Executable Specifications for a Human-Computer Interface

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

It is useful to be able to specify a proposed human-computer interface formally before building it, particularly if a mockup suitable for testing can be obtained directly from the specification. A specification technique for user interfaces, based on state transition diagrams, is introduced and then demonstrated for a secure message system application. An interpreter that executes the resulting specification is then described. Some problems that arise in specifying a user interface are addressed by particular features of the technique: To reduce the complexity of the developer's task, a user interface is divided into the semantic, syntactic, and lexical levels, and a separate executable specification is provided for each. A process of stepwise refinement of the syntactic specification, leading from an informal specification to an executable one is also presented. Since the state diagram notation is based on a nondeterministic model, constraints necessary to realize the system with a deterministic interpreter are given.

Writing a formal specification of the user interface of a computer system before building it permits the interface designer to consider a variety of possible user interfaces and describe them precisely and compactly without actually having to code them. It also permits the application of human performance models to the specifications to obtain information about the user interfaces they describe before building them [1, 15]. The specifications can be checked for certain undesirable properties of the user interface, such as almost-alike states [14], interactive deadlock [4], and character-level ambiguity [18]. Further benefits accrue if a prototype or mockup of the user interface of the proposed system can be constructed directly from the specification. Problems with the proposed user interface can then be identified early in the design process, when they are easier to fix. While many prospective users will find a formal specification of a proposed system difficult to understand, they will have much less trouble evaluating a mockup system and identifying deficiencies in its user interface, both through informal demonstrations and formal experiments.

This paper describes a specification technique for user-computer interfaces and its use in the development of a secure military message system. The specifications are executed interpretively to provide a working prototype of the system. The paper describes the specification technique by means of an example. Then, it shows the decomposition of both the design process and the specification itself into the semantic, syntactic, and lexical levels and describes how stepwise refinement can be used at the syntactic level. The constraints on the specification necessary for a system to be realized with a deterministic interpreter are given, and the implementation of the interpreter is described.

Overview of the Specification Technique

Time sequence is an important aspect of the surface structure of an interactive system as seen by a user. Specifications based on state transition diagrams are most suitable for describing interactive human-computer interfaces largely because they represent time sequence explicitly, in contrast to BNF, in particular, where it is implicit [12]. The state transition model has also been found useful in describing a user's mental model of an interactive computer system [4, 13]. Several investigators have proposed different specification nota-

© 1983 ACM 0-89791-121-0/83/012/0028 $00.75

tions based on state transition diagrams [2, 3, 5, 6, 14, 17, 20]. Each provides some unique benefits but also has some disadvantages [10]. Only a few are sufficiently formal or complete to be executed directly, and fewer still have interpreters [5, 19].

The present technique is based on the state transition diagrams introduced in [12] and has been refined based on experience applying it to a message system. It is an attempt to synthesize the most useful features of previous notations and to permit an interpreter to execute the specification. Using this approach, a working prototype of a receive-only secure military message system has been specified and constructed as one member of a family of prototypes built in the Secure Military Message Systems project at the Naval Research Laboratory. Experience with the prototype has shown that the design of the user interface is critical in a multi-level-secure system, both to ensure that the user maintains the security of the system and to prevent security considerations from hampering the progress of his or her work.

Details and Example of the Specification Technique

To describe the technique, consider a section of the syntax portion of the executable specification of the prototype message system, as shown in Figure 1. (The full executable specification is given in [11].) Each state is represented as a circle. 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 or more of the following:

- The name of an input token (which begins with **i** followed by a name in upper case, like **iQUIT**)
- An output token (**o** followed by upper case, like **oLOGNAME**)
- A nonterminal (in all lower case, like **login**), 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
- A condition, which may make arbitrary tests on external variables and must be true for this transition to be taken (**testret**, shown here, is a special type of condition that examines the state through which a called nonterminal diagram with several exits returned)
- An action, which may manipulate the external variables and which will be executed if this transition is taken (no examples appear in Figure 1).

