Survey and Examples of Specification Techniques for User-Computer Interfaces

Robert J.K. Jacob

Computer Science and Systems Branch
Naval Research Laboratory
Washington, D.C.

ABSTRACT

Formal and semi-formal specification techniques have been applied to many aspects of software systems. The module structure, the "uses" hierarchy, and the process structure of a system are examples of areas in which such techniques are useful. However, specification techniques have been far less successful in describing the interface between a system and its user.

This paper provides a survey of techniques suitable for specifying the user interface of a system and a detailed discussion of the relevant literature. Then, it presents a collection of examples of the application of several representative specification techniques to a common set of examples in order to compare the relative merits of the techniques. Section 1 discusses reasons for specifying a user interface, criteria that such a specification should satisfy, and the application of user interface specification techniques to the Military Message System Project at the Naval Research Laboratory. Section 2 describes the principal specification techniques that have been used; nearly all of them are found to fall into one of two classes—those based on BNF and those based on state transition diagrams. Section 3 provides a brief example to illustrate specifications of these two classes. Section 4 contains a detailed survey of the specification techniques and the ways in which they have been applied. Section 5 introduces the notation used in the examples of user interface specifications that follow; and Section 6 presents those specifications. Section 7 presents some conclusions and an annotated bibliography.

April 3, 1986

Survey and Examples of Specification Techniques for User-Computer Interfaces

Robert J.K. Jacob

Computer Science and Systems Branch
Naval Research Laboratory
Washington, D.C.

1. Introduction

Formal and semi-formal specification techniques have been applied to many aspects of software systems. The module structure, the "uses" hierarchy, and the process structure of a system are examples of areas in which such techniques are useful [1,2]. However, specification techniques have been far less successful in describing the interface between a system and its user. Specification of a user interface is of particular interest in the Military Message Systems (MMS) Project at the Naval Research Laboratory, because the project attempts to specify all aspects of the behavior of a family of message systems formally and because the security of a multi-level secure message system is dependent on its user interface.

This paper provides a survey of techniques suitable for specifying the user interface of a system and a detailed discussion of the relevant literature. Then, it presents a collection of examples of the application of several representative specification techniques to a common set of examples in order to compare the relative merits of the techniques.

The remainder of Section 1 discusses reasons for specifying a user interface, criteria that such a specification should satisfy, and the application of user interface specification techniques to the MMS Project. Section 2 describes the principal specification techniques that have been used; nearly all of them are found to fall into one of two classes—those based on BNF and those based on state transition diagrams. Section 3 provides a brief example to illustrate specifications of these two classes. Section 4 contains a detailed survey of the specification techniques and the ways in which they have been applied. Section 5 introduces the notation used in the examples of user interface specifications that follow; and Section 6 presents those specifications. Section 7 presents some conclusions and an annotated bibliography. The survey in Sections 1-4 is based on [3], and the examples in Sections 5-6 are based on [4].

A summary of this work is given in [5], which presents conclusions about the relative merits of the specification techniques studied. The purpose of this paper is to provide the background material that supports those findings, including more detailed discussions of the relevant literature and complete expositions of the actual specification examples that were studied.

1.1. The User Interface

The user interface is becoming an increasingly critical aspect of software systems. While the principal problem in building computer systems used to be providing sufficient processing power, it is now more often providing a good user interface. Getting computers to perform processing tasks is becoming a less serious problem, as hardware costs decrease. Instead, the major stumbling blocks now occur in permitting the user to communicate what he (or she) wants the computer to do and in receiving information from the computer in ways that are easy for him (her) to assimilate. This is particularly important since an increasing number of computer users are not trained programmers but rather specialists in some other field; such users are less tolerant of and less able to compensate for a poor user interface than are computer specialists. However, the designer trying to engineer a good user interface is handicapped without a clear and precise technique for specifying such interfaces.

The specification of the user interface is also important because, for a user, that description is identical to the description of the system itself. Experiments suggest that many users cannot distinguish the difference between changing the user interface to a system and changing the system itself [6]. For many of the people who must understand a software system, the user interface specification *is* the system specification.

1.2. Desiderata for User Interface Specifications

What qualities should a specification technique for user-computer interfaces possess?

- *The specification should be easy to understand.*

  Specifically, it should be easier to understand and take less effort to produce (and, possibly, be shorter) than the software that implements the interface. Someone who wants to answer a question about the behavior of a user interface should find it easier to do so by reading the specification than the code itself.

- *The specification should be precise.*

  It should leave no doubt as to the behavior of the system for each possible input.

- *The specification technique should be powerful.*

  This means it should be able to express complicated system behavior in a compact and easy to understand fashion. It should also be able to describe a wide variety of user interfaces, particularly those beyond the traditional one line-at-a-time typed interactive terminal dialogue. This includes graphical input and output, spoken communication, and analog devices (such as levers or position sensors). It should also include concurrent interactions using more than one communication mode—for example, spoken dialogue during manipulation of knobs and levers.

- *The structure of the specification should be closely related to the user's mental model of the system.*

  That is, the specification should represent the cognitive structure of the user interface (the mental constructs the user needs to operate it) rather than the physical actions required or such implementation details as internal data or control structures. The specification should, then, contain the constructs a user will keep in mind when learning about the system—the basic outline around which the user's mental model of the system will be built. Put another way, the specification should yield a table of contents for a user manual but not necessarily the material in the manual.

  The goal here is to make the structure of the specification and the structure of the user's manual as similar as possible. Singer [7] attempted to adhere to this goal: "In general, whenever I found a feature or concept difficult to explain clearly [in the specification language], I took this as a signal that the design itself was likely at fault. Whole versions of the editing requests were rejected because I could discover no simple way of explaining them." [7, p. 14]

  Two simple applications of this approach arise in common practice. One is the use of BNF to provide structure to a user's manual. Many command language manuals, such as IBM's CMS manual [8], give a single section for all the rules that define some nonterminal symbol, like **command line.** Many of those rules use other nonterminals, for example, **file specification. File specification** and other frequently-used nonterminals, in turn, are described in another section, and so on. A parallel example for state diagram notations is found in user manuals that describe states of the system (usually only for a few major states). The user is told that the system will be in, for example, the **monitor command state** at some point. If he calls an editor, it will be in the **editor state,** and a different set of meanings will be attached to his actions. Users are often explicitly told of such states by means of distinctive prompt characters. So, in two simple ways, the types of constructs often used in writing specifications of a user interface for designers have also been used to describe those user interfaces to users.

- *The specification should be easy to check for consistency.*

  It should make apparent inconsistencies and oversights in a user interface design and generally facilitate the evaluation of a user interface from its specification. Syntactic errors in the specification should imply errors in the design; and, if possible, awkward constructs in the specification should imply an awkward user interface. Reisner [9] and others have examined ways in which a specification of a user interface can be used to make specific evaluations of that interface.

- *The specification should emphasize the cognitive steps the user performs.*

  It is the user's cognitive or information-processing actions, rather than physical actions, that principally determine user performance and accuracy. Thus, for the specification to be useful for predicting user performance, it should describe such actions.

- *The specification should separate function from implementation.*

  It should describe the behavior of a user interface completely, precisely, and unambiguously but constrain the way in which it will be implemented as little as possible.

- *Ideally, a specification should be useful for simulating a user interface.*

  Given a specification, it should be possible to construct a prototype or mockup of the user interface of a system rapidly and, perhaps (with an executable specification), automatically.

1.3. The Military Message System Family

Current software technology has provided several techniques for specifying the external behavior of a system separately from its internal implementation. Such techniques have been applied to communication between software modules with useful results, but they have been less helpful for specifying the interactive communication between a user and a system in an understandable fashion. In the MMS Project [10], semi-formal specification techniques have been used to capture the functional capabilities of a family of message systems. These capabilities have been specified independently of the user interface. How to apply such specification techniques to the user interface of a military message system is now of interest.

A military message system with multi-level security makes particularly stringent demands on its user interface. Experience in the Military Message Experiment [11] showed that communicating the security consequences of an action and obtaining meaningful approval or disapproval from a user was not easy. Because the security of a system requires the user to understand the security-relevant consequences of his actions, the user interface is of special importance in secure systems.

Another reason for specifying the user interface is to make it possible to isolate code concerned with user-computer communication into a separate software module. Then, one can describe and implement different user interfaces for the same underlying system by making only localized software changes. Different interfaces might also be provided to suit different types of users. Such a view implies that there are two separate interfaces that must be considered. As shown in Figure 1 the user communicates with a "User Agent" via a *User Command Language* (UCL). Once it receives a command from the user, the User Agent translates the command into a standard form—a statement in the *Intermediate Command Language* (ICL)—and passes that to the Data Manager. Information returned by the Data Manager in response to user requests is delivered to the User Agent, which is responsible for displaying it to the user. This division permits new systems with different user interfaces to be constructed from an existing system with relative ease. If the user interface is changed, only the User Agent must be modified so that it will translate from the new UCL into the standard ICL; the Data Manager and other system components need not be changed (as long as they provide sufficient primitive operations for the new UCL commands).

2. Introduction to Specification Techniques

2.1. Static and Interactive Languages

Most previous work on formal and semi-formal language specifications has been devoted to the specification of *static* languages, as opposed to *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. Processing of an input text is affected little, if at all, by the previous input texts. Such is the case with the processing of the text of a computer program, and hence the field of programming language compilers is where much of this work is found. This contrasts with interactive languages, in which the system produces responses at various points during the input of a text, resulting in a dialogue. The input can be described as one long text, in which the computer may take actions and produce outputs at various points during the input or as a series of brief texts, where the processing of each input generally depends on previous processing. The user may also make use of intermediate outputs in formulating his subsequent inputs. He may also sometimes interrupt the system and begin entering a new statement in the input language. A specification of an interactive language must capture more information about the behavior of the system than one for a static language. That is, it must specify each of the system's responses and at precisely what point in a dialogue each will occur.

The language to be described in a user interface specification differs from conventional compiled computer languages in still another way. Not only is it interactive, rather than static, but its primitive symbols are often user actions other than entering strings of characters [12]. Such actions may include pushing special function keys, adjusting dials, operating a digitizer, speaking, or moving one's eyes. In order to keep levels of specification detail separate, terminal symbols of the high-level specification should consist of high-level operations involving virtual hardware devices. "Move cursor to date field" or "select send option" are such terminal symbols. These are then resolved into specific hardware operations at the next lower level of specification detail. "Move cursor to date field" may be mapped into a sequence of cursor control key operations or the manipulation of a light pen or some other scheme. That is, the *tokens* of the input language of the high-level specification are coherent groups of atomic user actions (the analogue of words in conventional languages). The *lexemes* are the individual user actions upon the hardware devices (the analogue of characters). The high-level specification describes the decomposition of the user interface into these tokens, and the lower-level specification describes the mapping from tokens to lexemes.

For the purpose of predicting user performance (*i.e.*, accuracy, speed, and the like) or evaluating a user interface from its specifications, it is important that the high-level specification emphasize the cognitive or information-processing steps the user must perform, rather than the physical actions. Although the former are often ignored in favor of the latter, it is the cognitive, not physical, actions that principally determine user performance and accuracy for a system [9]. Hence they are what specifications should capture in order to be useful for comparing two user interfaces.

2.2. Principal Specification Techniques for Static Languages

Two principal classes of techniques have been used for specifying static languages. One is based on production rule grammars and yields a specification in a notation such as Backus-Naur Form (BNF) [13]. The language is described by a set of production rules, from which all possible correct inputs in that language can be produced. Each rule gives the definition for some *nonterminal* symbol. Wherever that symbol itself appears (in the definition of some other symbol), it may be replaced by substituting the contents of its definition. In this way, the definition of a single starting symbol ultimately yields all legal strings in the language.

The other class of techniques is based on finite state automata and results in a state transition diagram (or, equivalently but more amenable to computer processing, a list of transitions

between the states of such a diagram). The specification consists of a set of nodes (states) and arcs between them (state transitions). Each transition is associated with a token in the input language. From any state, the next input language token received causes the transition labeled with that token to occur.

2.3. Application of the Specification Techniques to Interactive Languages

Both of these types of techniques must be modified to be used for interactive (rather than static) languages. The modifications are very similar to those made to these techniques to create compilers for static languages. The modification of BNF consists of associating an action with each grammar rule. Whenever that rule applies to the input language stream (received thus far), the associated action occurs. Such actions can include prompts and other feedback to the user and also actions that involve processing by the rest of the system (*e.g.*, [14]).

State transition diagrams are modified in a similar way. Each transition is associated with an action, which can include user feedback and also other processing, as with the modified BNF. Whenever a transition occurs, the system performs the associated action (*e.g.*, [15,16]).

Previous user interface specifications have suffered because they lacked an acceptable language for describing the "semantics" of the interface, *i.e.*, the actions associated with BNF rules or state transitions that the system performs in response to the user's commands. Since a complete description of such actions is in fact a specification of the entire system, putting it in the user interface specification clutters that specification with detail that belongs at another level [17]. What is needed is a high-level model that describes the operations that the system performs. Then, the user interface specification would describe the user interface in terms of the model, while the internal details of the model would be described in a separate specification.

The design used in the MMS provides one solution to this problem. The Intermediate Command Language *is* an abstract model of the services performed by a message system and is described in a separate specification [18]. The user interface specification, then, needs only to specify the *syntax* of the UCL as described above and the *semantics* of the UCL using actions that consist of commands in the ICL issued by the User Agent to the Data Manager.

3. Example

The following brief example is introduced to show what specifications in BNF and state diagram notation look like.

The `send message` command specified below takes two arguments, both of which are required—the name of the message draft to be sent and the name of its desired recipient. If the user fails to enter the necessary parameters, the system will prompt for the missing parameter(s). If the user enters "help," he will get a description of the next parameter to be entered.

