



A State Transition Diagram Language for Visual Programming

Robert J. K. Jacob, Naval Research Laboratory

Visual programming is about to emerge from its infancy; graphical programming languages are now taking advantage of its potential even though few graphical representations of abstract objects have been developed.

The diagrams, flowcharts, and other iconic representations we have long employed to communicate with other people can now be used directly to describe algorithms to computers. With the availability of graphics-based, personal workstations, these visual modes can eliminate the need to convert algorithms to the linear strings of symbols traditionally required by most computers. Linear, symbolic computer languages have been studied and refined extensively over the past 30 years, but computer language designers now face a new challenge: to provide convenient and natural *visual* programming languages.

Types of languages for visual programming

Languages for visual programming can be divided into two categories. In the first, the object being designed is itself a static graphical object—a menu, a screen layout, an engineering drawing, a typeset report, a font of type. While such objects are frequently designed with symbolic languages (for example, a picture might be programmed as a sequence of calls to Core graphics subroutines), they are obvious candidates for a direct manipulation¹ or “what-you-see-is-

what-you-get” mode of visual programming. A programming environment for such a visual programming language need only simulate the appearance of the final object and provide direct graphical commands for manipulating it. When the designer is satisfied with the graphical simulation, he “saves” it and has thereby “written” a visual program. Such systems combine great power with ease of use, because the visual programming language provides a natural way to describe the graphical object. It is often so natural that the system is not considered a programming language environment at all, but simply a what-you-see-is-what-you-get type of editor. Unfortunately, this approach is possible only where there is a one-to-one correspondence between a visual programming language and the static visual object being programmed.

A more difficult problem arises in the second category of visual programming language, representing something abstract—time sequence, hierarchy, conditional statements, frame-based knowledge. To provide visual programming languages for these objects, we must first devise suitable graphical representations or visual metaphors for them. The powerful what-you-see-is-what-you-get principle is not much help, since the objects are abstract, but



the successful application of the visual programming language paradigm to these situations still depends critically on choosing a good representation. Graphical representation of abstract ideas is a powerful form of communication, but a difficult one. In the absence of an applicable theory of graphical communication, proper use of such representations often requires extensive experimentation.²

Choosing a graphical representation. The use of visual programming is in its infancy and few good representations of abstract objects have been developed. It is possible, nonetheless, to adapt representations now being used as a medium for discussing algorithms and to examine how they function in visual programming paradigms.

State transition diagrams have been used widely by computer scientists—albeit with pencil and paper—to describe a variety of algorithms. In particular, they offer a good representation of the user interface of a computer system because of several of their properties:

- In each state, they make explicit the interpretations of all possible inputs.
- They show how to change to a state in which such interpretations would be different.
- They emphasize the temporal sequence of user and system actions in the dialogue.

State transition diagrams have been used in this way for several years^{3,4} and have been found preferable to other languages for describing user interfaces.^{5,6} It is time to take this pencil-and-paper tool and apply it to the new paradigm of visual programming.

The state transition diagram notation

First, we must consider the static form of the graphical language and

how it will be used to describe a user interface. A visual representation chosen for this purpose needs to describe the external (user-visible) behavior of the user interface of a system precisely, leaving no doubt as to the behavior of the system for each possible input. It should separate function from implementation, describing the behavior of a user interface completely, and precisely, without unduly constraining the way it will be implemented. The visual representation should be easier to understand and take less effort to produce than the more conventional symbolic software. Ideally, the overall structure of the visual program should represent the cognitive structure of the user interface. The program should describe 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. Finally, the visual representation must be directly executable in a visual programming environment.

The language illustrated here is an extended version of state transition diagrams. It has been used successfully to specify and directly implement user interfaces for several prototype systems.⁷⁻⁹ The language is based on the conventional graphical diagrams that describe finite state automata. A diagram consists of a set of nodes (states) and the links between them (state transitions). Each transition is associated with a token in the user input language. From any state, an input language token initiates a transition labeled with that token. A transition may be associated with an output token, which provides output to the user or a processing action that is performed by the system during that transition.

An important feature of the language is the ability of one diagram to call upon another, much as a program makes a procedure call. This feature is analogous to nonterminal symbols in

BNF—invented intermediate constructs that permit the specification to be divided into manageable pieces. Instead of labeling a transition with a single token to be recognized, we can give it the name of a nonterminal symbol. That symbol is then described by a separate state transition diagram. 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 will help them to construct their own mental models of the system.

One of the principal virtues of state diagram notations is that, by giving the transition rules for each state, they make explicit what the user can do at each point in a dialog and what the effect will be. Feyock¹⁰ makes good use of this property by using a computer-readable representation of the state diagram description of a system as the input to a system help facility. Based on the state diagram and the current state, the system can answer such questions as “What can I do next?” “Where am I?” and “How can I do . . .?”

