1989

Statemaster: a user interface management system based on statecharts

Pierre Wellner

Statemaster: A User Interface Management System based on Statecharts

Pierre Wellner

February, 1989

Masters thesis, submitted to the RIT Department of Computer Science.

Approved by:

Professor John A. Biles

Dr. Acco Hengst

Dr. Peter Anderson

TITLE OF THESIS:

STATEMASTER: A UIMS BASED ON STATECHARTS

I PIERRE WELLNER HEREBY GRANT PERMISSION TO THE WALLACE MEMORIAL LIBRARY OF RIT TO REPRODUCE MY THESIS IN WHOLE OR IN PART, PROVIDED THAT MY NAME APPEAR AS THE AUTHOR, AND THAT ANY REPRODUCTION NOT BE USED FOR COMMERCIAL USE OR PROFIT.

MARCH 9, 1989

Abstract

Designing and implementing a user interface (UI) is among the costliest tasks in developing interactive software systems. As a result, much work is being done in the area of User Interface Management Systems (UIMSs). These systems make it easier to develop complex user interfaces more quickly because specification of user interface dialogs is at a closer level to that at which UI designers express themselves. This paper describes the generic structure of a UIMS and several examples of previous work in the field. Many UIMSs are implemented as explicit state machines that use conventional state transition diagrams or state metaphors to specify dialogs.

The rest of this paper describes the design and implementation of Statemaster, an event-driven UIMS based on statecharts. Statecharts are a hierarchical extension of state diagrams well suited for describing complex reactive systems with a compact, visual notation. These diagrams are directly implemented by Statemaster with an object-oriented architecture in the C++ programming language. Statemaster has been found to be general enough to implement a wide range of user interface dialogs. It can be used as a prototyping tool for UI development, and it is efficient enough to be used as the final, target-intent implementation.

Keywords

UIMS, user interface, state diagrams, statecharts, visual programming, rapid prototyping.

Acknowledgements

The ideas presented in this paper were refined through numerous helpful discussions with Steve Dacek and Paul Wlodarczyk, and as the first real user of Statemaster, Linda Isaacson also provided valuable feedback. I especially thank Linda Malgeri for her many suggestions and unfailing support. The environment for this project was provided by Xerox Corporation.

Table of Contents

Chapter 1 : Introduction

Designing and implementing an effective graphical user interface (UI) is often among the largest and costliest tasks in developing interactive software systems. As a result, many systems today have poorly developed UIs, and they are difficult to understand or hard to use. To make effective user interfaces easier to create, much effort is being directed towards the development of User Interface Management Systems (UIMSs), which offer powerful means for specifying and modifying user interfaces. This paper describes several UIMSs and points out that many are implemented as state machines. Although conventional state diagrams have significant advantages, they suffer from drawbacks that make them impractical for specification of complex user interfaces. Statemaster is based on a graphical notation (Statecharts) that overcomes these drawbacks, and it can be used both for user interface prototyping and target-intent implementation.

1.1 Difficulties in developing user interfaces

There are no guidelines or design elements that can guarantee a user interface will be learnable and easy to use. Consequently, the only reliable method for creating quality user interfaces is to test prototypes with actual users and modify the design based on their problems, success, and feedback [Myer87]. This process of iteratively designing, implementing, testing and redesigning is very time consuming and expensive. The problem is compounded by the fact that in many organizations, the people who design the dialog and graphics of the UI have a very different set of skills from those who implement the UI software. The designers are the experts on what people want from the application and how they want to perform the task. They think in words and pictures, and their notations tend to be informal. The implementors, on the other hand, think in terms of how the machine can perform the task. In order to create a working system, they use precise formal notations, programming languages, and they consider technical issues that designers don't need to know about. These two groups need to work very closely through all the iterations, and communication is often poor. There is a long turnaround time between the design and test phase. Because the product must adhere to a schedule, few iterations are possible, and the quality of the resulting user interface is compromised.

This situation is very similar to the one that existed in the field of expert systems in the nineteen seventies. Expert systems were implemented in a general programming language such as LISP, and the domain expert had knowledge that could not be easily expressed in terms of formal rules. Knowledge engineers had to bridge the gap, translating the domain expert's knowledge into formal specifications that could be implemented on a machine. This was a difficult and time consuming task, so a great deal of effort was put into the development of expert-system building tools ,which bring the formal specification of knowledge closer to the level at which domain experts express themselves, using visual specification languages with rules, examples, and object hierarchies [Wate86]. Many classes of expert systems now can be developed in a fraction of the time and resources that were required in the seventies.

1.2 User Interface Management Systems

Similarly, to reduce the cost of developing graphical user interfaces, UIMSs have been developed for making user interfaces easier to create and modify. As with expert system building tools, UIMSs bring the specification of user interface dialogs closer to the way in which designers express themselves.

UIMSs provide a software architecture that is general enough for a range of user interface dialogs to be implemented with the same system. They separate the user interface software from the application and often provide it with some hardware independence. The goal of these systems, of course, is to shorten the design-implement-test cycle, thus facilitating the development of quality user interfaces.

1.3 Collaborative design and the UI Engineer

Design and implementation of user interfaces at Xerox and in other organizations is done by teams rather than by individuals. Human factors (HF) designers, graphic artists, and software engineers work together on dialogs. Organizational issues can play an important role in the use of a UIMS [Bigh87]. Users of the UIMS must be able to share and discuss UI specifications with other members of the team.

Some UIMSs claim to allow UI designers untrained in programming skills to develop user interfaces without having to program. This claim is difficult to

confirm or deny because the definition of programming is very nebulous. Some people feel that using a spreadsheet is too high level to be considered "programming," while others feel that "programming" skills are required even to use a graphics editor. At Xerox, for example, human factors designers do not have the programming skills required to develop complete user interfaces using currently available UIMSs. They may use a UIMS such as Trillium or Hypercard to illustrate components of their designs, but ultimately their designs are implemented by programmers. At the risk of creating yet another job title, the people who actually implement the user interfaces could be called "UI Engineers" because of the similarity of this job to that of a knowledge engineer. A knowledge engineer must implement a domain expert's knowledge using expert system building tools. Similarly, a UI engineer must implement a human factors designer's dialog using a UIMS [Bett87].

The first knowledge engineers were AI programmers who only learned what they had to about the domain they were building an expert system for. Today, knowledge engineers often have a stronger background in the domain of the expert system than they do in AI programming. The UI Engineers of today, on the other hand, have stronger backgrounds in programming than in human factors design. UIMSs are not yet at the point where a human factors designer can implement his or her creative ideas for interactive graphical user interfaces without requiring substantial programming skills.

Chapter 2: Previous work and background

The design for Statemaster has been influenced by research in several areas. It is most related to previous work done with UIMSs, especially those that have state-based architectures. Other key ideas came from Statecharts, an enhanced graphical state-transition notation, and from object-oriented programming languages. This chapter begins with a detailed discussion of the generic structure of a UIMS; then, after discussing state machines and Statecharts, it discusses several examples of previous UIMSs and object-oriented programming.

2.1 Generic structure of a UIMS

Most existing UIMSs have a common structure, which is illustrated by Figure 2.1. The systems described in this chapter are structured in this way, although very different approaches exist for implementing each of the components.

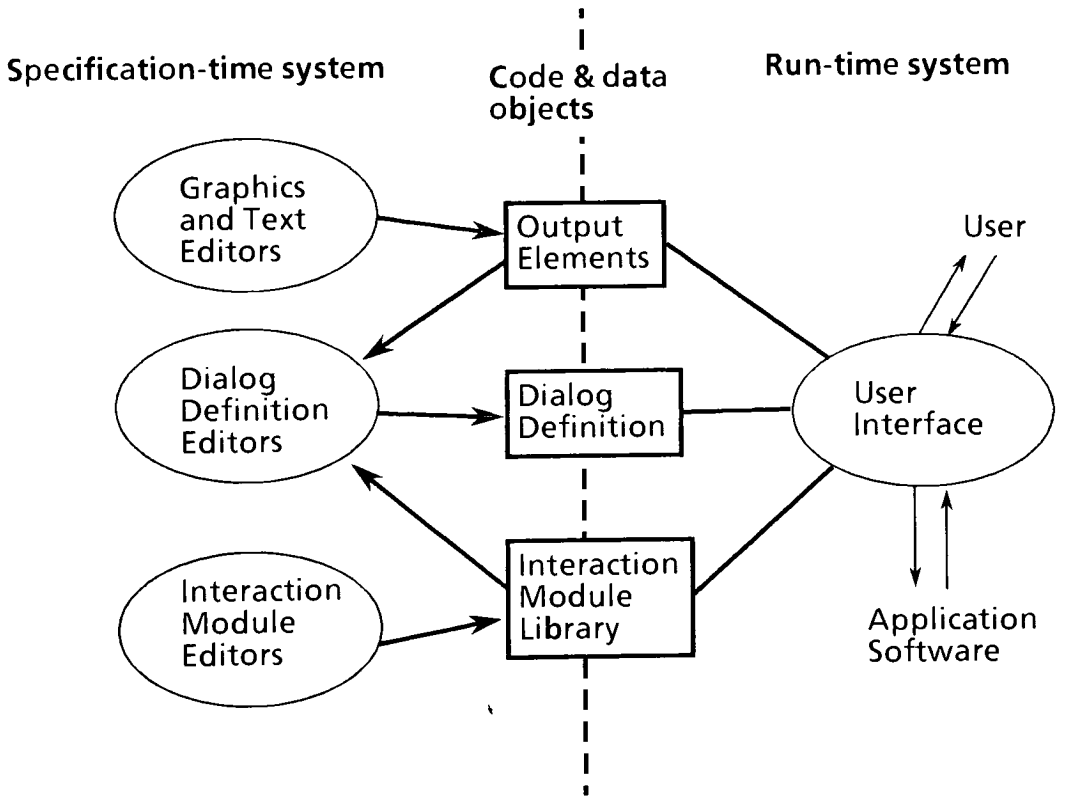


Figure 2.1
Generic Structure of a UIMS

All UIMSs have a specification-time system and a run-time system, although they are sometimes so well integrated that it is hard to distinguish them. Specification time is when the user interface is formally defined by the designer or software engineer. Run-time is when the user interface actually responds to user inputs and communicates with the application software.