3.1. BNF Specification

Syntax: Lower case names denote nonterminal symbols, which are subsequently defined in terms of terminal symbols. Upper case names are terminal symbols, which would be defined in a separate, lower-level specification. Actions invoked when a rule is satisfied are placed in braces.

```
send_cmd::=             SEND help_draft DRAFTNAME help_recip RECIPNAME

help_draft::=           single_help_draft
|                       single_help_draft help_draft

single_help_draft::=    NOTHING
|                       HELP {reply("Draft name can be...")}
|                       ANYTHING_ELSE {reply("Enter draft name")}

help_recip::=           single_help_recip
```

```
|                       single_help_recip help_recip

single_help_recip::=  NOTHING
|                       HELP {reply("Recipient name can be...")}
|                       ANYTHING_ELSE {reply("Enter recipient name")}
```

Note that the definition for terminal SEND, given in a lower-level specification, might be a function key or an abbreviation for the word "send" or a menu selection operation; its definition is immaterial to this specification.

3.2. State Diagram Specification

Syntax: Each transition between two states is shown as a labeled, directed arc between the two nodes that represent those states. Each transition is labeled with the name of an input token and, optionally, an action. The transition will occur if its input token is received; when it does, the system will perform the action.

Figure 2 gives the state diagram.

3.3. Text Representation of the State Diagram

Syntax: Each state transition is represented by a line of the form

s1: INP →*s2* **act:**reply(rep);

denoting a transition from state `s1` to state `s2` that expects input token `INP` and performs an action that prints `rep`.

sendcmd → *end*

start: SEND →*get_dr*

get_dr: DRAFTNAME →*get_re*
get_dr: HELP →*get_dr* **act:**reply("Draft name can be...");
get_dr: ANYTHING_ELSE →*get_dr* **act:**reply("Enter draft name");

get_re: RECIPNAME →*end*
get_re: HELP →*get_re* **act:**reply("Recipient name can be...");
get_re: ANYTHING_ELSE →*get_re* **act:**reply("Enter recipient name");
;

4. Survey of Specification Techniques

4.1. Techniques Based on BNF

The BNF notation that is used to describe static languages has been modified by various investigators to make it applicable for specifying interactive languages. This section discusses these uses of BNF-based techniques for specifying interactive user interfaces.

Several investigators have used a form of BNF specification with an action associated with each rule to describe a user interface to a prototype-building program (often called a compiler-compiler, from its original use in generating compilers). The system accepts a specification of a user interface in BNF plus associated actions and then creates an implementation of that user interface. Several such systems have been constructed, generally using a conventional programming language for specifying the actions [19,20,21,14,22]. At least one of those [21] has been used to construct widely-used, practical systems (*e.g.*, [23]), and another [20] provides the basis for an extensible data analysis language [24]. Efficient methods of parsing an input text,

given a BNF description of its language, have been studied extensively [25, 26].

Fenchel [27] describes a prototype-builder that gives a system the ability to describe itself. It uses a BNF specification to describe a user interface to an interpreter, which then implements that user interface. In addition, the interpreter provides a help facility based entirely on the information in the BNF specification.

Lawson *et al.* [28] propose and use a form of BNF with action descriptions, similar to those used in the compiler-compilers, as a specification technique for describing interactive user interfaces.

Reisner [9] provides a major example of the use of BNF for describing a user interface. While some other user interface specifications look more complex than Reisner's, it is because they express characteristics of interactive behavior that Reisner didn't need for her work, but that are necessary in general: the computer's responses and other actions. Reisner's BNF specification describes only the syntax of the input language; it gives no idea of what (if anything) the system will do as the user is entering commands. It is as though she had specified a static version of her system's interactive command language. Moreover, the system she specifies is a fairly simple one, but it still requires a considerable number of rules.

Reisner uses BNF specifications of two user interfaces to predict differences in the performance of their users (number and type of errors made). Embley [29] has also used BNF in a similar way but for a simpler problem. The terminal symbols in Reisner's specifications can be arbitrary user actions, rather than just character strings. The specifications themselves emphasize actions the user has to learn and remember (*i.e.*, *cognitive* factors, as opposed to more common but less useful factors, such as amount of hand movement). Reisner's goal is to use the specification of a user interface to evaluate the interface itself, in particular, its *cognitive simplicity*. Cognitive simplicity leads to two substantive specific criteria: *string simplicity* and *structural consistency*. String simplicity is easy to determine from a specification. It is the length of terminal strings (shorter is considered better). This criterion can be applied equally well to BNF- and state diagram-based specifications, and it does not depend on the choice of nonterminals. Reisner's interpretation of her structural consistency criterion, however, is more subjective. Intuitively, structural consistency means that the rules needed to describe similar situations in the user interface are themselves similar. Reisner's definition of structural consistency can be reformulated as: Look for grammar rules with similar left-hand sides and see to what extent their right-hand sides are also similar (more similar means more structurally consistent). This works for either BNF- or state diagram-based notations, but it does depend on the choice of nonterminals. Since BNF notations often require more nonterminals, while state diagram notations encourage a specifier to use fewer, the choice of notation indirectly bears upon this criterion.

Shneiderman [30] describes a new type of BNF-based grammar in which each nonterminal symbol is associated with either the user or the computer (or one of several parties, in a multiway conversation). This method actually appears to be the BNF-based analogue of Singer's state diagram-based notation [7]. In state diagram notations other than Singer's, each transition is labeled with a user input (which determines which transition will be selected) followed by a computer response. Singer instead permits only a single action (which could either be a user input or a computer response) on each transition, and then he allows such transitions to be combined in arbitrary ways. Shneiderman achieves a comparable result by permitting a "composite" nonterminal to be defined in a grammar rule as an arbitrary combination of computer- and user-labeled nonterminals.

In fact, conventional state diagram-based notations can accommodate all but one of the possibilities that Shneiderman's notation permits—and that one case yields an anomalous specification. The usual state transitions contain a user input followed by an arbitrary combination of computer actions (or no action). The only case that a combination of such transitions cannot describe is a transition that specifies a computer action and no user action. In Shneiderman's

notation, that would be described as:

```
greeting::=  computer: "hello"
|            computer: "how are you"
```

This says that the nonterminal `greeting` expands into one of either "The computer says 'hello'" or "The computer says 'how are you.'" Without any specification of the user input, however, this gives no indication of how the computer will decide which of the two responses to give. So this case does not yield a very useful specification. (A more conventional and more useful specification would be:

```
greeting::=  user: "short greeting please"  computer: "hello"
|            user: "long greeting please"   computer: "how are you"
```

which is straightforwardly expressible in conventional state diagram notation.) Shneiderman does not use the anomalous case above in any of his examples, but it is the only one in which his new grammar provides additional expressive capability. Shneiderman's notation requires a specification to contain one extra layer of nonterminal symbols (at the level immediately above the terminal symbols) in order to associate a party (user or computer) with each terminal. Finally, Shneiderman's goal of moving both sides of the dialogue into the syntactic domain could be satisfied simply by considering replies from the computer to be part of the grammar specification in a state transition diagram rather than simply one of a large class of arbitrary "semantic actions."

Shneiderman [30] also addresses the problem of specifying several important aspects of interactive user interfaces applicable to modern display terminals, but does not give any substantive insights into it. For describing visual attributes of CRT characters (*e.g.*, intensity, underlining), he gives a trivial set of BNF productions. For specifying multiple windows on a display, he gives a declaration, but he does not say how to use it in the grammar. Each window has a starting symbol. It is not clear whether the starting symbol expands into the dialogue for that window or how else the dialogue is generated. If it is from the starting symbol, then there will be several simultaneous dialogues, and the method does not say how to indicate which of them gets which of the user's inputs. A better alternative is for each computer-initiated nonterminal to specify the name of a window into which the expansion of that nonterminal will be placed. That is a reasonable notation, but it is not very different from conventional usage.

There are two general problems with BNF-based techniques. One is that interactive prompting, error handling, and correction that must occur at particular points in a dialogue are awkward to specify, because controlling exactly when a grammar rule is satisfied often requires introducing many artificial intermediate constructs. It is sometimes difficult in BNF to indicate exactly *when* something occurs (that is, after exactly *what* has been received). One remedy is to introduce additional delimiter symbols and associate actions with the recognition of these symbols. Another problem is finding a BNF representation for the counterpart of optional "detours" in a state diagram (as might be used to describe help functions or ignored inputs that return the user to the state he was in before the detour). The BNF technique is to introduce a nonterminal for each detour. One of the definitions for such a nonterminal is always a null input (since the detour is optional). Unfortunately, this approach often complicates a BNF specification considerably more than its counterpart does a state diagram. (The example given in Section 3 illustrates this situation.)

4.2. Techniques Based on State Transition Diagrams

A number of investigators have used techniques based on state transition diagrams to specify interactive user interfaces. Since the concept of time sequence is explicit in state diagram notations (while it is implicit in BNF-based notations), the former are more suited to describing *when* events occur. Most of the specification techniques based on state diagrams also include some special syntactic features to make interactive languages easier to represent, but these features differ considerably among the techniques. As a result, these notations vary widely in their expressive power, ranging from simple finite state automata to Augmented

Transition Networks [16], which are equivalent to Turing Machines. This section examines these specification techniques and their applications.

One of the most important differences among state diagram-based notations is the ability of one diagram to call upon another, much as a program can make a procedure call. This is analogous to using nonterminal symbols in BNF—invented intermediate constructs that permit the specification to be divided into manageable pieces. Introducing nonterminals and using separate diagrams to recognize them is important for describing non-trivial systems. Without it, a specification must consist of a single diagram, which, for any but a trivial system, is complex and unwieldy. With the introduction of nonterminals, instead of labeling a transition with a single token to be recognized, it can be labeled with the name of a nonterminal symbol. That symbol is, in turn, described by a separate state transition diagram. This makes complex diagrams easier to understand. Singer [7] gives a convenient syntax for describing calls upon sub-diagrams, similar to that for procedure calls in conventional programming languages.

One of the principal virtues of state diagram notations is that they make explicit precisely what the user can do at each point in a dialogue and what its effect will be (by giving the transition rules for each state). Feyock [31] makes good use of this property by using a computer-readable representation of the state diagram specification of a system as the input to a user help facility for that system. Based on the state diagram and the current state, the system can answer such questions from users as "What can I do next?" "Where am I?" and "How can I do ...?" Darlington *et al.* [32] suggest that state diagrams provide a good representation of users' cognitive models of interactive systems. Guest [33] observed that programmers preferred a state transition diagram-based front end for a specification interpreteter to a BNF-based one. State diagrams also make user interface problems such as character-level ambiguity [34] easy to discover. Brown [35] attempts to relate formal properties of a state diagram to properties of the corresponding user interface, much as Reisner [9] does for BNF, but without experimental validation.

Introducing sub-diagrams for nonterminals means that the notion of the user's current state must be expanded to include the state in the current sub-diagram, the name of that sub-diagram, the state and sub-diagram from which it was called, etc., much like the stack of saved environment frames that results from a series of procedure calls. The full description of "where the user is" in a dialogue thus comprises this entire stack (as Denert [36] and Feyock [31] explicitly recognize).

The use of nonterminal diagrams can also lead to ambiguities in the specification of an interactive user interface in much the same way that the use of nonterminals in BNF can. Two types of nondeterminism can occur. The first, called here *apparent* nondeterminism, denotes a system that appears to be nondeterministic when viewed from its top-level specification (*i.e.*, with nonterminals), but can (ultimately) be implemented deterministically (when all the nonterminals are expanded into terminal symbols). For example, consider a system that accepts two commands: An `MSG` followed by a `FORMAT` constitutes a request to print the message `MSG` using the indicated format. An `MSG` followed by a `DESTINATION` requests that the message be sent to the indicated destination. This can be specified with the two commands treated as nonterminals, each described in its own diagram. The main diagram would then call two other diagrams to recognize those two commands (syntax follows that of the example in Section 3):

start: print →*end*

start: send →*end*

Nonterminal `print` is defined in the following separate state diagram:

start: MSG →*getfmt*

getfmt: FORMAT →*return*

and send is defined by:

start: MSG →*getdest*

getdest: DESTINATION →*return*

The problem is that after the user enters an MSG, the "state" of the system is ambiguous. Since no action need occur until after the next input, the system can easily be implemented with a scheme that remembers both states and waits for the user's next input to decide from which to continue. The only problem here is that the "state" of the system after the first input cannot be described simply. This situation occurs in BNF too, but there is no explicit concept of state there.

The second type of ambiguity, called here *real* nondeterminism, results in the specification of a genuinely nondeterministic system. It arises because, when actions are associated with state transitions, it is possible to specify several different actions, any of which might occur in response to the *same* input symbol. For example, the diagram for print above could be replaced by the following:

start: MSG →*getfmt* **act:**reply("What format do you want to print in?");

getfmt: FORMAT →*return*

and that for send replaced by:

start: MSG →*getdest* **act:**reply("Where do you want to send it?");

getdest: DESTINATION →*return*

This specifies a system in which, as soon as the user enters an MSG, one of two prompts is printed—but the input from the user that the system needs to decide which of the two prompts it should print is not yet available. (An implementation for a static language could simply wait until it received enough input to decide between the two outputs.) State diagram-based notations that allow calls for nonterminals inherently permit this sort of nondeterminism, since the conflicting actions appear in different (nonterminal) diagrams. It remains the specification writer's responsibility to see that the system specified is realizable with a deterministic machine.

Conway [15] presents an early use of a notation based on state transition diagrams in which an action is associated with each transition. His object is to construct a compiler for a static language. Hence, most of the actions associated with his state transitions generate intermediate code (rather than performing interactive tasks, such as displaying a message to the user). His notation permits one diagram to call another for analyzing a nonterminal (much like a procedure call). Conway recognizes that this can cause ambiguities, as discussed above, and gives conditions sufficient to prevent them.

