To appear in:
Advances in Human-Computer Interaction,
ed. H.R. Hartson, Ablex Publishing Co., 1984.

Chapter ___

An Executable Specification Technique for Describing Human-Computer Interaction

Robert J.K. Jacob
Naval Research Laboratory
Washington, D.C. 20375

It is useful to be able to write a precise specification of the user interface of a computer system before building it, because the interface designer can thereby describe and study a variety of possible user interfaces without actually having to code them. The specifications can be checked for certain undesirable properties of the user interface, such as almost-alike states [21], interactive deadlock [4], and character-level ambiguity [25]. Human performance models can also be applied to the specifications to obtain information about the user interfaces they describe without actually building them [2, 22].

It is still more useful if a prototype or mockup of the user interface of the proposed system can be constructed directly from the specification. 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. Problems with the proposed user interface can then be identified early in the design process, when they are relatively easy to fix.

This chapter describes a specification technique for user-computer

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

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 chapter describes the specification technique by means of a detailed example, which serves to introduce the notation. Then, it discusses the reasoning behind this specification technique. 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. Some additional considerations are also discussed, including problems that arise when realizing a nondeterministic specification with a deterministic interpreter. Extensions of the technique for formally proving properties of the user interface and for describing concurrent dialogues are also discussed.

Description of the Specification Technique

Overview

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 therefore particularly suitable for describing interactive human-computer interfaces because they represent time sequence explicitly, in contrast to BNF, in particular, where sequence is implicit [16]. The state transition model has also been found useful in describing a user's mental model of an interactive computer system [4, 20], and programmers have been observed to prefer it to a BNF-based approach [9]. Several investigators have proposed different specification notations based on state transition diagrams [3, 5, 7, 19, 21, 24, 27]. Each provides some unique benefits but also has some disadvantages [14]. Only a few are sufficiently formal or complete to be executed directly, and fewer still have interpreters [5, 26].

The present technique is based on the state transition diagrams described in [16] 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 a user interface specification, thus providing a mockup of the specified system immediately. The notation supports a clear decomposition of the user interface description into semantic, syntactic, and lexical components [8] and a process of stepwise refinement of the syntactic specification that leads from an informal specification to a formal, executable one.

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. A second prototype with additional functions, including composing and sending messages and more sophisticated security operations, is presently being constructed in this same manner. In addition, an interactive graphic user interface for an expert system has been built and successfully demonstrated using this method.

The functions of the first prototype message system are described briefly, in order to clarify the examples in the rest of the chapter. This message system is much like a conventional electronic mail system, except that each message (actually, each field of each message), each file, and each user terminal has a security classification. The user first logs in to the system, as in a conventional time-sharing system, except that a security level is also entered. The user can then display messages, create files to hold them, move or copy messages among the files, and delete or un-delete messages and files. All these operations must be performed within the constraints of the security rules. For example, a user is not permitted to store a *SECRET* message in an *UNCLASSIFIED* file.

An Example Specification

To describe the specification technique, consider a section of the syntax portion of the executable specification of the first prototype message system, as shown in Figure 1. (The full executable specification is given in [15].) 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* 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**), as described below
- An action [27], which may manipulate external variables and which will be executed if this transition is taken (no examples appear in Figure 1)
- A condition [27], which may make arbitrary tests on the external variables and must be true for this transition to be taken (no examples in Figure 1).

Nonterminals. A nonterminal symbol is defined in a separate diagram, which may be called like a subroutine from a transition and must be traversed from start to end to complete that transition. If the called nonterminal has more than one exit state, the transition that calls it will also have several end states, one for each of the exits in that nonterminal. Each diagram has a name, which is the symbol by which it may be called as a nonterminal from another diagram.

*In fact, it is possible to combine several of the items listed on a single transition for compactness, but this feature has some drawbacks discussed below.

Description of tokens and nonterminals called by the **mms** *diagram.* 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. (The prompt actually consists of moving the cursor to the name field of the template.) **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 a valid combination of name, password, and security clearance or not; the **setup** nonterminal initializes a user session and shows incoming messages; the **cmd** nonterminal describes the user commands and is given below.

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 give a **quit** command to exit from the system entirely. If the login is successful, the user is prompted for a command, or he or she 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.

Remarks. It should be clear that the selection of certain transitions depends on user inputs, but others, such as those bearing output tokens, will always be "selected," regardless of the input. Hence, for the system to be realizable with a deterministic machine without backtracking, any state that has more than one transition leading from it must have all of its transitions associated with mutually exclusive inputs or conditions, rather than outputs or actions. (A more precise formulation of the conditions necessary for a deterministic realization is given below.)

Also, the diagram in Figure 1 describes only correct inputs. Since a

designer usually thinks in terms of correct inputs first, this corresponds to a particular stage of the stepwise refinement process discussed below. The next stage would add additional transitions to those states marked with plus (+) signs to take care of unexpected inputs.

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 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, actions, and conditions are expressed in this form appear in Figures 2 and 4. The use of various typefaces in the printed version is incidental; this notation can be parsed without regard to the typefaces. The plus (+) signs at the beginnings of lines denote "user-visible" states, discussed below.

Additional nonterminals. Figures 3 and 4 shows more of the specification of the prototype message system: a portion of the diagram for the **cmd** nonterminal, which was called from Figure 2 (with only three of its individual commands shown because the rest are repetitive); the nonterminal that obtains the arguments to one of the commands and executes it (**copy_mc**); and the **scroll** nonterminal, which permits the user to scroll through output that does not fit in the screen window.

The main command loop: **cmd** *nonterminal.* The **cmd** diagram obtains a command name, clears the error window if the command name was valid, calls a 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 (if the

command generated no output to the user, as is generally the case with the **copy_mc** command), to display output to the user using the **scroll** nonterminal and then return (if the command generated output, as with the **display_msg** command), or to display an error message (**oCMDERR**) and then return (if command execution resulted in an error).

A command: **copy_mc**. The **copy_mc** diagram prompts for (**oMSGNUM**) and then accepts (**iMSGNUM**) a message number and then a file name. It then executes the requested operation, copying the designated message from the user's current message file to the designated file. Finally, it returns through one of two possible exits, depending on whether an error occurred in the copy operation.

