A Specification Language for Direct-Manipulation User Interfaces

ROBERT J. K. JACOB
Naval Research Laboratory

A direct-manipulation user interface presents a set of visual representations on a display and a repertoire of manipulations that can be performed on any of them. Such representations might include screen buttons, scroll bars, spreadsheet cells, or flowchart boxes. Interaction techniques of this kind were first seen in interactive graphics systems; they are now proving effective in user interfaces for applications that are not inherently graphical. Although they are often easy to learn and use, these interfaces are also typically difficult to specify and program clearly.

Examination of direct-manipulation interfaces reveals that they have a coroutine-like structure and, despite their surface appearance, a peculiar, highly moded dialogue. This paper introduces a specification technique for direct-manipulation interfaces based on these observations. In it, each locus of dialogue is described as a separate object with a single-thread state diagram, which can be suspended and resumed, but retains state. The objects are then combined to define the overall user interface as a set of coroutines, rather than inappropriately as a single highly regular state transition diagram. An inheritance mechanism for the interaction objects is provided to avoid repetitiveness in the specifications. A prototype implementation of a user-interface management system based on this approach is described, and example specifications are given.

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

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Direct manipulation, specification language, state transition diagram, user-interface management system (UIMS)

1. INTRODUCTION

A direct-manipulation user interface presents its user with a set of visual representations of objects on a display and a repertoire of generic manipulations that can be performed on any of them [26]. Some of these techniques were first seen in interactive graphics systems; they are now proving effective in user interfaces for applications that are not inherently graphical. With a direct manipulation interface, the user seems to operate directly *on* the objects in the computer instead of carrying on a dialogue *about* them. Instead of using a command language to describe operations on objects that are frequently invisible, the user "manipulates" objects visible on a graphic display.

Portions of this work were supported by the Space and Naval Warfare Systems Command under the direction of H. O. Lubbes.
Author's address: Naval Research Laboratory, Code 5590, Washington, D.C., 20375.

This ability to manipulate displayed objects has been identified as *direct engagement* [13]. The displayed objects are active in the sense that they are affected by each command issued; they are not the fixed outputs of one execution of a command, frozen in time. They are also usable as inputs to subsequent commands. The ultimate success of a direct-manipulation interface also requires directness in the form of low *cognitive distance* [13], the mental effort needed to translate from the input actions and output representations to the operations and objects of the problem domain itself. The visual metaphor chosen to depict the problem domain should thus be easy for the user to translate to and from that domain, and the actions required to effect a command should be closely related to the meaning of the command in the problem domain.

This paper examines the characteristics of a direct-manipulation dialogue as seen from the user's point of view. Then it introduces a specification language for describing such dialogues, in which the structure of the language is modeled on the characteristics of the dialogue. Its purpose is to describe direct-manipulation user interfaces from a high-level view, with a structure designed to match the user's perceptions (rather than the programmer's). Since the new language can be executed, it can also serve as the basis for a user-interface management system for direct-manipulation interfaces.

2. SPECIFYING A DIRECT-MANIPULATION USER INTERFACE

It is useful to be able to write a specification of the user interface of a computer system before building it, because the interface designer can thereby describe and study a variety of possible user interfaces without having to code them. Such a specification should describe precisely the user-visible behavior of the interface, but should not constrain its implementation. Specification techniques for describing the user-visible behavior of conventional user interfaces without reference to implementation details are gaining currency; most have been based on state transition diagrams [6, 15, 18, 22, 29, 33] or BNF [23, 25] (and a few on other models listed below); there are some reasons to prefer the state diagrams [15].

If the specification language itself can be executed or compiled, it can also serve as the basis for a user-interface management system (UIMS). To be useful, a UIMS needs a convenient and understandable way for the user-interface designer to describe the desired interface. The choice of specification language is thus at the heart of the design of a UIMS. UIMSs have been built using BNF or other grammar-based specifications [21], state-transition-diagram-based specifications [4, 14, 24, 32], programming-language-based specifications [17], frames [9], flow diagrams [35], and other models [1, 2]. More recently, several investigators have used an object-oriented approach [8, 19, 28]. Research is also under way in describing user interfaces by example, where the interface designer is not concerned with a programming or specification language [20].

Although direct manipulation can make systems easy to learn and use, such user interfaces have proved more difficult to construct and specify. Direct-manipulation interfaces have some important differences from other styles of interfaces, and these must be understood in order to develop an appropriate specification technique for them. Although state-transition-diagram-based notations have proved effective and powerful for specifying conventional user

interfaces, they must be modified to handle direct-manipulation interfaces. State diagrams tend to emphasize the modes or states of a system and the sequence of transitions from one state to another. Although direct-manipulation user interfaces initially appear to be modeless and thus unsuited to this approach, they will be shown below to have a particular, highly regular moded structure, which can be exploited in devising a specification technique for them.

3. THE STRUCTURE OF A DIRECT-MANIPULATION DIALOGUE

In order to develop an appropriate specification language for direct-manipulation interfaces, it is necessary to identify the basic structure of such an interface as the user sees it. The goal of this specification method is not strictly compactness or ease of programming, but rather capturing the way the end user sees the dialogue. Many existing specification techniques could be extended in various ways to describe the unusual aspects of direct-manipulation dialogues. However, the real problem is not just to find *some* way to describe the user interface (since, after all, assembly language can do that job), but to find a language that captures the user's view of a direct-manipulation interface as perspicuously as possible and with as few ad hoc features and extensions to the specification technique as possible. The object is to describe the interface or dialogue between the system and its end user, as seen by that user, rather than to describe the structure of the system or its components at some other level.

First, consider what a dialogue specification should describe. Trying to capture the layout and precise appearance of the display of a direct-manipulation interface at every turn would make the top level of the dialogue specification excessively detailed and complex. Instead, the initial specification should be centered around the *sequence* of abstract input and output *events* that comprise the dialogue. The syntax of an interactive user interface—whether conventional or direct manipulation—is effectively described by such a sequence of input and output events, with the specification of the meanings of the events in terms of specific input actions or display images deferred [14]. The abstract input or output events themselves are called *tokens* and are then described individually in separate specifications. Information about display representation and layout is isolated there, rather than as part of the description of the syntax of the dialogue.

This decomposition of direct-manipulation dialogues follows the model of general user–computer dialogues introduced by Foley and Wallace [5, 6]. The sequence of input and output tokens comprises the syntactic level, while the individual token descriptions comprise the lexical level. The semantic level is defined by a collection of procedures that implement the functional requirements of the system; they are invoked from the syntactic-level specification. This three-level separation has been used to good effect in user-interface management systems [4, 8, 16, 21]. Separating the abstract dialogue sequence and overall display organization (syntactic) description from the precise input and output format (lexical) description is of particular importance for direct-manipulation interfaces, because such interfaces typically provide rapid and rich graphical feedback and may vary the appearance of the display considerably during a dialogue. Users may also be permitted to rearrange windows and other images arbitrarily to suit their preference. Despite such variations there are some

underlying cognitive characteristics of the dialogue, which are more stable. A more fundamental characterization of the dialogue than moment-to-moment display appearance should thus be identified and used as the foundation for a clear specification; the sequence of abstract events or tokens is proposed to provide this foundation.

The issue did not arise with early user interfaces based on teleprinters or scrolling display terminals. The sequence of specific input and output events precisely determined the appearance of the display in a simple and straightforward way. Later display terminals added some special commands, such as clear screen, vertical tab, or cursor motions, which disrupted the relationship between sequence of inputs and outputs and display appearance. These have required some extensions to conventional specification techniques [25, 31]. With a full graphic display, however, much more complex user interfaces have been built. It is still true in principle that the sequence of input and output events completely determines the final appearance of the display, but in a far less straightforward way—a way that the user-interface specifier should not have to understand. The specification writer needs to be able to speak about the display appearance at a higher level: the sequence of input and output events. Details about graphical representations, sizes, windows, particular input/output devices, and the like can then be abstracted out of the dialogue specification. Even the choice of particular modes of user–computer communication can be isolated, since an output token can be any discrete, meaningful *event* in the dialogue, including, for example, an audible or tactile output.

Note that building the syntax specification around the sequences of tokens does not preclude semantic-level feedback. For example, as a file icon is dragged over various directory icons, those directories (and only those) into which the user is currently permitted to move that file might be highlighted. The specification technique permits such an operation, but it divides the description of the feedback into its three appropriate aspects. The decision as to which directories should be highlighted is given in the semantic-level specification (it will be a "condition" procedure in the language introduced below); the specification of when in the dialogue such highlighting will occur is given in the syntactic-level specification (as transitions that test the condition and call a highlight token); and the description of the highlighting operation itself is given in the lexical-level specification (as the definition of the highlight token).

Consider next the basic sequence of events in a direct-manipulation dialogue. A direct-manipulation user interface resembles an interacting collection of active and/or responsive objects more than it does a single command language dialogue with the user. The display typically presents a variety of graphical objects. Users can select any of them (most often by moving a cursor). Once selected, the user can begin a dialogue about that object—adjusting a parameter, deleting or moving an object, etc. Each object thus has its own particular dialogue, which the user may activate or deactivate at any time. Further, some object dialogues remember their state between activations. For example, if the user moves the cursor to a type-in field and types a few characters, moves it somewhere else and performs other operations, and then returns to the type-in field, the dialogue within that field would be resumed with the previously entered characters and insertion point

intact. As a better example, if the user had begun an operation that prompted for and required him or her to enter some additional arguments, the user could move to another screen area and do something else before returning to the first area and resuming entry of the arguments where he or she had left them.

Given this structure, it is unnatural, though possible, to describe the user interface of a direct-manipulation system as a conventional dialogue by means of a syntax diagram or other such notation. Instead the user sees a multitude of small dialogues, each of which may be interrupted or resumed under the control of a simple master dialogue. Each of the individual objects on the screen thus has a particular syntax or dialogue associated with it. Each such dialogue can be suspended (typically if the user moves the cursor away) and later resumed at the point from which it was suspended. The relationship between the individual dialogues or branches of the top-level diagram is that of *coroutines*.

So, the basic structure of a direct-manipulation interface is seen to be a collection of individual dialogues connected by an executive that activates and suspends them as coroutines. The specification technique for direct-manipulation interfaces will thus allow the individual dialogues to be specified individually and to exchange control with each other through a coroutine call mechanism.

4. MODES IN THE USER INTERFACE

Many traditional user interfaces are highly *moded*, and this has made it convenient to specify them using state transition diagrams. Modes or states refer to the varying interpretation of a user's input. In each different mode, a user interface may give different meanings to the same input operations [30]. Some use of modes is necessary in most user interfaces, since there are generally not enough distinct brief input operations (e.g., single keystrokes) to map into all the commands of a system. A moded user interface requires that users remember (or the system remind them) of which mode it is in at any time and which different commands or syntax rules apply to each mode. Modeless systems do not require this; the system is always in the same mode, and inputs always have the same interpretation.