Description of tokens and nonterminals called by **mms**. The output tokens **oLOGIN** and **oLOGNAME** display an empty login template and a prompt for the user's name, respectively, and are described in a separate lexical-level specification in the same notation. **iQUIT** and **iLOGOUT** are input tokens whose internal details are also described in the lexical-level specification. **login** is a nonterminal consisting of the entire log-in sequence; its definition is not shown here, but it does have two possible exits, depending on whether the user entered valid login data. The **setup** nonterminal initializes a user session and displays incoming messages. The **cmd** nonterminal describes the user commands and is given below.

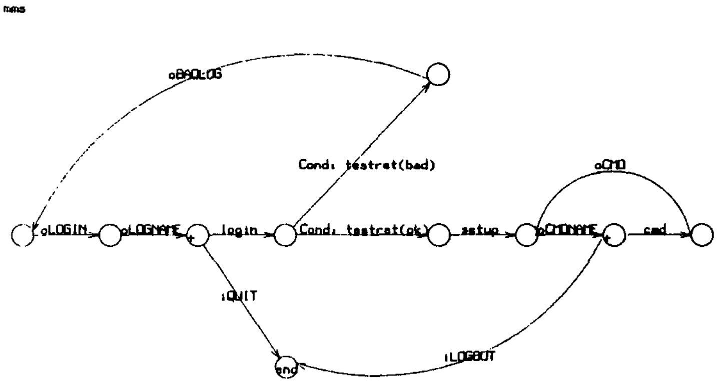

Figure 1. Specification of the prototype message system syntax—first diagram.

When the system is started, the login template and then the prompt for the user's name are displayed. The user can then attempt to log in or else enter a **quit** command to exit from the system entirely. If the login is successful, the user is prompted for a command, or he may log out. After each command other than **logout**, a fresh command template (**oCMD**) is displayed, and the user is prompted (**oCMDNAME**) to enter another command name.

Text form. This same diagram can be represented in text form, as shown in Figure 2. The diagram in this form is the actual input to the interpreter, as well as to the program that produced Figure 1. 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: oLOGIN →*promptlog*

denoting a transition from state **st** to state **promptlog** that produces output token **oLOGIN**. Examples of how input tokens, nonterminals, and conditions are expressed in this notation appear in Figure 2. The use of various typefaces in the printed version is incidental; the plus signs denote "user-visible" states, discussed later.

```
mms →end

st:          oLOGIN →promptlog

promptlog:   oLOGNAME →getlog

+getlog:     login →gotlog
+getlog:     iQUIT →end

gotlog:      cond:testret("ok"); →setup
gotlog:      cond:testret("bad"); →badlog

badlog:      oBADLOG →st

setup:       setup →promptcmd

promptcmd:   oCMDNAME →getcmd

+getcmd:     cmd →ready
+getcmd:     iLOGOUT →end

ready:       oCMD →promptcmd
;
```

Figure 2. Text form of first diagram of the message system specification.

Additional nonterminals. Figure 3 shows more of the specification of the prototype message system. First, it shows a portion of the diagram for the **cmd** nonterminal, which was called from Figure 2. (Most of the individual commands left out because they are repetitive.) **cmd** obtains a command name, calls the appropriate nonterminal diagram (such as **copy_mc**) to get the arguments to the command if any and execute it, and then decides whether to return immediately, to display output using the **scroll** nonterminal and then return, or to display an error message (**oCMDERR**) and return. Figure 3 also shows the nonterminal that obtains the arguments to one of the commands and executes it (**copy_mc**).

```
cmd →ret

+getcn:      iDISPLAY_MSG →ce_display
+getcn:      iCOPY_MC →ce_copy
+getcn:      iCREATE_MF →ce_create

ce_display:  oCLRERR →do_display
ce_copy:     oCLRERR →do_copy
ce_create:   oCLRERR →do_create

do_display:  display_msg →test
do_copy:     copy_mc →test
do_create:   create_mf →test

test:        cond:testret("noshow"); →ret
test:        cond:testret("show"); →show
test:        cond:testret("err"); →err

show:        scroll →ret

err:         oCMDERR →ret
;

copy_mc →(noshow,err)

promptn:     oMSGNUM →getn

+getn:       iMSGNUM →promptf

promptf:     oFILENAME →getf

+getf:       iFILENAME →test
             act:COPY_ME(*voCMDERR,
             viMSGNUM, GLOBAL_curmf,
             viFILENAME);

test:        cond:equal(voCMDERR, "OK");
             →noshow
test:        cond:NOT equal(voCMDERR, "OK");
             →err
;
```

Figure 3. Additional diagrams from the message system specification.

