Human–Computer Interface Development: Concepts and Systems for Its Management

H. REX HARTSON and DEBORAH HIX

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

Human-computer interface management, from a computer science viewpoint, focuses on the process of developing quality human-computer interfaces, including their representation, design, implementation, execution, evaluation, and maintenance. This survey presents important concepts of interface management: dialogue independence, structural modeling, representation, interactive tools, rapid prototyping, development methodologies, and control structures. *Dialogue independence* is the keystone concept upon which all the other concepts depend. It is a characteristic that separates design of the interface from design of the computational component of an application system so that modifications in either tend not to cause changes in the other. The role of a dialogue developer, whose main purpose is to create quality interfaces, is a direct result of the dialogue independence concept. *Structural models of the human-computer interface* serve as frameworks for understanding the elements of interfaces and for guiding the dialogue developer in their construction. *Representation of the human-computer interface* is accomplished by a variety of notational schemes for describing the interface. Numerous kinds of *interactive tools for human-computer interface development* free the dialogue developer from much of the tedium of "coding" dialogue. The early ability to observe behavior of the interface—and indeed that of the whole application system—provided by *rapid prototyping* increases communication among system designers, implementers, evaluators, and end-users. *Methodologies for interactive system development* consider interface management to be an integral part of the overall development process and give emphasis to evaluation in the development life cycle. Finally, several types of *control structures* govern how sequencing among dialogue and computational components is designed and executed. Numerous systems for human-computer interface management are presented to illustrate these concepts.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; *methodologies*; *tools*; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; *structured programming*; *top-down programming*; *user interfaces*; D.2.9 [**Software Engineering**]: Management—*life cycle*; D.3.2 [**Programming Languages**]: Language Classifications; D.3.4 [**Programming Languages**]: Processors; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; I.3.6 [**Computer Graphics**]: Methodology and Techniques; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management; K.6.3 [**Management of Computing and Information Systems**]: Software Management

General Terms: Design, Human Factors, Management, Theory

Additional Key Words and Phrases: Dialogue developer, dialogue independence, dialogue management, human-computer dialogue, human-computer interface, human-computer interface management, interface management, user interface management systems, UIMS

CONTENTS

> As we grow more familiar with the intelligent environment, and learn to converse with it from the time we leave the cradle, we will begin to use computers with a grace and naturalness that is hard for us to imagine today. And they will help all of us—not just a few "super-technocrats"—to think more deeply about ourselves and the world.
>
> [Toffler 1980]

INTRODUCTION

As the "Gestalt of the computer" [Rosenburg 1974] becomes more pervasive in our society, the key to the real effectiveness of computers is usability by people other than computer professionals. As the above quotation suggests, the possibilities of this amazing machine are limited not by its power to compute, but rather by its power to communicate with its human users. Relative to advances in approaches to software design, the important issue of human–computer interface development has begun to be addressed only recently. The increasing interest in this area has been diverse and, at times, disorganized. Although many researchers have proposed viable solutions to specific issues, these issues have generally been addressed without a framework

or a broader strategy for managing the whole development of human–computer interfaces.

A key to building such a framework lies in reassessing the entire software development process, with particular emphasis on development of the human–computer interface as an integrated part of that process. That reassessment has begun, and this article identifies and examines the major concepts in human–computer interface management that have emerged. It uses specific systems to illustrate these concepts, which can be used to classify and describe system features and approaches.

This paper has been a long time in the writing. When it was started in 1982, its intent was to explain some of the basic early concepts in an embryonic field. Most of what is now in the field did not exist then. There was almost no common terminology; even now terminology is not consistently established. The ACM SIGCHI (Special Interest Group on Computer–Human Interaction) was formed during the time this manuscript was being written and revised. Human–computer interaction is now an area of research and practice with broadly recognized impact and increasing rate of growth. Like any survey, this paper is a representative snapshot of the subject at a given point in time.

Scope of the Paper

The focus of this survey is the management of the computer science, or constructional, aspects of human–computer interface development. We do not deal explicitly with interface form and content nor with behavioral aspects of interface development. Consequently, this is not a tutorial on guidelines and principles for creating quality interfaces; rather, it is a presentation of the means—the theories, the methodologies, and the tools—for incorporating dialogue design principles once they are known into human–computer interfaces. In fact, the concepts presented are independent of specific dialogue design principles. *The problem addressed here is not how to construct good interfaces; it is how to provide an environment in which good interfaces can be constructed.*

Psychological models of end-users and aspects of cognition, as well as empirical evaluation of interfaces, are not part of this paper. Although artificial intelligence considerations such as natural language understanding, knowledge-based end-user models, and expert systems are very much a part of the broader subject of human–computer interfaces, especially in the long term, they are excluded from the scope of this paper. The volume of research in each of these areas is such that they merit separate surveys.

Although the viewpoint from which this survey is prepared has a computer science orientation, it addresses concepts and issues beyond specific technical questions. Its intended audience is primarily computer science researchers and practitioners, as well as psychologists and human factors experts who wish to know more about the computer science aspects of human–computer interaction.

Background

The groundwork for development of effective human–computer interfaces was laid during the last decade. Not surprisingly, among the leaders of this early work were some who specialized in graphics [Foley and Wallace 1974; Newman and Sproull 1979]. Literature on human factors and behavioral science research addressed interface design from an empirical perspective [Miller and Thomas 1977]. Most of the computer science work in this period, however, subjectively addressed interface design principles and guidelines [Cheriton 1976; Hansen 1971; Kennedy 1974; Martin 1973; Wasserman 1973, 1981], and this work was not, in general, experimentally validated. Of the hundreds of guidelines compiled from the literature [Smith and Mosier 1986; Williges and Williges 1981], only a small portion have any empirical basis. Also, many contradictions and inconsistencies are found in the literature. Nonetheless, design principles and guidelines are important, and there is a need for

methodologies and tools to facilitate inclusion of these principles in interface design.

The interest in human engineering of computer systems has grown to the point that entire journal issues are devoted to research in this area [e.g., *ACM Computing Surveys* 1981; *Communications of the ACM* 1983; *IBM Systems Journal* 1981; *IEEE Computer* 1982; *IEEE Software* 1989]. Workshops are dedicated to the study of the human–computer interface and its design [e.g., ACM SIGGRAPH 1986; ACM SIGGRAPH 1988; Graphical Input Interaction Techniques 1983; Guedj and Tucker 1979; Guedj et al. 1980; National Research Council 1983; Olsen et al. 1984; Pfaff 1985]. A sequence of meetings in this field has led to the CHI (Computer-Human Interaction) conference and other similar conferences [ACM CHI 1983, 1985, 1986, 1987, 1988; ACM SIGSOC 1981; HCI Hawaii 1984, 1987; INTERACT 1984, 1987; NBS 1982]. Journals and books are specializing in this area [e.g., Card et al. 1983; Coombs and Alty 1981; Ehrich and Williges 1986; Hartson 1985; Hartson and Hix 1988; Moran 1984; Norman and Draper 1986; Shneiderman 1980, 1987; Sime and Coombs 1983; Smith and Green 1980].

This new area of work can, in fact, be considered revolutionary because it has profoundly changed the way human–computer interfaces are developed. It shares characteristics of other computer science revolutions such as those brought about by the emergence of high-level languages, database management, and structured programming. These revolutions have similarly been punctuated with tools having a significant impact on productivity. Each has also had its share of difficulties with initial acceptance. For most cases, however, the cost of not using new concepts and tools eventually became too high to resist.

Human–Computer Interface Management: Terminology for an Emerging Field

In response to the lack of useful, consistent definitions, we present working definitions for key terms, drawing from the sometimes conflicting literature. These definitions are intended to help in understanding the framework of concepts this paper establishes.

Ideally, the terms "human–computer dialogue" and "human–computer interface" (also called the "user interface") are defined separately to denote, respectively, the communication between a human user and a computer system and the medium for that communication. Thus, a *dialogue* is the observable two-way exchange of symbols and actions between human and computer, whereas an *interface* is the supporting software and the hardware through which this exchange occurs. The two terms, however, are tied closely together in the development process, and we shall use them synonymously here just as they are in most of the literature.

Even with a working definition, it is sometimes difficult to identify within a computer-based system what is dialogue and what is computation; there are gray areas between. For example, task analysis and other end-user-oriented modeling involve the entire system's behavior, both dialogue and computation. It might be possible to deduce a model of the entire system's behavior from the dialogue observables, but the terms "dialogue" and "interface" (as used in this paper) refer only to the end-user's inputs and the localized processing of these inputs and to the presentation of the computer's outputs.[1] They do not refer to the functional (algorithmic) transformation of inputs into output—that is the purview of the computational component.

Human–computer interface management or *dialogue management* [Ehrich and Hartson 1981] or *user interface management* [GIIT 1983] refers to the management of the computer science, or constructional, aspects of human–computer interface development, including representation, design, implementation, prototyping, execution, evaluation, and maintenance. User interface management systems (UIMS)

[1] Unfortunately, this use of the terms "input" and "output" favors the viewpoint of the computer over that of the end-user, but their usage seems too well established to be changed.

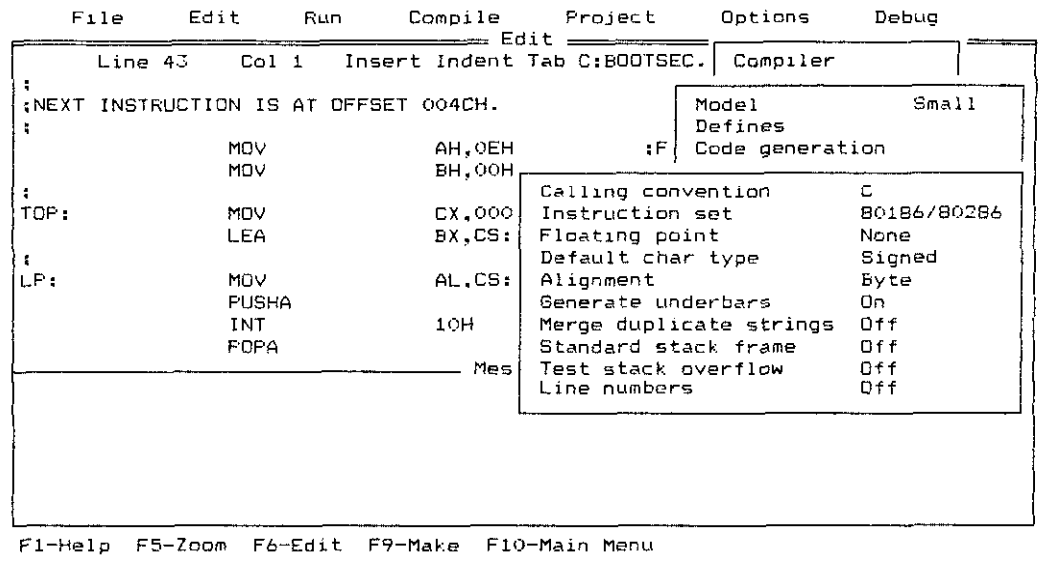


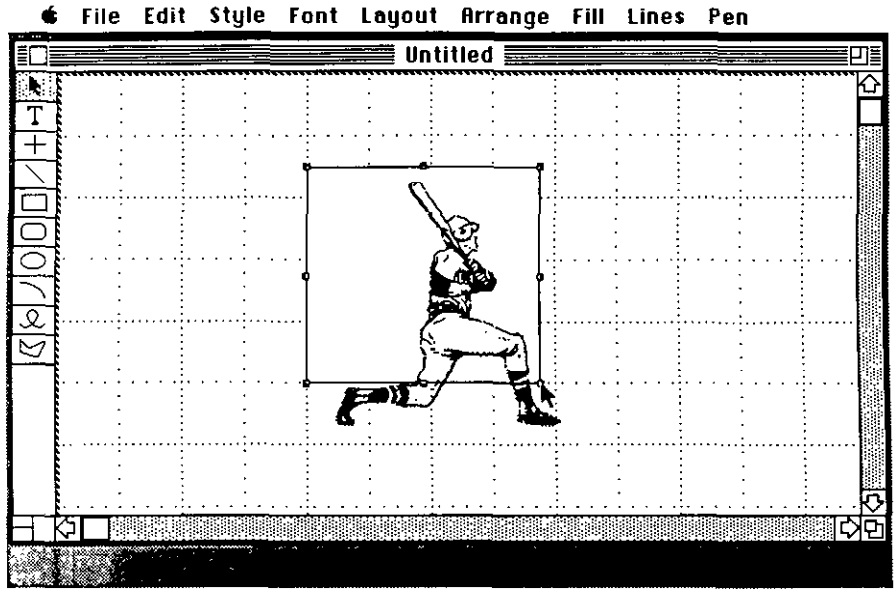
Figure 1. Example of sequential dialogue.

are interactive tools for supporting these interface management activities (see Section 4).

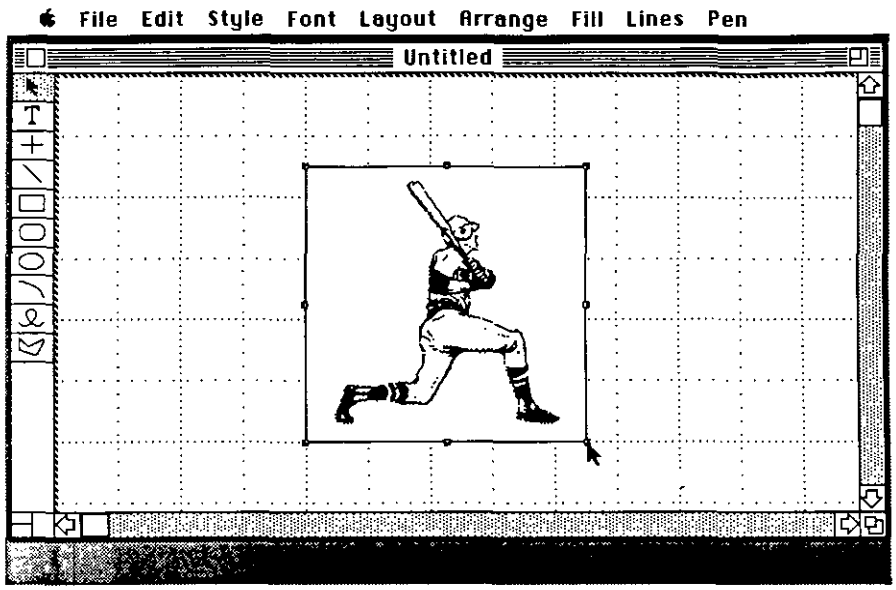
In order to discuss concepts of human–computer interface management, it is necessary to identify basic types of human–computer dialogue. According to Hutchins et al. [1986], there are at least two metaphors that describe ways in which humans interact with computers: the conversational world and the model world. These correspond to two general types of dialogue: sequential dialogue and asynchronous dialogue, respectively. In the *conversational world*, the end-user describes what to do, typically by using a command language. This kind of dialogue is typically called *sequential dialogue*, moving in a predictable manner from one part of the dialogue to the next. Sequential dialogue allows both end-users and developers to visualize specific logical sequencing behavior. Sequential dialogue includes request–response interactions, typed command strings, navigation through networks of menus, and data entry. Figure 1 is a screen from a personal computer running an interactive compiler. This screen results from a linear sequence through a hierarchy of menus. The end-user first selected "options" from the menu bar at the top of the screen, which produced a pop-up menu from which "compiler" was selected. "Code generation" was chosen from the subsequent menu, leading to the topmost menu shown in Figure 1.

In the *model world*, the end-user shows what to do by "grabbing" and manipulating (e.g., with a mouse) visual representations of objects. Thus, *direct manipulation* [Shneiderman 1983, 1987] is used to describe this interaction style. Figures 2a and 2b show an example of simple direct manipulation of a graphical object using a graphical paint package. The box in Figure 2a is not large enough to enclose the baseball player. The end-user can "grab" one of the "handles," in this case the lower right-hand corner (as shown in Figure 2a), and directly stretch the rectangle by moving the mouse until the rectangle is the desired size (as shown in Figure 2b).

Often associated with direct manipulation in the model world metaphor is *multi-thread dialogue*, a task-oriented concept referring to the multiplicity of task paths available to the end-user at any given instant during the dialogue. Figure 3 is a "dialogue box" that exhibits multi-thread dialogue. This dialogue box is displayed in response to an end-user request to open a document (file) from within a commonly



(a)



(b)

Figure 2. Example of direct manipulation dialogue.

used word processing system. The end-user can choose to do any of several different tasks, including scrolling through file names, selecting a file, opening the currently selected file, leaving the dialogue box, changing drives, ejecting a disk, or setting a read only mode. Work on these tasks can proceed in any order, without synchronization among them.

The general term for this kind of dialogue is nonsequential, or asynchronous, dialogue. In *sequential dialogue*, the system

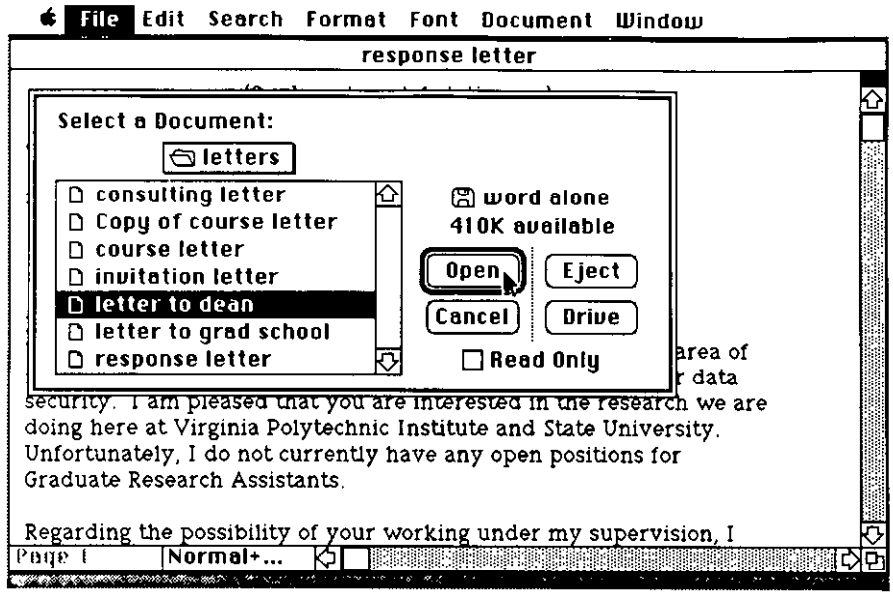


Figure 3. Example of a dialogue box that exhibits multi-thread dialogue.

presents the end-user's work one task at a time. In *asynchronous dialogue*, many tasks (threads) are available to the end-user at one time. Dialogue is asynchronous in the sense that sequencing of each thread is independent of the others. At almost any point in the work on one task, the end-user can switch to another task and, later, back to the first. Asynchronous, multi-thread dialogue is sometimes also called *event-based dialogue* because end-user actions that initiate dialogue sequences (e.g., clicking the mouse button on an icon) are viewed as input events. The system provides responses to each input event. *Concurrent dialogue* is multi-thread dialogue in which more than one thread can be executed simultaneously. While one task is executing, another can be started, overlapping the first. This represents concurrency from both the end-user's and the system's viewpoint. A simple example is a clock on the screen updated by a time-keeping process running concurrently with a word processor.

For conversational parts of a dialogue, end-users interact mainly via the keyboard. For model world parts of a dialogue, end-users interact by manipulating screen objects. A click of the mouse button can be used, for example, to select an object. This model world selection action corresponds directly to the naming of a menu choice, command, or object by the end-user in the conversational world. The large majority of direct manipulation interaction is accomplished by movement of the mouse. An object is dragged to a new location. A corner of a graphical box is moved to change its size and shape. Objects are moved to indicate an operation to perform on them; for example, a file icon can be moved to a new directory or to a trash can icon to request that it be copied or deleted.

The model world or direct manipulation style offers higher usability, in general, for end-users, but both it and the conversational world style are currently used in many interfaces. The trend, however, is toward asynchronous kinds of dialogue. This trend is one of the most significant phenomena in the field today and is exemplified by the Apple Macintosh personal computer. Following the Xerox PARC success of, among others, Smalltalk[2] [Cox 1986; Goldberg and Robson 1983; Tesler 1981], Star [Smith et al. 1982], and

[2] Smalltalk is a registered trademark of Casio, Inc.

the Apple Lisa[3] [Williams 1983], the Macintosh[4] [Williams 1984] introduced multiple windows and direct manipulation dialogues into the mass marketplace. The impact of these innovations and related projects and products has been enormous, changing the complexion of computing. They have become a de facto interface standard and are now copied in other personal computers, workstations, and even mainframes.

Organization of the Paper

As research has progressed in the field of interface management, some concepts have emerged upon which a framework for human–computer interface development can be based. This paper discusses those concepts, which are as follows:

- *Dialogue independence.* The characteristic of an interactive software system that separates the design of dialogue from the design of computational software so that changes in either tend not to cause changes in the other.
- *Structural modeling of the human–computer interface.* The description of the general process of human–computer interaction that can be used to direct the design of dialogue and dialogue development tools.
- *Representation of the human–computer interface.* The techniques used to represent definitions of specific instances of human–computer interaction.
- *Interactive tools for human–computer interface development.* Software and hardware to help automate system development, especially for the human–computer interface.
- *Rapid prototyping.* The process of building executable versions of partially constructed interactive systems to allow early observation of system behavior, especially its interface.
- *Methodologies for interactive system development.* System development processes and life cycles that treat interface development as an equal and integral part of the overall software development process.
- *Control structures.* The organization of dialogue and computational components and the mechanisms that govern logical sequencing and internal communication among dialogue and computational events.

This paper presents a definitional framework that describes each of these concepts and serves as a means for classifying various approaches and systems. Extensive use is made of examples from interface development systems to illustrate the concepts. Some systems reported in this article have aspects that incorporate one or two, but not all, of these interface management concepts. Other systems come nearer to representing all the concepts. The Appendix to this paper is a sampler that describes selected representative systems in the context of the conceptual framework. The presentation of each system in the sampler, given in a uniform format based on the concepts and other features for interface management, shows how that system embodies the concepts. Discussing the concepts in the context of specific systems gives a better understanding of both the concepts and the systems.

1. DIALOGUE INDEPENDENCE

Early approaches to interactive system development typically caused dialogue and computational software to be tightly interspersed. Dialogue became less and less changeable as the design progressed. Despite an enormous commitment of resources, the almost universal result was a poor human–computer interface. Because development of quality interfaces involves an iterative cycle of design and evaluation [Bennett 1984; Chapanis 1982; Good et al. 1984], however, an important criterion for development of human–computer interfaces is fast, easy modification.

Database researchers and designers encountered a similar problem in the need for easy modification of data without the necessity of changing the corresponding programs. The solution that emerged was data independence [Senko et al. 1972], a

[3] Apple Lisa is a registered trademark of Apple, Inc.
[4] Macintosh is a registered trademark of Apple, Inc.

concept that directs the design of data and data manipulation programs so that many changes in either do not necessitate changes in the other. A formal data definition for communication between data and programs allows the decoupling of data instances from the programs. An analogous concept, called *dialogue independence* [Ehrich and Hartson 1981], is based on a formal definition for communication between the human–computer interface and computational programs. Dialogue independence is an approach in which design decisions affecting only the human–computer dialogue are isolated from those affecting only application system structure and computational software. In practice this means that the appearance of the interface to the end-user and choices of interaction styles (e.g., command languages, menus, forms) used to extract inputs from the end-user are not known to the computational software. Dialogue independence is crucial to easy modification of the interface for iterative refinement, as well as ease of maintenance. Almost all modern approaches to human–computer interface management are based to some extent on dialogue independence. Section 1.2 discusses approaches to accomplishing dialogue independence.

1.1 Motivation for Dialogue Independence

Dialogue independence may be most easily understood by first considering its opposite. Without dialogue independence, both the way in which dialogue is structured and the details of how it is conducted with the end-user are directed by the computational requirements of the application system. Knowledge of dialogue details and decisions about interaction styles are intermixed with the computational component. Systems resist modification, and it is difficult to provide for human factors. These difficulties are illustrated in the following small examples.

The first example shows the importance of dialogue independence to flexibility of the interface design. Let us assume a customer specifies that a typed command language be used to convey commands and parameters from the end-user to Information System Z. Without dialogue independence to guide the development process, the program logic for sequencing through the typed commands and the data structures for collecting end-user's choices are programmed into the computational component. It is impossible to change the interface structure or details without affecting the computational component.

Now let us suppose that testing with a particular end-user community revealed a requirement to update the interface design for Information System Z to pull-down menus and dialogue boxes (of the type shown earlier in Figure 3). To replace the typed command string parser in the interface would require significant changes in the overall system code, especially validation of end-user inputs. A system development approach based on dialogue independence, however, would have allowed this change to be limited to the dialogue component and would not have required a change in the rest of the application system routines. Further, a view that recognizes the set of typed command strings as a grammatical structure within a command language could produce two different interfaces—pull-down menus and typed strings, say—that share the same underlying dialogue structure. The suggested changes would then be relatively easy to make, and both kinds of interfaces could be used simultaneously with the same computational component.

The point to be made is that two very different problems are involved: one a computer science problem, the other a human factors problem. With dialogue independence, separate solutions can be generated, each requiring very different problem-solving skills. In the computer science problem, each computational routine needs a set of valid input values, not a conversation with the end-user. The computational function should not care how these input values are validated, how they are entered by the end-user, or in fact where they come from at all. On the other hand, the human factors problem is to design the dialogue to adapt the computer as a tool to perform a task for the end-user. From this perspective the dialogue must be easy to use, accepting input values, providing feedback and

clarification where useful, checking input values against validation criteria, and guiding the end-user from inputs that do not meet the criteria to ones that do. The valid input values are then conveyed to the computational component.

In the next example (not hypothetical), lack of dialogue independence prevented refinement of a software product design to meet human factors requirements such as consistency. A major vendor marketed a multifunction office information system that prided itself on ease of use of its interface. Yet here are instructions for accessing the help facility:

> The comma key on the minikeypad is the HELP key for forms. While in the ABC-Style Editor and Calendar Management, use PF2 for HELP; use "H" for HELP while in the Desk Calculator; use the "GOLD" key plus an "H" key while using the XYZ-Style Editor. . . . By the way, if you need help creating a document, it is better to be in the Word Processing Menu when you press HELP rather than in the main menu . . . It is a good idea to remember the location and purpose of each key mentioned above.

The end-user, of course, must already know how to access help information in order to receive this message. When faced with this example, the vendor sales representative responded that this software package was an amalgamation of existing functions, with code coming from diverse sources, and that the interfaces of each function were so hopelessly interwoven into the application code that it would have been impossible to provide a single consistent interface in time to meet the marketing deadline. The result is obvious: a poor design to which negative customer feedback will eventually force many costly modifications.

The last example shows how dialogue independence is used as a design abstraction to allow a top-down approach, focusing first on high-level design issues while postponing commitment to details. In a hypothetical Calendar Management System (Mantei, personal communication, 1985), four overlapping windows are used to display an end-user's monthly, weekly, daily, and hourly appointment schedule. The developers know from their task analysis that some end-user function will be needed to bring each of these windows on top of the others at various times for viewing and manipulating. Further testing is required to decide what that function will be called and how it will be implemented in the interface. Possibilities include one single arrow key, or NEXT and PREVIOUS keys for toggling or sequencing through the windows. Since each successive window gives more detailed information, a ZOOM key is also a possibility. In the meantime, dialogue independence allows developers to call this function something and embed it in the high-level sequencing logic and computational routines, while deciding much later the end-user's name for the function and details of the dialogue to invoke it.

1.2 Approaches to Dialogue Independence

How, then, can this separation be achieved? On the surface it may appear simple. Programmers sometimes say, "Oh, I've been doing that for years," meaning that they put their end-user error messages in one or two procedures or files separate from the computational routines. Or they mean that all their input and output with the end-user is isolated into a few modules. This is, of course, good programming practice, but it does not ensure dialogue independence. Dialogue-oriented procedures are still linked to the computational code and cannot be modified independently of the rest of the program. Knowledge of dialogue details are often still woven into the application code, and the task of developing dialogue is still basically a programming task.

Dialogue independence is supported by the design-time separation of dialogue from computational software. This means that an interactive application system is composed of a *dialogue component*, through which all communication between the end-user and the system takes place, and a *computational component*, the functional processing mechanisms of the application system with which the human being does not directly interact. These components are kept separate as much as possible during system design, redesign, and maintenance but are bound together for rapid prototyping and execution. Dialogue independence,

however, goes well beyond just separation of the system into components and is not without difficulties or drawbacks. For example, some dialogue development tools (see Section 4) require considerable knowledge of the style of interaction (e.g., menu, use of a mouse) to be anticipated a priori in the dialogue interpreter and design tools. New interaction styles, techniques, or devices will then require significant new programming in these parts of the tools. A second difficulty stems from having a separate role to develop the dialogue component, potentially increasing the need for communication among developers and implementers. New development methodologies (see Section 6), however, are emerging to address this problem. Further, separation of dialogue code from computational code can potentially cause a decrease in performance, especially from increased internal communication among run-time components. This can be overcome to some extent by new system architectures emphasizing, for example, concurrent execution of the dialogue and computational components and by new workstation hardware for dialogue support.

For sequential dialogue, where it is easy to delineate a synchronous "turn-taking" pattern containing end-user input of requests, system computation, and system output of results, physical separation at design time of dialogue-related software and data from computational software is fairly straightforward. For multi-thread, direct manipulation dialogue, where end-users directly, visually, and asynchronously perform operations on interface representations of application objects, separation into components can be more difficult to achieve. This is because the execution of dialogue and computation tends to be more closely interleaved, and the two components often share a common data representation of the interface and application objects. Also in direct manipulation dialogue, there is a need for a closeness of the interface to application semantics (e.g., for semantic feedback in the interface) that works against the separation of dialogue and computation found in dialogue independence. This forces trade-offs in the system architecture [Hartson 1989]. Nonetheless, design decisions regarding appearance and behavior of the interface can often be kept independent of those for the software that manipulates the corresponding data structures. In nearly all cases where it can be achieved, the considerable advantages of dialogue independence appear to outweigh the disadvantages. All the systems surveyed in the Appendix exhibit some form of dialogue independence.

1.3 Dialogue Developer: Separation of Dialogue Creates a New Role

For many years, the two main roles involved in software development were those of the application programmer and the end-user of the system. These two types, however, frequently had severe communication problems. The programmer, impatient to begin coding, had difficulty understanding the end-user's needs. Similarly, the end-user was often not able to articulate requirements for the system and was baffled by the strange "computerese" in which the programmer tried to explain what was happening. The role of systems analyst evolved to provide an understanding of both the technical (programmer) and non-technical (end-user) sides of the system. Also, the role of application domain expert emerged to supplement end-user knowledge about requirements of specific kinds of systems. But neither the systems analyst nor the application domain expert is concerned primarily with the human–computer interface.

In the last few years, human factors specialists have become an increasingly important part of computer system development teams, as advocates of the end-user's need for an effective interface. This has led to a new role that we call the dialogue developer. This same role is also called a dialogue author, a dialogue engineer, an interface engineer, and a dialogue designer. The *dialogue developer* is a human factors specialist concerned with design, implementation, and evaluation of the form, style, content, and sequencing within human–computer interfaces. The dialogue developer's needs and constraints are different from those of the programmer. The dialogue developer is involved in the entire system life cycle,

including task analysis and system requirements specification. During design and implementation of the dialogue, the dialogue developer uses an understanding of psychology and human factors principles to build and iteratively evaluate and refine an interface that supports effective human–computer communication. Often, dialogue independence allows modifications to be made quickly, so that the evaluation and revision cycle can begin again.

Unlike the programmer, the dialogue developer must be sensitive to cognitive needs of the end-user. The dialogue developer role is a cross between a behavioral scientist and a systems analyst. As an analogy, the role of industrial engineer has been successfully introduced to represent a blend of skills that bridge the gap between the psychologist and the machine designer.

1.4 Internal and External Dialogue: Separation of Dialogue Creates a New Interface

Interaction between the end-user and the dialogue component is accomplished through what we call *external dialogue*—the human–computer interface. Separation of the dialogue component from the computational component creates a new interface between these components and a new kind of dialogue through that interface. The computational component, which contains no mechanism for direct communication with the end-user, engages in a less obvious *internal dialogue* with the dialogue component. This new internal interface and its special dialogue are the basis for communication between the dialogue developer and the application software developer at design time and are the focal point for binding end-user dialogue and computation together at run time.

Internal dialogue is not human understandable at execution time, but its formal representation at design time is a key to dialogue independence. Either the end-user interface or the computational software can be changed without affecting the other, as long as both remain consistent with their common internal dialogue representation.

1.5 Dialogue Independence in the Evolution of Interface Management

Evolution of human–computer interface management follows a path from a monolithic approach of programmed dialogue to tool-supported development. Evolution began with device independence, which shields the application programmer from low-level device characteristics. The rudiments of dialogue independence existed at least as early as the 1960s. An approximation to the concept is to be found in some commercial products of that time. The approach was a logical extension to the notions of language independence, data independence, and machine independence, which were then gaining attention. A later example is Digital Equipment Corporation's TRAX operating system, circa 1975, an abortive commercial venture but one that included support for separate dialogue design. The dialogue language called ATL, implemented in TRAX, is a precursor of Digital's Forms Management System, presented in Section 4.

More recent literature contains descriptions of systems for which separation of dialogue from computation was attempted after the system was implemented. That is, generalized end-user interfaces were developed to be used as "add-on" front-ends to existing application systems. Black [1977] was one of the first to develop a front-end dialogue processor for parameter- or transaction-driven application systems. End-user inputs for complex command languages were reduced to a sequence of choices in a tree-structured representation of the grammar.

In Bass and Bunker [1981], the interface for a statistical analysis package operated in both a system command and a job control command environment, and it applied to both batch and on-line jobs. Essentially one interface was adapted to this diversity of use in a single application. In Wright and Brown [1978], the interface was custom coded to a single, specific medical application. Both these groups began with an existing application system, which did not have an easily usable interface, and attempted to revise the human–computer

dialogue to meet their needs. Thus, these systems achieved some separation of dialogue and computation, but not true dialogue independence, since the results were not generalizable and extensible to other systems or even to other interfaces for the same systems. These approaches also did not use an overall development approach that included consideration of a separate computational component.