Display output: **scroll**. The **scroll** diagram displays as much of **GLOBAL_dispobj** as will fit in its window. **GLOBAL_dispobj** is a variable that contains some object to be displayed on the screen; its value was set by a function call in a previous display command and is available to the specification of the output tokens **oSCROLLSCREEN** and **oSCROLLITEM**. If the object does not fit, the user is asked to press a key to display the next screen-full, scroll to the next single item, or cease.

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 or mixed case represent conventional operations on variables (like **equal** and **assign**); many of these would be provided by a typical programming language, but, to keep the interpreter language simple, they are treated syntactically as external functions. The value received by an input token (such as the actual number

entered for the token **iMSGNUM**) is placed 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 are character strings of arbitrary length and their names all have global scope.*

Several exit states. A diagram may have more than one exit state. In such a case, the header line would list them all:

login → *(ok, bad)*

Any transition that called this diagram must then list a similar number of end states:

getlog: login → *(setup, badlog)*

This means that if the call to **login** exits by reaching the state **ok**, then this transition in the calling diagram goes from **getlog** to **setup**; if the call to **login** exits through state **bad**, this transition goes from **getlog** to **badlog**.

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 [7] 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

*A naming convention is used to provide simple variable scoping. Variables local to one diagram have the name of the diagram prepended to their names; all other variables are global and have **GLOBAL** prepended as a reminder. In addition, variables whose names begin with **vi**, **vo**, or **vl** are always global, since they are used to pass data between syntactic and lexical specifications.

notation for specifying each of them separately to an interpreter.

The three levels represent increasing amounts of specificity and detail and decreasing abstraction, from the semantic level to the lexical level. They are defined by Foley and van Dam [8]:

- The *semantic* level describes the functions performed by the system. The semantic design tells what information is needed to perform each function and the result of performing it. It defines "meanings," rather than "forms" or "sequences," which are left to the lower levels.
- The *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. Tokens are the smallest units of meaning [with respect to the syntax of the dialogue] and cannot be further decomposed without losing this meaning.
- The *lexical* level determines how input and output tokens are actually formed from the primitive hardware operations (*lexemes*). It represents the binding of hardware actions to the hardware-independent tokens of the input and output languages.

The specification of each of these levels is discussed below. Each specification level is introduced, and then the notation used for that level is described.

The Semantic Level

In the specification language, the semantic level is concerned with the manipulation of internal variables; no actual input or output operations are described at this level, although the manipulation of values read in as inputs and the generation of values to be displayed as outputs are described. The semantic-level specification consists of descriptions of functions that operate on

these internal data, that is, the function parameters, their types, and the effects of the functions. 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 pseudo-code 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. The LISP interpreter runs as a separate process and provides the semantic operations upon request from the process running the state diagram interpreter. The semantic 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 [12]. The operations provided by that machine are described in a separate specification [11]. In this way, the details of the semantic-level specification of the user interface are partitioned from the syntactic- and lexical-level specifications and treated separately [16]. Dialogue independence [28] is thereby provided, since the dialogue-related aspects of the system (syntactic and lexical levels) are specified separately from the rest of the system (semantic level).

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 or she 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 4 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 [23] and Singer's version of state transition diagram notation [24] 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. Such notations typically describe user input on state transitions, but then they 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 problem is obscured because the systems for which specification techniques are demonstrated often have a trivial or highly regular output syntax, typically of the form: *after every state transition, make the display or window consistent with an internal data base.* This describes many screen editors and graphics systems and is frequently a good system design; but, if a specification language is designed with such a simple system in mind, one is

*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.

encouraged to ignore output syntax. Systems may have a more complex output syntax, and the specification technique should be able to describe it. As a simple example, in the message system prototype, if the user deletes a message while the list of messages is on the screen, that list is not updated until the user explicitly displays it again. This is an important (though perhaps not desirable) characteristic of the output syntax of the user interface, and it should be possible to specify it clearly at the syntactic level.

This problem is addressed here by making the syntactic-level specification describe output syntax in terms of output tokens in the same way that it has previously been used to describe input syntax in terms of input tokens. Since syntax is concerned with the names and sequences of input tokens, it is extended to include a description of the names and sequences of output tokens in the same state diagram notation. The concept of a token for output is, by analogy to input [8], a unit whose internal structure has no meaning with respect to this dialogue.*

Introducing output tokens (and actions) on state transitions on the same footing as input tokens increases the number of states in a diagram, making it more difficult to read. For some designs, this is unavoidable, since the alternative approach of combining these features into a single transition limits the generality of the language. The problem is mitigated by the process of stepwise refinement discussed below, which leads from an incomplete specification, containing input syntax plus an approximate description of system responses (that is, every transition has an input token and, optionally, a collection of outputs and actions), to a precise specification, in which the

*This is entirely true for lexical and syntactic output tokens, but, for semantic output, the notion is stretched somewhat in that the internal contents of the tokens do have meaning. The actual contents are described in the semantic specification, but, as with all other outputs, the syntactic specification continues to describe *when* the token occurs, and the lexical specification continues to describe *how* (in what format) it is presented.

additional states necessary to treat input and output separately and symmetrically are introduced. The special prompt notation described below can also frequently be used to reduce this proliferation of states.

The Lexical Level

While tokens are the smallest units of meaning with respect to the syntax of the dialogue, lexemes are the actual hardware input and output operations that comprise the tokens. The lexical-level specification describes the physical embodiment of each of the input and output tokens, by giving the lexemes that constitute each token. The lexical-level specification identifies the devices, display windows, and positions with which each token is associated and the primitive hardware operations that constitute them. All information about the organization of a display into areas and the assignment of input and output tasks to hardware devices is confined to this level. Some examples of lexical specifications in state diagram notation taken from the message system are given in Figures 5 and 6.

For an input token, the specification gives the sequence of primitive input lexemes (for example, key presses) and the device for each lexeme by which the token is entered as well as any lexical output that is produced. Such output constitutes prompts and acknowledgments for the individual lexemes that make up a token; most often, it consists of echoes.

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). For semantic output, the lexical specification gives the format in which the variable data should be displayed, but not the contents. For example, since all display objects are displayed in one of two formats in the

prototype message system (next full screen or next single item), there is an output token for each such *format* (**oSCROLLSCREEN** and **oSCROLLITEM**, respectively), even though they are used to display many different specific data items. For syntactic output (such as prompts), both the format and the contents (that is, the sequence of lexemes that constitute this token, which is usually constant) are specified at the lexical level.

The executable lexical-level specification is written in the same state transition diagram notation, avoiding the introduction of another notation and another interpreter. As shown in Figures 5 and 6, 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. In the lexical-level specification, 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; 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.