2.1.1 Code and data objects

The *code and data objects* are created at specification time, and the runtime system uses them to respond to the user. These code and data objects can be broken up roughly into three parts: the output elements, the interaction module library, and the dialog definition.

The *output elements* typically are made up of graphics and text messages. They may take the form of text files, bitmap arrays, sounds, melodies or display lists that draw complex images.

The *interaction module library* implements the basic techniques by which the user interface interacts with the user. An example of an interaction module might be a simple text-entry facility, a scroll bar, or a graphic button. Interaction modules can be implemented at high or low levels. At a low level, an interaction module might implement no more than some device-specific code. At a high level, an interaction module might implement complex input techniques with graphic feedback such as scroll bars or windows.

The *dialog definition* is sometimes referred to as the "glue" that ties the other pieces together. It determines how interaction modules refer to graphic or textual elements , and it usually determines the positions and layout of those elements. It also determines what syntax is expected for user entries, and which inputs are meaningful.

2.1.2 Specification time system

At specification time, the above three types of code and data objects are created and edited. Consequently, there are three corresponding editors: the output element editors, the interaction module editors, and the dialog definition editors. These three types of editing typically require different skills and may sometimes be performed by three different people or organizations.

2.1.2.1 Graphics and text editors

Editing the output elements is usually done with text editors and graphic design tools. Object-oriented graphics tools like MacDraw are the most useful because complex graphical interfaces are typically composed of objects like fill regions, lines, bitmaps, and text. The graphics tools also must compress their outputs into encoded data structures to minimize the space they take up on the runtime system.

2.1.2.2 Interaction module editors

Interaction modules are usually coded in a conventional programming language for maximum flexibility and optimal performance. For example, if a certain kind of button were needed in the interaction module library, the programmer might write a program using an editor, C compiler, and debugger. The interaction modules often can be modified only with programming tools, so they are more difficult to change than other parts of the UIMS and, consequently, are often considered fixed. A fixed library has the advantage that the UI will look and act consistently with others built from the same library. It has the disadvantage, however, of limiting the designer to the library's primitives. The interaction modules are critical in determining the range and styles of user interfaces that the UIMS can implement. If a new style of interaction is desired, then sometimes new modules must be added to the library by a programmer.

2.1.2.3 Dialog definition editors

The dialog definition editor organizes or "glues" interaction modules and user outputs together to construct a user interface. It allows the designer to try different layering, organization, and syntax to see what is most effective. Dialog definition is often the area where iterative design is most needed, so it is these editors that most UIMSs stress. Often the dialog definition editors are meant to be used by non-programmers (*i.e.* Trillium and Hypercard -- see sections 2.2.6 and 2.2.8). Sometimes the difference between editing the dialog definition and editing the interaction modules is fuzzy. A powerful dialog editor may be able to construct high level interaction modules, and the programming tools used for constructing interaction modules also can be used to construct dialogs. The choice is determined by the capabilities of each tool set and the preferences of the designer or software engineer.

2.1.3 Runtime system

The runtime system uses all the code and data objects created by the specification system and actually runs the user interface by responding to user inputs and communicating with the application software. The runtime system is often used as a prototype, which lets the designer try his ideas to get a feeling for what the interface looks like, but is not used as the final product. Many UIMSs are implemented with high level dynamic languages (*ie* Smalltalk or Lisp) and run on hardware with powerful processors and large virtual memory. Some of these features may be impossible to implement on the cost constrained target hardware. Consequently, the system has to be reimplemented to fit within memory, disk and MIPS limitations. Usually the target runtime system is implemented in a language more suited to low-level control and efficiency, and although time is saved by applying experiences gained from prototyping, the final system must nevertheless go through its own development cycle.

2.2 State based UIMSs

Finite state machines are used in theoretical computer science to specify and model arbitrary computations. Many UIMSs use explicit state machines to specify dialogs because they are simple and general.

2.2.1 State transition diagrams

State machines can be intuitively represented by state transition diagrams. These diagrams are much easier to understand than textual specifications. Three properties of state transition diagrams make them a good representation for user interface dialogs [Jaco85]:

- In each state they make explicit the interpretation of all possible user inputs.
- They show how to change to a state where the interpretations are different.
- They emphasize the time sequences of user inputs and system actions.

State transition diagrams provide a top-level point of view on the dialog that makes it easy to see at a glance how the user interface is organized.

2.2.2 Newman's Reaction Handler

Newman's "Reaction Handler" [Newm68] is generally considered to be the first UIMS. It was a system for implementing simple user interfaces from conventional state transition diagrams. The example used in his paper is a system for drawing rubber band lines with a light pen. The specification for that system is the state diagram illustrated in Figure 2.2..

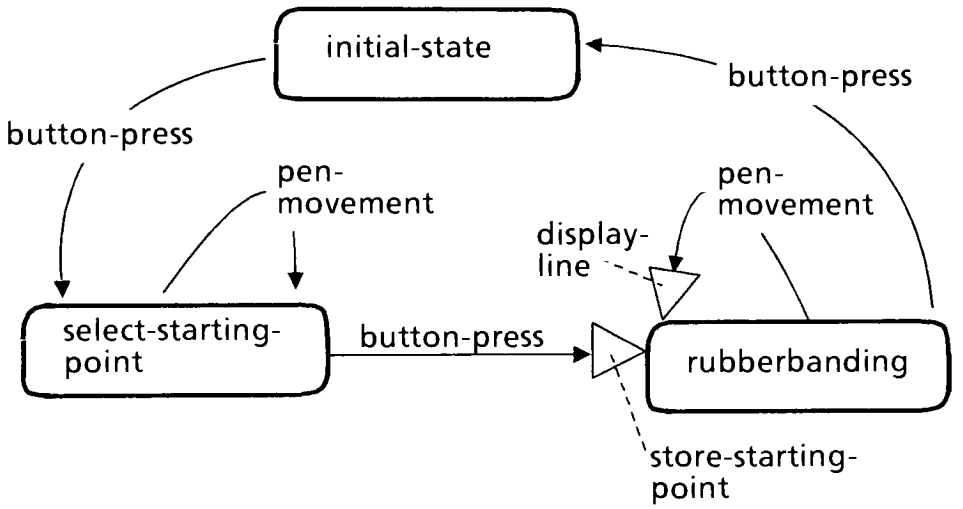


Figure 2.2
Specification of rubber banding for Newman's Reaction Handler

This example illustrates a state machine that responds to two events: button-press and pen-movement. The system starts out in the initial-state and a button-press brings it to the select-starting-point state. Here, pen-movement does not change the state of the system but another button-press causes a transition to rubberbanding after executing the store-starting-point action represented by the triangle. In rubberbanding every pen-movement executes the display-line action until another button-press, which brings the system back to the initial-state.

2.2.3 Drawbacks of state transition diagrams

Despite all their advantages, conventional state transition diagrams have several drawbacks that make them impractical for specification of large, complex systems. [Hare87b].

(a) Conventional state diagrams are 'flat.' They are not grouped to provide depth, hierarchy or modularity, and, therefore, do not support top-down or bottom-up development.

(b) Conventional state diagrams are uneconomical when it comes to transitions. An event that causes the same transition from a large number of states must be attached to each state separately.

(c) Conventional state diagrams require exponential growth in states as the system grows linearly, because every possible state must be represented explicitly.

(d) Conventional state diagrams are inherently sequential, and can not easily represent concurrent activities.

The following dialog design statements illustrate the type of things that are difficult to represent with conventional state diagrams.

> "Job programming consists of the following three modes: basic features, added features, and expert applications." [difficult because of (a)]
>
> "No matter where you are in the user interface, pressing the Reset icon will bring bring you back to the walkup screen, except if you're in a fault condition." [difficult because of (b)]
>
> "While in the basic features mode, the number pad is active, as are the following columns: paper size, copy contrast, copy output, . . . " [difficult because of (d)]

Since Newman's Reaction Handler, many more state-based UIMSs have been developed, and most attempt to address some or all of these drawbacks. The following sections discuss systems that attempt to overcome these limitations by extending state transition diagrams or by abandoning transition diagrams altogether in favor of state metaphors. The following sections describe Wasserman's RAPID/USE system and Jacob's visual programming system, both of which use state diagrams to specify user interfaces. Other systems, such as Trillium, Demo II and Hypercard don't use state transition diagrams, but instead use a metaphor for states. Trillium uses "frames," Hypercard uses "cards," and Demo II uses "slides."