Actions and conditions. An action or condition is represented as one or more function calls. Function names in upper case (e.g., **COPY_ME**) denote commands that create, modify, or display message system data objects. These commands constitute the semantics of the system and are described and implemented separately. Functions names in lower case represent operations on local variables (like **equal** or **assign**); many of these would be provided by a typical programming language, but, to keep the interpreter language simple, they are treated as external functions here. 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 number entered for the token **iMSGNUM**) is available in a variable named **v** plus the token name (e.g., **viMSGNUM**). When output tokens are to display variable data (rather than constant messages or prompts), such data may be passed to them with similarly-named variables (e.g., the variable **voCMDERR** contains the actual error message that will be displayed by the token **oCMDERR**; it was set by **COPY_ME**). All variables contain character strings of arbitrary length.

Three Levels of the Specification

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 [6] introduced the notion of describing an interactive user interface at the *semantic*, *syntactic*, and *lexical* levels, and that model is followed here. 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.

The three levels are defined by Foley and van Dam [7]: The *semantic* level describes the functions performed by the system. It tells what information is needed to perform each function and the result of performing it. The *syntactic* level describes the sequences of inputs and outputs. For the input, this means the rules by which sequences of words (*tokens*) in the language are formed into proper (but not necessarily semantically meaningful) sentences. The *lexical* level determines how input and output tokens are actually formed from the primitive hardware operations (*lexemes*).

The Semantic Level

In the actual specification, 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. Specification of the effects is not considered here, as it is a general problem in software specification, not unique to user interfaces. Techniques such as pseudocode or algebraic specifications would be appropriate. The semantic functions are simply supplied to the specification interpreter as code in a conventional programming language (C).

In the prototype message system implementation, the bulk of the semantic functions are implemented in a separate program written in LISP [9]. The LISP interpreter runs as a separate process and provides the semantic operations upon request from the process running the state diagram interpreter. The functions actually executed by the diagram interpreter simply send requests to and receive output from the process running the LISP program, which may thus be viewed as an abstract machine that implements the semantics of the message system. The operations provided by that machine are described in a separate specification [8]. The details of the semantic-level specification of the user interface are thereby partitioned from the syntactic- and lexical-level specifications and treated separately [12].

The Syntactic Level

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). The syntactic-level specification is written entirely in state transition diagram notation and is directly executable. Figures 1 through 3 show syntactic-level specifications. A transition in one of these diagrams may call a lexical diagram for a token, another syntactic diagram for a nonterminal symbol, or an action or condition consisting of one or more of the semantic functions defined above.

With the present technique, 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. It is analogous both to Shneiderman's multi-party adaptation of BNF [16] and Singer's version of state transition diagram notation [17] in that similar kinds of tokens or transitions are used separately for input and output.* It differs from most other state transition diagram-based notations that have been used to describe the syntax of interactive languages in that they describe user input on state transitions, but then append to the transitions actions that both produce output and modify internal data. Thus they describe the input syntax clearly but confound the output with internal actions and input transitions.

*This could obviously be extended to more than two-way conversations by choosing better names for the directions of the multi-party conversation than the present **iTOKENNAME** and **oTOKENNAME**, which stand for *input* and *output* token names.

The Lexical Level

The lexical level specification describes the physical embodiment of each of the input and output tokens, including identifying the devices, display windows, and positions with which they are associated and the primitive lexemes 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.

The executable lexical-level specification is written in the same state transition diagram notation, avoiding introducing another notation and another interpreter. As shown in Figure 4, the lexical-level specification 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. At this level, output is described by special actions tacked onto the state transitions; such actions are expressed and coded as function calls in the same way as the semantic actions; and 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.

For an input token, the lexical-level 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. Lexical output constitutes prompts and acknowledgments for the individual lexemes that make up a token; most often, it consists of echoes. The lexical-level specification consists of state diagrams that call lexemes, which are either individual hardware input actions (with names entirely in upper case, like **NEWLINE**) or else sub-diagrams that directly call those hardware actions (with names of the form **l** followed by a lexeme name in upper case, like **lULCHAR**, any upper- or lower-case alphabetic character). Lexical outputs (echoes) are given using the special output actions.