Direct-manipulation user interfaces appear to be modeless. Many objects are visible on the screen; and at any time the user can apply any of a standard set of commands to any object. The system is thus nearly always in the same "universal" or "top-level" mode. This is approximately true of some screen editors, but for most other direct-manipulation systems, where the visual representation contains more than one type of component, this is a misleading view. It ignores the input operation of moving the cursor to the object of interest. A clearer view suggests that such a system has many distinct modes. *Moving the cursor to point to a different object is the command to cause a mode change, because once it is moved, the range of acceptable inputs is reduced and the meaning of each of those inputs is determined.* This is precisely the definition of a mode change. For example, moving the cursor to a screen button, such as the "Display" buttons in the message system shown later in Figures 16 and 17, should be viewed as putting the system into a mode where the meaning of the next mouse button click is determined (it displays that message) and the set of permissible inputs is

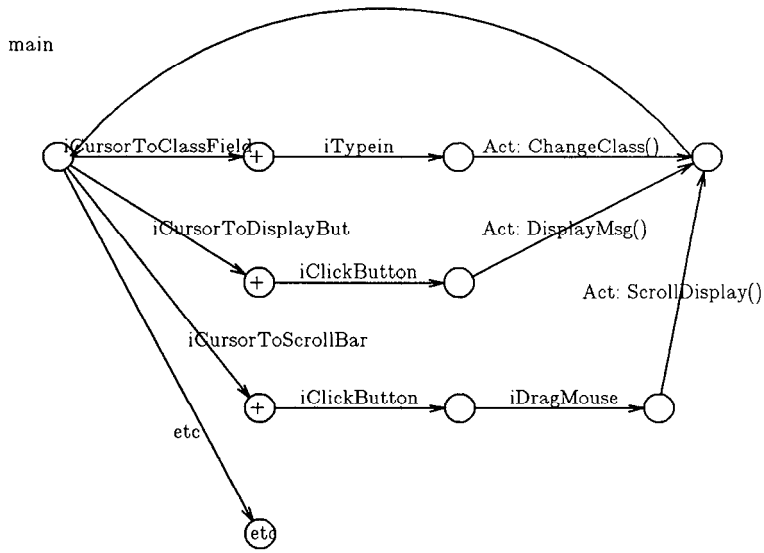


Fig. 1. State-diagram specification of the top level of a simple direct-manipulation user interface.

circumscribed (e.g., keyboard input could be illegal or ignored). Moving the cursor somewhere else would change that mode. As shown in Figure 1, the top level of a typical direct-manipulation interface such as the message-system example could thus be described by a large state diagram with one top-level state and a branch (containing a cursor motion input) leading from it to each mode (marked with a "+"). Each such branch continues through one or more additional states before returning to the top-level state. There is typically no crossover between these branches.

If direct-manipulation user interfaces are not really modeless, why do they appear to have the psychological advantages over moded interfaces that are usually ascribed to modeless ones? The reason is that they make the mode so apparent and easy to change that it ceases to be a stumbling block. The mode is always clearly visible (as the location of a cursor), and it has an obvious representation (simply the echo of the same cursor location just used to enter the mode change command), in contrast to some special flag or prompt. Thus the input mode is always visible to the user. The direct-manipulation approach makes the output display (cursor location to indicate mode) and the related input command (move cursor to change mode) operate through the same visual representation (cursor location). At all times the user knows exactly how to change modes; he or she can never get stuck. It appears, then, that direct-manipulation user interfaces are highly moded, but they are much easier to use than traditional moded interfaces because of the direct way in which the modes are displayed and manipulated.

5. A SPECIFICATION LANGUAGE

Figure 1 shows a typical direct-manipulation user interface represented as a state transition diagram. Although a simple direct-manipulation interface could be

specified in this fashion, it has some shortcomings. The top-level state diagram for each new direct-manipulation interface will be a large, regular, and relatively uninformative diagram with one start state and a self-contained (i.e., no cross-over) path to each mode and thence back to start state. It is essentially the same for any direct-manipulation system and need not be specified anew for each system. Moreover, since the individual paths are usually self-contained and interact with each other in very limited ways, it would be clearer to separate their specifications. A more serious problem with this approach is that there is often a remembered state within some of the paths (partial type-in on a field, an item awaiting confirmation, etc.), which are suspended when the cursor leaves the field and resumed when it reenters. This requires that the paths of the diagram be handled separately. Each path will thus now be specified separately (as a coroutine), and an executive will be given for the outer dialogue loop.

A specification language based on the characteristics found in the foregoing examination of direct-manipulation interfaces can now be described:

—A direct-manipulation interface was found to comprise a collection of many relatively simple individual dialogues. Thus the specification will be centered around a collection of individual objects, called *interaction objects*, each of which will have a separate specification. Each of the dialogues of the direct-manipulation interface will be specified as a separate interaction object with an independent dialogue description.

—The individual dialogues of a direct-manipulation interface were found to be related to each other as a set of coroutines. Thus the specification language will permit the dialogue associated with each interaction object to be suspended and resumed, with retained state, like a coroutine. A simple executive will be defined to manage the overall flow of control. It specifies the interconnection of the interaction object dialogues, allocates input events, and suspends the individual dialogues to relinquish control to others as needed.

—Because of the complexity and variability in the layout of the display of a direct-manipulation interface, it was found that the dialogue should be specified as a sequence of abstract input and output events, with layout and graphic details given separately. Thus the dialogue specification for each interaction object will be written using input and output tokens, which represent input or output events. The dialogue specification will define the possible sequences of input and output tokens. The internals of the tokens themselves will then be specified separately from the dialogue. These token definitions will contain details of layout, graphical representation, and device handling.

—Direct-manipulation interfaces were seen to have definite modes or states, despite their surface appearance. This applied both to the overall structure and to the retained state within each coroutine. Thus state transition diagrams are a suitable notation for describing the individual interaction-object dialogues. The state diagrams will assume coroutine calling between them.

Given this structure, a direct-manipulation user interface will be specified as a collection of individual, possibly mutually interacting interaction objects, organized around the manipulable objects and the loci of remembered state in the dialogue. These objects will often coincide with screen regions or windows, but need not. A typical object might be a screen button, individual type-in field,

scroll bar, or the like. Each such object will be specified separately, and then a standard executive will be defined for the outer dialogue loop. Thus, to describe a direct-manipulation user interface, it will be necessary to

(1) define a collection of interaction objects,
(2) specify their internal behaviors, and
(3) provide a mechanism for combining them into a coordinated user interface.

As noted, a goal of this notation is to capture the way the end user sees the interface. The underlying claim is thus that the user indeed sees the direct-manipulation dialogue as a collection of small, individual objects or dialogues, each suspendable and resumable like a coroutine, joined by a straightforward executive.

The specification language is defined by devising a mechanism for each of the three tasks in the preceding paragraph:

1. *How should the user interface be divided into individual objects?* An interaction object will be the smallest unit with which the user conducts a meaningful, step-by-step dialogue, that is, one that has continuity or syntax. It can be viewed as the smallest unit in the user interface that has a state that is remembered when the dialogue associated with it is interrupted and resumed. In that respect, it is like a window, but in a direct-manipulation user interface, it is generally smaller—a screen button, a single type-in field on a form, or a command line area. It can also be viewed as the largest unit of the user interface over which disparate input events should be serialized and combined into a single stream, rather than divided up and distributed to separate objects. Thus an interaction object is a locus both of maintained state and of input serialization.

2. *How should an input handler for each interaction object be specified?* Observe that, at the level of individual objects, each such object conducts only a single-thread dialogue, with all inputs serialized and with a remembered state whenever the individual dialogue is interrupted by that of another interaction object. Thus a conventional single-thread state diagram is the appropriate representation for the dialogue associated with an individual interaction object. The input handler for each interaction object is specified as a simple state transition diagram.

3. *How should the specifications of the individual objects be combined into an "outer loop" or overall direct-manipulation user interface?* As noted, a direct-manipulation interface could be described with a single, large state diagram, but since the user sees the structure of the user interface as a collection of many semi-independent objects, that is not a particularly perspicuous description. Instead, a standard executive will be defined that embodies the basic structure of a direct-manipulation dialogue and includes the ability to make coroutine calls between individual state diagrams. This executive operates by collecting all of the state diagrams of the individual interaction objects and executing them as a collection of coroutines, assigning input events to them and arbitrating among them as they proceed. To do this, a coroutine call mechanism for activating state diagrams must be defined. This means that whenever a diagram is suspended by a coroutine call to another diagram, the state in the suspended diagram is remembered. Whenever a diagram is resumed by a coroutine call, it will begin executing at the state from which it was last suspended. The executive causes

the state diagram of exactly one of the interaction objects to be active at any one time. As the active diagram proceeds, it reaches each state, examines the next input event, and takes the appropriate transition from that state. It continues in this way until it reaches a state from which no outgoing transition matches the current input. Then, the executive takes over, suspending the current diagram, but remembering its state for later resumption. (It follows that a diagram can only be suspended from a state in which it seeks an input token.) The executive examines the diagrams associated with all the other interaction objects, looking at their current (i.e., last suspended from) states to see which of them can accept the current input. It then resumes (with a coroutine call) whichever diagram has a transition to accept the input. If there is more than one such diagram, one is chosen arbitrarily. In typical designs, however, there will be only one diagram that can accept the input. Since entering and exiting disjoint screen regions will be important input tokens in a typical direct-manipulation interface, this is straightforward to arrange when the interaction objects correspond to screen regions. (In some situations, such conflicts can also be detected by static analysis of the interface specification.) Depending on the overall system design, an input token acceptable to no diagrams could be discarded or treated as a user error. While the language assumes a single top-level executive, the use of component objects and synthetic tokens described below allows the specification to use a deeper hierarchy in describing systems.

The initial design for the executive called for a list of acceptable input events or classes to be associated with each state in each diagram. This list would act like a guard in a guarded command [11] or a **when** clause in a **select/accept** statement in Ada. By associating different guards with different states, a diagram could dynamically adjust the range of inputs that it will accept. The executive for such a system would examine the guard associated with the current state of every diagram in execution to decide which diagram should be called to accept each new input. The current design should be viewed as achieving the same result, even though it does not identify the guards explicitly. What would have been given as the guard for each state is now derived implicitly from the range of inputs on the transitions emanating from that state. This requires somewhat more care in specifying "catchall" transitions, but greatly reduces the redundancy and bulk of the specification. (The operation of the executive is described further in Section 15.)

The new specification language also makes heavy use of techniques of object-oriented programming. The interaction objects themselves are specified and implemented as objects, in the sense of Smalltalk [7] or Flavors [34], and diagram activations and tokens are implemented as messages. The notion of coroutines, however, is superimposed upon the objects as the means for describing how the individual interaction objects are bound together into the top-level dialogue that the user ultimately sees. Other recent work on specifying and building graphical user interfaces has also used an object-oriented approach [8, 19, 28]. Typically, they model the dialogue by a collection of separate objects, each with an input handler. However, they have not proposed that the input handlers explicitly specify their state-dependent responses by means of state transition diagrams or that they retain their states during execution by coroutine activation. Cardelli

and Pike [3] achieved a similar result using communicating finite-state machines with actual concurrency. The use of coroutines in the present language, combined with the synthetic tokens described below, can also be mapped into the abstract device model introduced by Anson [1], but that, too, does not use state diagrams to describe the state and behavior of the abstract devices. Anson points out the weakness of a single-thread state diagram for describing direct-manipulation interfaces: "It cannot simulate a device . . . which retains its value between uses and which can be changed by the user at any time" [1]. The present technique attempts to remedy this problem without giving up the benefits of state diagrams for depicting device state and state-dependent behavior.