Other investigators have also found the state transition model helpful in describing a user's mental model of an interactive computer system,^{11,12} and some have built diagram interpreters.^{13,14} The choice of a state-diagram-based notation is also supported by the empirical observation of Guest,⁶ who was surprised to find that programmers preferred for a specification interpreter a state-transition-diagram-based front end to a BNF-based one.

Programming methodology for interfaces

The state transition diagram language used here is part of a methodology for designing and specifying user interfaces. The method is outlined below and described in more detail elsewhere.⁷ To reduce the complexity

of the designer's task, the process of designing and programming a user interface is divided into three levels, and a notation suitable for each level is provided. Foley and Wallace¹⁵ introduced the notion of describing an interactive user interface at the *semantic*, *syntactic*, and *lexical* levels. That model¹⁶ is followed here, but the three levels require more precise delineation, particularly with respect to output and to suitable notations for programming them visually.

Semantic level. The semantic level describes the functions performed by the system. The semantic design indicates the information needed for each function and gives the result. It defines "meanings," rather than "forms" or "sequences," which are left to the lower levels. The semantic-level specification provides the high-level model or abstraction of the functions of the system, removing from the graphical description of the user interface syntax details that would obfuscate the structure of the dialog.

In the state transition diagram language, the semantic level manipulates internal variables; no actual input or output operations are described, although the manipulation of values read in as inputs and the generation of values to be displayed as outputs are described. The semantic-level specification consists of descriptions of functions that operate on the internal data—the function parameters, their types, and the effects of the functions. For actual execution, these functions are coded in a conventional programming language. The semantic level also provides dialog independence¹⁷ by permitting the details of the semantic level of the user interface to be partitioned from the syntactic- and lexical-level specifications and treated separately.

Syntactic level. The syntactic level describes the rules by which sequences

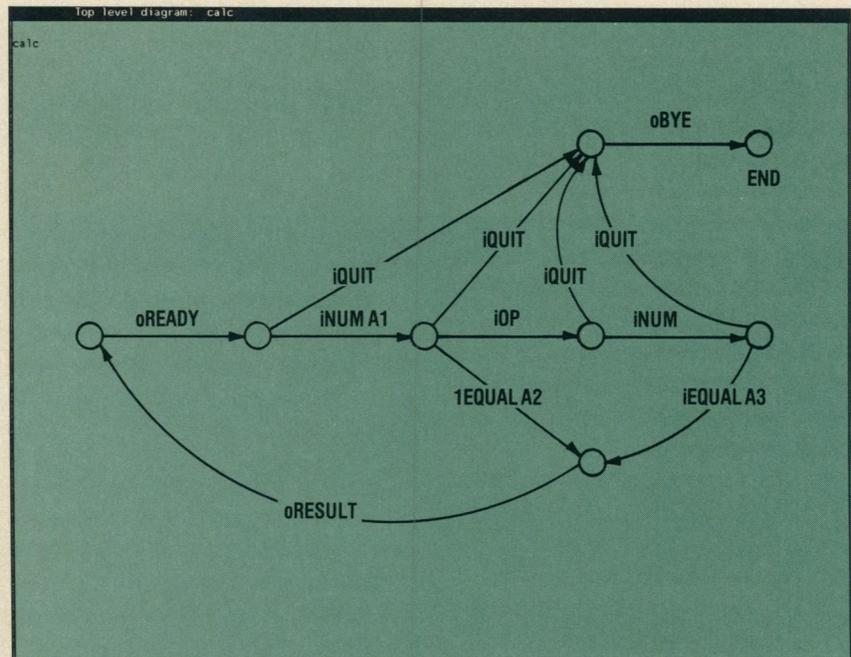


Figure 1. State diagram description of a simple desk calculator.

of words (*tokens*) in the language are formed into proper (but not necessarily semantically meaningful) sentences. The syntactic-level specification describes the sequence of the logical input, output, and semantic operations, but not their internal details. A logical input or output operation is an input or output *token*. Its internal structure is described at the lexical level, while the syntactic level calls it by name, like a subroutine, and indicates when the user may enter it and what will happen next (for an input token) or when the system will produce it (for an output token). A state transition may be associated with an input token or an output token, but not both. Treating outputs as separate tokens on separate transitions (rather than as a special kind of action) at the syntactic-level permits the specification to be more symmetric in the way it describes input and output.