Woods [16] also describes the use of a state transition diagram notation for analyzing a static language. His notation includes two extensions: the action associated with each transition manipulates a (global) data structure that will represent the translation of the input; and the conditions for making a state transition can be arbitrary Boolean expressions (not just names of tokens or nonterminals to be recognized, but also expressions depending on the data structure). As with Conway [15], one diagram can call another for recognizing a nonterminal.

An early application of state diagram-based notation to interactive (rather than static) languages is provided by Engelbart and English [37]. Their notation is difficult to follow, and they do not discuss it in detail. They do not mention the possibility of using nonterminal symbols in their specification.

Parnas [38] proposes using "terminal state diagrams" for describing user interfaces for interactive languages. He introduces the notion of terminal state (as opposed to complete state), which is similar to the difference between syntax and semantics in other language specifications. A terminal state is "all system conditions (relative to a given user) in which the set of interpreted messages, and the set of interpretations is constant" (*i.e.*, a particular *syntactic*

construct). The content of a system response to a user can vary within a single terminal state (*i.e.*, various *semantic* responses to the same *syntactic* construct). A terminal state transition diagram, then, describes the changes between terminal states as a user interacts with the system. Parnas' examples show only simple cases, and the paper does not address how this scheme would be extended for less trivial cases. The simple syntax given is adequate for the examples in the paper, but it would have to be extended to describe more realistic systems. Since it does not include nonterminals, the nondeterminism problems mentioned above do not arise. In addition, the specifications do not describe any system actions other than displaying messages to a user.

Barnett [39] also defines the user interface for a simple system with a state diagram. Foley and Wallace [12] advocate the use of a state transition diagram to represent the user interface of an interactive system. While their notation is clear and easy to understand, they do not examine the specification problem in much detail. Denert [36] uses state diagram notation to describe and implement a simple user interface. Dwyer [40] uses a state diagram representation as the input to a simple program that directly implements the user interface described by the diagram. Kasik [41] proposes a triply-linked tree as an alternative to conventional state diagrams. For the menu-oriented system he describes, the tree provides a data structure that can be traversed by a deterministic algorithm. Thus it can be executed more efficiently than conventional state diagrams containing nondeterminism. For the purpose of describing a user interface, though, it offers no advantages and introduces unnecessary complexity (the extra links) and unnecessary constraints (the one item per level, menu-driven style of interaction).

The standard for the MUMPS interactive computer language [42] provides an example of a specification of a non-trivial system using a state diagram-based notation. Nonterminal symbols are defined in separate diagrams, and the authors recognize and exploit the resulting "apparent" nondeterminism. The specification gives a precise, detailed description of the rules by which the automaton that executes the diagrams operates. It includes the deterministic algorithm for executing such nondeterministic diagrams (*i.e.*, step through the diagram for one nonterminal until a dead end is reached, then go back and try another nonterminal). This description eliminates the vagueness and informality seen in some other state diagram-based specifications. The notation also introduces some additional criteria for selecting a transition. Each call upon a separate diagram for a nonterminal can return one of up to four possible constant values, and the calling diagram can select its next transition based upon this value. Woods [16] and Singer [7] both provide more general mechanisms for achieving such a result through the use of an external data structure, which actions may set and transition rules may examine. The actions associated with the transitions in this specification are unusually comprehensive; they comprise a complete specification of the semantics of the MUMPS language.

Wasserman and Stinson [43] use a notation based upon that of the MUMPS specification [42] to specify the user interface of a very simple interactive system. Their notation is easy to follow, although the example they provide is too simple to provide a serious test of the notation. The example is strictly deterministic (left imperative grammar), and they do not address whether they intend the technique to handle the apparent nondeterminism discussed above. Their example also suggests (by omission) the value of using nonterminals in state diagrams, since it would be improved by the addition of two nonterminals with separate diagrams (specifically, in Figure 8 of [43], one for `get-account-number` and another for `get-amount`). By contrast, if their example is translated into BNF one is virtually forced to add the two (along with a number of other un-helpful nonterminals).

The use of the token ANY in their notation is also somewhat imprecise. Each transition rule from a state gives a token that should appear in the input for this transition to occur. If none of those tokens appears in the input, then the transition labeled ANY is made. In the resulting state, the input is again examined to determine the next transition. It is not clear whether the same token (the one that matched ANY) is examined again or whether the input

stream is advanced to the next token. The former interpretation is more conventional [26], results in a more versatile notation, and is the one that is implied by the precise definition given in [42]. Applying that interpretation to one of Wasserman and Stinson's diagrams, however, yields an infinite loop, while applying the other interpretation to the diagram results in the specification of an unusually poor user interface.

Another ambiguous detail in their notation is the specification of messages to the user. Messages are associated with states rather than transitions. This makes it difficult to determine whether a message occurs before or after the other actions associated with a state transition. It also requires their example to contain two extra states, each with only one entry and one exit, whose sole purpose is to hold messages. Associating messages with transitions between states, as are actions, would be clearer.

Another problem is a confusion between syntax and semantics. One diagram (Figure 8) includes a transition for a syntactic error (three unrecognized inputs in a row) that can only be made as a result of an action in the semantic domain (set and test a counter). Such a mechanism is necessary for errors that are genuinely in the semantic domain (*e.g.*, an unknown account number given in the correct format for account numbers), but it is here applied to a syntactic error.

It is interesting to see how Reisner's criterion of structural simplicity [9] can be translated into the terms of this notation. It would become: Given two nonterminals that are called with similar commands from the same state (*e.g.*, CREDIT and DEBIT), how similar are their individual diagrams? The example in Wasserman and Stinson's paper does not give enough data to permit one to apply the criterion to it, though.

Wasserman and Shewmake [44] describe an interpreter that accepts a specification in this form of state transition diagram notation and then simulates the specified system. Semantic actions are given as individual commands in the command language of the operating system.

Singer [7] presents a state diagram-based specification of a significant system (in contrast to much other work in the field, which describes fairly trivial examples). The notation is more precise and more general than most other versions of state diagrams, but it is more complex and difficult to follow. It uses separate diagrams for nonterminals, and a global data structure, which is set by arbitrary semantic actions. All transitions are selected by examining values in this data structure. While the two appear quite different, most parts of Singer's notation could be mapped into that of the MUMPS specification [42].

For simple cases, Singer's notation appears quite clumsy. One reason is that it introduces three states to (1) read something (into the data structure), (2) print a message (the content of which may depend on the value of the thing just read or on other data in the data structure), and (3) select the next state (also based on predicates on the data structure). Thus, the following transition rules (taken from the example in Section 3):

get_dr: DRAFTNAME →*get_re*

get_dr: HELP →*get_dr* **act:**reply("Draft name can be...");

get_dr: ANYTHING_ELSE →*get_dr* **act:**reply("Enter draft name");

would be written in Singer's notation as:

```
get_dr:    read-cmd  → got_cmd

got_cmd:   cmd=DRAFTNAME?  → get_re
got_cmd:   cmd=HELP?  → was_help
got_cmd:   cmd=ANYTHING ELSE?  → was_else

was_help:  print "Draft name can be..."  → get_dr
was_else:  print "Enter draft name"  → get_dr
```

In fact the general two-step procedure of "*read-X; X=...?*" in Singer's notation is confusing. An even worse problem arises when *X* is a complex command. Then the procedure can become "*read-X; parse-X; X-is-of...type?*" This has the effect that what should be several levels of syntactic and semantic detail about the makeup of complex command *X* in the specification are collapsed, because the details of the syntax of *X* are in semantic definitions of the action *parse-X* and the predicate *X-is-of...type?* One possible reason for preferring this two-step form of receiving user inputs to the more widely-used implicit form (such as that of [43] or the example in Section 3) is that it could permit several parallel streams of user input. A transition could call for *receive-into-stream1*, and another transition could later call for *receive-into-stream2*. Then, either of the streams could be tested, using the appropriate semantic predicate, in order to select the next transition. It is not clear, however, what parallel input from a user means. As long as he or she has a single input device (like a keyboard), all inputs are ultimately sequential, even though the user may intend them to form several logical streams. Parallel input makes sense when the user has several devices that he can operate simultaneously in real time (a joystick and a pedal or a keyboard and an eye tracker). The price paid for Singer's more general notation may be worthwhile for such systems, but the systems he actually studies have only one input stream.

Singer emphasizes the distinctions between interactive behavior, syntax, and semantics and shows how his notation permits the interactive behavior to be described separately from syntax and semantics. The interactive behavior is defined by the states and transitions, and the syntax and semantics are specified in the definitions of the actions and predicates on the data structure. Unfortunately, syntax and semantics are not well separated in this notation. Moreover, Singer does not clearly distinguish interactive behavior from syntax and semantics. For example, he gives no guidance that would militate against an extraordinarily unenlightening specification in which interactive behavior is entirely described by the following diagram:

`ready:` *receive-an-input-act-from-user-and-store-it-in-buffer* → `proc`
`proc:` *perform-processing-on-buffer-and-output-results-if-any* → `ready`

while all other characteristics of the system (including syntax, semantics, and everything interesting about its interactive behavior) are described in the definition of the semantic action *perform-processing-....*

Singer's description for the interrupt mechanism in his system is theoretically interesting in that it is self-contained—it is expressed in the same state transition diagram notation as the rest of the system—but it is almost impossible to read. Such a specification is precise, but if it does not make the behavior of the system easier to understand than does the program code itself, it is of questionable value. The intuitive concept of his interrupt mechanism is fairly simple to explain, but the state diagram notation for it, using a stack of saved environments and a scheduler to select among them, is quite opaque. The result here is a user interface specification that makes the system more, not less, difficult to understand.

One useful aspect of Singer's notation is that it contains a reasonable text representation for state transition diagrams. His syntax for making calls on sub-diagrams to recognize nonterminal symbols is particularly convenient. His language for specifying semantic actions is less helpful. Semantic actions manipulate a portion of a declared data structure, in a way rather like arbitrary computer program statements might. The entire data structure is global in scope to all of the actions. Modularization and scope rules for the semantic objects would help. Unfortunately, Singer's language for semantic actions has no provision for these, so that it militates toward rediscovering the complexity and related software engineering problems that arise when a large system consists of only one module (*i.e.*, all data are global).

Wasserman and Stinson's state diagram notation [43] can be converted into Singer's provided the former does not contain any nondeterminism. Wasserman and Stinson's example is strictly deterministic at every level, but specifications for most non-trivial languages would not be. In addition, Singer's notation would yield an extra state (with an action) for each of the input operations that Wasserman and Stinson imply by placing a token name on a transition

arc. This means that Singer's diagram would have many more states; but its notation is also more general, because it allows inputs without associated actions as well as the converse.

ZOG [45,46,47] is an outgrowth of PROMIS (Problem Oriented Medical Information System) [48] and provides a unified conception of a man-machine interface based on the notion of screens. It consists of a fairly simple, menu-based system that is fundamentally similar to a state transition diagram. Each screen or menu corresponds to a state, and each of the menu options or other commands available in that state corresponds to a transition from the state. This approach may be too limited to for specification of a general user interface, but its conceptual simplicity is appealing.

Finally, state diagram notations have also been found useful by a number of investigators for describing and analyzing protocols for communication between machines, rather than between people and machines [49,50].

4.3. Relationships Between BNF and State Diagrams

The formal equivalence of BNF and various state transition diagram notations is well established (*e.g.*, [51]). Most state transition diagram notations discussed here introduce sub-diagram calls. The ability for a diagram to make *recursive* sub-diagram calls and the use of a genuinely nondeterministic automaton has been shown to give the notation the power of a context-free grammar [52]. Adding arbitrary actions and conditions (and the ability to create new variables dynamically) to either BNF or to this state diagram notation makes either equivalent to a Turing Machine.

Hence, the two classes of notations examined here are formally equivalent. A description in one notation can be transformed algorithmically into a corresponding description in the other. However, the two resulting specifications, while equivalent, are likely to differ considerably in their clarity to human readers. The ability to transform one notation into another with an algorithm demonstrates theoretical equivalence of the two types of notations, but the translations they produce do not provide generally useful or readable specifications.

To clarify the relationship between BNF and state diagram notation, several such algorithmic transformations between notations of these two types can be considered:

1. BNF with nonterminals into state diagram with the same nonterminals. This can be done with a straightforward algorithm. Each instance of concatenation in a BNF rule introduces an intermediate state in the state diagram translation. Each "or" in a BNF rule causes a fork in the diagram.

2. State diagram with nonterminals into BNF with the same nonterminals. This is less straightforward, but it too could be automated through the use of any technique for enumerating paths through a graph. The method is to create a BNF rule for each diagram. The left-hand side of the rule is the name of the diagram. Then, enumerate all paths from the initial to final state of the diagram and list them, joined by "or" as the right hand side of the resulting BNF rule. An additional complication arises when the graph contains loops; these would be translated into recursive BNF rules.

3. Either notation with terminals only into the other with nonterminals. This requires rules to create nonterminals, and it cannot be done well by an algorithm. It is possible to specify a system using no nonterminals (except for those used recursively). It is also possible to imagine an algorithm that creates a nonterminal for every group of terminals that appears more than once in the specification. But neither of these techniques is likely to yield a particularly clear or helpful specification of a user interface.

4. BNF with or without nonterminals into state diagram with terminals only. This can be done with an algorithm and, in fact, has practical use. BNF-driven parser-generators operate by solving this problem and then simulating the resulting state machine [25]. The result is not a readable specification, because it contains the full (high- through low-level) description of the system in a single unwieldy state diagram; but it is useful in that it can be executed by a simple interpreter.

Given a particular choice of nonterminals, then, it is possible to convert algorithmically between state diagram and BNF specifications. The principal difference one finds is that a BNF-based specification with very few nonterminals (relative to the complexity of the system) is generally more difficult to understand than the corresponding state diagram. Conversely, a state diagram is generally easier to understand when it has fewer nonterminals than a BNF description of the same system. For this reason, the direct state diagram translation of a typical BNF specification is likely to contain many very simple diagrams; while the BNF translation of a typical state diagram will generally contain only a few, very complicated rules.