6. TOKENS IN INTERACTION OBJECTS

To complete the user-interface specification, it will be necessary to define a collection of low-level inputs and outputs, which can be invoked by the state diagrams. These will correspond to tokens. Examples for input are button clicks (both down and, where supported, up), cursor entering or exiting regions, and keyboard characters; for output, they include highlighting or dehighlighting regions, displaying or erasing graphical objects, and "rubber band" or other continuous "dragging" feedback. These tokens can be associated with transitions in the state diagrams. The internal details of these low-level input and output operations will be specified separately from the state diagrams that call them and in a different, more suitable notation, perhaps even as short procedures in a programming language or calls to an interactive graphics package. In practice, the use of inheritance discussed in Section 9 will make most tokens easy to describe irrespective of the choice of notation.

7. AN EXAMPLE SPECIFICATION

Figure 2 shows a specification of a single screen button as an individual interaction object using this approach and a simple Ada-based notation. This particular button is highlighted whenever the cursor is inside it. If the user presses the left mouse button while pointing to it, the message file **inbox** is displayed. An interaction object such as the one in Figure 2 is an object, comprising a collection of variables, methods, and other impedimenta, most of which are subject to inheritance. Specifically, the specification of an interaction object can contain the following components:

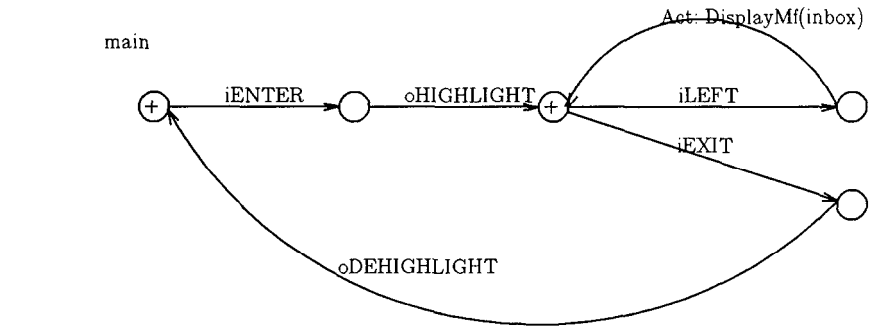
FROM: A list of other interaction objects from which this one inherits elements, with ordering rules similar to those for Flavors (i.e., components from objects listed first override those listed later).

IVARS: A list of the instance variables for this object and their initial values. These may also include other, lower level interaction objects that will be used as component parts of this one.

METHODS: Procedure definitions unique to this object. In addition, each interaction object may be required to supply (possibly by inheriting default definitions) certain standard procedures, such as **Init** or **Destroy**.

TOKENS: Definitions of each of the input and output tokens used in the syntax diagram for this interaction object. In Figure 2 they are given in English;

```
INTERACTION_OBJECT MessageFileDisplayButton is

IVARS:
position        := { 100, 200, 64, 24 };  --i.e., coordinates of screen rectangle

METHODS:
Draw ()         { DrawTextButton(position, "Display"); }

TOKENS:
iLEFT           { --click left mouse button-- }
iENTER          { --locator moves inside rectangle given by position-- }
iEXIT           { --locator moves outside rectangle given by position-- }
oHIGHLIGHT      { --invert video of rectangle given by position-- }
oDEHIGHLIGHT    { --same as oHIGHLIGHT-- }

SYNTAX:
```




```
end INTERACTION_OBJECT;
```

Fig. 2. Specification of a direct-manipulation screen button.

in practice they could be given as code in a conventional programming language or in a simple notation specialized for this purpose.

SYNTAX: The input handler for this interaction object, expressed as a conventional state transition diagram, which will be called by the executive as a coroutine. The diagram specifies the *sequence* in which the tokens and other actions defined above will occur. In the diagram, each state transition can have an input or output token, the name of another diagram to be called as a subroutine, an action to be performed, a condition to be tested, or nothing, in which case it is taken when no other transition can be. Names of input tokens begin with **i**, and names of output tokens begin with **o**. An action, such as **DisplayMf(inbox)** in Figure 2, calls a procedure that is defined in the application (semantic) code, which is separate from the user-interface specification. A condition calls a similar procedure that returns a Boolean value. Further details of this notation are found in [14] and [16]. For readability, states from which it is possible for the dialogue to be suspended are marked with "+"; they are a subset of the *user-visible* states [16]. This diagram could also have been entered or displayed in a text form, rather than the graphical form shown in the figure. (With the inclusion of arbitrary functions in the condition and action transitions, this state-diagram language has the formal power of a Turing machine. Without condition and action transitions, but with recursive calls to subdiagrams permit-

ted and nondeterministic execution, the language would be equivalent in power to a context-free grammar [12, 16].)

SUBS: Additional state diagrams, called as subroutines by the syntax diagram above.

STATES: A list of "mixin" [34] or "kernel" [27] states, which are used to define standard sets of behaviors, such as sensitivity to abort or help keys. They can be applied to states in the above diagrams, as a convenient abbreviation, so that such descriptions do not have to be repeated for each state.

8. DISCUSSION OF FIGURE 2

How does the syntax diagram given for this interaction object operate with the executive? When the cursor enters the screen area for this button, the input token **iENTER** is generated. This token **iENTER** is defined locally within object **MessageFileDisplayButton** as an event that occurs when the cursor enters the screen region given by the local variable **position.** (Other interaction objects might also use a token named **iENTER**, but each would provide a different local definition.) Since no other interaction object will have state transitions that accept **iENTER** as defined here, the diagram for this object will be called as a coroutine by the executive. This diagram will take over, accept the input, highlight the button, then wait for more input (in the second of the two states marked with a "+"). If the next input is the button press (**iLEFT**), this object performs its action. (At this point, this screen-button object should be the only interaction object that is in a state ready to accept the **iLEFT**; hence the executive will activate this object and no others.) If the next input is the cursor exiting this region (**iEXIT**), this object dehighlights itself and returns to its start state. There it waits only for another **iENTER** and ignores other inputs. (In particular an **iLEFT** or other button click will no longer be accepted by this object, but would probably be accepted by some other object.) Returning to the second state marked "+," if the next input received in that state is anything other than **iLEFT** or **iEXIT** (e.g., a keyboard key), another diagram that has a transition that can accept that input will be called by the executive. As soon as another input that this diagram can accept occurs, it will be resumed in the same state (the second one marked "+").

Why does the state diagram look so complex for an operation that seems intuitively simple to describe? The reason is that there are several possible plausible alternative behaviors for the precise handling of sequences of clicks and mouse motions in a screen button. There are other ways in which the exiting and dehighlighting could be handled. Or the screen button could be highlighted when the mouse button is depressed and perform the action when it is released (see Figure 5). The writer of the user-interface specification must be able to indicate exactly which of these possibilities is intended. It is not sufficient to describe the user interface imprecisely and leave the details up to a coder. Nor is it sufficient to supply one standard version of a screen button and prevent the designer from changing it. Given that the user-interface designer must provide this precision, state transition diagrams are an appropriate notation for so doing.

Where are the semantic and lexical levels? The bulk of Figure 2 describes the syntactic level of the dialogue. The lexical-level specification consists of the

definitions of any input or output tokens used in the dialogue; in Figure 2 they are shown as comments, and in Figure 15 as LISP code for a particular input/output package. The semantic specification consists of the definitions of any semantic actions or conditions called by the dialogue (in this example, **DisplayMf**); they are written as procedures in a conventional programming language. The interface from the syntactic level to the lexical level is via calls from the syntax diagram to the tokens; the interface from the syntactic level to the semantic level is through calls from the diagram to semantic action and condition procedures. (Asynchronous feedback from the semantic domain is given by synthetic tokens, which are discussed in Section 12; an example of such feedback appears in Figure 11.)

9. INHERITANCE

The remaining problem with this notation is that the interaction-object descriptions for a nontrivial direct-manipulation system are going to become bulky and repetitive. The solution is inheritance of the parts of the interaction objects. In the present specification language, an interaction object inherits all of the **IVARS, METHODS, TOKENS, SUBS,** and **STATES** of the objects listed in its **FROM** section and adds them to any that the object itself declares. If the object declares an **IVAR, METHOD, TOKEN, SUB,** or **STATE** of the same name, it overrides the inherited one. In turn, all of an object's own and inherited parts are inherited by its children. The entire **SYNTAX** diagram is also inherited and may be overloaded as a unit. (A notation for selectively overloading portions of it may be added in the future. It would allow all the transitions from a state to be inherited and then supplemented with other transitions from the same state, or overriden by transitions from the same state that use the same token. Note also that much of this effect can be achieved with the present implementation by dividing the diagram appropriately and using inheritance of **SUBS** and **STATES**.)

Figure 3 shows part of a library of generic interaction objects, from which components can be inherited. The library object **GenericItem** defines some tokens that are applicable to a wide range of screen items. Note that the tokens **iENTER** and **iEXIT** are defined generically, in terms of an unspecified instance variable, **position**, which is to be supplied by the inheriting object. Like a "mixin" flavor [34], it is not expected that **GenericItem** will be instantiated by itself, but will contribute its token definitions to other, more specific objects by inheritance. **Highlighter** is another mixin object that defines items related to highlighting. By specifying it as a separate object, definitions related to highlighting can be collected together and maintained more easily. It contains the tokens **oHIGHLIGHT** and **oDEHIGHLIGHT**, again defined in terms of the instance variable **position,** to be supplied by another object. It also provides an example of the use of subdiagrams, **enterhigh** and **exitdehigh**, which can now be called as subroutines from the **SYNTAX** diagram of any object that inherits from **Highlighter**. Finally, a generic screen button **GenericButton** is defined. It is similar to the one specified in Figure 2, but useful for other applications. This, too, is a mixin object, not expected to be instantiated by itself. It inherits all the components of **Highlighter** and **GenericItem**. It defines the **Draw** procedure

INTERACTION_OBJECT GenericItem **is**

TOKENS:

iENTER { *--locator moves inside rectangle given by position--* }
iEXIT { *--locator moves outside rectangle given by position--* }
--position is a local variable, to be provided by objects that inherit from GenericItem

iLEFT { *--click left mouse button--* }
iMIDDLE { *--click middle mouse button--* }
iRIGHT { *--click right mouse button--* }

iCHAR { *--keyboard character, value returned in variable viCHAR--* }

end INTERACTION_OBJECT;

INTERACTION_OBJECT Highlighter **is**

IVARS:

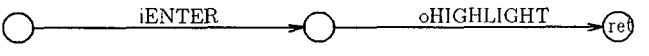
isHighlighted := false;

TOKENS:

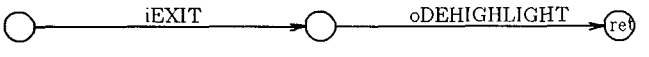
```
oHIGHLIGHT    { if not isHighlighted then
                    InvertRect(position); isHighlighted := true;
                end if; }
oDEHIGHLIGHT  { if isHighlighted then
                    InvertRect(position); isHighlighted := false;
                end if; }
```

SUBS:

enterhigh



exitdehigh



end INTERACTION_OBJECT;

INTERACTION_OBJECT GenericButton **is**

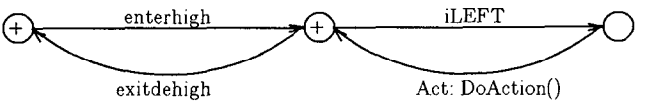
FROM: Highlighter GenericItem;

METHODS:

Draw () { DrawTextButton(position, legend); }

SYNTAX:

main



end INTERACTION_OBJECT;

Fig. 3. Some library interaction objects.

```
INTERACTION_OBJECT MessageFileDisplayButton2 is

FROM:          GenericButton;

IVARS:
position       := { 100, 200, 64, 24 };
legend         := "Display";
file           := inbox;

METHODS:
DoAction ()    { DisplayMf(file); }

end INTERACTION_OBJECT;
```

Fig. 4. Specification of the screen button of Figure 2, using inheritance from Figure 3.

generically, in terms of an instance variable **legend** instead of a constant, and it provides an inheritable syntax diagram that describes a "standard" screen button. The action in the syntax diagram calls a procedure named **DoAction**, which each inheriting object can define in its own way. The **SYNTAX** diagram also uses some of the tokens and subdiagrams defined in and inherited from **Highlighter** and **GenericItem**.

