Using Formal Specifications in the Design of a Human-Computer Interface

ROBERT J. K. JACOB *Naval Research Laboratory*

Robert J.K. Jacob is investigating formal specification techniques for describing user-computer interaction for both the design and construction of user interfaces for computer systems.

This work was supported by the Naval Electronic Systems Command under the direction of H. O. Lubbes.

Author's Present Address: Robert J.K. Jacob, Computer Science and Systems Branch, Code 7590, Naval Research Laboratory, Washington, D.C. 20375.

0001-0782/83/0400-0259 75¢.

***ABSTRACT:** Formal specification techniques are valuable in software development because they permit a designer to describe the external behavior of a system precisely without specifying its internal implementation. Although formal specifications have been applied to many areas of software systems, they have not been widely used for specifying user interfaces. In the Military Message System project at the Naval Research Laboratory, the user interfaces as well as the other components of a family of message systems are specified formally, and prototypes are then implemented from the specifications. This paper illustrates the specification of the user interface module for the family of message systems. It then surveys specification techniques that can be applied to human-computer interfaces and divides the techniques into two categories: those based on state transition diagrams and those based on BNF. Examples of both types of specifications are given. Specification notations based on state transition diagrams are preferable to those based on BNF because the former capture the surface structure of the user interface more perspicuously. In either notation, a high-level abstraction for describing the semantics of the user interface is needed, and an application-specific one is used here.*

INTRODUCTION

Formal specification techniques have been applied to many aspects of software development. Their value is that they permit a designer to describe the external behavior of a system precisely without specifying its internal implementation. However, such techniques have been applied only rarely to the specification of user interfaces, despite the fact that the user interface is increasingly being recognized as a critical element in many software systems. One is handicapped in trying to design a good user interface without a clear and precise technique for specifying such interfaces.

The design of the user interface for a military message system has a special importance because of its role in maintaining the security of classified messages. Enforcement of system security requires that the user understand the security-related consequences of his or her actions, but often such consequences are not intuitively obvious. Recent experimental results indicate that communicating the security implications of an action and obtaining meaningful approval or disapproval from a user can be very difficult [18].

In the Military Message System (MMS) project at the Naval Research Laboratory, formal specification techniques are being used to construct a family of secure message systems [6, 7]. Compatible specification techniques must therefore be applied to the human-computer interfaces for such systems. Rapid prototypes of military message systems will then be constructed based on these specifications [8], and, later, full-scale prototypes will be built.

This paper describes the specification of the user interface module for the family of message systems, surveys specification techniques that can be applied to human-computer interfaces, provides examples of specifications, and presents some conclusions [10].

USER INTERFACE SPECIFICATIONS FOR THE MILITARY MESSAGE SYSTEM FAMILY

Each member of the MMS family consists of several components. Two are of interest here: the User Agent and the Data Manager. As shown in Figure 1, the user communi-

cates with the User Agent via a ***User Command Language*** (UCL). Once it receives a command from the user, the User Agent translates the command into a standard form—a statement in the ***Intermediate Command Language*** (ICL)—and passes that to the Data Manager. Information returned by the Data Manager in response to user requests is delivered to the User Agent, which is responsible for displaying it to the user. This division permits new systems with different user interfaces to be constructed from an existing system with relative ease. If the user interface is changed, only the User Agent must be modified so that it will translate from the new UCL into the standard ICL; the Data Manager and other system components need not be changed. For example, two members of the MMS family could provide the same basic functions but differ in their user interfaces. They would have two

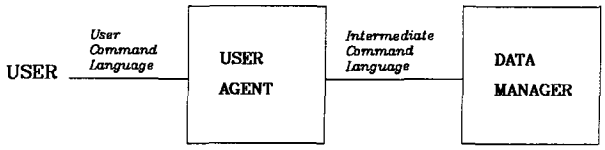


FIGURE 1. Components of the Military Message System.

different User Agents but would share the same Data Manager. Separating the user interface in this way makes it possible to experiment with different user interfaces and to evaluate them from their specifications (as Reisner [15] and Embley [3] do) as well as from a prototype (as Hanau and Lenorovitz [5] do).

This division also makes the specification of the user interface clearer. Previous user interface specifications have suffered because they lacked an acceptable language for describing the "semantics" of the interface, that is, the actions 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 [2]. What is needed is a high-level model that describes the operations that the system performs. Then, the user interface specification would describe the user interface in terms of the model, while the internal details of the model would be described in a separate specification.