Thus an important difference between the two types of techniques is that BNF requires more nonterminals to make it readable. BNF tends to be less readable than state diagrams when there are only a few nonterminals, so it forces the writer to create more. This could be described as requiring the naming of otherwise irrelevant intermediate objects. State diagrams require the creation (and, for a text representation, the naming) of intermediate states, the names of which are often not of interest. Text representations of state diagram-based specifications tend to be longer than BNF-based specifications since each new nonterminal adds two or more lines to the former and only one to the latter, but this is a consequence of the particular text representations most widely used for the state diagrams. It is also related to the fact that, while they are compact, the typical BNF-based techniques do not specify the timing of actions conveniently. The choice of BNF- or state transition diagram-based notations is less important in determining the clarity of a specification than the selection of the nonterminal symbols used to describe a system. While the effects of making a particular choice of nonterminals are different for the two types of specifications (*i.e.*, BNF requires more nonterminals), choosing them is a crucial step in constructing a specification—and it requires human understanding of the underlying system.

4.4. Other Specification Techniques

Nearly all of the work in specifying user interfaces has used a specification language based on BNF or state transition diagrams. However, a few other techniques have been proposed.

Moran [53] provides a LISP-like notation for describing a user's knowledge about a computer system at several levels, from an overall task analysis 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 transition 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.

Barron [54] attempts to use Petri nets to describe the user interface of a system, but the resulting description appears unnecessarily complex.

The Human Operator Simulator [55] is an extensive software system for simulating human operator performance (hand movements, short-term memory recall, and the like). In doing this, the system uses its own, rather detailed FORTRAN-like language for describing user interfaces. This language tends to favor computer- rather than user-initiated dialogues, in that it specifies what the user must do next, rather than the range of options available to him.

Green [56] proposes a technique that closely resembles the specifications used to describe software modules, but, because it describes the user's command language as a collection of disjointed procedure calls, it fails to capture the overall surface structure of that language clearly.

5. Introduction to Specification Examples

The following section presents a collection of specifications and fragments of specifications of user interfaces that is intended to apply several of the specification techniques discussed above to some example systems. The purpose of studying these examples is to compare the relative merits of several representative specification techniques by considering how well they can

be applied to a common set of problems. Several of these examples are specifications of subsets of systems or very small systems. In these cases, just enough of the system is specified to shed some light on the specification techniques themselves. In other cases, only certain techniques are applied to a particular example. This is done where the omitted techniques provide no additional information, because they are very similar to some included technique.

Several of the examples used in these specifications are taken from papers that propose a specification method. In such cases, the reader is directed to the original paper to see that technique applied to the example. Here, one or more other techniques are applied to the same example in case the original example was contrived in a way that made it easy to describe using the technique proposed, but difficult with others. In such cases, applying a specification technique across a variety of examples, as presented in the specifications below, where each example was originally constructed to suit a different technique, will give a clearer picture of the merits of the technique.

The specification techniques used here fall into two classes: those based on Backus-Naur Form (BNF) and those based on state transition diagrams. The syntax for each of the techniques is described briefly below.

5.1. BNF

This is straightforward 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. In some specifications, definition rules have been annotated with Boolean conditions, replies, or actions, placed in braces. Where a condition is given, it 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 reply and perform the action, if any are given. The special token `NULL` represents no input; and the token `RETURN` denotes the carriage-return key. A token or nonterminal name followed by an asterisk* stands for *zero or more instances of* that symbol.

5.2. YACC Input Form

YACC is a compiler-compiler that accepts a BNF description of an input language plus actions and generates an executable system directly from the specification. The specifications are essentially BNF, but some additional syntactic conventions are used, some extra programming features are needed, and actions are written in a real programming language (in this case, C [57]), rather than informally as in the BNF above. In addition, to obtain an executable system, a lexical scanner for recognizing low-level tokens must be provided. This is done using another, very simple specification language, which drives a lexical analyzer-generator, LEX, just as BNF drives the compiler-generator. Each of the specifications given in YACC input form has been tested by annotating its definitions with actions and constructing and executing the system directly from the resulting specification using YACC. See [21] for a description of YACC and LEX, and [58] and [59] respectively, for the specific details of their input syntaxes.

5.3. Graphic State Transition Diagram

The notation for these diagrams follows fairly widely-used conventions. Each state is represented by a circle. Each transition between two states is shown as a labeled, directed arc between two state circles. It is labeled with the name of an input token (or another diagram) and, in some cases, a Boolean condition or an action. The transition will occur if the input token is received (or the other diagram is traversed) and the condition is satisfied; when it does, the system will perform the action. This notation is most similar to Augmented Transition Networks [16] in the way diagrams can impose arbitrary Boolean conditions, invoke actions, and call other diagrams.

It should be noted that all of the diagrams that appear here were generated directly from computer input consisting of the text representation described below plus a small amount of additional positioning information.

5.4. Text Representation of State Transition Diagram

This is a text form of the state transition diagram notation discussed above. Each diagram begins with a header line that gives the name of the diagram and the name(s) of its exit state(s). A list of the transitions that comprise the diagram follows, each represented by a line of the form:

s1: INP →*s2* **act:**ThisAction;

denoting a transition from state `s1` to state `s2` that expects input token `INP` and performs action `ThisAction`. A condition would be specified in a way similar to the action. Instead of an input token `INP`, the name of another diagram, in lower case, could be given, meaning that that diagram would be traversed, and, upon exit from it, a transition to state `s2` would be made. An ampersand in place of the first state, `s1`, means that this transition starts from the same state as the transition listed immediately above it.

No single method found in the literature for representing state transition diagrams has been entirely satisfactory, hence the method used here is a modification and combination of several such methods. Existing methods vary in their use of conditions, outputs, and actions. Also, the ability to give diagram names in place of input tokens means that the resulting system may need to scan several diagrams nondeterministically to determine which scan will actually be completed; existing methods vary on this point. In this notation, 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.

While these text representations seem fairly difficult to follow, they are sufficient to generate automatically the graphic state transition diagrams mentioned above. They are also sufficient to drive a simulator of a user interface, in much the same way much that a BNF specification drives the YACC simulator.

5.5. Singer's Text Representation of State Transition Diagrams

Singer [7] uses a text state diagram notation that is considerably different from that of most other investigators. In his syntax, underlined (italicized) lower case strings represent "semantic" actions and predicates, which manipulate and test an external data structure. They are specified separately from the state diagram, in a programming language-like notation. As a result, more information about the system is contained in that auxiliary specification than is the case for any of the other techniques used here. Complete details of his notation are given in [7].

5.6. Comments

In all of the notations used here, comments are enclosed between `/*` and `*/`.

6. Specification Examples

6.1. Parnas' Examples

Parnas [38] presents examples of portions of a user interface and proposes representing such by state transition diagrams. His paper presents graphic state diagrams (the third technique discussed above) for them. Two of his examples (his Figures 1 and 3) are shown here, using each of the remaining four techniques discussed above:

6.1.1. BNF

```
/*
 * Parnas' Figure 1 in BNF
 *
 * Assumes that s1 and s2 in the paper
 * are to be considered the final states
 * (i.e., you must reach one of them to exit
```

```
 * successfully, but it doesn't matter which).
 */

cmd::=        loadnews* logon12

loadnews::=   LOAD {reply: description of load}
|             NEWS {reply: news}

logon12::=    logon {reply: welcome #1}
|             logon2 {reply: welcome #2}


/*
 * Parnas' Figure 3 in BNF
 *
 * Assumes that s3 and s4 are to be considered the final states
 */

cmd::=        name_of_program {reply: give parameter} illegal* legal

illegal::=    illegal_value {reply: error}

legal::=      legal_value {reply: prompt 1}
|             legal_value_2 {reply: prompt 2}
```

6.1.2. YACC Input

This specification is similar to that above, except that it is executable using YACC. YACC does not provide a compact notation for *zero or more* instances of a symbol, so an extra rule is needed, corresponding to each such construct in the previous specification.

A simple lexical analyzer also had to be specified for each of the two examples. For the first example, it simply returns the token name NEWS for the user input "NEWS," the token name LOAD for user input "LOAD," and the tokens LOG1 or LOG2 for user inputs corresponding to two types of logon commands, "LOGON" and "LOGON2."

```
/*
 * Parnas' Figure 1 as YACC input
 *
 * Assumes that s1 and s2 in the paper
 * are to be considered the final states
 * (i.e., you must reach one of them to exit
 * successfully, but it doesn't matter which).
 */

%token LOAD NEWS LOG1 LOG2

cmd: loadnews logon12 ;

loadnews: singleloadnews | singleloadnews loadnews ;

singleloadnews
  : /* NULL */
  | LOAD {reply(description of load)}
  | NEWS {reply(news)}
  ;
```

```
logon12
  : LOG1 {reply(welcome #1)}
  | LOG2 {reply(welcome #2)}
  ;

/*
 * Parnas' Figure 3 as YACC input
 *
 * Assumes that s3 and s4 are to be considered the final states
 */

%token NAME ILLEGAL LEGAL1 LEGAL2

cmd:        NAME {reply(give parameter)} illegals legal ;

illegals:   /* NULL */ | illegal | illegals illegal ;

illegal:    ILLEGAL {reply(error)} ;

legal:      LEGAL1 {reply(prompt 1)}
|           LEGAL2 {reply(prompt 2)} ;
```

6.1.3. Text State Diagram

```
/*
 * Parnas' Figures 1 and 3 in Text State Diagram Notation
 */
```

figure1 →*(s1,s2)*

init: LOAD →*init* **act:**reply(description of load);
init: NEWS →*init* **act:**reply(news);
init: logon →*s1* **act:**reply(welcome #1);
init: logon2 →*s2* **act:**reply(welcome #2);
;

figure3 →*(s3,s4)*

s1: name_of_program →*s2* **act:**reply(give parameter);

s2: legal_value →*s3* **act:**reply(prompt 1);
s2: legal_value_2 →*s4* **act:**reply(prompt 2);
s2: illegal_value →*s2* **act:**reply(error);
;

6.1.4. Singer State Diagram

```
/*
 * Parnas' Figures 1 and 3 in Singer's Notation
 */

FIGURE1 → (return) ::
```

```
I:      read-something → (IA)

IA:     something-is-logon? → (IA1)
      & something-is-logon2? → (IA2)
      & something-is-news? → (IA3)
      & something-is-load? → (IA4)

IA1:    print-welcome1 → (S1)

IA2:    print-welcome2 → (S2)

IA3:    display-news → (I)

IA4:    print-description-of-load → (I)

FIGURE3 → (return) ::

S1:     read-something → (S1A)

S1A:    something-is-name-of-program? → (S1A1)

S1A1:   print-giveparameter1 → (S2)

S2:     read-something → (S2A)

S2A:    something-is-legal-value? → (S2A1)
      & something-is-legal-value2? → (S2A2)
      & something-is-illegal-value? → (S2A3)

S2A1:   print-prompt1 → (S3)

S2A2:   print-prompt2 → (S4)

S2A3:   print-error → (S2)
```

6.2. Wasserman and Stinson's Example System

Wasserman and Stinson [43] present examples of a very simple banking system specified as a state diagram (similar to the graphic state diagram notation discussed above) and also in a tabular form (similar to the text state diagram discussed).

6.2.1. BNF

The top level of their banking system can be specified in BNF fairly straightforwardly.

```
/*
 * Top Level of Wasserman and Stinson's Example in BNF
 */

pgm::=    cmd cmd*

cmd::=    "HELP"
|         "CR" credit
|         "DB" ACCTNO AMOUNT
|         "BAL" ACCTNO
|         "NEW" newacct
```

```
|          "CLOSE" ACCTNO
|          "QUIT"

credit::= ACCTNO AMOUNT
```

6.2.2. YACC

Several YACC specifications are presented here; they describe several variations of Wasserman and Stinson's system. Each version uses the same, simple lexical scanner. It returns each alphabetic character as a separate token; the tokens HELP and DELIM for two special characters; ACCTNO for an integer; and AMOUNT for a real number.