The syntactic-level specification of a simple desk calculator program with this state transition diagram notation

is shown in Figure 1. Each circle corresponds to a state; the start state is at the left; the end state (or states) is named inside its circle. Each transition between two states is shown as a directed arc. It may be labeled with one of the following:

- An input token (lower case *i* followed by upper case, e.g. *i NUM*);
- An output token (lowercase *o* followed by upper case; *oREADY*);
- A nonterminal (all lower case) defined by a separate diagram, called like a subroutine; it must be traversed from start to end to complete the transition;
- An action that calls a semantic-level function; it will be executed if the transition is taken; e.g. *A1*; or
- A condition, defined in the semantic level, which must be true for the transition to be made.

The details of actions and conditions are shown in numbered footnotes to avoid clutter. Each consists of one or more calls to the semantic functions. The semantic actions called by the digram in Figure 1 are shown in Figure

2, along with their definitions, programmed in C. A prompt (consisting of an output token) may also be associated with a state; whenever that state is reached, its prompt token will be output. The actual value received by

an input token (such as the actual number obtained for the token iNUM) is available in a variable named *v* plus the token name (e.g., viNUM); the actual value to be output by an output token is, similarly, set in a variable named *v* plus the token name (e.g., voRESULT). Further details and more precise semantics of this form of state transition diagram notation are provided elsewhere.⁷

Lexical level. The lexical level determines how input and output tokens are actually formed from the primitive hardware operations (*lexemes*). It represents the binding of hardware actions to the hardware-independent tokens of the input and output languages. While tokens are the smallest units of meaning with respect to the syntax of the dialog, lexemes are the actual hardware input and output operations that comprise the tokens. The lexical level identifies the devices, display windows, positions with which each token is associated, and the primitive hardware operations that constitute them. All information about the organization of a display into areas and the assignment of input and output tasks to hardware devices is confined to this level. For an input token, the lexical-level description gives the sequence of primitive input lexemes (for example, key presses) and the device for each lexeme that is used to enter the token, as well as any lexical output that is produced. For an output token, the lexical level tells how (that is, with which devices, windows, positions, formats, colors, and the like) the token appears to the user. The actual information to be presented by an output token may have been set by a semantic action or may be constant; the lexical level shows the format of the variable data displayed, but not its contents.

The lexical level is represented in the same state transition diagram notation as the syntactic. It consists of a separate state diagram for each input or

```

char prevop[100] = "+", prevnum[100] = "0";

A1: assign(*savenum1,viNUM);
/*
 * assign is a built-in function that performs variable assignment
 */

A2: const(*voRESULT,savenum1);
/*
 * The function const performs a constant mode calculation
 * Repeats previous operation with new operand num,
 * and returns answer in result
 */
const(result,num) char **result, *num; {
    int ians; char strans[100];

    if (STREQ(prevop,"+") ians=atoi(prevnum)+atoi(num);
    else if (STREQ(prevop,"-") ians=atoi(num)-atoi(prevnum);
    else if (STREQ(prevop,"*") ians=atoi(prevnum)*atoi(num);
    else if (STREQ(prevop,"/") ians=atoi(num)/atoi(prevnum);

    sprintf(strans,"%d",ians); assign(result,strans);
}

A3: calc(*voRESULT,savenum1,viOP,viNUM);
/*
 * The function calc performs calculator operation op
 * on operands num1 and num2, and returns answer in result
 */
calc(result,num1,op,num2) char **result, *num1, *op, *num2; {
    int ians; char strans[100];

    if (STREQ(op,"+") {
        ians=atoi(num1)+atoi(num2);
        strcpy(prevnum,num1);
    }
    else if (STREQ(op,"-") {
        ians=atoi(num1)-atoi(num2);
        strcpy(prevnum,num2);
    }
    else if (STREQ(op,"*") {
        ians=atoi(num1)*atoi(num2);
        strcpy(prevnum,num1);
    }
    else if (STREQ(op,"/") {
        ians=atoi(num1)/atoi(num2);
        strcpy(prevnum,num2);
    }

    strcpy(prevop,op);
    sprintf(strans,"%d",ians); assign(result,strans);
}

```

Figure 2. Actions called by the diagram in Figure 1.



output token, each of which can be called from the syntactic-level diagrams, like other sub-diagrams for nonterminals. In the lexical-level diagrams, output is described by actions on the state transitions that call special functions. Such functions can be called only at the lexical level. At the syntactic level, output is performed only by output token transitions to avoid mixing output actions with input transitions. At the lexical level, all outputs (other than lexical echoes) have already been separated from inputs.

Certain low-level lexical operations are cumbersome to describe. Tracking a mouse with a cursor, highlighting states in reverse video as they are traversed, making font changes, or even echoing characters—these are better described in a more intuitive fashion than in a large state diagram with a very regular structure. As a simple alternative, such operations could be programmed directly into the definitions of the special output functions mentioned above. However they are represented, it is important that such details be captured at and encapsulated in the lexical-level specification. Once they are associated with token names, their specification has no bearing on the specification of the syntax of the dialog or the use of the state-diagram-based notation for that purpose.

