's first consider the issue of integration.

Figure 4 shows a fragment of an integrated, application and interface knowledge base that might occur in an electronic CAD system. The shared knowledge base contains the concept of a resistor. Associated with that concept can be any number of basic facts (not shown in the figure) about resistors and about how resistors can be used in electronic circuits. These facts are then available both to the application program (the CAD system) and to the interface. So, for example, the fact that a resistor is a rigid physical object may be used both by the CAD system in enforcing design rules that prohibit multiple physical objects from being in the same place at once, and by

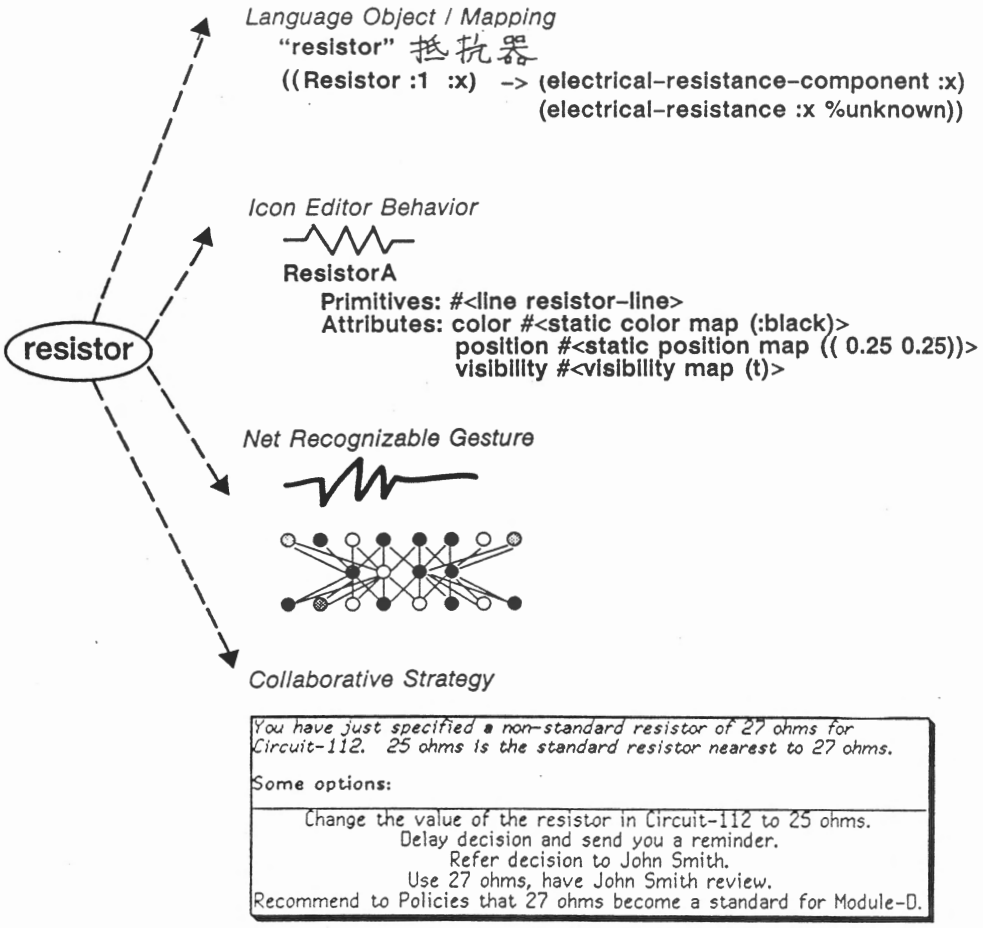


Figure 4: An Integrated Knowledge Base

the graphical display component of the interface in deciding on a screen layout that effectively displays a circuit. Similarly, both the CAD system and the advisory system can use knowledge about basic electronic properties of the resistor, such as Ohm's law.

In addition to knowledge that can be used by both the application and the interface, the integrated knowledge base must also contain representations that describes the view of a resistor from the perspective of each of the various interface components. Several examples of this are shown in Figure 4:

- To support a natural language interface, we need to know the words that can be used to refer to a resistor. In the figure, we show this for English and Japanese; other languages could, of course, be added. We also need to know that resistors are often referred to by a number that quantifies the amount of electrical resistance, e.g., '10K', instead of by a word. Electrical parts are often referred to by numbers and the magnitude of the number is frequently sufficient information for people to know whether the speaker means resistors, capacitors, or some other type of component.
- To support flexible graphical interfaces, we need to know standard symbology. Two standard symbols for resistor are a number followed by a capital Greek letter omega (Ω) and a zigzag of about seven line-segments. Users expect to write, say, read, and hear such symbols rather than the word 'resistor' or pictures of physical

resistors. The size, color, and orientation of such symbols as shown in an interface are likely to differ from the size, color, and orientation of resistors in the knowledge base. On the other hand, the position of such symbols may reflect the position of resistors in some larger context.

- To support collaboration, we need not only knowledge about how this system allows users to manipulate resistors, but also knowledge about how people think about resistors: the different ways they conceptualize them, the kinds of problems that can arise in the use of resistors in circuit design, and possible misconceptions about the roles and properties of resistors.

Returning now to the issue of modularity of knowledge in HITS, it becomes useful to look again at Figure 3 and consider knowledge from the point of view of when it is created and used. This reveals two categories of knowledge. The first is knowledge that supports interface design and implementation (boxes one, two, and four in the figure). It is, of course, possible to construct from scratch an interface of the sort we are considering by relying only on one's intuitions and experience as a source of good design principles and by hand-coding all of the necessary interface components. But the cost of constructing such an interface can be reduced and the quality of the resulting interface increased if relevant knowledge bases are contained within HITS and are exploited by the interface builder at interface construction time. A key theme of HITS is to make an increasing amount of this knowledge available to tool users. Examples of the kinds of knowledge that can be useful here are:

- **Presentation Knowledge:** how to depict information in context and to provide context sensitive manipulation methods.
- **Graphical Knowledge:** descriptions of icons, their behaviors, and graphical constraints.
- **Gesture and Sketch Recognition Knowledge:** mechanisms that allow the system to recognize gestures[4] and sketch-based forms of interaction.
- **Natural Language Knowledge:** rules that describe the syntax of a language and rules that describe systematic relationships between words in the language and concepts in a knowledge base.
- **Design Knowledge:** depictions of design knowledge of various types of interface techniques, such as graphical design principles.
- **Advising Knowledge:** general facts about effective advisory strategies and more specific facts about the interfaces constructed with HITS tools.
- **Data Analysis Knowledge:** general knowledge of statistical and descriptive procedures for analyzing data, as well as more specific knowledge about how different kinds of data might be used to evaluate interfaces implemented with HITS.

Because most of this knowledge is about interfaces, rather than about specific applications, most of it is portable across application domains. The generality of this knowledge means that one emphasis in building HITS is on the capture and representation of this application independent knowledge within HITS.

Continuing with the perspective of when interface knowledge is created and used, a second category can be identified. It contains knowledge that supports the run-time use of an interface (the third box in the Figure 3). Some of this knowledge is domain-independent, some is not. This knowledge can be further subdivided on the basis of when it is created. Static knowledge must be available at run-time but can be created any time prior to then. The domain-independent parts of this knowledge can be provided as part of HITS. For the domain-dependent parts, HITS

[4] In our system this is implemented using neural networks.

contains tools to assist in knowledge acquisition. Dynamic knowledge about a particular user and a particular user-system interaction must be created at run-time by the interface run-time system. HITS provides run-time modules that create this knowledge using representations that are consistent with the other knowledge bases that HITS provides and supports.

One way to summarize this overview of knowledge in HITS is to say that HITS supports an integrated knowledge base. The knowledge itself covers a broad range of topics, all of which impinge in significant ways on the design and use of a flexible interface. In addition, the role played by HITS with respect to the acquisition of this knowledge must be tailored to the generality of the knowledge itself. General, domain-independent knowledge is represented once and provided as part of HITS. Specific, domain-dependent knowledge cannot be provided that way. So, instead, the focus in HITS is on the construction of effective tools to support knowledge base development.

We will now step backwards through each of the first three stages[5] of Figure 3 and describe in more detail the knowledge and the tools that HITS contributes.

Supporting the Run-Time Execution of the Interface

The heart of HITS is a run-time system that supports collaborative multimedia interfaces. Components of HITS support the construction of this run-time system. We will talk first about the characteristics we require of run-time systems, and then, in succeeding sections, describe the HITS component tools that make such systems possible.

A HITS-based run-time system has two parts: a set of modules that perform actions in the interface and a set of knowledge bases that those modules rely on. The processing modules and their associated knowledge bases must be separable enough that they can be used relatively independently, since not all interfaces will require all of the interface modalities that HITS supports. Some interfaces may not need natural language, for example; others may not need graphics. But it must also be possible to integrate components so that a single, coherent interface can be presented to an end user. As we mentioned earlier, this integration is accomplished in HITS by tying all the components to a common knowledge representation system. We assume a hierarchically organized frame system in which the interface objects that can be manipulated are defined. This may be augmented with assertional methods that permit representation of facts about the current state of the application and the interface.

The kernel of the HITS-supplied run-time system is a dialogue manager that handles both the user's input and the system's output. This dialogue manager performs low-level handling of input and output, and routes what it sees to appropriate components of the run-time interface system. It also builds a history of the interaction, and makes this history available to those interface components that need it. It is important not only that this discourse history capture events in all the modalities within the interface, but that it either be implemented as a single integrated history or that separate discourse histories be able to communicate. Examples of ways in which discourse history are used include:

- **Understanding English Sentences.** Suppose the user says, "I just created a new icon. Now I want to copy it." To interpret the word *it* correctly requires that there be a history at least as long as this two sentence dialogue, since *it* must be recognized as referring to the icon that was mentioned in the previous sentence. Now suppose that the user simply says, "Copy the last icon I created." Assume that the user has been creating icons using the direct manipulation capabilities of a sketching interface like the one in the copier example. Now, to identify correctly the referent of the the phrase "the last icon I created," requires a dialogue model that also contains a record of the user's actions within the sketching system. These kinds of actions emphasize the need for a dialogue model that captures multiple interaction techniques.

---

[5]Specialized HITS tools to support the fourth state in Figure 3 have not been developed yet.

- **Understanding Gestures.** Suppose a user indicates with a gesture, perhaps a shake of a stylus or a finger, that the system has incorrectly identified a portion of a sketch. To respond appropriately to this it is necessary to have a model not only of the user's actions but also of the system's actions.
- **Generating Appropriate Advice.** Suppose a user asks, "What went wrong?" In order to answer such a question, it is necessary to be able to look at the last several interaction events and to explain the actions that the system took.
- **User Modeling.** This dialogue model should provide advisory strategies with information about the commands that are and are not understood by the user, based on the user's patterns of usage. In addition to user modeling the history should also support disambiguating actions for which multiple interpretations might exist, such as whether a "delete" request associated with a mouse click is meant to refer to a single icon or a larger configuration of objects.
- **Critiquing and Evaluating Tool Use.** The discourse history can be used to detect inefficient use of tools, either by the system (leading to coaching or critiquing), or by human protocol analysts.

Supporting Interface Construction

The only parts of an interface run-time system that HITS can provide directly are those that are domain-independent. Domain-dependent structures must be built for each new application interface, so the best support HITS can provide is a suite of tools that enable the efficient implementation of those structures.[6] Each tool in HITS is designed to support the construction of one facet of the interface. Just as the run-time systems in HITS must be both modular and integrated, so too must be the construction-time tools. HITS uses the same approach here as it did in the structure of its run-time support: it ties all the tools together through a single knowledge representation system. By doing this, we facilitate the construction of the sort of unified representation of application and interface knowledge demonstrated in the copier control panel example.

The fundamental notion behind all the HITS construction tools is that interface manifestations of objects are conceptually tied to the representation of those same objects in the application knowledge base in ways such as those shown in Figure 4. This suggests that the way to support building interface knowledge is to couple that process with the process of building the application knowledge base. There are several ways that this can be done:

- Support the explicit statement of interface knowledge by the interface builder. This approach, in turn, has two facets, both of which are currently used in HITS:
  - Build interface entities and allow the tool to build the associated application objects. This approach is illustrated by Pogo, a system described below for building graphical objects.
  - Build application knowledge base objects and allow the tool to build the associated interface entities. This approach is illustrated by Luke, another system that is described below.
- Support the implicit acquisition of interface knowledge as a task is being performed. This approach is illustrated by a neural net approach to the recognition of gestures, another system that will be described later in this section.
- Support users of HITS by providing design expertise which may complement their own knowledge and assist them in maintaining interface consistency. This approach

---

[6]We do envision that portions of new application interfaces will frequently be derived from copying and tailoring components from existing applications that are similar. See also the discussion of *Tool Chains* below.

is illustrated by Designer [8], a system for critiquing graphical views based on representations of graphical design principles.