The lexical-level specification of an input token consists of a state diagram that calls lexemes, which are either individual hardware input actions (with names entirely in upper case, like **NEWLINE**) or else sub-diagrams that call those hardware actions (with names of the form **I** followed by a lexeme name in upper case, like **IULCHAR**, any upper- or lower-case alphabetic character). Lexical outputs (echoes) are given using the special output actions.

The lexical-level specification of an output token 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., **INAMEWIN**), 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 definitions of the output actions. The diagram is executable, like the other diagrams, but it is really just a linear sequence of lexemes and actions. The output task is divided so that most of the individual output functions perform small, easily-understood tasks; some built-in output functions are also provided in the interpreter to handle common situations.

Certain low-level lexical operations are cumbersome to describe in state transition diagram notation [10]. Tracking a mouse with a cursor, highlighting menu items as they are traversed, making font changes, or even echoing characters could best be described in some more intuitive fashion than a large state diagram with a very regular structure. Regardless of how they are represented, though, it is important that details such as these be captured at and confined to the lexical-level specification. They should not have any bearing on the specification of the syntax of the dialogue nor the use of the state diagram-based notation for that purpose.

Errors

Different kinds of errors are handled at the three different levels in the specification. The definitions, timings, and names of lexical errors (that is, incorrectly combining the lexemes that comprise a token) are given as state transitions in the lexical specification. The timings and names of syntactic errors (wrong token) and semantic errors are given in the syntactic specification, as are the definitions of syntactic errors. The definitions of what constitutes semantic errors are given in semantic-level functions that can be

called as conditions on state transitions in the syntax diagrams. For example, Figure 7 shows the syntactic-level diagram **copy_mc** (originally given in Figure 4) with simple transitions added to handle syntactic errors; and the lexical-level diagram **iFILENAME** (originally given in Figure 6) with transitions for lexical errors. The detection of a semantic error in the syntactic-level specification is seen in Figure 7 in the condition in **copy_mc** that tests the value of **voCMDERR**.

Distinguishing the Specification Levels

Despite the guidelines discussed above, there are not absolute rules to determine what information belongs at each of the three levels of the specification; some judgment is still required. For example, nothing has been said that would prevent one from putting everything interesting about the system into one action at the semantic level, and providing entirely uninteresting syntactic and lexical specifications as follows. The syntactic specification would be:

st:	iANYINPUT →*doit*
doit:	→*showit* **act:** processinput(*voDISPLAY, viANYINPUT);
showit:	oDISPLAY →*st*

and the lexical specification would define **iANYINPUT** as just that, and **oDISPLAY** to take the full display image in **voDISPLAY** and display it. Then, the semantic action **processinput(d,i)** examines input **i** and, based on its value and any relevant prior inputs, puts the appropriate new display image into **d**, including error messages, prompts, and the like in their correct positions. Thus the description of the syntactic and lexical design of the user interface has been moved out of the syntactic and lexical specifications and confined entirely to the semantic specification. The specification technique does not prevent this; judgment is needed to use its three levels more effectively, as in Figures 1

through 6.

Stepwise Refinement of the Syntax Specification

While the first aspect of the syntax to be designed is often the input, 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. In the early stages of the process, the specification comprises an abstract description of the user interface intended principally for people to read (although it can be executed). Later in the process, it becomes the complete executable specification that is input to the interpreter. The basic notation is the same at every stage.

User-visible states. Observe that every state in a syntax diagrams falls into one of two categories. In some states, the system is waiting for input from the user, for example, state **getlog** in Figure 2. These are the only states in which the user ever observes the system. There are also states in which the system is not waiting for user input, but is about to produce output, perform an action, or test a condition, for example, **promptlog** in Figure 2. From the user's point of view, such states are internal; the system never stops at them but proceeds directly to the next state that requires input. States of the first type are called *user-visible* and marked with plus (+) signs in the specification (for the aid of the designer; the distinction is immaterial to the interpreter). Initially the designer should focus attention on these user-visible states. The first two steps in the stepwise refinement process produce diagrams that contain only user-visible states. Such states also comprise all the places at which help, abort, and similar functions should be provided.

Step 1. The first step is a diagram of the input syntax only, with no actions or outputs. All states in this specification are user visible. The resulting

diagram resembles the syntax diagrams used to describe non-interactive programming languages [3, 14, 27]. Figures 8 and 9 show a portion of the **mms** diagram of Figure 2 (with the **login** sub-diagram expanded out) in this form, as well as examples of the next two steps.

Step 2. Next, informal descriptions of the actions and outputs are added to each transition. While a transition may have only a single input token or nonterminal, it may, in this step, have arbitrary sequences of actions, outputs, and conditions. This constitutes a description of the system in terms of transitions between user-visible states; each user action always causes the system to move from one user-visible state to another. On any one transition, the sequence of the actions, outputs, and conditions within that transition is not formally specified. It will be resolved subsequently, when internal states are introduced. At this step, the specification resembles the state diagrams most commonly used to describe interactive systems [8, 26].

Step 3. 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 makes the specification of the sequence of outputs, actions, and conditions formal. It also means that internal states are finally introduced into the specification, as shown in Figures 8 and 9.

Step 4. 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 4 all show syntactic-level specifications corresponding to this step.

Step 5. 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.

Figure 7 shows this step; a complete example is given in [15].

Unlike non-interactive languages, an interactive system generally accepts any user input and, if it is not what the designer intended, displays some error message. Thus a specification of an interactive system must say what it will do for all possible inputs, including those that are not in the input language as designed. This suggests the following completeness constraint on the specification: *At each user input state, a transition should be provided (either directly or, more likely, through a call to a sub-diagram) for every input that could physically be entered at that point.*

Describing errors, abort keys, help facilities, and the like are aspects of the specification for which the state transition diagram technique is least helpful, because the same features are applicable to many states, but not particularly revealing about each individual state. To improve readability, these transitions can be listed separately at the end of the text form of the diagram and suppressed from drawings. Figure 7 shows examples of how transitions are added to the state diagrams to handle errors.

There is also a provision for modifying the conceptually nondeterministic traversal of the diagrams to help in this step. One or more state transitions can be considered "error" transitions; such will be tried only after all other transitions from a given state have been tried unsuccessfully. Transitions listed after the statement **%error** in the text form of a diagram are considered error transitions (see Figure 7); they will be tried last and can be suppressed from the drawings. The statement **%noerror** permits regular transitions to be entered again, within each diagram.