Visual programming with the state diagram language

The language described thus far is a static one that has been used to specify the behavior of a variety of user interfaces. In each case, the diagrams were designed, entered as strings of text, then executed by an interpreter. To use this language in a visual programming paradigm, we need a more graphically-oriented and more interactive programming environment. A visual programming environment for this language is cur-

rently being implemented on a SUN workstation. The displays in Figures 3-7 are taken directly from this system, although all of its parts, particularly the interactive editor and the handling of “co-diagram” calls, are not yet fully implemented.

In the visual programming version of the state diagram language, the programmer enters the state diagrams with a graphical editor and affixes the necessary labels and actions. Each nonterminal, token, and lexeme is diagrammed separately in its own window on the display screen so that the individual graphical objects being edited do not become too complicated. Each diagram is given a name by which it may be called from a transition in another diagram. One diagram is designated the top-level diagram. It calls all the rest directly or indirectly.

Semantic actions. While the diagrams are used to describe both the syntactic and lexical levels of the design, the actions called by transitions in them comprise the semantic level. They are currently programmed in a conventional (non-visual) programming language, and their definitions can be shown and edited with a text editor in a separate window for each diagram. Since they are currently written in a compiled language, they must be re-compiled after they are edited. While a similar interpreted language could be used instead, a more interesting possibility is a visual, interpreted language. One approach is to provide a (compiled) library of basic actions (add, subtract, assign, print), then allow the programmer to draw additional state transition diagrams that combine these standard actions to perform the desired functions. This would permit the programmer to remain within the same visual programming language, even when describing the semantic actions of the system. However, the state diagrams were

chosen because they are a good visual metaphor for describing the user interface behavior of a system. As a general-purpose programming language, they are likely to be bulky and obscure for many actions that could be programmed clearly in other languages. Hence they are preferred for programming the syntax but not the semantics. Ideally, the present state diagrams would call the semantic actions by name, as they do now, and these actions would be programmed in a separate, more suitable *visual* programming language. This programming is possible within the present framework, but awaits the invention of a suitable visual metaphor for the action descriptions.

Interactive execution environment.

The state diagram notation is an executable language, so that the diagrams can be directly executed after they have been entered. A system for executing such diagrams in a non-visual programming paradigm has been developed⁷ and adapted to the interactive visual programming environment. It directly implements the behavior of the user interface as specified in the diagrams. “What you see is what you get” is not a useful maxim in this environment, since the programmer is trying to describe a temporal ordering of events rather than a static visual object. The programming environment has two types of windows. One can be used to demonstrate the user interface being programmed, the other to manipulate the abstract graphical representation of it. As shown in Figure 3, the simulator window on the left shows the newly-designed user interface (of the desk calculator example of Figure 1) as it would appear to the user. When the mouse is pointing to it, it allows the programmer to interact with it in the role of the user. The programming windows on the right show the state diagrams and the action descriptions and allow the programmer

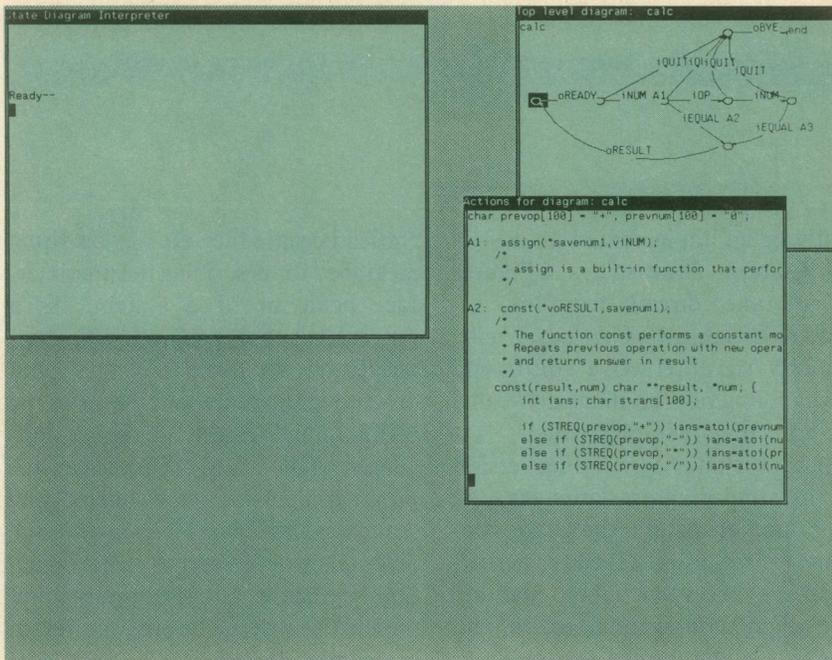


Figure 3. Initial display screen during execution of the calculator diagram.