If we consider this close coupling of the tasks of building an application knowledge base and of building corresponding interface knowledge bases, and if we also acknowledge the important role that application knowledge can play in a collaborative interface, it becomes tempting to view the entire knowledge base construction problem as an interface building issue. Thus, one approach we could take would be to include in HITS a general purpose knowledge editor and then both the application and the interface knowledge could be built at once. Because of this and because of the importance of knowledge editing to the design of HITS, we are building HITS around a general facility for editing knowledge bases. A user HITS can employ that system directly to create representations of interface knowledge. In addition, we are augmenting that system in several ways that are described below. One goal of these additions is to make it easy to create the kinds of interface mappings shown in Figure 4.

Supporting Interface Design

Fundamental to the design of effective interfaces is understanding the cognitive tasks that will be supported. This means that one must understand the particular tasks that users will employ an interface to accomplish as well as the larger setting within which those tasks are situated. In an important sense an interface is a reification of the designer's view of users' tasks and associated task settings. To take a *user-centered* approach to interface design means taking a deep look at users' real tasks, related tasks, and the larger personal and organizational contexts in which the tasks are situated.

Understanding users' tasks means having a detailed model of them and of users' conception of them at a level that permits an effective instantiation of interfaces to support them. The analysis of protocols collected from users of systems or from the performance of tasks that interfaces are being designed to support is a key technique to assist in constructing the detailed models required. The importance of protocols and the current lack of facilities to assist with their analyses motivates us to include a set of protocol collection and analysis tools within HITS. The view of protocol analysis being taken is that it is a form of competitive argumentation [17] in which one wants to provide structure to and annotation of a protocol. Tools are required to support maintaining multiple theories about the protocol [13], viewing of the analysis at multiple levels, and retrieving annotated segments that match flexible retrieval specifications.

Another aspect of supporting interface design is providing design expertise that complements that of users of HITS. An underlying premise is that each of the tools should incorporate representations about their own use and about types of expertise that users might find useful. For example, as mentioned above, the tool to help one form mapping rules between words in a natural language portion of an interface to concepts in a knowledge base has a representation of lexical semantics that permits it to offer alternative mapping rules as well as assist users in maintaining mapping consistency in a large knowledge base. As another example, graphical tools can be augmented with the ability to provide critiques according to graphic design principles. In all of these cases, the ability to support design is facilitated by our commitment to an underlying integrated knowledge base.

Supporting Interface Evaluation

The discourse history built from the sequence of events pooled across interface modalities can serve as a powerful data base for evaluating the use of HITS and the use of interfaces built with HITS, in addition to serving the run-time functions described elsewhere in this report. In addition to a history recorded by the system at run-time, protocols of user-system interaction may be recorded simultaneously by external means (such as video or audio taping); in addition, other data may be generated following an interaction, either by automated analysis programs or by a human protocol analyst.

Although the design of our protocol collection and analysis system is in early stages, the evolutionary nature of the HITS tool chain, described below, requires a flexible and extensible

approach to the handling of user-system interactions. Prior to run-time, the user of the protocol collection system may indicate which data is to be converted to the common format used by the facility and preserved permanently, where the run-time system would otherwise discard it. For example, the record of the lowest-level user actions -- mouse or stylus movements, button and key presses -- are of less interest to the run-time system than the higher-level actions they initiate. But at times this level of information is needed to understand user actions or misunderstandings, and under certain circumstances this level of "raw input" might be directed back into the system to "replay" a user session.

In examining an interaction, a protocol analyst may view, search, annotate, and categorize the collected data, perhaps carrying out statistical analyses and constructing alternative models of the interaction. Such analyses may aid in evaluating use of a tool or interface, or may provide knowledge of user tasks and common misunderstandings that will contribute to the construction of critiquing, advising, and user modeling systems.

Supporting Collaboration: Tool Chains and Collaboration Profiles

In addition to supporting the sequence of processes that must occur in the design of a particular interface, HITS supports the idea of a *tool chain*, in which general tools can be used to craft more specific ones, that can in turn be used to produce tools or to create specific application interfaces. The idea of a tool chain is a powerful one because it enables collections of activities to be done once for entire families of applications rather than requiring them to be duplicated for every member of the family. So, for example, a HITS user might use HITS to produce a new set of tools $HITS_{phys\text{-}sys}$, that is specialized for building interfaces to programs that simulate physical systems. This new tool set differs from HITS in that it contains a set of graphical icons that have been designed to display parameters of such simulations, appropriate words (such as momentum, mass, density, etc.) in its natural language lexicon, and knowledge about how to display complex simulations effectively using successively deeper levels of detail. Now another tool developer can use $HITS_{phys\text{-}sys}$ to create an even more specialized set of tools $HITS_{auto}$ for building interfaces to simulations of automobiles. This new tool set has been further augmented with icons and lexical entries for cars and their components. Now additional users can use $HITS_{auto}$ to create interfaces to various automobile simulation programs.

It is easy to see how the notion of tool chains can be applied to any application domain. For example, the basic HITS tools could be used to create a set of specially tailored tools for constructing interfaces for new automated teller functions (e.g. loan applications). One might first create a general $HITS_{Banking}$ containing a knowledge base of information appropriate to banking applications and a set of interface components that manifest the specific interface "look and feel" required by a given corporation. Notice that at each link in the tool chain we empower users with less interface design and system level knowledge with the ability to participate in the interface construction process. The tool chain notion fits well with our view of interface design as a multi-person and multi-disciplinary activity. Linkage of the tools also provides mechanisms to collect data about usage and to permit the tools and the interfaces they are used to construct to be more readily modified in principled ways.

Another idea supported by HITS is the notion of a *collaboration profile*. Each user of HITS can have a profile that influences the form of interaction with the system. This starts with a default profile and will evolve over time via modifications that the user makes to it as well as modifications the system makes as a result of the history of the user's interactions. It serves both as a user modeling component and as a communication mechanism between the user and system about the style of interactions with HITS.

HITS RESEARCH QUESTIONS

Limitations of current interfaces are the major impediments to fully exploiting computation in the conceptualization and solution of complex problems. We take as premises that the two major courses of interface development in the future will be toward much more collaborative and adaptive interfaces, interfaces in which tasks are more effectively shared between users and computational systems, and toward interfaces that provide facile access to immensely more functionality than is present in current systems. Interfaces of the future will not only provide collaborative support to individual users of high-functionality systems but will also facilitate collaboration between individuals. These premises lead us to focus our research on the development of flexible, natural, multimodal interfaces, interfaces that approximate the ease and richness of person to person communication, and on how such interfaces can be effectively developed. This in turn has led us to HITS and the construction of an integrated environment for interface design.

The research issues that we are addressing with HITS derive primarily from our interest in understanding how to develop collaborative interfaces to high-functionality systems. There are a host of specific research questions associated with each of the disciplines underlying our work that must must be answered if we are to be able to provide principled support for the design and implementation of integrated interfaces. Here we focus on those research questions that span the disciplines represented by the HITS effort. These include a core set of common issues that all interface modalities need to address. For example, to support collaboration the interface must make sense out of often highly ambiguous input data, whether the input data is a phrase in natural language or a gesture on our interactive worksurface. Similarly, the interface must present information in context sensitive ways that can be readily interpreted by users. Underlying these simple descriptions of collaborative interfaces are a set of difficult research issues. We are addressing a number of these in our research programme.

Multiple Active Interface Agents

We must support multiple active interface agents if the the interface is to be collaborative.

- We need to support N-way communication among the agents, one of which is the user, some of which are application programs, and some of which are the various interface agents. It's this issue that has primarily motivated our development of a flexible blackboard-based control structure for HITS.
- We need to support genuine mixed initiative dialogues. This is why we're concerned with a uniform treatment in the HITS blackboard of both input and output interactions and why the issue of processes and control is centrally important.
- We need to provide a coherent view to users of the various internal agents' individual communications. Although we might implement interfaces as a collection of independent agents, it must also be possible for the user to perceive the entire system as an integrated whole. Thus, we need to investigate strategies for managing coherent conversations.
- If we're going to support integrated multimodal interfaces then we need to allow the modalities to interact at multiple levels of granularity (as opposed just to, say, complete commands). This is another major motivation for the use of a blackboard structure. It is also an important motivation for the use of a common knowledge base. Each of these devices provides integration capability at several levels of granularity.
- We need a shared discourse history across agents if they are going to act in a coordinated way.

Tools for Effective User-Centered Interfaces

To understand our research agenda it is important to note that we're not just trying to build effective user-centered interfaces. We're trying to build a set of tools that will enable effective user-centered interfaces to be readily built. This has a number of implications. First, we need an ontology of interfaces and knowledge about interfaces. If we only wanted one interface, we wouldn't need to make this explicit. Having it in the heads of the human designers would be good enough. But to provide powerful design tools and to enable less skillful designer to build more effective interfaces, this knowledge must be explicit. Second, modularity becomes very important so that the right pieces selected from the building blocks provided by the general tool suite can be mixed and matched for each new interface. Third, we need to support the evolution of tool chains of more specialized HITS with specialized HITS knowledge bases and libraries.

Knowledge-Based Tools and Interfaces

Because of the importance of knowledge in both our tools and the interfaces that those tools help build, we cannot avoid dealing with questions of how large knowledge bases can be built and maintained. Thus, we must worry about supporting individuals in the knowledge base construction task. Our choice of a knowledge editor (HKE) as a demonstration platform for our work is motivated in large part by this concern. We must also consider supporting groups of people doing this task together. We need to support "corepresenting communities," which may span disciplines and experience levels.

Future Research Positioning

In addition to the research topics that we are addressing in our work on the individual modalities supported by HITS, there are a host of research questions that HITS will enable us to consider in the future. These include the following:

- Given that all of this technology is intended to make possible the construction of new kinds of interfaces, what are they particularly good for? How can we use this technology to build interfaces that help people get their jobs done? Which interface modalities support which kinds of communications? What's the best general way to support the interleaving of modalities?
- How can we provide efficient implementations of the general architecture that we are formulating? For example, how do we structure and index the blackboard and how do we specify control within the blackboard?
- What building blocks should go into the general tool set? For example, are there "parsers" for common kinds of languages where languages are broadly construed to include gestures? Are there knowledge sources for common ways of structuring dialogues? Are there domain-independent forms of advice?
- How can the individual modalities provide leverage for each other? For example, how might knowledge about current discourse context be used to influence the interpretation of a sketch?
- What real advantages does an integrated knowledge base provide?

BUILDING INTERFACES WITH HITS

HITS consists of an integrated suite of tools. In this section we present a selective overview of the tools oriented towards characterizing their functionality and the knowledge that they rely on and provide. In subsequent sections we describe a detailed application of HITS for knowledge editing and then explain the HITS blackboard system. The underlying knowledge base and blackboard system are the keys to the integration of HITS.

BUILDING GRAPHICAL INTERFACES

The most effective graphical and direct manipulation interfaces [3, 4] are typically highly specialized to particular applications. Their appearance and functionality are peculiar to specific domains. Consequently, graphical interfaces should be built by people who have expert understanding of what people who use a particular application require, perhaps in conjunction with graphics design specialists. Rarely are such experts programmers, but they currently need to be in order to implement new interfaces. This is even made more difficult because interactive graphical interfaces are very complex programs to write. Our solution is to provide libraries of simple graphical components, methods of building complex graphics by composing simpler components, and techniques for coupling graphical behavior of components in an interface to numerical behavior of state variables in an application. Menu-driven editors write code for pieces of interactive graphical interfaces according to the items people choose from the libraries.

Building Graphical Interfaces: Icon Editor

The individual boxes, buttons, borders, and backgrounds that make up a graphical interface[7] are built by a tool called the Icon Editor, depicted in Figure 5. We use the term "icon" in a very broad sense. All elementary objects that may appear in an interface are collectively called icons and the purpose of a session with the Icon Editor is to implement or refine these icons. Once entered in the knowledge base by the Icon Editor, an icon is ready to be instantiated in interfaces. Each icon has three kinds of characteristics:

- **structure**, what parts an icon has and how they are connected;
- **appearance**, what one sees when an icon is presented;
- **behavior**, how an icon acts to show state or to respond to user actions.