The work on a demographic database system reported in Evans et al. [1981] is somewhat similar. Here an adaptable end-user interface provided more than one dialogue to an existing software system as needed to suit the varying requirements of different end-user communities. Separation of the interface code from the rest of the system made this possible.

Display management, which provided general development tools for parts of the end-user interface, with emphasis on screen displays, led to application generators. Application generation was—and is—an approach for increasing productivity in the implementation of interactive systems by partially automating code production for specific kinds of applications. Application generators represent an evolutionary step in which the concepts of human–computer interface management, especially dialogue independence, began to take tangible form. In application generators, special-purpose high-level languages are used to produce display screens (e.g., menus and forms) and accept inputs (including command languages and data entry). Dialogue independence depends on the appropriate separation of resulting code modules. Syntactic and lexical details can be isolated because other modules do not need knowledge of how these modules obtain end-user inputs.

From application generators, the evolutionary path led to user interface management systems, or UIMS, and other interface development tools. In UIMS, the concept of dialogue independence is explicitly recognized and supported. Most UIMS are based, at least to some extent, on dialogue independence. This is true especially of those for developing sequential dialogue, but asynchronous dialogue creates some problems in maintaining dialogue independence (see Section 7). Several primarily research-oriented systems incorporated dialogue independence into their approach to application development. Hayes et al. [1981] referred to the independence of the end-user interface from the application program or end-user's tool as "tool independence." Many application systems (end-user tools) share the development cost of this single intelligent interface system. Since dialogue independence allows more than one dialogue component for a single computational component, an application system can have two or more very different end-user interfaces. Foley [1981] and Feldman and Rogers [1982] have captured this concept in their Abstract Interaction Handler (AIH), which contains knowledge of interaction styles, allowing their style-independent applications to be used with more than one kind of interface. Dialogue independence has also been an important driving force for both theoretical and implementational development of the Dialogue Management System (DMS) [Ehrich and Hartson 1981]. In DMS, surface details of an interface are decoupled from its deep structure through levels of abstraction. Work by each of these three research groups (COUSIN, GWUIMS, and DMS, respectively) is detailed in the Appendix.

Dialogue independence is less important in the context of toolkits, which are libraries of routines for implementing human–computer interface features. Toolkits are compatible with dialogue independence but do not necessarily support it; maintaining dialogue independence is incumbent on the application programmer who is using the toolkit.

Numerous systems that represent these important steps in the evolution of human–computer interface management are discussed in Section 4 and in the Appendix. They embody various approaches to separation of the dialogue from the computational software of an application system, basic to the concept of dialogue independence and of human–computer interface management.

2. STRUCTURAL MODELING OF THE HUMAN-COMPUTER INTERFACE

2.1 Types of Interface Modeling

There are many kinds of modeling applied to human–computer interaction; three of the most prevalent are for task analysis, structural description, and interface representation. The first kind, task-oriented modeling, is used to analyze and describe the details of a particular end-user task, often by hierarchical decomposition into levels of subtasks. Task-oriented models are typically used to drive the process of design for specific interfaces and often include a description of the knowledge an end-user has or needs about the task and how to perform it [Kieras and Polson 1985]. At detailed levels, task descriptions are very dialogue and device dependent, being specific to keystrokes and other actions by the end-user of a particular interface. Although they may or may not be structural models of the computing task, task-oriented models are not structural models of human–computer interaction. Therefore, although such models and their analysis are important to the dialogue development process, they are outside the scope of this paper and will not be discussed further.

Directives from workshops on human–computer interaction mandate a need for "a model of interaction and a language for specifying end-user interactions . . . which have been subjected to experience in real-world applications" [GIIT 1983]. This quote refers to the other two areas of modeling, which we feature as concepts of interface management. *Structural models of the human–computer interface* are descriptive of the general process of human–computer communication; that is, they theoretically and generically describe the structure of end-user exchanges with computers. For example, some of these models identify dialogue objects, such as prompts, inputs, validations, echoes, messages, and their relationships. Such models guide a dialogue developer and help organize the dialogue development process. In contrast, *interface representations (specifications)* are schemes for representing particular instances of human–computer interaction; that is, they are used by dialogue developers to describe details of form, content, and sequencing for parts of a specific interface design. Methods for representation can be based on a structural, descriptive model. In such cases the structural model can guide the developer during the process of representing the dialogue design and can guide readers of the recorded design. Structural, descriptive models are the subject of this section, and interface representation schemes are discussed in Section 3.

Some models tend to overlap both these types of models. An example is the Command Language Grammar (discussed in Section 3.4), which cuts across several types of models because of the level of detail it is capable of representing.

Much as it was in the early stages of software engineering development, current approaches to human–computer dialogue design are often ad hoc and unstructured. This lack of a framework for the constituent parts of human–computer interaction leads to dialogue development procedures that are also ad hoc and unstructured. In many interface development tools (see Section 4), the model of dialogue is implied and must be inferred by tool users. In other tools, the model of dialogue is explicit and provides terminology and organization upon which to build dialogue designs. *Structural, descriptive modeling of the human–computer interface* is a fundamental concept of interface management, necessary to understanding the nature of human–computer interaction and therefore necessary to the interface development process. All structural models discussed here appear to be for describing sequential dialogue. A sequential model, however, might be used to describe each thread of a multi-thread dialogue, with implicit movement among threads governed by a high-level controller. Even this does not capture the true asynchronous nature of a multi-thread dialogue. Research on structural modeling of asynchronous dialogue is still embryonic; it is difficult in large part because such dialogue is less structured than sequential dialogue.

2.2 Linguistic Models

2.2.1 Dialogue as Languages

Human–computer dialogue, especially sequential dialogue, can formally be modeled as the grammar and vocabulary of a human–computer "interaction language" [Foley and Wallace 1974]. Content and format, as well as logical sequencing, of sequential dialogue is extremely important in determining how well an end-user can understand and manipulate the system. To understand the idea of dialogue as an interaction language, consider the use of an ordinary command language. Each typed command line is accepted, lexically analyzed, parsed according to a grammar, recognized as either a valid command or an error, and acknowledged as either a valid command (sometimes implicitly through the presentation of the next prompt) or an error. The action requested by the command, if valid, is performed. There is no difficulty in seeing this kind of interaction sequence as one involving language that can be formally described by a grammatical definition.

Depending on the interaction style, tokens expressing the end-user's needs can each be conveyed in different lexical and syntactic forms—menus, programmed function keys, touch panels, voice input/output, graphical picking of icons (e.g., by a mouse), and ordinary question-and-answer text. For example, a direct manipulation style such as the mouse can be used to build up the same kind of command found in a typed command string by selecting a function and then selecting, one at a time, its operands and options. The system may coach the end-user for each item, and the command syntax is not as apparent. Nevertheless, even if commands and operands are selected in a direct manipulation style, each of the resulting tokens can be seen as part of a command that is also representable in a formal grammar definition. A linguistic model of dialogue is useful for seeing beyond the surface differences in dialogue form and dealing with similar interaction structures in a uniform manner.

The idea of language is involved in both structural modeling and interface representation, but in different ways. In the first case, structural models typically relate to the language of the end-user, that is, the interaction language in which the human communicates with the computer. In the other case, an interface representation language used by the dialogue developer is a metalanguage for defininining the end-user's interaction language. The idea of viewing an end-user interface from a linguistic viewpoint—at conceptual, semantic, syntactic, and lexical levels—was pioneered by Foley [Foley 1980; Foley and van Dam 1982; Foley and Wallace 1974] and appears in the GWUIMS in the Appendix. The "conceptual level" is the collection of basic system goals and functions that an end-user must understand. The "semantic level" encompasses input operations and output presentation techniques. The "syntactic level" contains specific token sequences to invoke semantic actions, as well as specific form and content of output. The "lexical level" defines token structure in terms of hardware.

The lexical level of Foley's conceptual, semantic, syntactic, and lexical levels has been further decomposed into two levels: lexical and pragmatic [Buxton 1983]. Foley's lexical level encompasses a broad range of diverse features, including composition of tokens, spatial display concerns, devices, and physical gestures. Buxton's "lexical level" addresses only token composition information, whereas the "pragmatic level" subsumes issues of layout, devices, and gestures. Buxton states that this pragmatic layer is the main level of interaction between a human being and computer system. It therefore has the greatest influence on an end-user's perception of the system and should be given special attention.

2.2.2 A Dialogue Transaction Model

The "dialogue transaction model" [Hix and Hartson 1987; (Johnson) Hix 1985] is a descriptive model of the structure of human–computer interaction, providing the framework for designing, representing, and

implementing interfaces using the Dialogue Management System [Hartson et al. 1984]. The model is based on simple relationships between formal languages and state machines and is empirically derived from observations of many interface styles and techniques.

In the dialogue transaction model, a "linguistic object" is an identifiable entity in the observable symbols of dialogue. The principle linguistic object is a "token," an abstraction representing the smallest unit of end-user input that can have formally defined meaning in terms of the application or task. An example is a simple data value or command that can be entered, say, as a series of typed keystrokes, a menu choice, the press of a programmed function key, a mouse selection of an icon, or a word to a voice recognizer. Each linguistic object in the dialogue is processed by a corresponding "constructional object." A token is processed by a constructional object called an "interaction," which is a dialogue function that maps a "raw" (uninterpreted and device dependent) end-user input to a validated and "normalized" token value. Token values are validated within interactions according to their lexical definitions and constraints. A normalized token value is a device independent and interaction style independent value that is globally understood by the rest of the application system.

There are some cases in which more than one end-user action is required to express a token. For example, a command name might be typed on an alphanumeric keyboard or a multidigit numeric value picked by a mouse from a picture of a calculator keyboard on the screen. Each separate character is an instance of the linguistic object called a "lexeme," the smallest unit of raw input from the end-user. Lexemes are processed by a corresponding constructional object called an "action," so named because it is one to one with end-user actions.

In addition, tokens themselves can be grouped together in semantically related sequences called "sentences." Sentences are processed by constructional objects called dialogue "transactions." The set of all valid sentences expressible by an end-user comprises the end-user's "transaction language." Relationships among tokens and constraints relating their values make up the grammar of the transaction language.

Each constructional object is composed of "constituent objects," such as "display objects," that can be static (completely defined at design time) or dynamic (not bindable until execution time); "input objects," used to accept, but not validate, end-user inputs; and "dialogue computation objects." Dialogue computation objects perform computation directly in support of dialogue, such as computing default token values for an interaction, validating end-user input against predefined lexical and syntactic criteria, or mapping raw token values into normalized token values. The hierarchical relationship among constructional objects—transaction, interaction, and action—serves as an aid in organizing dialogue into levels of abstraction, each level helping control complexity by hiding detail of levels below it. Figure 4 shows a typical configuration of constructional model objects and their constituent objects.

The dialogue transaction model is well suited for sequential dialogue, which usually has a linguistic structure among parts of the dialogue relating to commands, parameters, selection of choices, data entry, and values requiring parsing and/or validation. The model has also been applied to direct manipulation style dialogue, which involves entry of token values by the end-user, but is not typically amenable to linguistic structuring.

2.3 Nonlinguistic Models

2.3.1 Dialogue Cells

A "dialogue cell" has been developed as a nonlinguistic-based model for describing and developing sequential human–computer dialogue [Borufka and Pfaff 1981; Borufka et al. 1981, 1982]. A dialogue cell consists of four basic elements that define the dialogue structure, as shown in Figure 5. The "prompt" prepares the system for end-user input actions and indicates the type of data to be entered by the

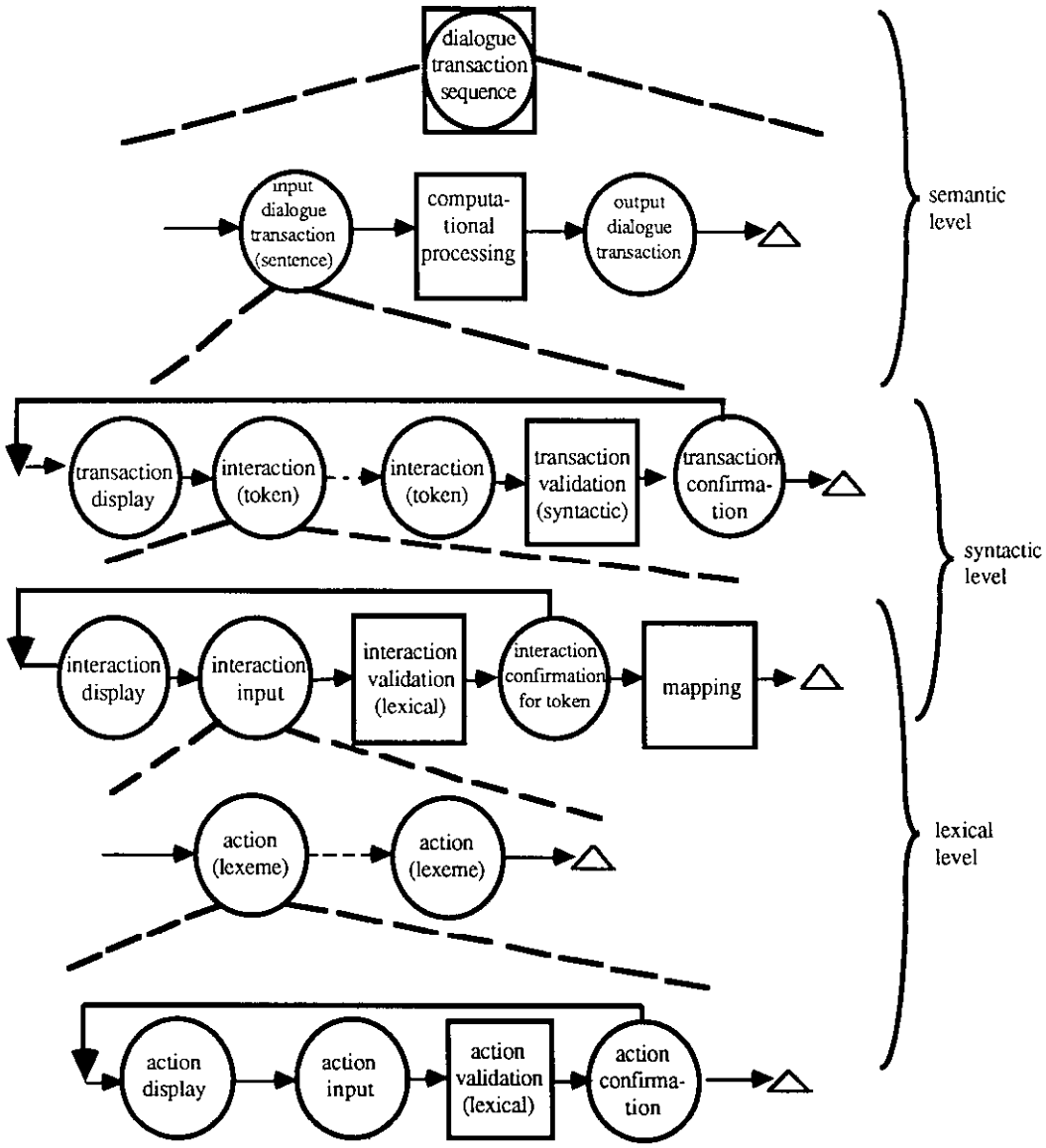


Figure 4. Dialogue transaction model (adapted from Hix and Hartson [1987]).

human. The "symbol" is the input of the end-user; it is a construct for specifying type and value of that input. The "echo" is the system interpretation of the symbol entered by the end-user. The "value" is the mapping of the input symbol to data usable by the application program. In sum, a dialogue cell is a unit that describes sequential dialogue interaction with an end-user, including information to the end-user, end-user input action, evaluation of end-user input, echoing of end-user input, mapping from end-user input to value, and delivery of the resulting value to the computational component.

The four parts of a dialogue cell (prompt, P; symbol, S; echo, E; value, V) are connected as shown in Figure 5. This figure also shows the order in which cell elements are developed. Cells are initialized through

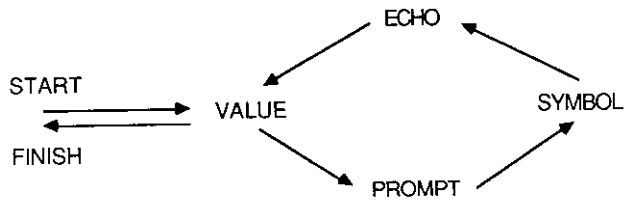


Figure 5. Dialogue cell (from Borufka et al. [1982]). Reprinted with the kind permission of H. G. Borufka. © 1982 IEEE

an initial value $v(i)$ and produce a return value $v(o)$. A dialogue cell for input has a nonempty return value $[v(o)]$ for the value entered by the end-user. A dialogue cell for output has a nonempty initial value $[v(i)]$. Basic dialogue cells (consisting of P, S, E, and V sets) and their elements can be combined hierarchically to represent sequences of human–computer exchanges. In such an organization, cells have parameters for passing data, much like parameter passing in conventional programming languages.

A dialogue developer creates dialogue, but in a "dialogue language," programming it much the same way an application programmer would. Dialogue cells provide a useful framework for dialogue programming; this framework revolves around defining an interface in terms of its basic dialogue elements and defining the structure for handling interactions at both global and local levels. That is, a human–computer interface is structured into cells, and cells are structured into prompts, symbols, echoes, and values. Tools provided for a dialogue developer include an input/output system for accessing graphics devices (through the Graphics Kernel System) and a dialogue language for specifying data structures and control flow. This approach may be used conceptually, even in the absence of a system for executing dialogue cells, to produce human–computer interfaces.

2.3.2 *Interaction Events*

An "interaction event" has been proposed as the basis of another nonlinguistic interface model [Benbasat and Wand 1984]. The basic premise is that sequential dialogue is composed mainly of a series of "interaction events." Such an event is composed of a system "prompt," an end-user "input," a system processing "action," and "flow control" to determine the next interaction event. Through the prompt, the system indicates to the end-user that it wants an input. The end-user then provides the input, and the system responds with an action based on the input. System flow of control decides which interaction event will follow. Other dialogue features include "input checks" to validate end-user input, a "help" feature invokable by the end-user, an "escape" mechanism that allows the end-user to skip the current input request, and "default values" that are responses assumed by the system if the end-user provides no input. The complete "interaction cycle," composed of these events, is shown in Figure 6. Interaction events are executed by a dialogue generator. A fully functional dialogue generator has been implemented, the execution of which is based on a tabular form of interaction events called "reference sets."

This model is quite thorough, and examples of text, check, action, and flow control tables, which the dialogue generator processes, are presented in Benbasat and Wand [1984]. Interaction events, however, appear descriptive only of sequential dialogue. The definition of the end-user input seems, at least from the examples, to be limited to simple data typing, without provisions for more complete aspects of input definition. Also, the definition of an interaction event encompasses more than just the dialogue. The action component and some of the flow control component are really part of the computational and global control components of the system, stretching the domain of the model into the

1. Prompt.
2. Input (get from user or use default).
3. Escape: if Input = "escape," then
   1. Set "next event" indicator
   2. End cycle.
4. Help: if Input = "help," then
   1. Display additional information
   2. End cycle.
5. Check: apply input checks. If errors, then
   1. Report errors
   2. End cycle.
6. Action: invoke related processing.
7. Flow Control: set "next event" indicator.

Figure 6. Interaction cycle (from Benbasat and Wand [1984, p. 108]). Reprinted with permission from Izak Benbasat.

complete human–computer system, not just the interface. Despite this, the basic benefits of this research are sound: practically, to produce a dialogue generator to facilitate human–computer dialogue implementation and, theoretically, to provide a better understanding of these dialogues through a common set of model components.

2.3.3 Other Nonlinguistic Models

In the Graphical User Interface Design Environment (GUIDE), a UNIX[5]-based dialogue design system from the University of Glasgow [Gray and Kilgour 1985], the components of a hierarchical model of dialogue are mapped to the UNIX file management system. Dialogue is described in "dialogue scripts" that are sections of (mostly textual) UNIX files. UNIX directories correspond to major dialogue units (e.g., for processing whole commands). Subdirectories represent smaller components such as prompts, echoes, and responses. A dialogue interpreter executes dialogue by traversing a script. In this environment, tools are available to an end-user as well as a dialogue developer.

An interface processor is the basis for another human–computer interaction model [Edmonds 1982]. This interface processor consists of input, output, and dynamic processes, which perform simple transformations (e.g., keyboard input to character strings) and determine actions of the computer. This model appears to dwell on physical processes at the expense of providing insight into the essential nature or structure of human–computer communication. And although the article postulates that, using this model of an interface, "we could clearly arrive at . . . a description of the system" and " . . . arrange that the end-user's model matched construction of the interface," the means for accomplishing this was not explained.

One working group of the Seillac II Conference [Guedj et al. 1980] proposed a high-level model of interaction based on a processing paradigm. A control level and a performance level between human and computer represent, respectively, "what" interaction occurs and "how" it occurs. Although basically sound, the model, because of its high level of abstraction, provides little insight into the specific task of designing interfaces.

Norman has proposed four distinct stages of human activity during interaction with a computer: intention, selection, execution, and evaluation [Norman 1984]. Because each stage has different implications for system design, different supporting tools are needed. The premise is that the four stages can be used to guide screen design (e.g., evaluation is essentially a feedback stage, so appropriate information should be given to the end-user). Although the stages appear realistic, they seem to lack specific means for direct application to organizing the interface development process.

2.4 Architectural Abstractions

Some models, rather than being directly descriptive of human–computer interaction, are architectural descriptions of how the human–computer interface relates to the rest of an application system.

The "Seeheim model" [Green 1985; Pfaff 1985] of Figure 7 is a run-time architectural model of human–computer dialogue. The "presentation component" contains device-dependent details and specifics of displays, as well as interaction style descriptions.

[5] UNIX is a trademark of AT&T Bell Laboratories.

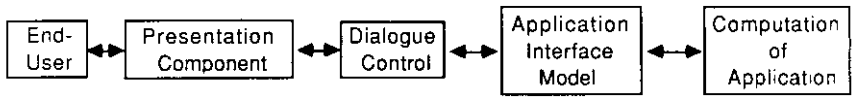


Figure 7. Seeheim human-computer interface model (adapted from Green [1985]). Reprinted with the kind permission of Mark Green.

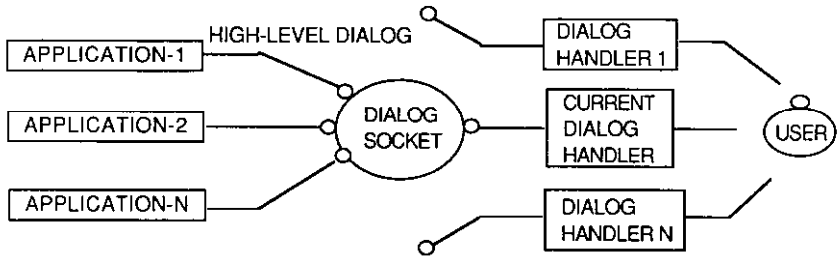


Figure 8. Dialogue socket (from Coutaz [1985, p. 30]). Reprinted with permission from J. Coutaz (Laboratoire Génie Informatique, Ihag, Grenoble, France). © 1985 IEEE

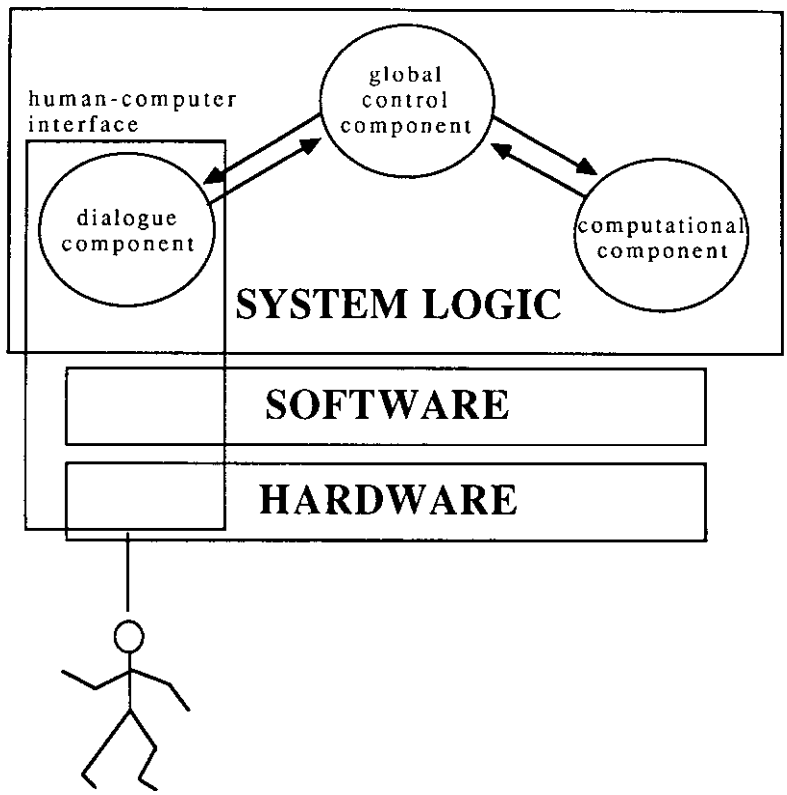


Figure 9. DMS application system architecture.

The "dialogue control component" does dialogue processing and sequencing, while the "application interface model" contains the application view of the interface and the interface view of the application. Communication with the application is via procedure calls and data structures, described at an abstract, implementation independent level.

In the "dialogue socket model" [Coutaz 1985], toolkit abstractions are used to relieve a dialogue developer from dealing with low-level details of interaction. The dialogue socket is a high-level abstraction that connects lexical and syntactic specifications of the dialogue with an object-oriented view of lower level input and output. As shown in Figure 8, each terminal or workstation has its own device-dependent dialogue handler that plugs into one side of the dialogue socket, while the application plugs into the other side. Both sides are agents that perform operations on shared interface objects. The socket maps dialogue from the lexically and syntactically specific dialogue handler to the object/operation view required by the application. The socket becomes a "virtual user" to the application.

Architecture of an application system produced using the Dialogue Management System [Hartson et al. 1984] approach is shown in Figure 9. Logic of the application system is broken into three components. The "computational component" contains semantic functionality of the application system but contains *no* dialogue. The "dialogue component" is composed of dialogue transactions (see Section 2.2). These contain all dialogue functionality, logic, and content, such as displays, error messages, and input processing. The dialogue component also contains some computation but *only* computation that directly supports dialogue, such as validation of end-user input. It does *not* contain any semantic computation of application system functionality. The "global control component" governs logical sequencing among dialogue and computation, invoking dialogue and computation as needed at run time.

3. REPRESENTATION OF THE HUMAN-COMPUTER INTERFACE

Dialogue developers require a mechanism for expressing and recording their designs. Numerous techniques have been used to support the concept of *representation of the human–computer interface*. Among the earliest methods were, of course, written programs. Since then, other mechanisms have emerged, including textual and graphical representation languages. Recently, systems for dialogue development have begun to provide automated tools for interactive production of the interface representation. This section presents representation techniques; discussion of tools here is limited to illustration of these techniques. Tools are discussed in a broader context in Section 4.

3.1 Issues in Representation of the Interface

3.1.1 Metalanguages

Metalanguages—languages for representing other languages—have several well-known problems. They often involve notations that are almost unreadable to the average person. Formal language definitions meet with resistance, especially from those who are more pragmatic, because they are so cryptic and often difficult to understand. It is sometimes impossible to separate metalanguage symbols from symbols of the language being defined. Numerous language representation techniques have been developed for programming and command languages, including Backus–Naur Form (BNF), regular expressions, context-free grammars, state transition diagrams, and Petri nets. As interaction languages evolved, the need for their representation became apparent. Researchers tried to use many of these existing means of language representation usually with only limited success.

Most well-known methods of language syntax representation are useful primarily for static programming languages. They are not powerful enough for expressing all concepts of programming languages (e.g., context sensitivity and semantics), not to

mention representation of the dynamic aspects of interaction languages [Jacob 1983]. Even for ordinary sequential dialogue, such representational methods must be augmented with other techniques. Language-oriented representational techniques are largely inappropriate for representing direct manipulation style dialogue. Although the direct manipulation interface paradigm is one of the most popular, it is also one of the most difficult to represent, largely because of its highly interactive nature.

The problem of interface representation goes beyond language concerns. For example, there are many visual and other perceptual aspects to be represented. BNF and state transition diagrams are primarily means for representing grammatical relationships (e.g., logical sequencing of a command and its parameters) among end-user inputs. But neither BNF nor state diagrams show the process of how, for example, the command is solicited by the system (e.g., the appearance of a menu or set of graphical icons on a screen) and entered by the end-user (e.g., by typing a choice code or picking an icon) or how the system responds with semantic feedback (e.g., changes in the cursor during dragging). Additional techniques are required to represent these. In RAPID/USE [Wasserman 1985], for example, arcs of state transition diagrams are used to show grammatical connections among nodes, but screen appearance and input mechanisms within each node are represented by a textual dialogue programming language.

Jacob [1983] has done an extensive survey and thoughtful comparison of techniques for interface representation and specification. Two classes of techniques are most prevalent: those based on BNF-type definitions and those based on state transition diagrams. His comparison of those two techniques concludes that state transition approaches provide more comprehensible language representations because they show time sequencing and surface structure of the human–computer interface more directly than BNF does. They are therefore a better cognitive match to the programmer's and the dialogue developer's mental models.

Another comparison has shown similar results [Guest 1982]. One tool, a powerful syntax-directed translator (SYNICS), was compared to a dialogue description language. SYNICS had an input structure based on BNF-like production rules, whereas the dialogue description language was based on an approach similar to state transition diagrams. The results pointed to the diagrams of the dialogue description language as a much easier method of defining dialogue than production rules. The explanation for this was that programmers found writing production rules more difficult than creating transition diagrams. Production rules are declarative, but most of the programmers tested tended to think and code procedurally.

3.1.2 Completeness of Representation

It is desirable to have a physical and notational process for recording results of the dialogue developer's mental process of conceptual development. It appears that no single representational technique will suffice. Rather, a set of techniques is required for recording behavioral, structural, and detailed representation of both visible and nonvisible aspects of human–computer interfaces.

Further, these representational techniques must serve all developer roles throughout the system life cycle, applying to the behavioral domain of human factors experts and end-users, as well as the constructional domain of system developers. Ideally, the techniques should have a sound formal basis, be independent of tools through which they may be implemented, and be complete in their ability to represent interfaces. For the dialogue developer, details can be overwhelming: end-user navigation and sequencing, grammar and other syntactic constraints, lexical rules for input, appearance of displays (e.g., graphics, positioning, clearing screen, character-by-character cursor movement, echoes, highlighting, color, movement of objects), message content and format, device and interaction style dependencies, data flow, data typing, semantics, conditional and adaptive behavior, scrolling, paging, windowing, and so on.

Representational needs even extend to the development process itself. Several kinds of information are usually lost in the development process but are needed during maintenance such as: Why was a particular design decision made? How much time did developers spend on various parts of the system, especially the interface? What is the version history of a particular feature? How can the satisfaction of system requirements be traced to specific system modules?

The problem of completeness in interface representation is still unsolved, but new techniques such as scenarios and prototypes are successfully being used to augment existing, more formal methods.

3.2 Techniques for Representation of Sequential Dialogue

3.2.1 BNF Representation

One of the best known systems for representing the syntax of a language is the BNF [Naur 1963]. BNF, however, has several deficiencies in its power to represent languages, particularly its inability to represent context sensitivity. In addition, BNF is difficult for humans to understand. It is a highly structured, hierarchical metalanguage that results in a "fan-out" problem. That is, nonterminals in an expression can be replaced by more nonterminals through several successive iterations before a terminal symbol is finally reached. This multilevel tree structure is difficult for human beings to follow, since by the time the leaves (terminals) are reached, the root (highest level expression) and the language structure may long be forgotten. It is consequently very difficult to visualize sentences in a language by looking at its BNF definition.

Nonetheless, BNF has been used extensively in representation of human–computer interfaces. Simulation systems have been developed that accept as input BNF production rules with associated actions and produce a prototype of the human–computer interface [Hanau and Lenorovitz 1980a, 1980b].

Extended LL(1) grammars with added graphical information have been used as interface representations [Olsen 1983; Olsen and Dempsey 1983] in the SYNGRAPH system for automatically generating interactive systems. SYNGRAPH (discussed in Section 4) generates the end-user interface for interactive graphics applications. The BNF-like definition of the interaction language as well as Pascal code that is invoked to perform the related semantic actions are both used as input to the generator. Output is a recursive descent parser for the interaction language, as well as a scanner and a screen manager. SYNGRAPH has produced the notion of an "interactive pushdown automaton" as the basis for describing the interface syntactic components [Olsen 1984b]. Although this system relieves a developer of having to code the interface, the grammar that describes it must still be produced.

MIKE, an outgrowth of SYNGRAPH, can be used to generate text-based interactions; interaction languages are represented as Pascal procedures and functions. Representing the interface with these expressions differs significantly from the SYNGRAPH grammatical approach and has proved to be much easier to learn.

One variant of BNF, adapted specifically to represent interaction languages rather than static languages, is the multi-party grammar [Shneiderman 1982]. Features that differentiate this extension from standard BNF are the labeling of nonterminals with a party [i.e., either human (H:) or computer (C:)] identifier, assignment of values to nonterminals when appropriate, and definition of a nonterminal that will match any input string if no other parse of that input is successful. The grammar permits terminal strings entered by the end-user to be fed back in a later part of the dialogue. Other characteristics peculiar to interactive displays, such as visual features, are also specifiable. A small example for an "open file" command is shown in Figure 10. Note the nonterminals (in ⟨ ⟩), labeling of nonterminals (with H: or C:), and the dialogue variable (FILENAME) used in the computer's "OPEN-ACK" response to the human.

BNF-based representation methods should not be considered structural models of human–computer interaction in the

< CMD> :: = < H : OPEN> <C : OPEN-ACK>

< H : OPEN> :: = OPEN <H : FILENAME>

< C : OPEN-ACK> :: = [<H : FILENAME>] IS NOW OPEN

Figure 10. Example using multi-party grammar representation. (adapted from Shneiderman [1982]). Reprinted with permission from Ben Shneiderman. © 1982 IEEE

sense that we defined such models in Section 2. The metalanguage symbols of BNF do not provide a structural organization or explanation of the nature of this interaction. BNF notation (e.g., the multi-party grammar) is, rather, a syntactic notation for representing specific instances of a dialogue.

