USING FORMAL SPECIFICATIONS IN THE DESIGN OF A HUMAN-COMPUTER INTERFACE

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

INTRODUCTION

Formal and semiformal specification techniques have been applied to many aspects of software development. Their value is that they permit a designer to describe precisely the external behavior of a system without specifying its internal implementation. In the Military Message Systems (MMS) Project at the Naval Research Laboratory, the external behavior of a family of message systems has been described using semiformal specification techniques [5,6]. Currently, the human-computer interfaces for such systems are being specified using similar techniques. These specifications will be used in the construction of a prototype military message system.

The design of the user interface for a military message system has 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 [16].

This paper surveys specification techniques that can be applied to human-computer interfaces, provides examples of specifications, and presents some conclusions drawn from the author's experience using the techniques for specifying the user interface of the message system.

THE MILITARY MESSAGE SYSTEM

Each member of the MMS family consists of several components. Two are of interest here: the User Agent and the Data Manager. The user communicates 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. For most changes to the user interface, 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. 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 [13] and Embley [2] do) as well as from a prototype (as Hanau and Lenorovitz [4] 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, _i.e._, 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. What is needed is a high-level model that describes the operations that the system performs. Then, the user interface specification describes the user interface in terms of the model, while the internal details of the model are 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 it is formally described in a separate specification [6]. The user interface specification, then, needs only to describe the syntax of the UCL with a language specification technique and the semantics of the UCL with 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.

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 [7]. 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. In an interactive language, the computer may take actions and produce outputs at any point in a dialog. Hence, a specification for such a language must capture not only the system actions and outputs but also their timing.

Most specifications for both static and interactive languages have been based on one of two formal models: Backus-Naur Form (BNF) [13] and state transition diagrams [12]. 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, system actions and their timing.

BNF

For BNF, the necessary modification consists of associating an action with each grammar rule. Whenever that rule applies to the input language stream (so 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 [13] provides an example of how BNF can be used to describe a user interface. Unlike several other published specifications, this one specifies a nontrivial, real-world system. It does leave out the system actions and responses, however, since Reisner did not need them for her purposes. She uses the BNF specifications of two systems to predict differences in the performance of their users. More complex or inconsistent BNF rules lead to predictions of user errors. Several predictions are then verified empirically.

Schneiderman [14] also examines the use of BNF for describing interactive user interfaces and proposes a modified form of BNF in which each nonterminal symbol is associated with either the computer or the user. This type of grammar can be mapped into a conventional state transition diagram (with the exception of one rarely-occurring nondeterministic case).

A BNF specification can also be used as input to a compiler-compiler, such as that described in [9]. Given a specification in which an executable action is associated with each BNF rule, such a program can automatically construct a prototype of the system being specified.

One general problem that arises with BNF-based techniques is that it is sometimes difficult to specify exactly when something occurs (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 dialog. Often, it requires the introduction of many 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 time sequence is explicit in a state diagram (while it is implicit in BNF), the former is more suited to specifying the times when 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 [17] also describes a notation, 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.

Both investigators 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. This makes it possible to divide complex diagrams into more manageable pieces.

Parnas [12] 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. The paper contains some very simple examples but does not address how the scheme would be extended for real-world systems.

Foley and Wallace [3] 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 [10] 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 description of the rules for interpreting them (since their use can otherwise require a nondeterministic automaton). The actions associated with the transitions in this specification comprise a complete specification of the semantics of the MUMPS language.

Singer [15] 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 is set by arbitrary semantic-domain actions. Transitions are then selected by examining values in this data structure, rather than the input tokens directly. While the two notations appear quite different, most aspects of Singer's can be mapped into that of the MUMPS specification.

Moran [11] 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 cases 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 re-enter 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 enter a correct one and proceed; otherwise, he must begin the whole command again. 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 re-enter 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.