An icon's structure is defined to be either a primitive icon or, recursively, an assembly of icons. Primitive icons are those that are predefined in the knowledge base. They range from the typical set of graphical primitives (such as line, box, ellipse, and character string) to an *ad hoc* collection of more sophisticated icons (such as dial, graph, column, help button, camera, window, and constraint) whose potential usefulness is anticipated. Primitives can be combined into hierarchical assemblies, theoretically forming arbitrarily sophisticated icons. Because graphical interfaces are highly specialized, we expect that the most frequently used icons (including both visual and behavoral characteristics) will not be stock primitives, but custom-built. One feature that helps with customization is recursion: every assemblage of icons is usable as a component in more complex icons. In addition, we expect custom libraries of icons to continue to grow and be reused across interface applications.

Components of icons do not necessarily correspond to part-whole relationships found among application objects. For example, the interface entity for a resistor may contain a list of colors (that may indicate current through the resistor) while the application entity for a resistor may not have color at all.[8] The appearance of an icon is separated into abstract and concrete aspects. Abstract aspects specify relationships between the components of an icon; for example, that the title of a window lies along the top edge, is centered, uses 12-point boldface Helvetica, and sits in a rectangle whose height is 1/20th of the height of the view. These relationships hold regardless of the medium through which an icon is displayed. Concrete aspects specify details that pertain

---

[7]We currently make use of the Symbolics window and presentation systems to provide some low level capabilities. A portion of these facilities are also represented within the HITS knowledge base. Implementation of windows and menus are done using the normal Symbolics tools.

[8]The ability to provide interface manifestations of important conceptual properties can be quite powerful. In the Steamer system [14] for example, one of the most salient aspects of the interface was the ability to depict the causal topology of a propulsion plant by showing flow rates in the pipes that connected components. Nowhere in the underlying high fidelity simulation were these flows directly represented. Still they could be readily computed and used to augment the visual display in ways that were very revealing to users.

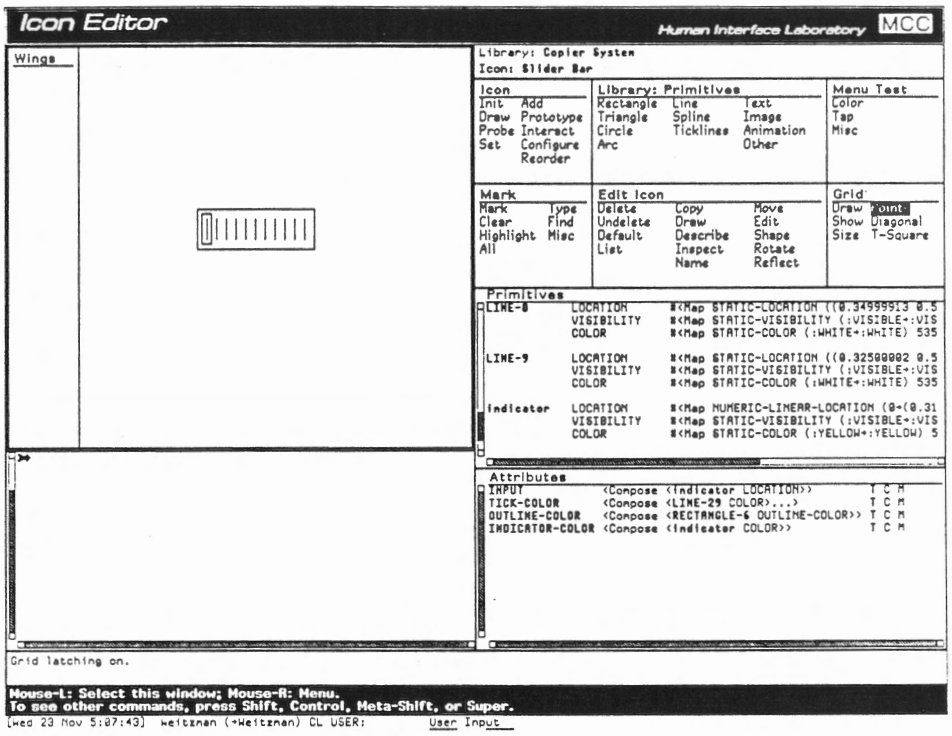


Figure 5: Icon Editor

to a particular medium, for example, the color of the line segments that make up the resistor and the list of coordinate points that comprise the segments. While abstract aspects are always present in the knowledge base, concrete aspects may be known only to a process that renders icons on a particular device.

An icon's behavior includes changing its visibility, color, texture, intensity, 3-D position, sound, or shape. One very simple behavior makes an icon visible whenever a boolean is true and invisible when it is false. Another behavior makes an icon's size directly proportional to the value of a single variable. Changes may be singular or may cycle through a fixed sequence, as in multiple-frame or color-table animation. A more complicated behavior cycles segments of an icon through colors, as in a movie marquee, as long as a function of various states and events returns a value within a certain range. An icon's behavior is simply chosen from a menu. Behavior can be associated with any part of an icon. A user of the Icon Editor can compose new behaviors readily from a stock of primitives. Should one wish novel behavior that can't be composed from the primitives provided, such as having an icon wrap itself around a sphere, or blow up, or wave like a flag, there is no way to obtain it short of writing code. While the general problem of describing arbitrary behavior remains unsolved, the Icon Editor provides a powerful set of stock behaviors and ways of composing them. Once a behavior has been described, it is stored in the behavior slot of an interface entity.

Building Graphical Interfaces: Graphics Editor

An entire interface's appearance is built by a tool called the Graphics Editor [8]. One uses it to instantiate elementary objects or icons, place them in scenes, and arrange their activity. Most of its capabilities are similar to those of conventional object-oriented graphics editors, the details of which should be familiar to the reader. Its main difference is that its output is a single interface entity that is entered into the knowledge base. That entity and its components are thus available

for inspection and reasoning when an application runs under that interface.

The coupling of an application and a graphical interface must be highly flexible. Users are never satisfied with a fixed number of views of a sophisticated application. The most important function of the Graphics Editor is to couple an application and an interface, a process called *tapping*. A tap is an object that associates an icon with behavioral functions. Icons may evaluate their behavioral function whenever a variable is changed, or at regular intervals, or by a user's command, or according to a complicated function of events and states. Maps are specified in the same way as behaviors. A number of standard maps are provided; a user selects one or composes simpler ones into a more complex map and tailors its parameters to fit a specific application. This approach minimizes the burden on application entities; each entity need only respond to a 'probe' message. Interface entities are responsible for probing specific application entities at suitable times. This tends to concentrate the details of coordinating an interface in the interface, rather than spreading them throughout the application. Such a balance seems well suited to animated interfaces, where sometimes a riot of events in the application signifies only that an application's variables are sitting within nominal ranges. Also, it makes it easier to attach different interface perspectives to a single application, since an application is unaffected by the coming and going of entities in the interface.

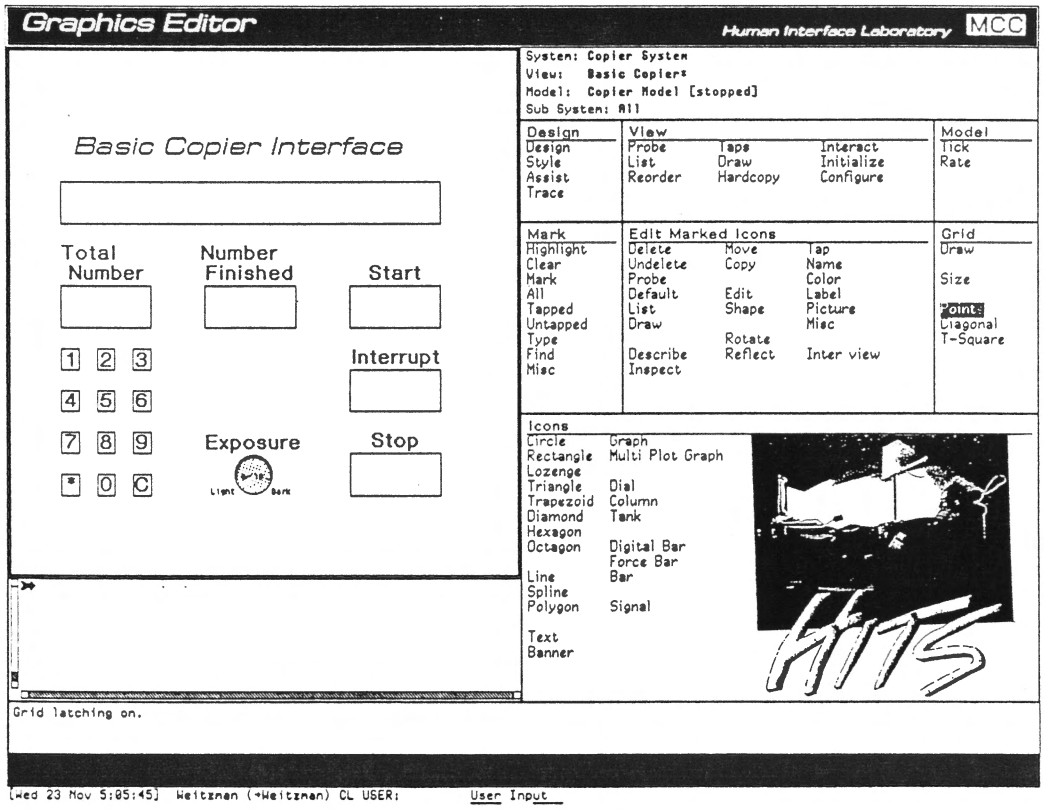


Figure 6: Graphics Editor

Building Graphical Interfaces: Pogo

We use a tool called Pogo[9] to create an architecture for 3-D graphical interaction by designing a

[9]At present Pogo is less completely integrated into HITS than the other tools discussed in this paper. Pogo represents one direction we expect for future graphics systems to take and for full utilization will require more sophisticated graphics hardware than is available on our present research platform. The Icon Editor and Graphics Editor exploit traditional bitblt types of graphics. Although Pogo is organized around the notion of display drivers and can be used on conventional hardware, it is best used with display-list graphic hardware.

hierarchical set of classes, attributes, and methods. Pogo has four jobs, one abstract, one declarative, one concrete, and one procedural. Its abstract job is to divide graphics into a hierarchy of concepts that can represent displayable objects for any application. Its declarative job is to represent graphical relationships and operations as rules in a knowledge base. Its concrete job is to translate graphical abstractions into tangible objects, frames of reference, and resolutions peculiar to various media. Its procedural job is to program the issuing of commands that render graphical abstractions onto various devices.

For every graphical component of an interface constructed with HITS, there is an object in the HITS knowledge base. Every graphical view has a root component that corresponds to the entire view, as well as a set of other components that correspond to pieces of the view. A view can be presented on any device by simply handing its root to a driver for that device. Each device driver is responsible for making optimizations that allow views to be presented efficiently. Device drivers themselves are objects in the knowledge base.[10] For each medium, one builds device drivers that interpret abstract entities. That way, only device drivers contain hardware-dependent library calls, but there is a performance penalty.

One uses Pogo to enter declarative specifications of graphical primitives into a knowledge base. Graphical abstractions should cover a range of complexity, from bottom-level primitives, such as line, to intermediate-level primitives, such as window, to top-level primitives, whose names are always controversial. It is in the attributes and methods of top-level abstractions that one settles fundamental issues such as 2D vs. 3D, real-time vs. non-real-time, device independence, and coercion. Prior systems that were based on procedural primitives (drawing commands) and concrete primitives (pixels) were judged too complicated, too hardware-dependent, and too application-specific. Primitives should cover a spectrum of specialization. General-purpose primitives such as polygons and strings of text are needed by all applications. Special-purpose primitives are needed both to give initial applications a head-start and to provide detailed examples for other applications to mimic.

The greatest benefit of representing all graphical information in a knowledge base is that all parts of the system are equally well informed about what the user can see. The application itself, the advisory system, the display device, and the interface all share the same entity. They do not have to build and maintain their own model of what is on the screen. Changes brought by one part are known to all other parts.