Given these primitives, the particular button defined in Figure 2 can now be written more compactly by inheriting the aspects that are common to other items and screen buttons and defining only those specific to this particular button. The specification in Figure 4 defines the same object as that of Figure 2, taking advantage of the library. It inherits the components of **GenericButton** and, through it, **Highlighter** and **GenericItem**. It defines only the instance variables **position** (which is used by the tokens in **GenericItem** and **Highlighter**) and **legend** (used by **Draw** in **GenericButton**) and the procedure **DoAction** (called by the syntax diagram in **GenericButton**). Everything else is inherited from the generics, including the syntax diagram itself from **GenericButton**. If a user-interface designer did want this particular screen button to be different from the standard ones, the designer would simply overload those aspects of the generic objects that he or she wanted to change.

In practice, a more convenient way to assemble a collection of similar screen buttons would be to define **MessageFileDisplayButton2** as a type and then instantiate it for each individual button and message file, parameterized by the instance variables of **MessageFileDisplayButton2**. For example, in Ada-like notation,

```
x := new INTERACTION_OBJECT MessageFileDisplayButton2
  (position=>{100, 250, 64, 24}, file=>newbox);
y := new INTERACTION_OBJECT MessageFileDisplayButton2
  (position=>{100, 300, 64, 24}, file=>anotherfile);
etc.
```

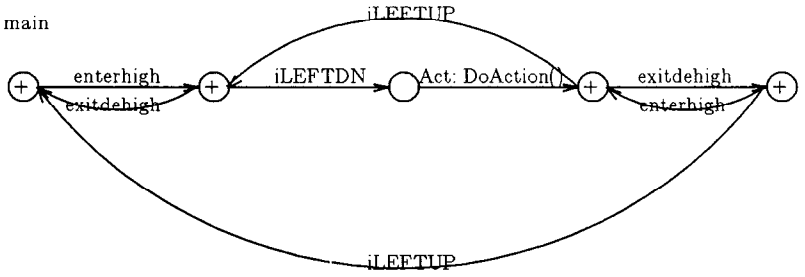
Given a library of useful generic interaction objects, the job of designing and specifying a new direct-manipulation interface becomes much less arduous. The principal aim of the research at the stage reported here, however, is to devise an appropriate specification language, based on the clearest and simplest adequate model, rather than to develop a complete library or tool kit. It has been seen that

INTERACTION_OBJECT GenericButton2 **is**

FROM: Highlighter GenericItem;

METHODS:
Draw () { DrawTextButton(position, legend); }

SYNTAX:



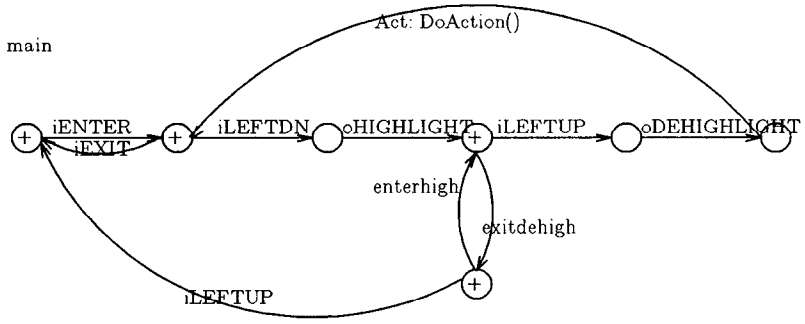
end INTERACTION_OBJECT;

INTERACTION_OBJECT GenericButton3 **is**

FROM: Highlighter GenericItem;

METHODS:
Draw () { DrawTextButton(position, legend); }

SYNTAX:



end INTERACTION_OBJECT;

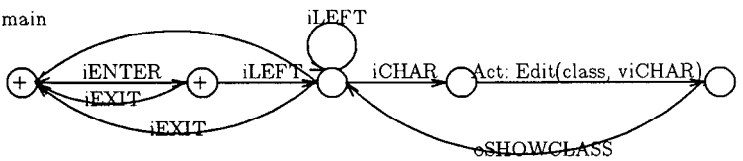
Fig. 5. Some alternative types of screen buttons.

this language can be used either to describe individual user interfaces from scratch, or to describe and build up a library of interaction techniques and then specify user interfaces built from them.

10. FURTHER EXAMPLES

Figure 5 shows some alternative types of screen buttons. Unlike the one in Figure 3, these two assume an executive that reports both up and down transitions of the mouse buttons individually and a version of **GenericItem** that defines

```
INTERACTION_OBJECT TypeinField is
FROM:          GenericItem;
IVARS:
position;
class          := "UNCLASSIFIED";
METHODS:
Draw ()        { DrawText(position, class); }
TOKENS:
oSHOWCLASS     { DrawText(position, class); }
SYNTAX:
```




```
end INTERACTION_OBJECT;
```

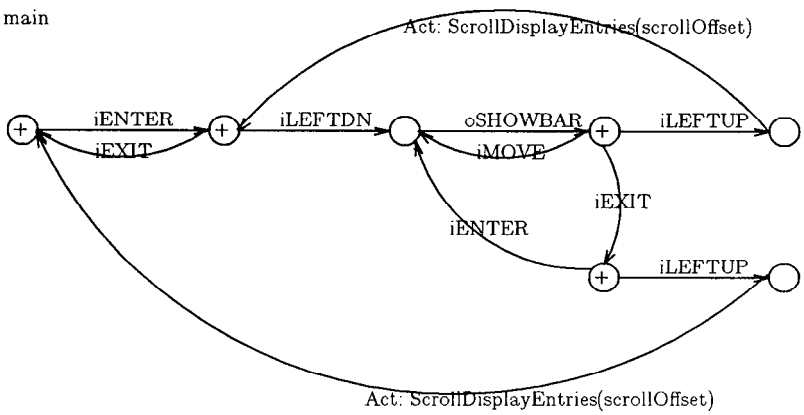
Fig. 6. Specification of a type-in field.

separate tokens for them (i.e., **iLEFTUP**, **iLEFTDN**, **iMIDDLEUP**, etc.). **GenericButton2** behaves just like **GenericButton** in Figure 3, but its syntax diagram has transitions that accept both the **iLEFTDN** and **iLEFTUP** tokens to work with such an executive. The specification clearly shows that the action occurs after the button goes down, rather than up. It also handles the unusual case in which the button is depressed, action occurs, the cursor is moved out of the region, and then other input events occur while the button is still down. In that case, the other inputs would cause **GenericButton2** to be suspended (from the state shown at the extreme right of the diagram) and another interaction object, which had a transition for the new input, to be resumed. Some time later, when the left button is finally released, **GenericButton2** would be resumed and make the transition (shown as the larger lower loop) back to its initial state. This is appropriate because that "dangling" **iLEFTUP** event logically belongs to this dialogue, despite any delay or intervening interactions. (A more convenient and general mechanism for translating sequences of up and down events into simple click events will be given in Figure 10.)

GenericButton3 shows a screen button with a different behavior. This one does not highlight itself until the mouse button is depressed, and then it performs its action when the button is released—provided the cursor is still within the screen button.

Figure 6 shows a type-in field, like the ones for security classification ("Class") that will be seen in Figures 16 and 17. As specified, it allows a user to click on the field and then type characters into it. Each time the user moves to any other dialogue, he or she must again click on this field before resuming typing in it. This is specified in the syntax diagram with the aid of the unlabeled transition shown leading toward the leftmost state in the diagram. This transition will be

```
INTERACTION_OBJECT ScrollBar is
FROM:          GenericItem;
IVARS:
position;
legend         := "Scroll:",
scrollOffset   := 0;
METHODS:
Draw ()        { DrawBar(position, legend, scrollOffset); }
TOKENS:
iMOVE          { --Any mouse motion within boundaries of position,
                 --return scaled X coordinate of mouse in scrollOffset-- }
iLEFTDN        { --Overloads standard definition of iLEFTDN with one that accepts
                 --same click then sets scrollOffset:= scaled X coord of mouse -- }
oSHOWBAR       { --Fill or erase bar up to location corresponding to scrollOffset-- }
SYNTAX:
main
```




```
end INTERACTION_OBJECT;
```

Fig. 7. Specification of a scroll bar.

taken whenever no other transition can be taken from the state that normally accepts and processes the **iCHAR**s. After that transition to the leftmost state is taken, if the diagram still cannot handle the input, it will be suspended from its leftmost state. The result is that if, during type-in of the text, an input destined for another interaction object is received, the **TypeinField** object makes the transition to its leftmost state before it is suspended. This guarantees that a new **iENTER** and **iLEFT** must be received before the user can return to typing the text. The function **Edit(line,char)** shown on one of the transitions implements a simple input line editor. The variable **viCHAR** contains the data returned by token **iCHAR**, following a standard naming convention.

Figure 7 shows a simple interactive graphical image—a scroll bar, like those in Figures 16 and 17. To use it the user points to it and depresses the left mouse button; then, as the mouse is moved, the bar drags on the screen to follow it.

When the mouse button is released, the display is scrolled in proportion to the new position of the bar. As with **GenericButton2** above, this syntax diagram explicitly handles the unusual case where the user moves the mouse to the scroll bar, depresses the button, then, while holding it down, exits the scroll bar, possibly performs other interactions, and then later reenters the scroll bar with the button still held down. This object will resume dragging the bar when the cursor reenters it. Other behaviors could have also been specified and distinguished in this notation. For example, exiting the scroll bar could finish scrolling the display and then the subsequent **iLEFTUP** could be consumed and ignored.