First, Wasserman and Stinson's entire system (the top level plus the one lower-level command they specify) is presented, including the actions, which are written in C and enclosed in braces.

```
/*
 * Wasserman and Stinson's Example as YACC input
 */

%token ACCTNO AMOUNT HELP DELIM

pgm
  : cmd
  | pgm cmd
  ;

cmd
  : 'c' 'r' credit

  | 'd' 'b' ACCTNO AMOUNT {
        bal[$3]-=$4;
        printf("Account %d balance is now %10.2f",$3,bal[$3]);
    }

  | 'b' 'a' 'l' ACCTNO {
        printf("Account %d current balance is %10.2f",$4,bal[$4]);
    }

  | 'n' 'e' 'w' newacct {
        printf("new account");
    }

  | 'c' 'l' 'o' 's' 'e' ACCTNO {
        bal[$6] = -9999;
        printf("Account %d closed",$6);
    }

  | 'q' 'u' 'i' 't' {
        printf("Goodbye");
        exit();
    }

  | error {
        printf("Illegal Command");
    }
```

```
  ;

credit
  : scantodelim getacct getamt {
        bal[$2]+=$3;
        printf("Account %d balance is now %10.2f",$2,bal[$2]);
    }
  ;

scantodelim: nondelims DELIM;

nondelims: /* NULL */ | nondelim | nondelim nondelims;

nondelim: alphchar | HELP | ACCTNO | AMOUNT;

getacct: helpjunk1 ACCTNO { $$ = $2;};

helpjunk1: singlehj1 | singlehj1 helpjunk1 | /* NULL */;

singlehj1
  : HELP {
        printf("CRED1: ACCOUNT NUMBER");
        printf("HLP2: ACCOUNT NUMBER IS 7 DIGITS");
    }

  | scanhj1 {
        if (errcount>=3) printf("FAIL!"); else errcount+=1;
        printf("CRED1: ACCOUNT NUMBER");
    }
  ;

scanhj1: nonhj1s DELIM;

nonhj1s: nonhj1 | nonhj1 nonhj1s;

nonhj1: alphchar | AMOUNT;

getamt: helpjunk2 AMOUNT { $$ = $2;};

helpjunk2: singlehj2 | singlehj2 helpjunk2 | /* NULL */;

singlehj2
  : HELP {
        printf("CRED2: AMOUNT");
        printf("HLP3: AMOUNT IN DOLLARS & CENTS");
    }

  | scanhj2 {
        if (errcount>=3) printf("FAIL!"); else errcount+=1;
        printf("CRED2: AMOUNT");
    }
  ;

scanhj2: nonhj2s DELIM;
```

```
nonhj2s: nonhj2 | nonhj2 nonhj2s;

nonhj2: alphchar | ACCTNO;

alphchar:
    'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
    'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z';

newacct: /* NULL */ ;
```

Because Wasserman and Stinson's example system is so simple, it does not provide a serious test of the specification technique. Thus, two modifications of their system, which provide some of the features that would be found in a realistic system, are specified below. For simplicity, these versions do not include the lower-level help features for the `credit` command that were shown in the specification above. In the first version, a user can type a help key at any point (between two lexical tokens) in the input, and the system will reply with a help message appropriate to the context of the current command. This specification is of interest because this sort of help facility is often cumbersome to describe in BNF (as seen in this example). It requires an extra nonterminal symbol `help` to be specified at every position at which a user could ask for help. Each of these `help` nonterminals then has, as one of its definitions, a null input, since the help request is optional.

```
/*
 * Wasserman and Stinson's Example
 * plus help facility
 * as YACC input
 */

%token ACCTNO AMOUNT HELP DELIM

pgm
  : cmd
  | pgm cmd
  ;

cmd
  : help1

  | 'c' 'r' help2 credit

  | 'd' 'b' help3 ACCTNO help4 AMOUNT {
        bal[$4]-=$6;
        printf("Account %d balance is now %10.2f",$4,bal[$4]);
    }

  | 'b' 'a' 'l' help5 ACCTNO {
        printf("Account %d current balance is %10.2f",$5,bal[$5]);
    }

  | 'n' 'e' 'w' help6 newacct {
        printf("new account");
    }

  | 'c' 'l' 'o' 's' 'e' help7 ACCTNO {
        bal[$7] = -9999;
```

```
        printf("Account %d closed",$7);
    }

  | 'q' 'u' 'i' 't' {
        printf("Goodbye");
        exit();
    }

  | error {
        printf("Illegal Command");
    }
  ;

credit
  : ACCTNO help8 AMOUNT {
        bal[$1]+=$3;
        printf("Account %d balance is now %10.2f",$1,bal[$1]);
    }
  ;

newacct: /*NULL*/ ;

help1: helpkey {printf("legal commands are...");}        | /*NULL*/;

help2: helpkey {printf("credit command next--acctno");} | /*NULL*/;

help3: helpkey {printf("account number for debit");}     | /*NULL*/;

help4: helpkey {printf("amount for debit");}             | /*NULL*/;

help5: helpkey {printf("account number for balance");}   | /*NULL*/;

help6: helpkey {printf("new account command");}          | /*NULL*/;

help7: helpkey {printf("acct you want closed");}         | /*NULL*/;

help8: helpkey {printf("amount for credit");}            | /*NULL*/;

helpkey: HELP;
```

Another modification of their system is shown below. Here, the names and arguments of the commands can be entered in any arbitrary order. This is a desirable capability for many practical systems; it is possible when the types of inputs can all be distinguished lexically (rather than at a higher level). However, without extending the syntax of a BNF- or state diagram-based notation, it is cumbersome to specify, because it requires enumerating all possible permutations of the input tokens. The example below shows how this would be done in BNF notation for YACC.

```
/*
 * Wasserman and Stinson's Example
 * plus ability to accept commands and arguments in any order
 * as YACC input
 */

%token ACCTNO AMOUNT HELP DELIM
```

```
pgm
  : cmd
  | pgm cmd
  ;

cmd
  : 'c' 'r' credit

  | debit

  | balance

  | 'n' 'e' 'w' newacct {
        printf("new account");
    }

  | close

  | 'q' 'u' 'i' 't' {
        printf("Goodbye");
        exit();
    }

  | error {
        printf("Illegal Command");
    }
  ;

credit
  : ACCTNO AMOUNT {
        bal[$1]+=$2;
        printf("Account %d balance is now %10.2f",$1,bal[$1]);
    }

  | AMOUNT ACCTNO {
        bal[$2]+=$1;
        printf("Account %d balance is now %10.2f",$2,bal[$2]);
    }
  ;

debit
  : 'd' 'b' ACCTNO AMOUNT {
        bal[$3]-=$4;
        printf("Account %d balance is now %10.2f",$3,bal[$3]);
    }

  | 'd' 'b' AMOUNT ACCTNO {
        bal[$4]-=$3;
        printf("Account %d balance is now %10.2f",$4,bal[$4]);
    }

  | AMOUNT ACCTNO 'd' 'b' {
        bal[$2]-=$1;
        printf("Account %d balance is now %10.2f",$2,bal[$2]);
```

```
    }

  | ACCTNO AMOUNT 'd' 'b' {
        bal[$1]-=$2;
        printf("Account %d balance is now %10.2f",$1,bal[$1]);
    }

  | AMOUNT 'd' 'b' ACCTNO {
        bal[$4]-=$1;
        printf("Account %d balance is now %10.2f",$4,bal[$4]);
    }

  | ACCTNO 'd' 'b' AMOUNT {
        bal[$1]-=$4;
        printf("Account %d balance is now %10.2f",$1,bal[$1]);
    }
  ;

balance
  : 'b' 'a' 'l' ACCTNO {
        printf("Account %d current balance is %10.2f",$4,bal[$4]);
    }

  | ACCTNO 'b' 'a' 'l' {
        printf("Account %d current balance is %10.2f",$1,bal[$1]);
    }
  ;

close
  : 'c' 'l' 'o' 's' 'e' ACCTNO {
        bal[$6] = -9999;
        printf("Account %d closed",$6);
    }

  | ACCTNO 'c' 'l' 'o' 's' 'e' {
        bal[$1] = -9999;
        printf("Account %d closed",$1);
    }
  ;

newacct: /* NULL */ ;
```

6.2.3. Singer State Diagram

Again, the top level of the banking system is specified first, in a relatively straightforward manner.

```
/*
 * Top Level of Wasserman and Stinson's Example
 * in Singer's Notation
 *
 * In each state the additional rule:
 *      anything-else → (errorstate)
 * is intended.
 */
```

```
MAIN → (return) ::
init:   PROCESS-CMD → (return)

PROCESS-CMD → (ret) ::

init:   read-a-token → (S)

S:      token-is-cr? → (credit)
      & token-is-db? → (debit)
      & token-is-bal? → (balance)
      & token-is-new? → (newacct)

credit: CREDIT-CMD → (ret)

debit:  DEBIT-CMD → (ret)

balance:BALANCE-CMD → (ret)

newacct:NEWACCT-CMD → (ret)
```

Next, one of the modifications to Wasserman and Stinson's example discussed above is specified. This is the version in which the names and arguments of the commands can be entered in arbitrary orders. There are two ways to specify this system in Singer's notation. The first one, below, relegates all of the detail about how commands are entered into the definitions (not given here) of the semantic domain predicates of the form *command-type-is-*....

```
/*
 * Wasserman and Stinson's Example
 * plus ability to accept commands and arguments in any order
 * in Singer's Notation
 */

MAIN → (return) ::

init:   get-command → (1)

1:      parse-command → (2)

2:      command-type-is-debit? → (debit)
      & command-type-is-credit? → (credit)
      & command-type-is-balance? → (balance)
      & command-type-is-newacct? → (newacct)

debit:  do-debit-actions → (init)

credit: do-credit-actions → (init)

balance:do-balance-actions → (init)

newacct:do-newacct-actions → (init)
```

The second way to specify this modification of Wasserman and Stinson's example in this notation is to extend Singer's formalism to permit nondeterminism, in a way analogous to that

used by the other state diagram notations presented here.

```
/*
 * Wasserman and Stinson's Example
 * plus ability to accept commands and arguments in any order
 * in a modification of Singer's Notation that permits
 * non-determinism
 *
 * Also, this specification includes the counterpart of the
 * BNF rule, pgm::= (one-or-more)cmd
 *
 * In each state the additional rule:
 *      anything-else → (errorstate)
 * is intended.
 */

MAIN → (ok) ::
init:   PROCESS-CMD → (ok)
ok:     PROCESS-CMD → (ok)

PROCESS-CMD → (ret) ::
init:   CREDIT → (ret)              /*
        DEBIT → (ret)                * note that this is a
        BALANCE → (ret)              * nondeterministic rule
        NEWACCT → (ret)              * for state init
        CLOSE → (ret)                */

CREDIT → (ret) ::
init:   read-a-token → (S)

S:      token-is-acctno? → (1)
      & token-is-amount? → (2)

1:      read-a-token → (A)
A:      token-is-amount? → (proc)

2:      read-a-token → (B)
B:      token-is-acctno? → (proc)

proc:   do-credit-actions → (ret)

DEBIT → (ret) ::
init:   read-a-token → (S)

S:      token-is-acctno? → (1)
      & token-is-amount? → (2)
      & token-is-db? → (3)

1:      read-a-token → (A)
A:      token-is-amount? → (4)
      & token-is-db? → (5)

2:      read-a-token → (B)
B:      token-is-acctno? → (6)
      & token-is-db? → (7)
```

```
3:      read-a-token → (C)
C:      token-is-acctno? → (8)
      & token-is-amount? → (9)

4:      read-a-token → (D)
D:      token-is-db? → (proc)

5:      read-a-token → (E)
E:      token-is-amount? → (proc)

6:      read-a-token → (F)
F:      token-is-db? → (proc)

7:      read-a-token → (G)
G:      token-is-acctno? → (proc)

8:      read-a-token → (H)
H:      token-is-amount? → (proc)

9:      read-a-token → (I)
I:      token-is-acctno? → (proc)

proc:   do-debit-actions → (ret)

BALANCE → (ret) ::
init:   read-a-token → (S)

S:      token-is-acctno? → (1)
      & token-is-bal? → (2)

1:      read-a-token → (A)
A:      token-is-bal? → (proc)

2:      read-a-token → (B)
B:      token-is-acctno? → (proc)

proc:   do-balance-actions → (ret)
```

6.3. Reisner's System

Reisner [9] represents the user interface to a simple, but real system in BNF. This is in contrast with the artificial (and nearly trivial) example systems seen above. Her paper presents a complete specification of one interactive graphics system in BNF, except that the system's actions and replies are not specified. The specifications below attempt to represent the same system using state diagrams. This example is rich enough to reveal that there are several ways to transform a BNF specification into a state diagram, and some of these possibilities are shown below.

6.3.1. Straightforward Translation from BNF to State Diagram

This first version is simply a one-to-one mapping from BNF. The result is more cumbersome than BNF and not much more illuminating, except possibly that the state diagram notation emphasizes exactly what the user can do at each point in a dialogue, while that informa-

tion is only implicit in BNF.

```
/*
 * Reisner's Example in Text State Diagram Notation
 * Version 0 -- Subset of the original system only
 */

picture →return
init:           colored_shape →ok
ok:             colored_shape →ok
&               ANY →return
;

colored_shape →ret
init:           color →s
&               shape →c
s:              shape →ret
c:              color →ret
;

color →ret
init:           new_color →ret
&               old_color →ret
&               starting_default_color →ret
;

new_color →ret
init:           CURSOR_IN_RED →ret
&               CURSOR_IN_BLUE →ret
&               CURSOR_IN_GREEN →ret
;

old_color →ret
init:           ANY →ret
;

starting_default_color →ret
init:           ANY →ret
;

shape →ret
init:           discrete_shape →ret
&               continuous_shape →ret
&               text_shape →ret
;

discrete_shape →ret
init:           separate_d_shape →ret
&               connected_d_shape →ret
;
```

```
separate_d_shape →ret
init:           select_separate_d_shape →sa
sa:             describe_separate_d_shape →ret
;
```