The greatest disadvantage of having all graphical information in the knowledge base is that a lot of sophisticated engineering has to be done to balance the competing needs for rapid interaction and semantically meaningful feedback. For meaningful feedback, every interaction should reach from I/O devices all the way to the knowledge base. For high performance, every trivial interaction should be handled entirely within I/O devices.

BUILDING GESTURAL INTERFACES

When an interface allows freehand input, it presents a recognition problem analogous to that of continuous speech. The user's hands are in almost continuous motion and what the system wants to receive is not a fast stream of coordinates but a slow trickle of very high-level interface entities. If the user sketches a graphical symbol that signifies a resistor, the device should encode the stream of gestures into the interface entity that represents a graphic for a resistor, having the position, size, and orientation that was sketched. This has motivated a tool called the Gesture Editor that trains a neural net to recognize users' sketches and gestures. After a neural net [12] has learned to recognize them its weights are entered into the knowledge base, to be retrieved whenever users want to draw icons in that vocabulary.

---

[10]Workstations today are in an awkward phase after the time when information was mostly textual and before the time when fast implementations of a flexible, well-thought-out, widely accepted, graphical standard are commonplace. We would like to use a graphical standard but the only routines that are acceptably fast come from proprietary libraries. Earlier work in multimedia led to the conclusion that interfaces need to be built out of abstract entities whose semantics are independent of media.

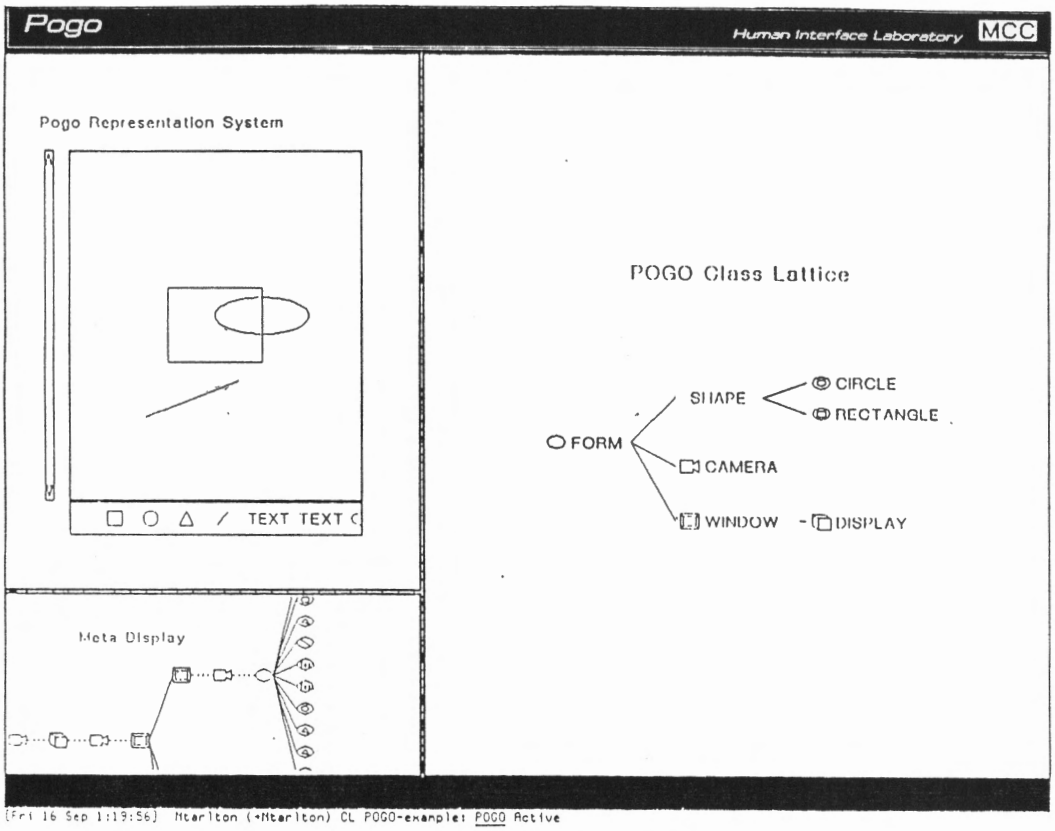


Figure 7: Pogo

Building Gestural Interfaces: Interactive Worksurface

The Interactive Worksurface is a system that can recognize and interpret notations and sketches that are drawn freehand. It consists of a large, horizontal, 72 bpi plasma panel coupled with a 1000 bpi digitizer and a 16 bpi touch sensor. As an output device, it functions as a raster device compatible with SunViews, X11, and the Symbolics window system. As an input device, the digitizing stylus sends 200 coordinate pairs per second over a VME-bus to software on a host computer that converts them to strokes, thence to a normalized bitmap, which is input to a neural net. The net classifies the pattern and posts an object representing the sketch on the blackboard of the HITS run-time system. Touch inputs are posted directly as coordinate pairs.

Building Gesture-Based Interfaces: Gesture Editor

The Gesture Editor is a tool for collecting, editing, labeling, and transforming training samples for neural networks. It was built to ease the task of creating and handling the large training sets required for building effective networks. Originally designed to collect and edit hand notation gestures for a stylus-based digitizer such as that employed in the Interactive Worksurface, it has been expanded to process scanned image samples and to generate artificial patterns as well. Samples are presented via a film-strip metaphor. The editor contains a neural net simulator and can output pattern files for use by the Rumelhart-McClelland simulator [12]. Network specification is simpler and learning is faster than in the Rumelhart-McClelland simulator.

Numerous facilities are supported by the Gesture Editor. These include facilities for switching between different coding schemes for inputs (e. g., size of array, dots vs. strokes) and outputs (e. g., conversions from feature codes to category codes), for capturing stroke images from commonly available digitizers, for using a network to label images as they are captured, and for mapping schemes to assign image labels to specific neural network output patterns. In addition, the Gesture Editor provides a facility for random variation of images that can be used when

training a network, so that on each pass through an image set a slightly different image is produced to feed to the network. Several sources of random variation are available, including rotation, aspect ratio stretch, stroke stretching and shrinking, and bitmap noise (pixel lossage and trashing). This facility is similar in purpose to the training-with-noise facility of the Rumelhart-McClelland simulator, but is much more sophisticated. It can be used both with the internal neural network simulator and with the pattern file generator. The Gesture Editor, like all HITS tools, can also be used interactively on the IWS.

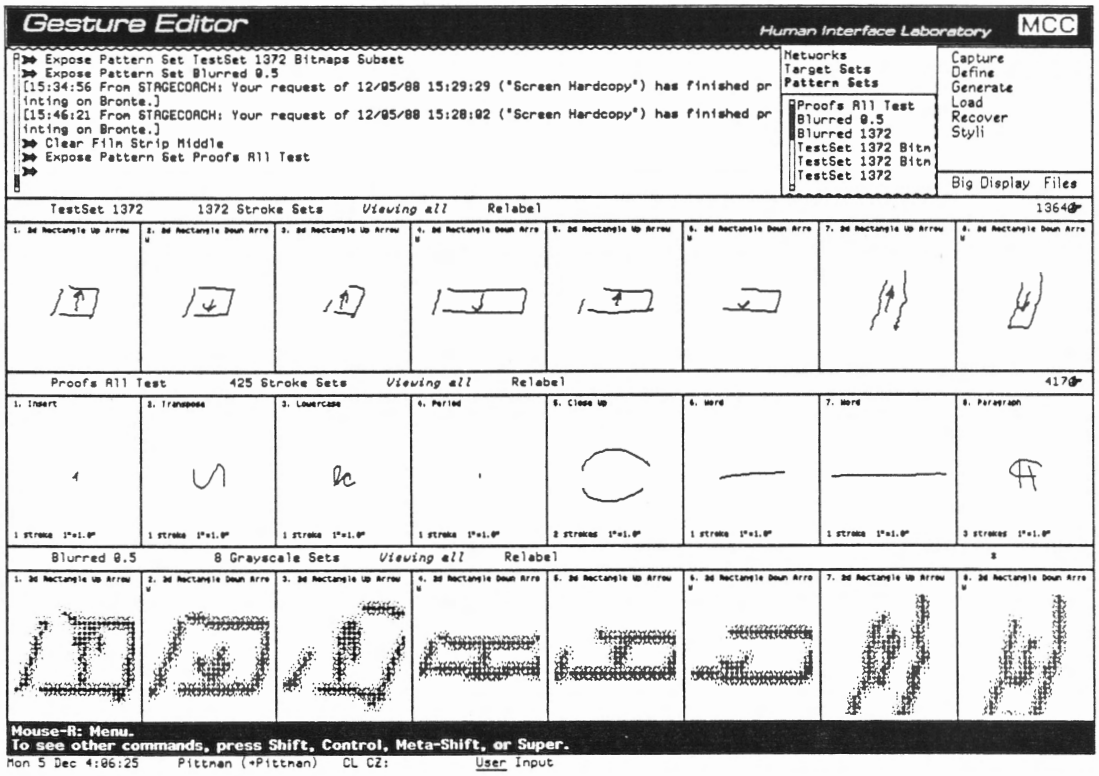


Figure 8: Gesture Editor

BUILDING NATURAL LANGUAGE INTERFACES

Natural language understanding and generation by computer requires complex programs. For a computer to use language as humans do requires a large body of knowledge about the world in general, knowledge about particular domains, knowledge about general discourse conventions and the particular discourse at hand, as well as knowledge about the sentence structure and words of a language and their mappings to representations in some internal store such as a knowledge base. The computational state of the art in natural language processing falls far short of general human discourse, but nevertheless limited forms of natural language can be usefully employed in the interfaces to restricted application domains.

The main rationale for use of natural language in an interface is that natural language provides the best means available for users to refer to sets of objects whose makeup is not fixed and finite, whose unique references are typically not known (or remembered) by a user, or whose reference by other modalities might be awkward and/or time consuming. For example, consider a computer model of something like a copier shown schematically in Figure 9.

As such a complex model evolves over time, typically involving many people, the need grows for some commonly understood language by which users can refer to concepts in the knowledge

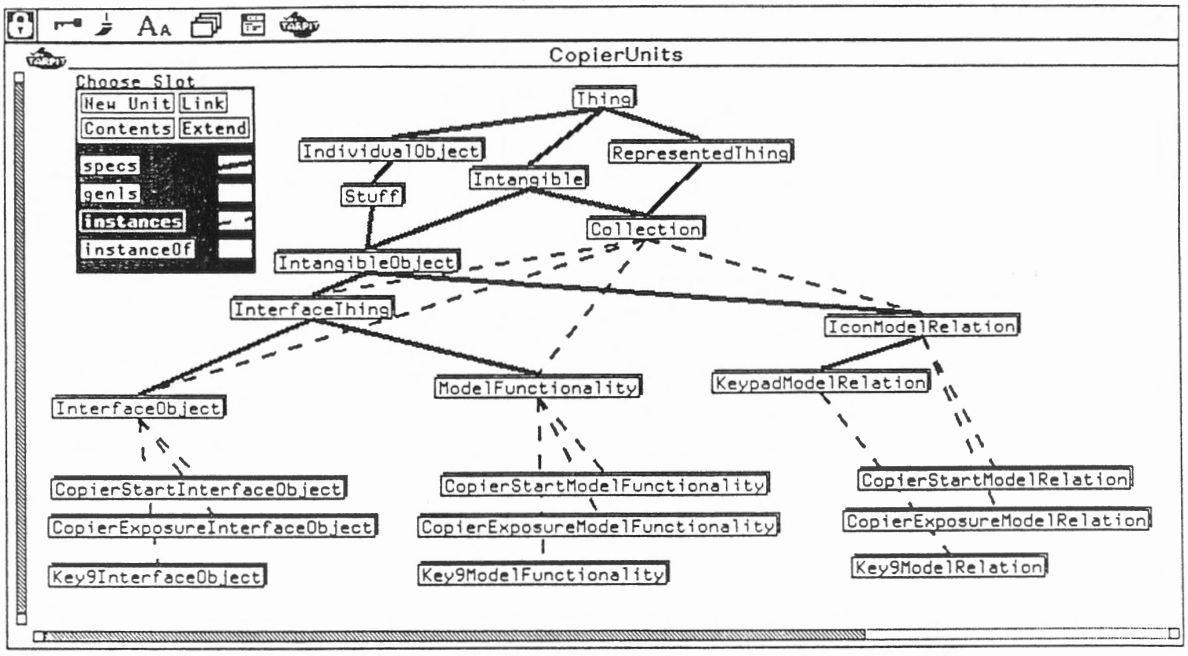


Figure 9: A complex reference domain

base. There seems to be nothing as suitable as a natural language like English in such cases. Unique names given to entities in a knowledge base are likely to be unpredictable and hard to remember. In knowledge editing and retrieval applications, the usefulness of graphical navigational techniques goes down in proportion to the size and complexity of the knowledge base. In many other kinds of applications, users may need to refer to knowledge concepts without being conscious of it. Such is the case in the copier interface design scenario described in the outset of this report, where a user taps an icon to the concept in a copier simulation model denoted by the English phrase "the start button". Phrases that would have worked equally well include "a button to start the machine", "the start function", "starting the copier", etc.