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 (for semantic output) or may be constant (for syntactic output). The lexical specification gives the format in which the data should be displayed, and, in the case of syntactic output, the contents. The lexical-level output specification is also written in state diagram notation, again calling the special actions to perform the actual output. Some primitive objects used for producing output are defined as output lexemes, specified in their own sub-diagrams. In particular, all window selections are considered lexemes (e.g., **lNAMEWIN**), so that each token specification can make explicit which window it uses by calling the lexeme for that window, rather than putting that information in the output function definitions.* The diagram is executable, like the other diagrams, but it is generally just a linear sequence of lexeme and function calls.

```
oBADLOG →ret
st:           lERRWIN →refresh
refresh:      lREFRESH →print
print:        →ret act:print("Sorry, try again -
               or press ESC to exit");
;

oCMDERR →ret
st:           lERRWIN →refresh
refresh:      lREFRESH →print
print:        →ret act:print(voCMDERR);
;

iCOPY_MC →ret
st:           lCMDWIN →getit
getit:        lFKEY7 →ret
              act:print("copy_mc");
;

iFILENAME →ret

st:           lCMDWIN →firstchar

firstchar:    lULCHAR →more
              act:{print(vlULCHAR);
              assign(*viFILENAME, vlULCHAR)};

more:         lULCHAR →more
              act:{print(vlULCHAR);
              append(*viFILENAME, vlULCHAR)};

more:         NEWLINE →ret
;
```

Figure 4. Examples of lexical specifications from the message system.

*Shneiderman [16] introduced a comparable scheme to an extended form of BNF. By making the window selection an output lexeme here, the notation need not be extended to handle this situation.

Stepwise Refinement of the Syntax Specification

While the first aspect of the syntax to be designed should usually be the input syntax, more details must eventually be provided, still at the syntactic level, to yield a complete (executable) specification. Beginning with a specification of the input language, a description of the output is added in a process of stepwise refinement that leads to a complete syntactic specification.

The first step is a diagram of the input syntax only, with no actions or outputs. At every state in this specification, the computer is waiting for input from the user. Next, informal descriptions of the actions and outputs are added to each transition, but the sequence of the actions, outputs, and conditions associated with any single transition is not formally specified. In the third step, new states are introduced into the diagrams. Each individual action and condition is put on its own state transition, and each output operation is defined as a separate output token and put on its own transition. This means that new "internal" states are introduced into the specification, in which the system is not waiting for user input. The user never observes the system in any of these states; he only sees it in the states in which it is waiting for input. The latter are called *user-visible* states and may be marked with plus signs in the specification. In the fourth step, the individual actions and conditions are specified formally, that is, as function calls to specific semantic-level functions. Figures 1 through 3 all show syntactic-level specifications corresponding to this step. Finally, provisions for handling errors and features, such as help, abort-command, and escape to monitor, are made in the fifth step. State transitions for these purposes are added to some or all of the user-visible states.

To aid in the early stages of this process of stepwise refinement, the specification interpreter may be told to provide stubs for missing sub-diagrams in a specification and simply to print descriptions of actions instead of trying to execute them. Thus, the specification in its early, informal stages may still be parsed, drawn, and executed automatically.

Restrictions on Nondeterminism

Introducing output tokens into the syntax diagrams on their own separate transitions implies that there should not be a "fork" in a diagram (a state with more than one transition leading from it) where there is an output token. That is, any state with a choice of transitions leading from it must make that choice by accepting different *input* tokens (or testing conditions), rather than different *output* tokens, since a transition with an output token is always "selected." This is actually a special case of a more general restriction that must be placed on these specifications to make them realizable by a deterministic interpreter, irrespective of whether output is specified by separate tokens. The syntax diagrams describe a nondeterministic automaton, which is simulated by a deterministic interpreter. The interpreter selects an arbitrary path, tries it, and, if it reaches a dead end, backtracks and tries another path instead. In an interactive system, it is meaningless to backtrack over a path that has already generated output to the user. The following constraint will prevent this: *Starting at each state at which there is a fork, the inputs that will cause the machine to reach any transition that will produce an output to the user must be disjoint from the inputs that will cause it to reach any other transition with an output.* That is, from any state, the same initial input cannot cause two different output transitions, even though subsequent input might disambiguate them.