3.2.2 State Transition Diagram Representation

State transition diagrams (essentially finite state machines) constitute another formal representation technique frequently used for language definition. Whereas the primary means of creating BNF descriptions is textual, state transition diagrams are a graphical means of representing sequential dialogue, using graph nodes for states and arcs for sequencing of transitions among states. Since conditions upon which transitions are made depend on end-user inputs, state transition diagrams can be used for representation of interaction languages.

One of the earliest uses of state transition diagrams for language representation was in specification of a compiler for a programming language [Conway 1963]. Actions associated with each state transition indicate what is to happen when the transition occurs. An early use of state transition diagrams for interface representation is found in Newman [1968]. Use of state transition diagram representations for the design of interactive computer systems [Parnas 1969] evolved in response to the increased need for consideration of human interaction in the system design process. In particular, state transition diagrams specify appropriate messages at each state of an interactive system. Augmented transition network (state transition diagrams supplemented with stacks) grammars have been used to analyze natural language structure [Woods 1970]. Current research has now progressed far beyond this point, but these ideas were quite novel when first proposed.

Wasserman's RAPID/USE [Wasserman 1980, 1982, 1985; Wasserman and Shewmake 1985; Wasserman and Stinson 1979; Wasserman et al. 1986] is a system for representing not only the end-user interface but an entire interactive information system. Transition diagrams are used to describe the language of the end-user and for production of rapid prototypes [Wasserman and Shewmake 1982]. Jacob [1985] presents the Military Message System (MMS) as an example of a system developed using state transition diagrams, with associated actions, as a formal representation technique for its interface. These representations are then converted into system prototypes. In both these approaches the machine representation of the state transition diagrams is an interpretable textual encoding in a "node and arc" language.

Figure 11 shows an example of a USE transition diagram. Transition diagrams can be produced, using an interactive transition diagram editor, and then be automatically converted to the skeleton of the textual code (i.e., the node names and control flow). Contents of the nodes are then filled in by a dialogue developer using a dialogue programming language. Figure 12 shows some of the corresponding textual encoding for Figure 11.

Jacob's [1985] representation of dialogue is heavily based on semantic, syntactic, and lexical levels, using separate diagrams for each level. State transitions are associated with an input or an output token, but not both. That is, output is treated as a separate token, rather than as a special action, allowing representation of output dynamics. Output tokens include prompts, acknowledgments, and echos. Through a process of stepwise refinement, states are added to the state transition diagrams, making representation of the interface more precise. The resultant representations are detailed and voluminous, providing device independence and screen and cursor control. Messages can be constructed independently of node definitions. Subdiagrams are used to control complexity by providing modularity

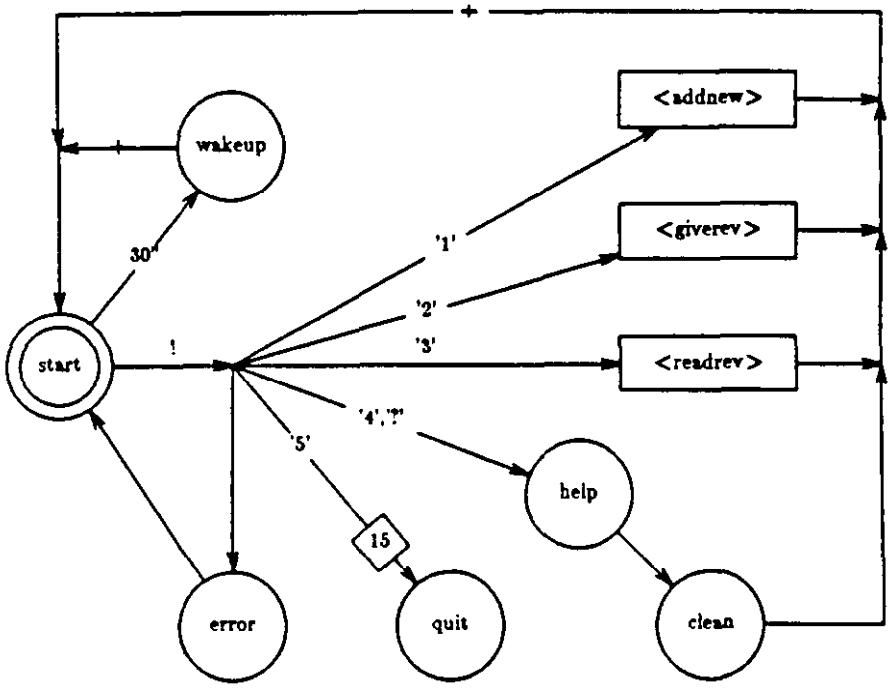


Figure 11. USE transition diagram (from Wasserman and Shewmake [1985, p. 196]). Reprinted with permission from the authors and Ablex Publishing Corporation.

and an ability to decompose designs into levels of abstraction. Both Wasserman's and Jacob's approaches are presented in more detail in the Appendix.

A representation technique that is similar to state transition diagrams was used to describe primarily sequential dialogue in early versions of the Dialogue Management System. The SUPERvisory Methodology And Notation (SUPERMAN) embodies both data flow and control flow in a unified graphical system representation [Yunten and Hartson 1985]. SUPERMAN represents design of an interactive system with a "supervisory structure," which is a network of "supervisory cells." A supervisory structure is shown in Figure 13. Each cell contains a single "supervisory function" and a "supervised flow diagram" (SFD) representing both control flow and data flow among its dialogue and computational functions.

The graphical function symbols of SUPERMAN reflect SUPERMAN's emphasis on separation of dialogue from computation. A circle inscribed in a square is a dialogue–computation function, always a supervisory function, that eventually decomposes into pure dialogue and pure computation. A dialogue transaction, represented by a circle, provides communication between human and computer and is implemented by a dialogue developer. A computational function, represented by a square, is a software function that performs only computation and is implemented by an application programmer. Because the notation used in SUPERMAN to represent the design is a graphical programming language, an executable form of an application system's control structure and dialogue can be directly compiled from the supervised flow diagrams. This graphical representation is also interpreted for rapid prototyping.

Functional requirements of human–computer interaction in Casey and Dasarathy [1982] are expressed in terms of a finite state machine (or state transition diagram). Their taxonomy of interfaces is separated

```
diagram irg entry start exit quit

node start
  cs, r2,rv, c_'Interactive Restaurant Guide',sv,
  r6,c5,  'Please make a choice: ',
  r+2,c10,  '1: Add new restaurant to database',
  r+2,c10,  '2: Give review of a restaurant',
  r+2,c10,  '3: Read reviews for a given restaurant',
  r+2,c10,  '4: Help', r+2,c10, '5: Quit', r+3,c5, 'Your
            choice:', mark_A

node help
  cs, r5,c0,  'This program stores and retrieves infor-
              mation on',
  r+1, c0,  'restaurants, with emphasis on San Fran-
              cisco.',
  r+1, c0,  'You can add or update information
              about restaurants',
  r+1, c0,  'already in the database, or obtain infor-
              mation about',
  r+1, c0,  'restaurants, including the reviews of
              others.',
  r+2, c0,  'To continue, type RETURN.'

node error
  r$−1,rv, 'Illegal command.', sv,  'Please type a
    number from 1 to 5.',
  r$,  'Press RETURN to continue.'

node clean
  r$ − 1,cl,r$,cl

node wakeup
  r$,cl,rv,'Please make a choice',sv, tomark_A

node quit
  cs,  'Thank you very much. Please try this program
       again',
  nl,  'and continue to add information on restau-
       rants.'

arc start single_key
  on  '1' to ⟨addnew⟩
  on  '2' to ⟨giverev⟩
  on  '3' to ⟨readrev⟩
  on  '4','?' to help
  on  '5' to quit
  alarm 30 to wakeup
  else to error

arc error
  else to start

arc help
  skip to clean

arc clean
  else to start

arc ⟨addnew⟩
  skip to start

arc ⟨readrev⟩
  skip to start

arc ⟨giverev⟩
  skip to start
```

Figure 12. Some corresponding textual code for Figure 11 (from Wasserman and Shewmake [1985, p. 200]). Reprinted with permission from the authors and Ablex Publishing Corporation.

into classes of stimuli and responses. The addition of checkpoints to validate input and timers for performance measurements extends the model. Application-specific vocabulary and semantics are used to specify system requirements in a Real-Time Requirements Language (RTRL). A comparison of RTRL to an informal English prose version of the requirements specification for a system showed RTRL to produce more complete, consistent specifications.

General Transition Networks (GTNs) also use state transition diagrams as the basis for describing an interface [Kieras and Polson 1983]. GTNs have been proposed as a method both for describing the behavior of an interactive system and for developing a simulation of its end-user interface. Nodes of GTNs represent states, arcs are labeled with both conditions and actions, and examination of the conditions is done in a specified order to trigger transitions. The GTN's key feature, according to Kieras and Polson [1983], is its ability to describe hierarchies of modes or states of the system. The work done with GTNs appears to be oriented mostly toward task analysis. Although Kieras and Polson [1983] claim that GTNs are powerful enough to describe very complex systems easily, the example given in their paper is "a simplified form of portions" of a specific word processor.

State diagrams are also used in SYNICS to represent dialogue and global control [Edmonds 1981]. Many others have used state transition diagrams to represent the human–computer interface [Denert 1977; Dwyer 1981; Green 1981]. The latter uses, however, have been theoretical and apparently have not applied the diagrams to any sizable real-world application.

Interface representation methods that use state transition diagrams should not be considered structural models of human–computer interaction as we described such models in Section 2. Just as a BNF metalanguage does not provide a general, structured organization of the nature of this interaction, neither do the symbols used in state transition diagrams. Like BNF, the diagrams are used to represent specific dialogue instances. They are actually a

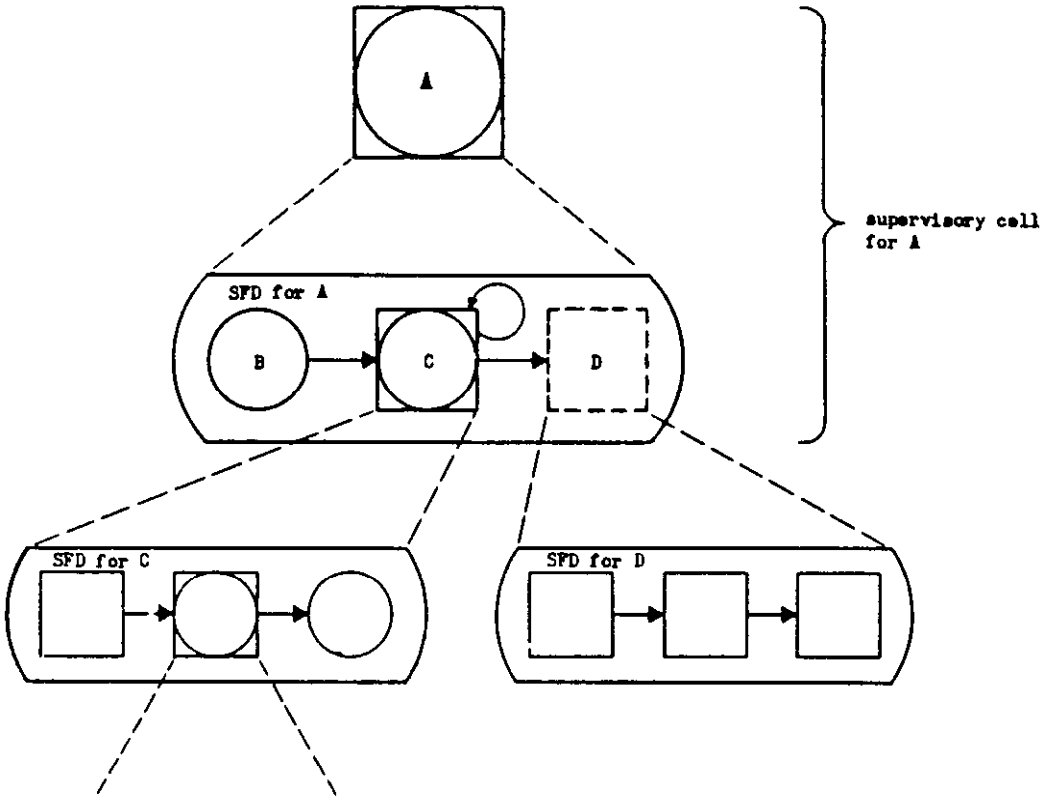


Figure 13. Supervisory structure (from Hartson et al. [1984]).

graphical notation for representing control flow. Without the guidance of a structural model, however, semantic control is often mixed with lexical and syntactic control at the same level of abstraction. This can cause complexity problems that adversely affect dialogue independence.

3.2.3 *Dialogue Language Representation*

Dialogue can also be effectively represented by a high-level "dialogue programming language." Such languages have constructs for representing dialogue-specific features, such as visual attributes, positioning, and devices. For example, a node of a RAPID/USE state transition diagram can represent a screen of an alphanumeric terminal device. The display and end-user input for a specific screen are described in a textual language, one line at a time. To illustrate,

```
r2, rv, c80, c_
  'Interactive Restaurant Guide', sv
```

denotes that at row 2, in reverse video and centered within 80 columns, the screen is to display "Interactive Restaurant Guide" (as a title for a menu) and then reset to standard video.

IBM's Interactive System Productivity Facility (ISPF) relies almost entirely on a dialogue programming language as a means for representing dialogue. Each "panel" (screen) is a procedure written in a textual language designed for representing dialogue. A panel can call and be called by other procedures. A typical panel has a set of actions it performs when the panel is entered, a body that produces the specific dialogue of the screen, and some actions it performs as it leaves the panel. Panels are

invoked by calling a display service (e.g., a SELECT service for a menu panel). The design for functions such as input validation, end-user navigation among fields of a form, and editing of input values is largely predetermined and hard wired.

3.3 Techniques for Representation of Asynchronous Dialogue

3.3.1 Event-Based Representation

Most techniques for representing asynchronous dialogue are variations of event-based mechanisms (see Section 7). Green [1986] surveyed three formal classes of techniques for describing dialogue, based generally on transition networks, grammars, and events. Green concludes that events have the greatest descriptive power, a conclusion consistent with the fact that event-based mechanisms can be used to represent both sequential and asynchronous dialogue.

Design of the human–computer interface using the University of Alberta UIMS [Green 1985] is built around representation through event handlers that are described in an "event language" similar in syntax to the C programming language. The text of a program written in the event language contains at least one event handler definition. At run time, instances of the event handler are created. The skeleton for an event handler declaration is shown in Figure 14. The first part of the declaration indicates input and output tokens the handler can process. The second section consists of declarations of local variables, and the third section contains event declarations, each of which consists of one or more C statements. The visual part of the interface is represented through use of an interactive layout facility built on a window-based graphics package. The University of Alberta UIMS is based generally on the Seeheim model (see Section 2.4). Its event-based interface representation—because it is so similar to C—may not, however, be preferable to simply coding the representation directly in C [Hill 1987]. This would seem to limit its usage to C programmers, a requirement that excludes many dialogue developers.

```
Eventhandler event_handler_name Is
  Token
    token_name event_name;
      .
      .
      .
  Var
    type variable_name = initial_value;
      .
      .
      .
  Event event_name:type |
    statements
  |
      .
      .
      .
  Event event_name:type |
    statements
  |
  end event_handler_name;
```

Figure 14. Skeleton of an event handler declaration (adapted from Green [1985]). Reprinted with the kind permission of Mark Green.

An approach to representing multi-thread, event-based dialogue is based on A Language for Generating Asynchronous Event handlers (ALGAE) [Flecchia and Bergeron 1987]. ALGAE uses an event-based, message-passing mechanism within a multiprocess execution environment for concurrency. An event is represented by a structure that has a type and a value. Interruptible dialogue is accomplished by stacking. The ALGAE run-time environment queuing system handles the message passing. As in the University of Alberta UIMS, event specifications are written in a programming-like language and are used to generate event handlers, which are used to accept each kind of input.

The Event-Response System (ERS) has been used as the basis for representing the syntax of multi-thread dialogues [Hill 1987]. An Event-Response Language (ERL) is based on representation of system responses to events that occur as the result of end-user actions. Complex dialogue is represented as a set of simpler dialogues running in parallel, using a compact, structured representation technique based on production rules. An end-user input event can render a production rule "firable," and searching for firable rules determines state transitions within deterministic finite automata. The ERL has been incorporated

into the Sassafras interface development system.

3.3.2 *Other Approaches*

An unusual paradigm for representation of the interface is embodied in the "by demonstration" mechanism of the Peridot UIMS [Myers 1987]. The dialogue developer represents how input devices are to be handled by showing examples of their use. Sample values for parameters and actions allow Peridot to infer the general input operation and generate the corresponding code automatically. Peridot can be used to represent devices found in direct manipulation interfaces, including mouse and touch tablet.

The User Interface Development Environment (UIDE) [Foley et al. 1988] uses a knowledge base for representing the human–computer interface. Several "schemata" or "frames" are used for the knowledge base, including schemata for objects, actions, parameters, pre-conditions, and post-conditions. Pre-conditions represent predicates that must be true for an action to occur, and post-conditions exist after an action is executed. Transformations can be applied to the representations in the knowledge base to generate alternative interfaces with equivalent functionality.

3.4 Other Techniques for Representation

An approach to interaction language design and representation of interactive computer systems has been introduced in the Command Language Grammar (CLG) [Moran 1981]. Even though the CLG model is a task-oriented model and a structural model, it can also serve as a model for dialogue representation. In fact, the CLG formalism creates a framework for describing many aspects of the end-user interface, not merely representation of the interaction language itself. The CLG partitions an interface into three major components, as shown in Figure 15. Each component is divided into levels, each of which is a refinement of previous levels. At the highest level, the "conceptual component" contains the tasks ("task level") and abstract concepts ("semantic level") from which the system is derived. The "communication component" is composed of the command language ("syntactic level") and the dialogue structure ("interaction level"). At the lowest level, the "physical component" contains the descriptions of the input/output devices and display graphics ("spatial layout level") and all other physical features ("device level").

Conceptual Component:	Task Level
	Semantic Level
Communication Component:	Syntactic Level
	Interaction Level
Physical Component:	(Spatial Layout Level)
	(Device Level)

Figure 15. Levels in the CLG (from Moran [1981, p. 6]). Reprinted with the kind permission of Thomas P. Moran.

The description of each level contains procedures, written in a very high-level programming-language-like notation, that describe all tasks addressed by the system in terms of the actions available at that level, through a process of stepwise refinement. A small message-processing system is given as an example; its description at all levels takes many pages.

Moran [1981] describes how CLG can be considered from three different viewpoints. A linguistic view sees CLG as an analysis of the structure of a system's interface (i.e., a structural model). A psychological view sees CLG as describing the knowledge an end-user has about a system (i.e., a task analysis model). A design view sees CLG as a representation mechanism for system design (i.e., a representation model). CLG thus overlaps all three types of modeling associated with human–computer interfaces. The CLG representation is thorough, providing a representation of an interactive system ranging from the end-user's cognitive level to the system's device level. It appears, however, to be primarily theoretical in nature and not executable. Its major contribution is its thorough addressing of many issues involved in describing an end-user interface at several levels. In Browne et al. [1986], CLG is extended to make the interface model sensitive to context and

end-user characteristics, such as expertise level.

Another proposal for formal representation of human–computer interfaces is use of first-order logic using the rule-based language Prolog [Roach and Nickson 1983]. This method of modeling, designing, and developing dialogues allows a uniform syntactic and semantic description of the interface. Because Prolog translators are available, this representation is also executable and allows for rapid specification, implementation, and modification of an interface. An application example involving representation of a carrier-based air traffic control system took about 100 Prolog rules versus more than 5000 lines of Pascal code. Although the idea of using Prolog as both a representation and an implementation tool is interesting, the complexity of learning to create Prolog programs needs to be investigated; first-order logic is certainly not common knowledge.

In GUIDE [Granor and Badler 1986], dialogue is represented interactively in terms of contexts. A "context" contains interaction tasks, application-generated pictures, application actions, and control decisions. Actions are invoked by use of tasks. Contexts are sequenced by decisions and may be stacked.

3.5 Representation as Part of Interface Evaluation

Interface representation has also been used to facilitate experimentation with human factors in interactive displays [Feldman 1981; Foley 1981]; languages and metrics for interface representation and ergonomic evaluation [Bleser 1981; Reisner 1981] are of particular interest. Because metalanguages (e.g., BNF based) have been used to provide formal representations of at least some aspects of interfaces, research has led to the use of formal grammar description as a prediction mechanism for use in evaluating alternative human–computer interface designs [Bleser and Foley 1982; Reisner 1981, 1982]. For example, Reisner's action grammar is used to describe both cognitive and input actions, which are then converted to a predicted performance time or error representation. Sentences are created that represent particular tasks or end-user classes (e.g., "move cursor" = time to move cursor). Then, a set of "prediction assumptions" is compared to the sentences to determine resultant comparative times. Metrics are applied to the grammar itself to compare alternative interface designs and to find inconsistencies that might cause end-users to make mistakes. Such evaluations using formal language representations allow early identification of design inconsistencies that are likely to lead to end-user errors and allow analysis of the interface for incorporation of human factors principles [Reisner 1983a, 1983b]. Human factors experiments are used to validate this analysis.

Another formal approach is intended specifically to describe the human factors of an interface [Bleser and Foley 1982; Foley 1981]. It defines the lexical and syntactic aspects of both the input and output of an interface. The input definition defines tokens and their relationships, whereas the output definition defines screen characteristics and content. Once the interface is formally represented, evaluative metrics are applied to the representation in order to determine potential design flaws.

Rule-based "expert" dialogue tools [Fischer 1982; Roach et al. 1982] guide dialogue developers in evaluating the application of human factors considerations, graphic design principles, and guidelines for effective communication in the design of their interfaces. In DIADES [Hoffmann 1985] the editor used for producing interface design representations also collects dialogue design decisions. Human factors design guidelines, stored in a knowledge base, are used by a Prolog-based expert system to evaluate quantitatively how well the guidelines are met by a specific design. Also, the Rapid Intelligent Prototyping Language (RIPL), discussed in the Appendix, has both a consultation and an evaluation expert system that use a knowledge base derived from the Smith and Mosier [1986] dialogue design guidelines.

4. INTERACTIVE TOOLS FOR HUMAN–COMPUTER INTERFACE DEVELOPMENT

People working in the field of human–computer interaction seem to be very tool

oriented. Many recent articles in the literature describe tools for development and testing of human–computer interfaces. This is undoubtedly the first wave of an even more comprehensive phenomenon, related to the fact that people who are developing human–computer interfaces for others are not content to have poor quality interfaces for their own work.

Interactive tools for human–computer interface development are often used to automate the process of interface representation. Dialogue developers also use these tools for other activities, including prototyping, evaluation, analysis, and implementation. Outputs from these tools are usually either program code that can be executed to produce the interface or declarative descriptions (e.g., database records) that can be interpreted to produce the interface. Interface development tools often include state diagram editors, graphical editors, text editors, database management systems, and even rapid prototypers. Unfortunately, the tools that are easiest to use are often the most limited in the kinds of interfaces they produce. Some tools were presented in Section 3 as examples to highlight various representation techniques only. This section presents tools in a broader context.

Terminology for interface development tools has not yet stabilized. Drawing on the most common use of the terms in the literature, we shall use the generic term *tool* to refer to anything from a complete interface development environment to a library routine for a single small interface feature. A User Interface Management System (UIMS) is a set of high-level interactive programs for designing, prototyping, executing, evaluating, and maintaining end-user interfaces, all integrated under a single dialogue development interface. Unfortunately, the UIMS appellation has been devalued by casual use, referring to almost any software tool related to human–computer interfaces. The term is now used, for example, to refer to a program that merely helps build screens or that only does interface prototyping or involves interactive graphics. Finally, a *toolkit* is a library of callable program routines to implement lower level interface features (e.g., display an object, accept input) that can be called from within a UIMS or from any other program code.

4.1 Requirements for Interface Development Tools

As experience with tools for developing human–computer interfaces increases, more will be understood about the requirements for such tools. Some desired characteristics for interface development tools include the following:

- *Functionality.* Human–computer interfaces are very complex, consisting of a large variety of features and devices. Tools must therefore be able to produce complex interfaces containing this variety of features and devices. Functionality refers to what a tool can do, that is, what interface styles and techniques it can be used to produce and what input/output devices can be used in the interface a tool produces. The greater the functionality of a tool, the more types of interface features and devices it can be used to incorporate into the application interfaces it produces.
- *Usability.* Interactive tools for developing human–computer interfaces are complex software systems, often in relation to their functionality. Such tools for developing human–computer interfaces have complicated human–computer interfaces themselves. Usability of these tool interfaces is an important issue for productivity and satisfaction of the dialogue developers who use them.
- *Completeness.* A requirement that carries over directly from techniques (see Section 3.1) to tools and related to functionality, completeness is elusive. Consider a small detail such as a field for a "date" value in a MM/DD/YY format. There are lexical rules (e.g., 'DD' must be an integer with a value between 1 and 31), syntactic rules (e.g., governing the order of MM, DD, and YY inputs), and semantic rules (e.g., if MM = 02, then $01 \leq DD \leq 28$, except for leap year). This one small data field of one screen of a whole interface exemplifies the large

number of details that must be represented by a dialogue developer.

- *Extensibility.* Because absolute completeness is unattainable in interface development tools, tools must be extensible. Dialogue development tools are usually more specialized than programming languages. Although this specialization makes a tool easier to use for its specific purpose, it narrows the scope of that tool's applicability. Since the possibilities for human–computer interfaces are unlimited, specific tools cannot address every need. There are at least two ways tools can be made extensible to handle new interface features, interaction styles, and devices: The tools themselves can be easily modified, or the interface representations produced by the tools can be easily modified.
- *Escapability.* In cases in which the tool is inadequate and extension is not feasible (e.g., a rarely used interface feature), it should be possible to escape from the tool and use ordinary programming to produce the interface feature. Use of tools must therefore be compatible with use of programming in the same environment. Without this ability to escape from the tools, the unavoidable limitations of the tools become dead ends for the dialogue developer.
- *Direct manipulation.* Direct manipulation is particularly desirable in the dialogue developer's interface for interface development tools. The dialogue developer works directly with visual (graphical and/or textual) representations of the end-user's task-related objects—"visually programming" [MacDonald 1982]—and results are immediately visible and easily reversible. As indicated by Böcker et al. [1986], the problem-oriented visualization offered by a direct manipulation style tool interface aids in understanding of problems. A taxonomical survey of visual programming, programming by example, and program visualization is given in Myers [1986].
- *Integration.* A set of tools for developing interfaces should have a single integrated interface for accessing all the tools and a uniform interface style across all tools. Further, tools need to have a common output representation to assure composability of tool products within an interface. An underlying database management system is a desirable repository for storing tool output in a common format.
- *Locality of definition.* For consistency in an interface under development, it is desirable for a dialogue developer to be able to give localized definitions that apply to large parts (or all) of an interface. For example, it is useful to represent once, for all menus in the application system, a standard format of the screen title and layout, color, and position of various types of objects on the screen. If an attribute is modified, updating a single representation in a single place can accomplish the change for the entire application system. These capabilities often are found in tools as a design template or "shell," giving default or initial values for various object attributes. Object-oriented implementation environments (see Section 4.5) are excellent for supporting this capability because of their strong inheritance properties within a hierarchical structure of object definitions.
- *Structured guidance.* Without help from the tools in organizing the interface development process, a dialogue developer can be confronted with a confusing mass of detail. Because a structural, descriptive model of the human–computer interface (see Section 2) explains the elements and their relationships in the interface, it is also useful as a framework for interface representation. Other means of developer guidance, such as built-in tutorials, computer-aided instruction, and on-line help, are also desirable. Such materials should include liberal use of examples.

4.2 Application Generators and Other Early Tools

Some of the early tools for human–computer interface development could be classified as application generators and display managers. These tools, although addressing the problems of interface design, lack a

generality in approach. Application generators and display managers are not based on a system development methodology, nor do they involve comprehensive dialogue modeling. In addition, they are typically oriented toward the development of a specific format or interaction technique (e.g., menus or forms). They are also often limited to specific devices and tailored to certain classes of applications.

IBM's Interactive System Productivity Facility (ISPF) [1983, 1985] and Development Management System [1983] are archetypical examples of commercial form-filling and menu-oriented display management tools. ISPF, tailored specifically to the characteristics of the IBM 3270 display terminal and MVS/TSO or VM/CMS operating systems, is used to construct display "panels" (screens) for interactive applications. It provides services to support dialogue in various host environments. When used in conjunction with the Program Development Facility (PDF), ISPF provides an application generator.

Formerly (before October 1983), ISPF and PDF were combined in a predecessor product called the System Productivity Facility (SPF) [Maurer 1983]. Dialogue managed by ISPF consists mainly of selection panels (menus), functions (command procedures such as TSO CLISTs, CMS EXEC files, or programs written in PL/I, ASM, COBOL, or FORTRAN) and data entry panels (form filling). The menu is displayed and the end-user's choice is accepted, causing a particular EXEC file or program to be invoked. ISPF also includes features for menu–tree traversal, split screen management, programmed function key interpretation, and support for producing application help facilities. ISPF panels are analogous to program procedures, able to call or be called by other panels or procedures. Since data communication between the end-user's screen and the program is handled by variables and tables, data structures for these must be declared in both the program and ISPF. Thus, use of ISPF is not for a dialogue developer; dialogue development is still a programming activity. These tools result in at least a superficial separation in that dialogue is contained in separate procedures, but there is no model or methodology to guide the process. Nonetheless, of the more than 12,000 CMS installations, ISPF is the only software product that runs on every one, attesting to the need for such tools. An advanced version of ISPF called E-Z Vu is now available for personal computers.

IBM has a similar product, the Development Management System [1983] (*nee* Display Management System), available for use with VM/CMS and CICS/VS-DOS and especially suited for ADP-type applications. Screen and function management are similar to those of ISPF, including a multiscreen paging ability. Data entry screens provide for record insertion and updating. An application programmer defines files and records, displays, and, to some extent, dialogue sequencing logic. Data communication is handled by programmer-defined data structures, just as it is in ISPF. Such application generators have been shown to reduce both the time to develop applications and the number of errors in the implementation [Canning 1983]. Application skeletons can be stored and adapted for sharing in other applications.

Another IBM system, REXX/FSX [IBM VM/SP 1983], gives limited graphics support to a dialogue developer. FSX provides the graphics support package for textual screens. REXX, a procedural command language similar to the VM/CMS-supported PL/I, cooperates with application routines and the FSX graphics. A dialogue developer must program dialogue as REXX procedures. Nevertheless, IBM reports increasing numbers of systems developed with REXX/FSX end-user interfaces.

A number of other specialized tools for solving specific end-user interface development problems (still, however, lacking modeling and methodology) arose in the late 1970s and early 1980s. As an example, Data General's PRESENT Information Presentation Facility [1982] provides the capability to retrieve data from files and databases and to format that data into reports and graphical displays. Specialized nonprocedural commands are provided for producing pie charts, bar charts, and report formats. Digital Equipment Corporation's Form Management System (FMS) [DEC's

VAX11 FMS 1984] simplifies development of application systems that have form-filling interfaces. A menu-driven interactive form editor allows development of forms by directly manipulating parts of the form.

An Interactive Extension Facility (IEF) [Helander 1981] is a display manager that organizes human–computer interactions into "sessions" to facilitate end-user activities with system objects. Basically, IEF is a set of simple tools for providing "add-on" interfaces to connect the end-user to operating system commands and utilities. Another example of a system that provides for some dialogue design is Screen Rigel [Rowe and Shoens 1983], a set of input/output features for Rigel, a high-level database programming language. Screen Rigel, however, is intended for use by a database application programmer, not a dialogue developer, and the facility is not contained as a part of a broader system design methodology.

A Dialogue Generator (DIAGEN) [Kaiser and Stetina 1982] is a generalized software tool for creating an interactive interface that separates the dialogue from the dialogue-driven application program. A specialized DIAGEN language is used by a dialogue developer to "program" a scenario that describes the dialogue; this scenario is then interpreted. A single run-time message for erroneous end-user input was hard wired; the system could only respond "Wrong answer. DIAGEN repeats the question," the whole sequence is repeated. Several of the above, and other, commercially available software tools for interface management are compared in a survey [Britts 1987].

4.3 User Interface Management Systems

4.3.1 Historical Perspective

The term "user interface management system" (UIMS) appears to have first been used by Kasik [1982], although the idea existed earlier. Most early literature on UIMS was not concerned with the end-user, human factors of interfaces, or methodologies for software or interface development. In particular, early UIMS work did not emphasize human–computer interface development activity. Rather, its focus was on support—especially through graphics software—for execution of the interface [GIIT 1983; Guedj et al. 1980]. Early UIMS-produced interfaces were specified by special languages or grammars (e.g., BNF-style definitions) or, more simply, by directly coding the interface in a programming language rather than by interactive Thus, early UIMS tools were strictly tools for programmers.

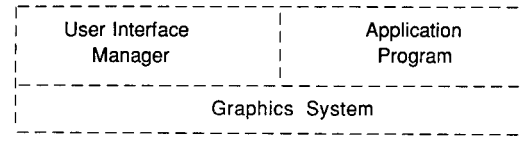


Figure 16. User interface management system model (from Graphical Input Interaction Technique [1983, p. 16]). Reprinted with permission from James J. Thomas.

The summary report of the Graphical Input Interaction Technique Workshop [1983] illustrates this emphasis on execution by stating that "the role of a UIMS is to mediate the interaction between an end-user and an application. . . ." Also illustrating the emphasis on graphics, it says that the "model underlying most of the presentations at both this workshop and Seillac II [Guedj et al. 1980] [is] a ternary division into an application program, a user interface management system (UIMS), and a graphics system." This model is shown in Figure 16. In this model, connection to the end-user is not shown. For interface implementation, this model calls for a "UIMS manager" to accept and store interface representations. Other diagrams of UIMS also sometimes emphasize run-time aspects, as in, for example, Figure 1 of Hayes et al. [1985].

A "reference model" [Lantz et al. 1987] for the implementation of interactive software, shown in Figure 17, is a model that is more developed than that of Figure 16. It includes consideration of concurrent tasks, distribution, and multiple media. The emphasis of this reference model is still, however, execution and not design, and separation suffers because all layers of the model appear to have access to devices.