A complete natural language system includes the ability both to understand input from the user and to generate appropriate responses in natural languages. Because almost all of our effort up to this point has been devoted to the understanding side of such a system, we will focus our discussion here on this side of the problem.

Figure 10 shows a schematic diagram of a natural language understanding system. The ovals represent the necessary knowledge sources. The rectangles represent the processes that use the knowledge. The domain-independent HITS runtime system supplies all the rectangles in the figure. Thus, HITS supplies the standard operational components of an natural language understanding system. e.g., a parser, a semantic interpreter, an anaphora resolver, and a pragmatic processor.

Unfortunately, HITS cannot go quite so far in specifying all the necessary knowledge in a domain-independent way. It can go a long way toward providing the top oval, since syntactic knowledge varies relatively little as a function of the domain that is being described. The bottom oval must, of course, be redone for each application, but we make the assumption that the natural language system will use the same domain knowledge base that the application program itself uses. The middle oval must be created especially for each new natural language interface. To do this, HITS provides a tool, Luke, that will be described below.

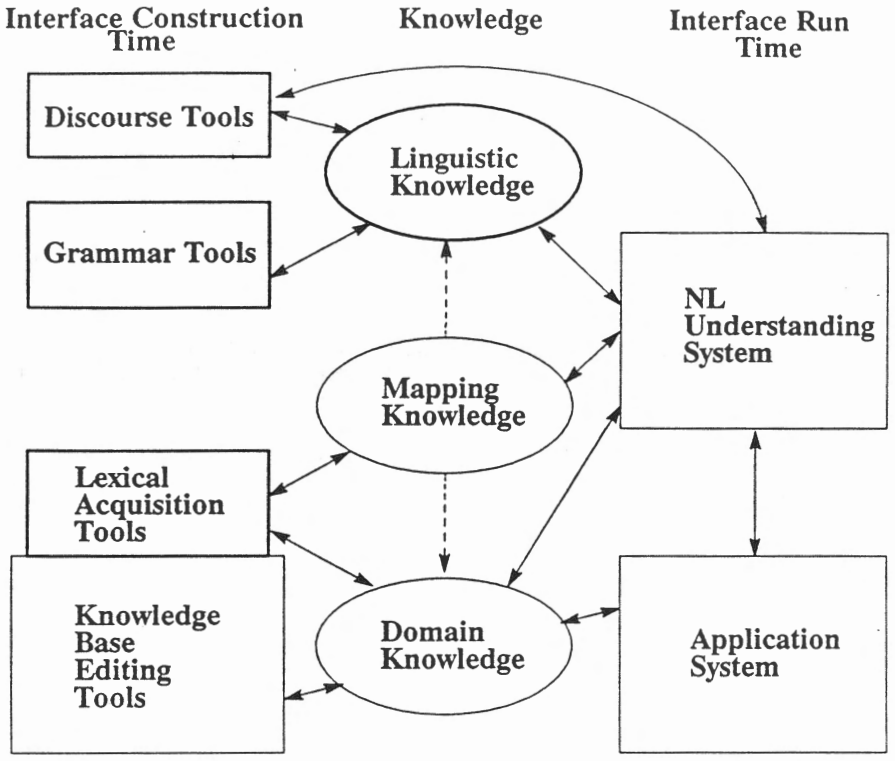


Figure 10: A Natural Language Understanding System

Building Natural Language Interfaces: Lucy

Lucy is the English understanding component of HITS. It is comprised of a set of tools for incorporating partial or full use of natural language into an interface. Lucy is integrated with other HITS tools in such a way that the use of natural language can be combined with other interface modalities such as graphics, sketching, pointing, menus, and command languages in the interface. Designers can then mix and match according to the needs of each particular interface being built.

The major components of Lucy are:

- a discourse and dialogue management module
- a semantic mapping module
- morphological and syntax modules
- an English grammar and core lexicon

The HITS run-time system, described later, integrates the processes involving these knowledge sources, as well as others in the interface, through a central blackboard. The blackboard allows for flexible interleaving of knowledge sources during analysis. This design for system architecture has been the result of several years of research on natural language understanding systems in which other approaches to the control problem have been tried and rejected.

As we saw in Figure 10, three kinds of knowledge are necessary to support a natural language understanding system. We will briefly discuss the support HITS provides for the acquisition of each of them.

Linguistic knowledge (the top oval) is mostly portable from one domain to another. Although the words we use vary as a function of what we are talking about, the syntactic rules that govern the use of those words vary little. Thus, HITS provides a syntactic grammar for each language that it

covers. But HITS also provides a set of grammar development tools. Interface designers (and perhaps even users) will probably want to do some extensions and/or customization of the grammar and discourse components as each new Lucy-based interface comes into being. Interface designers will certainly need to do testing. And further, such tools would of course be useful should HITS users want to write grammars for languages other than English. The major tool for grammar development is Lucy Lab, shown in Figure 11. It is an interactive environment used to examine the state of an analysis by inspecting and stepping through a graphical display of the underlying rule firings in a chart/blackboard. Off-line testing of a corpus of input strings is available through a batch test facility.

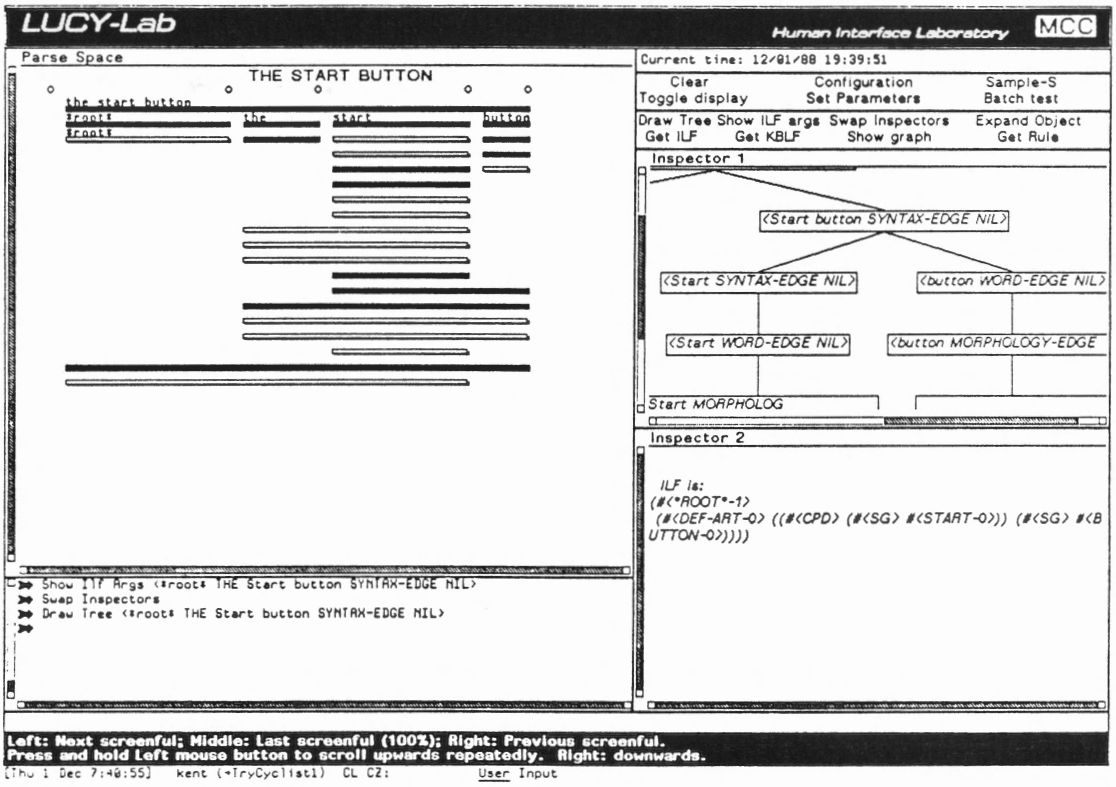


Figure 11: Lucy Lab Screen

Discourse Lab (Figure 12) is an analogous set of interactive debugging and development tools for inspecting the state and processes of the discourse component. An example where Discourse Lab would be used is in the development of a interface component to understand a pronoun such as "it". As pronouns come into the dialogue, the discourse history needs to be consulted to assign a meaning to the pronoun. Lucy makes the best guess consistent with heuristics and the known constraints. Sometimes, however, insufficient information is available for Lucy to determine the user's meaning. In such cases the user is engaged in a clarification dialogue that in turn is added to the history of the interaction.

Domain knowledge (the bottom oval) must mostly be redone for each new domain (although there may even be portable parts of this [9]). The premise that underlies the HITS approach to natural language understanding is that the natural language system will not have its own domain knowledge. Instead it will use the domain knowledge that the application program exploits. Thus the creation of the knowledge in this bottom oval is not specifically an interface creation problem. It can be done using the standard knowledge base tools upon which HITS is built.