The design used in the Military Message System project provides one solution to this problem. The ICL *is* an abstract model of the services performed by a message system and is described in a separate specification [7]. The user interface specification, then, needs only to describe the ***syntax*** of the UCL using a language specification technique and the ***semantics*** of the UCL using ICL statements.

PROPERTIES OF A SPECIFICATION TECHNIQUE

In selecting a technique for specifying a human-computer interface, one should seek the following properties:

- The specification of a user interface should be easy to understand. In particular, it must be easier to understand and take less effort to produce than the software that implements the user interface.
- The specification should be precise. It should leave no doubt as to the behavior of the system for each possible input.
- It should be easy to check for consistency.
- The specification technique should be powerful enough to express nontrivial system behavior with a minimum of complexity.
- It should separate what the system does (function) from how it does it (implementation). The technique should make it possible to describe the behavior of a user interface, without constraining the way in which it will be implemented.
- It should be possible to construct a prototype of the system directly from the specification of the user interface.
- The structure of the specification should be closely related to the user's mental model of the system itself. That is, its principal constructs should represent concepts that will be meaningful to users (such as answering a message or examining a file), rather than internal constructs required by the specification language. Alternatively, the specification should directly yield a reasonable table of contents for a user manual, but not necessarily the material in the manual.

SURVEY OF SPECIFICATION TECHNIQUES

Much of the work applicable to techniques for specifying human-computer interfaces has been concerned with static rather than interactive languages. 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, such as a program, has been processed. Processing of an input text is affected little, if at all, by the previous inputs. In an interactive language, the input can be described as a series of brief texts, where the processing of each input generally depends on previous inputs, or, equivalently, as one long text, in which the computer takes actions and produces outputs at various points during the input, resulting in a dialogue. Hence, a specification for such a language must capture not only the system actions and outputs but also their sequence with respect to portions of the input.

Most specifications for both static and interactive languages have been based on one of two formal models: Backus-Naur Form (BNF) [15] and state transition diagrams [14]. Each of these methods provides a syntax for describing legal streams of user inputs. In order to be used to specify interactive languages, the techniques must be modified to describe—in addition to user inputs—the system actions and their sequence with respect to the input.

BNF

For BNF, the necessary modification 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. (As mentioned, for the MMS family members, these system actions can be described as ICL statements issued by the User Agent to the Data Manager.)

Reisner [15] provides an example of how BNF can be used to describe a user interface. Unlike several other published specifications, Reisner's specifies a nontrivial, real-world system. It does leave out the semantics of the user interface—the system actions and responses—since Reisner did not need them for her purposes. That is, it provides a standard BNF description of the input language, but no specification of any outputs. Reisner uses formal properties of the BNF specifications of two sys-

tems to predict differences in the performance of their users. More complex or less consistent BNF rules lead to predictions of user errors. Several such predictions are then verified experimentally.

Shneiderman [16] also examines the use of BNF for describing interactive user interfaces and proposes a modified form of BNF in which each nonterminal symbol may be associated with either the computer or the user. With the exception of one unusual nondeterministic case, this type of grammar can be mapped into a state transition diagram similar to Singer's [17].

A BNF specification can also be used as input to a compiler-compiler, such as YACC [11]. Such a program can be given a specification in which an executable action (rather than a routine to generate object code) is associated with each BNF rule; then it can automatically construct a prototype of the specified system. While the input syntax is described in BNF, the actions must generally be given in a conventional programming language.

One general problem that arises with BNF-based techniques is that it is sometimes difficult to determine exactly ***when*** something will occur, that is, after exactly ***what*** input tokens have been recognized. This makes it awkward to specify interactive prompting, help messages, and error handling, which must occur at particular points in a dialogue. Often, it requires the introduction of a collection of otherwise irrelevant nonterminal symbols into the specification.

STATE TRANSITION DIAGRAMS

To represent interactive languages, state transition diagrams are modified in a way similar to that for BNF-based techniques. Each transition is associated with an action; whenever the transition occurs, the system performs the associated action. Since the concept of sequence is explicit in a state diagram (while it is implicit in BNF), the former is more suited to specifying the points in a dialogue at which events occur.

Conway [1] presents an early use of a notation based on state transition diagrams in which an action is associated with each transition. His goal, however, was to specify and construct a compiler for a static language, so he did not address the problems of interactive user interfaces.