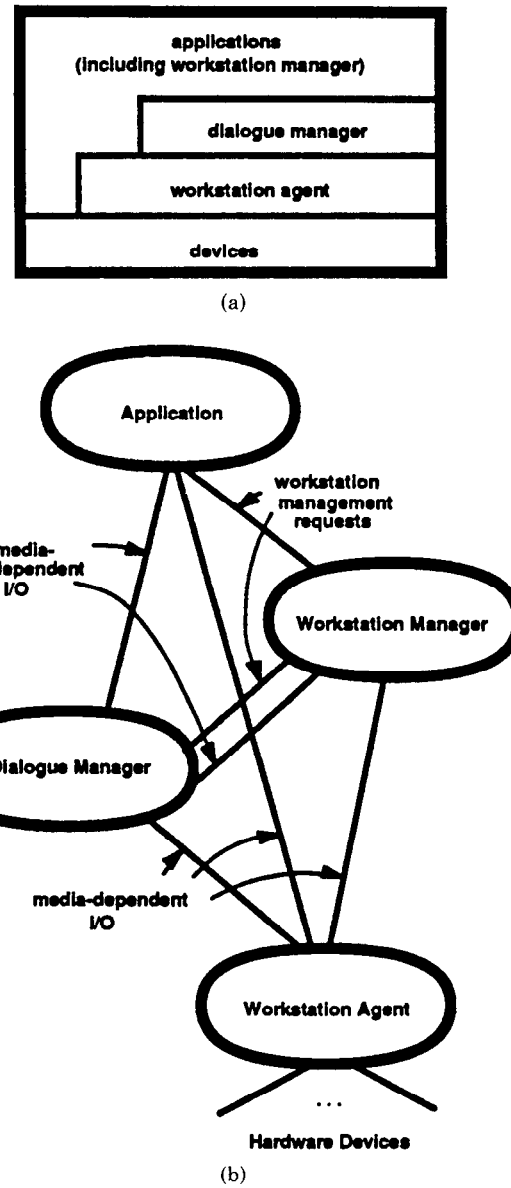


Figure 17. Reference model: Two views: (a) layered view; (b) modular view, with interfaces (from Lantz et al. [1987, p. 88]). Reprinted with permission from the authors and publishers.

The model does represent all possible information flows for all possible applications, not just the flow for one application. Both this reference model and the GIIT model could also be considered architectural models of the application system (see Section 2.4).

Later, the UIMS view began to broaden [e.g., Olsen et al. 1984; Tanner and Buxton 1984] to include considerations of the end-user and the dialogue developer. Although much of the recent UIMS literature still refers to programming of the interface—and it is unlikely that interfaces can ever

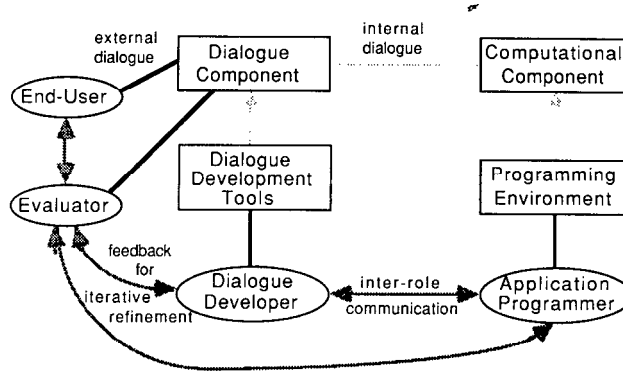


Figure 18. Typical basic structure of a UIMS.

be constructed totally without programming—there is much emphasis on a nonprogramming role and design-time tools for the dialogue developer.

The typical basic structure of a UIMS is illustrated in Figure 18, along with the appropriate roles involved. A dialogue developer interacts with automated tools for developing the application system's human–computer interface; these tools produce an internal stored representation of the dialogue that is executed at run time to produce the interface. An application programmer produces the application system's computational software, providing its functionality. These two developer roles communicate and coordinate their development efforts. End-users and system evaluators give feedback about the interface and system functionality. The entire process shown in Figure 18 forms a cycle of iterative refinement. The remainder of this section describes, generally in chronological order, representative examples of systems to show the diversity of UIMS-like tools. Where appropriate, the description highlights unique or unusual features of the tools. Several of these tools are presented in more detail in the Appendix.

4.3.2 Examples of UIMS

The toolkit UIMS (formerly TIGER) [Kasik 1982], discussed in the Appendix, uses a "dialogue programming language," which extends Pascal specification and declarative structures while retaining Pascal's control structures. The MENULAY [Buxton et al. 1983] UIMS design and implementation tool is a high-level language preprocessor that translates dialogue (graphics and menus) design code written by an application programmer into C language programs that, when executed, produce the graphics and menus of the interface.

In the Abstract Interaction Handler, Foley [1981] and Feldman and Rogers [1982] advocate separation of the end-user front end from the system semantics, even to the point of being able to customize interfaces for individual end-users [Feldman 1981]. The independence of their dialogue from system semantics allows experimental evaluation of various interfaces while holding the underlying computational system constant. A successor to AIH, the GWUIMS, is presented in the Appendix.

The Dialogue Management System [Ehrich and Hartson 1981; Hartson et al. 1984; Hix and Hartson 1986] is a research UIMS that has been developed as a test bed for interface management concepts. It contains an integrated set of interface development tools called the Author's Interactive Dialogue Environment, or AIDE, in early versions of DMS. These tools embody a structural model, methodology, representational notation, life cycle management, and rapid prototyping. Tools include a display tool, several menu tools, a forms tool, and primitives libraries. In addition it contains several generic tools for

developing interfaces not supported by specific tools. DMS itself has a direct manipulation interface. The DMS approach to interface development considers human–computer interface management as an integral part of software engineering and is detailed in the Appendix.

The SYNGRAPH (SYNtax directed GRAPHics) [Olsen and Dempsey 1983] and MIKE (Menu Interaction Kontrol Environment) [Olsen 1984a] UIMS use textual languages for dialogue representation and code generation for dialogue implementation. Input to SYNGRAPH is a BNF grammar that a programmer (not a dialogue developer) uses to describe the command language in terms of menu items, function buttons, valuators, and a single locator device. From this description, a segment of Pascal code is generated and then compiled—along with some standard interface code and the application's semantic code—to create the final interactive program.

A key issue tested by SYNGRAPH is the automatic allocation of screen space based on the interface description grammar. The programmer can divide the dialogue into levels or modes, each of which is characterized by a menu or set of enabled devices. The software then analyzes each level, determines what interactive resources are needed, and allocates them appropriately. This approach was not very satisfying, however, because of its rigidity and the indirectness of defining graphics with a textual language. In SYNGRAPH 2 (now called GRINS) [Olsen et al. 1985], end-users seem to prefer a layout editor provided for performing the display layout functions. Another SYNGRAPH emphasis is error recovery. In the interface description grammar, every nonterminal item definition can have a "cancel" production that is entered whenever the cancel button is selected. This allows the semantics programmed in the computational code to perform any recovery needed.

MIKE [Olsen 1984a] was created in response to the large amount of effort required to teach programmers how to use SYNGRAPH. MIKE is based on command procedures that define the set of interactive commands. Initially the programmer gives MIKE a list of procedures (functions) and the types and names of their parameters. An initial interface simply displays the names of all procedures in a menu and allows the end-user to select a procedure by typing any unique abbreviation of the procedure's name. The end-user is then prompted for the first parameter and is given a menu of all functions that return that parameter's type as their result. This process continues until a complete expression has been input, at which point the appropriate procedures are called to execute the semantics. This approach is simple to teach to programmers but is not very graphical. MIKE has a profile editor that allows the interface to be interactively tailored into a more acceptable form. The profile editor can edit the names of commands to more end-user acceptable terms; map commands to function buttons, graphical icons, or textual prompts; and organize commands into menu trees for a more suitable structure.

State transition diagrams provide an ideal graphical language to be supported by direct manipulation dialogue development tools. An example is seen in the Transition Diagram Editor (TDE) [Mills and Wasserman 1984] of RAPID/USE (previously discussed in Section 3). The TDE is a graphical editor for state transition diagrams, based on menu selection using a mouse and keyboard. Because the TDE knows about the connectivity of nodes and arcs being created, it can automatically do some of the formatting. This allows the system designer to concentrate on diagram semantics. To create a node, the designer points with the mouse to the desired screen position. To create an arc, the designer points to the nodes to be connected, and the TDE draws the lines and arrowheads. Contents of the nodes, which contain, for example, the dialogue language code for displaying a menu, are created using a text editor. The TDE directly generates USE transition diagram language descriptions for the RAPID/USE Transition Diagram Interpreter (TDI). Other systems use similar approaches to interactive representation of the interface; for example, Jacob's

[1985] work has produced a state diagram specification interpreter. Both RAPID/USE and the state diagram specification interpreter are presented in the Appendix.

Apollo Computer's Domain/Dialogue (D/D; formerly known as A Dialogue Manager) [Schulert et al. 1985] is again typical of the UIMS approach in which emphasis is on mechanisms for handling execution-time aspects of interfaces. D/D dialogue is programmed using a compiler and a run-time dialogue library. The dialogue is defined around a set of interaction techniques, which form the basis for the interface to the computational code. A set of interaction techniques is assembled to define the end-user interface. Given a particular task set, a variety of end-user interfaces can be developed to carry out run-time interactions, including menus, pointing, forms, and function keys. D/D is a commercially available product running on an Apollo workstation, using bit mapped graphics. Open Dialogue, the successor to D/D, is described in the Appendix.

Unicad [1985] has developed a UIMS to ease the task of implementing interactive computer-aided design (CAD) systems. A CAD environment requires interactive graphics support; the Unicad system is built on a graphics package that provides support for graphical interaction techniques, particularly at the lexical level. The Unicad system is, however, for use by application programmers, not dialogue developers.

The Trillium UIMS provides an example of composability of interface objects; composite objects are hierarchical compositions of dialogue primitives, defined by LISP code [Henderson 1986]. A library of primitives and composites can be shared as building blocks for interface development.

The Graphical User Interface Development Environment (GUIDE) [Granor and Badler 1986] is an interactive graphical system for designing and generating graphical end-user interfaces. It provides flexibility to the system designer while minimizing the amount of code the designer must write. The primary goal of GUIDE is to provide a simple, interactive way for a dialogue developer to specify an application interface. Style of the interface should be determined by the developer, and the developer should be able to describe with GUIDE any interface that could be coded by hand.

GUIDE provides a great deal of freedom in representation of the control path and parameters to action routines. The developer may refer to application constants, types, variables, and functions in defining the interface. This ability greatly reduces the number of states needed to define the interface. Actions are provided to perform application functions, and may have parameters based on inputs and application values. Multiple control paths may be represented by the dialogue developer based on inputs, application values, and end-user characteristics. Inclusion of a developer-defined end-user profile allows the developer to represent different interfaces within a single system for different end-users. Various interaction styles and devices can be used, including menus, forms, picking, and keyboard. The developer may choose among any that are suited to a task and may, in fact, allow the end-user to choose among several styles or devices to provide a particular input.

Enter/Act is a product from Precision Visuals [1987] with a set of high-level tools to handle all aspects of the human–computer interface, particularly those developed on Digital Equipment's VAX hardware. Enter/Act emphasizes prototyping and extensive graphics based on DI-3000 graphics software. It includes various practical aids for enhancing developer productivity, such as debugging mechanisms and command macros.

The SmethersBarnes Prototyper [Prototyper 1987] is a commercially available tool that, despite its name, is more a UIMS than a prototyper. It can be used to develop Macintosh-style interfaces, including windows, pull-down menus, radio buttons, and check boxes. Its interface generally uses direct manipulation to produce the application interface. Application semantics can be coded in one of several programming languages and linked to the interface for execution. Prototyper is detailed in the Appendix.

4.4 Toolkits and Related Graphics Support

Several systems for interface management are built on top of graphics packages such as the Core system [*IEEE Computer Graphics* 1979] or the Graphics Kernel System (GKS) [*IEEE Computer Graphics* 1984], but "standard" graphics packages rarely provide enough functionality for most state-of-the-art interface needs.

A command interpreter (kommandointerpreter—KI) has been developed as the basis for one such support environment [Borufka and Pfaff 1981]. The KI commands form the linkage between graphical input and output data and provide the dialogue developer with functions for prompting, echoing, and editing dialogues. At the heart of the system is GKS, which provides a set of functions for graphical data processing independent of specific graphical input/output devices, programming languages, and application systems. It includes two-dimensional input/output primitives and a segment facility for subdividing graphical pictures. Graphical output can be routed to multiple workstations. On top of GKS are the KI kernel commands, which serve two purposes: to allow interactive use of the GKS functions by providing the end-user with a common set of commands and to provide system functions for echoes, error messages, help screens, and editing. Extensions to the KI include end-user-defined command sequences and menus and commands defined by end-users and dialogue developers.

The ACM/SIGGRAPH/GSPC Core System is another standardized graphics package that is being used to support interface management systems. This Core standard provides a high degree of device independence. The Abstract Interaction Handler (AIH) [Kamran and Feldman 1983] has been built on top of the Core system, with emphasis on various interaction techniques and styles. The uncoupling of interaction-supporting code from application-supporting code is a major feature. One-level Core "segments" are the basis for a screen handler. This package of routines creates a higher level structure of these segments in order to handle logical screens, to which all output is written. Binding of interactions to logical windows is accomplished by the screen handler, whereas binding of interactions to specific devices is handled by the basic Core system. The implementation of Functional Language Articulated Interactive Resources (FLAIR) [Wong and Reid 1982] (see the Appendix) required a significant extension of the Core standard.

GKS is a standard for static graphical images. The Programmers Hierarchical Interactive Graphics Standard (PHIGS) [Brown 1985] has improved capabilities for dynamic interaction. PHIGS, however, inherited many of the drawbacks of the GKS input model [Meads 1987; Puk 1986; van Dam et al. 1987]. The number of input device classes is limited, and there is no window management. As a result, PHIGS cannot support many of the new input techniques.

The graphics systems described above are general software packages with emphasis on graphics power; interface considerations are secondary. Their drawbacks led to a new class of graphics support toolkits, oriented more toward problems of interface development. Many of these systems are based on window managers. Most window manager ideas come from Xerox PARC systems such as Smalltalk and Star. Window managers allow systems to be designed so that the end-user can interact with several tasks, each in a different "viewport." Typically, only one window at a time can be actively awaiting input, attached to the keyboard and mouse. Many window managers, however, allow multiple windows to receive output at the same time. Each window acts as a separate logical terminal device with its own input and output services. End-users can manipulate windows, and communication among windows is typically cut and paste using a clipboard concept. Most window managers are toolkits in the sense that they contain libraries of window functions—and possibly other interface features—that programmers can invoke.

The X Window System [Scheifler and Gettys 1986], developed at the Massachusetts Institute of Technology, supports an arbitrarily branching hierarchy of resizable,

overlapping windows upon which human–computer interfaces can be built. A base window system provides high-performance, high-level, device independent graphics to this hierarchy. This base window system provides facilities to build applications and managers for input and windows. X provides "widgets," which are primitives from which various interface styles can be constructed. This feature makes it attractive as a support environment for developing human–computer interfaces. In fact, X is now one of the most widely used support environments for both research and commercial products. Its libraries and tools for facilitating interface development are rapidly expanding.

Display PostScript [Perry 1988], by Adobe Systems, is a graphics support toolkit that uses a UNIX "troff"-like language for describing display pictures and inputs. It evolved from the PostScript language for describing Apple Laserwriter output. Despite X Window's popularity, Display PostScript is generally considered technically superior. Sun Microsystems is building their Network Extendible Window System (NeWS), previously called SunDew, using Display PostScript. PostScript programs are downloaded to the NeWS window manager to improve performance. One of the most common window managers in the IBM PC world is Microsoft's Windows [Puglia et al. 1986].

4.5 Other Support

4.5.1 Database Management

Outputs of all tools within a UIMS, including interface definitions, documentation, and even program code, must be stored and retrieved. Massive amounts of secondary storage are required to store representations of interfaces and the objects they contain—screen descriptions, graphics, text, sequencing relationships, input validation criteria, and so on (see Section 3). Most interface definitions are retrieved and executed or interpreted at prototyping time and run time.

Early UIMS and other dialogue tools used file systems provided by host operating systems. This is convenient for experimenting with prototype tools but lacks the power and flexibility needed for real applications. High-performance database system support is required. Descriptions of individual interface objects are stored separately for sharing and reusability. It is a significant performance challenge to a database management system to bring them back together through relational joins, for example, which are computationally complex operations.

Some researchers have found that most commercially available systems either are too expensive or do not provide the flexibility and performance necessary for this demanding application and have devoted considerable effort to developing their own supporting database management system. Two examples of interface management systems that use their own internally developed relational database systems as support are RAPID/USE and DMS. It is interesting to note that both these database systems, nontrivial development efforts in themselves, were entirely produced using the respective software development methodologies. Current object-oriented programming systems provide some of their own internal capability to store and retrieve large numbers of object definitions. The state of this technology is, however, still limited in the size of application system designs that can be stored.

4.5.2 Object Orientation

Human–computer interface development tools need the ability to support rapid design changes without recompiling or relinking, which can take substantial amounts of time for large application systems. Because of their interpretive nature and dynamic binding capabilities, languages such as LISP and its variations are popular for implementing interface development tools. Object-oriented programming environments have also attracted the attention of tool implementers [Fischer 1987; Sibert et al. 1988], with languages such as FLAVORS and LOOPS combining LISP with object orientation. An object-oriented programming environment, such as Smalltalk [Cox 1986; Goldberg and Robson 1983], offers the advantage, for tool implementa-

tion over conventional programming, of a capability for hiding enough information to represent objects independently of their implementation. Because of its event-based nature, object orientation is effective for representing asynchronous dialogue and for representing the behavior of specific interface features (e.g., windows) regardless of their context. The capability for dynamic binding, hierarchical definition with inheritance of attributes and procedures, and communication by message passing work together to support sharability, reusability, consistency, flexibility, and low code bulk within interface implementations.

A disadvantage of object orientation is the tendency to obscure temporal relationships in the high-level sequencing behavior in the application interface. Because of this limitation, object orientation has yet to be proven useful for representing the view of the dialogue developer or the end-user; it is possibly better suited for the tool developer than the tool user. Other disadvantages include a steep learning curve for programming and a high performance penalty due to interpreted code, dynamic binding, and message passing. The current version of the Dialogue Management System is implemented using Smalltalk; the architecture of the George Washington University UIMS is broadly object oriented and is discussed in detail in the Appendix. Object orientation is also suitable for supporting prototyping tools [Diederich and Milton 1987]. The class concept allows easy implementation of variations of tools, for example, by defining tool P to be the same as tool Q except for certain features.

4.5.3 Workstations

Beyond graphics standards are technological advances that have produced powerful graphics workstations useful for supporting interface management. These workstations are typically stand-alone micro- or minicomputers with their own substantial operating systems and most graphics functions implemented in high-performance hardware. Several interface development tools have been implemented on graphics workstations—for example, RAPID/USE on a SUN, RIPL on a MicroVAX, Domain/Dialogue and Open Dialogue on an Apollo, DMS on a Silicon Graphics IRIS, and a state transition diagram interpreter on a Symbolics machine.

5. RAPID PROTOTYPING

Present methods for developing and evaluating human–computer interfaces are more analytic than synthetic in nature. That is, something must first be built, then analyzed, then iteratively refined. The present state of human factors does not allow synthesizing a human–computer interface and "getting it right" the first time, and this is unlikely to change soon. In comparison with software design, which is often correctness driven, interface design must be a self-correcting process. As Carroll and Rosson [1985] point out, design activity is essentially empirical "not because we don't know enough yet, but because in a design domain we can never know enough." The process of iterative refinement involves two important roles: One is related to computer science, the other to behavioral science. The latter role is responsible for dialogue principles and human factors, which are not in the scope of this survey. The computer science role, however, is to provide human factorability of interfaces, which is at the heart of this article.

5.1 Motivation for Rapid Prototyping

Building systems is expensive and time consuming. The alternative is to build prototypes rather than complete systems. Human factorability calls for dialogue development tools that will rapidly produce prototypes to allow early observation of interface behavior and that will allow easy modification of designs. Thus, *rapid prototyping* is a major concept of human–computer interface management. Rapid prototyping of interfaces is also sometimes called dialogue simulation, and prototypes are sometimes called scenarios or executable specifications. Prototypes can also be written just as programs.

Prototyping is an approach to system development that involves production of at

least one early version of the application system, demonstrating essential features of the later operational system. With rapid prototyping the process is accelerated so that many alternatives can be evaluated and the effects of each modification can be promptly observed. Rapid prototyping brings together both interface representation and execution, often under the aegis of a UIMS. Rapid prototyping is, though, primarily a technique, not a tool. Valuable insight can be derived from use of paper and pencil interface prototypes early in the interface development process. Key ingredients of a rapid prototyping approach include an early ability to observe end-user and system behavior, use of scenarios, end-user participation, and an evaluation orientation to development. The iterative nature of human–computer interface development imposes changes in the traditional linear development life cycle [Hartson and Hix 1989]. A prototype reduces the chances of surprises to the end-user, helps solve the problem of the end-user's inability to give complete specifications to system designers, and "gives the end-user a more immediate sense of the proposed system" [Wasserman and Shewmake 1982]. It reveals misunderstandings that arise between developers and end-users because of their different backgrounds and experience [Gomaa and Scott 1981].

Whereas testing, verification, and validation are intended to indicate whether a design meets a requirements specification, prototyping can show up errors in the requirements. These errors in requirements are difficult to detect and even more difficult to correct [Boehm et al. 1984]. The goal is fast communication of interface design alternatives to developers, end-users, and implementers. Rapid prototyping allows the process of iterative refinement to occur earlier in the design process. For more than 15 years, the literature has called for involving the end-user in system design; rapid prototyping provides a way, for the first time, to do this effectively and efficiently. The emphasis on this approach to system design is evidenced by the appearance of several surveys and workshops [Carey and Mason 1983; Freeman 1980; Zelkowitz 1982] (Cochran, private communication, 1984).

At first, especially among developers, there was some question as to whether a working prototype was necessary. Why couldn't anyone follow the requirements and design documentation, especially the procedural parts, and see for themselves what the target application system was like? Wasserman and Shewmake [1985] respond very well: "While *some* customers are willing to buy *some* cars simply from a brochure containing technical specifications and photographs, most customers prefer the opportunity to take a test drive, *even if the car that they test is not identical to the one that they will purchase.*"

Alavi [1984] compared prototyping with the traditional life cycle approach to software development on twelve information projects in six different organizations. The study concluded that prototyping, especially in the face of unclear or ambiguous end-user requirements or where there is a need for experimentation (which is true, of course, for most systems with human–computer interfaces), was effective as an approach to interactive system development. In particular, results showed that end-users of systems developed using a prototype were more favorable toward the final system than were end-users of nonprototyped systems. Prototyping also facilitated communication between end-user and developer, but it did cause some difficulty with managing the design process.

In another multiproject experiment involving seven software teams, Boehm et al. [1984] reported that prototyping, compared to complete a priori specification as a development approach, produced software with equivalent performance but with about 40 percent less code and 45 percent less effort. Although the prototyped software rated lower on functionality and robustness, it was judged easier to learn and use. These conclusions indicate that human–computer interface concerns are supported by prototyping as a system development approach. The areas that suffered represented software concerns more

than dialogue concerns, a fact that indicates the need for a software development methodology that integrates prototyping, especially interface prototyping, as part of the whole system development life cycle. The next section, on Methodologies for Interactive System Development, discusses methodological issues that address this problem. Hartson and Smith [1988] give more discussion of advantages and disadvantages of prototyping in the development process.

The idea behind interface prototyping is not new; it can be thought of as an extension of software simulation, with emphasis on the human–computer interface. The experimental work on a help facility in the Interactive Chart Facility [Clark 1981] is an example of an early approach making this transition from software to interface simulation. Even before software simulation, the same ideas were used (for the same basic reasons) in the early prototyping of hardware logic designs [Hartson 1969; Hays 1969; Linebarger and Brennan 1964].

5.2 Kinds of Prototypes

Approaches to prototyping can be classified along at least three (more or less orthogonal) dimensions: revolutionary versus evolutionary, interface only versus whole system, and intermittent versus continuous [Hartson and Smith 1988]. A "revolutionary" development process is one in which a prototype is designed, built, evaluated, and scrapped before work begins anew on the real system. A revolutionary prototype is most useful when built as early as possible, without a large commitment of resources. In an "evolutionary" development process, a prototype evolves through iterative modification into a complete implementation of the target application system. The evolutionary approach avoids wasted effort and the difficult question of when to discard the prototype and start working on the real system.

"Interface only" prototypes are very common; a mock-up facade is fairly easy to construct and execute. Dan Bricklin's Demo Program [1987] and Skylights [1987] are among the increasing number of products currently available that use a "slide-show" concept to build and view sequences of screens (scenarios), including automatic pacing and end-user-directed branching. Such tools, however, rarely have a dialogue model or predesigned dialogue constructs (e.g., menus or forms) and often accept only alphanumeric keyboard input. In some cases these products have been augmented with ways to connect calls to semantic (computational) routines, and some have added code generators. FLAIR [Wong and Reid 1982] and GIDS [Overmyer and Campbell 1984] are examples of prototyping systems that build detailed, complex graphical mockups yet are still interface only. "Whole system" prototypes, however, have advantages. As computational functions are developed, it is desirable to see them in action in the prototype. A disadvantage is that whole system prototypes are difficult to build; their execution environment requires much more technically complicated support.

Prototypes for which the ability to demonstrate system behavior is "intermittent" can be exercised only at times in the development process when a particular version of the system has been completely constructed. A coded implementation of a prototype (slow prototyping) is an intermittent type. There are long intervals between complete versions where the code cannot be compiled and run. This approach is not responsive to the needs of iterative development. Prototypes that can be exercised on a more or less "continuous" basis are more desirable for interface development and do not depend on complete development of a specific version of the system. Prototyping of incomplete designs, however, poses challenging problems in the support environment, primarily because software is fragile. The slightest error or missing piece can prevent it from running. Even stubbed systems must be syntactically complete and correct. The nature of prototypes, especially early ones, is to be incomplete, ambiguous, tentative, and error prone. The support environment must keep a prototype running despite these initial defects.

5.3 Approaches to Rapid Prototyping

Because rapid prototyping usually requires interpretation of interface representations, approaches to rapid prototyping are related to the corresponding approaches to interface representation, discussed earlier in Section 3. In particular, there are prototyping approaches for interfaces represented by state transition diagrams, BNF-style grammars, and event-based mechanisms. Most of the sequential prototypers (i.e., those based on BNF or state diagrams) are similar conceptually, the differences being mostly in ease of use of the representation scheme. Since time sequencing is an important aspect of sequential dialogue prototyping, the state transition diagram representation (which graphically shows sequential relationships) may be preferred by nonprogrammers. For event-based dialogue, HyperCard (discussed at the end of this section) represents another kind of approach.

5.3.1 State Transition Diagram-Based Prototyping

Both Jacob and Wasserman have developed similar rapid prototyping systems, which, like their interface representation schemes, are based on state transition diagrams. Both systems are discussed in the Appendix. Wasserman and Shewmake's [1982, 1985] RApid Prototypes of Interactive Dialogues (RAPID/USE) is a part of their broader User Software Engineering (USE) methodology to support construction of prototypes and interactive information systems. For prototyping, local storage variables are added to store state information and to communicate with the semantic part. Since semantic actions are associated with each arc, an entire interactive information system could be implemented using the RAPID/USE interpreted approach to state transition diagrams, by invoking programmed semantic routines when needed during the dialogue sequence. By adding this functionality to the prototyping process, RAPID/USE can provide realistic dialogues more closely associated with the appropriate semantic action instead of fixed, predetermined stub messages.

A Transition Diagram Interpreter (TDI) executes the coded representations and simulates the interface. Because the TDI interprets the coded representations, recompiling is not necessary when changes are made. During the use of RAPID, logs are maintained for metering raw inputs, allowing analysis of keystroke-level events and playback of scenarios. Time-stamped transition-level events are also logged for analysis. A commercially available version of RAPID/USE is now marketed by Interactive Development Environments, Inc.

Jacob's [1983] use of state transition diagrams is similar. As in RAPID/USE, state transition diagrams are converted to a corresponding text form (a state transition textual language) much like a high-level programming language. This textual form is executable, providing rapid prototyping of the interface.

A third system based on a representation technique similar to state diagrams is the Dialogue Management System (DMS) [Hartson et al. 1984] in which rapid prototyping is done by a subsystem called the Behavioral Demonstrator. All parts of the dialogue are produced using the Author's Interactive Dialogue Environment (described in Section 4.3), with the declarative representations stored in a database and interpreted for prototyping. Graphical representation of the global control structure—which in DMS is separate from the dialogue but controls sequencing between dialogue and computation—is directly interpreted, without an intermediate textual language. The two representations, together with their respective tools, allow refinement of logical sequencing, dialogue form and content, and interaction styles during prototyping. The Behavioral Demonstrator demonstrates those parts of the system that are implemented and uses dialogue developer-provided samples of values for data of those parts that are not yet implemented. The Behavioral Demonstrator provides a life support system for partially completed designs, which is able to execute as much of the dialogue (and whole system) as is completed at a given time. As they become available, actual application dialogue and computational functions become part of what is demonstrated. The

final system gracefully evolves without the added effort of throwaway prototype code. An experimental version of the Behavioral Demonstrator has had limited, but successful, use.

Mason and Carey [1983] have analyzed ACT/1, once a commercially available product for rapid prototyping of end-user interface scenarios. Sequential dialogue in this screen-oriented system is created "by example" through filling in parts of a screen. The logical sequencing mechanism is basically the same as in state transition diagrams, but the representation is in tabular form, showing procedural links. Control flow is represented by textually noting the relationship between end-user inputs and successor routines and screens. ACT/1 uses an "architecture-based" methodology, with which the system designer, much like a building architect, develops the external appearance of the system and works inward to develop the system design. An application is seen as a series of ⟨input screen, process, output screen⟩ sequences, and linkages or control among these screens. Screen scenarios are used as the communication mechanism between the application end-user and the system developer. The end-user can follow a fixed script through screens without any application logic having been developed. Logic flow definition is by example through the screens, indicating the successor screen for each possible end-user input.

Suggestions made by the end-user while exercising the ACT/1 scenarios are incorporated during successive iterative refinement processes. A demonstration phase with partially implemented application logic evolves finally into a first prototype of the target application system. Dialogue screens developed in the specification stage are directly usable in this production version. ACT/1 has had more than 100 end-users and has been widely applied and evaluated in the development of interactive information systems [Mason and Carey 1981].

TRW's FLAIR system [Wong and Reid 1982] is similar to ACT/1 in that it directs a dialogue developer through sequences of menu screens to translate a scenario into a form that can be executed or simulated. FLAIR was one of the first to use a language, called a Dialogue Design Language (DDL), as a dialogue representation instead of a formal language grammar. FLAIR's voice menu-driven DDL allows system designers to construct highly graphical human–computer interfaces and allows the end-user to interact with a prototype of the final system through scenario simulation, facilitating experimentation with various interfaces. FLAIR can create, store, and retrieve static frames, as well as allow the end-user to define and control a hierarchy of menus. FLAIR and another rapid prototyper called the Rapid Intelligent Prototyping Language (RIPL) are presented in the Appendix.

5.3.2 *BNF-Based Prototyping*

Very similar to the state transition diagram-based approaches for prototyping are those based on BNF-like grammatical interface representations, although there are fewer of these. The grammars are interpreted with mechanisms that are basically finite state machines. The different type of representation carries with it a slight difference in emphasis, treating the interface in a more language-oriented way with a less direct emphasis on control structure and sequencing.

The Interactive Dialogue Synthesizer (IDS) developed by Hanau and Lenorovitz [Hanau and Lenorovitz 1980a, 1980b; Lenorovitz and Ramsey 1977] is an example of a set of tools to create simulations of end-users' interactive dialogues for which dialogue is defined using a BNF grammar. Displays are defined as machine-independent semantic actions, the meanings of which are defined in terms of an abstract machine, attached to rules of the grammar. Language processors are automatically generated from the grammar and used to execute interface representations as a simulation. Initial conceptual "snapshots" of scenarios are predrawn and, along with dynamically constructed displays, are used to simulate the external appearance of the desired application system. Real-time display updates are simulated with timed sequences of static displays. IDS has been successfully applied to a number of diverse

real-world application areas, mostly command and control systems.

5.3.3 Event-Based Prototyping

An example of prototyping using an event-based mechanism, more suitable for prototyping asynchronous, multi-thread dialogue, is found in the use of Apple Macintosh's HyperCard [Goodman 1987] as a prototyper. In HyperCard, a "card" is a screen of text and graphics objects, and cards are grouped into "stacks." A dialogue developer can define "hot spots" associated with objects on the screen, making such objects selectable in response to end-user actions such as mouse button clicking. The dialogue developer also defines a response to each event; for example, a mouse click action on an arrow object can be made to cause execution to move ahead to the next card in the current stack. Using HyperCard, the dialogue developer can create a template-like background for a particular style of card to be used throughout a stack. To this background are then added foreground objects specific to each individual card. For example, a name, address, and phone number background card could be created and then numerous individual cards with specific instances of name, address, and phone number overlaid on the background. A library of icons (symbols), many even with semantics (response to end-user actions) attached, greatly facilitates interface development. Use of the HyperTalk programming language to create program "scripts" allows linkage among cards and serves as a general means for providing semantics where necessary. HyperCard has been called programming for nonprogrammers, and, as such, is a suitable system for use by dialogue developers.

6. METHODOLOGIES FOR INTERACTIVE SYSTEM DEVELOPMENT

A methodology for system development consists of a set of *procedures* that indicate a step-by-step development process over a life cycle and a *notational scheme* that is the means for documenting designs that evolve during that life cycle. We discussed notational representation schemes for the human–computer interface in Section 3; in this section we will focus on the procedural, life cycle aspects and, in particular, on connections of human–computer interface development to the development process for the rest of a target application system. Although technical matters abound, the subject of development life cycles is also a management issue [Mantei 1986].