6.3.2. BNF to State Diagram, with Some Nonterminals Removed

This version still retains the same basic organization as Reisner's original BNF specification, but it removes some of the nonterminals that were necessary in BNF, but are not necessary in state diagram notation.

```
/*
 * Reisner's Example in Text State Diagram Notation
 * Version 1
 *
 * Note:  State numbers below correspond to the number of
 * Reisner's rule that contains the + that required this
 * extra state.
 */

picture →return
init:           colored_shape →ok
ok:             colored_shape →ok
&               ANY →return
;

colored_shape →ret
init:           color →s
&               shape →c
s:              shape →ret
c:              color →ret
;

color →ret
/*
 * Note: This nonterminal looks as though it should also have
 * been collapsed in the original BNF to
 * color::= CURSOR_IN_RED ¦ CURSOR_IN_BLUE ¦ CURSOR_IN_GREEN ¦ NULL
 */
init:           CURSOR_IN_RED →ret
&               CURSOR_IN_BLUE →ret
&               CURSOR_IN_GREEN →ret
&               ANY →ret
;

shape →ret
init:           select_old_d_shape /* actually = NOTHING */ →s9
&               select_line →s13
&               select_box →s13
&               select_circle →s13
&               select_horizontal →s9
```

```
&               select_vertical →s9
&               select_old_c_shape /* actually = NOTHING */ →s68
&               select_c_switch →s71
&               describe_text_shape →ret

s9:             CURSOR_AT_LINE_POINT_1 →s34
&               CURSOR_AT_CIRCLE_CENTER →s34
&               CURSOR_AT_BOX_CORNER →s34
&               CURSOR_AT_H_LINE_POINT_1 →s47
&               CURSOR_AT_V_LINE_POINT_1 →s47

s13:            GO_1 →s9
s34:            START_GO_2 →s33
s33:            CURSOR_AT_LINE_POINT_2 →s39
&               CURSOR_AT_CIRCUMFERENCE →s39
&               CURSOR_AT_DIAGONAL_CORNER →s39
s39:            END_GO_2 →s55
s47:            START_GO_2 →s46
s46:            CURSOR_AT_H_LINE_END_POINT_ON_Y_AXIS →s39
&               CURSOR_AT_V_LINE_END_POINT_ON_X_AXIS →s39

s55:            CURSOR_AT_LINE_POINT_2 →s39x
&               CURSOR_AT_CIRCUMFERENCE →s39x
&               CURSOR_AT_DIAGONAL_CORNER →s39x
&               select_horizontal →s63
&               select_vertical →s63
&               ANY →ret

s39x:           END_GO_2 →s55
s63:            CURSOR_AT_H_LINE_END_POINT_ON_Y_AXIS →s39x
&               CURSOR_AT_V_LINE_END_POINT_ON_X_AXIS →s39x

s71:            GO_1 →s68
s68:            set_knob →s73a
s73a:           ROTATE_KNOB_FULL_COUNTERCLOCKWISE →s83a
s83a:           POSITION_CURSOR →s83b
s83b:           ROTATE_KNOB_FULL_CLOCKWISE →s73b
&               ROTATE_KNOB_PARTIAL_CLOCKWISE →s73b

s73b:           ROTATE_KNOB_FULL_CLOCKWISE →s85
&               ROTATE_KNOB_PARTIAL_CLOCKWISE →s85
&               ROTATE_KNOB_PARTIAL_COUNTERCLOCKWISE →s85
&               ANY →s85

s85:            MOVE_CURSOR →s73c
&               ANY →s73c

s73c:           ROTATE_KNOB_FULL_COUNTERCLOCKWISE →ret
;

select_line →ret
init:           BOX_SWITCH_DOWN →s17
&               CONTINUOUS_SWITCH_DOWN →s17
```

```
&               HORIZ_SWITCH_DOWN →s17
&               VERT_SWITCH_DOWN →s17
s17:            LINE_SWITCH_UP →ret
;

select_box →ret
init:           LINE_SWITCH_DOWN →s20
&               CONTINUOUS_SWITCH_DOWN →s20
&               HORIZ_SWITCH_DOWN →s20
&               VERT_SWITCH_DOWN →s20
s20:            BOX_SWITCH_UP →ret
;

select_circle →ret
init:           CONTINUOUS_SWITCH_DOWN →s23
&               VERT_SWITCH_DOWN →s23
&               HORIZ_SWITCH_DOWN →s23
s23:            LINE_SWITCH_UP →s25a
&               BOX_SWITCH_UP →s25b
s25a:           BOX_SWITCH_UP →ret
s25b:           LINE_SWITCH_UP →ret
;

select_horizontal →ret
init:           LINE_SWITCH_DOWN →s26
&               BOX_SWITCH_DOWN →s26
&               VERT_SWITCH_DOWN →s26
&               CONTINUOUS_SWITCH_DOWN →s26
s26:            HORIZ_SWITCH_UP →ret
;

select_vertical →ret
init:           LINE_SWITCH_DOWN →s29
&               BOX_SWITCH_DOWN →s29
&               HORIZ_SWITCH_DOWN →s29
&               CONTINUOUS_SWITCH_DOWN →s29
s29:            VERT_SWITCH_UP →ret
;

select_c_switch →ret
init:           CONTINUOUS_SWITCH_UP →s72a
&               select_line →s72b
&               select_box →s72b
&               select_circle →s72b
s72a:           select_line →ret
&               select_box →ret
&               select_circle →ret
s72b:           CONTINUOUS_SWITCH_UP →ret
;

set_knob →ret
init:           WIDTH_ANGLE_SWITCH_DOWN →s75a
```

```
&               WIDTH_ANGLE_SWITCH_UP →s76a

s75a:           ROTATE_KNOB_FULL_CLOCKWISE →s75b
&               ROTATE_KNOB_PARTIAL_CLOCKWISE →s75b
s75b:           WIDTH_ANGLE_SWITCH_UP →ret

s76a:           ROTATE_KNOB_FULL_CLOCKWISE →s76b
&               ROTATE_KNOB_PARTIAL_CLOCKWISE →s76b
s76b:           WIDTH_ANGLE_SWITCH_DOWN →ret
;

describe_text_shape →ret
init:           ADDITIVE_BLOCKED_SWITCH_UP →s91a
&               ADDITIVE_BLOCKED_SWITCH_DOWN →s91a
&               SINGLE_DOUBLE_SWITCH_UP →s91b
&               SINGLE_DOUBLE_SWITCH_DOWN →s91b

s91a:           SINGLE_DOUBLE_SWITCH_UP →s91c
&               SINGLE_DOUBLE_SWITCH_DOWN →s91c
s91b:           ADDITIVE_BLOCKED_SWITCH_UP →s91c
&               ADDITIVE_BLOCKED_SWITCH_DOWN →s91c
s91c:           POSITION_CURSOR →s95

s95:            START_GO_2 →s94
s94:            SYMBOL →s94
&               TYPING_CONTROL_OPERATION →s94
&               ANY →ret
;
```

6.3.3. BNF to State Diagram, Completely Reorganized

Finally, the following specification represents a complete revision of the BNF specification, using a different overall organization. Rewriting the specification as a state diagram from the start results in a much clearer specification than the previous two versions. Thus, the previous versions really constitute state diagram notation superimposed on a BNF-based organization. While the two notations are formally equivalent, each works best when the nonterminal symbols used are chosen with a view toward the particular notation. An additional benefit of this version is that it handles the few dependencies that Reisner could not express in BNF, but had to describe in footnotes.

```
/*
 * Reisner's Example in Text State Diagram Notation
 * Version 2
 */

picture →return
init:           colored_shape →ok
ok:             colored_shape →ok
&               ANY →return
;

colored_shape →ret
```

```
init:       color →s
&           shape →c
s:          shape →ret
c:          color →ret
;

color →ret
init:       CURSOR_IN_RED →ret
&           CURSOR_IN_BLUE →ret
&           CURSOR_IN_GREEN →ret
&           ANY →ret
;

shape →ret
init:       line1 →l
&           box1 →b
&           circle1 →c
&           horiz1 →hv
&           vert1 →hv
&           continuous →ret
&           text →ret

l:          line2 →l
&           ANY →ret

b:          box2 →b
&           ANY →ret

c:          circle2 →c
&           ANY →ret

hv:         horiz2 →hv
&           vert2 →hv
&           ANY →ret
;

line1 →ret
init:       select_line →s13 act:prev:=line;
&           ANY cond:prev=line; →s9
s13:        GO_1 →s9
s9:         CURSOR_AT_LINE_POINT_1 →s34
s34:        START_GO_2 →s33
s33:        CURSOR_AT_LINE_POINT_2 →s39
s39:        END_GO_2 →ret
;

box1 →ret
init:       select_box →s13 act:prev:=box;
&           ANY cond:prev=box; →s9
s13:        GO_1 →s9
s9:         CURSOR_AT_BOX_CORNER →s34
s34:        START_GO_2 →s33
```

```
s33:            CURSOR_AT_DIAGONAL_CORNER →s39
s39:            END_GO_2 →ret
;

circle1 →ret
init:           select_circle →s13 act:prev:=circle;
&               ANY cond:prev=circle; →s9
s13:            GO_1 →s9
s9:             CURSOR_AT_CIRCLE_CENTER →s34
s34:            START_GO_2 →s33
s33:            CURSOR_AT_CIRCUMFERENCE →s39
s39:            END_GO_2 →ret
;

horiz1 →ret
init:           select_horiz →s9 act:prev:=horiz;
&               ANY cond:prev=horiz; →s9
s9:             CURSOR_AT_H_LINE_END_POINT_ON_Y_AXIS →s34
s34:            START_GO_2 →s33
s33:            CURSOR_AT_H_LINE_END_POINT_ON_Y_AXIS →s39
s39:            END_GO_2 →ret
;

vert1 →ret
init:           select_vert →s9 act:prev:=vert;
&               ANY cond:prev=vert; →s9
s9:             CURSOR_AT_V_LINE_END_POINT_ON_X_AXIS →s34
s34:            START_GO_2 →s33
s33:            CURSOR_AT_V_LINE_END_POINT_ON_X_AXIS →s39
s39:            END_GO_2 →ret
;

line2 →ret
init:           CURSOR_AT_LINE_POINT_2 →s39x
s39x:           END_GO_2 →ret
;

box2 →ret
init:           CURSOR_AT_DIAGONAL_CORNER →s39x
s39x:           END_GO_2 →ret
;

circle2 →ret
init:           CURSOR_AT_CIRCUMFERENCE →s39x
s39x:           END_GO_2 →ret
;

horiz2 →ret
init:           select_horizontal →s63
s63:            CURSOR_AT_H_LINE_END_POINT_ON_Y_AXIS →s39x
s39x:           END_GO_2 →ret
;
```

```
vert2 →ret
init:       select_vertical →s63
s63:        CURSOR_AT_V_LINE_END_POINT_ON_X_AXIS →s39x
s39x:       END_GO_2 →ret
;

select_line →ret
init:       BOX_SWITCH_DOWN →s17
&           CONTINUOUS_SWITCH_DOWN →s17
&           HORIZ_SWITCH_DOWN →s17
&           VERT_SWITCH_DOWN →s17
s17:        LINE_SWITCH_UP →ret
;

select_box →ret
init:       LINE_SWITCH_DOWN →s20
&           CONTINUOUS_SWITCH_DOWN →s20
&           HORIZ_SWITCH_DOWN →s20
&           VERT_SWITCH_DOWN →s20
s20:        BOX_SWITCH_UP →ret
;

select_circle →ret
init:       CONTINUOUS_SWITCH_DOWN →s23
&           VERT_SWITCH_DOWN →s23
&           HORIZ_SWITCH_DOWN →s23
s23:        LINE_SWITCH_UP →s25a
&           BOX_SWITCH_UP →s25b
s25a:       BOX_SWITCH_UP →ret
s25b:       LINE_SWITCH_UP →ret
;

select_horizontal →ret
init:       LINE_SWITCH_DOWN →s26
&           BOX_SWITCH_DOWN →s26
&           VERT_SWITCH_DOWN →s26
&           CONTINUOUS_SWITCH_DOWN →s26
s26:        HORIZ_SWITCH_UP →ret
;

select_vertical →ret
init:       LINE_SWITCH_DOWN →s29
&           BOX_SWITCH_DOWN →s29
&           HORIZ_SWITCH_DOWN →s29
&           CONTINUOUS_SWITCH_DOWN →s29
s29:        VERT_SWITCH_UP →ret
;

continuous →ret
init:       select_c_switch →s71
&           ANY →s68
s71:        GO_1 →s68
```

```
s68:        set_knob →s73a
s73a:       ROTATE_KNOB_FULL_COUNTERCLOCKWISE →s83a

s83a:       POSITION_CURSOR →s83b
s83b:       ROTATE_KNOB_FULL_CLOCKWISE →s73b
&           ROTATE_KNOB_PARTIAL_CLOCKWISE →s73b

s73b:       ROTATE_KNOB_FULL_CLOCKWISE →s85
&           ROTATE_KNOB_PARTIAL_CLOCKWISE →s85
&           ROTATE_KNOB_PARTIAL_COUNTERCLOCKWISE →s85
&           ANY →s85

s85:        MOVE_CURSOR →s73c
&           ANY →s73c

s73c:       ROTATE_KNOB_FULL_COUNTERCLOCKWISE →ret
;

select_c_switch →ret
init:       CONTINUOUS_SWITCH_UP →s72a
&           select_line →s72b
&           select_box →s72b
&           select_circle →s72b
s72a:       select_line →ret
&           select_box →ret
&           select_circle →ret
s72b:       CONTINUOUS_SWITCH_UP →ret
;

set_knob →ret
init:       WIDTH_ANGLE_SWITCH_DOWN →s75a
&           WIDTH_ANGLE_SWITCH_UP →s76a

s75a:       ROTATE_KNOB_FULL_CLOCKWISE →s75b
&           ROTATE_KNOB_PARTIAL_CLOCKWISE →s75b
s75b:       WIDTH_ANGLE_SWITCH_UP →ret

s76a:       ROTATE_KNOB_FULL_CLOCKWISE →s76b
&           ROTATE_KNOB_PARTIAL_CLOCKWISE →s76b
s76b:       WIDTH_ANGLE_SWITCH_DOWN →ret
;

text →ret
init:       ADDITIVE_BLOCKED_SWITCH_UP →s91a
&           ADDITIVE_BLOCKED_SWITCH_DOWN →s91a
&           SINGLE_DOUBLE_SWITCH_UP →s91b
&           SINGLE_DOUBLE_SWITCH_DOWN →s91b

s91a:       SINGLE_DOUBLE_SWITCH_UP →s91c
&           SINGLE_DOUBLE_SWITCH_DOWN →s91c
s91b:       ADDITIVE_BLOCKED_SWITCH_UP →s91c
&           ADDITIVE_BLOCKED_SWITCH_DOWN →s91c
s91c:       POSITION_CURSOR →s95
```

```
s95:        START_GO_2 →s94

s94:        SYMBOL →s94
&           TYPING_CONTROL_OPERATION →s94
&           ANY →ret
;
```

6.4. Example Military Message System Commands

Two example commands suitable for a simple Military Message System are specified below.

The `Login` command prompts the user to enter his or her name. If the system 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 whom 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.