Woods [19] also describes a notation, "Augmented Transition Networks," based on state transition diagrams for analyzing a static language. His notation includes an extension to conventional state transition diagrams: a (global) data structure. The actions associated with each transition manipulate this structure, and the conditions for making a state transition can include arbitrary Boolean expressions that depend on the data structure.

Conway and Woods both introduce into their state diagrams a feature analogous to BNF nonterminal symbols. With this feature, instead of labeling a state transition with a single input token, the transition may be labeled with the name of a nonterminal symbol. That symbol is, in turn, defined in a separate state transition diagram. The labeled path in the main diagram is then taken if the entire separate diagram can be traversed at the current position in the input stream. This makes it possible to divide complex diagrams into more manageable pieces. It also makes it easier to introduce nondeterminism into the specification.[1] If recursive calls to these separate diagrams are permitted, the notation is equivalent in power to BNF [9]. Adding the actions and conditions (and the ability to create new variables dynamically) either to these state transition diagram notations or to the BNF notations makes the resulting notation equivalent in power to a Turing Machine.

Parnas [14] proposes the use of state diagrams to describe user interfaces for interactive languages. He differentiates "terminal state" from "complete state" in a way analogous to the separation of syntax from semantics in other specifications. Parnas' paper contains some very simple examples but does not address how the scheme would be extended for more complex real-world systems.

Foley and Wallace [4] also advocate the use of a state diagram to represent the user interface of an interactive system. While their notation is clear and easy to understand, they, too, do not examine the problem of specifying real-world systems.

The standard for the MUMPS interactive computer language [12] provides an example of a specification of a complex system that uses a notation based on state diagrams. The specification uses nonterminal symbols extensively and gives a precise deterministic procedure for interpreting diagrams containing them (since their use can, in the general case, require a nondeterministic automaton). The specification is noteworthy in that the actions associated with its transitions comprise a complete specification of the semantics of the MUMPS language.

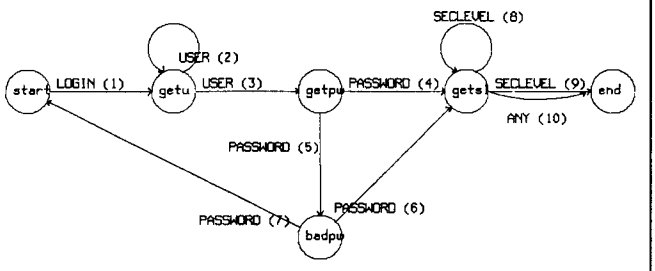
Singer [17] presents a state diagram-based specification of a nontrivial system. His notation is more precise and more general than most other versions of state diagrams, but it is also more complex and difficult to understand. It uses separate diagrams for nonterminal symbols and a global data structure, which can be set by arbitrary semantic-domain actions. Transitions can only be selected by examining values in this data structure, rather than the input tokens directly. Hence a transition involving receipt of a particular token is described by two transitions in Singer's notation—one to read it into the data structure and one to test the value just stored. While the two notations appear quite different, most aspects of Singer's notation can be mapped into that of the MUMPS specification.

Moran [13] provides a notation for describing the user's view of a computer system at several levels, from the overall tasks performed to individual key presses. This notation results in an unusually long and detailed specification. At the "Interaction Level," Moran's specification can be mapped onto a state diagram. His notation does not contain a state diagram representation of the Interaction Level of the user interface, but it does record a number of properties such a diagram would have. These properties are sufficient to generate a state diagram specification or (in a specification where only a few properties are specified) a set of diagrams.

EXAMPLES OF SPECIFICATIONS

To illustrate the use of some specification techniques, two commands from an hypothetical military message system are specified here. The ***Login*** command prompts the user to enter his or her name. If it does not recognize that name, it asks the user to reenter it, until he enters a valid name. Then, the system requests a password; if the password entered is incorrect, the user gets one more try to

[1] This must be done with care in an interactive system. Even though a deterministic automaton with backtracking can execute a nonrecursive nondeterministic diagram, in an interactive system it is not possible to backtrack over a path that has already generated output to the user.