2.2.4 Wasserman's system

Wasserman's RAPID/USE system [Wass85] is similar to Newman's but includes a graphic editor for specifying state diagrams. He found that in using his system, he often needed to create very large and complex diagrams that could not be easily written or understood. This often arose from the desire to specify error handling and help facilities. To solve this problem he introduced "subconversations" as a means for one diagram to invoke another diagram and represent diagram hierachies. An example of subconversations in the UI of an automated teller machine is illustrated in Figure 2.3.

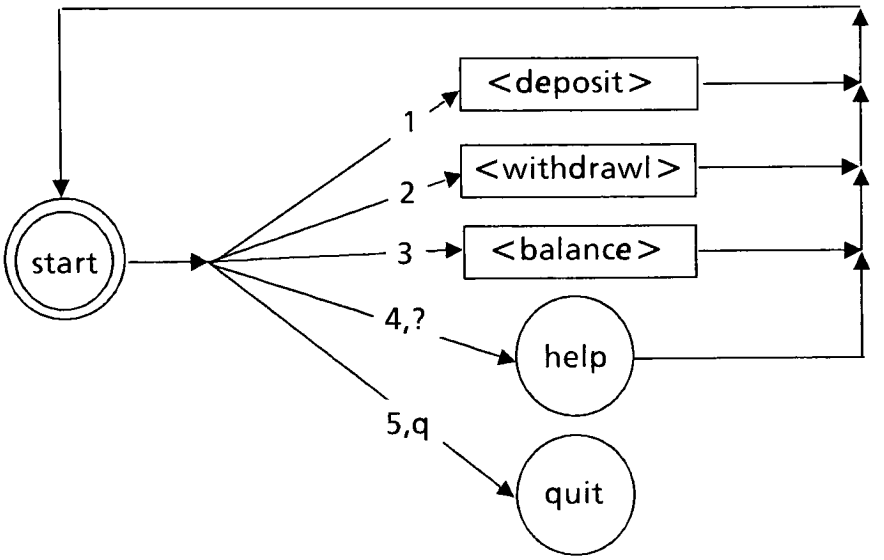


Figure 2.3
Use of subconversations in RAPID/USE

Each of the boxes labeled <deposit>, <withdrawal>, and <balance> refer to separate state diagrams. These diagrams must have a state labeled "exit," which makes them return control to the calling diagram. The RAPID/USE system is currently available as a product from Interactive Development Environments. It is integrated with a relational database and a suite of other software engineering tools that can perform consistency checks and create an executable prototype of the user interface.

2.2.5 Jacob's system

Jacob's system [Jaco85] is similar to Wasserman's in that it uses a graphic editor to input state transition diagrams, and his system provides for

"subdiagrams," which are essentially the same thing as Wasserman's subconversations. In contrast to Wasserman's system, however, he provides a way of specifying concurrency through the use of "codiagrams." When diagram A makes a codiagram call to diagram B, then diagram B is entered at the state from which B last executed a codiagram call. B is then traversed until it makes another codiagram call. If B calls A, for example, then A would resume execution from the point where it had called B. Although this approach does provide a way of specifying concurrency, it does not seem to indicate the concurrency in a clear visual way, and his paper does not illustrate what codiagram calls look like.

2.2.6 Trillium

Trillium [Hend86] originally was developed at Xerox for designing hard user interface panels for copiers. Its dialog definition editors do not use state transition diagrams, but they do allow the designer to place items such as buttons, icons, light bars and dials on the screen. All the generic UIMS components are very well integrated. The specification-time and run-time modules are almost indistinguishable, allowing the designer to immediately run, suspend, and edit his or her interface at any time. Trillium's interaction modules are called *itemtypes* and are implemented in Interlisp. They have been extended far beyond their original set of hard panel controls, and now Trillium is used inside and outside of Xerox to prototype a variety of screen based graphical user interfaces.

A dialog is constructed in Trillium by placing itemtypes and artwork onto Interfaces, Frames, and Superframes. The structure of Trillium dialogs has much in common with a state machine. Each frame can be thought of as a state, and "frame changes" are like state transitions. Several frames can contain the same Superframe to specify grouping, and a frame can contain several simultaneously active superframes to specify concurrency.

A large part of the effort that goes into implementing a Trillium prototype can not be used in the target implementation if it runs on different hardware because Interlisp code is not easily ported to embedded systems. The code must be entirely rewritten for the target, and specific dialog processing tools must be written to translate Trillium Lisp structures into target-intent data structures.

Another problem with Trillium is that it has inadequate facilities for presenting the overall structure of a complex dialog. The only way to understand how a dialog is put together is to examine every frame to see what superframes it points to and also to examine the "change frame" buttons, which execute state transitions. This is because Trillium does not provide any visual representation of its dialogs. One reason for this is that the "superframe" relationship is difficult to represent in a compact visual way that preserves the properties of grouping and concurrency.

2.2.7 Dan Bricklin's Demo II program

Demo II is a PC-based tool meant for prototyping PC applications, demonstrations, and tutorials. [Bric87] Demo II dialogs are "slide shows," and a "slide" is the metaphor for a state. There are two types of slides: text and bitmapped, and each slide fills the entire screen. Special editors are included for designing the appearance of slides. Text slides can be overlayed to allow displaying of several slides at once. The bottom slide is the "background," and overlayed slides can be partially transparent to allow lower slides to show through. Each slide points to a list of event-action pairs, which are called the Run Actions. The *View Slide* action performs a state transition to another slide. Most of the other actions are for creating interaction modules and provide a general purpose programming language. The runtime system passes each user event to the active slide, and the slide executes the action associated with that event. If the slide does not have that event in its Run Actions, it can optionally pass the event to another slide. Otherwise, the "Global Run Actions" are searched.

Demo II is an event-driven state machine based on slides, and it addresses some of the drawbacks of conventional state diagrams by providing grouping, but it doesn't provide concurrency. Grouping in Demo II is done with the *Use* action. A set of slides that must be grouped all can have a default action to *Use* the same slide. Grouping the slides visually is also easy with Demo II's overlay mechanism. Specifying concurrency is not much better than with conventional state diagrams, however. The best the designer can do is link all the concurrent states by a chain of *Use* actions. This will work when a single slide in the group needs each event, but it will not work when several slides respond to the same event.

Unlike Trillium, Demo II clearly separates the specification time system from the runtime system. In fact, the runtime system can be separated and distributed without royalties. Unfortunately, the system is completely dependent on the IBM PC's text screen architecture, and its usefulness is limited to PC-compatible text screen hardware.

As with most UIMSs it is difficult for the designer to get a top-level or global point of view of his design with Demo II. The designer can see only one slide at a time and must edit a slide to see what it is connected to. With small designs this does not present a problem, but as the user interface grows more complex, it becomes difficult to track which slides are used by which, and what the overall structure of the interface is.

2.2.8 Hypercard

Hypercard [Good87] runs on the Macintosh and, like Trillium, is graphics-based and has completely integrated specification and run time systems. It has excellent built-in graphics editors, and it allows the designer to edit a specification and immediately run and suspend it. Hypercard is also a state-based system, and its metaphor for a state is the "card." Consequently, the user interface is called a "stack." State transitions are done with "links" between cards. Usually links are invoked by "buttons," which are simply icons that execute an action when clicked on by the mouse. A useful feature of Hypercard is that link buttons can be specified by example. For more complex actions a general purpose language is provided called Hypertalk, which allows a programmer to code "scripts."

Hypercard events are passed to the current card. If that card does not have a script associated with the event, then it is passed to the background card. If the background card can't handle the event it goes to the stack.

Grouping in Hypercard, therefore, is done by putting several cards on the same background. Only one background is active at a time, so a card with a background cannot be the background to an other card. This means that groupings cannot be nested more than one level deep.

Concurrency is not really possible in Hypercard either. Several buttons can be placed on a card at the same time, but only one of them will respond to a

given event. It is not possible for several cards sharing the same background to be active at the same time.

Hypercard is fast, intuitive and simple to use, and it includes a general forms package, with fast search and retrieve capabilities. The system is inherently based on a button based style of interaction. This decision was probably made in order to keep the system simple and consistent with Mac users' expectations, but Hypercard's dialog model does not seem to work well for more general user interfaces. Allowing concurrent responses to non window based events is difficult, and dialog hierarchies deeper than a few levels can not be specified.

Hypercard also suffers from the lack of a top-level visual representation of dialog structure, making it difficult to navigate through large stacks.

2.3 Statecharts

Statecharts [Hare87a] are a graphical notation that extends conventional state transition diagrams to overcome their drawbacks while preserving all of their benefits. Although not used by any UIMSs, the notation has been used to specify state-of-the-art avionics systems at the Israel Aircraft Industries, and it is used by a software engineering product (STATEMATE by i-Logix, Inc.) for designing complex realtime software and hardware systems. The two most important additions that Statecharts bring to state diagrams are grouping and concurrency.

2.3.1 Grouping

If several states have identical transitions triggered by the same event, then those states can be grouped in Statecharts, and a single transition is specified from the group instead.