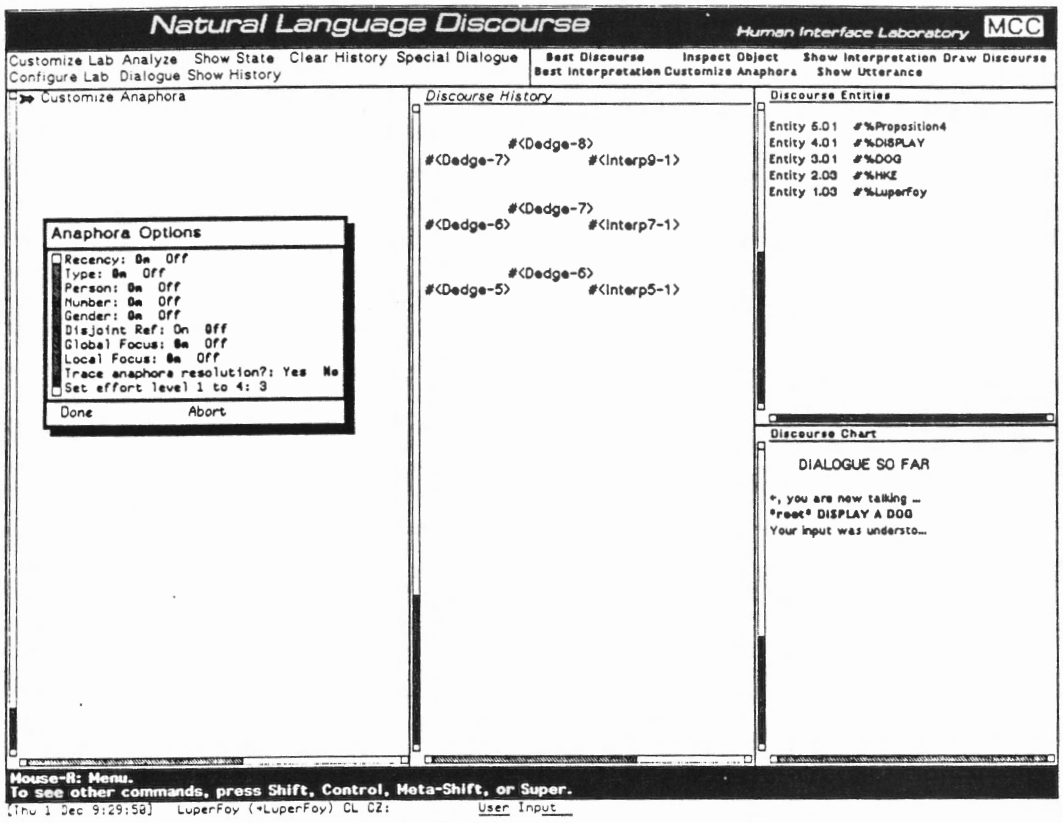


Figure 12: Discourse Lab Screen

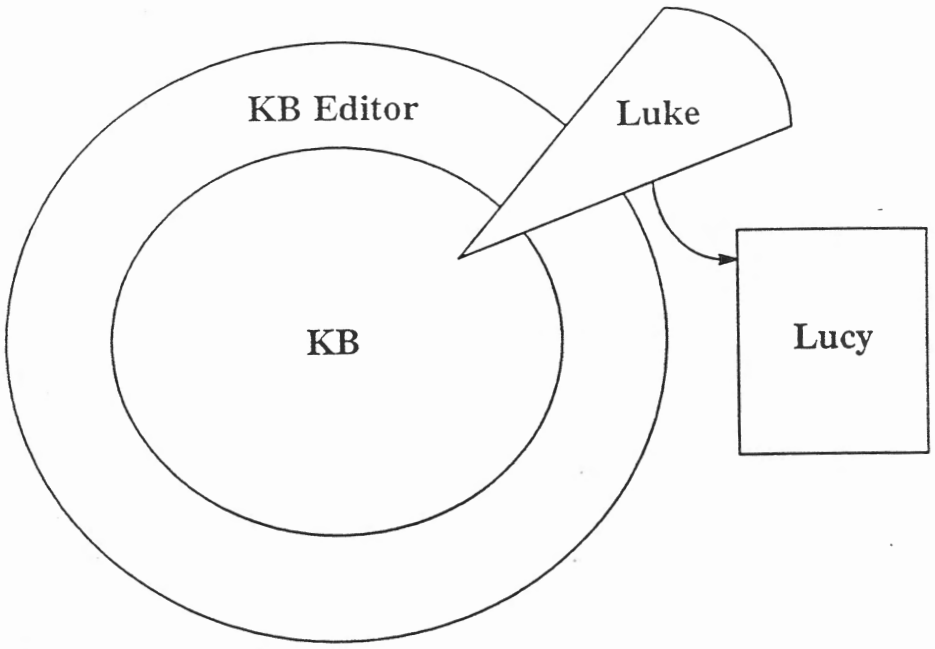


Figure 13: A Layered Tool for Knowledge Base Construction

Building Natural Language Interfaces: Luke

Mapping knowledge (the third oval) is the main bottleneck in building a natural language interface for a new application program. HITS provides a tool called Luke to help to overcome this problem. Luke facilitates the creation of the semantic mapping rules that are needed by a natural language system when it maps from words in a language to/from concepts in a knowledge base. The key idea behind the design of Luke is that, since these mappings are conceptually linked to the knowledge base objects to which they refer, the right time to build the mappings is roughly the same time that the knowledge base objects themselves are being created. Thus, Luke is implemented as a set of hooks into the base knowledge base editor on which HITS is built. The architecture of Luke is shown in Figure 13. Whenever a knowledge base object is created, the Luke command *Associate Word* can be invoked to associate one or more words with the new object. These words can be defined to map to the object itself or to some path that goes through the knowledge base and that refers to the new object. Luke incorporates a general model of lexical semantics so that it can, in many cases, guess the correct form for the semantic mapping rules that define a new word. It then displays its guess to the user, who can easily edit the new rules before they are actually stored.

Luke is a colloquial name for an extension to the HITS Knowledge Editor (HKE) that enables editing and maintaining lexical knowledge. Lexicon-building tools are crucial for making natural language processing an integral part of the interface designer's toolkit; yet, such interface designers cannot be expected to be experts in natural language processing. Luke attempts to bridge this gap by representing and acquiring lexical knowledge in the same manner as any other knowledge representation task. Thus any user of HKE has almost all the requisite skills to build lexicons for the run-time Lucy natural language system.

Luke provides a set of special display and editing methods for units representing "open class" English words: those that can gain new entries over the course of everyday usage, such as nouns, verbs, and adjectives. The "closed class" words such as articles and pronouns are automatically entered in the lexicon from the start. HKE users define new words by invoking one of a small number of commands. For example, "Associate Noun" defines a particular noun to semantically denote a set of units in the knowledge base, as well as defining that noun's syntactic properties, such as whether it has mass or count properties, its pluralization, etc.

As a result of one of these special commands, several new units are created representing orthographic, syntactic, and semantic aspects of the word. These can be inspected in the same way as any other unit, although they have special display methods and so may appear slightly different than ordinary units. These units are processed by a special "lexicon compiler", that produces a compact indexed runtime structure for the Lucy parser. As soon as acquisition of lexical units is completed, Luke compiles the new word sense and makes it available in the natural language lexicon.

Unfortunately, word definitions are as likely to need modification over time as programs. The definitions of words can be debugged in HKE since the Lucy parser is an integral part of the HKE interface. After defining a new word, it instantly becomes a legal word to use in the HKE interface itself. Thus in one command, the user can define the noun "bear" to mean some instance of the class "#%Bears" and then in the next command make use of it by issuing the command:

```
:Inspect Unit "a brown bear in Wisconsin"
```

instead of

```
:Inspect Unit #%BrownBear76235
```

and expect the noun phrase to be parsed and analyzed correctly (assuming, of course, that "Wisconsin" and "brown" and "in" had been previously defined as words in a similar manner.) In this way Luke brings the same immediacy to lexical acquisition that has proven itself in the world of exploratory programming.

BUILDING COLLABORATIVE INTERFACES

Our approach to the acquisition of knowledge to support collaboration and advice follows the same two-step approach described above. First, it should be possible to derive some of it from a declarative representation of the application. Ideally, this representation *is* the definition of the application, and not a post-hoc representational depiction of the application developed independently of the application. There are several kinds of knowledge that could be constructed in this way:

- **Static application representations.** Any implementation of an application contains definitions of the components of the application: the numbers and kinds of inputs and outputs each of these procedures has, constraints on the data that are accepted and produced by these procedures, and so on. In present-day applications, this information is all buried inside the application in unanalyzable procedural code. If these components were implemented in an inspectable, declarative form, it would be possible to derive advisory knowledge structures characterizing these aspects of the application's procedures (or, better still, to use these knowledge structures as the basis of advisory reasoning). This is rather low-level, abstract knowledge about the application, but it is important information nevertheless.
- **Knowledge about basic interface capabilities from advisory-laden components.** A powerful interface development environment should make building blocks available to interface designers that can be combined to yield working interfaces. We envision these building blocks as supporting basic interface operations; for object-oriented graphical interfaces, these operations would include the creation and deletion of screen icons, and techniques for drawing links between these icons. In addition, these building blocks would contain knowledge structures capable of informing an advisor about how the operations can be applied correctly (*and incorrectly*) in an application. For instance, a building block for drawing links between screen objects should contain the interface-level code that portrays the creation and deletion of these links on the graphics display, and the application-level code that implements the functional interconnection between the application elements connected by the link. This component should also contain knowledge relevant to the linking of objects: what users must do to create and delete links, what effects a link has on the application program, and common misconceptions about the manipulation of links (i.e., invalid ways of drawing links, and invalid models of how links support information transfer between application elements). The point of this approach is to define this knowledge once, and to have it be inherited by any application advisor that utilizes this link-drawing building block.
- **Guided acquisition of advisory knowledge.** A final important classification of advisory knowledge is the knowledge that is highly application-specific. As noted above, a representation of the number of inputs to a statistical procedure could be derived from a declarative representation of that procedure, but that same representation could not support the derivation of knowledge about what role that procedure plays in the application, why one would want to compute such a statistic, or what one would do with one if one existed. Because of its application specificity, it is also unlikely that this knowledge could be inherited from some other application, in the way that knowledge about linking might be inherited from an intelligent interface toolkit. The only way to get this knowledge into the system is to encode it, through standard knowledge acquisition techniques.

Two aspects of HITS make the task of representation of knowledge to support collaboration and advising easier than it might otherwise be. The first is a rich knowledge base about interface topics. This provides a strong foundation for representing application-specific knowledge. This is operationally no different from the normal knowledge acquisition procedures; the power comes from the accumulation and richness of the knowledge available, and the shorter conceptual distance between the concept to be represented and those already present in the knowledge

base. The second aspect is a set of techniques for guiding the knowledge acquisition process. Just as Luke prompts the knowledge enterer for semantic mapping rules corresponding to concepts being entered into the HITS lexicon, a similar set of techniques can prompt the enterer for the advisory knowledge discussed earlier: the common models and misconceptions of these concepts. Together, these features serve to structure, and thereby ease, the knowledge representation task.

As was the case with our approach to natural language understanding, our approach to collaboration and advising draws on several types of knowledge. THEMIS [6], [10], which handles user queries and supplies advisory strategies to their solution, is composed of a procedural component and a set of knowledge bases to which the component refers. Some of the knowledge bases contain domain-independent knowledge; this knowledge is also provided as part of HITS. As an example, many of the advisory strategies that are exploited by THEMIS are very general (e.g., the strategy for describing a plan is very general, even though the content of a particular plan is, of course, tied to a problem domain.) Some of the required knowledge is necessarily domain-dependent. What HITS can provide for these domain-specific knowledge bases is a set of tools to aid in their construction.

Building Collaborative Interfaces: Conversation Tool

A number of recent empirical studies have pointed out problems inherent in both human-human and human-computer communication [7] [15]. Consequently, exploring techniques for dealing with communication problems has become a focus of our research in advising. We search for ways advice seekers can take a more active role in the advisory interaction, allowing them to redirect the course of the interchange, to follow up on parts of it, or to suspend the interaction and resume it at a later time. In short, we seek to move toward a *collaborative* interaction, in which the advisory system and the user share the ability to direct the interaction [16]. The HITS Conversation Tool is an experimental interface built on this foundation that supports collaboration design by making it easy for designers to obtain relevant information about existing collaborations and allowing them to pursue alternative conversational avenues beginning from any piece of advice.

There are several ways for collaboration designers to initiate an advisory interaction. One is by indicating the knowledge structure they are interested in they can set the advisory focus to relevant strategies. The collaboration designer then can seek advice by pointing at the focus unit and the relevant strategies associated with it. In addition, two important types of advice-seeking, obtaining descriptions and obtaining instructions, can also be initiated.

When an advisory interaction completes, the structure of the interchange is shown on the display. Collaboration designers can follow up on any piece of advice by clicking on it. This brings up a menu of further advice available from that point. The possibilities for further advice include getting an elaboration of the instruction, receiving an explanation of any of the concepts referred to, and being shown the actual knowledge base objects that were referred to.

Collaboration designers also may redirect the interaction dynamically. When advice is being given, they may indicate trouble and the system responds by asking them to describe which of its communications caused the trouble and then to choose from the further advice that is available. After this clarification subdialogue has been completed, the collaboration designer may resume the interrupted interaction.

AN EXAMPLE APPLICATION: THE HITS KNOWLEDGE EDITOR

The HITS Knowledge Editor (HKE) is a example application of HITS in the domain of CYC knowledge base editing. Our purpose in building HKE is twofold. First, HKE provides a working example of the use of HITS. Second, HKE is a research vehicle for understanding the demands that the knowledge editing task places on interfaces. As a knowledge editor, HKE supports browsing and editing of the CYC knowledge base. This is a substantial application because:

- **Complexity of the Knowledge Base** The CYC knowledge base is large

(approximately 30000 units), complex (20 inference methods, thousands of classes of objects), and users need effective methods of visualizing the knowledge base's structure, contents, and inference processes without being overwhelmed with information. CYC knowledge enterers must not only enter information but must also ensure that the knowledge is coordinated with work being done simultaneously by many other knowledge enterers.

- **Complexity of Knowledge Editing** Knowledge editing is not a single task. It consists of the multiple complex tasks that are involved in deciding how to conceptualize and represent the important aspects of domain knowledge for various types of applications. The representational difficulty varies with the type of domain being represented and the amount of epistemologically primitive representations required to tie the domain into the CYC knowledge base. This can vary from having to make deep epistemological decisions to being able to exploit the copying and editing of already existing closely related representations.