Login

(1) resp: "Enter name"

(2) cond: not EXISTS_USER($USER)
resp: "Incorrect user name--reenter it"

(3) cond: EXISTS_USER($USER)
resp: "Enter password"

(4) cond: $PASSWORD=GETPASSWD_USER($USER)
resp: "Enter security level"

(5) cond: $PASSWORD≠GETPASSWD_USER($USER)
resp: "Incorrect password--reenter it"

(6) cond: $PASSWORD=GETPASSWD_USER($USER)
resp: "Enter security level"

(7) cond: $PASSWORD≠GETPASSWD_USER($USER)
resp: "Incorrect password--start again"

(8) cond: $SECLEVEL>GETCLEARANCE_USER($USER)
resp: "Security level too high--reenter it"

(9) cond: $SECLEVEL≤GETCLEARANCE_USER($USER)
act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL)

(10) resp: "Your security level is Unclassified"
act: CREATE_SESSION($USER,$PASSWORD,Unclassified)

FIGURE 2. State Diagram Specification of the "Login" Command.

enter a correct one and proceed; otherwise, he must begin the whole command again.[2] Next, the system requests a security level for the session, which must be no higher than the user's security clearance. If he enters a level that is too high, he is prompted to reenter it, until he enters an appropriate level. If he does not enter an appropriate security level, he is given the default level, ***Unclassified.***

The ***Reply*** command permits a user to send a reply to a message he has received. The user can give an optional input indicating to which message he wants to reply; otherwise, the default is ***CurrentMsg.*** He then enters the text of his reply. Following this, he can enter some optional lists containing additional addressees to which he wants this reply to be sent (in addition to those on the distribution list of the message to which he is replying). Each of these lists consists of the word "To" or "Cc" (depending on how the reply should be addressed to these people) followed by one or more addressees.

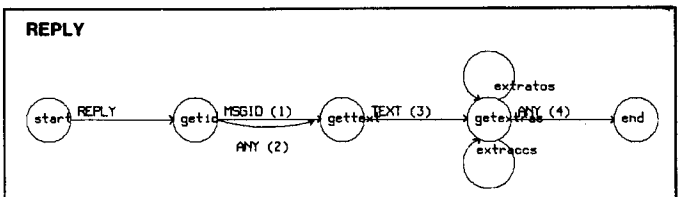
In Figure 2, the ***Login*** command is specified using state transition diagrams; the ***Reply*** command is specified in Figure 3. The notation follows widely used conventions.

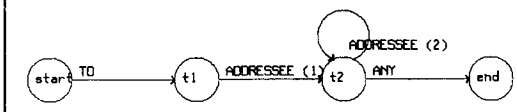
[2] In the specifications of this command given below, this counting is peformed using an extra state or an extra BNF rule. This approach is clear for the simple command here, but it would obviously be inconvenient if the system allowed the user a maximum of *100* tries to enter his password or if the limit were dependent on some other stored data. In such a case, the action and condition features of either notation, described below, would be used. The state transition or BNF rule corresponding to entry of an incorrect password would have an action that increments a counter variable, and that counter would then be tested in a condition associated with a subsequent state transition or BNF rule.

Each state is represented by a circle. The start and end states are so named inside the circles. Each transition between two states is shown as a directed arc. It is labeled with the name of an input token, in capital letters, plus, in some cases, a footnote containing Boolean conditions, system responses, and actions. A given state transition will occur if the input token is received and the condition is satisfied; when the transition occurs, the system displays the response and performs the action.

Instead of an input token, a transition may be labeled with the name of another diagram (in lower case). Such a transition will be made if the named diagram is traversed successfully at this point in the input. This notation permits breaking the specification up for clarity; otherwise, the text of the called diagram could simply have been inserted at this point in the calling diagram (provided no recursive calls are made).

In the actions, procedure names in upper case denote ICL statements, but their specific meaning is not material to this discussion. Some simple programming languagelike facilities, such as assignment and comparison, are assumed here, although the corresponding tasks could have been handled by defining some additional ICL statements. A token name preceded by a dollar sign stands for the





Reply

(1) resp: "Enter text field"
act: replyid:=REPLY_MSG($MSGID);
replybuf:=OPENFOREDIT_MSG(replyid)

(2) resp: "Enter text field"
act: replyid:=REPLY_MSG(CurrentMsg);
replybuf:=OPENFOREDIT_MSG(replyid)

(3) act: SETTEXT_MSG($TEXT,replybuf)

(4) act: UPDATE_MSG(replyid,replybuf); CLOSEEDIT_MSG(replyid)

extratos

(1) act: SETTO_MSG(replybuf,GETTO_MSG(replybuf)+$ADDRESSEE)