In Figure 2.4, there is a switch that toggles between states A and B. When event g occurs, power is turned off to the switch and it becomes inactive, whether it was in state A or B. Event h returns power to the switch. In the statecharts representation, State D is an XorGrouper containing A and B. This means that being in state D is the same as being either in state A or B, but not both. The little arrow pointing to B shows the default state (off) that is active when D is entered. From a bottom-up point of view, D can be thought of as an abstraction of A and B, because it captures a property that both states

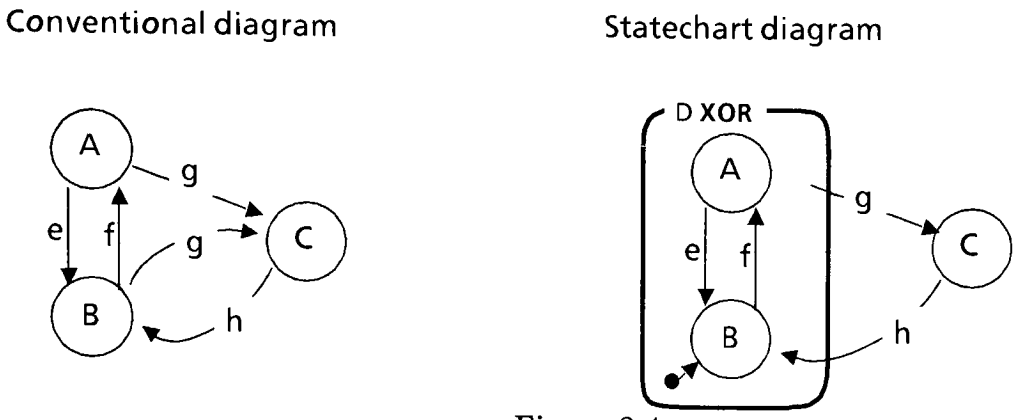


Figure 2.4
Grouping in statecharts

have in common. From the top-down point of view, A and B can be thought of as a refinement of D. This capability of grouping states, when applied hierarchically to large collections of states, can eliminate a great deal of explicit state transitions and make state diagrams much easier to understand. This approach is simpler and more visually intuitive than the "subconversation" approach used by Wasserman and Jacobs.

2.3.2 Concurrency

When several small state machines are active at the same time, the total number of possible states is the product of the number of states in each machine. A Statechart representation of this case uses AndGrouper states, making it grow linearly while conventional state representations grow exponentially.

Figure 2.5 illustrates two toggle switches, both of which are active at the same time. When event p occurs, both become inactive. In Statecharts, State Y is an AndGrouper. Being in state Y is the the same as being in both F and G at the same time, where both F and G are XorGroupers. Compared to Jacob's means of specifying concurrency, this approach is much more visually intuitive, and it works as a simple extension of the grouping principle described above.

2.3.3 Broadcasting and conditional transitions

Statecharts can be made even more flexible flexible by adding the concepts of *broadcasting* and *conditional transitions*. These extensions are sometimes

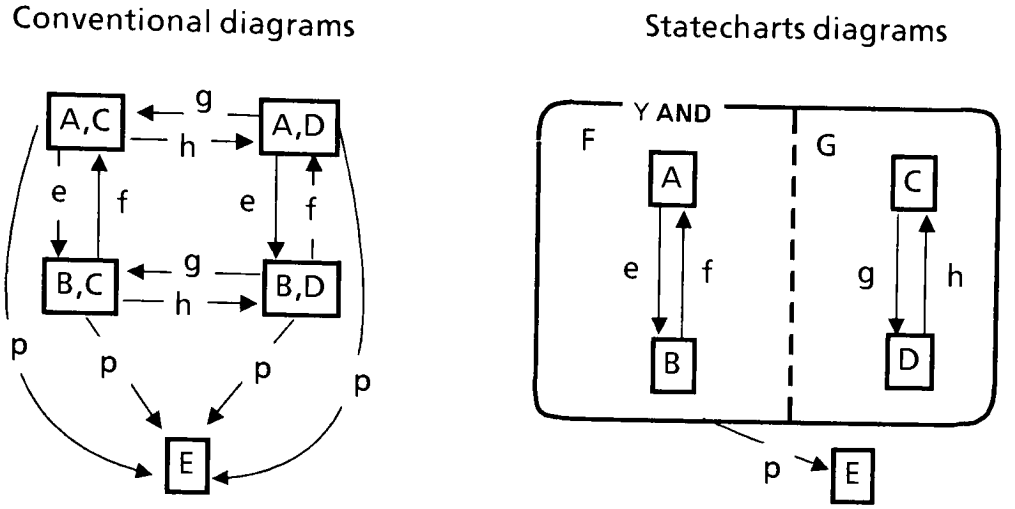


Figure 2.5
Concurrency in Statecharts

used in the context of conventional state diagrams, and they provide a good way to specify what are sometimes referred to as "implications and constraints."

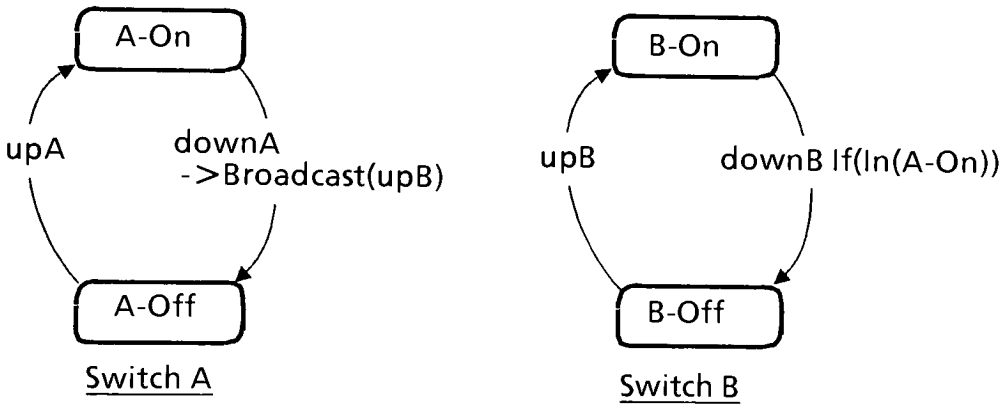


Figure2.6
State Transition Diagrams

An *implication* is when an action in the user interface has an additional effect in another, separate part of the interface. This is represented by tying an additional action to a state transition. For example, in Figure 2.6, assume that the event downA turns off switch A and that this must automatically turn on switch B if it is off. The transition for turning off A must execute the action to broadcast event upB. When an event is broadcast, it is as if the event

had actually occurred, so any transitions that depend on that event will be executed.

A *constraint* is when an action in the user interface can only occur under certain conditions. This can be specified with state transition diagrams by using conditional transitions. Let us say that in the above example switch B is constrained to stay on as long as A is off. A condition would be added to the transition from B-On to B-Off. Unless the value of In(A-On) is true, then state B-Off can not be entered.

The ideal UIMS should be based on a representation that effectively represents the cognitive structure of user interface dialogs. In other words, it should accurately reflect the constructs that designers and users keep in mind when thinking about the system. This paper is based on the idea that statecharts is a good representation of this cognitive structure. Statemaster implements dialogs in terms of statecharts in the hopes that this representation will make user interfaces easier to create, understand and modify.

2.4 Object-oriented programming and C++

Statemaster is also based ideas from object-oriented programming languages. These languages provide a powerful programming paradigm that encourages modularity and the reuse of existing code. They are widely used in programming expert systems, and they facilitate the development of effective graphic user interfaces. Although object-oriented languages are primarily programmers' tools, their features benefit non-programmer designers as well. Both Hypercard and Trillium are object-oriented, and Statemaster is implemented in C++, a relatively new object-oriented programming language.

C++ is a general purpose language designed by Bjarne Stroustrup at AT&T Bell Laboratories [Stro86]. It is a superset of C that retains the efficiency of C while providing strong type-checking and facilities for object-oriented programming. Programs in C++ tend to be shorter, easier to understand, and easier to maintain than equivalent programs in C. Some of the features of C++ include classes, constructors, and inheritance.

2.4.1 Classes

Classes provide data abstraction, which is the separation of a data object's representation from how it is used. A familiar example of data abstraction is how the department of motor vehicles interfaces to the outside world. When a car owner needs to find out some information about his driver's license, he does not walk into the DMV, open up a filing cabinet , find his record of convictions and copy it into his notebook. Instead, he must fill out a form, or call a certain phone number. If the DMV becomes computerized or changes their computers, the telephone number and form stay the same. No one besides DMV employees need to know about the change.

The C built-in type *float* works in the same way. All a programmer needs to know about a *float* object is how to use it. Two floats can be added together without concern for how they are represented internally .

With C++, the programmer can create his own classes to provide data abstraction. To do this he specifies the internal representation for a class and the operations that can be performed on it. The class then becomes a programmer defined type, and any other part of the program can use it in the same way that it uses a built in type -- without concern for how it is implemented. The C++ translator signals a syntax error when an instance of a class is not used properly. If the implementation of the class needs to change then all changes can be limited to the class code itself. The rest of the system can remain unchanged as long as the interfaces are maintained.

