To appear in:
Proceedings of the Second Symposium
on Empirical Foundations of Information and Software Sciences,
Atlanta, Ga., 1984.

Designing a Human-Computer Interface with Software Specification Techniques

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

ABSTRACT

The process of designing a user interface for an interactive computer system is greatly facilitated if the designer can describe such user interfaces clearly and precisely. While formal specification techniques have been used in software engineering to describe many aspects of software systems, they have not been widely applied to the problem of specifying user interfaces. A formal specification of a user interface permits the designer to describe and study a variety of interfaces before building one. Human performance models derived from empirical data can also be applied to a specification to predict user performance with the proposed user interface before it is built. Finally, if an executable specification language is used, it is possible to build a prototype or mockup of the user interface directly from the specification. The mockup can then be used to gather experimental data about the proposed user interface early in the design process.

Specification techniques that can be used to describe human-computer interfaces are examined and divided into two categories, according to their underlying formal models: state transition diagrams and BNF. While the two can be made formally equivalent, specifications based on state transition diagrams are preferred because they more clearly show the sequence of the dialogue as the user sees it. Further, it is important to separate the description of the semantics of the system, so that it does not confound the description of other aspects of the user interface. Low-level details, such as choice of function keys or arrangement of prompts on a display, should also be separated from the higher-level design of the dialogue. These issues are addressed by dividing the user interface specification into three levels: semantic, syntactic, and lexical. An executable notation based on state transition diagrams that supports separate development and specification of the three levels of the user interface is introduced, and its use in prototype systems is described.

Introduction

It is difficult to design anything in the absence of a technique that permits one to describe it without actually building it. However, user interfaces for computer systems are often "designed" in just this way. While formal specification techniques have been used in software engineering to describe many aspects of software systems, they have not been widely applied to the problem of specifying user interfaces. The process of designing a user interface for an interactive computer

system would be considerably facilitated if the designer could describe proposed user interfaces clearly and precisely. A formal specification technique for user interfaces permits the designer to describe and study a variety of interfaces before building one. Human performance models derived from empirical data can then be applied to the specification to predict user performance with the proposed user interface before it is built.

The user interface increasingly being recognized as a critical aspect of software systems. While the principal problem in building computer systems was once providing sufficient processing power, it is now more often providing a good user interface–permitting the user to communicate what he (or she) wants the computer to do and to receive information from the computer in ways that are easy to assimilate. This is particularly important since an increasing number of computer users are not trained programmers but rather specialists in some other field; such users are less tolerant of and less able to compensate for a poor user interface than are computer specialists. The specification of the user interface is also important because, for a user, that description is identical to the description of the system itself. Experiments suggest that many users cannot distinguish the difference between changing the user interface to a system and changing the functionality of the system itself [17]. For many of the people who must understand a software system, the user interface specification *is* the system specification.

Specifying a User Interface

A specification of a user interface should describe the external (user-visible) behavior of the user interface of a system precisely. Ideally, it should also be easier to understand and take less effort to produce (and, if possible, be shorter) than the software that implements the interface. Someone who wants to answer a question about the behavior of a user interface should find it easier to do so by reading the specification than the code itself.

Given a specification of a user interface, it is desirable to attempt to evaluate the specified interface without building it. This can be done by applying human performance models to predict user performance (e.g., accuracy, speed) with the specified design. To facilitate such evaluation, the specification should describe the *cognitive* or information-processing steps the user

performs, rather than, for example, the physical actions. Although the former are often ignored in favor of the latter, it is the cognitive, not physical, actions that principally determine user performance and accuracy for a system [20]. Hence they are what specifications should capture in order to be useful for comparing two user interfaces.

In addition, the overall structure of the specification should represent the cognitive structure of the user interface (the mental constructs the user needs to operate it). The specification should describe the constructs a user will keep in mind when learning about the system–the basic outline around which the user's mental model of the system will be built. Put another way, the specification should yield a table of contents for a user manual but not necessarily the material in the manual. The goal is to make the structure of the specification and the structure of the user's manual as similar as possible. Singer [23] described his attempt to adhere to this goal: "In general, whenever I found a feature or concept difficult to explain clearly [in the specification language], I took this as a signal that the design itself was likely at fault. Whole versions of the editing requests were rejected because I could discover no simple way of explaining them."

Two simple applications of this approach are found in common practice. One is the use of BNF to provide structure to a user's manual. Many command language manuals (e.g., IBM's CMS manual) [13] give a single section for all the rules that define some nonterminal symbol, like *command line*. Many of those rules use other nonterminals, for example, *file specification*. *File specification* and other frequently-used nonterminals, in turn, are described in another section, and so on. Another example is the use of state diagram notation, where a user manual may describe the states of the system (usually only for a few major states). The user is told that the system will be in, for example, the *monitor command state* at some point. If he invokes an editor, it will be in the *editor state*, and a different set of meanings will be attached to his actions. Users are often explicitly told of such states by means of distinctive prompt characters. So, in two simple ways, the types of constructs often used in writing specifications of a user interface for designers have also been used to describe those user interfaces to users.

Finally, an important use of a specification of a user interface is to obtain a prototype or

mockup of the user interface. If an executable specification language is used, such a prototype can be obtained directly from the specification. The mockup permits the designer to conduct experimental studies of the proposed user interface and obtain empirical data to guide his work early in the design process.