Another feature helps in describing operations that are the same for many states. Instead of a single state, a list of states may be given as the start for any transition, meaning that this transition can be made from each listed state. For

example, in Figure 2, there might be an abort command that, from any input state, returns to the beginning of the message system login sequence:

getlog,getcmd: iABORT →*st*

The end state of any transition may also be given as **SAME**, meaning the transition returns to the state from which it started. For example, a standard help operation (perhaps consisting of a help command followed by some further dialogue) might be available at several states, always returning the user to the state from the help was requested:

getlog,getcmd: help_operation →*SAME*

To aid in the early stages of this process of stepwise refinement, actions referred to in a diagram may be designated as "informally specified" (by beginning them with an asterisk), and the interpreter will simply print their descriptions instead of trying to execute them. The interpreter may also be told to provide stubs for all missing sub-diagrams in a specification. Thus, the specification in its early, informal stages may be parsed for syntactic errors, drawn, and executed automatically. (Depending on which actions, conditions, or sub-diagrams are missing, the first version of the specification may not be interesting to execute.) It may then be transformed into a fully-executable formal description of the system by this process of stepwise refinement, and, at every step along the way, the resulting specification remains in the same notation and continues to be parsed, drawn, and executed by the same tools.

Thus, the notation here could be considered to comprise two languages. One is a human-readable abstract specification, analogous to pseudo-code, as seen in the first few stages of the refinement process. The other is the final executable specification, used as input to the interpreter, as seen in the last two stages. In its level of detail, the latter sometimes resembles a conventional programming language more than it does an abstract specification, but it is a

programming language into which it is particularly easy to translate the high-level specification, since the notation is identical. Thus the usual manual process of translating from specification to implementation consists of, at most, adding details, but never of translating between notations. The result is a process that is more like stepwise refinement or transformational implementation [1] of a specification, in which the same notation is used for all steps, and less like the conventional process of translating from specification to program. The diagram interpreter and drawing program support both "languages."

Additional Considerations

Prompts

A particular problem in specifying output syntax is the timing of prompt messages. If they have considered prompts at all, most previous techniques have associated a prompt with a state: whenever that state is reached, its prompt is displayed. This makes it difficult to describe a system that gives different prompts depending on how a state was reached (for example, a special prompt when returning to a state after entering an incorrect input and receiving an error message).

Another approach [5, 8] is to associate a type of input token with each state. That type then implies a prompt: whenever a particular type of input token is expected, its associated prompt is displayed. This gives an especially compact and well-structured notation, but again fails to handle the case where there may be different specific prompts for the same input type in different states or different paths to the same state. Creating many slightly different types of inputs with different prompts is a potential, but unhelpful solution, as Feldman and Kamran observe [6].

A more serious problem is that a system may, at some point, prompt for one type of input but be waiting for (or willing to accept) other types of input, such as a cancel or help command, perhaps entered with a different input device. For full generality, then, it is necessary to separate the prompt from the input and to indicate, on two separate state transitions, what the system prompts for and what it will actually accept, as seen in Figures 1 and 2. The internal details of the prompt itself and of the inputs themselves are still given in the lexical specification.

As an aid in reducing the resulting proliferation of states, the present interpreter does permit associating a prompt (an output token) with any state. Every time the state is reached, that output token will be produced. This is simply an abbreviation for one additional state and transition; if the design being described permits it, this allows the diagram to be somewhat more compact, but the more general form is always available.

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. The problem is that, in an interactive system, it is meaningless to backtrack over a path that has already generated output to the user. Unlike a compiler, which can remove a previously-made entry from its parse tree, this system cannot take back output the user has seen or heard. Hence a specification of a system with interactive outputs must be constrained by its writer so that it will never be necessary to backtrack over an output transition. (In general, the interpreter permits an arbitrary amount of backtracking as long as no output is involved.) The following constraint is sufficient: *Starting at each state at which there is a fork, the inputs that will cause the machine to reach transitions that produce outputs to the user must be disjoint from each other.*

That is, from any state, the same initial input cannot cause two different output transitions,* even if subsequent input would disambiguate them, because the interactive system still must take one of the two paths, produce its output, and sometimes have to backtrack to rescind its choice. This constraint is the dual of the requirement of handling all possible user inputs given above. That rule said that there should be at least one path for every input; this rule says that there should not be more than one path for the same input (if the paths produce any output). It follows that a sub-diagram called from a fork may not begin with an output.

For example, in state **getlog** in Figure 2, the called diagram **login** must not produce any output until it has decided that the input it received will be accepted (in this case, that it has the format of a login name and could not be part of the token **iQUIT**). If the input turned out to be part of **iQUIT**, it would be necessary to rescind the output produced by **login**. This is why the call to **login**

*even if they produce the same actual output. A smarter interpreter might be willing to backtrack over the most recent transition that produced output if it subsequently took another transition that produced the same output again. The interpreter would realize that the two canceled each other out.

in Figure 2 had to be preceded by the prompt **oLOGNAME** in the calling diagram, rather than putting the prompt at the beginning of the **login** diagram. In fact, this is an appropriate representation because, in state **getlog**, the system is prompting for one input but actually willing to accept either of two inputs, and the specification of this fact belongs at a level above the separate specifications of each of the individual inputs.

Singer [24] places a stronger constraint on his state diagrams that achieves the same purpose: No transition at a fork may accept input tokens; they must all test conditions instead. This means that, to read an input, there would be one transition with an input token leading to a second state with several transitions, each with a condition that examines the token just read in. The disadvantage of this approach is that the real description of the syntax of the language is hidden in the definitions of these conditions. The clarity and power of the conventional syntax diagram for describing syntax is lost. Feldman and Rogers [5] also restrict each state to accepting a single type of token; branches of a fork must be selected based on the value of the token, but the entire operation is accomplished in one transition rather than two. Zave [29] recognizes the same problem with these notations and solves it by introducing "filters."

A further restriction in the use of the state transition diagram notation is not strictly necessary but advisable for clarity. While the language and interpreter permit any state transition to have an input or output token, a condition, *and* an action, it is generally clearer to have only one of these per transition in the syntactic-level diagrams. This makes the specification independent of the order in which the interpreter accepts tokens, tests conditions, performs actions, and backtracks within each transition. For example, it could become necessary to back up over a called sub-diagram if the

condition on the calling transition turned out to be false, and this means actions in the sub-diagram may be performed and then rescinded under certain conditions.