2.4.2 Constructors

Constructors provide guaranteed initialization of class objects. Declaring an object guarantees that its constructor will be called before it is ever used. When an object goes out of scope, it is implicitly destroyed with a call to its destructor. Constructors and destructors save the programmer from having to explicitly allocate memory, set initial values, and clean up afterwards. Forgetting to to do these things at the right time is often the cause of terribly annoying bugs.

2.4.3 Inheritance

Inheritance is provided in C++ through derived classes. A derived class is a specialization of the base class and has all the functions and data of its parent.

The derived class can redefine some of its parent's functions, and it can add its own data structures. These redefined functions are called "virtual functions." A virtual function implicitly checks the object type and determines at *runtime* which redefined function must be called.

Let us say for example, that a programmer needed several kinds of classes that he can display: Button, Folder, Clock, and Window. With inheritance, the programmer can implement a different *Display()* function for each class, and let them all inherit from the base class DisplayableObject. If a list of DisplayableObjects needs to be put up on the screen, the programmer can simply tell each object to *Display()* itself and let it figure out specifically which C function needs to be executed. Similarly, there may be several kinds of Buttons, each of which looks slightly different but has some properties in common with regular Buttons. Inheritance can be used to define a SpecializedButton class. This class can reuse the code from the regular Button, and specialize it for its own use only where needed.

Chapter 3 : Description of Statemaster

This chapter begins with a brief description of the goals and features of Statemaster, then it describes how to use Statemaster and describes its implementation.

3.1 Goals and Features

Figure 3.1 illustrates the three primary goals that Statemaster attempts to achieve. Arrows point to the features that respond to those goals.

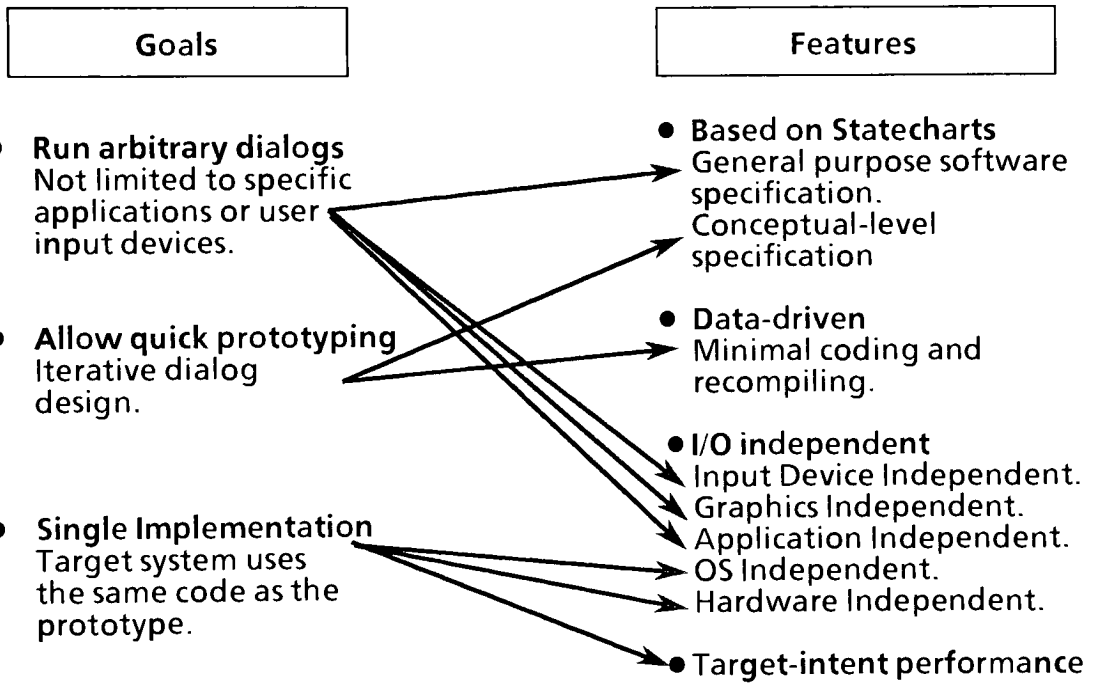


Figure 3.1
Goals and Features of Statemaster

3.1.1 Run arbitrary dialogs

The system must be able to implement a very wide range of dialogs. Some examples include:

- Simple command-line text-based dialogs used by trained copier service representatives.
- Cascade-based graphical dialogs that respond to hard button presses by highlighting and dehighlighting certain parts of the screen.

- Touch-based dialogs with dozens of simultaneously active screen segments that individually respond to inputs.
- Highly interactive image editing dialogs that accept input from multiple devices.
- Mouse and menu-based dialogs running on graphic workstations that use windows and pop-up menus.

The wide range of dialogs to be supported requires a specification technique that is general and able to describe large and complex systems. Statecharts have been shown to be capable of describing complex avionics systems, and they are used by STATEMATE, a commercial software engineering product for designing realtime systems.

The other feature that enables Statemaster to be used for arbitrary dialogs is its I/O independence. It does not assume any specific means of displaying graphics (*e.g.* Xwindows or News); it does not require input from any specific devices (*e.g.* touch or hard buttons), and it does not assume any specific application (*e.g.* copier or image editing). All input and output to Statemaster is done through a well defined, layered communications protocol similar to the ISO reference model [Tane81]. To interface Statemaster with an I/O device or application software, one chooses the most appropriate level of communication and implements a driver for that level. The rest of the code remains unchanged.

3.1.2 Allow quick prototyping

Any general purpose programming language like C or C++ would be adequate for implementing dialogs if generality were the only goal. In order to allow quick prototyping, however, the specification must be at a conceptual level, close to the way in which designers express themselves. Statecharts are much closer to this level than any general purpose programming language. Furthermore, Statecharts naturally creates building blocks through its grouping mechanism. Using these building blocks, the designer can specify dialogs at a high level or at a low-level, then group and link them together.

Statemaster can be used as a prototyping tool only by someone with programming skills. This person might be called a "UI Engineer," because of the similarity between this job and that of a "Knowledge Engineer." When

implementing an expert system, the knowledge engineer must implement the domain expert's knowledge using an expert system building tool. Similarly, the UI Engineer must implement the HF designer's dialog using Statemaster, and he or she should have some understanding of both the domain (UI design in this case), and the implementation.

3.1.3 Single Implementation

Many UIMSs generate a runtime prototype that cannot be used as the target implementation. The two separate runtime implementations are necessary because the UIMS often depends on features of its native environment that cannot be easily implemented on cost constrained target hardware. The two systems are often programmed in different languages and have very different software architectures even though they are superficially alike. They have different bugs and different performance. The target implementation must go through its own development cycle, which is partially redundant to that of the prototype.

Statemaster can be used both for prototyping and for the target implementation. The single implementation saves development time and makes the prototype more accurate in terms of behavior and response time. The primary means by which it accomplishes this goal is through hardware independence. It is implemented in C++, which can be ported to any processor with a C compiler and provides the low-level control and efficiency of C. Statemaster does not depend on special features of any particular operating system (*e.g.* UNIX or MSDOS), and it does not depend on virtual memory or garbage collection. As a prototype, it is able to run on a PC or workstation long before the target hardware has been built or even designed. Dialogs can be implemented and refined on the prototyping hardware, and then the *same* code and data are ported to the target.

3.2 How to use Statecharts and Statemaster

The statecharts that Statemaster uses to represent user interface dialogs are based on David Harel's statecharts, but they have been adapted to more easily specify user interface dialogs. The statecharting conventions described in this section have been used successfully by UI engineers to specify working Statemaster dialogs.

The three fundamental components of statecharts are *States*, *Events*, and *Actions*. *Behaviors* are made up of Event - Action pairs.

3.2.1 States

Each state represents a specific configuration of the UI at a point in time. The entire UI dialog is represented by a state machine that is composed of a collection of states, only some of which are active at any given moment. As the user and the application interact with the dialog, the currently active states change.

It is important to distinguish States from Bitmaps. Many states have specific bitmaps associated with them, and whenever one of these states is active, its bitmap appears on the screen. A state and its bitmap are not the same, however. A single state may use several different bitmaps, and the system can change states without changing bitmaps.

A State is represented in Statecharts by a box with rounded corners, as illustrated in Figure3.2. The name of the state appears inside the box or on the box's upper edge. The state name is always in lower case. If the name is composed of several words, then they are joined by underscores.

Figure 3.2
Simple states

In Statecharts there are three different kinds of states: simple states, AndGrouper states, and XorGrouper states. AndGroupers and XorGroupers are states that have children; simple states do not.

3.2.2 XorGrouper States

When an XorGrouper state is active, one and only one of its children is active. The Xor relationship determines that if one child state becomes active, then all other children become inactive. XorGroupers look like simple states, except that they have child states inside, and they have **XOR** following their names.