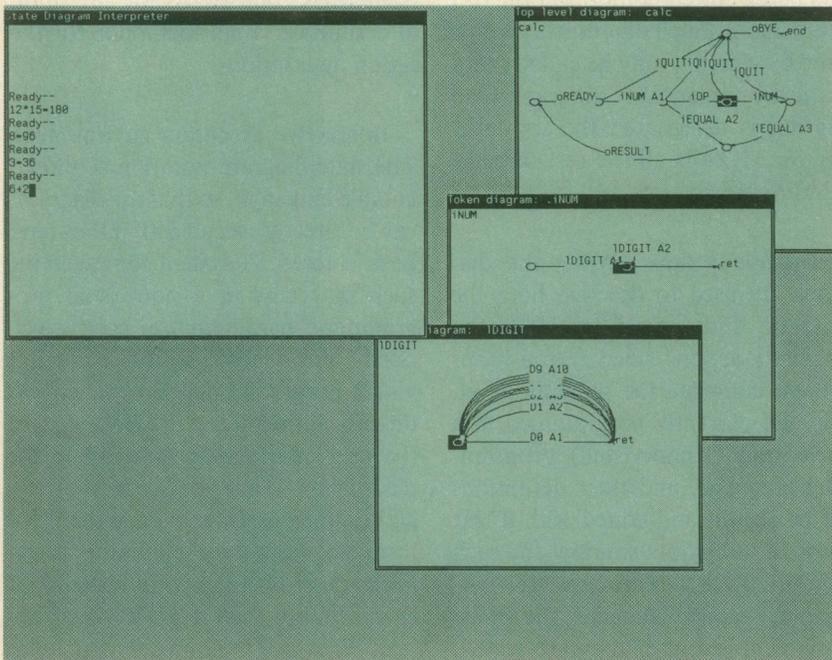


Figure 4. Display screen after execution of the calculator diagram has progressed to the point where subdiagram calls for tokens are pending.

to modify them as desired. This approach does not preclude designing a user interface that, itself, involves several windows; it just requires that the programmer view a picture of it that is small enough to leave room on the screen for the programming windows.

When execution begins in the simulator window, a programming window with the top-level diagram pops up. As

execution of the diagram progresses, the current state in the diagram is highlighted on the screen. As each subdiagram is called for a nonterminal, token, or lexeme, a window containing the diagram pops up. During such calls, the current state of the system consists of a state in the lowest level diagram currently being executed, along with a stack of pending

“subroutine” calls to it from the higher level diagrams. It is displayed as a highlighted state in the lowest level diagram with a pile of windows behind it showing the chain of diagrams that called it. In each window, the state from which the pending call was made is highlighted. Figure 4 shows the display in such a situation (the user has also suppressed the display of the windows with the action bodies in this and subsequent examples).

During execution, the programmer can cause the interpreter to move directly to any state in the diagram by pointing to it. He can cause the interpreter to move to a state via a particular path by pointing to one transition or a sequence of them. (As with other interactive debuggers, the programmer must avoid skipping over a transition that performs an action and subsequently invokes a transition that depends on that action.) He can also edit any of the diagrams with a graphical editor while the system is running. Since the state diagram language is entirely interpreted, the program can be modified arbitrarily while it is running. Of course, if the modification deletes the current state, the programmer will have to point to another state from which execution is to resume.

Nondeterministic programs. It is possible to write a nondeterministic program in this language; there could be several transitions leading from the same state, all accepting the same input token but clarified by subsequent input. In fact, it is possible to write a nondeterministic program inadvertently with calls to sub-diagrams. However, when an interactive system is being described, there cannot be a nondeterministic choice at any point in the diagram involving output. Whenever the interpreter makes a transition that produces output, it is committed to the path with that transition because an interactive system cannot rescind

outputs that the user has seen or heard. It must produce outputs at the specified points in the dialog. It cannot wait for additional input to select between two transitions. The interpreter itself is a deterministic machine that executes a nondeterministic program by arbitrarily trying one of the possible paths. If it reaches a dead end, it backtracks and tries another. The language permits arbitrary nondeterminism (simulated by backtracking), but the programmer must avoid backtracking over output. The visual programming environment makes this situation easier to see. Whenever there are several possible paths to be taken, the system will show all of them, as separately highlighted target states or, more typically, as separate windows with possible pending subdiagram calls. As soon as user input is sufficient to permit a choice of one of the paths, the rejected options disappear from the display. Figure 5 shows what this nondeterministic choice would look like in a simplified situation. In this state, the user could enter an operator (the token iOP) or an equals sign (iEQUAL) or a quit command (iQUIT). The three subdiagrams for the tokens appear on the screen at this point. In this example, it is easy to see that the very next user input will select from among three clear options (since the three sets of lexemes are disjoint in their current states), and two of the subdiagrams will disappear.