Specification Techniques

Most previous work on formal and semi-formal language specifications has been devoted to the specification of *static* languages, as opposed to *interactive* languages [23]. In a static language, an entire text in the input language is (conceptually) present before any processing begins or any outputs are produced. All of the outputs are then produced together, usually after a fairly long input text has been processed, as is the case with the processing of the text of a computer program. This contrasts with interactive languages, in which the system produces responses at various points during the input of a text, resulting in a dialogue. Such input can be described as one long text, in which the computer may take actions and produce outputs at various points during the input, or as a series of brief texts, but where the processing of each input generally depends on previous processing. The user may also make use of intermediate outputs in formulating his subsequent inputs. He may also sometimes interrupt the system and begin entering a new statement in the input language. A specification of an interactive language must, therefore, capture more information about the behavior of the system than one for a static language: it must specify each of the system's responses and at precisely what point in a dialogue each will occur.

Specification Techniques for Static Languages

Two principal classes of techniques have been used for specifying static languages. One is based on production rule grammars and yields a specification in a notation such as Backus-Naur Form (BNF). The language is described by a set of production rules, from which all possible correct inputs in that language can be produced. Each rule gives the definition for some *nonterminal* symbol. Wherever that symbol itself appears (in the definition of some other symbol), it may be replaced by substituting the contents of its definition. In this way, the definition of a sin-

gle starting symbol ultimately yields all legal strings in the language.

The other class of techniques is based on finite state automata and results in a state transition diagram (or, equivalently but more amenable to computer processing, a list of transitions between the states of such a diagram). The specification consists of a set of nodes (states) and links between them (state transitions). Each transition is associated with a token in the input language. From any state, the next input language token received causes the transition labeled with that token to occur.

Specification Techniques for Interactive Languages

Both of these types of techniques must be modified to be used for interactive (rather than static) languages. The modifications are similar to those made to these techniques to create compilers for static languages [1, 5].

The modification of BNF consists of associating an action with each grammar rule. Whenever that rule applies to the input language stream (received thus far), the associated action occurs. Such actions can include prompts and other feedback to the user and also actions that involve processing by the rest of the system.

State transition diagrams are modified in a similar way. Each transition is associated with an action, which could provide user feedback or perform other processing, as with the modified BNF. Whenever a transition occurs, the system performs the associated action.

In examining the techniques that have been proposed or used to specify user interfaces for interactive systems, one finds nearly all of them are based on one of these two formalisms–BNF or state diagrams [15]. While the two are usually made formally equivalent by extensions, their surface differences have an important effect on the comprehensibility of the specifications. Since the concept of time sequence is explicit in state diagram notations (while it is implicit in BNF-based notations), the former are more suited to describing *when* events occur. Existing techniques based on state diagrams vary considerably in their syntax and expressive ability, although it is possible to combine the desirable features of several such notations.

One of the principal virtues of state diagram notations is that they make explicit precisely what the user can do at each point in a dialogue and what its effect will be (by giving the transition rules for each state). Feyock [9] makes good use of this property by using a computer-readable representation of the state diagram specification of a system as the input to a user help facility for that system. Based on the state diagram and the current state, the system can answer such questions from users as "What can I do next?" "Where am I?" and "How can I do ...?"

An important feature is the ability of one diagram to call upon another, much as a program can make a procedure call. This is analogous to using nonterminal symbols in BNF–invented intermediate constructs that permit the specification to be divided into manageable pieces. With the introduction of nonterminals, instead of labeling a transition with a single token to be recognized, it can be labeled with the name of a nonterminal symbol. That symbol is, in turn, described by a separate state transition diagram. An important criterion for a user interface specification is that its principal constructs–the main nonterminal symbols and states–represent concepts that will be meaningful to users and helpful to them in constructing their own mental models of the system. Indeed, some of the states can correspond to users' own notions of what a system does ("text entry" state, "logged-out" state). In this way, a mapping can be maintained from the user interface specification to the user documentation.

Other investigators have also found the state transition model useful in describing a user's mental model of an interactive computer system [6, 18], and some have also built specification interpreters [8, 25]. The choice of a state diagram-based notation is also supported by the empirical observation of Guest [12], who was surprised to find that programmers preferred a state transition diagram-based front end for a specification interpreteter to a BNF-based one.

Methods for specifying user interfaces have suffered because they lacked a satisfactory language for describing the "semantics" of the interface, i.e., the actions associated with the BNF rules or state transitions that the system performs in response to the user's commands. Since a complete description of such actions is in fact a specification of the entire system, putting it in the user interface specification clutters that specification with detail that belongs at another level.

What is needed is a high-level abstraction or model that describes the operations that the system performs [15]. Then, the user interface specification need only describe the user interface in terms of that model, while the internal details of the model would be described in a separate specification (for which conventional software specification techniques would be suitable). The particular model used depends on the application; an example is shown in this paper.

A synthesis of the features of several state diagram-based notations is thus used here to specify interactive user interfaces in conjunction with an appropriate semantic specification. An interpreter has been developed to take this user interface specification and execute it directly [14].

Managing Complexity in the Design Process

To reduce the complexity of the designer's task, the process of designing a user interface is divided into three levels. A specific notation suitable for each level is then provided. Foley and Wallace [10] introduced the notion of describing an interactive user interface at the *semantic, syntactic,* and *lexical* levels and defined them further in [11]. This model is followed here, but an attempt is made to delineate the three levels more precisely, particularly with respect to output, and to provide a specific notation for specifying each of them separately to an interpreter [16].

The *semantic* level describes the functions performed by the system. This corresponds to a description of the functional requirements of the system, but does not address how the user will invoke the functions. Next, the *syntactic* level describes the sequences of inputs and outputs necessary to invoke the functions described. Finally, the *lexical* level determines how the inputs and outputs are actually formed from primitive hardware operations.