(2) act: SETTO_MSG(replybuf,GETTO_MSG(replybuf)+$ADDRESSEE)

extraccs

(1) act: SETCC_MSG(replybuf,GETCC_MSG(replybuf)+$ADDRESSEE)

(2) act: SETCC_MSG(replybuf,GETCC_MSG(replybuf)+$ADDRESSEE)

FIGURE 3. State Diagram Specification of the "Reply" Command.

```
Login

start:    LOGIN resp: "Enter name" →getu

getu:     USER cond: not EXISTS_USER($USER)
              resp: "Incorrect user name--reenter it" →getu
getu:     USER cond: EXISTS_USER($USER)
              resp: "Enter password" →getpw

getpw:    PASSWORD cond: $PASSWORD=GETPASSWD_USER($USER)
              resp: "Enter security level" →getsl
getpw:    PASSWORD cond: $PASSWORD≠GETPASSWD_USER($USER)
              resp: "Incorrect password--reenter it" →badpw

badpw:    PASSWORD cond: $PASSWORD=GETPASSWD_USER($USER)
              resp: "Enter security level" →getsl
badpw:    PASSWORD cond: $PASSWORD≠GETPASSWD_USER($USER)
              resp: "Incorrect password--start again" →start

getsl:    SECLEVEL cond: $SECLEVEL>GETCLEARANCE_USER($USER)
              resp: "Security level too high--reenter it" →getsl
getsl:    SECLEVEL cond: $SECLEVEL≤GETCLEARANCE_USER($USER)
              act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL) →end
getsl:    ANY resp: "Your security level is Unclassified"
              act: CREATE_SESSION($USER,$PASSWORD,Unclassified) →end
```

FIGURE 4. Text Representation of Figure 2.

value most recently read in for that input token. For example, $USER stands for the actual name the user typed.

The special token ANY is defined such that if no other transition can be made, the transition labeled with ANY is made, and the current input token is scanned again when the new state is reached. If the system reaches a state from which no transition can be made, given the current input, then there is an error in the input, and a transition would be made to an error-handling procedure. For clarity, such procedures have not been included in these examples. (Clearly, this cannot arise in a state from which there is a transition with the token ANY.)

The tokens themselves can be defined in a separate specification, which captures lower-level details of the user–computer interaction. For example, the token LOGIN

```
Reply

start:    REPLY →getid

getid:    MSGID resp: "Enter text field"
              act: replyid:=REPLY_MSG($MSGID);
              replybuf:=OPENFOREDIT_MSG(replyid) →gettext
getid:    ANY resp: "Enter text field"
              act: replyid:=REPLY_MSG(CurrentMsg);
              replybuf:=OPENFOREDIT_MSG(replyid) →gettext

gettext:  TEXT act: SETTEXT_MSG($TEXT,replybuf) →getextras

getextras: extratos →getextras
getextras: extraccs →getextras
getextras: ANY act: UPDATE_MSG(replyid,replybuf);
              CLOSEEDIT_MSG(replyid) →end


extratos

start:    TO →t1
t1:       ADDRESSEE act: SETTO_MSG(replybuf,
              GETTO_MSG(replybuf)+$ADDRESSEE) →t2
t2:       ADDRESSEE act: SETTO_MSG(replybuf,
              GETTO_MSG(replybuf)+$ADDRESSEE) →t2
t2:       ANY →end


extraccs

start:    CC →c1
c1:       ADDRESSEE act: SETCC_MSG(replybuf,
              GETCC_MSG(replybuf)+$ADDRESSEE) →c2
c2:       ADDRESSEE act: SETCC_MSG(replybuf,
              GETCC_MSG(replybuf)+$ADDRESSEE) →c2
c2:       ANY →end
```

FIGURE 5. Text Representation of Figure 3.

could represent the typed string "Login," a function key, or a hit of a graphic input device on a menu display, without affecting the specification shown here. Similarly, the definition of TEXT would include a specification of the delimiter used to indicate the end of an input string.

Figures 4 and 5 show how the specifications can be represented in text form. This is often more convenient for computer input and output than the graphical diagrams. The text representation consists of a list of the transitions that comprise the diagram, each represented by a line of the form

$$s1\text{: INP resp: "Hello"} \rightarrow s2$$

denoting a transition from state *s1* to state *s2*, which expects input token INP and displays response "Hello." Conditions or actions are specified in a way similar to the response. Instead of an input token, the name of another diagram could be given (in lower case), meaning that that diagram would be traversed, and, upon exit from it, a transition to state *s2* would be made. Other features of this notation are the same as for the state diagrams above.