The actions invoked by these commands are expressed compactly and conveniently in terms of the MMS Intermediate Command Language [18]. It provides a high-level model of the functions that a message system can perform and a notation for describing them that is well suited to use in the action portion of this type of specification.

6.4.1. BNF

```
/*
 * Example MMS Login command
 */

login::=        badpw* goodpw {reply: "Enter security level"} getseclevel

badpw::=        loguser onetry PASSWORD
                        {cond: vPASSWORD≠GETPASSWD_USER(vUSER)
                        reply: "Incorrect password--start again"}

goodpw::=       loguser PASSWORD {cond: vPASSWORD=GETPASSWD_USER(vUSER)}

|               loguser onetry PASSWORD {cond: vPASSWORD=GETPASSWD_USER(vUSER)}

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

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

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

onetry::=       PASSWORD {cond: vPASSWORD≠GETPASSWD_USER(vUSER)
```

```
                        reply: "Incorrect password--reenter it"}

getseclevel::=  badsl* {reply: "Your security level is Unclassified"
                        act: CREATE_SESSION(vUSER,vPASSWORD,Unclassified)}

|               badsl* SECLEVEL {cond: vSECLEVEL<=GETCLEARANCE_USER(vUSER)
                        act: CREATE_SESSION(vUSER,vPASSWORD,vSECLEVEL)}

badsl::=        SECLEVEL {cond: vSECLEVEL>GETCLEARANCE_USER(vUSER)
                        reply: "Security level too high--reenter it"}

/*
 * Example MMS Reply command
 */

reply::=        REPLY getid {reply: "Enter text field"
                        act: replybuf:=OPENFOREDIT_MSG(replyid)}
                        TEXT {act: SETTEXT_MSG(vTEXT,replybuf)}
                        extras* {act: UPDATE_MSG(replyid,replybuf),
                        CLOSEEDIT_MSG(replyid)}

getid::=        MSGID {act: replyid:=REPLY_MSG(vMSGID)}

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

extras::=       extratos | extrraccs

extratos::=     TO toaddressee toaddressee*

toaddressee::=  ADDRESSEE {act: SETTO_MSG(replybuf,
                        GETTO_MSG(replybuf)+vADDRESSEE)}

extraccs::=     CC ccaddressee ccaddressee*

ccaddressee::=  ADDRESSEE {act: SETCC_MSG(replybuf,
                        GETCC_MSG(replybuf)+vADDRESSEE)}
```

6.4.2. Graphic State Diagram

Figure 3 shows the Login and Reply commands in state diagram notation.

6.4.3. Text State Diagram

```
/*
 * Example MMS Login command
 */
```

login →*end*

start: LOGIN →*getu* **act:**reply("Enter name");

getu: USER **cond:**not EXISTS_USER(vUSER); →*getu*
act:reply("Incorrect user name--reenter it");

```
&          USER cond:EXISTS_USER(vUSER); →getpw act:reply("Enter password");

getpw:     PASSWORD cond:vPASSWORD=GETPASSWD_USER(vUSER); →getsl
           act:reply("Enter security level");

&          PASSWORD cond:vPASSWORD≠GETPASSWD_USER(vUSER); →badpw
           act:reply("Incorrect password--reenter it");

badpw:     PASSWORD cond:vPASSWORD=GETPASSWD_USER(vUSER); →getsl
           act:reply("Enter security level");

&          PASSWORD cond:vPASSWORD≠GETPASSWD_USER(vUSER); →start
           act:reply("Incorrect password--start again");

getsl:     SECLEVEL cond:vSECLEVEL>GETCLEARANCE_USER(vUSER); →getsl
           act:reply("Security level too high--reenter it");

&          SECLEVEL cond:vSECLEVEL<=GETCLEARANCE_USER(vUSER); →end
           act:CREATE_SESSION(vUSER,vPASSWORD,vSECLEVEL);

&          ANY →end act:{ reply("Your security level is Unclassified");
           CREATE_SESSION(vUSER,vPASSWORD,Unclassified)};
;

/*
 * Example MMS Reply command
 */

reply →end

start:     REPLY →getid

getid:     MSGID →gettext act:{ reply("Enter text field");
           replyid:=REPLY_MSG(vMSGID); replybuf:=OPENFOREDIT_MSG(replyid)};

&          ANY →gettext act:{ reply("Enter text field");
           replyid:=REPLY_MSG(CurrentMsg);
           replybuf:=OPENFOREDIT_MSG(replyid)};

gettext:   TEXT →getextras act:SETTEXT_MSG(vTEXT,replybuf);

getextras: extratos →getextras

&          extraccs →getextras

&          ANY →end act:{ UPDATE_MSG(replyid,replybuf);
           CLOSEEDIT_MSG(replyid)};
;

extratos →end

start:     TO →t1
```

t1: ADDRESSEE →*t2*
act:SETTO_MSG(replybuf,GETTO_MSG(replybuf)+vADDRESSEE);

t2: ADDRESSEE →*t2*
act:SETTO_MSG(replybuf,GETTO_MSG(replybuf)+vADDRESSEE);

& ANY →*end*
;

extraccs →*end*

start: CC →*c1*

c1: ADDRESSEE →*c2*
act:SETCC_MSG(replybuf,GETCC_MSG(replybuf)+vADDRESSEE);

c2: ADDRESSEE →*c2*
act:SETCC_MSG(replybuf,GETCC_MSG(replybuf)+vADDRESSEE);

& ANY →*end*
;

6.5. Tape Recorder

Finally, it is useful to consider a user interface for a system that is less tied to the conventional sort of terminal input and output seen in the examples above, in order to see how the specification techniques can apply to hardware used for newer, less conventional user interfaces. For a simple example of a system with non-text input and output primitives, the most basic type of tape recorder is specified using several notations.

Its only controls are push buttons for PLAY, RECORD, Fast Forward (FF), REWIND, STOP, and EJECT. Pressing each of these corresponds to a terminal symbol in the specifications below. In addition the user can INSERT a tape and can change the setting of a volume control (CHANGE_VOLUME). In defining actions, the symbol vCHANGE_VOLUME denotes the new value to which the control was set in the most recent CHANGE_VOLUME operation. It is assumed that, whenever a new tape is inserted, it is positioned at the beginning of the tape (*i.e.*, this resembles a reel, rather than cartridge, recorder). Finally, to add an interesting detail, assume that the recorder automatically stops whenever it reaches the end of the tape.

6.5.1. BNF

```
/*
 * Simple Tape Recorder in BNF
 */

operate_recorder::= change_vol* insert_and_run

insert_and_run::=   NULL

|                   INSERT {l:=start of tape} run* EJECT

run::=              change_vol* prff* change_vol*

prff::=             PLAY {begin playing from location l at volume v,
                          begin increasing l at low speed}
                          change_vol* {continue playing at volume v}
```

```
                        stop_or_eot {cease increasing l, cease playing}

|                   RECORD {begin recording from location l at volume v,
                            begin increasing l at low speed}
                            change_vol* {continue recording at volume v}
                            stop_or_eot {cease increasing l, cease recording}

|                   REWIND {begin decreasing l at high speed}
                            change_vol* stop_or_bot {cease decreasing l}

|                   FF {begin increasing l at high speed}
                            change_vol* stop_or_eot {cease increasing l}

stop_or_eot::=      STOP

|                   NULL {cond: l=end of tape}

stop_or_bot::=      STOP

|                   NULL {cond: l=beginning of tape}

change_vol::=       CHANGE_VOLUME {v:=CHANGE_VOLUME}
```

6.5.2. Graphic State Diagram

Figure 4 presents a state diagram specification of the same tape recorder.

6.5.3. Text State Diagram

```
/*
 * Simple Tape Recorder in Text State Diagram Notation
 */
```

operate_recorder →*end*

empty: CHANGE_VOLUME →*empty* **act:**v:=vCHANGE_VOLUME;

& INSERT →*stopped* **act:**l:=start of tape;

& ANY →*end*

stopped: CHANGE_VOLUME →*stopped* **act:**v:=vCHANGE_VOLUME;

& PLAY →*playing* **act:**{ begin playing from location l at volume v; begin increasing l at low speed };

& RECORD →*recording* **act:**{ begin recording at location l at volume v; begin increasing l at low speed };

& REWIND →*rewinding* **act:**begin decreasing l at high speed;

&	FF →*ffing* **act:**begin increasing l at high speed;
&	EJECT →*empty*
playing:	CHANGE_VOLUME →*playing* **act:**{ v:=vCHANGE_VOLUME; continue playing at volume v };
&	STOP →*stopped* **act:**{ cease increasing l; cease playing };
&	ANY **cond:**l=end of tape; →*stopped* **act:**{ cease increasing l; cease playing };
recording:	CHANGE_VOLUME →*recording* **act:**{ v:=vCHANGE_VOLUME; continue recording at volume v };
&	STOP →*stopped* **act:**{ cease increasin.g l; cease recording };
&	ANY **cond:**l=end of tape; →*stopped* **act:**{ cease increasing l; cease recording };
rewinding:	CHANGE_VOLUME →*rewinding* **act:**v:=vCHANGE_VOLUME;
&	STOP →*stopped* **act:**cease decreasing l;
&	ANY **cond:**l=start of tape; →*stopped* **act:**cease decreasing l;
ffing:	CHANGE_VOLUME →*ffing* **act:**v:=vCHANGE_VOLUME;
&	STOP →*stopped* **act:**cease increasing l;
&	ANY **cond:**l=end of tape; →*stopped* **act:**cease increasing l;
;	

7. Conclusions

Nearly all techniques that have been used to specify user interfaces for interactive systems are based on Backus-Naur Form or state transition diagrams. While the two are formally equivalent, their surface differences have an important effect on their comprehensibility. In particular, notations based on state transition diagrams explicitly contain the concept of a state and the rules associated with it, while it is implicit in BNF-based notations. Since the concept of state is important in representing sequence in the behavior of an interactive system, state diagrams are preferable to BNF in this regard.

Existing techniques based on state diagrams vary considerably in their syntax and expressive ability. A synthesis of the best features of several such notations is recommended. Moreover, in either notation, the use and careful choice of meaningful nonterminal symbols is vital to the overall clarity of the specification.

An important criterion for a user interface specification is that its principal constructs—the main nonterminal symbols and states—represent concepts that will be meaningful to users and helpful to them in constructing their own mental models of the system. In this way, a mapping can be maintained from the user interface specification to the user documentation.

A synthesis of the features of several state diagram-based notations was thus selected to specify the user interface for a prototype military message system. The explicit description of states in this notation makes the sequence of actions clearer than in BNF. In addition, some of the states correspond to users' own notions of what a system does ("text entry" state, "logged-out" state).

Acknowledgments

This work has benefited from discussions with L. Chmura and with my colleagues on the Military Message System Project, M. Cornwell, C. Heitmeyer, and C. Landwehr. It was supported by the Secure Military Message System Project of the Naval Electronic Systems Command under the direction of H.O. Lubbes.

References and Annotated Bibliography

[1] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, pp. 1053-1058, 1972.

[2] D.L. Parnas, "On a "Buzzword" Hierarchical Structure," *Proc. IFIPS Congress*, pp. 336-339, 1974.

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

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

[5] R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Comm. ACM*, vol. 26, pp. 259-264, 1983.

[6] H. Ledgard, J.A. Whiteside, A. Singer, and W. Seymour, "The Natural Language of Interactive Systems," *Comm. ACM*, vol. 23, pp. 556-563, 1980.

[7] A. Singer, "Formal Methods and Human Factors in the Design of Interactive Languages," Ph.D. dissertation, Computer and Information Science Dept., Univ. Massachusetts, 1979.

Presents a specification of a non-trivial system using a modified version of state transition diagrams. This contrasts with much other work in the field, which describes fairly trivial examples.

[8] IBM, "CMS User's Guide," Document GC20-1818-0, IBM Corporation, 1976.

[9] P. Reisner, "Formal Grammar and Human Factors Design of an Interactive Graphics System," *IEEE Transactions on Software Engineering*, vol. SE-7, pp. 229-240, 1981.

Uses BNF specifications of two user interfaces to predict differences in the performance of users using the two systems and then tests the predictions experimentally.

[10] C.L. Heitmeyer and S.H. Wilson, "Military Message Systems: Current Status and Future Directions," *IEEE Transactions on Communications*, vol. COM-28, pp. 1645-1654, 1980.

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

[12] J.D. Foley and V.L. Wallace, "The Art of Graphic Man-Machine Conversation," *Proceedings of the IEEE*, vol. 62, pp. 462-471, 1974.

Uses a state diagram in a short example, but gives very little detail on this method. The paper also contains good general advice on the design of man-machine interfaces.

[13] J.W. Backus and others, "Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, vol. 3, pp. 299-314, 1960.

Describes BNF notation and uses it to specify a static language.

[14] D.R. Lenorovitz and H.R. Ramsey, "A Dialogue Simulation Tool for Use in the Design of Interactive Computer Systems," *Proc. 21st Annual Meeting of the Human Factors Society*, Santa Monica, Calif., 1977.

Uses BNF specification with an interactive action specified for each rule to describe a user interface to a simulator.

[15] M.E. Conway, "Design of a Separable Transition-Diagram Compiler," *Comm. ACM*, vol. 6, pp. 396-408, 1963.

An early use of state diagrams annotated with actions to construct a compiler for a static language.

[16] W.A. Woods, "Transition Network Grammars for Natural Language Analysis," *Comm.*

ACM, vol. 13, pp. 591-606, 1970.

Describes the use of a state transition diagram, augmented by associating actions with each transition, for analyzing static languages.

[17] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976. pp. 209-217

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

[19] P.R. Hanau and D.R. Lenorovitz, "Prototyping and Simulation Tools for User/Computer Dialogue Design," *Computer Graphics*, vol. 14, no. 3, pp. 271-278, Seattle, 1980.