Implementation

The specification interpreter is written in C (about 2000 lines of code) and runs under UNIX on a VAX. A common front end, constructed with YACC and LEX from a BNF description of the specification language, is used to parse the specification, both for interpreting it and for converting it to diagram form. The semantic functions and the output functions used by the lexical-level specification are coded in C and then linked with the interpreter. Device-independent facilities for full-screen text terminals and also graphical output devices are available to these functions.

Conclusions

This paper has presented a technique for specifying the user interface of an interactive computer system and described how it has been used to produce a formal and executable specification of the user interface of a military message system. The technique permits the designer of a user interface to describe the interface completely and obtain a prototype of it directly from the specification. The notation uses state transition diagrams to emphasize the time sequence aspects of the user-visible behavior of the system. It permits both the specification and the design process to be separated into the semantic, syntactic, and lexical levels, and it supports a process of stepwise refinement at the syntactic level.

Acknowledgments

I would like to thank Connie Heitmeyer, Mark Cornwell, and Carl Landwehr for their helpful suggestions and comments on this work. This work was supported by the Naval Electronic Systems Command under the direction of H.O. Lubbes.

References

1. T. Bleser and J.D. Foley, "Towards Specifying and Evaluating the Human Factors of User-Computer Interfaces," *Proc. Human Factors in Computer Systems Conference*, pp. 309-314 (1982).
2. MUMPS Development Committee, *MUMPS Language Standard*, American National Standards Institute, New York (1977).
3. M.E. Conway, "Design of a Separable Transition-Diagram Compiler," *Comm. ACM* **6** pp. 396-408 (1963).
4. 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).
5. M.B. Feldman and G.T. Rogers, "Toward the Design and Development of Style-independent Interactive Systems," *Proc. Human Factors in Computer Systems Conference*, pp. 111-116 (1982).
6. J.D. Foley and V.L. Wallace, "The Art of Graphic Man-Machine Conversation," *Proceedings of the IEEE* **62** pp. 462-471 (1974).
7. J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass. (1982).
8. C.L. Heitmeyer, "An Intermediate Command Language (ICL) for the Family of Military Message Systems," Naval Research Laboratory Technical Memorandum 7590-450:CH:ch (13 November 1981).
9. C.L. Heitmeyer, C.E. Landwehr, and M.R. Cornwell, "The Use of Quick Prototypes in the Military Message Systems Project," *ACM SIGSOFT Software Engineering Notes* **7** pp. 85-87 (1982).
10. R.J.K. Jacob, "Survey and Examples of Specification Techniques for User Interfaces," NRL Report, Naval Research Laboratory, Washington, D.C. (1983).
11. R.J.K. Jacob, "Formal Specification of the User Interface of a Receive-only Secure Military Message System Prototype," Naval Research Laboratory Technical Memorandum 7590:RJ:rj (1983).
12. R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Comm. ACM* **26** pp. 259-264 (1983).
13. 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.
14. 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).
15. P. Reisner, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Transactions on Software Engineering* **SE-7** pp. 229-240 (1981).
16. B. Shneiderman, "Multi-party Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-12**(2) pp. 148-154 (March 1981).
17. A. Singer, "Formal Methods and Human Factors in the Design of Interactive Languages," Ph.D. dissertation, Computer and Information Science Dept., Univ. Massachusetts (1979).
18. H. Thimbleby, "Character-level Ambiguity: Consequences for User Interface Design," *International Journal of Man-Machine Studies* **16** pp. 211-225 (1982).
19. A.I. Wasserman and D.T. Shewmake, "Rapid Prototyping of Interactive Information Systems," *ACM SIGSOFT Software Engineering Notes* **7** pp. 171-180 (1982).
20. W.A. Woods, "Transition Network Grammars for Natural Language Analysis," *Comm. ACM* **13** pp. 591-606 (1970).