Case History

This approach and specification notation will be demonstrated by examining the design and specification of an interactive graphics system, which was built directly from the executable specification of its the user interface. This system provides an interactive graphics editor for a specialized type of grid chart, similar to one used in a military application. It provides an automated aid for manipulating, displaying, and printing the grid chart and provides a graphical

user interface to a rule-based expert system that performs inferences on the data in the chart [7].

The system contains a data base of *reports*, to which it attempts to assign ***identifications***. The reports and their identifications are displayed as colored boxes on the grid chart. The system permits the user to view all or any portion of the grid on an interactive graphic display and to manipulate the displayed data to change identifications and then observe the effects of such manipulations on the display.

The system begins with a data base of reports; it then attempts to assign an identification to each of them and displays the results in the grid chart format. The user may then take any report and manually give it a different identification, and the system will update the display, reflecting both the change entered by the user and any other changes in the identifications of other previously-identified reports made by the expert system in light of the new entry. He can also display or print all or any part of the grid selectively. The user can also ask the system for more information about any report or for reasons why a particular report identification was made. The system provides editing facilities that would be difficult with a paper chart, such as the ability to associate arbitrarily-long text annotations with chart entries and display them separately upon request. It also permits the user to save or restore the data base at any point, to give him a "what-if" capability; that is, he can examine the effect of a tentative identification and then either retain it or remove it from the data base.

The User Interface

The basic user interface design adopted for this system is that each user command consists of a function key corresponding to the name of the command followed by zero or more arguments, as required by the particular command. The function keys are labeled on the keyboard. When one is chosen, the name of the corresponding command is echoed in the ***command window*** on the screen. If arguments are needed, the user is then given a prompt for and description of each argument, one at a time, in the *argument window* and ***explanation window*** on screen, respectively. The argument is echoed in the argument window as it is entered. Text output appears in the *text window* on the screen, which the user may scroll; graphical output, consisting of a view of

all or a portion of the grid chart, appears on a separate monitor.

The Conceptual Model

Before beginning the actual design, a conceptual model must be considered [11]. This defines the basic objects that the user sees and manipulates and their relationships to one another. The basic objects in this system are the reports and the chart. The principal relationship of interest is the mapping of reports onto positions on the chart, where each position on the chart corresponds to a particular identification of a report. Every report has a label, a collection of data fields that describe the report in more detail, a current identification, and, optionally, a text annotation. The label and data fields are derived from the original report, and are not changed by the user. The identification is made automatically by the expert system, but may subsequently be changed by the user or the expert system. The text annotation may be entered by the user and displayed on command.

The Semantic Level

The semantic level describes the functions performed by the system. The semantic design tells what information is needed to perform each function and the result of performing it. It defines "meanings," rather than "forms" or "sequences," which are left to the lower levels. The semantic-level specification provides the high-level model or abstraction of the functions of the system, thereby removing such details from the description of the syntax of the user interface, where they obfuscate the specification.

In the present specification language, the semantic level is concerned with the manipulation of internal variables; no actual input or output operations are described at this level, although the manipulation of values read in as inputs and the generation of values to be displayed as outputs are described. The semantic-level specification consists of descriptions of functions that operate on these internal data, that is, the function parameters, their types, and the effects of the functions. Formal specification of the effects is not addressed here, as it is a general problem in software specification, not unique to user interfaces. The importance of this level is that it per-

mits the details of the semantic level of the user interface to be partitioned from the syntactic- and lexical-level specifications and treated separately, providing dialogue independence [26].

For the example system, the semantic level simply describes the principal operations available to the user. They involve assigning identifications and annotations to reports (the former implicitly includes a call to the expert system); displaying graphical charts and text; and housekeeping functions such as saving and restoring data in files. The specific functions and their meanings are listed in Figure 1. In the present system, they are simply written as procedures in C that either perform the desired operation or communicate with the process running the expert system in LISP, which performs the operation. There are also a few, additional functions of a syntactic nature, which are not described in the semantic specification; these include commands for scrolling the text window or redrawing the display in case it becomes garbled.

The Syntactic Level

The syntactic level describes sequences of inputs and outputs, that is, the rules by which sequences of words (*tokens*) in the language are formed into proper (but not necessarily semantically meaningful) sentences. The specification of the syntactic level describes the sequence of the logical input, output, and semantic operations, but not their internal details. A logical input or output operation is an input or output *token*. Its internal structure is described in the lexical-level specification, while the syntactic-level specification calls it by its name, like a subroutine, and describes when the user may enter it and what will happen next if he does (for an input token) or when the system will produce it (for an output token). A state transition may be associated with an input token or an output token, but not both. Treating outputs as separate tokens on separate transitions (rather than as a special kind of action) in the syntactic-level specification permits the specification to be more symmetric in the way it describes input and output.

The description of the user commands for the example system at the syntactic level is given as a state transition diagram shown as a drawing in Figure 2 and listed as text in Figure 3. The notation in Figures 2 and 3 is directly executable; it is described in more detail in [14, 16]. Briefly, each circle in Figure 2 corresponds to a state. The start state is the one at the left side of

the diagram; the end state (or states) is named inside its circle. Each transition between two states is shown as a directed arc. It may be labeled with one of the following:

- The name of an input token (which begins with **i** followed by a name in upper case, like **iASSIGNCMD**)
- An output token (**o** followed by upper case, like **oREDRAW**)
- A nonterminal (in all lower case), which is defined by a separate diagram that is called like a subroutine and must be traversed from start to end to complete this transition
- An action, which may manipulate variables and which will be executed if this transition is taken
- A condition, which may make tests on the variables and must be true for this transition to be taken

Actions and conditions are shown in numbered footnotes, to save space. A prompt (consisting of an output token) may also be associated with a state; whenever that state is reached, its prompt token will be output. Prompts are also shown in footnotes.