Describes a simulation tool that accepts a specification of a user interface in BNF with associated actions and then creates a mockup of the user interface.

[20] L.E. Heindel and J.T. Roberto, *LANG-PAK: An Interactive Language Design System*, American Elsevier, New York, 1975.

Describes the LANG-PACK compiler-compiler, a system that accepts a specification of a user interface in BNF plus actions (written in FORTRAN) and produces an implementation of it.

[21] S.C. Johnson, "Language Development Tools on the Unix System," *IEEE Computer*, vol. 13, no. 8, pp. 16-21, 1980.

Describes YACC, a system that accepts a specification of a user interface in BNF (along with actions written in C) and produces an implementation of that interface.

[22] M. Rodeh, "Command Languages," Research Report RJ3234, IBM Corporation, 1981.

Describes a system that uses a BNF specification plus semantic actions specified in PL/I to describe a user interface to an interpreter which then provides a prototype of the specified system.

[23] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, "AWK -- A Pattern Scanning and Processing Language," *Software -- Practice and Experience*, vol. 9, pp. 267-279, 1979.

Describes a system whose user interface was constructed using YACC.

[24] S. Guthery, "DDA: An Interactive and Extensible Language for Data Display and Analysis," *Computer Graphics*, vol. 10, no. 1, pp. 24-31, 1976.

Describes a system whose user interface was constructed using the LANG-PACK compiler-compiler.

[25] A.V. Aho and S.C. Johnson, "LR Parsing," *Computing Surveys*, vol. 6, pp. 99-124, 1974.

Describes parsing techniques that can be used with BNF specifications.

[26] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

Describes the theory of parsing techniques for BNF specifications.

[27] R.S. Fenchel, "An Integral Approach to User Assistance," *ACM SIGSOC Bulletin*, vol. 13, no. 2-3, pp. 98-104, 1982.

Uses a BNF specification to describe a user interface to an interpreter. The interpreter can then provide interactive help messages using information in the BNF specification.

[28] H.W. Lawson, Jr., M. Bertran, and J. Sanagustin, "The Formal Definition of Man/Machine Communication," *Software -- Practice and Experience*, vol. 8, pp. 51-58, 1978.

Uses BNF plus actions to specify a fairly simple interactive user interface.

[29] D.W. Embley, "Empirical and Formal Language Design Applied to a Unified Control Construct for Interactive Computing," *International Journal of Man-Machine Studies*, vol. 10, pp. 197-216, 1978.

Uses a BNF description of a static user interface (a programming language control construct) to predict user performance.

[30] B. Shneiderman, "Multi-party Grammars and Related Features for Defining Interactive

Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-12, pp. 148-154, 1982.

Describes a new type of BNF grammar in which each nonterminal symbol is associated with either the user or the computer (or one of several parties to a multi-way conversation).

[31] S. Feyock, "Transition Diagram-Based CAI/HELP Systems," *International Journal of Man-Machine Studies*, vol. 9, pp. 399-413, 1977.

Uses a state diagram as the data base for a user help facility.

[32] J. Darlington, W. Dzida, and S. Herda, "The Role of Excursions in Interactive Systems," *International Journal of Man-Machine Studies*, vol. 18, pp. 101-112, 1983.

Examines psychological issues related to user models of interactive systems. In particular, the authors find that a state transition automaton is a good representation of such user models.

[33] S.P. Guest, "The Use of Software Tools for Dialogue Design," *International Journal of Man-Machine Studies*, vol. 16, pp. 263-285, 1982.

[34] H. Thimbleby, "Character-level Ambiguity: Consequences for User Interface Design," *International Journal of Man-Machine Studies*, vol. 16, pp. 211-225, 1982.

[35] J.W. Brown, "Controlling the Complexity of Menu Networks," *Comm. ACM*, vol. 25, pp. 412-418, 1982.

[36] E. Denert, "Specifications and Design of Dialogue Systems with State Diagrams," *International Computing Symposium*, pp. 417-424, North-Holland, Amsterdam, 1977.

Uses state diagrams to describe an interactive system. To implement the system, the state diagram is translated directly into an ALGOL program.

[37] D.C. Engelbart and W.K. English, "A Research Center for Augmenting Human Intellect," *Proc. 1968 Fall Joint Computer Conference*, pp. 395-410, AFIPS, 1968.

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

Proposes state diagrams for describing user interfaces. The diagrams describe transitions among terminal states, rather than all possible system states.

[39] M.R. Barnett, "Implementing Interactive Systems in PL/1," *Proc. ONLINE 72 Conference*, pp. 267-289, 1972.

Uses a simple one-level state diagram to represent the user interface to a small system. Most of the article is concerned with the specific details associated with implementing such a user interface in PL/1.

[40] B. Dwyer, "A User-Friendly Algorithm," *Comm. ACM*, vol. 24, pp. 556-561, 1981.

Uses a state diagram representation to drive a fairly simple program that implements the user interface described by the diagram.

[41] D.J. Kasik, "Controlling User Interaction," *Computer Graphics*, vol. 10, pp. 109-115, 1976.

Describes theory and implementation of a triply-linked tree for representing the user interface of menu-driven systems.

[42] MUMPS Development Committee, *MUMPS Language Standard*, American National Standards Institute, New York, 1977.

Uses state diagrams to describe the MUMPS computer language.

[43] A.I. Wasserman and S.K. Stinson, "A Specification Method for Interactive Information Systems," *Proc. Specifications of Reliable Software Conference*, pp. 68-79, 1979. IEEE Catalog No. 79CH1401-9C.

Uses a state transition diagram to specify the user interface to a simple example system.

[44] A.I. Wasserman and D.T. Shewmake, "Rapid Prototyping of Interactive Information Systems," *ACM SIGSOFT Software Engineering Notes*, vol. 7, pp. 171-180, 1982.

Describes a prototype builder that uses a state transition diagram specification of a user interface. Semantic actions are given as Unix Shell commands.

[45] M.S. Fox and A.J. Palay, "The BROWSE System: Phase II and Future Directions," ZOG Memo, Computer Science Dept., Carnegie-Mellon University, 1979.

[46] G. Robertson, A. Newell, and K. Ramakrishna, "ZOG: A Man-Machine Communication Philosophy," Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[47] G. Robertson, D. McCracken, and A. Newell, "The ZOG Approach to Man-Machine Communication," Technical Report, Computer Science Department, Carnegie-Mellon University, 1979.

[48] J. Schultz and L. Davis, "The Technology of PROMIS," *Proceedings of the IEEE*, vol. 67, pp. 1237-1244, 1979.

[49] G.V. Bochmann, "Finite State Description of Communication Protocols," *Computer Networks*, vol. 2, pp. 361-372, 1978.

Uses coupled finite state automata to represent two machines communicating with each other via a protocol. Then, it is possible to prove properties of the protocol using the state transition diagram representation.

[50] C.A. Sunshine, "Survey of Protocol Definition and Verification Techniques," *Computer Networks*, vol. 2, pp. 346-350, 1978.

Discusses techniques that have been used to describe (machine-to-machine) communication protocols. Most of them are based either on state diagram notation or else on a procedural description of the algorithm.

[51] J.E. Hopcroft and J.D. Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

[52] J.F. Hueras, "A Formalization of Syntax Diagrams as k-Deterministic Language Recognizers," M.S. thesis, Computer Science Dept., Univ. California, Irvine, 1978.

Investigates state diagram notation with sub-diagram calls and relates it to the formal models of automata theory.

[53] T.P. Moran, "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *International Journal of Man-Machine Studies*, vol. 15, pp. 3-50, 1981.

The Command Language Grammar attempts to capture the knowledge a user has about a system. It yields a very detailed description of several levels of a command language system.

[54] J. Barron, "Dialogue and Process Design for Interactive Information Systems Using Taxis," *Proc. ACM SIGOA Conference on Office Information Systems*, pp. 12-20, 1982.

[55] M.I. Streib, F.A. Glenn, and R.J. Wherry, "The Human Operator Simulator," Technical Report 1320, Analytics Inc., 1978.

Describes an elaborate software system for simulating human operator performance (hand movements, short-term memory recall, and the like).

[56] M. Green, "A Methodology for the Specification of Graphical User Interface," *Computer Graphics*, vol. 15, no. 3, pp. 99-108, 1981.

Uses a specification technique much like software module specifications to describe the commands a user can give.

[57] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1978.

[58] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler," in *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, N.J., 1979.

[59] M.E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," in *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, N.J., 1979.

Figure 1. Components of the Military Message System. (Boxes represent software modules; lines represent data flow.)

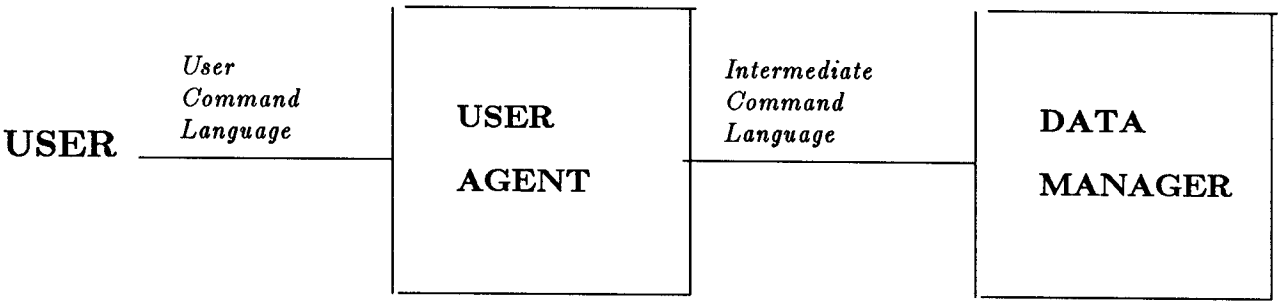

Figure 2. Example State Diagram Specification

sendcmd

(1) **Act:** reply("Draft name can be...")
(2) **Act:** reply("Enter draft name")
(3) **Act:** reply("Recipient name can be...")
(4) **Act:** reply("Enter recipient name")

Figure 3. State Diagram Representation of Example MMS Commands

Figure 3. State Diagram Representation of Example MMS Commands (continued)

Figure 3. State Diagram Representation of Example MMS Commands (continued)

login

(1) **Act:** reply("Enter name")
(2) **Cond:** not EXISTS_USER(vUSER)
(3) **Act:** reply("Incorrect user name--reenter it")
(4) **Cond:** EXISTS_USER(vUSER)
(5) **Act:** reply("Enter password")
(6) **Cond:** vPASSWORD=GETPASSWD_USER(vUSER)
(7) **Act:** reply("Enter security level")
(8) **Cond:** vPASSWORD≠GETPASSWD_USER(vUSER)
(9) **Act:** reply("Incorrect password--reenter it")
(10) **Cond:** vPASSWORD=GETPASSWD_USER(vUSER)
(11) **Act:** reply("Enter security level")
(12) **Cond:** vPASSWORD≠GETPASSWD_USER(vUSER)
(13) **Act:** reply("Incorrect password--start again")
(14) **Cond:** vSECLEVEL>GETCLEARANCE_USER(vUSER)
(15) **Act:** reply("Security level too high--reenter it")
(16) **Cond:** vSECLEVEL<=GETCLEARANCE_USER(vUSER)
(17) **Act:** CREATE_SESSION(vUSER,vPASSWORD,vSECLEVEL)
(18) **Act:** { reply("Your security level is Unclassified");
CREATE_SESSION(vUSER,vPASSWORD,Unclassified)}

reply

(1) **Act:** { reply("Enter text field"); replyid:=REPLY_MSG(vMSGID);
replybuf:=OPENFOREDIT_MSG(replyid)}
(2) **Act:** { reply("Enter text field"); replyid:=REPLY_MSG(CurrentMsg);
replybuf:=OPENFOREDIT_MSG(replyid)}
(3) **Act:** SETTEXT_MSG(vTEXT,replybuf)
(4) **Act:** { UPDATE_MSG(replyid,replybuf); CLOSEEDIT_MSG(replyid)}

extratos

(1) **Act:** SETTO_MSG(replybuf,GETTO_MSG(replybuf)+vADDRESSEE)
(2) **Act:** SETTO_MSG(replybuf,GETTO_MSG(replybuf)+vADDRESSEE)

extraccs

(1) **Act:** SETCC_MSG(replybuf,GETCC_MSG(replybuf)+vADDRESSEE)
(2) **Act:** SETCC_MSG(replybuf,GETCC_MSG(replybuf)+vADDRESSEE)

Figure 4. State Diagram Representation of Tape Recorder

Figure 4. State Diagram Representation of Tape Recorder (continued)

operate_recorder

(1) **Act:** v:=vCHANGE_VOLUME
(2) **Act:** l:=start of tape
(3) **Act:** v:=vCHANGE_VOLUME
(4) **Act:** { begin playing from location l at volume v; begin increasing l at low speed }
(5) **Act:** { begin recording at location l at volume v; begin increasing l at low speed }
(6) **Act:** begin decreasing l at high speed
(7) **Act:** begin increasing l at high speed
(8) **Act:** { v:=vCHANGE_VOLUME; continue playing at volume v }
(9) **Act:** { cease increasing l; cease playing }
(10) **Cond:** l=end of tape
(11) **Act:** { cease increasing l; cease playing }
(12) **Act:** { v:=vCHANGE_VOLUME; continue recording at volume v }
(13) **Act:** { cease increasin.g l; cease recording }
(14) **Cond:** l=end of tape
(15) **Act:** { cease increasing l; cease recording }
(16) **Act:** v:=vCHANGE_VOLUME
(17) **Act:** cease decreasing l
(18) **Cond:** l=start of tape
(19) **Act:** cease decreasing l
(20) **Act:** v:=vCHANGE_VOLUME
(21) **Act:** cease increasing l
(22) **Cond:** l=end of tape
(23) **Act:** cease increasing l