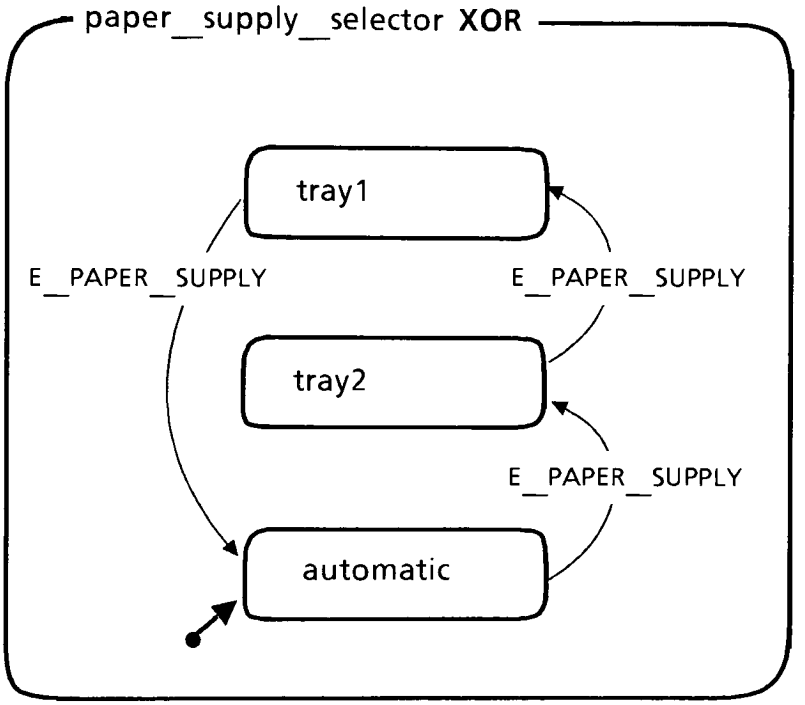


Figure 3.3
An XorGrouper state

Figure 3.3 shows a statechart for the paper supply selector for a copier machine. The paper_supply_selector state is an XorGrouper with three child states. That means the selector can be in one of three positions, but it cannot be in more than one position at a time. The default arrow points to automatic, meaning that when paper_supply_selector is entered for the first time the automatic state will be entered instead of one of the other two child states.

3.2.3 Events

Events are usually inputs to Statemaster. An Event can be caused by the user, the application, or statemaster itself. The paper_supply_selector in Figure 3.3 responds to only one event: E_PAPER_SUPPLY. It could be generated when the user presses a button on the control panel labeled "Paper Supply" to select a desired paper tray. Events are always written in upper case with underscores, and they start with E_. In the example above, each event is tied to a transition, represented by an arrow. The arrow points to the state that will be entered when the Event occurs. This statechart shows that whenever the E_PAPER_SUPPLY Event occurs, the paper_supply_selector changes its currently active child state.

3.2.4 AndGrouper States

Like XorGrouper states, AndGrouper states have children, but when an AndGrouper is active, *all* of its children are also active. When any child of the AndGrouper is entered, the AndGrouper becomes active and so do all of its children. AndGroupers are used to specify concurrent states, or when two or more states are active at the same time. There are in fact several other children of basic_features, but they are left out for this example.

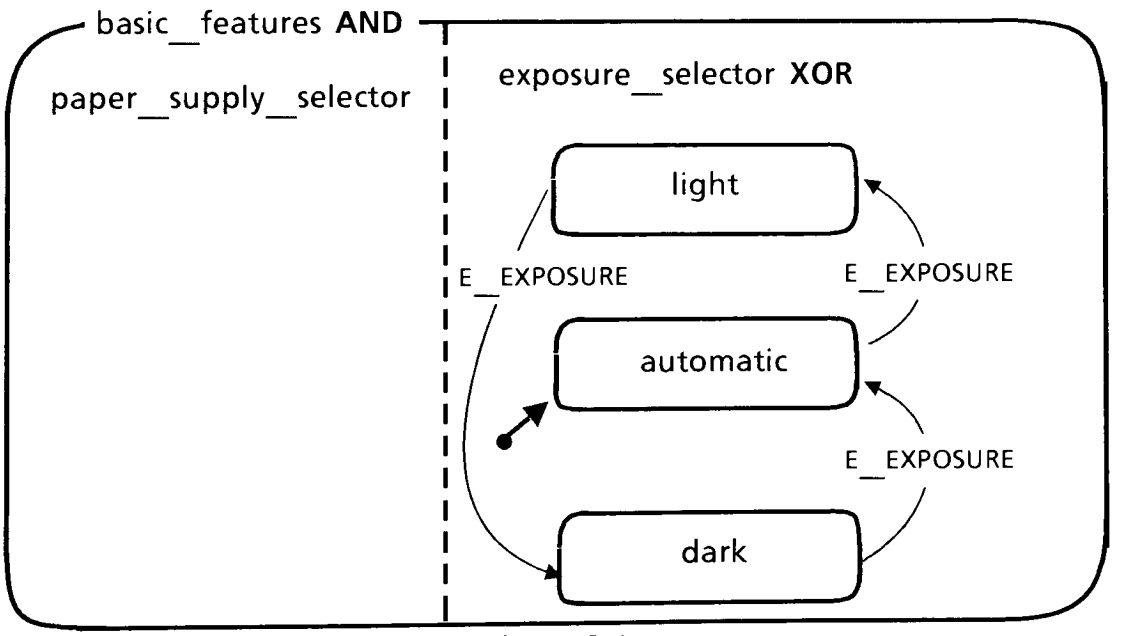


Figure 3.4
An AndGrouper state

Figure 3.4 specifies the exposure selector. This selector looks just like the paper_supply_selector, but it has differently named child states and responds to a different Event. In order to show that the paper_supply_selector and the exposure_selector are both active at the same time they are put together inside an AndGrouper named basic_features.

Notice that in Figure 3.4, paper_supply_selector is not specified in detail, and it doesn't even have XOR next to its name. This is because paper_supply_selector is completely specified in another statechart (Figure 3.3), although both states could have been specified in full detail on the same chart. In general, it is best to make each statechart very simple and not put too many child states in one chart. This makes the charts easier to follow and easier to update with design changes.

Notice also that both paper_supply_selector and exposure_selector have a child state named "automatic." These states have the same names even though they refer to different states. There is not a conflict, however, because although the states have the same *simple* names, they have different parent states. Their *distinguishing* names include the parent and child names. They are exposure_selector$automatic and paper_supply_selector$automatic. These names are implicit in the statecharts, so it is not necessary to use them as labels. There are some situations, however, when the simple names are ambiguous. For example, if we were to specify the statecharts that separately describe the details of exposure_selector$automatic and paper_supply_selector$automatic, we would have to use the distinguishing names to label the outermost state for automatic in both charts. The convention followed for these statecharts is that the simple name is used whenever it is unambiguous either because of its name or its context in the statechart. If two or more states have the same simple name, then the parents' names are prepended to the simple names joined by a $. The distinguishing names of states go only as far back up their ancestry as necessary to resolve the ambiguities.

3.2.5 Actions

Actions are usually outputs from Statemaster. For example, one typical Action is to display a bitmap on the screen. Another might be to turn a light

on or off. Figure 3.4 could be implemented by displaying bitmaps on a screen, or it could be implemented by turning LEDs on and off on a hard control panel. The statecharts (and Statemaster) are device independent, so only the Actions determine what exactly are the system outputs. Any state can optionally have entry or exit Actions. The entry actions are executed whenever that state is entered. The exit actions are executed when the state is exited. The convention is that all Actions begin with an upper case A and end with a parenthesized list of arguments. If the name is composed of several words, they are joined together with each word beginning with a capital letter *e.g.* ADisplay(BUTTON1), or ATurnOn(LIGHT1). A state with entry or exit actions has a thin line drawn horizontally across the state. Entry actions are listed above the line, and exit actions are listed below the line.

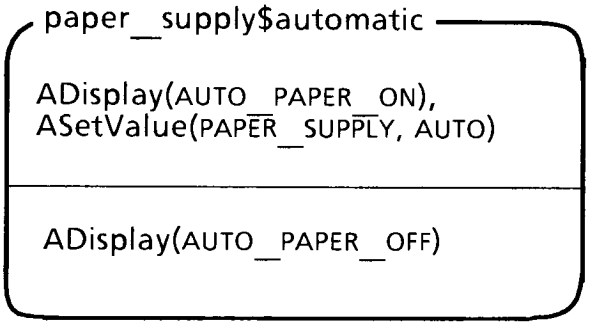


Figure 3.5
Entry and Exit Actions

In Figure 3.5, the state paper_supply$automatic has two entry actions and one exit action. When paper_supply$automatic is entered, the two Actions are executed, displaying the AUTO_PAPER_ON bitmap and setting the PAPER_SUPPLY GlobalValue equal to AUTO. When the state is exited, it displays the AUTO_PAPER_OFF bitmap.

3.2.6 Behaviors

In addition to entry and exit actions, every state can have behaviors. A behavior is simply an Event-Action pair that specifies an Action to be executed when a specific event occurs. The transitions in Figure 3.4 are examples of behaviors. A behavior of the exposure_selector$automatic is that if the Event E_PAPER_SUPPLY occurs, then ATransition(light) will be executed. ATransition is a special kind of behavior because it can be

graphically represented by an arrow. Most other behaviors are specified explicitly.