In Figure 3 the same information is presented in a text format. Each diagram begins with a header line that gives the name of the diagram (This is the name by which it could be called as a nonterminal from another diagram.) and the name(s) of its exit state(s). Then, each transition is listed in a line of the form:

st: iASSIGNCMD →*mgetr* **act:**DoIt(a,b,c);

denoting a transition from state **st** to state **mgetr** that accepts the input **iASSIGNCMD** and then calls the action **DoIt(a,b,c)**. An action or condition is represented as one or more procedure calls. Procedure names in upper case denote the semantic functions described above; they are specified and implemented separately. Procedure names in lower case represent support functions and also special functions for generating output; they appear in the lexical-level specification only. In the procedure calls, a variable name preceded by an asterisk denotes a reference parameter; all other parameters are passed by value. The actual value received by an input token (such as the

actual report number obtained for the token **iREPORT**) is available in a variable named **v** plus the token name (e.g., **viREPORT**).

The Lexical Level

The lexical level determines how input and output tokens are actually formed from the primitive hardware operations (*lexemes*). It represents the binding of hardware actions to the hardware-independent tokens of the input and output languages. While tokens are the smallest units of meaning with respect to the syntax of the dialogue, lexemes are the actual hardware input and output operations that comprise the tokens. The lexical-level specification identifies the devices, display windows, and positions with which each token is associated and the primitive hardware operations that constitute them. All information about the organization of a display into areas and the assignment of input and output tasks to hardware devices is confined to this level. For an input token, the specification gives the sequence of primitive input lexemes (for example, key presses) and the device for each lexeme by which the token is entered as well as any lexical output that is produced. For an output token, the lexical specification tells how (that is, with which devices, windows, positions, formats, colors, and the like) the token is presented to the user. The actual information to be presented by an output token may have been set by a semantic action or may be constant; the lexical specification gives the format in which the variable data should be displayed, but not the contents.

The executable lexical-level specification is written in the same state transition diagram notation as the syntactic-level specification. The lexical-level specification for the example system is shown in text format in Figure 4. It consists of a separate state diagram for each input or output token, each of which may be called from the syntactic-level diagrams just as they call other sub-diagrams for nonterminals. In the lexical-level specification, output is described by special actions on the state transitions; such actions are expressed and coded as function calls in the same way as the semantic actions; they perform the actual output. These functions may only be called at the lexical level. At the syntactic level, output is only performed by output token transitions, to avoid mixing output actions with input transitions. At the lexical level, all outputs (other than

lexical echoes) have already been separated from inputs.

To illustrate the distinction between the syntactic and lexical levels, observe that the syntactic-level specification makes no assumptions about the tokens **oREPORT** and **iREPORT**. It simply indicates *when* in the dialogue they are produced or accepted. At this level, the actual method of designating a report need not be specified. It could equally well use an analog pointing device or a code number entered from the keyboard. The lexical-level specification for **iREPORT** encapsulates this decision. The token **iREPORT** will be specified to accept input from the appropriate device, and **oREPORT** will generate a suitable prompt. The variable **viREPORT** returns the actual code number of the chosen report in a device-independent form, regardless of how the user designated the report. In this specification, only the lexical-level specification (Figure 4) reveals that the keyboard entry method was chosen over a pointing device (unfortunately).

Results

The system described here ran directly from the syntactic- and lexical-level specifications shown, using an interpreter that accepts this specification language. The only exception is that there were also a *CANCEL* key and a state-dependent *HELP* key, which were deleted from these figures to make them easier to read. A procedure was written in C for each of the procedures called in actions in the state diagram specification. For the semantic functions, these procedures simply communicated the requests to and obtained output from the process running the expert system in LISP, which maintained the data base and performed the identifications. For output actions, most procedures simply displayed a given text in a designated screen window. The *ReDraw* procedure is the only one that generated graphic output. Using the data in the LISP system and the current settings of the view window limits, it produced a device-independent description of the entire display, which was then transmitted on a local network to the host on which the frame buffer system was located.

This system was designed to demonstrate the potential of an automated aid to users who had previously been performing similar functions by hand. It was therefore particularly

important that its user interface be easy for a novice to master and understandable in terms already familiar to its potential users. The expert system used to assign identifications to reports was sophisticated, but complex and potentially obscure to users accustomed to manual operations. The present graphics system was intended to provide a convenient, easily understandable, "direct manipulation" [22] style of user interface as a front end to the expert system. Potential users were indeed able to understand the system after a minimal introductory session. A further purpose of the system is to obtain empirical feedback from users by having them operate it for a brief session. Again, to be useful in this capacity with limited experimental sessions, it must be easy to operate and understand, and it must avoid ensnaring the novice user in syntactic complexities.

Evaluating the User Interface

Given a formal specification of a user interface such as the present one, how can empirical data be used to measure or improve the design? Two basic approaches may be considered. One is to use information about human performance that has been derived from empirical observations and encoded in models of human performance. Such models may be applied directly to a sufficiently precise and formal specification of a user interface and used to predict user performance with the specified interface. For example, the Keystroke-Level Model [4] could be applied to the present specification. Reisner [21] has attempted to develop and apply performance models to BNF specifications of user interfaces and has been able to verify the resulting performance predictions empirically. Bleser and Foley [2] have also developed a design methodology that applies performance models to the formal specifications of a user interface. Brown [3] has attempted to relate formal properties of a state diagram specification to properties of the corresponding user interface, but without experimental validation.