The need to view human–computer interface management as an integral part of the software engineering process is being recognized [Draper and Norman 1985; Hartson and Hix 1989; Hartson et al. 1984]. Interfaces cannot be developed as "add-on" parts of an interactive system, with their development carried out in isolation from development of the rest of the application system. Thus, an important concept in human–computer interface management is a *methodology for interactive system development.* In particular, a holistic approach to development provides a comprehensive methodology for software design, emphasizing interface development as an integral and equal part of the process. Procedures and notations are provided specifically for representing and designing the dialogue. The definition of "system" is enlarged to consider both humans and computers as components.

Also, an approach to interface development integrated with software engineering must support some form of prototyping. One of Boehm's [1983] seven basic principles of software engineering is to perform continuous evaluation. Prototyping is an effective way to begin evaluation and testing, traditionally relegated to the end of the life cycle, as early as the requirements specification phase. But prototyping can introduce serious management problems unless the process and its impact on the life cycle are well understood [Hartson and Smith 1988].

Development of a large software system is a complex task even without considering the necessity for an effective end-user interface. Many current software development methodologies are aimed at reducing this complexity for the application programmer by providing tools and guidelines for software analysis, design, implementation, and maintenance. Recent

methodological advances have begun to emphasize the role of a human factors expert in the traditional system development life cycle by including—parallel to those in place for software development—new guidelines, methods, and tools to produce quality interfaces.

It is true that addition of a dialogue developer to the overall system design team increases the need for communication in an already sizable group, which can include systems programmer, application programmer, systems analyst, application expert, end-user, and human factors engineer. A holistic system development methodology, led by a role sometimes called a systems engineer, however, embodies the activities and principles of both software engineering and human factors engineering, providing appropriate development and communication tools for each.

Human–computer interface management is a logical extension of current work in software engineering and automated development environments. Numerous methodologies fully or partially cover the software life cycle activities with varying levels of automation. Most of these methodologies use a top-down development strategy. A questionnaire-based evaluation of 24 methodologies is presented by Porcella et al. [1983]. Among these, the Jackson [1975, 1983] and Warnier–Orr [Hosier 1978] methodologies use data as the basis of design, and both derive the program structure from data structures. These, however, do not help a dialogue developer who is concerned with the human being's role during the operation of a system. The Structured Analysis and Design Technique (SADT) [Ross and Schoman 1977], Structured System Analysis (SSA) [Weinberg 1980], and Structured Design [Myers 1975, 1978; Stevens 1981; Stevens et al. 1974; Weinberg 1980; Yourdon and Constantine 1979] use the notion of a "process" as the basis of design and build the design around functions of the final system but still do not emphasize management of interface development.

Automated support tools and environments for programming already exist, and interest in them is increasing, especially within larger programming language and methodology efforts such as Ada and the associated Software Technology for Adaptable, Reliable Systems (STARS) program [*IEEE Computer* 1983]. As already noted, however, most automated tools associated with these methodologies support the developer in the coding phase of software production, not the design phase, and they generally lack tools to facilitate construction of human–computer interfaces. On the other hand, interactive tools specifically for producing interfaces, such as those presented in Section 4, often exist, but without being integrated into a methodological approach to whole system design.

We found only two software methodologies, the USE methodology and the DMS methodology, that explicitly support dialogue development as an integral part of the software development process. Both USE and DMS, described in the Appendix, provide interactive tools to support their respective methodologies.

The USE methodology features a life cycle of several phases [Wasserman et al. 1986]. Those phases that lead to creation of the human–computer interface begin with an initial analysis phase for describing activity, data, and end-user characteristics. Next is the external design of end-user interfaces, using an "outside-in" approach, working from end-user characteristics toward how the end-user will request system functions and how the system will display outputs. Then an executable prototype of the interface is created and the process is iterated until end-user and developer agree on the results. The RAPID/USE methodology has been used successfully on a large number of both experimental and commercial system development projects.

The methodology of the Dialogue Management System [Hartson and Hix 1989] also treats the human–computer interface as an integral, but clearly delineated, part of an application system. Traditional system development life cycles are primarily sequential, reflecting a "waterfall" process of moving from one distinct phase to another. Integration of interface management into the process can have a significant effect on this life cycle paradigm. Rapid prototyping assumes an important role. Also, evaluation of designs and feedback of

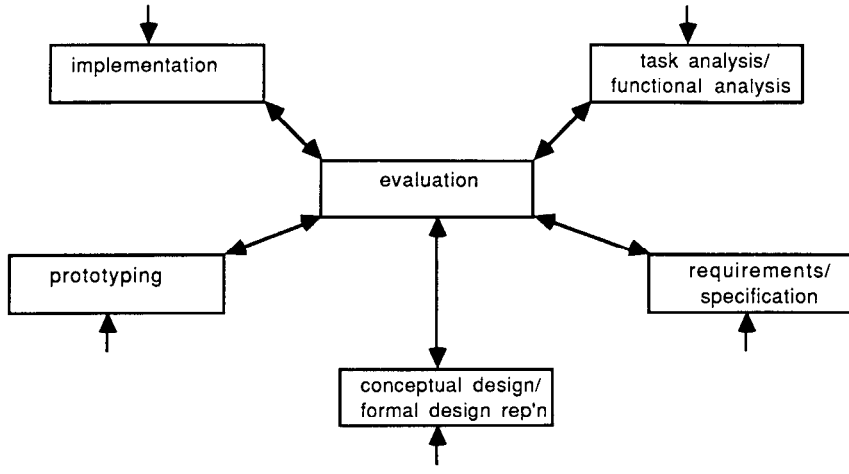


Figure 19. Star life cycle for human–computer interface development (from Hartson and Hix [1989]).

usability testing into redesign promote an iterative refinement approach that implies a truly cyclic process.

Based on qualitative empirical observations of dialogue developers producing different kinds of human–computer interfaces, Hartson and Hix [1989] concluded that human–computer interface development most naturally occurs in "alternating waves" of two kinds of complementary activities. Typical activities that are bottom-up, synthetic, empirical, and related to the end-user's view alternate with activities that are top-down, analytic, structuring, and related to a system view. These results suggest a "star" life cycle for human–computer interface development, as shown in Figure 19. This star life cycle, with evaluation at its center, supports iterative refinement and rapid prototyping. Because of its high interconnectivity, it allows almost any ordering of development activities and promotes rapid alternation among them.

7. CONTROL STRUCTURES FOR HUMAN–COMPUTER INTERFACE MANAGEMENT

Simply stated, control is the governing of logical sequencing within an interactive software system. Control flow, along with data flow, has always been a major concern in the software engineering of an interactive system. With the advent of special emphasis on the human–computer interface and its separation from noninterface parts of the system, new software architectures for application systems arose and the placement of control within those architectures became a research question. The role and placement of control in the architecture of a UIMS has become a correspondingly interesting question. Control structures are used to accomplish the sequencing and synchronization of events during execution of an interactive application system. The *control structure* of an application system can influence the way in which the system is designed, represented, implemented, and prototyped and is thus an important concept in human–computer interface management.

Control mechanisms within a target application system can be classified as either local or global. *Local control* is the control within dialogue or within computation. Dialogue control is local control for sequencing of dialogue operations such as display of prompts, acceptance of an input, validation, mapping, and resolving input errors with the end-user. Computational control is local control (e.g., for looping) used within algorithms of the functional semantic routines. *Global control* is the control that governs sequencing among dialogue and computational components.

Corresponding to the two basic types of dialogue discussed in the Introduction, there are the same two basic kinds of

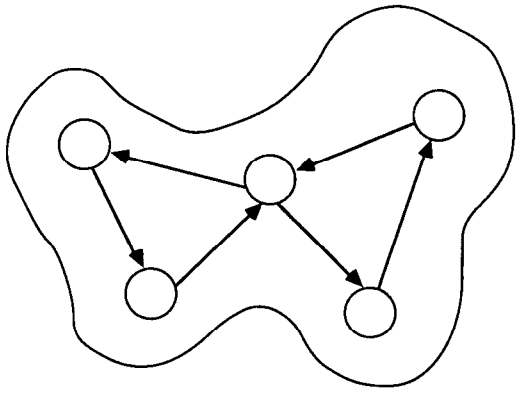


Figure 20. Control structure in a problem/solution model.

dialogue control—sequential and asynchronous. Historically, much attention has been given to control structures for sequential dialogue, but recently emphasis on asynchronous control structures has emerged.

7.1 Sequential Dialogue Control

At the highest level in the traditional top-down system development process, the problem and the solution requirements must be stated. Whenever the first "procedural" statement of system functions appears, it is often a graph-structured model indicating only the highest level of sequencing within the problem solution. At this level, very little may be known about the dialogue content or the algorithmic details of the computation that will eventually be desired. The graph structure of Figure 20 abstractly represents (without reference to a particular notation) the high-level control flow (sequencing) in the problem/solution model for some application. Each node of this graph could typically represent large amounts of both dialogue and computation.

As further development takes place, more becomes known about the functional nature of the computation performed by the computational parts and the interaction performed by the dialogue parts. These parts now become separated in the representation and require a means for sequencing them in the logical flow of the application system.

Two models of sequential control are described in some of the early UIMS literature [Rosenthal and Yen 1983]: internal control and external control. With *internal control*, the control structure of the problem/solution model is contained internally to the computational part, which invokes separately defined dialogue functions when input and output are required. *External control* is external to the computational part and is therefore, presumably, in the dialogue part. The computational part is divided into various functions that are invoked by the dialogue part, which dictates the overall sequencing. Since the terms "internal" and "external" are used with respect to the computational part, it is clear that the perspective of this work was from the computational viewpoint and not from that of the end-user. We will use the terms "computation dominant" control to refer to internal control and "dialogue dominant" control for external control.

7.1.1 Computation Dominant Control

Computation dominant control, also referred to as "embedded control" [Kamran and Feldman 1983], is illustrated in Figure 21. Application systems using prepackaged graphics software, but not UIMS, typically employ computation dominant control [Kamran and Feldman 1983]. A "slave UIMS" (not a typical UIMS) is a UIMS that conducts dialogue under direction of the computational part [Rosenthal and Yen 1983]. Computation dominant control provides a system structure that can be efficient in execution but lacks the flexibility necessary for easy modification of overall system sequencing. Also, because of the need to associate this overall system sequencing with the dialogue, this kind of system structure is awkward for early interface prototyping.

7.1.2 Dialogue Dominant Control

With *dialogue dominant control*, illustrated in Figure 22, control resides in the dialogue component. Sequencing is dependent on end-user inputs. Many UIMS produce application systems that are more or less based on a dialogue dominant model of control. Most approaches to dialogue design representation based on state

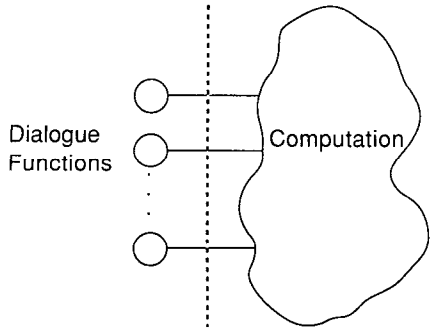


Figure 21. Computation dominant control.

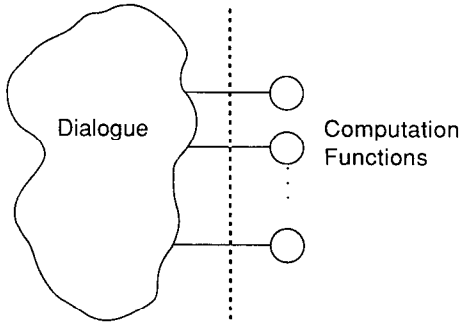


Figure 22. Dialogue dominant control.

transition diagrams or BNF use, explicitly or implicitly, the dialogue dominant control structure. The AIH, an early UIMS developed at George Washington University [Kamran and Feldman 1983], is representative. The AIH Interaction Language Interpreter interprets interface specifications and guides logical sequencing of interaction tasks. When computation is required, the Interaction Language Interpreter activates semantic routines and passes them the input values it has acquired from the end-user. Other examples of approaches and UIMS—many of which are presented in the Appendix—that produce application systems with dialogue dominant control include IDS, FLAIR, ACT/1, RAPID/USE, and Jacob's State Transition Diagrams.

In the dialogue dominant configuration, dialogue of the application system is in charge of system execution, accepting inputs and invoking the computational component when semantic processing is needed. The computational part becomes a set of attached semantic functions.

Despite their popularity, however, dialogue dominant control structures have drawbacks. Perhaps the most serious shortcoming relates to abstraction, a process used to control complexity in a design representation by hiding details inappropriate to a given level. Dialogue dominant control can result in increased complexity due to its tendency to mix levels of abstraction. Lexical and syntactic details and local dialogue control are often represented at the same level with global control and invocation of functional semantics. This is especially evident in state transition diagrams where detailed functions such as token level (syntactic) error processing and help request handling are often found at the same level of abstraction as global transitions among dialogue and computational states. This mixture of abstraction levels also unnecessarily violates dialogue independence. Because global control is mixed with dialogue, the separation of developer roles is blurred. Global control design is now the responsibility of the dialogue developer.

On the positive side, dialogue dominant control is well suited for rapid prototyping because it tends to provide a behavioral model of the entire system. By placing global control in the dialogue, this model of control easily provides a dialogue-oriented simulation of the behavior of an application system, even when much of the semantics is stubbed. For small applications these prototypes can evolve into a functional implementation. The architecture, however, is still basically that of a dialogue-oriented simulator, and its execution "requires increased (possibly substantial) computer systems resources" [Mason and Carey 1983]. Because the dialogue must deal with control flow for the whole target application system, dialogue dominant control does not, in general, offer a good top-down production-style system architecture [Gomaa and Scott 1981].

7.1.3 Mixed Control

Localization of control enforced by the computation dominant or dialogue

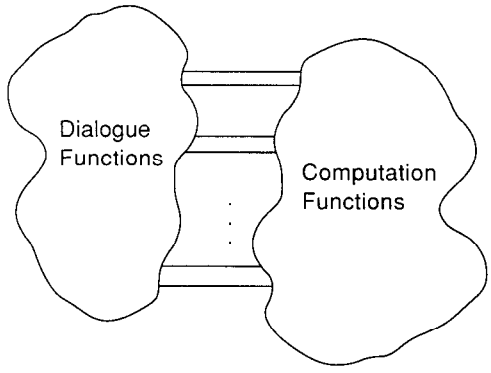


Figure 23. Mixed control.

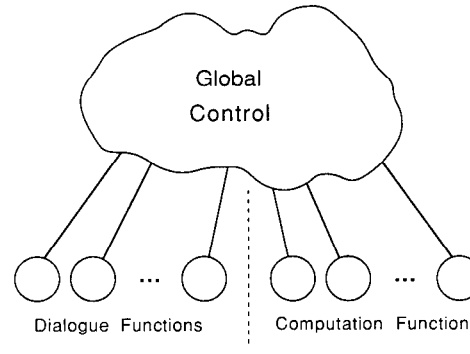


Figure 24. Balanced control.

dominant model is not always desirable. In a *mixed control* structure, shown in Figure 23, computational modules can initiate subdialogue to return intermediate results, handle errors, provide end-user feedback, and request additional information [Hayes et al. 1985]. Mixed control allows invocation of dialogue from the computational side and vice versa. This offers more flexibility but requires more discipline to maintain dialogue independence. The application programmer must subcontract dialogue design to the dialogue developer. Mixed control also means significant additional requirements are imposed on the interface definition—particularly for internal dialogue—within the UIMS [Hayes et al. 1985] to represent the additional complexity.

7.1.4 Balanced Control

Another separation, that of global control from both dialogue and computation, is also possible, yielding the control structure shown in Figure 24. Here, global control is at the top of a symmetrical, hierarchical structure. Early versions of DMS at Virginia Tech were based on this balanced model of control, requiring an application system to be divided into three independent, but communicating, components: dialogue, computation, and global control [Hartson et al. 1984]. (This was also shown in Figure 9.) The global control component governs sequencing among invocations of dialogue and computational functions.

7.2 Asynchronous Dialogue Control

Event-based mechanisms are currently the primary underlying control and communication techniques upon which asynchronous dialogue is constructed. End-user actions are sensed by device hardware and firmware (and possibly graphics software) and communicated to interface software as "events." An example of this kind of event is passing of the mouse cursor over an interface icon. The need for corresponding system action(s) is communicated by the interface. The system can still be divided into components. Communication among components is typically by message passing, and the mechanism becomes quite general by viewing each message within the system as an event.

For asynchronous control, especially for direct manipulation dialogue, there can be difficult trade-offs in making the separation into components. The direct manipulation interaction style brings the end-user cognitively closer to application semantics. To support this, the semantics must be brought closer to the end-user interface, something that tends to work against separation of the components. There are two ways the application semantics can be brought closer to the dialogue component: Build more semantic processing power into the dialogue component (especially the input part) or establish close communication between the dialogue component and the computational semantics [Hartson 1989]. The trade-off between these two approaches is essentially one that weighs a

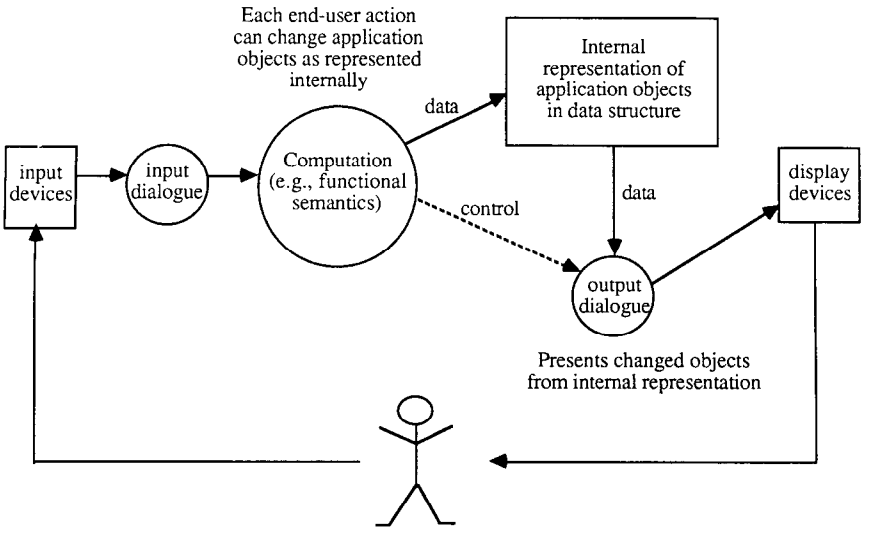


Figure 25. Communication among run-time components.

clean functional separation of components against the overhead of communication among them.

As an example of the need for semantics in the end-user interface, consider the dragging of a Macintosh file icon toward the trash can icon for deletion. If the file icon passes over a folder icon, the folder icon is highlighted to remind the end-user that there is a semantic relationship between files and folders. If the end-user releases the mouse button at that point, the file is deposited in the folder. If echoing of input actions is to be accomplished within the dialogue component, the dialogue must have semantic information about the relationship with the folder icon so that the icon can be highlighted as necessary. The alternative is for the dialogue component to communicate information about lower level input events to the computational component, which decides to highlight an object but must communicate back to the dialogue component to have the highlighting done.

Figure 25 shows a typical configuration for run-time control and communication among components. Here the dialogue component is subdivided into input dialogue and output dialogue. The input dialogue component is aware of all end-user interface and application objects and is sensitive to any events affecting objects as a result of an end-user action. The difference between sequential and asynchronous dialogue control lies primarily in whether the overall synchronizing control—which when added to the asynchronous control makes it sequential—is explicit (for sequential) or implicit (for asynchronous). Even implicit control must be real at run time. Sequential control requires the top level of control logic to be expressed explicitly by the dialogue developer. A similar top layer of control logic is required to provide synchronism even for the asynchronous case. The asynchronous control mechanism works because the input events get sent to and handled by the proper objects, and control is yielded to those objects for processing. The dialogue developer is thus afforded great freedom to isolate the behavior of individual objects and actions within complex direct manipulation, multi-thread dialogue without concern for the complicated network of control details in the high-level part of the structure.

The strong linkage between input dialogue for language parsing and graphical output dialogue necessary for responsive semantic feedback is discussed by Olsen et al. [1985] in the context of the GRINS

UIMS. Dialogue control in GRINS is based on interactive pushdown automata, which are generated from input descriptions and interpreted at run time. A layout editor is used to merge presentation information for output display under control of the automata.

Programs running on the Macintosh are not entirely asynchronous because of their strong synchronous relationship to a main control loop, which is explicitly imposed on the programmer [Apple Computer 1985]. The most significant consequence is the fact that, once a program is in control, control is absolute in the sense that no event (even outside the current window) can arbitrarily cause control to go elsewhere. It is the programmer's responsibility to respond to *any* event and voluntarily give control back to the main loop.

Mac App [Schmucker 1986] provides a high-level control structure in the form of an "application shell" for Macintosh programmers. The programmer writes routines to handle each standard type of event, and Mac App offers the control framework in which to imbed these routines. Mac App also has its own library of routines for scrolling, selecting, and launching from "empty" windows, pulldown menus, dialogue boxes, and desk accessories based on well-defined Macintosh interface standards. The programmer can fill in the contents, customize them to a specific application, and make connections to computational routines via an extendible set of standard interobject messages.

The Switchboard model of concurrent input synchronization [Tanner et al. 1986] is a good example of dialogue control specifically intended for concurrency. The Switchboard is used to route input from the multidevice stream to associated dialogue managers to handle various threads. Based on Harmony, a multitasking operating system with efficient message passing, Switchboard offers an approach that could implement the run-time components of Figure 25 as concurrent processes. Switchboard serves as the control and communication center, connecting input messages from "couriers" to computational tasks.

8. SUMMARY AND THE FUTURE OF INTERFACE MANAGEMENT

Human–computer interface management, from a computer science viewpoint, focuses on the process of developing quality human–computer interfaces, including their representation, design, implementation, execution, evaluation, and maintenance. Important concepts of human–computer interface management have been presented in this survey, providing a framework for classifying and comparing approaches to human–computer interface management. *Dialogue independence* is a characteristic that separates design of the interface from design of the computational component of an application system so that modifications in either tend not to cause changes in the other. Such independence allows easy modification of dialogues to meet the changing needs of end-users. The role of a dialogue developer, whose main purpose is to create interfaces that incorporate human–computer interface guidelines, is a direct result of the dialogue independence concept. *Structural models of the human–computer interface* serve as frameworks for understanding the elements of human–computer interfaces and for guiding the dialogue developer in their construction. *Representation of the human–computer interface* is accomplished by a variety of notational schemes for describing the interface. Numerous kinds of *interactive tools for human–computer interface development* free the dialogue developer from much of the tedium of "coding" dialogues and facilitate concentration on incorporating human factors into interfaces. The early ability to observe behavior of the interface—and indeed of the whole application system—provided by *rapid prototyping*, increases communication among system designers, implementers, evaluators, and end-users. This increased communication results in improved human–computer interfaces. A system created by a dialogue developer and an application programmer working in parallel must be developed by using an approach that gives equal emphasis to both dialogue and computational components of the software system. Such *methodologies for*

interactive system development consider interface management to be an integral part of the overall development process and give emphasis to evaluation in the development life cycle. Finally, there are several different types of *control structures* that govern how sequencing among dialogue and computational components is designed and executed.

Visions of future work in human–computer interface management are very exciting, offering opportunities in many areas within computer science, including formal modeling, graphics, software engineering, automated environments, database management, artificial intelligence, human factors, operating systems, and system performance evaluation. Interface support environments will become integrated into the operating system and hardware architecture. The current trend is away from the commonly used alphanumeric keyboard input and frame-oriented, screen-at-a-time displays toward graphics, dynamic displays, and unusual devices and communication media. With stereo 3-D graphics projected within helmets, end-users will "walk" through alternative realities of applications from molecular structures to architectural designs, navigating with body gestures and voice commands. Development methodologies and tools will have to accommodate devices, interaction styles, and input techniques we cannot now imagine. Direct manipulation will be used to even greater extents, and more attention will be given to helping a dialogue developer produce complex and dynamic output displays. There is an increasing need for a taxonomy of interface features and functions to help organize the field.

The trend toward more complex interfaces in which dialogue and its semantics are more tightly interwoven [Tanner and Buxton 1984] will present challenges to extend dialogue independence design techniques. The shift in emphasis toward asynchronism, concurrency, and multi-thread dialogue will continue. Important contributions will come from artificial intelligence, including knowledge-based systems for application areas, expert systems to aid dialogue design, and improved natural language processing. More "intelligence" will be used in interfaces to adapt to the variability among human users. Interface management and software engineering will continue to share the trend toward less code writing and more automatic code generation. Human–computer interface management will receive an increasing share of attention within the interactive system development process.

Progress on the computer science side of human–computer interface management will spawn requirements for future work by our human factors colleagues. For example, now that rapid prototyping is available, its effective use in the iterative refinement process to produce quality interfaces must be more thoroughly explored and exploited. Although summative evaluation will continue to be used for controlled testing of isolated principles and will continue to contribute to the theory of human factors for human–computer interfaces, formative testing of entire interfaces and entire systems will become a part of the development process [Williges 1984; Wixon et al. 1983]. Similarly, stronger inputs from the cognitive and behavioral sciences will contribute to a better understanding of the basic process of human–computer interaction and will direct future computer science work in this area. One of the most significant of these is the need for improved usability in the UIMS themselves.

The tools and techniques surveyed in this article will not remain solely in the domain of system developers. Much of what we have discussed here will be integrated into large applications themselves, becoming directly available to increasingly sophisticated end-users. Database management systems will be offered as utilities built into applications with a large variety of interface options. End-user performance metering aids will accompany commercial software. Tools similar to those used by dialogue developers will become available to end-users for customizing their own interfaces.

We have presented many concepts and ideas in the field of interface management; now it is time to see if they will really work. This "seeing" will involve significant evaluation effort and technology transfer [Ehrlich 1985] to real-world application

environments—everything from the space shuttle to the ubiquitous personal computer. Many of the easy questions are answered; many difficult questions remain. Human–computer interface management is currently much more art than science. Making the transition to a more scientific approach, while maintaining a human perspective, promises to be a challenge for human–computer interface management research well into the future.

APPENDIX: A SAMPLER OF SYSTEMS FOR HUMAN-COMPUTER INTERFACE MANAGEMENT

The concepts discussed in the main part of this paper establish a framework for the management of human–computer interface development. Embodiment of these concepts requires application system development facilities and tools that incorporate the concepts, from the first phases of design all the way through to implementation and into the iterative refinement and maintenance phases.

This appendix is a sampler that describes several systems, which, to varying extents, manage representation, design, implementation, prototyping, execution, evaluation, and maintenance of interfaces for interactive human–computer systems. We chose representative systems for their breadth of scope and the variety of ways in which interface management and the entire application system development process is approached. Many of these systems represent foundational, landmark pieces of work (either research or commercial) and, as such, deserve recognition. Most (BLOX, COUSIN, DMS, GWUIMS, Open Dialogue, RAPID/USE, Prototyper, State Diagram Specification Interpreter, toolkit UIMS, and the University of Alberta UIMS) are systems for management of the interface across many phases of the life cycle, whereas a few (FLAIR II and RIPL) are primarily rapid prototypers. These systems generally go beyond the realm of limited application-specific formats, devices, and interaction techniques and address the much more complex issues at the very heart of human–computer interface management. The rapid prototypers were included in this appendix to emphasize the importance of this stage of the life cycle on interactive system development and to present a variety of approaches to prototyping.

To collect the data for the systems in this appendix, we sent a sizable questionnaire to the groups developing each of these systems. We used information from the questionnaire extensively (in some cases, verbatim, with permission) to prepare this section, which is arranged essentially like that of the questionnaire: The system is described in general and then set in the conceptual framework developed in the paper. Several other features that classify and describe such systems, but are not explicit concepts of interface management, are also given. The systems are presented in alphabetical order. They are intentionally not compared since the purpose of this appendix is rather to give the reader an overview of each system. Descriptions are intended to be complete but not necessarily detailed, since many details have already been given in other sections of the paper when specific parts of these systems were used to illustrate a particular concept. The complete form of the questionnaire, including explanations of many of the terms, follows.

Format of Questionnaire

1. General description of system
2. Interface management concepts
   a. *Dialogue independence:*
   b. *Structural model of interface:*
   c. *Representation of interface:*
   d. *Interface development tools:*
   e. *Dialogue developer role:*
   f. *System development methodology:*
   g. *Rapid prototyping:*
   h. *Control structure:*
3. Features of system
   a. *Internal representation of interface definitions* (e.g., tables, relational databases, executable code):
      - *At implementation time:*
      - *At prototyping time:*
      - *At run time:*

b. *Lexical constraints* (e.g., Does your system handle character-at-a-time validation of end-user input?):

c. *Input dialogue* (Is your system used to produce dialogue to extract inputs from the end-user? If so, how does this dialogue validate (check for errors in) end-user input, or are inputs validated by the computational component?):

d. *Output dialogue* (Is your system used to format and display data dependent computational results—e.g., information retrieved from a database—as well as to develop input-related dialogue? If so, how does it deal with these types of output for which the form and values are not bound (or known) until run time?):

e. *Relationship between input and output dialogue* (How do you make this distinction, if you do, in your approach? How do you classify things like error messages, prompts, and help information in this regard?):

f. *Help* (Do your interface development tools provide specifically for the development of help information? If so, how?):

g. *Pragmatics* (Does your system or your work address end-user gestures and actions, physical device characteristics, and other "pragmatics" of interfaces? If so, how?):

h. *Multiple input devices* (Can more than one physical input device be active at one time? If so, how?):

i. *Support environment and graphics* (What hardware and software does your system run on? Does your system make extensive use of graphics?):

4. Miscellaneous questions

a. *Human factors built in* (Do the tools of your system enforce specific human factors principles within the dialogue development process?):

b. *Sequential versus asynchronous dialogue* (Do your dialogue development tools produce dialogue that is essentially sequential or can they also produce asynchronous dialogue?):

c. *Generality of interaction style* (Is your system oriented toward a specific interaction style—e.g., menus, graphical input, form filling—or is it more general?):

d. *Interface evaluation* (Does your work address evaluation of interfaces? If so, how?):

5. Implementation

a. *Languages:*

b. *Operating system:*

c. *Date work begun:*

d. *Status* (experimental, internal product, commercial product):

e. *Personnel* (computer scientists, human factors experts, psychologists, other):

f. *Self-creating* (Could your system be used to design/create itself?):

BLOX Graphics Builder

1. General Description of System

BLOX Graphics Builder, developed and marketed by Rubel Software of Cambridge, Massachusetts, is designed to reduce the amount of programming required to create graphics application screens, menus, and icons [Rubel 1982]. The end-user interface is developed interactively using the BLOX direct manipulation tools, TableGEN, and SymbolEDIT.

The BLOX screen editor, TableGEN, is used interactively to draw interface screen layouts and menus with pen or mouse. An interface can contain any number of screen layouts, including multiple work, message, and menu areas. Menus contain text and graphical icons. Each menu item has an associated prompt string and action subroutine, supplied either by BLOX or by the application developer. BLOX provides defaults for all interface attributes and responses. These defaults can be changed by the application developer.

With the BLOX icon editor, SymbolEDIT, pen or mouse is also used interactively to draw graphical icons and store them in sets. Each set of icons can be later retrieved and edited interactively. Icons can be used as menu items in an application interface or as part of an application data

display, such as a diagram or chart that includes predefined symbols. Icons can be displayed with any scale and rotation.

BLOX HelpGEN automatically generates an on-line help keyword file for an application. This file contains a keyword for each menu item or other area of the application screen. Any standard text editor can be used to insert customized help information into this file. On-line help is retrieved at run time by pressing a "help" button and pointing at the part of the screen in question.

Two BLOX Subroutine Libraries are provided for use with BLOX-built applications. These libraries provide many precoded functions for the application developer. The BLOX User Interface Library is a collection of subroutines for query and manipulation of the end-user interface. These include displaying prompt lines, pop-up menus, new screen layouts, icons, grids, and many more. The BLOX Graphics Library is a more traditional graphics subroutine library, with the ability to draw graphics primitives such as lines, circles, arrows, text, and filled areas.

BLOX facilitates rapid prototyping by allowing dialogue developers to draw symbol sets and interface screens quickly and then "testdriving" the end-user interface before linking to application code. BLOX development tools enable developers to produce a standard end-user interface for all graphics applications, regardless of hardware configuration. BLOX can be used to develop interactive graphical end-user interfaces for existing code, as well as for new applications under development. BLOX, written in FORTRAN, can be used with any programming language that is compiled and callable from FORTRAN.

Once an interface has been designed, BLOX automatically links it to application code through the BLOX Interaction Handler. The Interaction Handler responds to all end-user or machine-generated input. BLOX supports input devices such as pen, mouse, and keyboard and input techniques such as graphical menu selection and multiple button input. Typical response to input includes display of graphics, prompt messages, new screen layouts, pop-up menus, or calls to coded subroutines. Coded subroutines can be either BLOX or developer supplied and can be written in any compiled programming language. BLOX, being machine, device, and application independent, has been used to develop applications in diverse scientific, engineering, and academic areas.