Programming a multi-window user interface

The simplest type of user interface involving several windows has only one context for user interaction. All user input is considered part of one unbroken dialog, with no context switches. Output may appear in various windows for convenience, but it is all part of a single dialog. To describe this with the present tech-

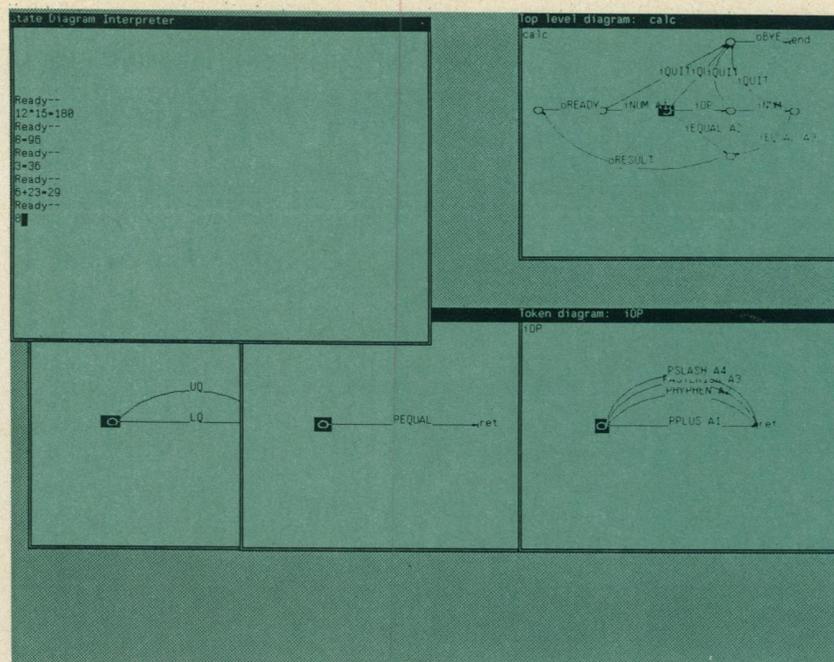


Figure 5. Display of a nondeterministic branch in the diagram being executed.

nique, each output token is tagged with a specific window in its lexical-level diagram, as Shneiderman¹⁸ has done in his extended form of BNF. Several systems with a multi-window user interface have been specified and built with this approach with the state transition diagrams.^{7,8}

Concurrent dialogs. A more interesting situation arises when there are several dialogs taking place. Multiple dialogs are typical with more sophisticated display window systems, where one window is associated with each conversation. This use of windows gives rise to a new style of user interface. For example, consider a multi-window user interface for a prototype military message system.⁹ It permits manipulation of several different messages, message files, and message file directories concurrently, each in its own window. The user can create display windows for various objects, conduct a separate dialog for viewing or manipulating the objects in each window, and change focus from one window to another at any time without losing the place in any of the dialogs. The user can also change the layout of the windows on the display at any point. Certain input actions are reserved for directives to a window man-

ager for changing the focus of the conversation or the layout of the display. All other inputs are considered input to the "current" dialog, as designated by the most recent command to the window manager. Figure 6 shows a display from such a system.

To describe such a system in a convenient visual notation, it is first necessary to consider the structure of the dialog. Its central concept is the *sequence* of (input and output) events that comprise it, not the layout of the windows on the display (although some logical layout information is provided with the tagged tokens mentioned above). Further programming of the details of the layout are easy to accommodate with a what-you-see-is-what-you-get screen editor, but the main user interface specification should concentrate on the sequence of the dialog.

The syntactic description of a concurrent dialog system thus has two levels. The top level describes the display-arranging commands of the window manager itself. Then, each dialog in each window (such as those for viewing or editing individual message files or other objects) has its own syntax, which is described at a second level although window manager commands can be entered at any time.