Somewhat more ad-hoc predictions can be made by using the specification to look for specific properties of a user interface, which have been claimed to be desirable or undesirable. Several (unverified) predictions have been made of specific undesirable characteristics of a user interface, which could be identified from its formal specification. For example, Parnas [19] has

identified "almost-alike states" as a trap for the unwary user. Darlington, Dzida, and Herda [6] have described "interactive deadlock," where the user reaches a state from which he does not know how to exit and nothing he can do in that state will give him the information he needs to learn how to exit from it. Thimbleby [24] has described "character-level ambiguity" in a user interface, where the user is not sure of the (state-dependent) interpretation of the next character he might type. All three of these properties can be readily discovered from a state diagram specification of a user interface.

The other basic evaluation approach does not rely on previously-observed data. Instead, the specification is used to build a prototype of the specified user interface. Experimental observations are then made using the prototype itself. By describing the proposed user interface with an executable specification language, such as the present one, the prototype is immediately available for use. This makes it possible to specify, empirically evaluate, and compare a variety of user interfaces before selecting a design.

Acknowledgments

This work was supported by the Naval Electronic Systems Command under projects directed by Steve Arkin and by H.O. Lubbes.

References

1. A.V. AHO AND J.D. ULLMAN, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).

2. T. BLESER AND J.D. FOLEY, "Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 309-314 (1982).

3. J.W. BROWN, "Controlling the Complexity of Menu Networks," *Comm. ACM* **25** pp. 412-418 (1982).

4. S.K. CARD, T.P. MORAN, AND A. NEWELL, "The Keystroke-Level Model for User Performance Time with Interactive Systems," *Comm. ACM*" **23** pp. 396-410 (1980).

5. M.E. CONWAY, "Design of a Separable Transition-Diagram Compiler," *Comm. ACM*" **6** pp. 396-408 (1963).

6. J. DARLINGTON, W. DZIDA, AND S. HERDA, "The Role of Excursions in Interactive Systems," *International Journal of Man-Machine Studies*" **18** pp. 101-112 (1983).

7. J.K. DIXON, "Design of a Brassboard Computer Aid for Submarine Analysis," Conference on Intelligent Systems and Machines, Oakland University, Rochester, Michigan (1984).

8. M.B. FELDMAN AND G.T. ROGERS, "Toward the Design and Development of Style-independent Interactive Systems," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 111-116 (1982).

9. S. FEYOCK, "Transition Diagram-Based CAI/HELP Systems," *International Journal of Man-Machine Studies*" **9** pp. 399-413 (1977).

10. J.D. FOLEY AND V.L. WALLACE, "The Art of Graphic Man-Machine Conversation," *Proceedings of the IEEE*" **62** pp. 462-471 (1974).

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

12. S.P. GUEST, "The Use of Software Tools for Dialogue Design," *International Journal of Man-Machine Studies*" **16** pp. 263-285 (1982).

13. IBM, "CMS User's Guide," Document GC20-1818-0, IBM Corporation (1976).

14. R.J.K. JACOB, "Executable Specifications for a Human-Computer Interface," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 28-34 (1983).

15. R.J.K. JACOB, "Using Formal Specifications in the Design of a Human-Computer Interface," *Comm. ACM*" **26** pp. 259-264 (1983).

16. R.J.K. JACOB, "An Executable Specification Technique for Describing Human-Computer Interaction," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1985). in press.

17. H. LEDGARD, J.A. WHITESIDE, A. SINGER, AND W. SEYMOUR, "The Natural Language of Interactive Systems," *Comm. ACM*" **23** pp. 556-563 (1980).

18. T.P. MORAN, "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal of Man-Machine Studies*" **15** pp. 3-50 (1981). The Interaction Level of the Command Language Grammar is similar to a state transition diagram specification.

19. D.L. PARNAS, "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proc. 24th National ACM Conference* pp. 379-385 (1969).

20. P. REISNER, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Transactions on Software Engineering*" **SE-7** pp. 229-240 (1981).

21. P. REISNER, "Formal Grammar as a Tool for Analyzing Ease of Use: Some Fundamental Concepts," pp. 53-78 in *Human Factors in Computer Systems*, ed. J.C. Thomas and M.L. Schneider, Ablex Publishing Co., Norwood, N.J. (1984).

22. B. SHNEIDERMAN, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*" **16** pp. 57-69 (1983).

23. A. SINGER, "Formal Methods and Human Factors in the Design of Interactive Languages," Ph.D. dissertation, Computer and Information Science Dept., Univ. Massachusetts (1979).

24. H. THIMBLEBY, "Character-level Ambiguity: Consequences for User Interface Design," *International Journal of Man-Machine Studies*" **16** pp. 211-225 (1982).

25. A.I. WASSERMAN AND D.T. SHEWMAKE, "The Role of Prototypes in the User Software Engineering (USE) Methodology," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1985).

26. T. YUNTEN AND H.R. HARTSON, "A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1985). in press

Figure 1. Semantic-level description of the system, showing a list of the basic functions performed.

ASSIGNREPORT (**r**: report; **i**: id)

Assigns identification **i** to report **r**, overriding the previous identification of this report. If **i** is **0**, this function removes any previous identification from this report and attempts to arrive at a new identification for it automatically. In either case, because of the inference procedure performed by the expert system, changing the identification of one report may affect the identifications of other reports, and all such reports will be updated on the display.

ANNOTATEREPORT (**r**: report; **t**: text)

Associates the text annotation **t** with report **r**.

GETTEXT (**r**: report)

Displays the data fields and annotation associated with report **r**.

GETEXPLAN (**r**: report)

Displays an explanation of why report **r** was given its current identification, in terms of the inference rules in the expert system.