2. Interface Management Concepts

a. *Dialogue independence:* Interface development is completely separated from that of computational code. Application action routines can be associated with menu buttons during menu design, but the two parts of the application are treated independently.

b. *Structural model of interface:* None.

c. *Representation of interface:* Done using interactive tools, TableGEN and SymbolEDIT.

d. *Interface development tools:* SymbolEDIT lets the dialogue developer produce iconic symbol sets that can be accessed during application screen and menu design. TableGEN lets the developer indicate where work areas, menu areas, and message areas are desired. In the case of menu areas, the contents are also represented interactively. All areas are given default properties, which can be easily changed by the developer.

e. *Dialogue developer role:* Role is supported by direct manipulation tools for interface development.

f. *System development methodology:* None.

g. *Rapid prototyping:* Simple end-user interfaces can be designed quickly. TableGEN lets the developer produce an interface with any combination of work, menu, and message areas. Once an interface has been designed in a TableGEN session, the end-user can "test-drive" it. The interface is shown full screen before any application code is linked. Several test-drive sessions are generally run before an interface is linked into application code.

h. *Control structure:* BLOX is a dialogue dominant or external control UIMS. It likes to be the "controller" of the application. The application and interface,

however, can be influenced by computational functions, as well as by input from the end-user.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Tables generated from drawings during a "test drive."
   - *At prototyping time:* Tables.
   - *At run time:* Tables compiled into binary.

b. *Lexical constraints:* Handled as input dialogue.

c. *Input dialogue:* End-user input validation is the responsibility of the computational component.

d. *Output dialogue:* Must be programmed by an application programmer.

e. *Relationship between input and output dialogue:* Three BLOX area types—work, menu, and message—provide the standard for distinguishing between input and output dialogue. BLOX work areas are for both end-user input and application output and display. Menus are solely for end-user input. Message areas are solely for application output. Error messages, prompts, and help are contained in message areas. Given these distinctions, it remains the decision of the application designer as to where and how these areas are used.

f. *Help:* A BLOX development tool, HelpGEN, automatically generates help keyword files for each BLOX area and menu item. These files can then be edited with a text editor.

g. *Pragmatics:* BLOX is based on interactive end-user input, through the use of menus and interactive work areas. BLOX Graphics Builder requires specific device drivers for particular graphics terminals.

h. *Multiple input devices:* Single process dialogue system.

i. *Support environment and graphics:* Currently runs on VAX/VMS, VAX/UNIX, and several 68000-based UNIX systems (SUN, Silicon Graphics, Masscomp, Cadmus, and Orcatech). BLOX includes an implementation of the Core graphics library. Although BLOX is written in FORTRAN, it can be used with any programming language that is compiled and callable from FORTRAN. BLOX was designed specifically for graphics applications. Icons are developed by the application developer and then used in the application at run time. BLOX areas are generally used for graphics display, although developers can use text in conjunction with graphics.

4. Miscellaneous Questions

a. *Human factors built in:* There are very few constraints placed on a dialogue developer. BLOX helps developers build applications quickly but does not eliminate the possibility of a badly designed interface.

b. *Sequential versus asynchronous dialogue:* BLOX is based on sequential, turn-taking dialogue. End-user input triggers a computer process, display, or other reaction.

c. *Generality of interaction style:* BLOX is designed around graphical menus. Items are selected by pointing, rather than by typing a selection's number. End-user's input can also be entered through the graphical work area.

d. *Interface evaluation:* Interfaces are evaluated during the design process (see earlier explanations of "test drive") by the application developers and others. BLOX itself makes no attempt to evaluate an interface. Because BLOX interfaces are easily modified, end-users can also critique interfaces.

5. Implementation

a. *Languages:* FORTRAN.

b. *Operating system:* VMS, UNIX.

c. *Date work begun:* 1982.

d. *Status:* Commercial product.

e. *Personnel:* Four computer scientists; three sales/marketing.

f. *Self-creating:* Yes. BLOX Graphics Builder's tools are based on the three BLOX areas and could have been built with BLOX. BLOX has been used to continue development of the product.

COUSIN

1. General Description of System

The COUSIN (COoperative USer INterface) system [Hayes 1985; Hayes and Szekely 1983; Hayes et al. 1981, 1985] of Phil Hayes, Eugene Ball, Raj Reddy, Richard Lerner, and Pedro Szekely at Carnegie-Mellon University has an artificial intelligence flavor and deals with natural language understanding but nicely illustrates some interface management concepts, especially dialogue independence or "tool independence."

Early COUSIN research revolved around definition of a quality interface that supports graceful interaction just as human-to-human communication is graceful and robust. Such interaction thus goes well beyond the traditional principles of human–computer interaction and into the realm of natural language understanding [Hayes and Reddy 1983]. Because of the enormous difficulties of producing such an interface, the researchers propose to amortize this effort by building a single, application-independent ("tool-independent") system to serve as the end-user interface for a variety of subsystems rather than developing a separate interface for each application system. The implication in Hayes et al. [1981] is that there is one interface for many application systems (tools) and that interface does not contain information about a particular application but obtains a declarative definition of the application from a tool description database. A subsequent report [Hayes and Szekely 1983] states that this declarative database contains definitions of the end-user communication (interface) needs of an application system and that a single tool-independent interface interpreter is used to instantiate the interface for that application system. A significant contribution of the COUSIN work is that its dialogue-dominant control structure departs from strict sequential dialogue and handles concurrency of many communication media (e.g., simultaneous pointing, speaking, and typing).

More recent work on COUSIN has evolved an interface definition centered around form-based interface abstractions, expressed in a language that is interpreted [Hayes 1985]. Such an interface definition consists of a declaration of the form name followed by a sequence of field definitions containing attributes. COUSIN's interface definition language is based on a communication abstraction between end-user and application in which communication takes place through a set of value-containing "slots"—one slot for each piece of information the end-user and application need to exchange. A simple print application might have slots for its parameters, such as the file to print and number of copies. External interface definitions expressed in form-based abstractions are used by COUSIN to provide a wide variety of applications with consistent, quality interfaces. End-users of systems produced with COUSIN interact with those systems by filling forms. End-users specify parameters to a command by filling in the appropriate fields in the form (in any order) and then execute the command. Further interaction with the command while it is running is also done by displaying or changing data in fields. Fields in the form correspond to slots in the interface definition. At run time there are two processes per application system: One is the application system itself and the other is COUSIN, operating to support the application system. COUSIN interprets the interface description, puts bits on the screen, and interprets keystrokes on behalf of the application system. Future efforts will concentrate on providing a larger specification process.

2. Interface Management Concepts

a. *Dialogue independence:* Yes. The dialogue developer is encouraged to think in terms of pieces of data exchanged by the application and the end-user, not in terms of how data are displayed or how the end-user modifies it. This is done with "slots." Application programs access slots with a set of accessor routines.

The end-user accesses slots via a graphical form-based representation of the slots.

b. *Structural model of interface:* None.

c. *Representation of interface:* Dialogue is represented by defining "interaction modes" of fields. The interaction mode specifies both how to display a field (icon, text string, menu, etc.), and how to interpret input events directed to that field. Control of the dialogue (what to do next) is not represented explicitly. Dialogue representation consists of a set of [attribute, value] pairs for each field of the form.

d. *Interface development tools:* The [attribute, value] pairs are represented as a text file with a text editor. COUSIN also has a graphical editor that can be used to edit the compiled version of the representation and is capable of generating a textual description. The COUSIN interface to construct the [attribute, value] pair that is the dialogue representation has a field for each attribute; the value of the field is the value of the attribute. A layout editor is a WYSIWYG editor to edit the layout of the end-user form. A mouse can be used to move edges of fields or to enter numbers in another form to represent coordinates of fields.

e. *Dialogue developer role:* COUSIN uses two roles. An application program builder develops the application program using a set of routines provided by COUSIN to access or update the values of slots. A dialogue developer, called an end-user interface writer, develops the interface description using the tools described above. The application program builder and the dialogue developer have to agree on the number and type of slots.

f. *System development methodology:* No particular methodology is used. Several "nontoy" systems, however, have been developed using COUSIN. Development is iterative, starting with a simple application program with a few slots and then adding more functionality by adding more slots (and commands).

g. *Rapid prototyping:* COUSIN allows easy building of facades/mockups of application programs. COUSIN can generate a usable form for an application program from the interface description, and there is an application program called "cappl" (*COUSIN–appl*ication), which can be used as a dummy with any interface description. Forms with fields for unimplemented commands are easily produced. The end-user can interact with the form, but upon invoking an unimplemented command, a "Not Yet Implemented" message is displayed.

h. *Control structure:* Sequencing is mostly external, controlled by the end-user. When the application program needs some data, however, it can take the initiative and force the end-user to respond to questions. This second mode is intended to be used only when necessary. COUSIN encourages the external, dialogue dominant method of communication.

3. Features of System

a. *Internal representation of interface definitions*

- *At implementation time:* Complicated set of data structures that represent "slot" values.
- *At prototyping time:* Complicated set of data structures that represent "slot" values.
- *At run time:* Complicated set of data structures that represent "slot" values.

b. *Lexical constraints:* COUSIN takes all characters and inserts them into a buffer. Validity can be checked when a "break" key is hit, but each interaction mode can check its input at any point. In this case input is not validated a character at a time.

c. *Input dialogue:* COUSIN extracts all input from the end-user. COUSIN knows about a few data types (e.g., integer, string, Boolean) and has a very simple constraint language. It can recognize simple errors and interact with the

end-user to correct them before giving the data to the application program. COUSIN cannot, however, recognize application program-specific semantic errors.

d. *Output dialogue:* COUSIN updates a display whenever a change is made to a slot, which is controlled by the interaction mode of that slot.

e. *Relationship between input and output dialogue:* Such components as prompts or error messages are not explicitly classified as either input or output. In COUSIN each interaction mode has its input strongly linked with its output, but interaction modes are independent of each other. Dialogue is divided into "chunks," each chunk being an interaction mode that is not divided into an input and an output part.

f. *Help:* COUSIN emphasizes a help system, generating two levels of help. Short help is a line of text specifying the purpose of a field. Long help allows the end-user to traverse a network of help frames describing the application program. Short help messages are represented in the interface definition. Help frames for long help are generated by COUSIN from the run-time state, the interface definition, and a text file.

g. *Pragmatics:* No.

h. *Multiple input devices:* Both keyboard and mouse are active at the same time. Events from devices are represented as messages from processes. Hence, events from multiple devices appear mixed with each other in a message queue and are handled by COUSIN with priorities by "time slicing" between them.

i. *Support environment and graphics:* COUSIN runs on Perq workstations with graphics display and mouse. The operating system is "Accent," which provides fast message-based communication between multiple processes. COUSIN runs as a separate process from the application program for which it provides an interface. COUSIN makes moderate use of graphics. A portion of the screen can be defined as a graphics area and information from this area passed to the application program. The application program directly calls graphics package routines to paint that area.

4. Miscellaneous Questions

a. *Human factors built in:* COUSIN produces interfaces that are fill-in-the-blank forms. No constraints, however, are placed on the format and content of these forms.

b. *Sequential versus asynchronous dialogue:* Since COUSIN and the application program execute in separate processes, the end-user can interact with the form of an application program even when the application program is computing. For example, an end-user can enter parameters for the next command while the current one is executing. A locking mechanism prevents interference between the end-user and the application program.

c. *Generality of interaction style:* Form filling, with limited graphical interaction.

d. *Interface evaluation:* No.

5. Implementation

a. *Languages:* Dialect of Pascal.

b. *Operating system:* Accent.

c. *Date work begun:* 1981.

d. *Status:* Experimental.

e. *Personnel:* Three computer scientists.

f. *Self-creating:* The tool to edit interface descriptions is itself a form produced by COUSIN.

Dialogue Management System (DMS)

1. General Description of System

The Dialogue Management System (DMS) [Hartson et al. 1984; Hix and Hartson 1986; Roach et al. 1982] being developed at Virginia Tech by H. Rex Hartson, Deborah Hix, and Roger W. Ehrich, is a comprehensive system for interface management. An application system developed using DMS is viewed as having three components: a dialogue component through which all

communication between the end-user and the application system is carried out, a computational component that contains all semantic processing algorithms, and a global control component that governs logical sequencing among dialogue and computational components. Dialogue independence forms the fundamental philosophy of DMS and helps ensure easy modification of the interface, allowing two or more very different interfaces to be used with the same computational and global control components.

Interface management is considered to be an integral part of the overall software engineering process. To support this premise, a system development methodology and evaluation-centered ("star") life cycle have been produced as part of the DMS research. The methodology (called the SUPERvisory Methodology And Notation, or SUPERMAN, in early versions) integrates development of all three components of an application system using a technique called supervised flow diagrams to represent the design of a system. Supervised flow diagrams are an executable representation of control flow and data flow in all components of the target application system being developed using DMS.

The DMS Design-Time Facility provides an integrated set of tools for interactively developing each of the three components. A dialogue developer designs, implements, and modifies the dialogue component (the human–computer interface) using a set of tools (called the Author's Interactive Dialogue Environment, or AIDE, in early versions). These direct manipulation tools allow the dialogue developer to work with objects, rather than source code, when developing an interface. DMS also contains a graphical programming language (GPL) editor that is used interactively to develop supervised flow diagrams for the global control and computational components. A specialized version of the GPL editor, used to develop supervised flow diagrams in the dialogue component, is constrained to conform to the DMS structural dialogue transaction model. The computational component is designed and implemented largely using a conventional programming support environment.

The rapid prototyper, called the Behavioral Demonstrator, allows early and continuous evaluation and modification of an application system design. At run time, dialogue executors and mechanisms for linking all the components support execution of application systems produced by DMS.

An evaluation of DMS has shown its tool-based approach to interface development to be faster than conventional methods involving source coding. In an empirical study, the use of AIDE for implementing an interface produced a nearly four-to-one improvement in speed over the use of programming for implementing the same interface. The DMS methodology and approach have been used successfully to develop a number of substantial applications, including a relational database, a document storage and retrieval system, and DMS itself. Two versions of DMS have been implemented, and DMS 3.0 was due for completion in late 1988. DMS 3.0 is built on a Smalltalk-80 (object-oriented) platform running on a Macintosh II. DMS 3 makes more extensive use of direct manipulation in its own interface than previous versions did. DMS 4 will have a greatly expanded capability to produce direct manipulation, multi-thread, and asynchronous dialogue in application system interfaces.

2. Interface Management Concepts

a. *Dialogue independence:* Dialogue is separated from computation and global control at design time; raw dialogue tokens to and from the interface are mapped to normalized tokens for use throughout the rest of the application system.

b. *Structural model of interface:* A dialogue transaction model describes the human–computer dialogue at three linguistic levels: semantic, syntactic, and lexical.

c. *Representation of interface:* At the semantic level, dialogue transactions are represented in supervised flow diagrams. At the syntactic and lexical levels, AIDE tools are provided for representing such interface objects as screen layouts, input definitions, and token mappings.

d. *Interface development tools:* The dialogue developer does not write source code to implement dialogue but rather uses AIDE to produce interface objects by direct manipulation.
e. *Dialogue developer role:* Yes; it uses AIDE to implement interfaces.
f. *System development methodology:* The SUPERvisory Methodology And Notation (SUPERMAN) is used to develop a human–computer system through all stages of an evaluation-centered ("star") development life cycle.
g. *Rapid prototyping:* The Behavioral Demonstrator is used to execute supervised flow diagrams, demonstrating parts of the evolving application system as they are developed. Stubs and temporary values for yet undefined variables are provided for those parts that are not yet developed.
h. *Control structure:* Balanced control in which the global control component governs sequencing among dialogue and computational functions.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Relations in database.
   - *At prototyping time:* Relations in database.
   - *At run time:* Relations in database.
b. *Lexical constraints:* End-user inputs are processed a single end-user action at a time, allowing immediate validation of inputs.
c. *Input dialogue:* At run time, lexically validated keystrokes or other actions from the end-user are collected into tokens (interactions) as directed by the input definition. These tokens are syntactically validated and collected into sentences (transactions, e.g., a complete command with its operands). Actions, interactions, and transactions are defined by the dialogue developer at design time using AIDE. Validated and normalized tokens of a transaction are passed, via the global control component, to the computational component at run time.
d. *Output dialogue:* At run time, a dynamic output executor accepts results of computational processing and displays results to the end-user based on definition of the output produced by the dialogue developer using AIDE at design time.
e. *Relationship between input and output dialogue:* Dialogue objects can be defined as both output and input objects so that these two kinds of dialogue can be linked (e.g., a graphical object can be displayed as computational output and then picked for manipulation by the end-user).
f. *Help:* Dialogue development tools do not provide specifically for the development of help information but can be used to produce help just as they are used for any part of the dialogue.
g. *Pragmatics:* Physical devices and end-user gestures are represented at the lowest level of the interface definition; above this, interface definition is device independent.
h. *Multiple input devices:* Theoretically, any number of physical input devices can be active at one time if the dialogue developer designs the interface this way. The run-time dialogue executor has the capability to poll devices and read the appropriate one(s).
i. *Support environment and graphics:* A Silicon Graphics IRIS 2400 Workstation running UNIX and Smalltalk-80 on a Macintosh II. Graphical display objects can be produced with a graphical editor, one of the DMS interface development tools.

4. Miscellaneous Questions

a. *Human factors built in:* No; theoretically there are no constraints on the interface.
b. *Sequential versus asynchronous dialogue:* All versions of DMS support a broad variety of sequential dialogue interaction styles. Direct manipulation and asynchronous dialogue are supported in DMS 3 and, to a greater, extent, will be in DMS 4.

c. *Generality of interaction style:* Any interaction style can, in theory, be created and executed using DMS development tools.
d. *Interface evaluation:* Metering of end-user input is allowed at each of the three linguistic levels. Metering of dialogue development activities will be provided in the future.

5. Implementation

a. *Languages:* DMS 2, —C; DMS 3, —Smalltalk-80.
b. *Operating system:* DMS2, —UNIX (on a Silicon Graphics IRIS 2400 Workstation); DMS 3 —Smalltalk (on a Macintosh II).
c. *Date work begun:* DMS 1, —1980; DMS 2, —1985; DMS 3, —1987.
d. *Status:* Experimental research product.
e. *Personnel:* Three computer scientists, varying numbers of graduate students and programmers.
f. *Self-creating:* In theory, yes; implementation is not advanced enough at this time for DMS to produce itself.

FLAIR II

1. General Description of System

FLAIR II (Functional Language Articulated Interactive Resource) [Wong and Reid 1982] developed at TRW by Peter Wong, Eric Reid, Phil Schmidt, and Christopher Barbay, is an interface rapid prototyping system capable of prototype generation and interpretive or compiled execution of the prototype. This color graphics based tool is intended for development of interactive systems for either menu-based or keyword-based interactive systems. Prototype developers can use FLAIR's show-by-example menu method to produce dialogues for single or multi-screen graphics systems. FLAIR can handle a variety of input devices, from keyboards to voice recognition devices. Menu selections allow an extensive library of FLAIR development tools (called microprimitives), such as maps, symbols, constructors, and other graphics entities, to be accessed at the touch of a button. The Shell, an interface developed specifically for FLAIR, can dynamically link the prototype's execution to the end-user application routines. Any VAX text editor can easily modify the command files generated by FLAIR to change the behavior, graphics, and application linkages.

The FLAIR interface is considered to be a Dialogue Definition Language (DDL), a menu-driven system that directs a dialogue developer through a coherent and orderly translation of the prototype into a form that is executable as an end-user system. FLAIR deciphers and codes initial protocols of the input/output devices as a single standard set of protocols for the prototype being developed. FLAIR has static frames, scenario dialogues, and dynamic system scenarios to support system development. Static frames allow construction, storage, and retrieval of a picture. Scenario dialogues display a sequence of frames, control the sequence of frames logically as a result of interactive end-user inputs, and audit and time-stamp end-user prototype interaction for later analysis. Dynamic system scenarios simulate multiple work station communications and environmental conditions in order to measure an end-user's responses.

FLAIR can be used in most interactive computing situations. The system has been used in prototyping computer-aided instruction (CAI) systems; command, control, communication, and intelligence (C^3I) systems; computer-aided engineering (CAE) systems; cartographic systems; and as a front-end driver for decision support for various prototype systems. Use of FLAIR has resulted in approximately a two-to-one reduction in time required for building graphical interactive displays.

2. Interface Management Concepts

a. *Dialogue independence:* Dialogue is separated from computation. FLAIR can calculate dynamic values; end-users can attach their own programs to FLAIR during run time for additional computational requirements.
b. *Structural model of interface:* None.

c. *Representation of interface:* Dialogue representation is done either of two ways: show by example or keyword entries. In show by example, the developer draws an example of what some portion of the interface is to look like. For keyword entries, the developer defines a typed command string; keywords of functions represent what the function is to do.
d. *Interface development tools:* Initial representation is done by pointing or keyword. Subsequent dialogue editing is done through a text editor. By using a pointing device one can show FLAIR how to proceed in a dialogue.
e. *Dialogue developer role:* Not explicitly; system designer provides the overall scheme.
f. *System development methodology:* Not bound to a specific methodology.
g. *Rapid prototyping:* FLAIR is primarily a rapid prototyper.
h. *Control structure:* Primarily dialogue dominant; control structures can be built into any graphics (dialogue) entities.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Semistate table, textual code.
   - *At prototyping time:* Semistate table, textual code.
   - *At run time:* Compiled state table with binary code.
b. *Lexical constraints:* System handles range, exception, and inclusion value checking if the developer wants to put these constraints in the system. String checking can be done for string size and string match.
c. *Input dialogue:* Input device dependent modules accept and translate end-user input device signals.
d. *Output dialogue:* Subroutine calls to FLAIR can be used by the system designer in the output dialogue.
e. *Relationship between input and output dialogue:* No distinction made.
f. *Help:* No specific tools.
g. *Pragmatics:* Handles graphics tablets, light pen, joystick, mouse, touch panel, trackballs, function keys, voice recognition, and (in the future) complete vision.
h. *Multiple input devices:* No, single device at a time.
i. *Support environment and graphics:* Runs under VMS 4.1 using Core graphics. FLAIR is based on graphical as well as keyword dialogue; both can interface to control structures.

4. Miscellaneous Questions

a. *Human factors built in:* No.
b. *Sequential versus asynchronous dialogue:* Generally produces sequential dialogues; however, an environmental generator can perform concurrent processing for some asynchronous dialogues.
c. *Generality of interaction style:* Menus, graphical input, keyword/sentences, voice recognition, and form prompting are supported.
d. *Interface evaluation:* Not directly; however, an audit trail can be generated for dialogue error evaluation.

5. Implementation

a. *Languages:* FORTRAN, Assembler.
b. *Operation system:* VMS 4.1.
c. *Date work begun:* February 1981.
d. *Status:* Internal product.
e. *Personnel:* Four computer scientists.
f. *Self-creating:* Probably.

George Washington University UIMS (GWUIMS)

1. General Description of System

This research represents an attempt to develop a general architecture for user interface management system development that is in some ways analogous to an expert systems shell for expert systems development. This UIMS was developed by John Sibert, Dave Hurley, and Teresa Bleser at the George Washington University. Based

on an object-oriented programming paradigm, it is a departure from traditional UIMS development [Sibert et al. 1988]. The GWUIMS is related to earlier work at GWU by its incorporation of Foley's three levels of an interaction language: lexical, syntactic, and semantic levels. These three levels are incorporated by embodying the boundaries between levels within object classes. This was determined by the observation that many of the hardest end-user interface design problems seem to involve the boundaries between Foley's levels and by the desire to provide for intelligence at those boundaries.

Interactive interfaces are represented by direct manipulation customizing of interface objects "cloned" from an object template library. The motivation is to provide interactive rapid prototyping for a variety of dialogue styles. This is a research-oriented system, currently supporting only a few interaction techniques and minimal design tools.

2. Interface Management Concepts

a. *Dialogue independence:* Computation is carried out by "application objects" that communicate with dialogue objects by message passing.

b. *Structural model of interface:* Layered, based on semantic, syntactic, and lexical language levels.

c. *Representation of interface:* Abstract objects; language representation is interactive customizing of "template" objects such as menus.

d. *Interface development tools:* Direct manipulation tools, often using menus.

e. *Dialogue developer role:* Builds and tests individual manipulation techniques as well as dialogues; essentially a combination of authoring and graphic design.

f. *System development methodology:* No specific methodology but uses a combination of top-down design with iterative enhancement and refinement.

g. *Rapid prototyping:* Rapid prototyping uses a set of generic application objects that simulate a variety of behaviors using probabilistic simulations.

h. *Control structure:* Dialogue dominant control is resident in the support environment.

3. Features of System

a. *Internal representation of interface definitions*

- *At implementation time:* Executable LISP code with some tables and compiled C code.
- *At prototyping time:* Executable LISP code with some tables and compiled C code.
- *At run time:* Compiled LISP and tables and compiled C code.

b. *Lexical constraints:* Lexical constraints handled with arbitrary interaction techniques written in C code and interfaced to the LISP environment.

c. *Input dialogue:* Some end-user errors are trapped by dialogue (e.g., clicking mouse when there is no target). Semantic errors are detected generally by application objects that report success or failure back to the appropriate dialogue object.

d. *Output dialogue:* All messages and prompts are defined as representation objects, which are output.

e. *Relationship between input and output dialogue:* Input dialogue is handled by a set of "interaction objects," and output dialogue is handled by a set of "representation objects."

f. *Help:* Developed in the same way as any message.

g. *Pragmatics:* Currently being researched.

h. *Multiple input devices:* Yes; uses a "listener object" similar to that found in most window systems. Uses a table-driven parser to determine an appropriate action in response to events from devices; any movement of an active device is an event.

i. *Support environment and graphics:* Runs under UNIX and is programmed in Franz LISP and Flavors. Graphics is all pervasive as icons, process indicators, and interaction techniques.

4. Miscellaneous Questions

a. *Human factors built in:* No; system is intended to provide for inclusion of human factors.
b. *Sequential versus asynchronous dialogue:* Can produce more than just sequential dialogues, although limited by operating system considerations in actual performance.
c. *Generality of interaction style:* General; new interaction techniques can be added.
d. *Interface evaluation:* Not specifically addressed at this time.

5. Implementation

a. *Languages:* LISP (Franz, Flavors).
b. *Operating systems:* UNIX.
c. *Date work begun:* March 1985.
d. *Status:* Experimental.
e. *Personnel:* Three computer scientists, one artist.
f. *Self-creating:* Yes.

Open Dialogue

1. General Description of System

Open Dialogue is a follow-on product to Domain/Dialogue, an Apollo product first released in 1985 [Schulert et al. 1985]. Like Domain/Dialogue, Open Dialogue allows a dialogue developer to use a declarative definition language to describe the human–computer interface to an application separately from the application itself. This interface definition can be bound with the application or saved in a file and loaded at run time. The application can be written in most conventional programming languages, including C, FORTRAN, and Pascal.

Open Dialogue differs from Domain/Dialogue in many ways. The most significant difference, from a practical point of view, is that it is designed to run on machines other than Apollo workstations. It is currently layered on UNIX and the X Window System, but could, in theory, be moved to other platforms.

Like Domain/Dialogue, Open Dialogue has an object-oriented design. An interface is constructed out of "objects," such as menus and pop-ups. Each object has a "class," or type, that defines its behavior. An interface is defined by specifying a set of objects and their interrelationships. Open Dialogue is extensible through addition of new classes by application developers. Internal interfaces and abstract classes are made available to developers, allowing them to implement additional classes in C++, the system implementation language. These classes are fully integrated with the rest of the system, including all interface definition tools.

Domain/Dialogue requires that an interface be defined separately from an application before it is run. Open Dialogue supports this model but also allows additional interface components, or even the entire interface, to be defined at run time. Furthermore, since the object definition facilities are accessible to applications, developers can write then own interface definition tools.

Open Dialogue does not impose a specific style on interfaces developed with it. It does, however, allow consistent interfaces to be encouraged through definition of "templates." A template describes a portion of an interface, a grouping of one of more objects along with attributes describing their appearance and behavior. A template can be instantiated any number of times. Each instantiation can be customized by further specifying or overriding object attributes.

External or dialogue dominant control of an application is encouraged. Open Dialogue allows the application to be written as a subroutine library. It will acquire information for all input parameters from the end-user based on the interface definition. It will take results of the function call and present them to the end-user, also based on the interface definition. The application can, however, treat Open Dialogue as a subroutine package itself, making calls to inquire and set values and to request a stream of input events.

2. *Interface Management Concepts*

a. *Dialogue independence:* A dialogue developer creates the interface in a dialogue definition source file, which is compiled separately from the application.

b. *Structural model of interface:* Human–computer interaction is modeled by three general pieces: application objects, graphic objects, and data transformer objects (some of which simply contain data). Application objects provide a means for application callbacks and returns. Graphic objects provide end-user interaction and layout pieces of the interface. Data transformer objects transform data from one type to another (e.g., string to integer) and hold data.

c. *Representation of interface:* A textual dialogue definition language is compiled to create an interface definition. Work is under way on an interactive design tool that will initially allow all graphical aspects of the interface to be described; ultimately, it will allow all aspects of the human–computer interface to be described. The system allows developers to define their own interface definition tools.

d. *Interface development tools:* The dialogue is specified in a text file with a text editor. An interactive design tool is currently being developed for graphical creation and manipulation of the interface.

e. *Dialogue developer role:* The dialogue developer builds an interface using the Open Dialogue interactive tools. This developer must work in concert with the application developer to agree on the internal dialogue. Additionally, if new interface components are needed, the dialogue developer can create these primitives (objects).

f. *System development methodology:* No specific system development methodology is used, but use of Open Dialogue facilitates iterative design and rapid prototyping.

g. *Rapid prototyping:* Rapid prototyping is an explicit step in the use of Open Dialogue. A stub application is provided with which an interface can be viewed without writing any application code. All dialogue described entirely within the interface definition can be initiated and tested. Any interaction that triggers application intervention either through a callback or return will note that such an event was triggered.

h. *Control structure:* The encouraged control structure is external or dialogue dominant. Internal or computation dominant control is also supported, however.

3. *Features of System*

a. *Internal representation of interface definitions*

- *At implementation time:* Object workspace (a collection of interaction objects that can be stored in a file or loaded into memory).
- *At prototyping time:* Object workspace.
- *At run time:* Object workspace.

b. *Lexical constraints:* The nature of the current interaction primitives supplied with Open Dialogue is such that there are few lexical constraints that can be violated by the end-user. For example, character input can be validated either a character at a time, when the end-user finishes the input, or at some later time, depending on the interface definition. Some validation, such as that for numeric input, can be performed by the primitives (objects) provided with the system. Other validation can be done by the application or through primitives added to the system by the application developer.

c. *Input dialogue:* All inputs are validated for lexical accuracy by the dialogue. Validation can be done by the transformer objects provided with Open Dialogue, by the application, or by a new transformer created as an extension to Open Dialogue. Transformers take one type of data, such as string, as input and provide a different type of data, such as integer, as output.

d. *Output dialogue:* All output is described as objects, with the exception of output to graphics areas. Objects are dynamic in that anything that can be set up in advanced through a dialogue description file can also be done at run time.

e. *Relationship between input and output dialogue:* Most objects can both contain or display output as well as receive input data, although objects are primarily oriented one way or the other. Error messages, help, and prompts are all instances of standard interaction primitives.

f. *Help:* Help text can be associated with any object in the interface. A standard end-user action is defined for accessing help for individual pieces of the interface. Help text is displayed within a pop up.

g. *Pragmatics:* Currently pragmatics are encapsulated within the X Window System.

h. *Multiple input devices:* The current implementation waits concurrently on keyboard and mouse input; additional input devices can be used as supported by X. Support is planned to allow input from any number of input sources. This input would be processed in round robin, run-to-completion fashion.

i. *Support environment and graphics:* Open Dialogue is currently layered on UNIX and Version 11 of the X Window System. Open Dialogue has been ported to Apollo workstations, SUN workstations, and Micro VAX workstations. Future ports include IBM RT personal computers. Open Dialogue is developed to run on bit-mapped workstations running X, making extensive use of graphics both for displaying the interface and the application output. Icons, menus, and other display and interaction techniques are all graphically oriented.

4. Miscellaneous Questions

a. *Human factors built in:* No human factors principles are enforced by Open Dialogue. Human factors principles can, however, be enforced among applications by use of templates in creating the dialogue. A number of interfaces can use the same set of templates to create a consistent look and feel across those interfaces.

b. *Sequential versus asynchronous dialogue:* In general the application invokes an Open Dialogue routine to wait for input. End-user interaction proceeds in the interface until either a return to the application is requested by the dialogue or an application callback routine is triggered. After a return is triggered and the application has completed any processing, the application can return control to Open Dialogue by invoking the event wait routine. At the completion of a callback routine, control is returned to Open Dialogue as well. Open Dialogue cannot produce dialogue that is other than sequential at this time. Future work is planned to address this limitation.

c. *Generality of interaction style:* Interaction techniques provided with Open Dialogue support menu and forms-oriented interfaces. Extensions, however, could be implemented to handle any desired interaction style. These interaction styles could be mixed and matched as desired.

d. *Interface evaluation:* No mechanisms are built into Open Dialogue for dialogue evaluation. The fact that an interface can be brought up and used with no application code, however, encourages rapid prototyping and early evaluation of proposed interfaces.

5. Implementation

a. *Languages:* C++.

b. *Operating system:* UNIX bsd4.2.

c. *Date work begun:* 1986, with much of the design work leveraged from Domain/Dialogue started in 1984.

d. *Status:* Commercial product.

e. *Personnel:* Five computer engineers.

f. *Self-creating:* Theoretically, the Interactive Design Tool could be used to create itself; however, work has not progressed to that stage yet.

RAPID/USE

1. General Description of System

RAPID/USE [Wasserman 1985; Wasserman and Shewmake 1985], developed at the University of California at San Francisco by Anthony I. Wasserman and David Shewmake, is designed to provide automated support for the User Software Engineering (USE) methodology. The USE methodology advocates independent design of the end-user interface(s) to an interactive system, along with end-user participation in early stages of the development process, largely through the ability to use and evaluate prototypes of the end-user interface to the developing system. RAPID/USE executes a transition-diagram-based representation of an interactive system. In the transition diagrams, nodes represent messages to be displayed; arcs represent transitions, which may be caused by end-user input or other events; and small boxes represent actions associated with the application. The executable formalism of the state transition diagrams is a very powerful way to represent and execute interactive systems.

With no actions implemented or linked, RAPID/USE can be used simply to "execute" the transition diagrams and to provide an executable prototype of the end-user interface. Actions may be linked into the system incrementally, thereby making it possible to evolve the resulting program from a mockup of the interface to a complete system. Actions may be written in commonly used programming languages (C, Pascal, FORTRAN 77), or in the data manipulation language for the Troll/USE relational database management system.

The front-end to RAPID/USE is a graphical editor (Transition Diagram Editor) that generates RAPID/USE code. When using the TDE, the developer is given the impression of a two-dimensional programming language. During execution of the RAPID/USE program, it is possible to animate the transition diagrams as a way to trace execution. RAPID/USE also contains logging mechanisms that can be used to replay or evaluate a session.

RAPID/USE makes no assumptions about interface style and simply gives the dialogue developer access to low-level control over the alphanumeric display. Higher level programs, such as a direct manipulation forms editor, are then built on top of RAPID/USE; that is, they generate a RAPID/USE program. A product based on RAPID/USE is now commercially available from Interactive Development Environments, Inc.