It becomes clear that the state transition diagram notation is being used to describe interaction down to a rather fine-grained level. For a direct-manipulation interface, such details are important; and since they may vary significantly across the objects, this is appropriate. For example, the loop consisting of **iMOVE** and **oSHOWBAR** could have been defined inside a single token that accepts the mouse motion and echoes it by drawing the bar, but they were shown more explicitly instead. A slightly different scroll bar might scroll continuously as the user moves the cursor; releasing the button would then just stop the scrolling. Its syntax diagram would differ precisely at the point of the **iMOVE–oSHOWBAR** loop in Figure 7.

11. COMPONENT OBJECTS

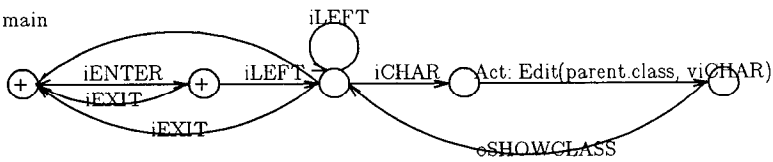
It may also be helpful to describe an interaction object as a combination of several other components, each of which is a separate interaction object. In contrast to inheritance, where an object inherits the properties of another type of object, here one object simply contains one or more smaller objects. Those component objects are full-fledged interaction objects, possibly with their own **SYNTAX** coroutine diagrams. They are automatically instantiated whenever the enclosing object is created (and at no other time) and similarly automatically drawn and destroyed. In other respects, the component objects are simply instance variables of the larger object.

For example, Figure 8 shows a combination object **TypeinObject** that combines a type-in field and its associated screen button, **TypeinField** and **TypeinButton**, respectively, like the security classification items that will be seen in Figure 16. By combining them, **TypeinField** and **TypeinButton** can more easily refer to their common variable, **class**. They are also instantiated, redrawn, and destroyed simultaneously, and their relative locations are specified more conveniently. Since they belong together logically, it is helpful to have a notation that allows them to be combined. The specification of the **TypeinButton** component also takes advantage of the **GenericButton** specified in Figure 3.

12. SYNTHETIC TOKENS

Such component objects may also be used to provide low-level input or output operations, such as cursor tracking, and to summarize sequences of such inputs or outputs into larger units to be processed by the higher level objects [1]. This design allows modularity by describing low-level tracking operations separately

```
INTERACTION_OBJECT TypeinObject is

IVARS:
position;
class           := "UNCLASSIFIED";
button          := new INTERACTION_OBJECT TypeinButton(
                     position=> position);
text            := new INTERACTION_OBJECT TypeinField(
                     position=> position + TextSize("Class:");

METHODS:
Draw ()         { DrawBorder(position); }

end INTERACTION_OBJECT;

INTERACTION_OBJECT TypeinButton is

FROM:           GenericButton;

IVARS:
position;
legend          := "Class:";

METHODS:
DoAction ()     { Reclassify(parent.class); }
                --Note: the pseudo-variable parent refers to the object of which
                --TypeinButton is itself an instance variable (i.e., TypeinObject)

end INTERACTION_OBJECT;

INTERACTION_OBJECT TypeinField is

FROM:           GenericItem;

IVARS:
position;

METHODS:
Draw ()         { DrawText(position, parent.class); }

TOKENS:
oSHOWCLASS      { DrawText(position, parent.class); }

SYNTAX:
```

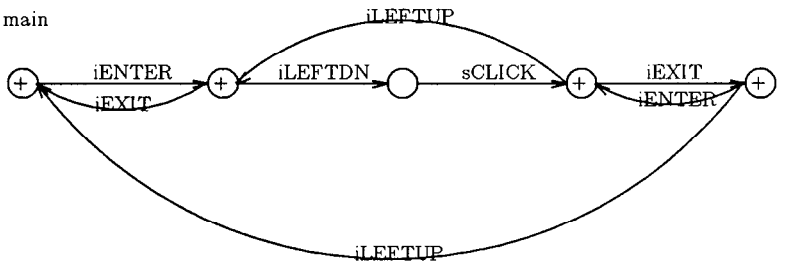



```
end INTERACTION_OBJECT;
```

Fig. 8. Specification of an entire type-in object, using component objects.

from their higher level syntactic meaning. To provide this feature, the syntax diagrams of interaction objects are extended to be able to send and receive synthetic "tokens" to or from each other. Thus a lower level object can receive and respond to a collection of user actions and then summarize them by trans-

```
INTERACTION_OBJECT Clicker is
FROM:       GenericItem;
IVARS:
position;
METHODS:
Draw ()     { null; } --Note this interaction object has no screen representation
TOKENS:
sCLICK      { SendTo(parent); }
SYNTAX:
```




```
end INTERACTION_OBJECT;
```

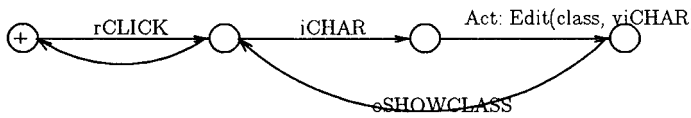
Fig. 9. Specification of an object that converts individual up and down operations of a button into single clicks, using synthetic tokens.

mitting a single synthetic token to its higher level object. The higher level object syntax diagram can have transitions to receive only the synthetic token, rather than the lower level input actions.

Such a synthetic token can be viewed as a rendezvous between the syntax diagrams of the lower and higher level objects. The tokens are given names of the forms **sTOKENNAME** and **rTOKENNAME** to send or receive the named event, respectively. When a diagram traverses a transition that produces **sTOKENNAME**, it is suspended in favor of any other diagram that has a transition that accepts the corresponding **rTOKENNAME**. The diagrams can also set and fetch variables of the form **vTOKENNAME** to communicate information at the rendezvous. Such tokens will generally be sent and received among (i.e., will be local to each set of) an interaction object and any of its component objects. That is, their scope is determined by the hierarchy of instantiations of interaction objects, rather than the inheritance hierarchy. Any object that sends a synthetic token declares the scope of that token; that is, it indicates which other interaction objects will receive the token by means of the **SendTo** statement in the token definition of the sending object. The recipient is normally an interaction-object instance variable of the sending object or else the containing ("parent") object of the sender.

Figure 9 shows an interaction object that provides support for receiving up and down button transitions inside a region and translating them into a single click event **sCLICK**. It receives the low-level inputs **iENTER**, **iEXIT**, **iLEFTDN**, and **iLEFTUP** and processes them into the simple higher level token **sCLICK**,

```
INTERACTION_OBJECT TypeinField2 is

FROM:          GenericItem;

IVARS:
position;
class          := "UNCLASSIFIED";
c              := new INTERACTION_OBJECT Clicker(position=>position);

METHODS:
Draw ()        { DrawText(position, class); }

TOKENS:
oSHOWCLASS     { DrawText(position, class); }

SYNTAX:

main
```




```
end INTERACTION_OBJECT;
```

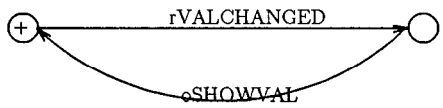
Fig. 10. An alternate way to specify the type-in field, using **Clicker** and synthetic tokens.

which it sends to the larger object within which it has been created as an instance variable. In this respect it is used like a process in Squeak [3]. It is intended for applications where entering and exiting the region are of no interest except insofar as they determine whether a down click is inside or outside the region, and up clicks are of no syntactic importance. An example is a simple screen button that just responds to a click of the mouse button in its region.

Figure 10 shows a revised version of the **TypeinField** shown in Figure 6, here using the new **Clicker**. To use **Clicker** the enclosing object (**TypeinField**) instantiates it as one of its instance variables and provides it the correct value for its **position**. Then, **TypeinField** need not process **iENTER**, **iLEFTDN**, and similar events, but simply waits to receive an **rCLICK** from the **Clicker** object. **Clicker** is used as a component (**IVAR**) of **TypeinField**, rather than as a mixin (**FROM**). It is thereby created as a separate interaction object, whose syntax diagram will run concurrently (as a coroutine) with all the other diagrams.

The synthetic tokens are intended to be used principally to build up composite input events, in order to build higher level objects with higher level tokens out of lower ones. By broadening the scope of such a token, it could also be used for more general interprocess communication, particularly for asynchronous events from the semantic or application code (which can run concurrently with the user-interface executive, rather than as a coroutine). An application could send such a token as a means for indicating an event to which the user interface must respond. Figure 11 shows a simple display panel meter using this approach. It responds to some external event (as recognized by the application code) and continuously displays a value; it has no user input. The application code sends

```
INTERACTION_OBJECT Meter is

IVARS:
position;
currentValue        := startingValue;

METHODS:
Draw ()             { --Draw a meter dial with pointer at currentValue-- }

TOKENS:
rVALCHANGED  { currentValue:= vVALCHANGED; }
oSHOWVAL     { --erase meter pointer and redraw at location for currentValue-- }

SYNTAX:

main
```




```
end INTERACTION_OBJECT;
```

Fig. 11. Specification of a display meter that monitors a value.

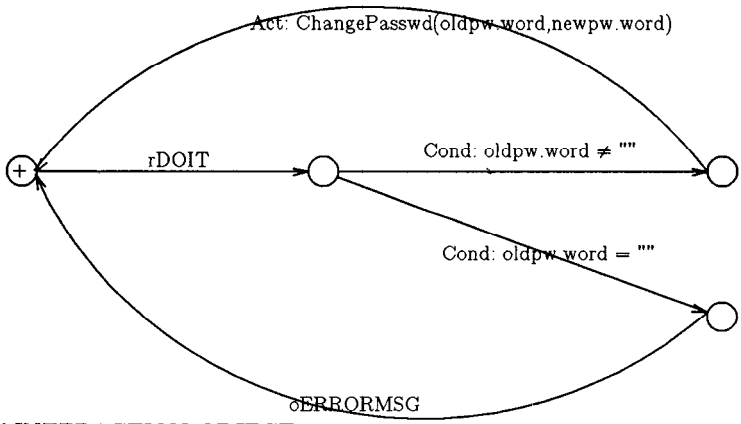
the synthetic token **sVALCHANGED** whenever the monitored value changes. It can also send data via the associated variable **vVALCHANGED**. The application (semantic) code need not know about the way in which **sVALCHANGED** will be handled by the dialogue (syntactic) specification, or even whether it will cause a change in the display. It merely announces the occurrence of an event that is meaningful to the dialogue, and the dialogue specification then indicates how the user interface will respond to it. (If a syntax diagram responds to this event in the same manner from several different states, the **STATES** notation can be used to describe the situation more compactly.)

A further extension would generalize the interface between the syntax diagram and the application (semantic) code from procedure calling to message passing. This would permit more complex protocols betwen the semantic and syntactic parts of the system than the current example. It would also make it easier to specify user-generated interrupts or aborts that occur during semantic processing. An individual state transition in the syntax diagram could either send a message (typically a request to perform a function) or await a message (most often indicating completion of the function). The procedure call protocol used in the examples would be a straightforward degenerate case of this scheme (every "send request" transition would be followed immediately by an "await completion message" transition).