SETVIEWWINDOW (**x, y**: real)

The graphic display represents a window onto a larger chart. This command changes the portion of that chart visible on the display by the proportions given in the arguments. (The user-level commands corresponding to this function provide more convenient zooming and panning operations.)

COPYVIEWWINDOW ( )

Makes a paper copy of the picture currently on the graphic display.

WRITEFILE (**n**: filename)

Saves the current chart data in the file named **n**.

READFILE (**n**: filename)

Loads the previously-saved chart data in the file named **n**.

Figure 2. State transition diagram description of the user interface. (Note: To save space, transitions for the *scrolltext* commands are not shown here, since they appear on every user-visible state; they are shown in Figure 3.)

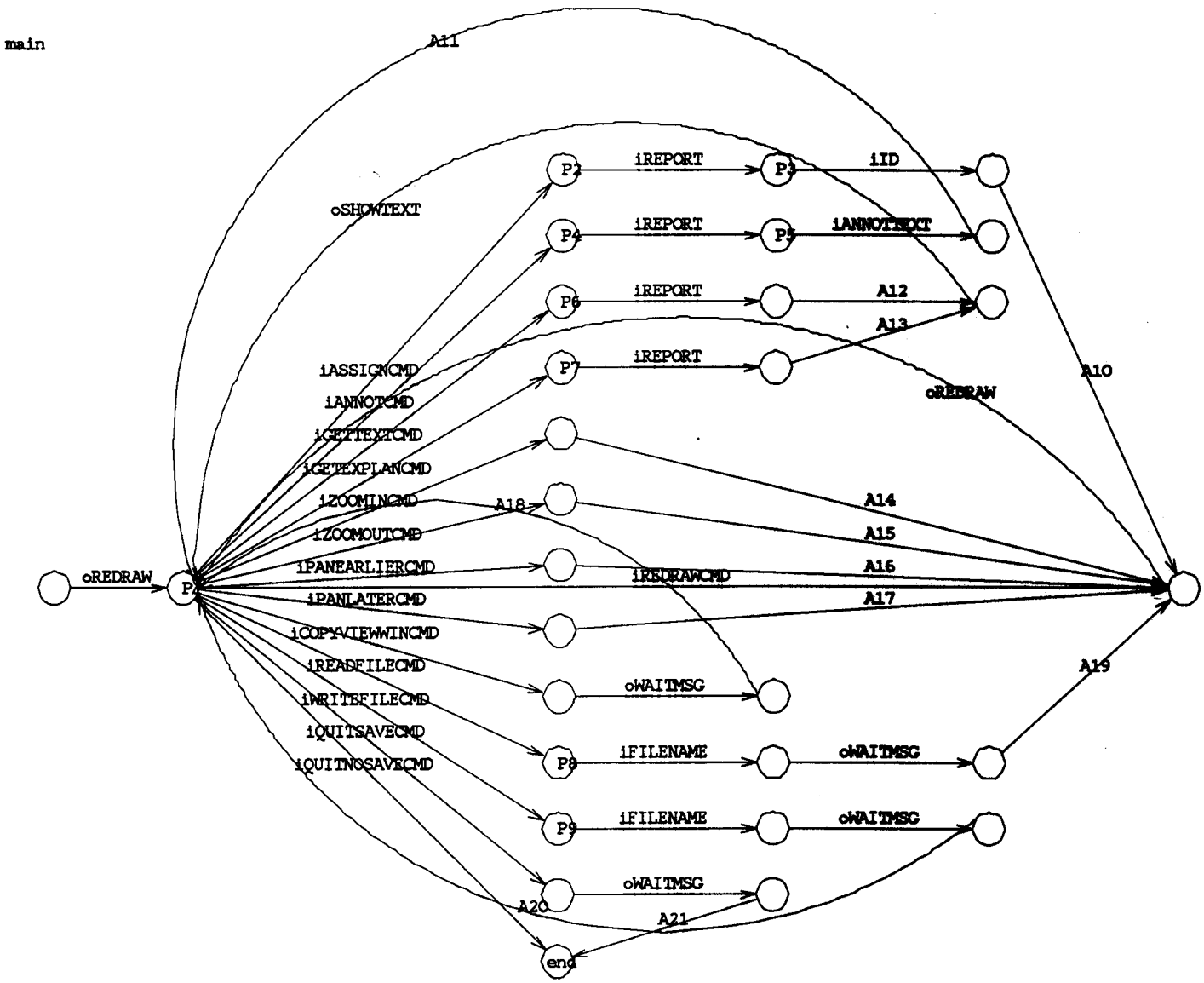

Figure 2 (continued).

main

(1) **Prompt:** oCMD
(2) **Prompt:** oREPORT
(3) **Prompt:** oID
(4) **Prompt:** oREPORT
(5) **Prompt:** oANNOTTEXT
(6) **Prompt:** oREPORT
(7) **Prompt:** oREPORT
(8) **Prompt:** oFILENAME
(9) **Prompt:** oFILENAME
(10) **Act:** ASSIGNREPORT(viREPORT,viID)
(11) **Act:** ANNOTATEREPORT(viREPORT,viANNOTTEXT)
(12) **Act:** GETTEXT(viREPORT,*GLOBAL_text)
(13) **Act:** GETEXPLAN(viREPORT,*GLOBAL_text)
(14) **Act:** SETVIEWWINDOW(+0.4, -0.4)
(15) **Act:** SETVIEWWINDOW(-0.4, +0.4)
(16) **Act:** SETVIEWWINDOW(-0.6, -0.6)
(17) **Act:** SETVIEWWINDOW(+0.6, +0.6)
(18) **Act:** COPYVIEWWINDOW
(19) **Act:** READFILE(viFILENAME)
(20) **Act:** WRITEFILE(viFILENAME)
(21) **Act:** WRITEFILE(".savedb")