This notation is directly translatable into that of the graphic state diagrams. In fact, Figures 4 and 5 were the input to a program that produced Figures 2 and 3 automatically.

```
Login ::=        badpw* goodpw {resp: "Enter security level"} getseclevel

badpw ::=        loguser onetry PASSWORD
                     {cond: $PASSWORD≠GETPASSWD_USER($USER)
                     resp: "Incorrect password--start again"}

goodpw ::=       loguser PASSWORD
                     {cond: $PASSWORD=GETPASSWD_USER($USER)}
|                loguser onetry PASSWORD
                     {cond: $PASSWORD=GETPASSWD_USER($USER)}

loguser ::=      LOGIN {resp: "Enter name"}
                     getuser {resp: "Enter password"}

getuser ::=      baduser* USER {cond: EXISTS_USER($USER)}

baduser ::=      USER {cond: not EXISTS_USER($USER)
                     resp: "Incorrect user name--reenter it"}

onetry ::=       PASSWORD
                     {cond: $PASSWORD≠GETPASSWD_USER($USER)
                     resp: "Incorrect password--reenter it"}

getseclevel ::=  badsl* {resp: "Your security level is Unclassified"
                     act: CREATE_SESSION($USER,$PASSWORD,Unclassified)}
|                badsl* SECLEVEL
                     {cond: $SECLEVEL≤GETCLEARANCE_USER($USER)
                     act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL)}

badsl ::=        SECLEVEL
                     {cond: $SECLEVEL>GETCLEARANCE_USER($USER)
                     resp: "Security level too high--reenter it"}
```

FIGURE 6. BNF Specification of the "Login" Command.

```
Reply ::=        REPLY getid {resp: "Enter text field"
                     act: replybuf:=OPENFOREDIT_MSG(replyid)}
                     TEXT {act: SETTEXT_MSG($TEXT,replybuf)}
                     extras* {act: UPDATE_MSG(replyid,replybuf);
                     CLOSEEDIT_MSG(replyid)}

getid ::=        MSGID {act: replyid:=REPLY_MSG($MSGID)}
|                NULL {act: replyid:=REPLY_MSG(CurrentMsg)}

extras ::=       extratos | extrraccs

extratos ::=     TO toaddressee toaddressee*
toaddressee ::=  ADDRESSEE {act: SETTO_MSG(replybuf,
                     GETTO_MSG(replybuf)+$ADDRESSEE)}

extraccs ::=     CC ccaddressee ccaddressee*
ccaddressee ::=  ADDRESSEE {act: SETCC_MSG(replybuf,
                     GETCC_MSG(replybuf)+$ADDRESSEE)}
```

FIGURE 7. BNF Specification of the "Reply" Command.

Figures 6 and 7 show the same commands in BNF notation. Lower case names denote nonterminal symbols, which are subsequently defined in terms of terminal symbols. Upper case names are terminal symbols, which would be defined in a separate, lower-level specification. Some definition rules are annotated with Boolean conditions, system responses, or actions, all placed in braces. If a rule contains a condition, that condition must be true at the point in the input stream corresponding to its position in the rule for the rule to be matched. When a rule is matched, the system will display the response and perform the action, if any are given.

The special token NULL represents no input. A token or nonterminal name followed by an asterisk stands for "zero or more instances of" that symbol. The other conventions used in the actions are the same as those for the state diagrams above.

The specifications in this notation could have been produced automatically from those above, but the resulting specifications would be difficult to read, as discussed below. Figures 2 and 3 were translated by hand to produce Figures 6 and 7, and several nonterminals thought to be helpful in understanding the specification were introduced in the process.

CONCLUSIONS

From examining these and other examples, one can observe that, while the techniques based on BNF and those based on state transition diagrams are formally equivalent, their surface differences have an important effect on the comprehensibility of the specifications. In particular, notations based on state transition diagrams explicitly contain the concept of a state and the transition rules associated with it, while this is implicit in BNF-based notations. Since the concept of state is important in representing sequence in the behavior of an interactive system, state diagrams are preferable to BNF in this regard.

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 into a new technique. The state diagrams shown above represent such a synthesis.

While the text representations of the state diagrams are somewhat more difficult to read than the graphical ones, they are a more convenient form of computer input. They do contain sufficient information to generate the graphic diagrams automatically (as was done here) and to drive a simulator of the user interface (which has recently been built).