13. SPECIFYING A FORM

A fill-in form or property sheet would be specified using the component object and synthetic token mechanisms introduced. Figure 12 shows the specification of a simple form, such as that for changing a user's password seen in Figure 17. The overall form is specified by a composite object **FillinForm**. It consists of

```
INTERACTION_OBJECT FillinForm is

IVARS:
position;
but             := new INTERACTION_OBJECT FillinButton(
                     position=>position);
oldpw           := new INTERACTION_OBJECT FillinField(
                     position=>position+Height(FillinButton),
                     legend=>"Old password",
                     word=>"");
newpw           := new INTERACTION_OBJECT FillinField(
                     position=>position+Height(FillinButton)+Height(FillinField),
                     legend=>"New password");

METHODS:
Draw ()         { DrawBorder(position); }

TOKENS:
oERRORMSG       { ShowErrorMsg("Please enter old password first"); }

SYNTAX:

main
```




```
end INTERACTION_OBJECT;
```

Fig. 12. Specification of a simple form, using synthetic tokens.

three display regions: a button that causes the password to be changed, a type-in field for entering the old password, and another type-in field for the new password. They are all specified as components of **FillinForm**: a **FillinButton** and two **FillinFields**, respectively. **FillinForm** and its components specify different levels of the syntax. The **FillinButton** and **FillinFields** specify the syntax for processing mouse clicks and characters within the individual fields. They pass a synthetic token to the **FillinForm**. **FillinForm** specifies the overall syntax of the form using the synthetic token and provides a scope for the shared variables. It describes the syntactic relationship of the three individual objects, but not the internal operation of each of them. It also enforces a simple syntactic constraint that requires that a value for the old password be entered before the user can attempt to change the password. In a more complex form or property sheet, the

INTERACTION_OBJECT FillinButton **is**

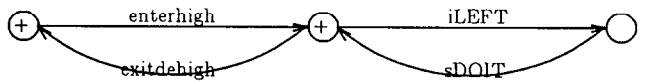
FROM: GenericButton;

IVARS:
position;
legend := "Change Password";

TOKENS:
sDOIT { SendTo(parent); }

--This syntax section overloads the standard one inherited from GenericButton
SYNTAX:

main



end INTERACTION_OBJECT;

INTERACTION_OBJECT FillinField **is**

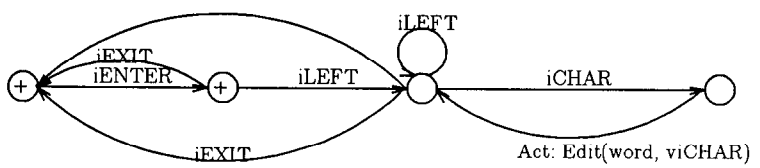
FROM: GenericItem;

IVARS:
position;
legend;
word;

METHODS:
Draw () { null; } *--does not display word*

SYNTAX:

main



end INTERACTION_OBJECT;

Fig. 12 (continued)

interrelationships of the components would be specified here. The syntax diagrams for the enclosing object would specify and restrict the permissible sequences in which it will accept the synthetic tokens from its components. For example, "car radio" buttons or option selections that appear and disappear based on actions in other fields would be specified in this way. (A better direct-manipulation interface for the password form shown here would hide or make inactive the **FillinButton** whenever its operation was disallowed rather than give an error message. It requires a slightly more complex diagram than the

simple example given, but it would still maintain the separation of specification of aspects of the dialogue syntax pertaining to the individual components from specification of the syntactic relationships between those components.)

The present approach also makes it easy to describe a general class of window-management systems. Most existing window managers permit only one window to receive user input at a time. While some systems allow several windows to receive and display outputs simultaneously, one window (the "listener") still receives all inputs. There is no fundamental reason for this asymmetry. It is possible to envision a window system in which text input from the keyboard is accepted in a typescript command window and simultaneous graphical input from the mouse is accepted in another window. The specification language can describe such cases conveniently and naturally. Each window is an interaction object with a state diagram that accepts a mutually exclusive set of inputs. For the example given, one diagram would accept mouse events, and the other, keyboard. It would be equally easy to describe a more unusual user interface, in which, for example, one window accepts the letter *a* and another accepts *b*. The traditional single-listener paradigm is simply a special case, in which a "change-listener" command puts one window into a state where it accepts all the inputs until the next change-listener command, and all the other windows into states where they accept only the signal resulting from the change-listener command, similar to the way in which car radio buttons are specified to enable and disable one another.

14. GRAPHICAL OBJECTS

Figure 13 gives a portion of the specification of a painting system similar to MacPaint, and Figure 14 shows it in operation. The basic structure of each command in the specification is like that of **ScrollBar** shown in Figure 7. For clarity, Figure 13 shows the operation of only two of the painting commands—drawing with a "brush" and dragging a "rubber-band" rectangle. The dialogues for all but two of the other commands (*text* and *polyline*) are identical to one of these two cases. Figure 13 gives the interaction object for the main graphical window, but it does not show the three other windows containing commands or option selections. Each of those other windows is implemented as a separate interaction object containing an array of buttons. The buttons are all specified similarly to **GenericButton3**, shown in Figure 5, but they communicate with each other through synthetic tokens so that only one button in each of the three windows can be selected at a time. Choosing a button in either the linewidth or pattern-selection window changes the value of a global variable used in subsequent drawing operations. Choosing a button in the command selection window sends the synthetic token **NEWCMD** to the paint window (i.e., the interaction object specified in Figure 13), with the value of **vNEWCMD** set to the name of the chosen command.

Note that a graphics system could have been represented in several other ways, depending on the nature of the dialogue it conducts with the user. In this and most typical systems, the main interaction in the graphics area is best described as a single-thread dialogue with a main command loop. It is thus represented as a single interaction object with one state diagram. Parameter setting windows,

INTERACTION_OBJECT PaintWindow **is**

FROM: GenericItem;

IVARS:

position := {200, 20, 300, 240};
prevMove;

METHODS:

Draw () { EraseRect(position); }

TOKENS:

iLEFTDN { *--Overloads standard definition of iLEFTDN with one that accepts --same click then sets viLEFTDN:= X,Y coordinates of mouse--* }

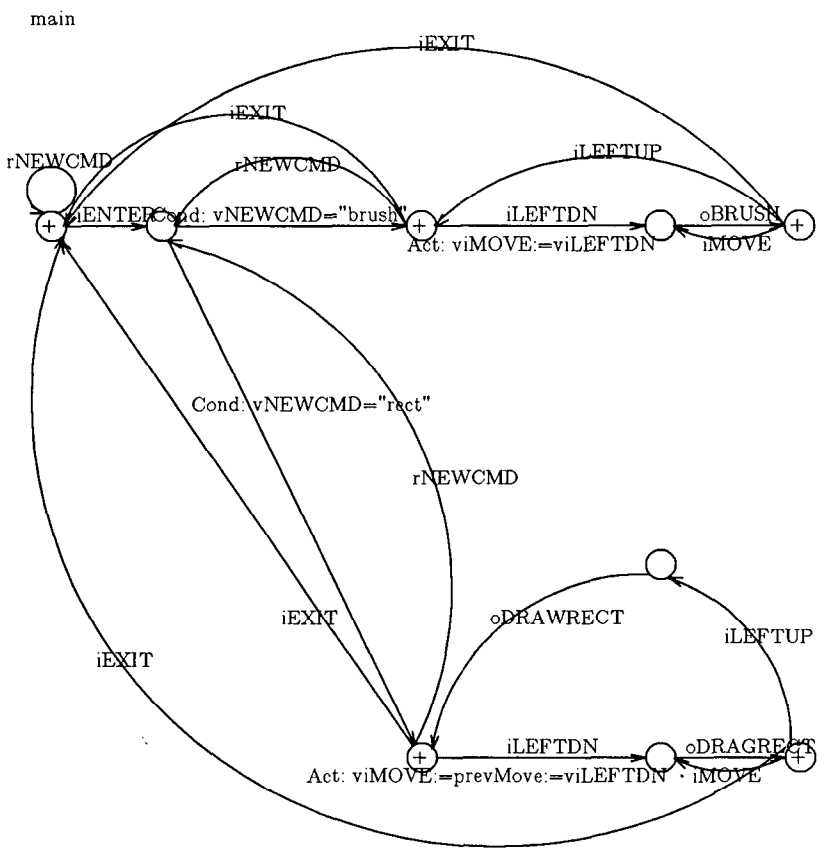
iMOVE { *--Accepts any mouse motion within boundaries of position --and sets viMOVE to the X,Y coordinates of the mouse--* }

oBRUSH { *--Fill the area defined by viMOVE and currentBrushShape --(a global variable set by a pull-down menu) with currentPattern --(global variable set in the pattern selection window)--* }

oDRAGRECT { *--Note: currentLineWidth is a global variable --set in line width selection window*
InvertOutline(viLEFTDN, prevMove, currentLineWidth);
InvertOutline(viLEFTDN, viMOVE, currentLineWidth);
prevMove:= viMOVE; }

oDRAWRECT { DrawOutline(viLEFTDN, viMOVE, currentLineWidth); }

SYNTAX:



end INTERACTION_OBJECT;

Fig. 13. Portion of specification of a painting system.