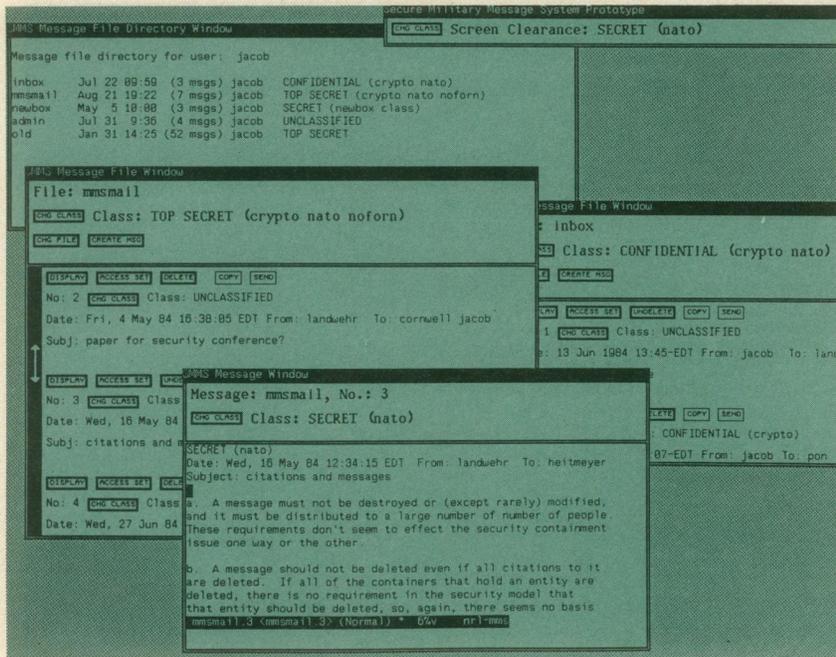


Figure 6. User interface of a prototype multiwindow military message system; the security classifications shown are simulated for demonstration purposes.

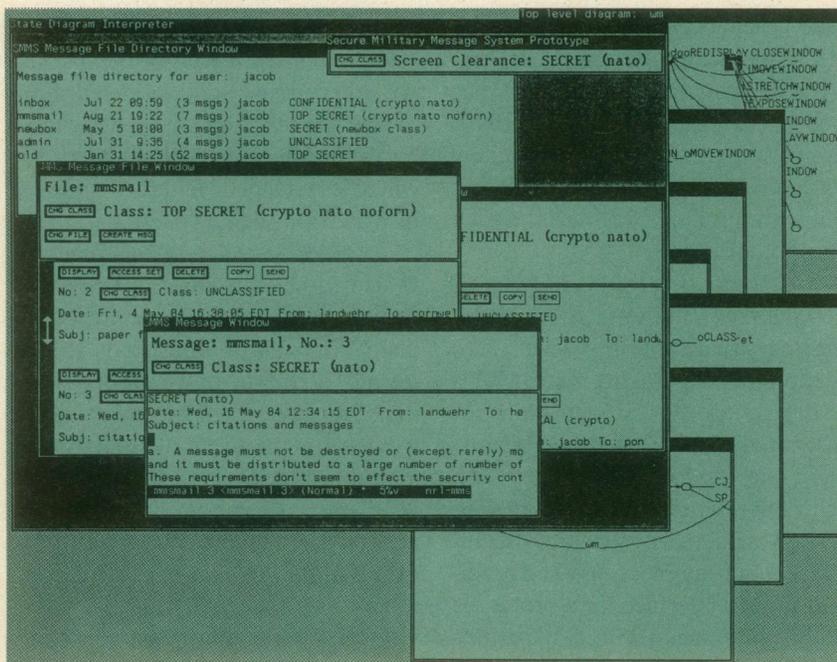


Figure 7. Appearance of the programming environment display when the system being designed (from Figure 6) itself involves several windows.

Such commands will immediately switch the context to the window manager itself, then back to the previous point in the current dialog or to a dialog in another window.

What is the abstract structure of such a multi-window style of dialog? It is really a set of coroutines, and the problem of representing it may be

solved by introducing a codiagram call into the present language. This is analogous to the subdiagram call, by which a transition in one diagram is used to call another subdiagram—like a subroutine. When a transition in Diagram A, for instance, makes a codiagram call to Diagram B, Diagram B is entered at the state from which B last

executed a codiagram call. It is traversed until it makes another codiagram call. For example, if it then called diagram A, A would be resumed at the end state of the transition from which it had called B. Whenever a diagram is entered by a codiagram call, it is resumed with its own stack of pending subdiagram calls intact. If it had made a subdiagram call, it would be resumed within that subdiagram, and upon exit from that subdiagram, it returns to the diagram that called the subdiagram.

With this approach, the syntax for the window manager commands is described in one top-level diagram. Whenever an input other than a window manager command is received, the top-level diagram makes a codiagram call to the syntax diagram for the currently-selected individual dialog. These syntax diagrams are conventional, except that every state permitting escape to the window manager command level is preceded by a transition that makes a codiagram call to the top-level window manager diagram. When the lower-level dialog is resumed after that call, it is ready for user input at the point in its input syntax at which it was interrupted.

Figure 7 shows how such a system might appear while being designed with the visual programming environment. The simulator window shows the entire multiwindow user interface of the prototype military message system (with its windows reduced or truncated), while the programming windows handle the graphical editing of the state diagrams for both the top-level window manager dialog and each of the individual window dialogs.