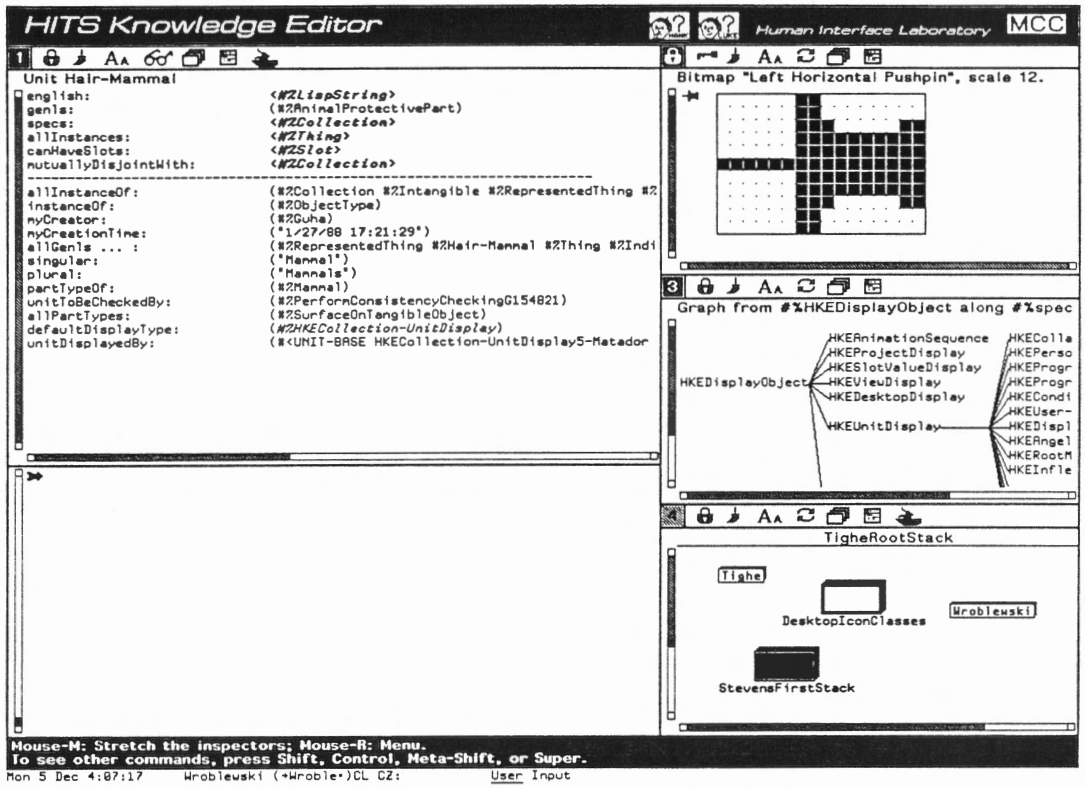


Figure 14: HKE Screen

HKE Is Implemented With HITS

As an example HITS application, HKE makes use of most of the components of HITS in its implementation. For instance, HKE uses the HITS blackboard (discussed later) to handle user input and exploits a representation of itself and the task of knowledge editing to collaborate with users. When a user types a command or clicks the mouse the knowledge base is consulted about how to handle the command or mouse gesture, and appropriate routines to respond to the user action are invoked.

HKE allows graphical views to be created and attached to portions of the knowledge base in order to browse or document its structure. For instance, suppose one were creating a portion of the knowledge base concerning the distribution, scheduling, and maintenance a set of computer workstations. Although the requisite units for the workstations need to be created, the enterer of this knowledge has the tools available to make a graphical view of the building floorplan, with

icons for various workstation types, etc. Subsequently, other knowledge enterers may choose to browse that part of the knowledge base via this graphical representation rather than the default tabular method.

HKE integrates natural language processing in its interface. At any point in the HKE command set where a unit's name may be typed, HKE also allows the Lucy natural language parser to intervene and parse the user input as a noun phrase. Conversely, the user may associate nouns, verbs, adjectives, prepositions, etc, with units and relations in the knowledge base as a method of semantically defining English words for later use. Thus, in exactly the same manner as with the previous graphical methods of documenting the workstation knowledge base, one may document or browse that section of the knowledge base by describing in English the units desired.

HKE uses a flexible *advice angel* mechanism to detect possible user errors and report them. This is done in conjunction with a personal agenda for each user. When one of HKE's angels detects a possible task for the user and presents it, the user has the option of ignoring it, dealing with it immediately, or scheduling it on his or her agenda for later consideration. The personal agenda mechanism helps avoid intrusive error messages while retaining the advantages of having an automated assistant to help.

HKE is Represented in CYC

As an example HITS application, major portions of HKE itself have been represented in CYC. For example, most of the HKE state information is stored in the knowledge base. This permits a powerful computationally reflective ability for HKE and other HITS tools. In addition, information such as the user's current tasks, personal agenda, etc., are stored in the knowledge base and remembered by HKE from one session to the next, even if the user switches machines or reboots.

One major advantage of representing this knowledge editor in the representation language it edits is that HKE can be customized by the user in exactly the same way as is involved in editing any other knowledge. For instance, a user can customize the display of the standard inspector menu icons by editing the units that represent those icons. This provides an interesting and useful interface prototyping environment, where effective ways to visualize information can be developed *concurrently* with the information structures themselves. Of course, the abilities of users to have access to this kind of facility can be modified and specialized variants of the HKE can be implemented as part of a tool chain of HKE editors.

THE HITS BLACKBOARD

One key to enabling collaborative multimodal interfaces like HKE is a run-time architecture that permits the flexible intermixing of multiple modes of interaction and the maintenance of overall dialogue and subdialogue histories. The HITS Blackboard serves this important integrating function. It provides:

- support for modular design of knowledge sources
- a problem solving mechanism
- a communication medium for HITS knowledge sources
- a common mode-independent format for sharing information between knowledge sources
- support for fine-grained integration of the problem-solving actions of the knowledge sources
- a priority-based agenda control structure for scheduling the possible actions of the knowledge sources
- a goal-oriented control structure for organizing the proposed actions of the

knowledge sources

The Basic Blackboard

The basic action of the blackboard is to schedule actions proposed by the knowledge sources. Every time an entry is posted on the blackboard, all the knowledge sources have an opportunity to examine the new posting and propose new actions for the blackboard to schedule. The scheduler component of the blackboard chooses from the proposed actions and executes one. The others normally remain on the agenda and are reconsidered at a later time. These actions typically operate on the new posting (possibly in addition to older postings) and produce additional new postings. The knowledge sources then have an opportunity to look at this new posting and the process repeats.

The blackboard processing cycle starts when some external process injects a *primitive event* onto the blackboard. This triggers the knowledge sources and the blackboard cycle takes over. A primitive event may represent a user action (e.g., keyboard input, a mouse click, or other gesture) or it may represent an event request from a knowledge source (e.g., to ask the user for some information, to tell the user something, or to perform any of a wide variety of other potential system actions).

A number of different perspectives help to illuminate the blackboard's operation. These include viewing the blackboard as priority queues or as goal trees. From the perspective of a priority queue, when a knowledge source proposes an action, it associates a priority or score with the action. The higher the priority, the more likely this action is to help solve the current goal. Priorities can be used to direct the search of the solution space in a propitious order. In a given search we may be searching for the best solution to our problem, or we may merely be searching for a good solution. If we are looking for the best solution, then we may have to explore the entire solution space. In the latter case the heuristic ordering may allow us to avoid searching portions of the solution space (based on knowledge gained earlier in the search).

In general, computation on the blackboard proceeds in a bottom-up fashion, with the knowledge sources reacting opportunistically to the appearance of events. However, without some organizational structure this type of computation can frequently become expensive and unproductive. In HITS blackboard information is structured and maintained as a goal tree. The goal tree serves several important functions: it provides a place to state which problem each knowledge source is trying to solve and associate methods for recognizing adequate solutions when they appear, to store heuristic information about how to efficiently organize the computation, and to describes how knowledge sources relate to one another in terms of priority, sequencing, and dependency.

The goal tree is primarily an and/or tree of goals in which sub-branches are allowed to be ordered or unordered. In the case of an ordered-or subtree, goal solutions are attempted sequentially until one succeeds or all goals fail. In the ordered-and subtree the goal solutions are attempted sequentially until one fails or all succeed. These ordered subtree types allow the specification of order dependencies between particular computations taking place on the blackboard. Unordered subtrees have the same satisfaction conditions except processing is allowed to proceed in parallel.

Each knowledge source that proposes to perform an action must indicate the goal in the goal tree the action is intended to help solve. This action is then scheduled on a locally controlled agenda. During each cycle of the blackboard the goal tree is searched for an unsolved goal with actions scheduled to perform. The action is then performed and the result is reported back to the goal. The goal decides whether the action was sufficient for solution. If so, the goal is marked as solved and its supergoal is notified. This is the first method of goal satisfaction. The goal is directly achieved as a result of executing an action scheduled on the goal itself. The supergoal can now either accept this solution and move on or reject the solution and instruct the subgoal to try again. The second method of goal satisfaction is through the satisfaction of associated subgoals. This is treated in the same manner as direct satisfaction and the supergoal is notified.

Use of the Blackboard within HITS

The blackboard is used within HITS in the performance of two major tasks: parsing and interpretation of information received from the users interaction with the system and communication between system components.

As an example let's consider what follows from a user initiating the command "Tap Icon" to relate the visual behavior of an icon to an object in the knowledge base. The command string is broken down by the command processor into the type of command being performed and the arguments specified. These constituents are then placed on the blackboard as depicted in Figure 15. Several knowledge sources are activated by the appearance of this information. One knowledge source is responsible for recognizing the type of action and linking it to its knowledge base equivalent. Others are responsible for determining the correctness of the arguments to the command and building partial command descriptions. Note that some operations simply add information to an existing blackboard description (elaboration) and others create a new description from existing descriptions (composition). Once all the arguments have been interpreted and accepted a description is built that contains enough information to perform the action. Unless some other knowledge source intervenes this action is performed and the resulting effects are described and placed back on the blackboard.

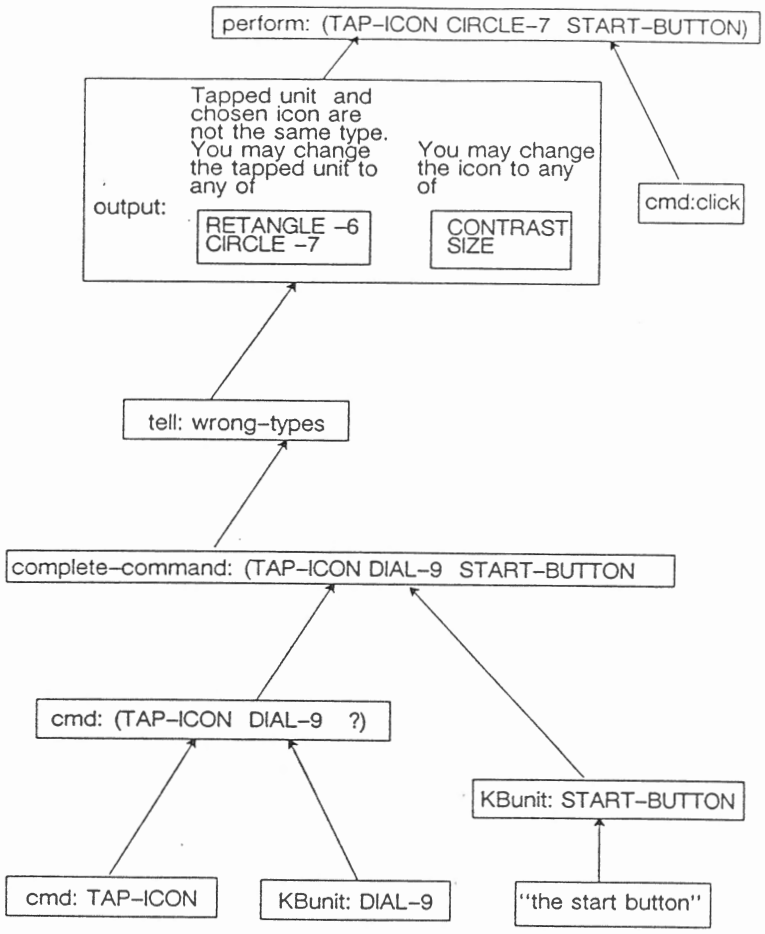


Figure 15: HITS Blackboard Example

The modularity provided by the blackboard architecture allows multiple mechanisms to operate on information during interpretation. In the example above a knowledge source that simply checks the arguments for validity is successful in resolving the first argument. In many cases, however, this simple checking mechanism will be unable to resolve the argument. The second

argument in the above command has been entered using the natural language phrase "the start button". This requires the activation of knowledge sources to handle the natural language input. Figure 16 shows the information placed on the blackboard by Lucy during the process of resolving this argument.