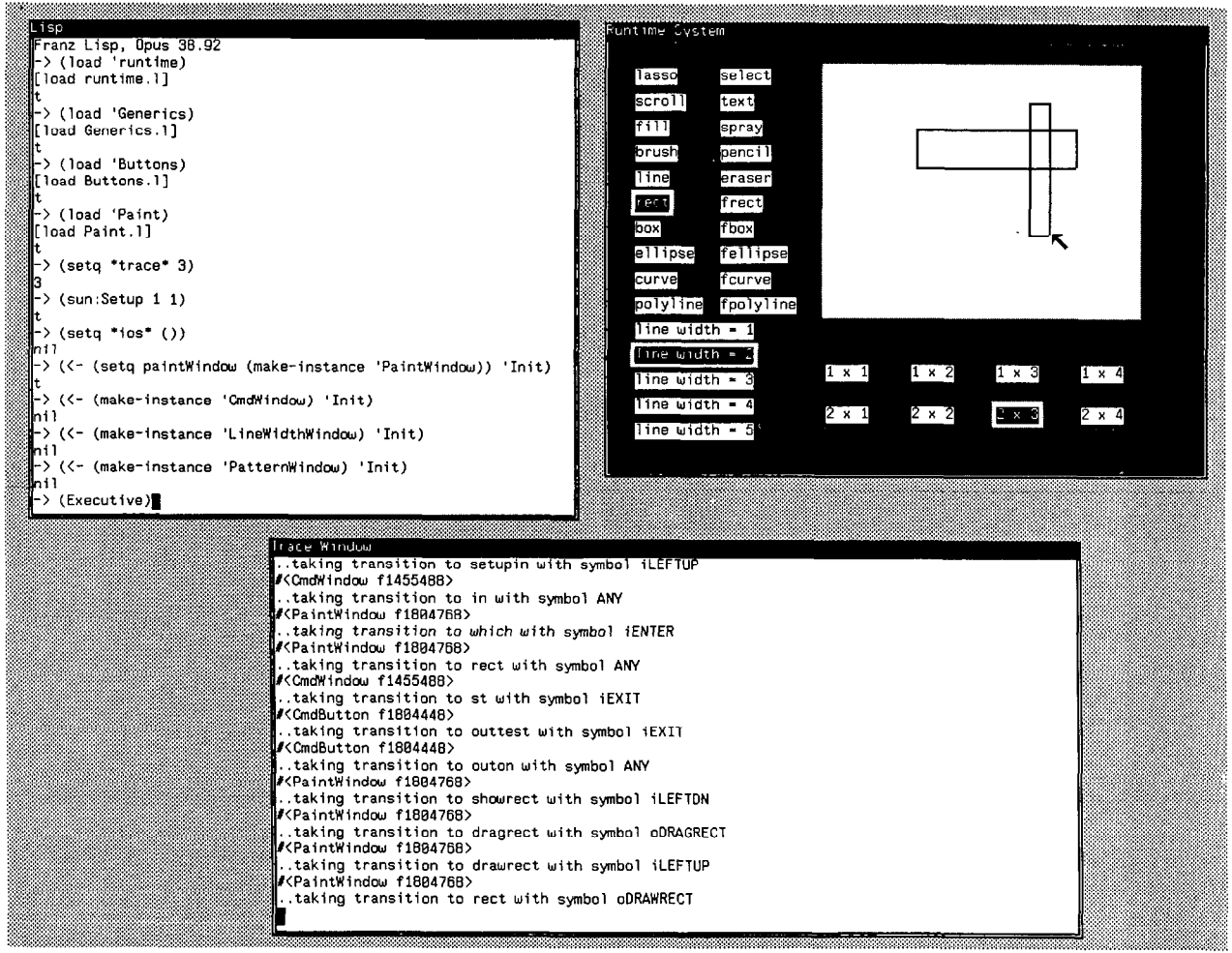


Fig. 14. The painting system specified in Figure 13 in operation.

pull-down menus, and the like each constitute an independent dialogue; they are appropriately described as separate interaction objects operating as coroutines.

The operation of the various individual painting commands in the main window could also have been given as separate interaction objects in this language. However, they do not really appear to the user as separate concurrent dialogues with saved states. They are more clearly represented as modes or states within a single main dialogue. The graphical images drawn by the user could also have been represented in the language as separate interaction objects, instantiated and destroyed as needed. Again, however, to the user they are essentially passive objects, which respond to the main command loop; they have no saved state and no specific individual syntaxes. The specification language allows the description of more exotic systems, in which the user can create and edit different kinds of active objects on the screen, which can then respond to commands and have different syntaxes. They would be described as separate interaction objects and instantiated dynamically for each new object the user creates.

15. IMPLEMENTATION

To test the specification language, a prototype user-interface management system has been built. It follows the structure described here and provides an executive that performs coroutine calls on the individual interaction-object diagrams. The system accepts the specifications of a set of interaction objects and implements the resulting user interface. It runs under UNIX[1] on a Sun Microsystems 68010-based workstation; it is written in Franz Lisp with the University of Maryland Flavors package and some C code to interface to the Sun window system. It has been used to build and test small systems, including the examples given here and many similar ones.

The implementation uses object-oriented programming extensively. Each interaction object has an instance variable containing its syntax diagram and another containing its current state. A coroutine call is implemented as a message from the executive to the interaction object, which causes the latter to assume control and execute until it reaches an input it cannot accept. It then returns control to the executive, which sends the same coroutine call message to each of the other interaction objects, effectively asking each of them to accept the current input and proceed. As noted in the language definition, if there is more than one object that can accept the input, the choice among them is considered nondeterministic; if there is none, the executive displays a diagnostic message, discards the unwanted token, and proceeds. Tokens are implemented as messages within interaction objects, defined locally to each object or else inherited. Since a single mouse motion may cause several input tokens (entering and exiting several regions), each interaction object defines a message that accepts a new mouse position and decides whether any input tokens defined within that object should be generated. Component objects are dynamically instantiated and initialized by code automatically appended in the **Init** procedure of the parent object (and destroyed similarly by the **Destroy** procedure). The current design allows a clean modularization (tokens, state diagram traversals, and

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

```
INTERACTION_OBJECT MessageFileDisplayButton is

IVARS:
position        := '(100 200 64 24);  --i.e., coordinates of screen rectangle

METHODS:
Draw ()         (DrawTextButton position "Display");

TOKENS:
iLEFT           --Click left mouse button
                (cond ((equal (car (NextLexeme)) 'iLEFTDN)
                       (ReadLexeme))
                      (t nil))

iENTER          --Locator moves inside rectangle given by position
                (cond ((and (equal (car (NextLexeme)) 'iENTER)
                            (equal (cadr (NextLexeme)) self))
                       (ReadLexeme))
                      (t nil))

iEXIT           --Locator moves outside rectangle given by position
                (cond ((and (equal (car (NextLexeme)) 'iEXIT)
                            (equal (cadr (NextLexeme)) self))
                       (ReadLexeme))
                      (t nil))

oHIGHLIGHT      --Invert video of rectangle given by position
                (InvertRect position)

oDEHIGHLIGHT    --Same as oHIGHLIGHT
                (InvertRect position)

SYNTAX:

+st:            iENTER →highlight

highlight:      oHIGHLIGHT →click

+click:         iLEFT →doit
&               iEXIT →dehighlight

doit:           →click act: (DisplayMf inbox);

dehighlight:    oDEHIGHLIGHT →st

end INTERACTION_OBJECT;
```

Fig. 15. Specification of the screen button of Figure 2, in the form used for input to the prototype user-interface management system.

interpretation of mouse motions are all local to the interaction objects) and a simple (10 lines of code) executive. It leads to some inefficiency whenever one object yields control to another (all objects are essentially polled at that point) and whenever a mouse is moved (again, all objects are polled). In a more efficient implementation, the executive might cache a table of which objects could currently accept which tokens, and a more efficient input handler would undoubtedly collect and centralize the processing of mouse motions, both at some cost in modular structure.

The specification language accepted by the system is the same as that shown in the figures in this paper, except that the bodies of the procedures and tokens are given in LISP rather than Ada or pseudocode comments and the state diagrams are entered in a text form [14, 16]. Figure 15 shows the **Message FileDisplayButton** of Figure 2 in the form accepted by this implementation, including the LISP code for the tokens themselves and the text form of the state diagram.

16. EXAMPLE OF A DIRECT-MANIPULATION SYSTEM

Figures 16 and 17 show the display of a more complete direct-manipulation military message system, one of a family of prototype message systems being built at NRL [10]. Such a message system is much like a conventional electronic-mail system, except that each message (actually, each field of each message), each file, and each user terminal has a security classification. On the screen a message is represented by an image similar to a traditional paper military message, and a file of messages is represented by a display of a list of the summaries (called *citations*) of the messages in the file. Some elements of each message citation in such a display can be changed directly by typing over them; they are indicated by borders around their labels. Other elements are fixed because the application requires it (e.g., the user cannot change the date of a message that has already been sent). In addition, each citation contains some screen buttons, which cause the indicated actions to occur. All the commands that the user could apply to a given message are shown on its citation as buttons. Specifications for these and most of the other items seen in Figures 16 and 17 have been given above; the type-in fields were specified in Figure 8, the screen buttons in Figure 4, and the scroll bar in Figure 7, and the "Change Password" form seen in Figure 17 was specified in Figure 12.

17. CONCLUSIONS

Direct-manipulation user interfaces involve a set of visual representations on a screen and a standard repertoire of manipulations that can be performed on any of them. This means that the user has no command language to remember beyond the standard set of manipulations, few changes of mode (i.e., most commands can be invoked at any time), and a continuous reminder on the display of the available objects and their states. Direct manipulation represents a powerful paradigm for designing user interfaces, but such interfaces have been difficult to specify and program conveniently.

Direct-manipulation interfaces were found to have a coroutine-like structure and, despite their surface appearance, a peculiar, highly moded dialogue. The specification technique introduced here exploits these observations. Each locus of dialogue is clearly and appropriately described as a separate object with a single-thread state diagram, which can be suspended and resumed, but always retains state. The overall direct-manipulation user interface is defined implicitly by the coroutine-based behavior of a standard executive, rather than inappropriately as a large, highly regular state transition diagram. Given a library of generic interaction objects and an inheritance mechanism, the collection of interaction-object specifications necessary to describe a nontrivial system need not become cumbersome or repetitive.

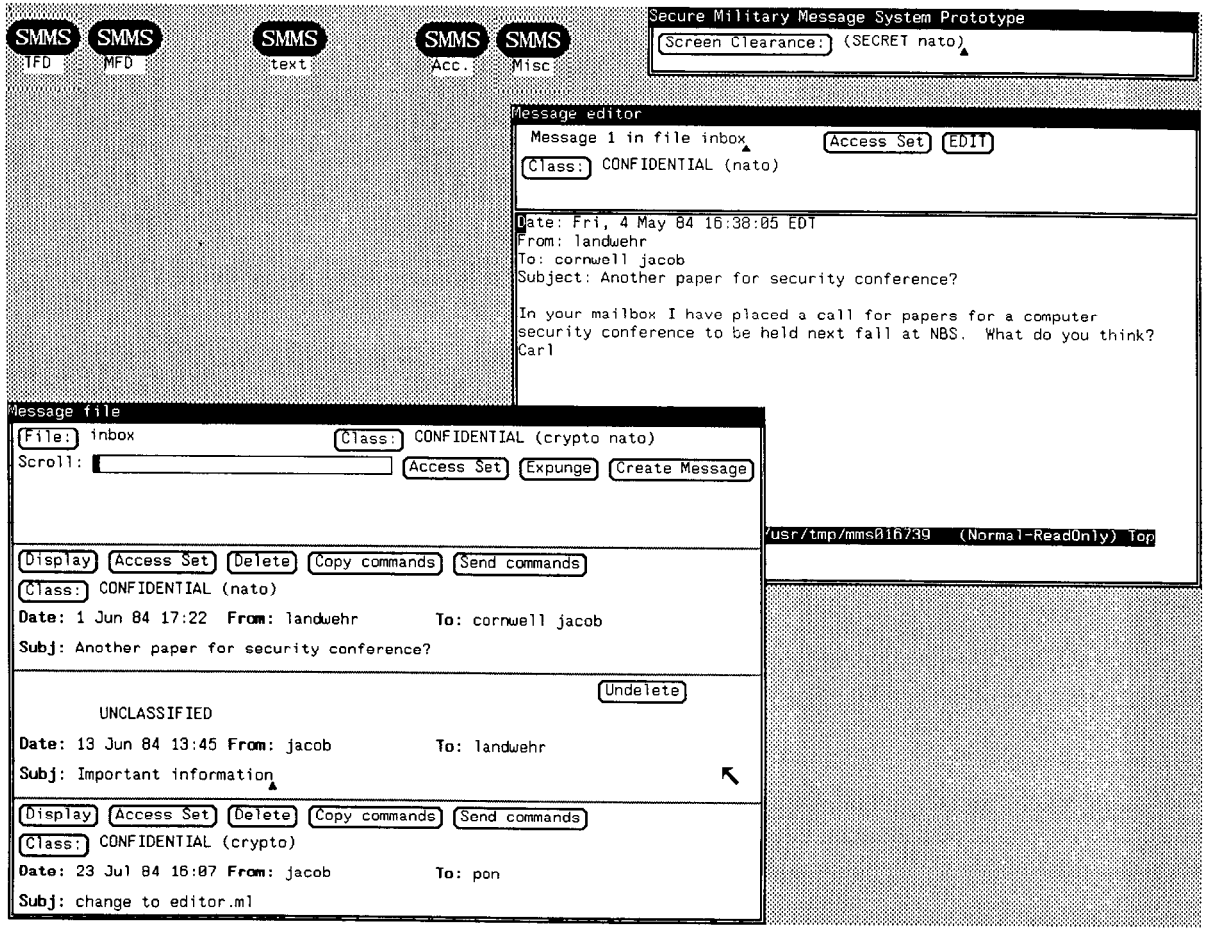


Fig. 16. Direct-manipulation military message-system prototype, showing message window (top) and message file window (bottom). The security classifications shown are simulated for demonstration purposes.