Use of the visual programming paradigm, particularly for abstract objects, is in its infancy. What-you-see-is-what-you-get types of user interfaces have been highly successful but have no obvious extension for non-graphical objects. To explore the



potential for visual programming, a demonstration of a visual programming environment with an already-established graphical language has been presented. To date, only a few suitable graphical representations of abstract objects have been developed for use with the visual programming paradigm. They are typically designed for specific purposes and are not widely applicable. The need now is for more general graphical representations of programming objects. In the long run, a theoretical understanding of visual perception is needed so that a designer can devise natural graphical representations for a wide variety of objects in a more general manner. □

Acknowledgments

This work was supported by the Naval Electronic Systems Command under the direction of H. O. Lubbes. I want to thank the guest editor, Bob Grafton, for encouraging me to think about the possibilities of visual programming.

References

1. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *Computer*, Vol. 16, No. 7, July 1983, pp. 57-69.
2. R. J. K. Jacob, "Facial Representation of Multivariate Data," in *Graphical Representation of Multivariate Data*, P. C. C. Wang, ed., Academic Press, New York, 1978, pp. 143-168.
3. D. C. Engelbart and W. K. English, "A Research Center for Augmenting Human Intellect," *Proc. Joint Computer Conf.*, Fall 1968, pp. 395-410.
4. 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 Conf.*, 1969, pp. 379-385.
5. R. J. K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *CACM*, Vol.

- 26, No. 4, Apr. 1983, pp. 259-264.
6. S. P. Guest, "The Use of Software Tools for Dialogue Design," *Int'l J. Man-Machine Studies*, Vol. 16, No. 3, Mar. 1982, pp. 263-285.
7. R. J. K. Jacob, "An Executable Specification Technique for Describing Human-Computer Interaction," in *Advances in Human-Computer Interaction*, H. R. Hartson, ed., Ablex Publishing Co., Norwood, N.J., 1985, pp. 211-242.
8. R. J. K. Jacob, "Designing a Human-Computer Interface with Software Specification Techniques," *Second Symp. Empirical Foundations of Information and Software Sciences*, 1984, Atlanta, Ga.
9. M. R. Cornwell and R. J. K. Jacob, "Structure of a Rapid Prototype Secure Military Message System," *Seventh DOD/NBS Computer Security Conf.*, Gaithersburg, Md., pp. 48-57.
10. S. Feyock, "Transition Diagram-Based CAI/HELP Systems," *Int'l J. Man-Machine Studies*, Vol. 9, 1977, pp. 399-413.
11. J. Darlington, W. Dzida, and S. Herda, "The Role of Excursions in Interactive Systems," *Int'l J. Man-Machine Studies*, Vol. 18, No. 2, Feb. 1983, pp. 101-112.
12. T. P. Moran, "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Int'l J. Man-Machine Studies*, Vol. 15, No. 1, July 1981, pp. 3-50.
13. M. B. Feldman and G. T. Rogers, "Toward the Design and Development of Style-independent Interactive Systems," *Proc. ACM Sigchi Human Factors in Computer Systems Conf.*, 1982, pp. 111-116.
14. A. I. Wasserman and D. T. Shewmake, "The Role of Prototypes in the User Software Engineering (USE) Methodology," in *Advances in Human-Computer Interaction*, H. R. Hartson, ed., Ablex Publishing Co., Norwood, N.J., 1985.
15. J. D. Foley and V. L. Wallace, "The Art of Graphic Man-Machine Conversation," *Proc. IEEE*, Vol. 62, No. 4, Apr. 1974, pp. 462-471.

16. J. D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., 1982, pp. 217-243.
17. T. Yuntan and H. R. Hartson, "A Supervisory Methodology and Notation (SUPERMAN) for Human-Computer System Development," in *Advances in Human-Computer Interaction*, H. R. Hartson, ed., Ablex Publishing Co., Norwood, N.J., 1985.
18. B. Shneiderman, "Multi-party Grammars and Related Features for Defining Interactive Systems," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. SMC 12, No. 2, Mar.-Apr. 1982, pp. 148-154.



Robert J. K. Jacob is a computer scientist in the Information Technology Division at the Naval Research Laboratory, Washington, DC, where he is working on man-machine interactions. He is focusing on his work on the formal specifications of user-computer interfaces and on the development of software engineering techniques to simplify program modification by users.

Jacob received his BA, MSE and PhD (EE/CS) from Johns Hopkins University in 1972, 1974, and 1976, respectively. He is currently on the faculty of George Washington University, where he teaches computer science.

Questions about this article can be addressed to Jacobs, Code 7590, Naval Research Laboratory, Washington, DC 20375.