2. Interface Management Concepts

a. *Dialogue independence:* Done through the use of a separate dialogue description file.
b. *Structural model of interface:* None.
c. *Representation of interface:* State transition diagram network is used, with variables and control mechanisms added to the basic transition network idea to extend it to describe human–computer interaction.
d. *Interface development tools:* These include a graphical Transition Diagram Editor (TDE), text editor for end-users without TDE, and form layout program for the specific case of database entry and retrieval.
e. *Dialogue developer role:* Not explicitly included, but RAPID/USE could be used with a method that supports a dialogue developer.
f. *System development methodology:* Explicitly supports the User Software Engineering (USE) methodology as an approach to system development.
g. *Rapid prototyping:* Dialogue representation, with transition diagrams or the RAPID/USE language, is directly executable. Prototyping is explicitly a step in the User Software Engineering methodology.
h. *Control structure:* Transition diagrams provide the control structure, which is inherently dialogue dominant.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Tables.
   - *At prototyping time:* Tables.
   - *At run time:* Tables.

b. *Lexical constraints:* Character or token handling is equally available.
c. *Input dialogue:* Dialogue representation allows checking of "types," such as numerical or character, plus range and length limits; other checks may be made through programmed actions.
d. *Output dialogue:* Dialogue representation includes variables, which may be passed to programmed actions; values may be communicated in both directions at run time.
e. *Relationship between input and output dialogue:* Output is associated with nodes and input is associated with transitions on arcs between nodes. Error messages, prompts, and help information are always treated as output where the preceding input has caused a transition to such a state (node).
f. *Help:* All dialogue is handled consistently; no special facilities are provided in the tool for help.
g. *Pragmatics:* Currently works only with a keyboard with possible time-outs. Current research focuses on handling direct manipulation.
h. *Multiple input devices:* Current research is modeling "loosely connected" dialogue processes represented as a set of transition diagrams.
i. *Support environment and graphics:* UNIX and SUN workstation or similar workstation (eventually). Graphics may be achieved through use of programmed actions that involve graphical routines but is not the focus of the current system. Current research is addressing highly interactive systems that include multiple windows and graphics.

4. Miscellaneous Questions

a. *Human factors built in:* No.
b. *Sequential versus asynchronous dialogue:* Sequential dialogue is possible; also an action can invoke a background process that can produce output, allowing some asynchronous interfaces.
c. *Generality of interaction style:* General alphanumeric display.
d. *Interface evaluation:* Logging in two forms: a raw keystroke file, and a transition log, with transition, input, output, action (if any), and time stamp for each state transition. An auxiliary tool, rapsum, summarizes the transition log. This tool provides information that can be used to evaluate interfaces built with the tool.

5. Implementation

a. *Languages:* C.
b. *Operating system:* UNIX.
c. *Date work begun:* 1979.
d. *Status:* Research and, recently, commercial product.
e. *Personnel:* Three computer scientists.
f. *Self-creating:* No.

Rapid Intelligent Prototyping Laboratory (RIPL)

1. General Description of System

The Rapid Intelligent Prototyping Laboratory (RIPL) [Flanagan et al. 1985], developed at Computer Technology Associates in Englewood, Colorado, is a hardware and software suite that supports prototypes that are realistic facades of complex computer systems. In an RIPL prototype, the external aspects of the interface appear realistically, but internal workings are an entirely different matter. The prototpye interface software is more complex than the eventual "real" system. This is necessary to allow measurement and evaluation of the interface. RIPL is an evolving product that eventually will support the entire system interface development life cycle. The life cycle as viewed by RIPL covers requirement definition, prototype generation, and building interfaces with standardized end-user interaction.

The initial RIPL does not include direct connection to a requirements definition capability or provide an ability to generate code; it supports the end-user application software development by providing screen and dialogue definitions in a structured form. RIPL initial interface design expertise is acquired from the guidelines compiled by Smith and Mosier [1986] and is

limited to the areas of display format, dialogue type selection, and physical input device selection.

RIPL has five major components. The "Executive" software set up all necessary file and library access and performs housekeeping functions of deleting, renaming, and backing up data. The "Prototype Build Subsystem" allows a dialogue developer to define and arrange end-user activity screen areas—referred to in RIPL as "tiles"—and to define everything necessary for prototyping. The "Simulation Subsystem" links definitions with end-user routines and libraries and then performs interface simulations. The "User Advisory Subsystem" consists of two expert systems, both operating from a consolidated knowledge base. The "Consultation Expert" provides general advice and guidance to the dialogue developer. The "Evaluation Expert" calculates design metrics and evaluates the prototype. The "Technical Librarian" software implements an electronic book metaphor for design guidelines and manuals. The initial RIPL is a single end-user workstation built on the Digital Equipment Corporation VAXstation.

2. Interface Management Concepts

a. *Dialogue independence:* Computation is treated as a response to dialogue stimuli.

b. *Structural model of interface:* Stimulus–response network. Stimuli are specified as Boolean combinations of end-user inputs (strings and picks) and system events (timers and tiles becoming active or inactive). Responses are changes to the tile set.

c. *Representation of interface:* Representation is interactive; a stimulus–response network is built with a direct manipulation interface. A developer can specify the tile interactions in a breadth or depth first manner. A "TBD" (To Be Defined) capability provides stubs for yet unspecified responses. The developer designates a stimulus by either doing it or describing it.

d. *Interface development tools:* Direct manipulation tools; manipulations are made to the real interface, not a description of the interface.

e. *Dialogue developer role:* Yes; it uses the direct manipulation interface and interacts with consultation and evaluation expert systems to analyze and redefine the dialogue.

f. *System development methodology:* At this point none is favored; the intention is for RIPL to support traditional development methodologies, not replace them. In the near future RIPL will be integrated with a Task Description Language (TDL) tool developed in-house. TDL is a formal grammar akin to PDL for software. TDL specifies system end-user tasks in a system, independent of whether or not the task is automated.

g. *Rapid prototyping:* Primarily a rapid prototyping system; future extensions will allow closer integration with system development methodologies and production of structures or code.

h. *Control structure:* Control structure is event based. Stimuli can be either dialogue oriented (e.g., end-user action) or system oriented (fixed duration timers or asynchronous zero duration timers started by application software).

3. Features of System

a. *Internal representation of interface definitions*
- *At implementation time:* Tables and end-user-defined code modules.
- *At prototyping time:* Tables and end-user-defined code modules.
- *At run time:* "To Be Defined" (see 2c above).

b. *Lexical constraints:* Keystrokes, strings, and pick locations are captured as stimuli and compared against internal tables to see which tiles are "interested." Limited ability to define such stimuli as any keystroke, a specific key, or any numeric keystroke is provided. There are no errors, as such, only stimuli that are not of interest to the prototype. These unused stimuli are captured for later analysis.

c. *Input dialogue:* Focus is on interaction mechanisms, and is less concerned in the initial version with the validity of inputs. Errors and error handling are

defined as stimuli and responses by the developer. Error-handling features such as ranges of valid input or list validation are not currently provided.

d. *Output dialogue:* Output displays are defined either a priori or generated by designer code within RIPL guidelines. A priori definitions provide templates with optional list items to use in the template positions or provide multiple instances of the same type tile.

e. *Relationship between input and output dialogue:* End-user inputs are stimuli; changes to tile attributes and contents are responses. Error messages or help output are irrelevant to the dialogue definition process; they are used to associate semantic meaning with tile areas for dialogue evaluation.

f. *Help:* Not a special case; basic tools are appropriate for developing help dialogues.

g. *Pragmatics:* An environment definition is used to represent details of target devices the prototype will run on. The evaluation expert takes this information into account, if specified. Dialogue itself is based on logical devices.

h. *Multiple input devices:* Supports multiple input devices and multiple display surfaces, and the environment can be orchestrated as one integrated prototype.

i. *Support environment and graphics:* Digital Equipment Corporation MicroVMS operating system; written in VAX Pascal using GKS for display output., Expert systems are implemented in MIT's NIL Common LISP. Handles icon-oriented direct manipulation as well as menus and queries.

4. Miscellaneous Questions

a. *Human factors built in:* It currently advises, using the consultation or the evaluation expert; it will eventually enforce environment specifications such as specific dialogue usages, error handling, and shape encoding.

b. *Sequential versus asynchronous dialogue:* There is no requirement that dialogue be strictly sequential; concurrent input–process–output is the approach. Three different processors make this possible: RIPL getting stimuli from an end-user; RIPL modifying tiles as a response; and an end-user-written routine either doing application processing or generating its own displays.

c. *Generality of interaction style:* Dialogues for direct manipulation, menus, queries, and form-driven interfaces can be created. Currently there is no hardware support for voice input/output, but it is theoretically possible.

d. *Interface evaluation:* Handled by three mechanisms:
   - Evaluation Expert System—Design metrics calculated from tile attributes and stimulus–response definitions.
   - Instruments—Start and stop timers between stimuli and responses and cursor trackers to monitor cursor movements.
   - Capture/Playback—Postsimulation evaluation done by replaying simulation from a capture file at different playback speeds.

5. Implementation

a. *Language:* VAX Pascal and MIT's NIL Command LISP.

b. *Operating system:* DEC MicroVMS with VAXstation user interface system.

c. *Date work begun:* October 1984.

d. *Status:* Internal currently; commercial in the future.

e. *Personnel:* Two computer scientists/software engineers; two programmers; one graphics specialist; one expert systems specialist; two human factors experts.

f. *Self-creating:* Yes; RIPL can prototype itself.

SmethersBarnes Prototyper

1. General Description of System

Prototyper, developed and marketed by SmethersBarnes in Portland, Oregon, is a tool for rapid design, prototyping, and testing of interfaces specifically for Macintosh

applications [Prototyper 1987]. Code generation adds the capability to create high-level (Pascal) code and Macintosh resource data structures, allowing stand-alone execution of interface prototypes. Since generated stand-alone prototypes contain a skeletal event-oriented program structure, they can be easily augmented to support additional human factors testing of logic or even to form the base of a final application.

Through Prototyper's dynamic creation of Macintosh data structures and its use of the Macintosh Toolbox, applications produced using Prototyper are highly compliant with accepted Macintosh interface standards. Prototyper's operational metaphors exploit widely used object-oriented drawing concepts; combined with a strong focus on graphic representation, this enables Prototyper to be accessible to professional software engineers and end-users alike.

The intention with Prototyper is to ease the task of learning the intricacies of application development on the Macintosh, to provide a tool that complements the developer's existing tools and skills, and to provide a smooth, intuitive tool for communication among all involved persons, whatever their contribution to the software development process.

Prototyper focuses on menus and windows, with a menu editor and a window editor that include a palette of standard interface objects. Immediate simulation of the menu or window currently being constructed is always available, as is a global simulation of the entire interface. Prototyper capitalizes on the reusability of Macintosh resource objects and can import such objects from other applications, saving redundant work. Design may proceed without regard to computational complexities, allowing nonprogrammers to express their ideas without technical expertise. Generated code is highly commented and logically structured, lending itself to extension.

2. Interface Management Concepts

a. *Dialogue independence:* Yes. The majority of interface objects are implemented as Macintosh resources, yielding instrinsic separation. Pascal units created for each window/dialogue/alert will call computational logic as necessary and are therefore easily isolated.

b. *Structural model of interface:* None.

c. *Representation of interface:* Graphical, object-oriented, direct manipulation language produced using interactive tools.

d. *Interface development tools:* Menu editor with concurrent simulator, window/dialogue/alert worksheets with palette of tools represented iconically. The dialogue developer creates interface objects by selecting a tool, clicking and dragging with mouse. A rapid context switching facility allows simulation of current window.

e. *Dialogue developer role:* Role is supported by direct manipulation tools; also the roles of software engineer, graphic designer, human factors specialist, test subject, evaluator, analyst, managerial staff, end-user, and student are supported.

f. *System development methodology:* No specific methodology, but Prototyper assists specification, design, implementation, testing, and maintenance phases of software life cycle.

g. *Rapid prototyping:* The strong suit of Prototyper, specific to the Macintosh environment. No technical knowledge is necessary to construct prototypes; participation can be solicited from all project members and clients.

h. *Control structure:* The event-driven architecture of Macintosh applications is followed. Menu initialization and handling of run-time inputs are isolated. Pascal code is generated from the interface design.

3. Features of System

a. *Internal representation of interface definitions*

- *At implementation time:* Internal state tables and Macintosh resources.
- *At prototyping time:* Internal state tables and Macintosh resources.
- *At run time:* Executable code and Macintosh resource objects.

b. *Lexical constraints:* Macintosh buttons, icons, pictures, check boxes, radio buttons, and static and editable text objects are supported; external logical behavior must be code in computational routines. Generated code handles graphical and linking support of interface only.
c. *Input dialogue:* Input validation is performed by computational components of the prototype generated using Prototyper.
d. *Output dialogue:* Presentation of output within application windows is the responsibility of the application programmer. Strong Macintosh conventions govern presentation and end-user interaction with output. Prototyper supports alerts, used for outputting error messages and end-user warnings, and window scroll bars allow end-user control of displayed text.
e. *Relationship between input and output dialogue:* No distinction is made in windows, modal dialogues, or modeless dialogues. Alerts are generally output features.
f. *Help:* Yes, there are tools that aid in design and implementation of on-line help for the end-user.
g. *Pragmatics:* Prototyper assumes bitmapped terminal, mouse, and customizable interface characteristics of the Macintosh environment.
h. *Multiple input devices:* Event-driven Macintosh architecture handles inputs from serial ports, data storage devices, keyboard, mouse, and so on.
i. *Support environment and graphics:* Currently limited to Macintosh operating system, with plans to expand to other graphical microcomputer environments. Graphics are central to the Macintosh environment, and Prototyper develops standard Macintosh interfaces.

4. Miscellaneous Questions

a. *Human factors built in:* Prototyper produces interfaces that embody those principles intrinsically in Macintosh architecture, specifically found in Apple Computer Inc.'s "Human Interface Guidelines." It stresses, for example, nonmodality, avoidance of sequentiality, and pull-down menus.
b. *Sequential versus asynchronous dialogue:* Asynchronous dialogue such as that found in Macintosh application interfaces is supported.
c. *Generality of interaction style:* Prototyper produces interfaces that are oriented to Macintosh-specific interaction styles and conventions.
d. *Interface evaluation:* No.

5. Implementation

a. *Languages:* Pascal.
b. *Operations system:* Macintosh.
c. *Date work begun:* 1986.
d. *Status:* Commercial product.
e. *Personnel:* Three computer scientists.
f. *Self-creating:* Yes, new versions are being designed with the existing product.

State Diagram Specification Interpreter

1. General Description of System

Research by Robert J. K. Jacob at the Naval Research Lab began as an attempt to develop and test a representation/specification technique for describing interactive end-user interfaces [Jacob 1983, 1985]. The technique is based on state transition diagrams (STDs) with a set of special features and extensions—a set kept intentionally small in order to retain the principal benefit of state diagram notation, which is its conceptual simplicity. In the course of testing and refining the specification language, it was necessary to build an interpreter that implements the behavior given in a representation. Motivation for the interpreter was to test and improve the representation/specification language, not the other way around.

State diagram specifications are executed by an interpreter to provide a working prototype of the specified system. The technique supports a decomposition of the end-user interface description into semantic, syntactic, and lexical components. A specific notation suitable for describing each

level to the interpreter in a separate document is then provided. A process of stepwise refinement of the syntactic representation from an informal representation to a formal, executable one within the same notation is also supported.

In addition to producing a token, a transition in a state diagram may call a semantic action, a condition, or a nonterminal. A nonterminal is defined in a separate diagram, called a subroutine. Syntactic level diagrams also introduce output tokens to describe output syntax analogously to the description of input syntax in terms of input tokens. Since syntax is concerned with the names and sequences of input tokens, it is extended to include a description of the names and sequences of output tokens in the same state diagram notation. The concept of a token for output is, by analogy to input, a unit whose internal structure has no meaning with respect to this dialogue.

The research has produced a technique for representing end-user interfaces based on state diagrams; a design notation that separates the end-user interface design and specification itself into the semantic, syntactic, and lexical levels; and an interpreter that accepts such a specification, implements, and executes it. Several fairly large systems have been built using this method and interpreter. Current research focuses on extending the technique to describe direct manipulation or coroutine-based interfaces.

2. Interface Management Concepts

a. *Dialogue independence:* Separate representation or code for semantics and syntax.
b. *Structural model of interface:* None.
c. *Representation of interface:* State transition diagrams.
d. *Interface development tools:* A graphical STD editor allows "on the fly" changes to diagrams as they are executed. Programs exist to draw STDs from text descriptions, pretty print text descriptions, and parse and translate (to LISP) the text form.
e. *Dialogue developer role:* Writes the STD; does not write the semantic routines, which are given to a programmer.
f. *System development methodology:* No specific software engineering methodology is used, but semantic/syntactic/lexical levels are separated throughout the design process, specification, and implementation. Stepwise refinement of representation/specification from early form to final form is supported, and early steps can be executed just as final steps.
g. *Rapid prototyping:* A prototype runs directly from the specification. Missing diagrams and actions can be stubbed automatically to obtain a prototype from an incomplete early version of the specification.
h. *Control structure:* Dialogue dominant, calling semantic actions like subroutines.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* (UNIX) text file describing STDs.
   - *At prototyping time:* (UNIX) text file describing STDs.
   - *At run time:* Internal tables containing parsed version of text file data.
b. *Lexical constraints:* Syntactic or lexical constraints are handled directly in the STD. Semantic constraints (e.g., name completion with respect to only those files that are readable by some end-user) are handled by calling a semantic action subroutine to check and return to a Boolean value, which is then used in traversing the STD.
c. *Input dialogue:* Format and timing of input events are controlled by the STD; contents are stored for use by semantic actions. Some input validation is done in the STD and some in the semantic subroutines.
d. *Output dialogue:* Output token contents are set by semantic actions, but format and timing of the token are controlled by the STD.

e. *Relationship between input and output dialogue:* All input or output consists of tokens. Input tokens and output tokens are treated as nearly symmetrically as possible; the two are distinguished by a naming convention in the specification.
f. *Help:* Not specifically; a dialogue developer can design help features as part of the STD, as has been done in all the systems built using this technique.
g. *Pragmatics:* Hidden entirely from dialogue representation, except at the lexical level, where they can be programmed as needed. Currently there is no specific help for programming the lexical level.
h. *Multiple input devices:* Yes, in principle. Transitions leading from a state may involve inputs from different streams, and whichever occurs first causes its transition. The current version of the lexical analyzer does not handle this; it can be changed.
i. *Support environment and graphics:* UNIX, written in C. Also an experimental version in Franz LISP on UNIX, and coming soon in Symbolics LISP on Symbolics. All dialogues, including graphical ones, are centered around sequences of tokens. Tokens could, for example, put colored boxes in positions on a graphic display instead of text on the terminal. The technique has been used to produce a graphical interface.

4. Miscellaneous Questions

a. *Human factors built in:* No.
b. *Sequential versus asynchronous dialogue:* Current research is focused on extending the STD approach to cover concurrent, asynchronous dialogues more completely.
c. *Generality of interaction style:* Based on sequences of input and output tokens; tokens internally can do anything.
d. *Interface evaluation:* This is supported only to the extent that putting the end-user interface into STD notation clarifies its behavior and helps one apply performance models. Any rapid prototyping tool is useful for doing empirical evaluations of end-user interface designs.

5. Implementation

a. *Languages:* C.
b. *Operating system:* UNIX.
c. *Date work begun:* 1981.
d. *Status:* Experimental.
e. *Personnel:* One computer scientist.
f. *Self-creating:* Probably.

Toolkit UIMS

1. General Description of System

The UIMS—formerly called TIGER—developed at Boeing Computer Services [Kasik 1982] by David J. Kasik, Henry W. Ramsey, and J. Randy Houser is part of a larger toolkit for the development of highly interactive graphics-based applications [Kasik 1985]. The toolkit is intended to isolate applications from operating systems, computing hardware, graphics hardware, and database management systems.

The toolkit UIMS strictly separates dialogue components from the application by formatting all dialogue sequences for display, managing all end-user defaults within and across sessions, accepting all end-user inputs, and handling operating system exceptions. Its goal is to keep the interactive syntactic aspects of an application consistent for both end-users and application programmers and thus improve the productivity of each.

The toolkit UIMS incorporates extensive end-user productivity aids. Default tracking is a compromise that combines ease-of-use characteristics of a menu-based system with the speed of parameter omission available in a command-based system. By preserving defaults for every dialogue sequence, the end-user must only change a limited amount of information while still seeing all legal options. An end-user can invoke other functions (e.g., view manipulation) without losing information already entered in another function. Another mode allows free traversal (i.e., no explicit reject sequence is needed to quit) when the end-user wishes to quit in midfunction. Illogical

or illegal choices can be disabled automatically to help prevent the end-user from making errors.

Physical interaction with the system is consistent across all applications and can present information in a number of ways. Two-dimensional windows present an overlapped text window containing large amounts of alphanumeric information. Panels give a "graphical forms mode" normally reserved for strictly alphanumeric terminals. A command macro language is provided for all applications to extend their functionality by combination. Pseudoconstruction allows an end-user to build data temporarily to aid in complex construction tasks.

Programmer productivity aids are provided for both the dialogue representation and run-time stages of application development. A programming language called ET (Extended TICCL) has been desgined as an extension of Pascal. ET contains new declaration and control structures that allow a programmer to construct a seemingly linear dialogue sequence. The ET compiler produces a Pascal procedure representing a state machine that is traversed by the run-time UIMS interpreter. The interpreter takes on the burden of dialogue formatting, default management (including heuristics to look ahead in the dialogue sequence), syntax checks on alphanumeric entry, pick queuing and feedback, and interactive device control. In this way, much of the bookkeeping associated with a complex interactive application is removed from the domain of application programming.

The toolkit UIMS is currently being used in a wide variety of applications, including three-dimensional geometry construction and manipulation for points, curves, surfaces, and volumes; finite element modeling; drafting and documentation; hierarchical design charts; space station analysis; oil well log history analysis; and interactive panel design. Overall experience with the toolkit UIMS as an interactive application development approach has been excellent in terms of quality of dialogue, amount of interactive application functions that can be effectively produced, extensibility, and portability.

2. Interface Management Concepts

a. *Dialogue independence:* A dialogue programmer represents dialogue in an independent language that is precompiled and traversed by a run-time interpreter.
b. *Structural model of interface:* None.
c. *Representation of interface:* Dialogue is characterized as a hierarchy with free traversal. A dialogue programmer uses a dialogue programming language that extends Pascal specification and declarative structures while keeping Pascal control structures.
d. *Interface development tools:* System text editor is used to produce dialogue programming language code.
e. *Dialogue developer rule:* No, dialogue is programmed.
f. *System development methodology:* Functional decomposition via an internally developed method called Prime/Common hierarchies. PCMAN is an application written with the toolkit UIMS for constructing other applications. Dialogue is written before application code is written.
g. *Rapid prototyping:* Dialogue can be exercised with application code stubbed out.
h. *Control structure:* Strictly adheres to dialogue dominant or external architecture.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Executable code representing state tables.
   - *At prototyping time:* Executable code representing state tables.
   - *At run time:* Executable code representing state tables.
b. *Lexical constraints:* The system validates end-user keystrokes under application-specified constraints, prevents erroneous picking, queues input when requested, and provides multiple feedback styles.

c. *Input dialogue:* Input validation is programmed.
d. *Output dialogue:* Output dialogue is programmed.
e. *Relationship between input and output dialogue:* Output dialogue is controlled by the application program. Services are provided for application error messages and help displays.
f. *Help:* Help is keyed to dialogue fragments and can be accessed at any time.
g. *Pragmatics:* Operates with logical devices.
h. *Multiple input devices:* All devices can be active simultaneously. The toolkit UIMS uses an interrupt driver and reads the input to determine the proper default path without application intervention.
i. *Support environment and graphics:* Operating system independent, but runs on VM/CMS, MS/DOS, and UNIX. Current application dialogue is textual. A forms mode provides convenient formatting for complex text mode entities.

4. Miscellaneous Questions

a. *Human factors built in:* No.
b. *Sequential versus asynchronous dialogue:* Current implementation allows only sequential dialogues; future research will include capabilities for asynchronous dialogue.
c. *Generality of interaction style:* Primarily menu based; an alternate command language interface (transparent to the application) is available.
d. *Interface evaluation:* Keystroke capture is possible.

5. Implementation

a. *Languages:* Pascal.
b. *Operating system:* IBM VM/CMS, MS/DOS, UNIX System V.
c. *Date work begun:* 1980.
d. *Status:* Internal product.
e. *Personnel:* Two computer scientists in design/implementation of UIMS; ten others in use of toolkit UIMS for application development.
f. *Self-creating:* No.

University of Alberta UIMS

1. General Description of System

The University of Alberta UIMS [Green 1985] is an experimental UIMS with three goals:

- To evaluate the Seeheim model of human–computer interfaces.
- To provide a test bed for new ideas in human–computer interfaces and UIMS.
- To provide a useful tool for development of human–computer interfaces.

The Seeheim model divides a human–computer interface into three main components: the presentation component, the dialogue control component, and the application interface model. The "presentation component" can be viewed as the lexical level of the interface, responsible for device level interactions. The "dialogue control component" manages dialogue between the human and the computer system. The "application interface model" forms the interface between the human–computer interface and the other parts of the application. The University of Alberta UIMS provides a collection of tools that can be used to design these three components. These tools can be used to describe screen layout, device assignments, dialogue structure, and interaction with the application program. The result of the design part of the UIMS is a detailed representation of the human–computer interface.

One of the main design goals is to give the human–computer dialogue developer as much freedom as possible. One way in which this has been done is to provide multiple tools for each of the three components. If the developer does not like one of the tools, the developer can switch to another of the tools. The UIMS has a number of well-defined ways in which the developer can modify it in order to fit the developer's personal design style and can easily add new interaction and display techniques from existing libraries. These new

techniques have the same status as system-supplied ones. The developer can also save commonly used parts of dialogues in a library. Another important feature of this UIMS is a concentration on interactive graphical techniques and direct manipulation in the design tools to increase productivity of the developers.

2. Interface Management Concepts

a. *Dialogue independence:* The human–computer interface and other parts of the application are viewed as separate processes, although they need not be implemented this way. As much as possible, the interface can be designed independently of other parts of the program.
b. *Structural model of interface:* None.
c. *Representation of interface:* Dialogue representation is based on events and event handlers. Events are similar to messages and can be generated by the end-user, the application program, or other event handlers. Event handlers are processes capable of processing events. There can be many concurrently executing event handlers, and the set of event handlers can change over time. Most common dialogue notations, such as transition networks, RTNs, ATNs, and grammars can be translated into event handlers.
d. *Interface development tools:* Two interface development tools exist. One is a high-level programming language based on event handlers. The other is a graphical recursive transition network (RTN) editor. Both tools produce a common representation for the dialogue. New interface development tools can easily be added to the UIMS as long as they produce this common representation as their output.
e. *Dialogue developer role:* Supported by interface development tools.
f. *System development methodology:* None; UIMS is an implementation tool.
g. *Rapid prototyping:* Most parts of the human–computer interface can be tried as they are designed; a completed application is not required to test the interface.
h. *Control structure:* An event-based control structure is under control of the dialogue developer and application programmer.

3. Features of System

a. *Internal representation of interface definitions*
   - *At implementation time:* Database and executable code.
   - *At prototyping time:* Database and executable code.
   - *At run time:* Database and executable code.
b. *Lexical constraints:* Handled by individual interaction techniques in a manner appropriate for that technique.
c. *Input dialogue:* Depending upon the type of validation, it is performed by the interaction technique, in the dialogue control component, or just before it is sent to the application. The application (computation) does not need to validate input.
d. *Output dialogue:* Most output from the UIMS is generated by display techniques, which are implemented as procedures in the underlying programming language. When information is displayed, the display technique extracts relevant data from the information that has been passed to the human–computer interface.
e. *Relationship between input and output dialogue:* There is essentially no distinction between input and output dialogue. Other parts of the application are viewed as an input device by the human–computer interface. Thus, all interface development tools can be used to interpret messages from the application. All prompts, help information, and most error messages are handled inside the human–computer interface; they are not the concern of the application.
f. *Help:* No special facilities for help exist; a dialogue developer can provide help through presentation and dialogue control component tools.

g. *Pragmatics:* Handled somewhat through the presentation component.
h. *Multiple input devices:* UIMS is based on the concept of concurrent processes; therefore, there can be any number of active dialogues or devices. The underlying run-time system is responsible for process switching and ensuring that each device and dialogue gets its share of processor time.
i. *Support environment and graphics:* Runs under the UNIX operating system but is relatively independent. All input and output is through a device-independent window manager. Most human–computer interfaces produced by this UIMS make extensive use of graphics. Its main application areas are computer-aided design and computer animation.

4. Miscellaneous Questions

a. *Human factors built in:* Tools do not enforce any human factors principles; human factors of the design are left to the dialogue developer.
b. *Sequential versus asynchronous dialogue:* Dialogue structure is completely under control of the dialogue developer. Any dialogue that can be programmed can, in theory, be implemented using the dialogue development tools.
c. *Generality of interaction style:* Can support a wide range of interaction styles; tools can be customized to generate interfaces with any particular interaction style. Most of this adaptation is accomplished through libraries of interaction and display techniques.
d. *Interface evalaution:* No.

5. Implementation

a. *Languages:* C.
b. *Operating system:* UNIX.
c. *Date work begun:* April 1984.
d. *Status:* Experimental.
e. *Personnel:* Five computer scientists.
f. *Self-creating:* Yes.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions to the DMS research mentioned in this survey by other members of our research team, especially our friend, colleague, and fellow traveler, Dr. Roger W. Ehrich. We also express appreciation to Dr. Marilyn Mantei for her careful reading of portions of the manuscript, her many suggestions for improvement, and her inspired Indian cookery. We wish to acknowledge the patience and assistance of Dr. Tony Wasserman, during his tenure as editor, whose knowledge of the field and numerous constructive suggestions helped us organize this paper. Thanks to Eric Smith and Antonio Siochi for providing thoughtful comments on some sections and to all who completed our lengthy questionnaire for the appendix. Thanks to Caroline Danby Woody for her cheerful typing of uncountably infinite versions of our scribbles. Pat Cooper, Sheila Casey, and Carole Shepherd also helped with typing along the way. And Jo-Anne Lee Bogner had the enviable task of typing the final version. Somehow we all lived through it!

Our DMS work is funded by the Software Productivity Consortium, and the Virginia Center for Innovative Technology. It has also been supported by the National Science Foundation, Dr. H. E. Bamford, Jr., Program Director, NSF Information Technology Program; as well as by IBM Federal Systems Division. Earlier support was given by the Office of Naval Research, Engineering Psychology Group.

REFERENCES

ACM CHI '83 Conference on Human Factors in Computing Systems. 1983. (Boston, Mass., Dec.). ACM, New York.

ACM CHI '85 Conference on Human Factors in Computing Systems. 1985. (San Francisco, Calif., Apr.). ACM, New York.

ACM CHI '86 Conference on Human Factors in Computing Systems. 1986. (Boston, Mass., Apr.). ACM, New York.

ACM CHI '87 + GI Conference on Human Factors in Computing Systems. 1987. (Toronto, Ontario, Canada, Apr.). ACM, New York.

ACM CHI '88 Conference on Human Factors in Computing Systems. 1988. (Washington, D.C., May). ACM, New York.

ACM Computing Surveys 1981. Special Issue: The Psychology of Human–Computer Interaction, vol. *13*, 1 (Mar.).

ACM SIGGRAPH Workshop on Software Tools for User Interface Management. 1986. (Seattle, Wash., Nov.). ACM, New York. In *Computer Graphics 21,* 2 (Apr.), 71–147.

ACM SIGGRAPH Symposium on User Interface Software. 1988. (Banff, Alberta, Canada, Oct.). ACM, New York.

ACM SIGSOC Conference on Easier and More Productive Use of Computing Systems. 1986. (Ann Arbor, Mich., May). ACM, New York.

ALAVI, M. 1984. An assessment of the prototyping approach to information systems development. *Commun. ACM 27,* 6 (June), 556–563.

APPLE COMPUTER, INC. 1985. *Inside Macintosh,* vol. II. Addison-Wesley, Reading, Mass.

BASS, L. J., AND BUNKER, R. E. 1981. A generalized user interface for applications programs. *Commun. ACM 24,* 12 (Dec.), 796–800.

BENBASAT, I., AND WAND, I. 1984. A structured approach to designing human–computer dialogues. *Int. J. Man-Mach. Stud. 21,* 105–126.

BENNETT, J. 1984. Managing to meet usability requirements. In *Visual Display Terminals: Usability Issues and Health Concerns,* J. Bennett, D. Case, J. Sandelin, and M. Smith, Eds., Prentice-Hall, Englewood Cliffs, N.J.

BLACK, J. L. 1977. A general purpose dialogue processor. In *Proceedings of the National Computer Conference.* ACM, New York, pp. 397–408.

BLESER, T. P. 1981. A formal language for describing and evaluating the ergonomics of user–computer interfaces. *ACM DC Chapter 20th Symposium* (College Park, Md., June). ACM, New York.

BLESER, T. P., AND FOLEY, J. D. 1982. Towards specifying and evaluating the human factors of user–computer interfaces. In *Proceedings of the Conference on Human Factors in Computer Systems* (Gaithersburg, Md., Mar.). ACM, New York, pp. 309–314.

BÖCKER, H.-D., FISCHER, G., AND NIEPER, H. 1986. The enhancement of understanding through visual representations. In *Proceedings of the ACM CHI'86 Conference on Human Factors in Computing Systems* (Boston, Mass., Apr.). ACM, New York, pp. 44–50.

BOEHM, B. W. 1983. Seven basic principles of software engineering. *J. Syst. Softw. 3,* 3–24.

BOEHM, B. W., GRAY, T. E., AND SEEWALDT, T. 1984. Prototyping vs. specification: A multi-project experiment. In *Proceedings of the 7th International Conference on Software Engineering.* ACM, IEEE, New York, pp. 473–484.

BORUFKA, H. G., AND PFAFF, G. 1981. The design of a general-purpose command interpreter for a graphical man–machine communication. In *Man-Machine Communication in CAD/CAM,* T. Sata and E. Warman, Eds. North-Holland Publ., Amsterdam.