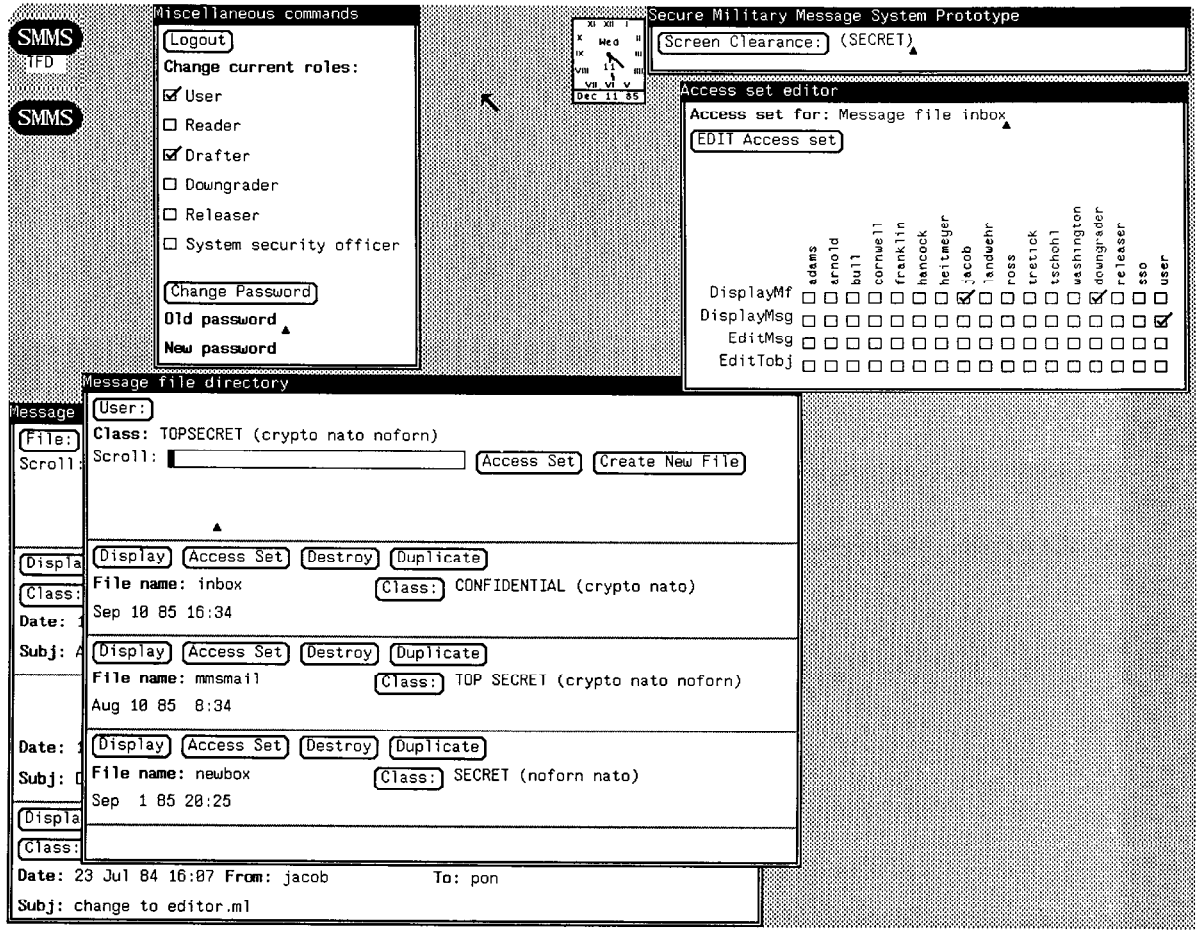


Fig. 17. Direct-manipulation military message-system prototype, showing user roles, access set editor, and message file directory windows.

ACKNOWLEDGMENTS

I want to thank Ben Shneiderman for introducing the idea of direct manipulation and helping me understand it, and Don Norman for insights that have helped to clarify the idea. Discussions with my colleagues on the WIS Command Language Task Force—Phil Hayes, Ken Holmes, Joe Hrycyszyn, Tom Kaczmarek, Jon Meads, and Brad Myers—helped me define the specification language. I thank Jim Foley, Ben Shneiderman, and several anonymous *ACM TOG* referees for their helpful comments on drafts of this paper. Finally, I want to thank Carl Landwehr for facilitating and encouraging this research.

REFERENCES

1. ANSON, E. The device model of interaction. *Comput. Graph. 16*, 3 (July 1982), 107–114.
2. BUXTON, W., LAMB, M. R., SHERMAN, D., AND SMITH, K. C. Towards a comprehensive user interface management system. *Comput. Graph. 17*, 3 (July 1983), 35–42.
3. CARDELLI, L., AND PIKE, R. Squeak: A language for communicating with mice. *Comput. Graph. 19*, 3 (July 1985), 199–204.
4. FELDMAN, M. B., AND ROGERS, G. T. Toward the design and development of style-independent interactive systems. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference* (Gaithersburg, Md., Mar. 15–17). ACM, New York, 1982, pp. 111–116.
5. FOLEY, J. D., AND VAN DAM, A. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1982.
6. FOLEY, J. D., AND WALLACE, V. L. The art of graphic man–machine conversation. *Proc. IEEE 62*, 4 (Apr. 1974), 462–471.
7. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
8. GREEN, M. The University of Alberta user interface management system. *Comput. Graph. 19*, 3 (July 1985), 205–213.
9. HAYES, P. J. Executable interface definitions using form-based interface abstractions. In *Advances in Human–Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, N.J., 1985, pp. 161–189.
10. HEITMEYER, C. L., LANDWEHR, C. E., AND CORNWELL, M. R. The use of quick prototypes in the military message systems project. *Softw. Eng. Notes 7*, 5 (Dec. 1982), 85–87.
11. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
12. HUERAS, J. F. A formalization of syntax diagrams as k-deterministic language recognizers. M.S. thesis, Computer Science Dept., Univ. of California, Irvine, 1978.
13. HUTCHINS, E. L., HOLLAN, J. D., AND NORMAN, D. A. Direct manipulation interfaces. In *User Centered System Design: New Perspectives in Human–Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Erlbaum, Hillsdale, N.J., 1986.
14. JACOB, R. J. K. Executable specifications for a human–computer interface. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference* (Boston, Mass., Dec. 12–15). ACM, New York, 1983, pp. 28–34.
15. JACOB, R. J. K. Using formal specifications in the design of a human–computer interface. *Commun. ACM 26*, 4 (Apr. 1983), 259–264.
16. JACOB, R. J. K. An executable specification technique for describing human–computer interaction. In *Advances in Human–Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, N.J., 1985, pp. 211–242.
17. KASIK, D. J. A user interface management system. *Comput. Graph. 16*, 3 (July 1982), 99–106.
18. KIERAS, D., AND POLSON, P. G. A generalized transition network representation for interactive systems. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference* (Boston, Mass., Dec. 12–15). ACM, New York, 1983, pp. 103–106.
19. LIEBERMAN, H. There's more to menu systems than meets the screen. *Comput. Graph. 19*, 3 (July 1985), 181–189.

20. MYERS, B. A., AND BUXTON, W. Creating highly-interactive and graphical user interfaces by demonstration. *Comput. Graph. 20*, 4 (Aug. 1986), 249–258.
21. OLSEN, D. R., AND DEMPSEY, E. P. SYNGRAPH: A graphical user interface generator. *Comput. Graph. 17*, 3 (July 1983), 42–50.
22. PARNAS, D. L. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 24th National ACM Conference*. ACM, New York, 1969, pp. 379–385.
23. REISNER, P. Formal grammar and human factors design of an interactive graphics system. *IEEE Trans. Softw. Eng. SE-7*, 2 (Mar. 1981), 229–240.
24. SCHULERT, A. J., ROGERS, G. T., AND HAMILTON, J. A. ADM—A dialog manager. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference* (San Francisco, Calif., Apr. 14–18). ACM, New York, 1985, pp. 177–183.
25. SHNEIDERMAN, B. Multi-party grammars and related features for defining interactive systems. *IEEE Trans. Syst. Man Cybern. SMC-12*, 2 (Mar. 1982), 148–154.
26. SHNEIDERMAN, B. Direct manipulation: A step beyond programming languages. *Computer 16*, 8 (Aug. 1983), 57–69.
27. SIBERT, J. L., AND HURLEY, W. D. A prototype for a general user interface management system. Tech. Rep. GWU-IIST-84-47, Institute for Information Science and Technology, George Washington Univ., Washington, D.C., 1984.
28. SIBERT, J. L., HURLEY, W. D., AND BLESER, T. W. An object-oriented user interface management system. *Comput. Graph. 20*, 4 (Aug. 1986), 259–268.
29. SINGER, A. Formal methods and human factors in the design of interactive languages. Ph.D. dissertation, Computer and Information Science Dept., Univ. of Massachusetts, 1979.
30. SMITH, D. C., IRBY, C., KIMBALL, R., AND VERPLANK, B. Designing the Star user interface. *Byte 7*, 4 (Apr. 1982), 242–282.
31. WASSERMAN, A. I. Extending state transition diagrams for the specification of human-computer interactions. *IEEE Trans. Softw. Eng. SE-11*, 8 (Aug. 1985), 699–713.
32. WASSERMAN, A. I., AND SHEWMAKE, D. T. The role of prototypes in the user software engineering (USE) methodology. In *Advances in Human–Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, N.J., 1985, pp. 191–209.
33. WASSERMAN, A. I., AND STINSON, S. K. A specification method for interactive information systems. In *Proceedings of the Specifications of Reliable Software Conference*. IEEE Press, New York, 1979, pp. 68–79. IEEE Catalog no. 79CH1401-9C.
34. WEINREB, D., AND MOON, D. Lisp Machine Manual. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1981.
35. YUNTEN, T., AND HARTSON, H. R. A supervisory methodology and notation (SUPERMAN) for human-computer system development. In *Advances in Human–Computer Interaction*, H. R. Hartson, Ed. Ablex, Norwood, N.J., 1985, pp. 243–281.

Received July 1986; revised December 1986; accepted January 1987