Implementation

The present specification interpreter does not actually simulate a nondeterministic automaton. Instead, it follows the algorithm given by [19], in which, effectively, the only states to which the machine can backtrack are those from which a sub-diagram was called. That is, once a transition is selected, the only way it can be rescinded is to reach a non-exit state in the current diagram from which no further transitions are possible, given the current input. In that case, the entire current diagram is abandoned, and the diagram that called it may try another transition. This limitation of the extent of nondeterminism provided means that the syntax description may have to be reorganized in places. Items that have common prefixes and are called from the same state must either be turned into nonterminals or else have their common prefixes combined into a single transition. Because of the constraint given above to prevent backtracking over output, a specification of a system with interactive outputs will generally have to be organized in this fashion anyway, so the limited nondeterminism has surprisingly little practical effect.

For example, suppose that the command tokens **iCOPY_MC** and **iCREATE_MC** in diagram **cmd** in Figure 4, began with the same character. This could *not* be represented as:

getcn:	C →*getOPY*
getcn:	C →*getREATE*
getOPY:	O →*getPY*

getREATE: R →*getEATE*

Instead, either the common prefix *(C)* would have to be given in a single transition:

getcn: C →*getOPYorREATE*

getOPYorREATE: O →*getPY*

getOPYorREATE: R →*getEATE*

or else the two command names would have to be turned into nonterminals, as was done in Figure 4:

getcn: iCOPY_MC →*copy*

getcn: iCREATE_MC →*create*

With a genuinely nondeterministic automaton and recursive calls to subdiagrams (but without conditions, actions, or outputs), the present notation has been shown to be equivalent in expressive power to BNF [13]. Adding arbitrary functions for the conditions and actions gives it the power of a Turing Machine.

The specification interpreter is written in C and runs under Berkeley 4.1bsd or 4.2bsd UNIX on a VAX 11/780. 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 common front end is about 400 lines of code, the interpreter is another 700, and the drawing program is 500. Both the semantic functions and the output functions used by the lexical-level specification are coded in C (about 500 lines for the message system) and then linked with the interpreter. Device-independent facilities for full-screen text terminals and also graphical output devices are used by these functions.

The state diagram specifications themselves are simply text files. Recompilation is necessary only when the functions written in C are changed; if

the specification itself is changed, the interpreter can be run again directly. Instructions for using the system and a detailed example are provided in [15]; further implementation details are given in [17].

Extensions to the Technique

Security Proofs

In designing a secure message system, it is desirable to prove assertions about the security of the system formally. Such proofs are usually based on a formal specification of the system (with the proviso that the final software and hardware correctly implement the specification). This approach has not generally been used at the user interface level, but, if one had a formal specification of the user interface, it would be possible to provide proofs about the user interface. One might want to prove an assertion such as the following: *When the system is ready for the user to enter a confirmation of an operation that alters the security classification of a message, the old and new classifications must both be visible on the screen (to prevent the user from misunderstanding the meaning of his or her confirmation).*

Such an assertion could be proven by tracing the state transition diagram specification as follows: Call the state in which the confirmation is accepted **s**. Trace backward along all paths in the diagram (or classes of paths in case of loops) that lead to **s** until the output tokens for the old and new classifications are found. Now, examine the output actions in the lexical specifications of any tokens that occur along those paths and show that none of them is capable of obliterating the two classifications on the screen. Note that it is possible that the intervening actions are *capable* in general of obliterating the desired information, but in fact they will never do so in this diagram. This procedure will not in general be able to discover this; hence, it proves a condition that is always

sufficient, but not always necessary. In such a case, a successful proof would require a formal definition of the semantics of the lowest-level output actions, most likely in a different notation from those used here. However, most of those lexical actions are simple, atomic operations, and should be amenable either to simple formal description or to convincing informal argument. For simple cases, one could take advantage of the fact that windows are explicitly bound to output tokens in the state diagram specification itself, not the code of the actions. Thus, using only the state diagram, one could show that all intervening output tokens involve different windows that do not overlap those with the desired security information.

Several Windows

The simplest case involving the use of several windows occurs when a display is divided into windows, but there is only one context for user interaction. All user input is considered part of one unbroken dialogue, with no context switches. During the dialogue, output may appear in various windows for convenience, but, to the user, it is all part of a single dialogue. To describe this with the present technique, all output items would be bound to specific windows in the lexical-level specification. A simple way to show this is to create output lexemes that select each window, as shown in the message system specification. Every lexical-level diagram begins with a call to one of these lexemes, followed by lexemes or actions that write in the chosen window, thereby making the binding to windows explicit at the lexical level. For example, the lexical specification in Figures 5 and 6 shows that **oCMDERR** consists of text printed in the error window (**ERRWIN**), while the lexical echo for **iCOPY_MC** is presented in **CMDWIN**. Shneiderman [23] 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.

Concurrent Dialogues

A more interesting situation arises when there are, from the user's point of view, several dialogues taking place, and the user may switch among them from time to time. This is frequently (but need not be) realized with several display windows, one associated with each conversation, and is seen in many recent bitmap-display-and-mouse user interfaces. In particular, consider a simplified window manager of this general type [18]. It permits a user to create several display windows, conduct a dialogue with a different program in each, and change focus from one window to another at any time without losing his or her place in any of the dialogues. The user can also change the layout of the windows on the display at any point. Certain input actions are reserved for directives to the window manager itself for changing the focus of the conversation or the layout of the display. All other inputs are considered input to the "current" dialogue, as designated by the most recent command to the window manager.

There are, thus, two levels to the syntactic description of such a system. The top level describes the display-arranging commands of the window manager itself. Then, each program running in a window (such as an editor, debugger, shell command interpreter, or programming language interpreter) has its own syntax, described at a second level, with the additional proviso that the window manager commands can be entered at any time. Such commands will cause an immediate switch of context to the window manager itself and thence back to the current dialogue at the precise point at which it was left or else to a dialogue in another window.

One straightforward way to specify such a system is to describe the top-level syntax of the window manager commands and then simply indicate that, whenever the user enters any non-command character, it is sent to the

"current" program, which processes it and possibly displays some output in its window. The problem is that each of the programs in the windows is likely to have some internal syntactic structure, and that should be representable in a specification of the syntax of the user interface of the overall system.