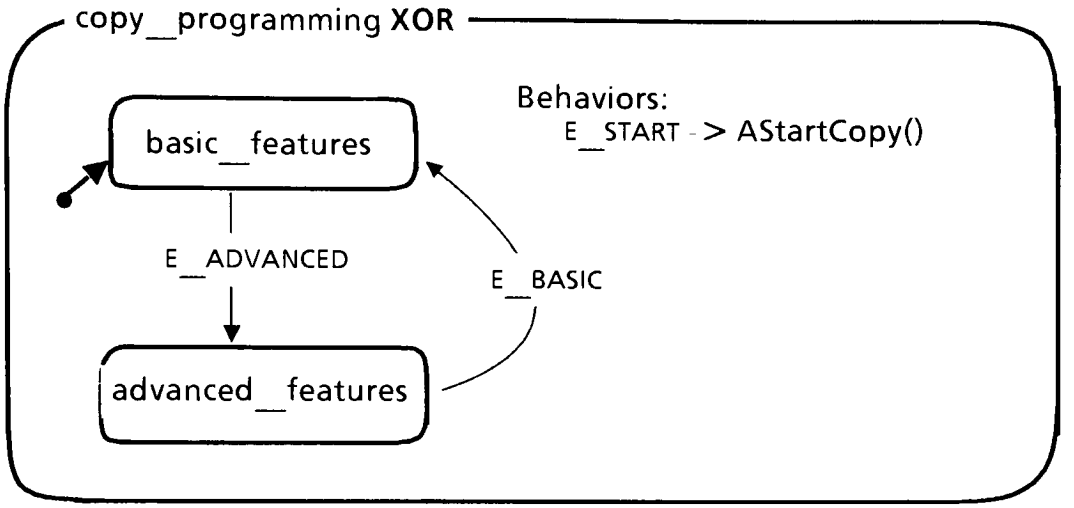


Figure 3.6

For example, in Figure 3.6, copy_programming has two modes: basic_features and advanced features. The user can only be in one at a time. At any time while the system is in copy_programming, if the user hits the Start button, then the UI must execute the action AStartCopy(). Any other behaviors for this state can be added to the explicit list the same way as the E_START -> AStartCopy() behavior. Note that because this behavior is for the copy_programming grouper, it is still valid no matter what child states the system is in. It could be anywhere inside of basic_features or added_features, but whenever the E_START Event occurs, then AStartCopy() will be executed.

Let us suppose that in basic_features the user makes some changes to paper_supply, leaving it in tray1. Then the user hits a button that generates the E_ADVANCED Event. The system exits basic_features and enters advanced_features. What happens when the user goes back to basic_features? Will Statemaster enter automatic, which is the default child, or will it enter tray1? The answer is tray1. When Statemaster enters an XorGrouper for the *first* time, it will enter the default child. If the state is exited and reentered after that, Statemaster enters the last currently active

child. Consequently, there must be a way to reset an XorGrouper to make the default child active again. This is done by executing the AReset(state_name) Action, specifying the state that should be reset.

Notice also that it is possible to specify non-deterministic behaviors with statecharts. For example, it is possible to have the same event cause a transition to two different states. Statemaster is of course a deterministic system, so in these cases it executes both transitions, causing the system to go into one state and then immediately into the other. Whichever transition is executed last will determine the state the system ends up in.

3.3 Implementation of Statemaster

Statemaster includes all of the components described in section 2.1 [*Generic structure of a UIMS*]. Figure 3.7 highlights the components that are unique to Statemaster.

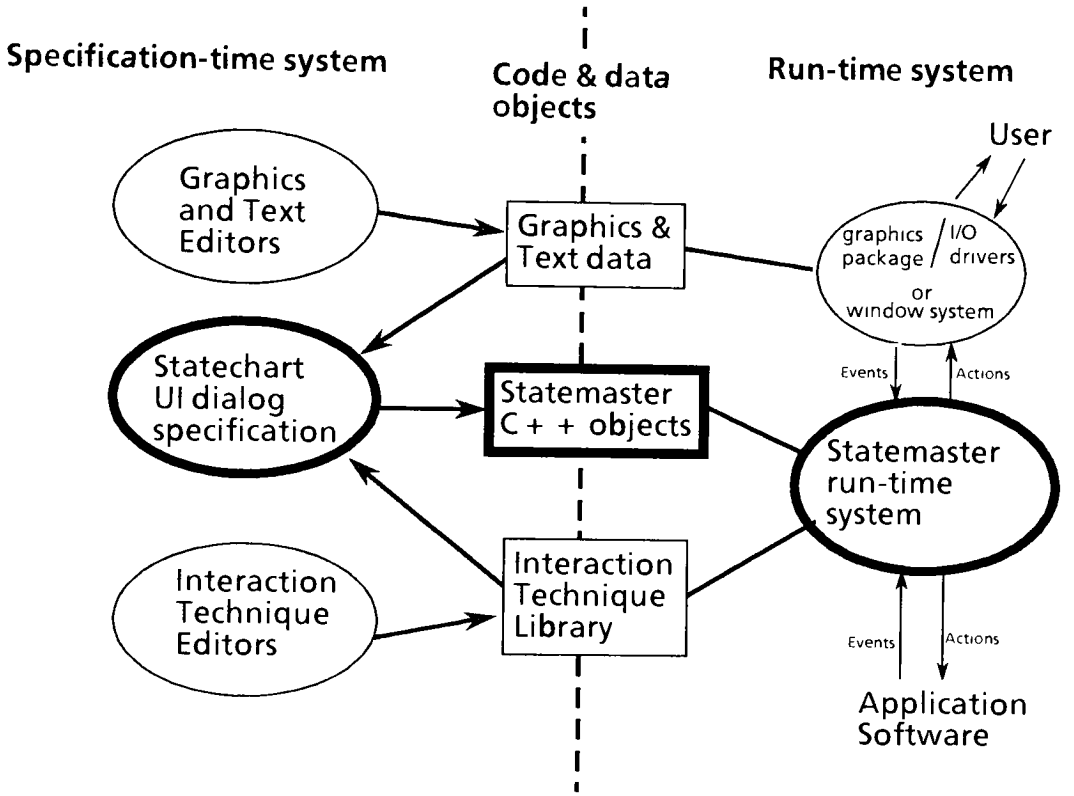


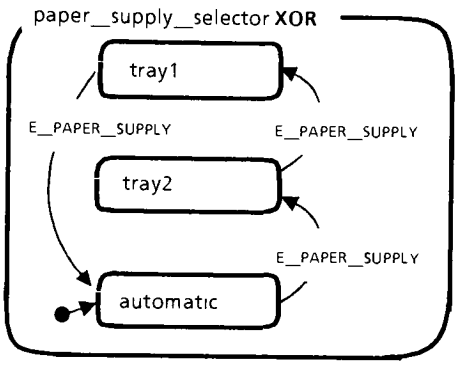
Figure 3.7
Unique components of the Statemaster UIMS

3.3.1 The specification-time system

Dialogs are specified by drawing Statecharts with an object-oriented editor that is provided as a standard part of Viewpoint, running on Xerox workstations. To make the task easier, a library of objects has been predefined so that the specifier can copy states, behaviors and groupers from a basic document then piece them together and modify them as needed.

The translation from the graphical statechart document to the C++ data structures is not yet automated, so any object-oriented editor (or even pencil and paper) could be used. The translation, although a manual process, is very straightforward as illustrated in Figure 3.8.

This graphical statechart specification is translated into the following text specification:




```
tray1 = State("tray1",                  /* name of state just for debugging */
    0,                                  /* Null entry_action                */
    0,                                  /* Null exit_action                 */
    BehaviorTable(1,                    /* 1 element in the BehaviorTable   */
        Behavior(E_PAPER_SUPPLY,        /* Event is E_PAPER_SUPPLY          */
            ATransition(automatic)      /* Action is ATransition to automatic */
            )
        )
    );

tray2 =      State( ...                 /* as above */
automatic = State(...                   /* as above */

paper_supply_selector = XOR(
    "paper_supply_selector",            /* name of state just for debugging */
    0,                                  /* Null entry_action                */
    0,                                  /* Null exit_action                 */
    0,                                  /* Null BehaviorTable               */
    MEMBERS(3,                          /*This state is a grouper with 3    */
        automatic,                      /* members. automatic is listed first */
        tray1,                          /* because it is the default.       */
        tray2
        )
    );
```

Figure 3.8
Translation of basic_features statechart into C++ constructors

The dialog specifier types calls to C++ constructors into a file, which then gets compiled. When this function runs, it constructs all the C++ objects of

the dialog and the runtime system uses these objects to actually run the dialog.

3.3.2 Run-time system

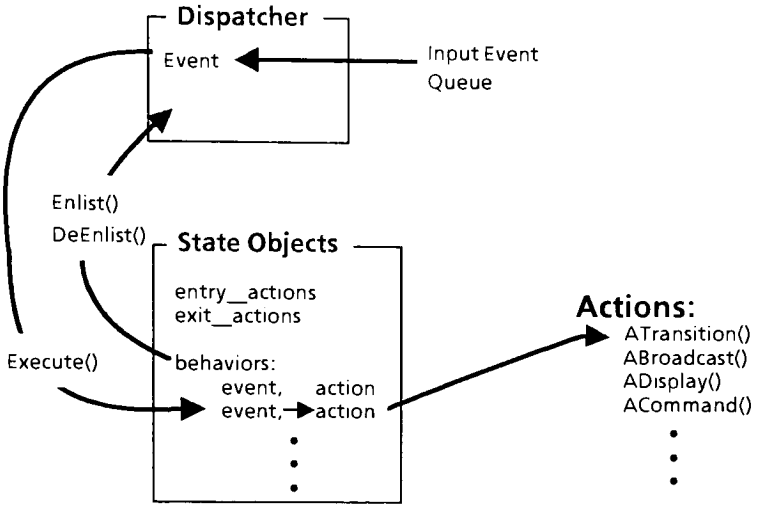


Figure 3.9
Statemaster run-time sytem architecture

The Statemaster run-time system architecture is summarized by Figure 3.9 and is controlled by the dispatcher, which keeps track of active states and the events that they are interested in. As states become active, they Enlist their behaviors with the dispatcher. When they become inactive, they DeEnlist their behaviors. Any event that comes off the queue is dispatched by executing all the actions that are currently enlisted to that event. The entire system is described in detail in the following sections.