In either state diagram or BNF notation, the judicious use and choice of meaningful nonterminal symbols is important to the overall clarity of the specification, sometimes more so than the choice of notation. The principal difference between the two types of notations in this regard is that a BNF-based specification with very few nonterminals (with respect to the complexity of the system) is generally more difficult to understand than the corresponding state diagram. Thus a direct translation of a typical BNF specification into state diagram notation is likely to contain many very simple diagrams; while a typical state diagram translated into BNF will contain only a few, very complicated rules. BNF, then, requires more nonterminals to make it readable.

A synthesis of the features of several state diagram-based notations was thus selected to specify the user interface for a prototype military message system. The explicit description of states in this notation makes the sequence of actions clearer than in BNF. In addition, some of the states correspond to users' own notions of what a system does ("text entry" state, "logged-out" state). The state diagram examples show how a portion of the User Agent can be specified in this manner. An interpreter has been developed to take such a specification directly and execute the specified user interface, issuing ICL commands to the rest of the message system.

Acknowledgments This work has benefited from discussions with my colleagues on the Military Message System project: M. Cornwell, C. Heitmeyer, and C. Landwehr, and with L. Chmura.

REFERENCES

1. Conway, M.E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396–408.
2. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976, pp. 209–217.
3. Embley, D.W. Empirical and formal language design applied to a unified control construct for interactive computing. *Int. J. Man-Machine Studies 10*, 2 (March 1978), 197–216.
4. Foley, J.D. and Wallace, V.L. The art of graphic man-machine conversation. *Proceedings of the IEEE 62*, 4 (April 1974), 462–471.
5. Hanau, P.R. and Lenorovitz, D.R. Prototyping and simulation tools for user/computer dialogue design. *Computer Graphics 14*, 3 (July 1980), 271–278.
6. Heitmeyer, C.L. and Wilson, S.H. Military message systems: Current status and future directions. *IEEE Transactions on Communications COM-28*, 9 (Sept. 1980), 1645–1654.
7. Heitmeyer, C.L. An intermediate command language (ICL) for the family of military message systems. Technical Memorandum 7590-450:CH:ch, Naval Research Laboratory, Washington, D.C., 13 Nov. 1981.
8. Heitmeyer, C.L., Landwehr, C.E., and Cornwell, M.R. The use of quick prototypes in the military message system project ACM SIGSOFT. *Software Engineering Notes 7*, 5 (Dec. 1982).
9. Hueras, J.F. A formalization of syntax diagrams as k-deterministic language recognizers. M.S. thesis, Computer Science Dept., Univ. California, Irvine, 1978.
10. Jacob, R.J.K. Survey and examples of specification techniques for user interfaces. NRL Report, Naval Research Laboratory, Washington, D.C., (To appear).
11. Johnson, S.C. Language development tools on the Unix system. *IEEE Computer 13*, 8 (Aug. 1980), 16–21.
12. MUMPS Development Committee. MUMPS language standard. American National Standards Institute, New York, 1977.
13. Moran, T.P. The command language grammar: A representation for the user interface of interactive computer systems. *Int. J. Man-Machine Studies 15*, 1 (July 1981), 3–50.
14. Parnas, D.L. On the use of transition diagrams in the design of a user interface for an interactive computer system. *Proc. 24th Nat'l ACM Conference* (1969), 379–385.
15. Reisner, P. Formal grammar and human factors design of an interactive graphics system. *IEEE Trans. Software Eng. SE-7*, 2 (March 1981), 229–240.
16. Shneiderman, B. Multi-party grammars and related features for defining interactive systems. *IEEE Trans. Systems, Man, and Cybernetics SMC-12*, 2 (March 1981), 148–154.
17. Singer, A. Formal methods and human factors in the design of interactive languages. Ph.D. dissertation, Computer and Information Science Dept., Univ. Massachusetts, 1979.
18. Wilson, S.H., Kallander, J.W., Thomas, N.M., III, Klitzkie, L.C., and Bunch, J.R. Jr. MME quick look report. Memorandum Report 3992, Naval Research Laboratory, Washington, D.C., 1979.
19. Woods, W.A. Transition network grammars for natural language analysis. *Comm. ACM 13*, 10 (Oct. 1970), 591–606.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Human Factors, Languages, Design

Additional Key Words and Phrases: interactive languages, specification languages, state transition diagrams, BNF, user agent

Received 3/82; revised and accepted 12/82