BORUFKA, H. G., TEN HAGEN, P. J. W., KUHLMANN, H. W., AND WEBER, H. R. 1981. On defining interactions by dialogue cells. Tech. Rep. GRIS 81-7, FG Graphische Interaktive Systeme Technische Hochschule Darmstadt.

BORUFKA, H. G., KUHLMANN, H. W., AND TEN HAGEN, P. J. W. 1982. Dialogue cells: A method for defining interactions. *IEEE Comput. Graph. Appl.* (July), 25–33.

Bricklin's Demo Program. 1987. Software Garden, Inc., P.O. Box 373, Newton Highlands, Mass. 02161.

BRITTS, S. 1987. Dialog management in interactive systems: A comparative survey. *ACM SIGCHI Bull. 18,* 3 (Jan.), 30–42.

BROWN, M. D. 1985. *Understanding PHIGS.* Megatek Corp., San Diego, Calif.

BROWNE, D. P., SHARRATT, B. D., AND NORMAN, M. A. 1986. The formal specification of adaptive user interfaces using command language grammar. In *Proceedings of the ACM CHI'86 Conference on Human Factors in Computing Systems* (Boston, Mass., Apr.). ACM, New York, pp. 256–260.

BUXTON, W. A. 1983. Lexical- and pragmatic considerations of input structures. *Comput. Graph. 17,* 1, 31–37.

BUXTON, W. A., LAMB, M. R., SHERMAN, D., AND SMITH, K. C. 1983. Towards a comprehensive user interface management system. *Comput. Graph. 17,* 3, 35–42.

CANNING, R. G., ED. 1983. Replacing old applications. *EDP Anal. 21,* 3 (Mar.), 1–16.

CARD, S. K., MORAN, T. P., AND NEWELL, A. 1983. *The Psychology of Human–Computer Interaction.* Lawrence Erlbaum Assoc., Hillsdale, N.J.

CAREY, T. T., AND MASON, R. E. A. 1983. Information systems prototyping: Techniques, tools and methodology. *Can. J. Oper. Res. Inf. Process.*

CARROLL, J. M., AND ROSSON, M. B. 1985. Usability specifications as a tool in interactive development. In *Advances in Human–Computer Interaction,* vol. 1. H. Rex Hartson, Ed. Ablex, Norwood, N.J., pp. 1–28.

CASEY, B. E., AND DASARATHY, B. 1982. Modelling and validating the man–machine interface. *Softw. Pract. Exper. 12,* 557–569.

CHAPANIS, A. 1982. Man-computer research at Johns Hopkins. In *Information Technology and Psychology: Prospects for the Future.* Praeger, New York.

CHERITON, D. R. 1976. Man–machine interface design for timesharing systems. In *Proceedings of the ACM Annual Conference.* ACM, New York, pp. 362–380.

CLARK, I. A. 1981. Software simulation as a tool for usable product design. *IBM Syst. J. 20,* 3, 272–293.

Communcations of the ACM 1983. Special Issue: Working Toward Successful Human–Computer Interface 26, 4 (Apr.).

CONWAY, M. E. 1963. Design of a separable transition-diagram compiler. *Commun. ACM 6,* 7.

COOMBS, M. J., AND ALTY, J. L., EDS. 1981. *Computing Skills and the User Interface.* Academic Press, Orlando, Fla.

COUTAZ, J. 1985. Abstractions for user interface design. *IEEE Computer, 18* (Sept.), 21–34.

COX, B. J. 1986. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Mass.

Data General's PRESENT Information Presentation Facility User's Manual. 1982. Data General Corporation Document 083-000168 (Apr.).

DEC's VAX11 Form Management System. 1984. Digital Equipment Corporation Document SPD AE-R440C-TE (Jan.).

DENERT, E. 1977. Specifications and design of dialogue systems with state diagrams. In *Interactive Computing Symposium.* North-Holland Publ., Amsterdam.

DIEDERICH, J., AND MILTON, J. 1987. Experimental prototyping in Smalltalk. *IEEE Software 20* (May), 50–64.

DRAPER, S. W., AND NORMAN, D. A. 1985. Software engineering for user interfaces. *IEEE Trans. Softw. Eng. SE-11,* 252–258.

DWYER, B. 1981. A user-friendly algorithm. *Commun. ACM 24,* 9 (Sept.), 556–561.

EDMONDS, E. A. 1981. Adaptive man–computer interfaces. In *Computing Skills and the User Interface,* M. J. Coombs and J. L. Alty, Eds. Academic Press, London.

EDMONDS, E. A. 1982. The man–computer interface: A note on concepts and design. *Int. J. Man-Mach. Stud. 16,* 231–236.

EHRICH, R. W., AND HARTSON, H. R. 1981. DMS—An environment for dialogue management. In *Proceedings of COMPCON81* (Washington, D.C., Sept.). IEEE, New York, p. 121.

EHRICH, R. W., AND WILLIGES, R. C., EDS. 1986. *Designing Human–Computer Dialogues.* Elsevier, Amsterdam.

EHRLICH, K. 1985. Factors influencing technology transfer. *SIGCHI Bull. 17,* 2, 20–25.

EVANS, R., FIDDIAN, N. J., AND GRAY, W. A. 1981. Adaptable user interfaces for portable, interactive computing software systems. *SIGSOC Bull. 13,* 2–3 (Jan.), 59.

FELDMAN, M. B. 1981. Tools to facilitate human-factors improvement in interactive information display systems. In *Proceedings of COMPCON81* (Washington, D.C., Sept.). IEEE, New York, pp. 117–118.

FELDMAN, M. B., AND ROGERS, G. T. 1982. Toward the design and development of style-independent interactive systems. In *Proceedings of the Conference on Human Factors in Computer Systems* (Gaithersburg, Md., Mar.). ACM, New York, pp. 111–116.

FISCHER, G. 1982. Symbiotic, knowledge-based computer support systems. In *Proceedings of the IFAC Conference on Analysis, Design, and Evaluation of Man–Machine Systems* (Baden-Baden, Germany, Sept.), pp. 351–358.

FISCHER, G. 1987. An object-oriented construction and tool kit for human–computer communication. *Comput. Graph.* SIGGRAPH Workshop on Software Tools for User Interface Development. ACM, New York.

FLANAGAN, D., LENOROVITZ, D., STANKE, E., AND STOCKER, F. 1985. RIPL Concept of Operations and System Architecture. CTA Internal Document (May), Boulder, Colo.

FLECCHIA, M. A., AND BERGERON, R. D. 1987. Specifying complex dialogs in ALGAE. In *Proceedings of the ACM CHI + GI'87 Conference.* (Toronto, Ontario, Canada, Apr.). ACM, New York, pp. 229–234.

FOLEY, J. D. 1980. The structure of interactive command languages. In *Proceedings of the IFIP Workshop on the Methodology of Interaction.* North-Holland Publ., Amsterdam, pp. 227–234.

FOLEY, J. D. 1981. Tools for the designers of user interfaces. Rep. GWU-IIST-81-07, George Washington University Institute for Information Science and Technology, Washington, D.C. (Mar.).

FOLEY, J. D., AND VAN DAM, A. 1982. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass.

FOLEY, J. D., AND WALLACE, V. L. 1974. The art of natural graphic man–machine conversation. In *Proc. IEEE 63,* 4, 462–471.

FOLEY, J. D., GIBBS, C., KIM, W. C., AND KOVACEVIC, S. 1988. A knowledge-based user interface management system. In *Proceedings of the ACM CHI'88 Conference on Human Factors in Computing Systems* (Washington, D.C., May). ACM, New York, pp. 67–72.

FREEMAN, P. A. 1980. A perspective on requirements analysis and specification. In *Tutorial on Software Design Techniques,* P. A. Freeman and A. I. Wasserman, Eds.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

GOMAA, H., AND SCOTT, D. B. 1981. Prototyping as a tool in the specifications of user requirements. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, Calif., Mar.). ACM/IEEE, New York.

GOOD, M. D., WHITESIDE, J. A., WIXON, D. R., AND JONES, S. J. 1984. Building a user-derived interface. *Commun. ACM 27,* 10, 1032–1043.

GOODMAN, D. 1987. *The Complete HyperCard Handbook.* Bantam Books, Toronto.

GRANOR, T. E., AND BADLER, N. I. 1986. GUIDE: Graphical User Interface Development Environment. In *Proceedings of Trends and Applications* (Silver Spring, Md.). IEEE Computer Society and National Bureau of Standards, Gaithersburg, Md., pp. 37–41.

GRAPHICAL INPUT INTERACTION TECHNIQUES WORKSHOP SUMMARY 1983. *Comput. Graph. 17,* 1 (Jan.), 5–30.

GRAY, P. D., AND KILGOUR, A. C. 1985. GUIDE: A UNIX-based dialogue design system. Departmental Research Rep. CSC/85/R8, Department of Computing Science, University of Glasgow, Lilybank Gardens, Glasgow, Scotland.

GREEN, M. 1981. A methodology for the specification of graphical user interfaces. *Comput. Graph. 15,* 3.

GREEN, M. 1985. The University of Alberta user interface management system. In *Proceedings of SIGGRAPH '85, 12th Annual Conference* (San Francisco, Calif., July 22–26). ACM, New York, pp. 205–213.

GREEN, M. 1986. A survey of three dialog models. *ACM Trans. Graph. 5,* 3(Jul.), 244–275.

GUEDJ, R. A., AND TUCKER, H. A., EDS. 1979. *Methodology in Computer Graphics: Seillac I.* (Seillac, France). North-Holland Publ., Amsterdam.

GUEDJ, R. A., TEN HAGEN, P. J. W., HOPGOOD, F. R. A., TUCKER, H. A., AND DUCE, D. A., EDS. 1980. *Methodology of Interaction: Seillac II* (Seillac, France)., Amsterdam.

GUEST, S. P. 1982. The use of software tools for dialogue design. *Int. J. Man–Mach. Stud. 16,* 263–285.

HANAU, P. R., AND LENOROVITZ, D. R. 1980a. A prototyping and simulation approach to interactive computer system design. In *Proceedings of the 17th ACM Conference on Design Automation* (Minneapolis, Minn., June 23–25). ACM, New York.

HANAU, P. R., AND LENOROVITZ, D. R. 1980b. Prototyping and simulation tools for user/computer dialogue design. In *Proceedings of ACM SIGGRAPH '80, 7th Annual Conference on Computer Graphics and Interactive Techniques* (Seattle, Wash., July 14–18). ACM, New York.

HANSEN, W. J. 1971. User engineering principles for interactive systems. In *Proceedings of the AFIPS Conference,* vol. 39. AFIPS Press, Reston, Va., pp. 523–532.

HARTSON, H. R. 1969. Digital control simulation system. In *Proceedings of the 6th Annual SHARE-ACM-IEEE Design Automation Workshop* (Miami Beach, Fla., June). ACM/IEEE, New York, pp. 113–128.

HARTSON, H. R., ED. 1985. *Advances in Human–Computer Interaction.* vol. 1. Ablex, Norwood, N.J.

HARTSON, H. R. 1989. Control and communication in user interface management. *IEEE Softw.* 6 (Jan.), 62–70.

HARTSON, H. R., AND HIX, D., EDS. 1988. *Advances in Human–Computer Interaction,* vol. 2. Ablex, Norwood, N.J.

HARTSON, H. R., AND HIX, D. 1988b. Toward empirically derived methodologies and tools for human–computer interface development. *Int. J. Man-Mach. Stud.* To be published.

HARTSON, H. R., AND SMITH, E. C. 1989. *Rapid Prototyping.* To be published.

HARTSON, H. R., (JOHNSON) HIX, D., AND EHRICH, R. W. 1984. A human–computer dialogue management system. In *Proceedings of INTERACT '84, First IFIP Conference on Human–Computer Interaction* (London, Sept.). International Federation for Information Processing, pp. 57–61.

HAYES, P. J. 1985. Executable interface definitions using form-based interface abstractions. In *Advances in Human–Computer Interaction,* vol. 1. H. Rex Hartson, Ed. Ablex, Norwood, N.J., pp. 161–190.

HAYES, P. J., AND REDDY, R. 1983. Steps toward graceful interaction in spoken and written man–machine communication. *Int. J. Man–Mach. Stud. 19,* 231–384.

HAYES, P. J., AND SZEKELY, P. A. 1983. Graceful interaction through the COUSIN command interface. *Int. J. Man–Mach. Stud. 19,* 295–306.

HAYES, P. J., BALL, E., AND REDDY, R. 1981. Breaking the man–machine communication barrier. *IEEE Comput. 14* (Mar.), 19–30.

HAYES, P. J., SZEKELY, P. A., AND LERNER, R. A. 1985. Design alternatives for user interface management systems based on experience with COUSIN. In *Proceedings of the ACM CHI'85 Conference on Human Factors in Computing Systems* (San Francisco, Calif., Apr.). ACM, New York, pp. 169–175.

HAYS, G. G. 1969. Computer-aided design: Simulation of digital design logic. *IEEE Trans. Comput.* (Jan.), 1–10.

HCI Hawaii, First International Conference on Human–Computer Interaction. 1984. (Honolulu, Hawaii, Aug.). International Commission on Human Aspects in Computing.

HCI Hawaii, Second International Conference on Human–Computer Interaction. 1987. (Honolulu, Hawaii, Aug.). International Commission on Human Aspects in Computing.

HELANDER, G. A. 1981. Improving system usability for business professionals. *IBM Syst. J. 20,* 3, 294–305.

HENDERSON, D. A., JR. 1986. The Trillium user interface design environment. In *Proceedings of the ACM CHI'86 Conference on Human Factors In Computing Systems* (Boston, Mass., Apr.). ACM, New York, pp. 221–227.

HILL, R. D. 1987. Event-response systems: A technique for specifying multi-threaded dialogues. In *Proceedings of the ACM CHI+GI'87 Conference* (Toronto, Ontario, Canada, Apr.). ACM, New York, pp. 241–248.

HIX, D., AND HARTSON, H. R. 1986. An interactive environment for dialogue development: Its design, use, and evaluation. In *Proceedings of the ACM CHI'86 Conference on Human Factors in Computing Systems* (Boston, Mass., Apr.). ACM, New York, pp. 228–234.

HIX, D., AND HARTSON, H. R. 1987. A structural model for hierarchically describing human–computer dialogue. In *Proceedings of INTERACT '87, Second IFIP Conference on Human–Computer Interaction* (Stuttgart, West Germany, Sept.). International Federation for Information Processing, pp. 695–700.

HOFFMAN, H.-J. 1985. Research work in design methodology for interactive programs. Tech. Rep. PU1R2/86, Department of Computer Science. Technische Hochschule Darmstadt, Darmstadt, West Germany.

HOSIER, J., ED. 1978. Structured analysis and design. *Infotech State of the Art Report,* pp. 195–208.

HUTCHINS, E. L., HOLLAN, J. D., AND NORMAN, D. A. 1986. Direct manipulation interfaces. In *User Centered System Design,* D. A. Norman and S. W. Draper, Eds. Lawrence Erlbaum Assoc., Hillsdale, N.J.

IBM Development Management System for CMS: Guide and reference 1983. IBM Document SC24-5198-1, White Plains, N.Y. (Dec.).

IBM Systems Journal. 1981. Special Issue: Human Factors *20,* 2.

IBM System Productivity Facility for MVS, General Information. 1983. IBM Document GC34-2039-0, White Plains, N.Y. (Aug.).

IBM System Productivity Facility, Dialog Management Guide, VM/SP. 1985. IBM Document SC 34-4009-0, White Plains, N.Y. (Sept.).

IBM VM/SP System Product Interpreter reference. 1983. IBM Document SC24-5239 (Sept.).

IEEE Computer. 1982. Special Issue: Human–Computer Interaction *15,* 11 (Nov.).

IEEE Computer. 1983. Special Issue: The DoD STARS Program *16,* 11(Nov.).

IEEE Computer Graphics. 1979. Special Report on the Graphics Standards Planning Committee *13,* 3.

IEEE Computer Graphics. 1984. Special Issue: Graphics Kernel System 7 (Feb.).

IEEE Software. 1989. Special Issue: Developing Human–Computer Interfaces—Software of a Different Sort *6* (Jan.).

INTERACT '84. 1984. *First IFIP Conference on Human–Computer Interaction* (London, Sept.). International Federation for Information Processing.

INTERACT '87. 1987. *Second IFIP Conference on Human–Computer Interaction* (Stuttgart, Sept.). International Federation for Information Processing.

JACKSON, M. A. 1975. *Principles of Program Design.* Academic Press, Orlando, Fla.

JACKSON, M. A. 1983. *System Development.* Prentice-Hall International, Englewood Cliffs, N.J.

JACOB, R. J. K. 1983. Using formal specifications in the design of the human–computer interface. *Commun. ACM 26,* 4 (Apr.), 259–264.

JACOB, R. J. K. 1985. An executable specification technique for describing human–computer interaction. In *Advances in Human–Computer Interaction,* vol. 1. H. Rex Hartson, Ed. Ablex, Norwood, N.J., pp. 211–244.

JOHNSON (HIX), D. 1985. The structure and development of human–computer interfaces. Ph.D. dissertation, Dept. of Computer Science, Virginia Polytechnic Institute and State Univ., Blacksburg, Va.

KAISER, P., AND STETINA, I. 1982. A dialogue generator. *Softw. Pract. Exper. 12,* 693–707.

KAMRAN, A., AND FELDMAN, M. B. 1983. Graphics programming independent of interaction techniques and styles. *Comput. Graph. 17,* 1 (Jan.), 58–66.

KASIK, D. J. 1982. A user interface management system. *Comput. Graph. 16,* 3, 99–106.

KASIK, D. J. 1985. An architecture for graphics application development. In *Proceedings of IEEE International Conference on Robotics and Automation* (Mar.). IEEE, New York, pp. 365–371.

KENNEDY, T. C. S. 1974. The design of interactive procedures for man–machine communication. *Int. J. Man–Mach. Stud. 6,* 309–334.

KIERAS, D., AND POLSON, P. G. 1983. A generalized transition network representation for interactive systems. In *Proceedings of the ACM CHI '83 Conference on Human Factors in Computing Systems* (Boston, Mass., Dec.). ACM, New York, pp. 103–106.

KIERAS, D., AND POLSON, P. G. 1985. An approach to the formal analysis of user complexity. *Int. J. Man–Mach. Stud. 22,* 365–394.

LANTZ, K. A., TANNER, P. P., BINDING, C., HUANG, K. T., AND DWELLY, A. 1987. Reference models, window systems, and concurrency. *Comput. Graph. 21,* 2 (Apr.), 87–97.

LENOROVITZ, D. R., AND RAMSEY, H. R. 1977. A dialogue simulation tool for use in the design of interactive computer systems. In *Proceedings of the Human Factors Society* (Santa Monica, Calif.). Human Factors Society, pp. 95–99.

LINEBARGER, R. N., AND BRENNAN, R. D. 1964. A survey of digital simulation. *Simulation* (Dec.).

MACDONALD, A. 1982. Visual programming. *Datamation* (Oct.), 132–140.

MANTEI, M. 1986. Techniques for incorporating human factors in the software lifecycle. In *Proceedings of Structured Techniques Association 3rd Annual Conference* (Chicago, Ill.), pp. 177–203.

MARTIN, J. 1973. *Design of Man–Computer Dialogues.* Prentice-Hall, Englewood Cliffs, N.J.

MASON, R. E. A., AND CAREY, T. T. 1981. Productivity experiences with a scenario tool. In *Proceedings of the IEEE COMPCON* (Washington, D.C., Sept.). IEEE, New York, pp. 106–111.

MASON, R. E. A., AND CAREY, T. T. 1983. Prototyping interactive information systems. *Commun. ACM 26,* 5 (May), 347–352.

MAURER, M. E. 1983. Full-screen testing of interactive applications. *IBM Syst. J. 22,* 3, 246–261.

MEADS, J. A. 1987. The standards pipeline. *Comput. Graph. 21,* 3 (Jun.), 235–237.

MILLER, L. A., AND THOMAS, J. C., JR. 1977. Behavioral issues in the use of interactive systems. *Int. J. Man–Mach. Stud. 9,* 509–536.

MILLS, C. C., AND WASSERMAN, A. I. 1984. A transition diagram editor. In *Proceedings of the 1984 Summer Usenix Conference* (Salt Lake City, Utah). ACM, New York.

MORAN, T. P. 1981. The command language grammar: A representation for the user interface of interactive computer systems. *Int. J. Man–Mach. Stud. 15,* 3–51.

MORAN, T. P., ED. 1984. *Human–Computer Interaction, A Journal of Theoretical, Empirical, and Methodological Issues of User Psychology and System Design.* Lawrence Erlbaum Assoc., Hillsdale, N.J.

MYERS, B. A. 1986. Visual programming, programming by example, and program visualization: A taxonomy. In *Proceedings of the ACM CHI'86 Conference on Human Factors in Computing Systems* (Boston, Mass., Apr.). ACM, New York, pp. 59–66.

MYERS, B. A. 1987. Creating dynamic interaction techniques by demonstration. In *Proceedings of the ACM CHI+CI'87 Conference* (Toronto, Ontario, Canada, Apr.). ACM, New York, pp. 271–278.

MYERS, G. J. 1975. *Reliable Software Through Composite Design.* Mason/Charter Publ.

MYERS, G. J. 1978. *Composite/Structured Design.* Litton Education Publishing Inc.

National Bureau of Standards Conference on Human Factors in Computer Systems. 1982. (Gaithersburg, Md., Mar.).

National Research Council Workshop on Software Human Factors. 1983. National Academy of Sciences (Washington, D.C., May).

NAUR, P., ED. 1963. Revised report on the algorithmic language ALGOL 60. *Commun. ACM 6* (Jan.).

NEWMAN, W. M. 1968. A system for interactive graphical programming. In *Proceedings of the AFIPS Spring Joint Computer Conference.* Thompson Books, Washington, D.C.

NEWMAN, W. M., AND SPROULL, R. F. 1979. *Principles of Interactive Computer Graphics,* 2nd ed. McGraw-Hill, New York.

NORMAN, D. A. 1984. Four stages of user activities. In *Proceedings of INTERACT '84, First IFIP Conference on Human–Computer Interaction* (London, Aug.) International Federation for Information Processing.

NORMAN, D. A., AND DRAPER, S. W. 1986. *User-Centered System Design.* Lawrence Erlbaum Assoc., Hillsdale, N.J.

OLSEN, D. R., JR. 1983. Automatic generation of interactive systems. *Comput. Graph. 17,* 1 (Jan.), 53–57.

OLSEN, D. R., JR. 1984a. User's manual for MIKE-2.0. Arizona State University Tech. Rep.

OLSEN, D. R., JR. 1984b. Pushdown automata for user interface management. *ACM Trans. Graph. 3,* 3, 177–203.

OLSEN, D. R., JR., AND DEMPSEY, E. P. 1983. SYNGRAPH: A graphical user interface generator. *Comput. Graph. 17,* 3, 43–50.

OLSEN, D. R., JR., BUXTON, W., EHRICH, R. W., KASIK, D. J., RHYNE, J. R., AND SIBERT, J. 1984. A context for user interface management. *IEEE Comput.* (Dec.), 33–42.

OLSEN, D. R., JR., DEMPSEY, E. P., AND ROGGE, R. 1985. Input/output linkage in a user interface management system. *Comput. Graph. 19,* 3 (July), 191–197.

OVERMYER, S. P., AND CAMPBELL, E. E., JR. 1984. Rapid prototyping: An approach to human–computer interface design. In *Proceedings of the 28th Annual Meeting of the Human Factors Society* (San Antonio, Tex.). Human Factors Society.

PARNAS, D. 1969. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the ACM National Conference.* ACM, New York, pp. 379–385.

PERRY, T. S. 1988. 'PostScript' prints anything: A case history. *IEEE Spectrum* (May), 42–46.

PFAFF, G., ED. 1985. *User Interface Management Systems.* Springer-Verlag, Berlin.

PORCELLA, M., FREEMAN, P., AND WASSERMAN, A. I. 1983. Ada methodology questionnaire summary. *Softw. Eng. Notes 8,* 1, 51–98.

PRECISION VISUALS, INC. 1987. *A Guide to Designing Friendly User/Computer Interfaces.* 6260 Lookout Road, Boulder, Colo. 80301.

PROTOTYPER. 1987. *SmethersBarnes Prototyper User's Manual.* P.O. Box 639, Portland, Ore. 97207.

PUGLIA, V., PETZOLD, C., STONE, M. D., DUNTEMANN, J., AND CHISHOLM, P. 1986. Operating in a new environment. *PC Magazine* (Feb.), 109–132.

PUK, R. 1986. Enhancements to PHIGS input model. X3H3.1/86-48 (Nov. 10).

REISNER, P. 1981. Formal grammar and human factors design of an interactive graphics system. *IEEE Trans. Softw. Eng. SE-7,* 2 (Mar.), 229–240.

REISNER, P. 1982. Further developments toward using formal grammar as a design tool. In *Proceedings of the Conference on Human Factors in Computer Systems* (Gaithersburg, Md., Mar.). ACM, New York, pp. 309–314.

REISNER, P. 1983a. Formal grammar as a tool for analyzing ease of use: Some fundamental concepts. *Human Factors in Computer Systems.* Ablex, Norwood, N.J.

REISNER, P. 1983b. Analytic tools for human factors of software. IBM Research Laboratory Rep. RJ 3803 (43605), San Jose, Calif.

ROACH, J., AND NICKSON, M. 1983. Formal specifications for modeling and developing human/computer interfaces. In *Proceedings of the ACM CHI '83 Conference on Human Factors in Computing*

Systems (Boston, Mass., Dec.). ACM, New York, pp. 35–39.

ROACH, J., HARTSON, H. R., EHRICH, R. W., YUNTEN, T., AND JOHNSON, D. HIX. 1982. DMS: A comprehensive system for managing human–computer dialogue. In *Proceedings of the Conference on Human Factors in Computer Systems* (Gaithersburg, Md., Mar.). ACM, New York, pp. 102–105.

ROSENBURG, V. 1974. The scientific premises of information science. *J. Am. Soc. Inf. Sci.* (July–Aug.), 263–269.

ROSENTHAL, D., AND YEN, A. 1983. User interface models summary. *Comput. Graph. 17,* 3 (Jan.), 16–20.

ROSS, D. T., AND SCHOMAN, K. E. 1977. Structured analysis for requirements definition. *IEEE Trans. Softw. Eng. SE-3,* 1 (Jan.).

ROWE, L. A., AND SHOENS, K. A. 1983. Programming language constructs for screen definition. *IEEE Trans. Softw. Eng. SE-9,* 1 (Jan.), 31–40.

RUBEL, A. 1982. Graphic based applications—Tools to fill the software gap. *Digit. Des. 3* (July), 17–30.

SCHEIFLER, R. W., AND GETTYS, J. 1986. The X window system. *ACM Trans. Graph. 5,* 3 (Apr.), 79–109.

SCHMUCKER, K. 1986. Mac App: An application framework. *BYTE 11,* 8, 189–193.

SCHULERT, A. J., ROGERS, G. T., AND HAMILTON, J. A. 1985. ADM—A dialogue manager. In *Proceedings of the ACM CHI '85 Conference on Human Factors in Computing Systems* (San Francisco, Calif., Apr.). ACM, New York, pp. 177–183.

SENKO, M. E., ALTMAN, E. B., ASTRAHAM, M. M., FEHDER, P. L., AND WANG, C. P. 1972. A data independent architectural model: Four levels of description from logical structures to physical structures. Rep. RJ982, IBM Corporation, Research Division, San Jose, Calif. (Feb.).

SHNEIDERMAN, B. 1980. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop Publ., Cambridge, Mass.

SHNEIDERMAN, B. 1982. Multi-party grammars and related features for designing interactive systems. *IEEE Trans. Syst. Man Cybern. 12,* 2 (Mar.–Apr.), 148–154.

SHNEIDERMAN, B. 1983. Direct manipulation: A step beyond programming languages. *IEEE Computer* (Aug.), 57–69.

SHNEIDERMAN, B. 1987. *Designing the User Interface: Strategies for Effective Human–Computer Interaction* Addison-Wesley, Reading, Mass.

SIBERT, J. L., HURLEY, W. D., AND BLESER, T. W. 1988. Design and implementation of an object-oriented user interface management system. In *Advances in Human–Computer Interaction.* vol. II, H. Rex Hartson and D. Hix, Eds. Ablex, Norwood, N.J.

SIME, M. E., AND COOMBS, M. J., EDS. 1983. *Designing for Human–Computer Communication.* Academic Press, Orlando, Fla.

SKYLIGHTS. 1987. Skylights Systems, Inc., Medford, Mass.

SMITH, D. C., IRBY, C., KIMBALL, R., VERPLANK, B., AND HARSLEM, E. 1982. Designing the Star user interface. *BYTE 7,* 4 (Apr.), 242–282.

SMITH, H. T., AND GREEN, T. R. G. 1980. *Human Interaction with Computers.* Academic Press, London.

SMITH, S. L., AND MOSIER, J. N. 1986. Guidelines for designing user interface software. Tech. Rep. ESD-TR-86-278. Hanscom Air Force Base, Mass.: USAF Electronic Systems Division (NTIS No. AD A177198).

STEVENS, W. P. 1981. *Using Structured Design.* Wiley, New York.

STEVENS, W. P., MYERS, G. J., AND CONSTANTINE, L. L. 1974. Structured design. *IBM Syst. J. 13.*

TANNER, P. P., AND BUXTON, W. A. S. 1984. Some issues in future user interface management system (UIMS) development. In *Seeheim Workshop of User Interface Management Systems.* Eurographics-Springer.

TANNER, P. P., MACKAY, S. A., STEWART, D. A., AND WEIN, M. 1986. A multitasking switchboard approach to user interface management. In *Proceedings of the ACM SIGGRAPH '86 Conference in Computer Graphics, 20,* 4 (Aug.). ACM, New York, pp. 241–248.

TESLER, L. 1981. The Smalltalk environment. *BYTE 6,* 8 (Aug.), 90–147.

TOFFLER, A. 1980. *The Third Wave.* Bantam Books, New York.

UNICAD, INC. 1985. *Unicad User Interface Manual.* 1695 38th Street, Boulder, Colo. 80301.

VAN DAM, A., SKLAR, D., MICHENER, J., AND FOLEY, J. 1987. PHIGS public reviews—input model. X3H3/87-69 (Jan. 18).

WASSERMAN, A. I. 1973. The design of idiot-proof interactive systems. In *Proceedings of the National Computer Conference* (Montvale, N.J.). ACM, New York.

WASSERMAN, A. I. 1980. Information system design methodology. *J. Am. Soc. Inf. Sci.* (Jan.), 5–24.

WASSERMAN, A. I. 1981. User software engineering and the design of interactive systems. In *Proceedings of the 5th International Conference of Software Engineering.* ACM/IEEE, New York. pp. 387–393.

WASSERMAN, A. I. 1982. The user software engineering methodology: An overview. In *Information System Design Methodologies,* A. A. Verrijn-Stuart, Ed. North-Holland Publ., Amsterdam, pp. 591–628.

WASSERMAN, A. I. 1985. Extending transition diagrams for the specification of human–computer interaction. *IEEE Trans. Softw. Eng. SE-11,* 8 (Aug.).

WASSERMAN, A. I., AND SHEWMAKE, D. T. 1982. Rapid prototyping of interactive information systems. *ACM SIGSOFT Softw. Eng. Notes* (Dec.), pp. 1–18.

WASSERMAN, A. I., AND SHEWMAKE, D. T. 1985. The role of prototypes in the user software engineering methodology. In *Advances in Human–Computer Interaction,* vol. 1, H. Rex Hartson, Ed. Ablex, Norwood, N.J., pp. 191–210.

WASSERMAN, A. I., AND STINSON, S. K. 1979. A specification method for interactive information systems. In *Proceedings of the IEEE Conference on Specification of Reliable Software* (Cambridge, Mass.). IEEE, New York, pp. 68–79.

WASSERMAN, A. I., PIRCHER, P. A., SHEWMAKE, D. T., AND KERSTEN, M. L. 1986. Developing interactive information systems with the user software engineering methodology. *IEEE Trans. Softw. Eng. SE-12,* 2 (Feb.), 326–345.

WEINBERG, V. 1980. *Structured Analysis.* Prentice-Hall, Englewood Cliffs, N.J.

WILLIAMS, G. 1983. The Lisa computer system. *BYTE* (Feb.), 33–50.

WILLIAMS, G. 1984. The Apple Macintosh computer. *BYTE 9,* 2 (Feb.), 30–54.

WILLIGES, B. H., AND WILLIGES, R. C. 1981. User considerations in computer-based information systems. Tech. Rep. CSIE-81-2, VPI&SU Departments of Computer Science and Industrial Engineering (Sept.).

WILLIGES, R. C. 1984. Evaluating human–computer software interfaces. In *Proceedings of the 1984 International Conference on Occupational Ergonomics* (Toronto, Canada, May).

WIXON, D., WHITESIDE, J., GOOD, M., AND JONES, S. 1983. Building a user-defined interface. In *Proceedings of the ACM CHI '83 Conference on Human Factors in Computing Systems* (Boston, Mass., Dec.). ACM, New York, pp. 24–27.

WONG, P. C. S., AND REID, E. R. 1982. FLAIR—User interface dialog design tool. *Comput. Graph. 16,* 3 (July), 87–98.

WOODS, W. A. 1970. Transition network grammars for natural language analysis. *Commun. ACM 13,* 591–606.

WRIGHT, P. R., AND BROWN, B. W. 1978. A processor for providing friendly environments for frequently used application packages. In *Proceedings of the ACM Annual Conference* (Washington, D.C., Dec.), vol. 1. ACM, New York, pp. 346–350.

YOURDON, E., AND CONSTANTINE, L. L. 1979. *Structured Design.* Prentice-Hall, Englewood Cliffs, N.J.

YUNTEN, T., AND HARTSON, H. R. 1985. A SUPERvisory Methodology and Notation (SUPERMAN) for human–computer system development. In *Advances in Human–Computer Interaction.* vol. 1, H. Rex Hartson, Ed. Ablex, Norwood, N.J., 243–281.

ZELKOWITZ, M. V., ED. 1982. *ACM SIGSOFT Workshop on Rapid Prototyping ACM* (Columbia, Md. Apr.). ACM, New York.

Received October 1982; final revision accepted January 1988.