Another straightforward approach is to describe the syntax of the entire system in a single state diagram, as if it were one dialogue encompassing the window manager commands and all the individual dialogues. Unfortunately this would require as many states as the cross product of the states of the individual dialogue diagrams and a considerable tangle of transitions; it would still shed little light on the user's view of the system as a collection of separate, concurrent dialogues.

Singer [24] introduced an approach for describing an interrupt handler that permitted a user to interrupt a dialogue, conduct a new dialogue, and then return to the original one. He constructed what might be called a "universal state diagram," which can execute other state diagrams, rather like a universal Turing machine. It can thus be used to save a state within one diagram, interrupt execution of that diagram, recall a state within a second diagram, and resume execution of the second diagram. The resulting specification is elegant, but still does not provide a particularly easy to understand description of the user's view of the overall system.

A new approach, using "co-diagrams" may provide a solution to this problem. Previously, a transition in one diagram was used to call another "sub-diagram" like a subroutine. Instead, a co-diagram call is like a coroutine. When a transition in one diagram (for example, diagram **a**) makes a co-diagram call to a second diagram (**b**), diagram **b** is entered at the state from which **b** itself last executed a co-diagram call, and it is traversed until it makes another co-diagram call. If, for example, it then called diagram **a**, **a** would be resumed at

the end state of the transition from which it had called **b**. Whenever a diagram is entered by a co-diagram call to it, it is resumed with its own stack of pending sub-diagram calls intact. That is, if it had made a sub-diagram call, it is resumed within that sub-diagram, and upon exit from that sub-diagram, it will return to the diagram that called the sub-diagram.

With this approach, the syntax for the window manager commands is described in one top-level diagram. Whenever an input that is not a window manager command is received, the top-level diagram makes a co-diagram call to the syntax diagram for the currently-selected individual dialogue. The syntax diagrams for the individual dialogues are conventional, except that every state from which escape to the window manager command level is permitted is preceded by a transition that makes a co-diagram call to the top-level window manager diagram. Whenever the lower-level dialogue is resumed after that call, it will then be ready for user input at the same point in its input syntax from which it was interrupted.

An analogous situation, but without the windows, is found in network virtual terminal programs, such as "telnet." There is a top-level syntax for commands to the local computer that cause it to make, break, or modify a connection to a remote computer and a lower-level syntax for the conversation with the remote computer itself, plus an escape to the top level. These could be described similarly by co-diagrams.

Conclusions

This chapter has presented a technique for specifying the user interface of an interactive computer system and described how it has been used to produce formal and executable specifications 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 the design process and the specification itself to be separated into the semantic, syntactic, and lexical levels, and it supports a process of stepwise refinement of the syntactic-level specification to aid the designer. An approach to proving security properties of a user interface using the specification was also described; and an extension to the technique for describing concurrent dialogues was given.

References

1. BALZER, R., "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering* **SE-7** pp. 3-22 (1981).

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

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

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

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

6. FELDMAN, M.B. AND KAMRAN, A., "Graphics Programming Independent of Interaction Techniques and Styles," *Computer Graphics* **17**(1) pp. 58-66 (1983).

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

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

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

10. HAYES, P.J., "Executable Interface Definitions using Form-Based Interface Abstractions," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1984).

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

12. HEITMEYER, C.L., LANDWEHR, C.E., AND CORNWELL, M.R., "The Use of Quick Prototypes in the Military Message Systems Project," *ACM SIGSOFT Software Engineering Notes* **7** pp. 85-87 (1982).

13. HUERAS, J.F., "A Formalization of Syntax Diagrams as k-Deterministic Language Recognizers," M.S. thesis, Computer Science Dept., Univ. California, Irvine (1978).

14. JACOB, R.J.K., "Survey and Examples of Specification Techniques for User Interfaces," NRL Report, Naval Research Laboratory, Washington, D.C. (1983).

15. JACOB, R.J.K., "Formal Specification of the User Interface of a Receive-only Secure Military Message System Prototype," Naval Research Laboratory Technical Memorandum 7590-203:RJ:rj (11 August 1983).

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

17. JACOB, R.J.K., "Installing and Using the User Interface Specification Tools," Naval Research Laboratory, Washington, D.C. (1984).

18. JACOB, R.J.K., "User-Level Window Managers for UNIX," *Proc. UniForum International Conference on UNIX*, (1984).

19. MUMPS DEVELOPMENT COMMITTEE,, *MUMPS Language Standard*, American National Standards Institute, New York (1977).

20. MORAN, T.P., "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.

21. 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*, pp. 379-385 (1969).

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

23. SHNEIDERMAN, B., "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).

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

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

26. WASSERMAN, A.I. AND SHEWMAKE, D.T., "The Role of Prototypes in the User Software Engineering (USE) Methodology," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1984).

27. WOODS, W.A., "Transition Network Grammars for Natural Language Analysis," *Comm. ACM* **13** pp. 591-606 (1970).

28. YUNTEN, T. AND HARTSON, H.R., "F-i-l-l i-n T-i-t-l-e H-e-r-e," in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1984).

29. ZAVE, P., "A Distributed Alternative to Finite-State-Machine Specifications," manuscript, Bell Laboratories, Murray Hill, New Jersey (1983).