State Diagram

In Figure 1, the "Login" command is specified using state transition diagrams; the "Reply" command is specified in Figure 2. The notation used follows widely used conventions. 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 labeled, directed arc. The arc 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 if 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 one assumes that the diagrams are deterministic).

In the actions, function names in upper case denote ICL functions, but their specific meaning is not material to this discussion. A token name preceded by a dollar sign stands for the value most recently read in for that input token. (*E.g.*, $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" 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.

Text Representation of State Diagram

Figures 3 and 4 show how the specifications above 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:  INP  resp: "Hello"  ->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.

BNF

Figures 5 and 6 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 lower-level specification. Some definition rules are annotated with Boolean conditions, system responses, or actions, all placed in brackets. 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.

CONCLUSIONS

From examining these and other examples [8], one can observe that, while 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 it is implicit in BNF-based notations. Since this concept 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 and also to drive a fairly straightforward simulator of a user interface.

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, often more so than the choice of notation. The principal difference between the two types of notations 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 is being used to specify the user interface for the 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 above show how a portion of the User Agent can be specified in this manner. The specification can then be used to produce a system that implements the specified user interface and issues ICL commands to the rest of the message system.

REFERENCES

1. Conway, M.E. Design of a Separable Transition-Diagram Compiler. Comm. ACM, 1963, 6, 396-408.

2. Embley, D.W. Empirical and Formal Language Design Applied to a Unified Control Construct for Interactive Computing. Int. J. Man-Machine Studies, 1978, 6, 197-216.

3. Foley, J.D. and Wallace, V.L. The Art of Graphic Man-Machine Conversation. Proc. IEEE, 1974, 62, 462-471.

4. Hanau, P.R. and Lenorovitz, D.R. Prototyping and Simulation Tools for User/Computer Dialogue

Design. Proc. ACM SIGGRAPH, 1980, 62, 462-471.

5. Heitmeyer, C.L. and Wilson, S.H. Military Message Systems: Current Status and Future Directions. IEEE Transactions on Communications, 1980, COM-28, 1645-1654.

6. Heitmeyer, C.L. An Intermediate Command Language (ICL) for the Family of Military Message Systems. Technical Memorandum 7590-450:CH:ch, Naval Research Laboratory, 13 November 1981.

7. Jacob, R.J.K. Survey of Specification Techniques for User Interfaces. Technical Memorandum 7590-303:RJ:rj, Naval Research Laboratory, 21 August 1981.

8. Jacob, R.J.K. Examples of Specifications of User Interfaces. Technical Memorandum 7590-008:RJ:rj, Naval Research Laboratory, 6 January 1982.

9. Johnson, S.C. Language Development Tools on the Unix System. Computer, 1980, 13, 16-21.

10. MUMPS Development Committee. MUMPS Language Standard. New York: American National Standards Institute, 1977.

11. Moran, T.P. The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems. Int. J. Man-Machine Studies, 1981, 15, 3-50.

12. Parnas, D.L. On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System. Proc. 24th National ACM Conference, 1969, 15, 379-385.

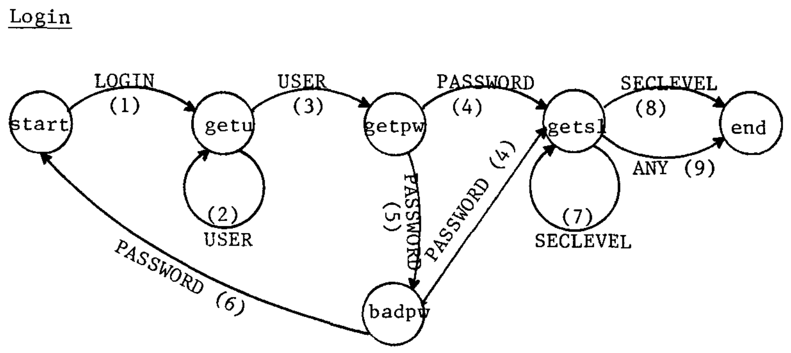
13. Reisner, P. Formal Grammar and Human Factors Design of an Interactive Graphics System. IEEE Transactions on Software Engineering, 1981, SE-7, 229-240.