The public aspect of the blackboard allows other HITS knowledge sources to look at and influence the information appearing on the blackboard. A knowledge source can react to the information appearing on the blackboard passively (as a monitor of user activity), constructively (deriving new information to place back onto the blackboard), or destructively (blocking information on the blackboard from being processing by other knowledge sources). One knowledge source within HITS which reacts in all these ways is the advisor. In the Tap Icon example, the user has attempted to tap a dial icon (a continuous type icon) to a discrete value. The advisor notices this discrepancy when the information appears on the blackboard in the form of an executable command. The advisor then supplies several options: attempt to proceed as is, select a discrete icon, or select a continuous tap variable. If either of the latter two options are taken then the current executable command is blocked and a new description is constructed and allowed to proceed.

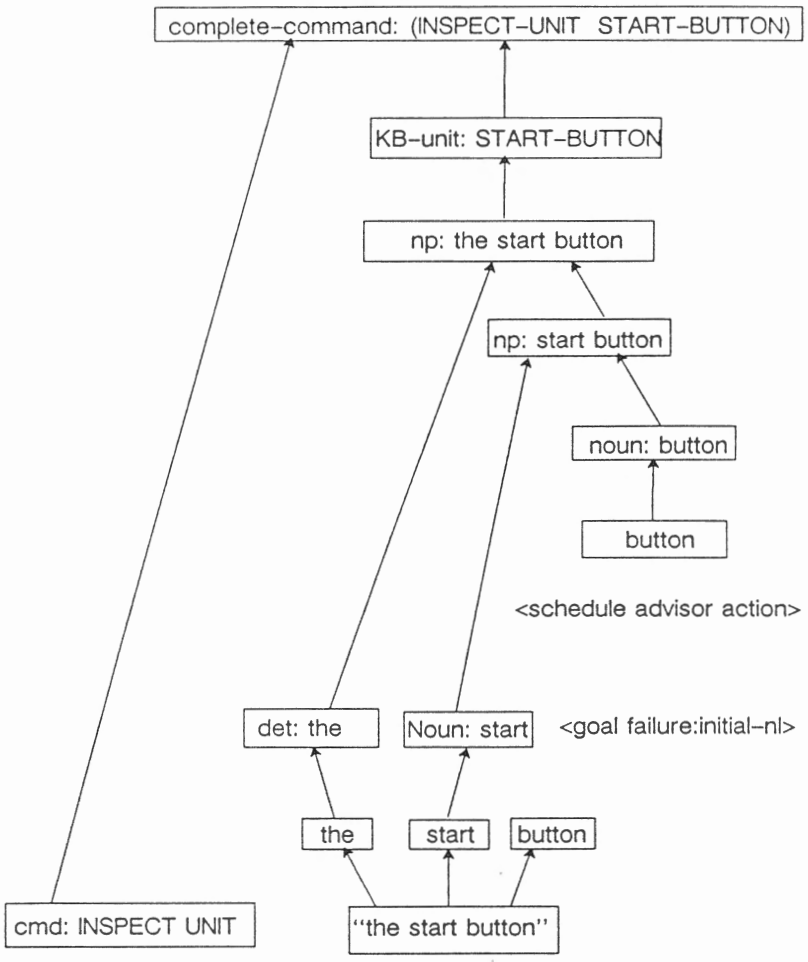


Figure 16: Lucy Blackboard Processing

Figure 17 shows the HITS Blackboard Goal Tree. At a high level the goal tree is a specification of the goals of the system while interacting with the user. Knowledge sources for interpretation of user input operate under the INTERPRET goal, their relative order and dependencies are determined by the subtree type of their supergoal. Knowledge sources that react to user input (e.g. advisory functions and applications) operate under the RESPOND goal. In the "TAP ICON"

example, the successful interpretation of the command causes the application to schedule the performance of the action on the STATE-CHANGE goal. The advisor also reacts to the command by scheduling some advice for the ADVISE goal. Since the ADVICE goal precedes the RESPOND goal in the goal tree the advice gets a chance to run first, thereby allowing it to block the scheduled performance of the command if necessary.

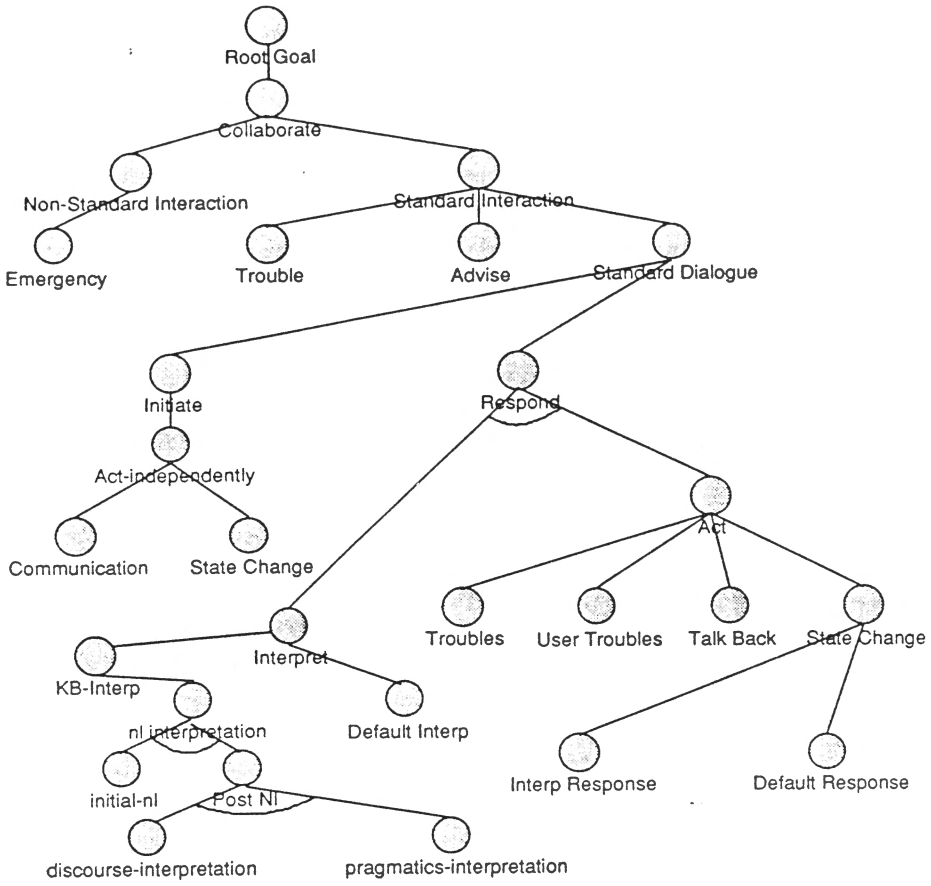


Figure 17: HITS Blackboard Goal Tree

SUMMARY

In this paper, we have explained the ideas motivating our research programme and introduced HITS, an integrated set of tools for building collaborative multimedia interfaces. The most important difference between HITS and the User Interface Management Systems (UIMS) to which it might be compared to is the emphasis on the role of knowledge representation. HITS provides an integrated knowledge base that covers a broad range of topics, all of which impinge in significant ways on the design of collaborative multimedia interfaces. General domain-independent knowledge is represented once and provided as part of HITS. HITS tools assist users in the construction of domain-dependent knowledge. We have assumed that application programs for which HITS-based interfaces are being built are themselves knowledge-based systems and that applications and interfaces are being designed and built together. This contrasts with other UIMS systems in which the application is viewed as a black box by the interface and any relevant parts of it must be modeled explicitly. In HITS we attempt to avoid duplicating the representation of the application in the interface.

Throughout the paper we have emphasized the knowledge-based parts of the interface. Thus, although such standard interface capabilities as properly presented menus, scrolling windows, and simple command interpreters are clearly important, we have focused on the more knowledge intensive aspects of interface design: understanding natural language and gestures, generating advice, modeling users and their tasks, and critiquing graphical design. Because HITS is a set of integrated knowledge-based tools focused on design knowledge and semantic mappings between interfaces and knowledge bases, it might be better characterized as what Foley [5] terms a UIDE, User Interface Design Environment, and in fact we envision HITS evolving into a General User Interface Design Environment, GUIDE.

Another unique aspect of HITS is its emphasis on allowing multiple input and output modalities (pointing, sketching, touch, natural language, graphics, video) to be employed in an integrated fashion by users. This permits interaction with HITS-designed interfaces in the ways most natural for the tasks at hand, and allows us to explore new research questions about how users exploit this freedom to mix modalities in ways best suited for a particular tasks.

ACKNOWLEDGMENTS

The design and construction of HITS is a project of the Human Interface Laboratory at MCC. All of the members of the laboratory are involved in its design and implementation. We want to acknowledge their efforts, ideas, and assistance and thank them for making it such an intellectually exciting activity. Portions of this paper are based on a presentation given at a workshop on Architectures for Intelligent Interfaces [11].

Members of the Human Interface Laboratory

Chinatsu Aone
Anthony Aristar
Jim Barnett
Bill Bohrer
Rich Cohen
Jonathan Grudin
Will Hill
Jim Hollan
Megumi Kameyama
Janet Kilgore
Carol Kroll
Bill Kuhlman
Susann Luper-Foy
Inderjeet Mani
Gale Martin
Paul Martin
Tim McCandless
Jean McKendree
Linda Mitchell
Martha Morgan
Maria Nasr
Michael O'Leary
Jay Pittman
Steve Poltrock
Mosfeq Rashid
Mark Rosenstein
Supoj Sutanthavibul
Mark Tarlton
Nong Tarlton
Loren Terveen
Steven Tighe
C. Unnikrishnan
Louis Weitzman
Wayne Wilner
Kent Wittenburg
Dave Wroblewski
Ken Zink

References

[1] Norman, D. A., S. W. Draper (editor). *Cognitive Engineering.* Lawrence Erlbaum Associates, 1986.

[2] Lenat, D., R. V. Guha, & D. V. Wallace. *The CycL Representation Language.* Technical Report, MCC Human Interface Laboratory, 1988.

[3] Hutchins, E. L., J. D. Hollan, & D. A. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction* 1:311-338, 1985.

[4] Hutchins, E. L., J. D. Hollan, & D. A. Norman. Direct Manipulation Interfaces. In D. A. Norman, & S. Draper (editor), *User Centered System Design: New Perspectives on Human-Computer Interaction.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[5] Foley, J.
personal communication.

[6] Hill, W. & J. R. Miller. Justified Advice. In *Proceedings of the CHI '88 Conference on Human Factors in Computing Systems.* 1988.

[7] Hill, W. C. *Advice Seeking, Giving, and Following at a Graphical Computer Interface.* PhD thesis, Northwestern University, 1988.

[8] Hollan, J. D., E. L. Hutchins, T. P. McCandless, M. Rosenstein, & L. Weitzman. Graphical Interfaces for Simulation. In W. Rouse (editor), *Advances in Man-Machine Systems Reseearch.* JAI Press, Connecticut, 1987.

[9] Lenat, D. B. & E. A. Feigenbaum. On the Thresholds of Knowledge. In *Proceedings IJCAI 87.* 1987.

[10] Masson, M., W, Hill, R. Guindon, & J. Conner. Misconceived Misconceptions. In *Proceedings of the CHI '88 Conference on Human Factors in Computing Systems.* 1988.

[11] Hollan, J., J. Miller, E. Rich, & W. Wilner. Knowledge Bases and Tools for Building Integrated Multimedia Intelligent Interfaces. In *Architectures for Intelligent Interfaces: Elements and Prototypes.* 1988.

[12] Rumelhart, D. E., J. L. McClelland, & the PDP Research Group. *Parallel Distributed Processing.* MIT Press, Cambridge, Mass., 1986.

[13] Pitman, K. M. *CREF: An Editing Facility for Managing Structured Text.* Technical Report, MIT A.I. Memo 829, 1985.

[14] Hollan, J. D., E. L. Hutchins, & L. Weitzman. Steamer: An Interactive Inspectable Simulation-Based Training System. *AI Magazine* 5(2):15-27, 1984.

[15] Suchman, L. *Plans and Situated Actions: The Problem of Human Machine Communication.* Cambridge: Cambridge University Press, 1987.

[16] Terveen, L. *Making Interaction Accountable.* Technical Report, MCC Human Interface Laboratory, 1988.

[17] VanLehn, K., J. S. Brown, & J. Greeno. *Competitive Arugmentation in Computational Theories of Cognition.* Technical Report, Xerox CIS-14, 1982.

MICROELECTRONICS AND
COMPUTER TECHNOLOGY
CORPORATION

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