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

mms

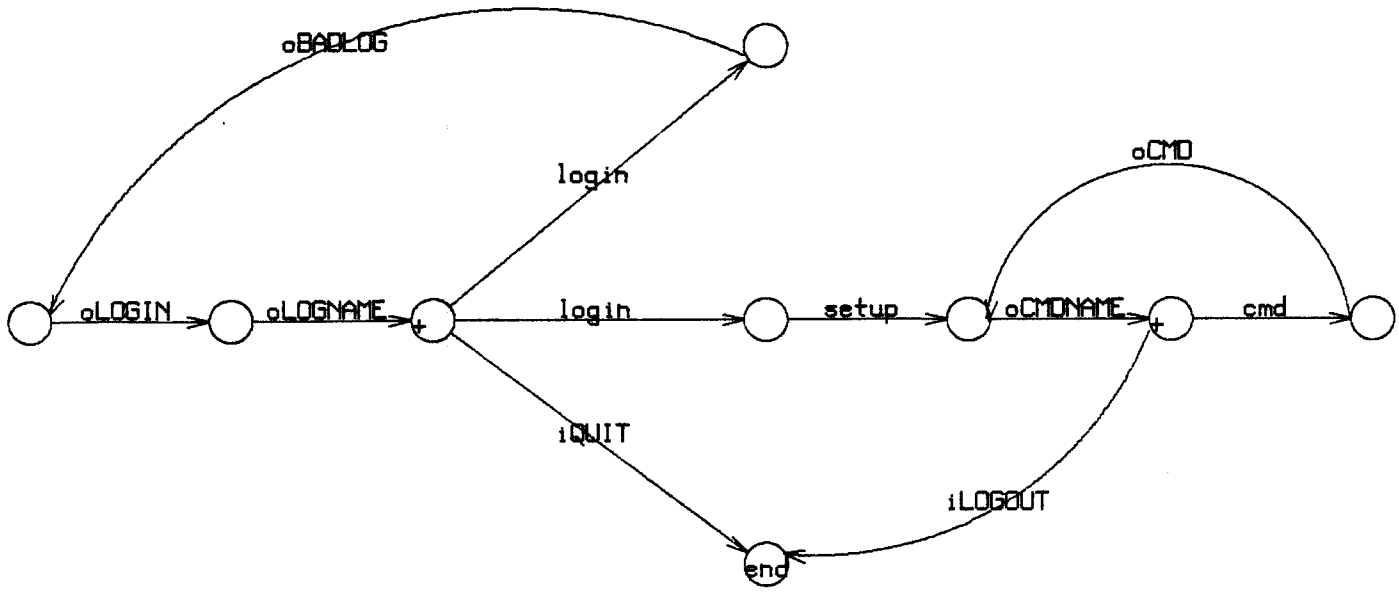


Figure 2. Text form of first diagram of the message system specification.

```
mms →end

st:          oLOGIN →promptlog

promptlog:   oLOGNAME →getlog

+getlog:     login →(setup,badlog)
+getlog:     iQUIT →end

badlog:      oBADLOG →st

setup:       setup →promptcmd

promptcmd:   oCMDNAME →getcmd

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

ready:       oCMD →promptcmd
;
```

Figure 3. Additional diagrams from the message system specification.

cmd

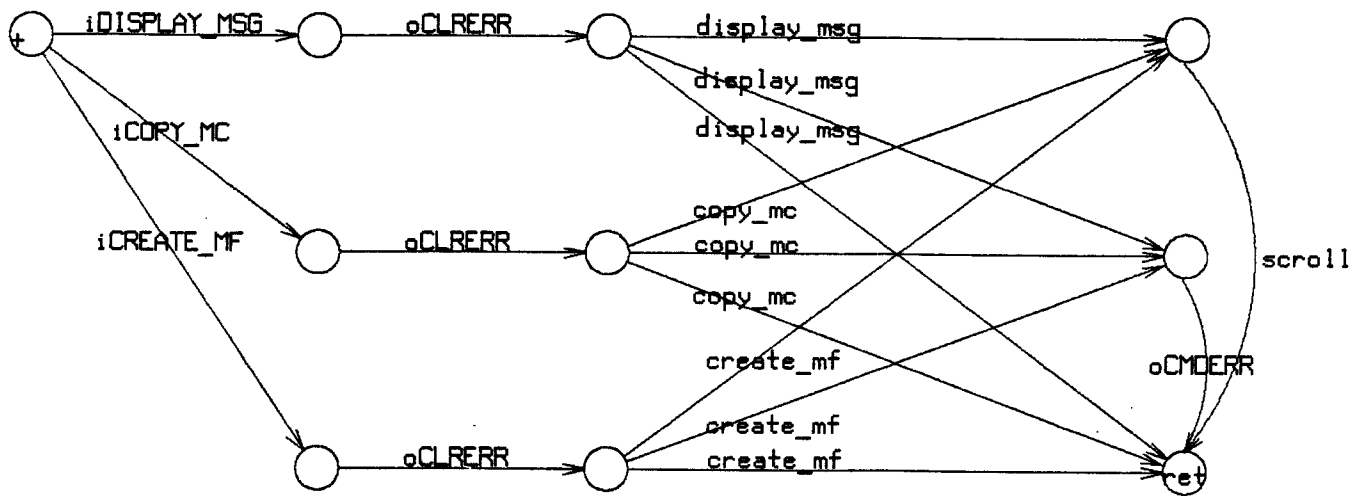
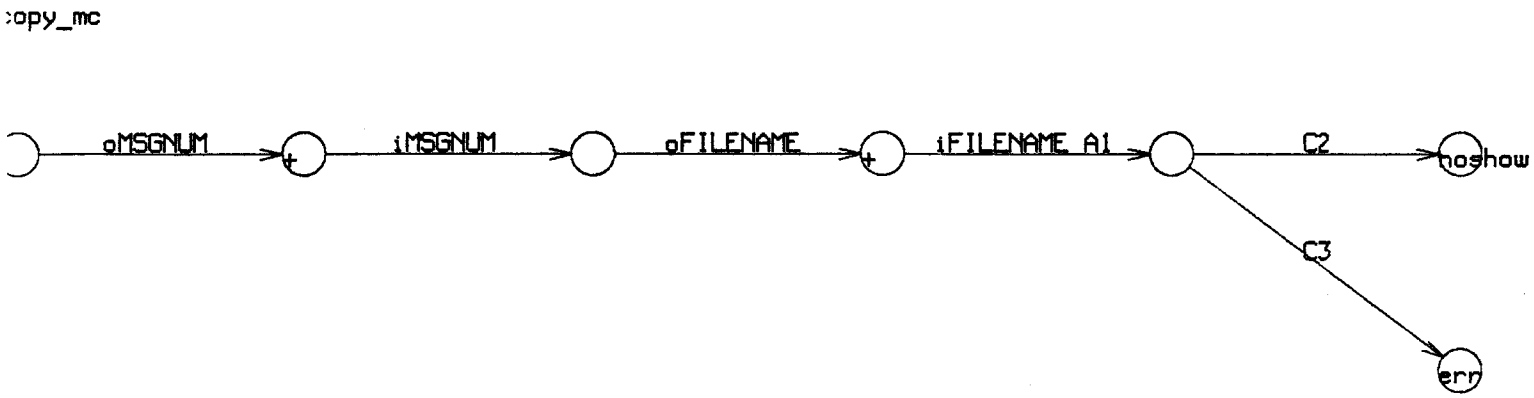


Figure 3. Additional diagrams from the message system specification, *continued.*