Figure 3. Text state transition diagram description of the user **interface syntax.**

main →*end*

init: oREDRAW →*st*

prompt:oCMD
st: iASSIGNCMD →*mgetr*
st: iANNOTCMD →*agetr*

st: iGETTEXTCMD →*tgetr*
st: iGETEXPLANCMD →*egetr*

st: iZOOMINCMD →*zidoit*
st: iZOOMOUTCMD →*zodoit*
st: iPANEARLIERCMD →*panedoit*
st: iPANLATERCMD →*panldoit*

st: iCOPYVIEWWINCMD →*cwait*
st: iREADFILECMD →*rgetf*
st: iWRITEFILECMD →*wgetf*
st: iREDRAWCMD →*redraw*
st: iQUITSAVECMD →*saveitwait*
st: iQUITNOSAVECMD →*end*

prompt:oREPORT
mgetr: iREPORT →*mgeth*

prompt:oID
mgeth: iID →*mdoit*

mdoit: ANY →*redraw* **act:** ASSIGNREPORT(viREPORT,viID);

prompt:oREPORT
agetr: iREPORT →*ageta*

prompt:oANNOTTEXT
ageta: iANNOTTEXT →*adoit*

adoit: ANY →*st* **act:** ANNOTATEREPORT(viREPORT,viANNOTTEXT);

prompt:oREPORT
tgetr: iREPORT →*tdoit*

tdoit: ANY →*showtext* **act:** GETTEXT(viREPORT,*GLOBAL_text);

prompt:oREPORT
egetr: iREPORT →*edoit*

```
edoit:          ANY →showtext act: GETEXPLAN(viREPORT,*GLOBAL_text);

showtext:       oSHOWTEXT →st

zidoit:         ANY →redraw act: SETVIEWWINDOW(+0.4, -0.4);
zodoit:         ANY →redraw act: SETVIEWWINDOW(-0.4, +0.4);
panedoit:       ANY →redraw act: SETVIEWWINDOW(-0.6, -0.6);
panldoit:       ANY →redraw act: SETVIEWWINDOW(+0.6, +0.6);

cwait:          oWAITMSG →cdoit

cdoit:          ANY →st act: COPYVIEWWINDOW;

prompt:oFILENAME
rgetf:          iFILENAME →rwait

rwait:          oWAITMSG →rdoit

rdoit:          ANY →redraw act: READFILE(viFILENAME);

prompt:oFILENAME
wgetf:          iFILENAME →wwait

wwait:          oWAITMSG →wdoit

wdoit:          ANY →st act: WRITEFILE(viFILENAME);

saveitwait:     oWAITMSG →saveit

saveit:         ANY →end act: WRITEFILE(".savedb");

redraw:         oREDRAW →st

/*
 * Here, we list the scroll text commands, which are always available
 * (ie, in every user-visible state)
 */

st, mgetr,mgeth, agetr,ageta, tgetr, egetr, rgetf, wgetf: scrolltext →SAME
;

scrolltext →ret
st:             iSCROLLTEXTFORWARD →fwd
st:             iSCROLLTEXTBACKWARD →back

fwd:            oSCROLLTEXTFORWARD →ret

back:           oSCROLLTEXTBACKWARD →ret
;
```