14. Schneiderman, B. Multi-Party Grammars and Related Features for Defining Interactive Systems. IEEE Transactions on Systems, Man, and Cybernetics, 1981, SE-7, 229-240.

15. Singer, A. Formal Methods and Human Factors in the Design of Interactive Languages. Ph.D. dissertation, Computer and Information Science, Univ. Massachusetts, September 1979.

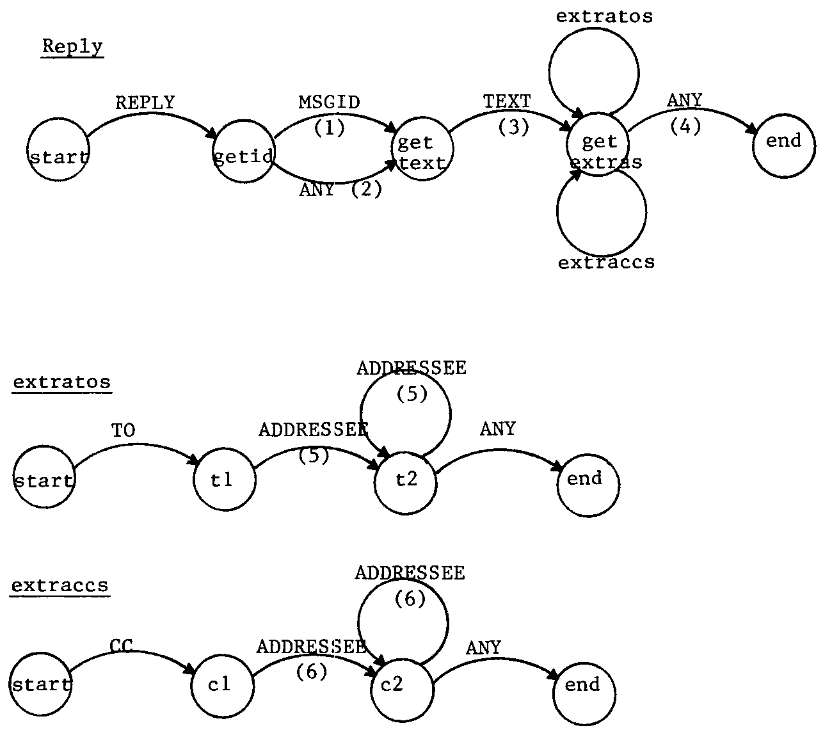
16. Wilson, S.H., Kallander, J.W., III, N.M. Thomas, Klitzkie, L.C., and Bunch, J.R. Jr. MME Quick Look Report. Memorandum Report 3992, Naval Research Laboratory, 1979.

17. Woods, W.A. Transition Network Grammars for Natural Language Analysis. Comm. ACM, 1970, 13, 591-606.



```
(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: "Incorrect password--start again"
(7)   cond: $SECLEVEL>GETCLEARANCE_USER($USER)  resp: "Security level too high--reenter it"
(8)   cond: $SECLEVEL<=GETCLEARANCE_USER($USER)  act: CREATE_SESSION($USER,$PASSWORD,$SECLEVEL)
(9)   resp: "Your security level is Unclassified"  act: CREATE_SESSION($USER,$PASSWORD,Unclassified)
```

Figure 1. State Diagram Specification of the "Login" Command



```
(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)
(5)   act: SETTO_MSG(replybuf,GETTO_MSG(replybuf)+$ADDRESSEE)
(6)   act: SETCC_MSG(replybuf,GETCC_MSG(replybuf)+$ADDRESSEE)
```

Figure 2. State Diagram Specification of the "Reply" Command

```
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 3. Text Representation of Figure 1

```
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 4. Text Representation of Figure 2

```
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 5. 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 6. BNF Specification of the "Reply" Command