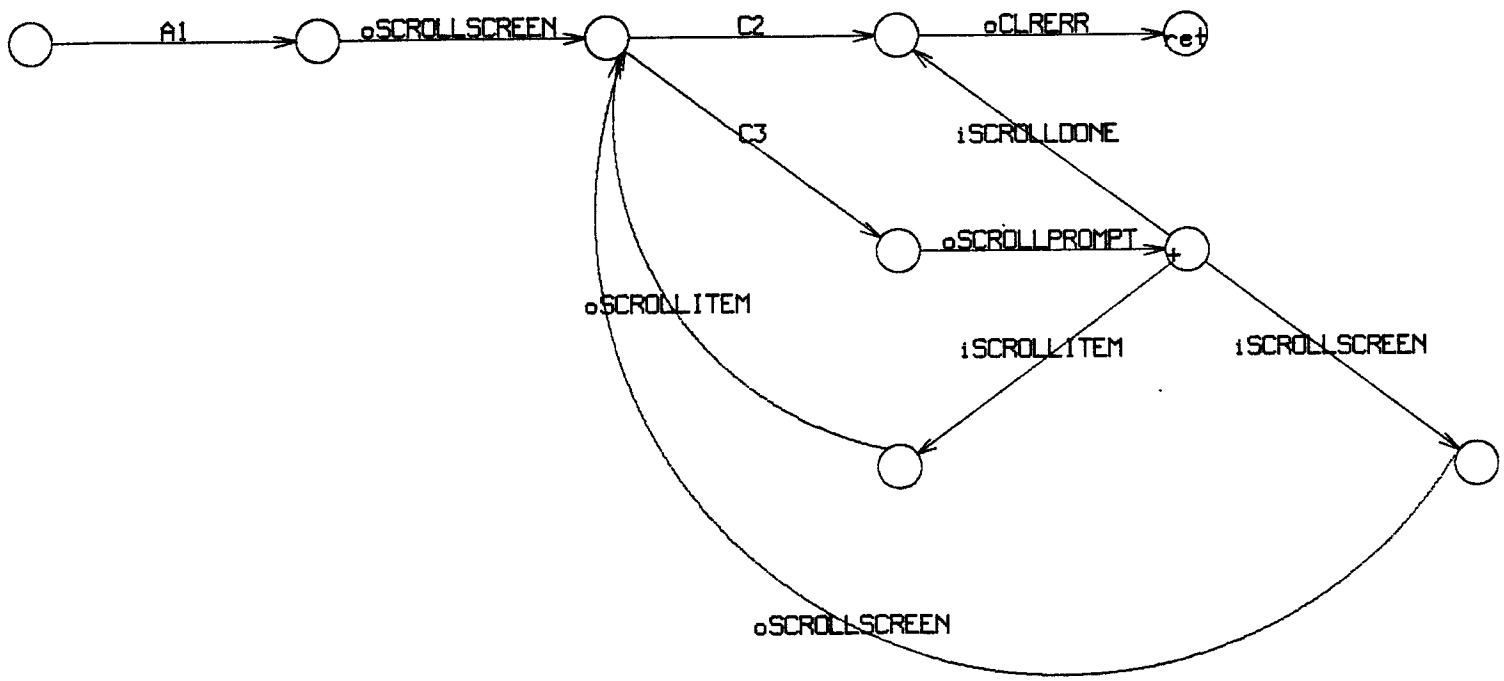
(1) **Act:** COPY_ME(*voCMDERR, viMSGNUM, GLOBAL_curmf, viFILENAME)

(2) **Cond:** equal(voCMDERR, "OK")

(3) **Cond:** NOT equal(voCMDERR, "OK")

Figure 3. Additional diagrams from the message system specification, *continued.*

scroll



(1) **Act:** assign(*GLOBAL_LinesShown, "0")
(2) **Cond:** shownAll(GLOBAL_dispobj, GLOBAL_LinesShown)
(3) **Cond:** NOT shownAll(GLOBAL_dispobj, GLOBAL_LinesShown)

Figure 4. Additional diagrams from the message system specification—text form.

```
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 →(ret,show,err)
do_copy:       copy_mc →(ret,show,err)
do_create:     create_mf →(ret,show,err)

show:          scroll →ret

err:           oCMDERR →ret
;

copy_mc →(noshow,show,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
;

scroll →ret

st:            →first act:assign(*GLOBAL_LinesShown, "0");

first:         oSCROLLSCREEN →test

test:          cond:shownAll(GLOBAL_dispobj, GLOBAL_LinesShown); →doneit
test:          cond:NOT shownAll(GLOBAL_dispobj, GLOBAL_LinesShown);
               →promptit

doneit:        oCLRERR →ret
```

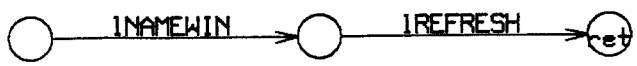
```
promptit:     oSCROLLPROMPT →getit

+getit:       iSCROLLDONE →doneit
+getit:       iSCROLLITEM →oneitem
+getit:       iSCROLLSCREEN →onescreen

oneitem:      oSCROLLITEM →test
onescreen:    oSCROLLSCREEN →test
;
```

Figure 5. Examples of lexical specifications from the message system.

oLOGNAME



oBADLOG

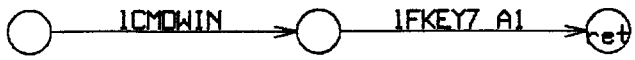
(1) **Act:** print("Sorry, try again - or press ESC to exit")

oCMDERR

(1) **Act:** print(voCMDERR)

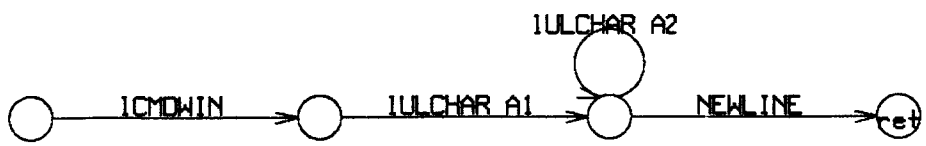
Figure 5. Examples of lexical specifications from the message system, *continued.*

iCOPY_MC



(1) **Act:** print("copy_mc")

iFILENAME



(1) **Act:** {print(vlULCHAR); assign(*viFILENAME, vlULCHAR)}
(2) **Act:** {print(vlULCHAR); append(*viFILENAME, vlULCHAR)}