Figure 4. Text state transition diagram description of the user interface lexical level.

```
/*
 * Command keys and their prompt
 */

oCMD →ret
st:             lEXPLAINWIN →explain
explain:        ANY →stprompt act: PrintWin(
                "Press any function key F1 through F8 for desired command",
                "Use HELP key whenever you want help",
                "Use CANCEL key to cancel any command");
stprompt:       lCMDWIN →promptit
promptit:       ANY →ret act: PrintWin("Command: ");
;

iASSIGNCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY5 →ret act: PrintWin("Assign a report");
;

iANNOTCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY6 →ret act: PrintWin("Annotate a report");
;

iGETTEXTCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY2 →ret act: PrintWin("Show text associated with a report");
;

iGETEXPLANCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY1 →ret act: PrintWin("Explain assignment of a report");
;

iZOOMINCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY8 →ret act: PrintWin("Zoom in (make chart larger)");
;

iZOOMOUTCMD →ret
st:             lCMDWIN →echoit
echoit:         lFKEY4 →ret act: PrintWin("Zoom out (make chart smaller)");
;

iPANEARLIERCMD →ret
```

```
st:          lCMDWIN →echoit
echoit:      lFKEY3 →ret act: PrintWin("Pan left (show earlier reports)");
;

iPANLATERCMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY7 →ret act: PrintWin("Pan right (show later reports)");
;

iCOPYVIEWWINCMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY6S →ret act: PrintWin("Make paper copy of display screen");
;

iREADFILECMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY2S →ret act: PrintWin("Read in previously saved data base");
;

iWRITEFILECMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY1S →ret act: PrintWin("Save copy of current data base");
;

iREDRAWCMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY5S →ret act: PrintWin("Redraw display screen");
;

iQUITSAVECMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY4S →ret act: PrintWin("Save data and quit");
;

iQUITNOSAVECMD →ret
st:          lCMDWIN →echoit
echoit:      lFKEY3S →ret act: PrintWin("Quit without saving data");
;

/*
 * Arguments and their prompts
 */

oREPORT →ret
st:          lEXPLAINWIN →explain
explain:     ANY →stprompt act:
             PrintWin("Enter report number, then press RETURN key",
             "(use BACKSPACE key if needed to correct errors)");
stprompt:    lARGWIN →promptit
```

```
promptit:       ANY →ret act: PrintWin("Report number: ");
;

iREPORT →ret
st:             lARGWIN →getit
getit:          iNUMBER →ret act: assign(*viREPORT,viNUMBER);
;

oID →ret
st:             lEXPLAINWIN →explain
explain:        ANY →stprompt act:
                PrintWin("Enter identification number, then press RETURN key",
                "or enter 0 to remove previous assignment and re-compute.",
                "(use BACKSPACE key if needed to correct errors)");
stprompt:       lARGWIN →promptit
promptit:       ANY →ret act: PrintWin("ID number: ");
;

iID →ret
st:             lARGWIN →getit
getit:          iNUMBER →ret act: assign(*viID,viNUMBER);
;

iNUMBER →ret
st:             lDIGIT →more act:{ EchoWin(vlDIGIT); assign(*viNUMBER,vlDIGIT);};
st:             CH →st

more:           lDIGIT →more act:{ EchoWin(vlDIGIT); append(*viNUMBER,vlDIGIT);};
more:           CH →more act:{ RuboutWin(viNUMBER); Truncate(*viNUMBER);};
more:           CJ →ret

%error
more:           iEATALEX →SAME act: Beep;
;

oANNOTTEXT →ret
st:             lEXPLAINWIN →explain
explain:        ANY →stprompt act: PrintWin("Type in text of annotation",
                "Use RETURN key to separate lines as needed",
                "Press RETURN key twice to end the annotation",
                "(use BACKSPACE key if needed to correct errors)");
stprompt:       lARGWIN →promptit
promptit:       ANY →ret act: PrintWin("Text: ");
;

iANNOTTEXT →ret
st:             lARGWIN →first

first:          iLINE →more act:{ assign(*viANNOTTEXT,viLINE);
                AppendNL(*viANNOTTEXT); ScrollWin;};
```

```
more:           iLINE →more act:{ append(*viANNOTTEXT,viLINE);
                AppendNL(*viANNOTTEXT); ScrollWin;};
more:           CJ →ret

%error
more:           iEATALEX →SAME act: Beep;
;

oFILENAME →ret
st:             lEXPLAINWIN →explain
explain:        ANY →stprompt act: PrintWin("Enter file name then, press RETURN key",
                "(use BACKSPACE key if needed to correct errors)");
stprompt:       lARGWIN →promptit
promptit:       ANY →ret act: PrintWin("File name: ");
;

iFILENAME →ret
st:             lARGWIN →getit
getit:          iLINE →ret act:assign(*viFILENAME,viLINE);
;

iLINE →ret
first:          lULCHAR →more act:{ EchoWin(vlULCHAR); assign(*viLINE,vlULCHAR);};
first:          lDIGIT →more act:{ EchoWin(vlDIGIT); assign(*viLINE,vlDIGIT);};
first:          lPUNCT →more act:{ EchoWin(vlPUNCT); assign(*viLINE,vlPUNCT);};
first:          SP →more act:{ EchoWin(" "); assign(*viLINE," ");};
first:          CH →first

more:           lULCHAR →more act:{ EchoWin(vlULCHAR); append(*viLINE,vlULCHAR);};
more:           lDIGIT →more act:{ EchoWin(vlDIGIT); append(*viLINE,vlDIGIT);};
more:           lPUNCT →more act:{ EchoWin(vlPUNCT); append(*viLINE,vlPUNCT);};
more:           SP →more act:{ EchoWin(" "); append(*viLINE," ");};
more:           CH →more act:{ RuboutWin(viLINE); Truncate(*viLINE);};
more:           CJ →ret

%error
more:           iEATALEX →SAME act: Beep;
;

/*
 * Other output tokens
 */

oWAITMSG →ret
st:             lERRHELPWIN →showit
showit:         ANY →ret act: PrintWin("This takes a few moments...please wait");
;

/*
 * This redraws the graphic display
 */
oREDRAW →ret
```

```
st:             ANY →ret act: ReDraw;
;

/*
 * This displays the value of GLOBAL_text on the text window,
 * starting from line number GLOBAL_textline of the data
 */
oSHOWTEXT →ret
st:             lTEXTWIN →print
print:          ANY →ret act:{ assign(*GLOBAL_textline,0);
                ShowText(GLOBAL_text,GLOBAL_textline);};
;

/*
 * Operations for scrolling the text window
 */

iSCROLLTEXTFORWARD →ret
st:             lFKEY8S →ret
;

iSCROLLTEXTBACKWARD →ret
st:             lFKEY7S →ret
;

oSCROLLTEXTFORWARD →ret
st:             lTEXTWIN →move
move:           ANY →print act: ScrollText("+",GLOBAL_text,*GLOBAL_textline);
print:          ANY →ret act: ShowText(GLOBAL_text,GLOBAL_textline);
;

oSCROLLTEXTBACKWARD →ret
st:             lTEXTWIN →move
move:           ANY →print act: ScrollText("-",GLOBAL_text,*GLOBAL_textline);
print:          ANY →ret act: ShowText(GLOBAL_text,GLOBAL_textline);
;

/*
 * Lexemes for designating windows
 */

lCMDWIN →ret
st:             ANY →ret act: SetCmdWin;
;

lARGWIN →ret
st:             ANY →ret act: SetArgWin;
;

lEXPLAINWIN →ret
```

st: ANY →*ret* **act:** SetExplainWin;
;

lERRHELPWIN →*ret*
st: ANY →*ret* **act:** SetErrHelpWin;
;

lTEXTWIN →*ret*
st: ANY →*ret* **act:** SetTextWin;
;