3.3.2.1 State objects

States are C++ objects. This means that they have data and member functions that operate on that data. Every instance of a state has its own separate values for its data, but all instances share the same code. As described above, there are three kinds of states: Basic states, AndGrouper states, and XorGrouper states. The base class State implements basic states. AndGrouper and XorGrouper inherit from State. The State member functions are implemented differently for AndGrouper and XorGrouper than for the base State class.

All states have an entry_action, exit_action, and behaviors. AndGrouper States and XorGrouper states also have children.

- entry_action

  The entry_action is executed every time the State is entered. See section 3.3.2.4 [*Actions*] for the various types of actions that are available.
- exit_action

  The exit_action is executed every time the State is exited.
- behaviors

  Behaviors points to a BehaviorTable, which consists of Event Action pairs. These behaviors get Enlisted with the dispatcher when the state becomes active. Then, when one of these events occurs, the dispatcher executes the corresponding action.
- children

  Children is an array of pointers to states. Only AndGrouper and XorGrouper States have children in addition to the other data fields.

State member functions include DoEntry(), DoExit(), Enlist(), DeEnlist() and Reset().

- DoEntry()

  executes the entry_action and then recursively calls itself for active child states.
- DoExit()

  recursively calls itself for all currently active child states, then it executes the exit_action.
- Enlist()

  enlists all the state's behaviors with the dispatcher and then recursively calls itself for active child states.
- DeEnlist()

  recursively calls itself for all currently active child states, then it DeEnlists its own behaviors from the dispatcher.
- Reset()

  Resets an XorGrouper to its default child after exiting the currently active children. An AndGrouper recursively Resets all its children, and a simple state does nothing.

3.3.2.2 Transitions

Executing the Transition() action activates a new destination state after the following steps have been performed :

1. DoExit() all currently active states whose scope the destination state is not within. Because of statechart's hierarchical structure, entering one state may imply entering and/or exiting several other states. Statemaster finds the lowest active ancestor of the destination state to determine which states to exit. For example, in Figure 3.10, the transition from K to J affects only those two states, but the transition from J to L affects several: B, D, and E must be exited, while C, F, G and H must be entered in addition to L. One approach to the situation is to require the designer (in the entry and exit actions) to explicitly specify which states must be exited and entered. Instead, Statemaster automatically determines the least number of state changes which must be made in order to execute a transition to the destination state.

The way in which this is done is by internally representing states as a tree. The lowest common ancestor (LCA) of the two states must be determined before a state can execute a transition to an other state. Entering the destination state first requires exiting the current child of the LCA, then all parent states of the destination state are entered in decending order.

2. DoEntry() for the destination states. Executing the entry action of the destination state is done before anything else because it is important to give the user feedback before any internal processing. The entry actions usually include output to the user.

3. Enlist() the destination state's behaviors with the dispatcher so that it knows what behaviors are active while this state is active.

4. DoEntry() for the appropriate child states. If the destination state is an XorGrouper, the current child is entered. If the state is an AndGrouper then all its children are entered. This recursive call is done last because entry actions are best executed from the top most ancestor down.

Exiting a state requires exiting its currently active children states, executing its exit actions, and DeEnlisting its behaviors.

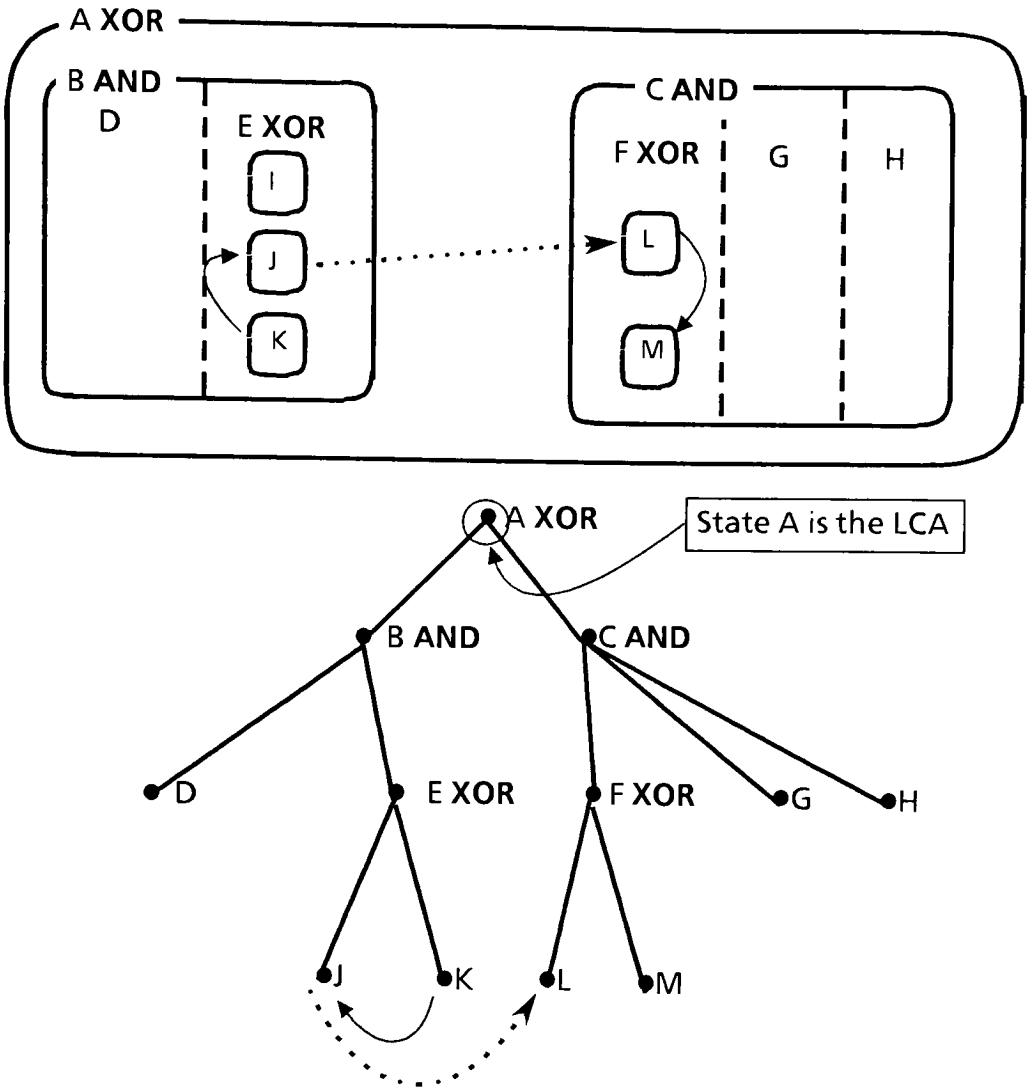


Figure 3.10
State transtion implications

3.3.2.3 Events

All inputs to the system are treated uniformly by Statemaster as incoming Events, and they are read off of the InputEventQueue one at a time. An event can be generated by an input to a keyboard, touch screen, serial port, parallel port, mouse or by an operating system routine. Events are not only generated by physical devices but also by the application software and by Statemaster itself through the Broadcast() action. Any input device that is connected to the system must generate an Event in order for Statemaster to be able to respond to it.

3.3.2.4 Actions

Statemaster executes Actions in response to events. Once an Action has been constructed, the only thing that can be done with it is to *Execute()* it. Actions are implemented in Statemaster by a C++ base class named Action that has only one member function: Execute(). All actions inherit from the Action class so C++ can dynamically bind specific action functions to the dispatcher's call to Action::Execute().

By convention, all action names begin with an upper case A. Some of the Actions provided with the initial implementation of Statemaster include the following:

- ATransition(State)

  will executed a transition to the named state as described in section 3.2.2.2.
- ABroadcast(Event)

  generates the event as though it had been generated by an external device.
- ADisplay(DisplayId)

  displays a bitmap that corresponding to the DisplayId.
- AIf(Condition, Action)

  executes the action only if the condition evalueates to true.
- AIfElse(Condition, Action, Action)

  If the condtion evaluates to true, then the first action is executed, otherwise the second action is executed.

Notice that these Actions have arguments even though the runtime system can only Execute an action without passing arguments. That is because Actions are C++ objects, and each instance of an Action stores its own argument values which are set when the Action is constructed at specification-time. The Execute() function then uses these parameters at run-time

3.3.2.5 Behaviors

A Behavior is simply an Event-Action Pair. Every State has a BehaviorTable, and each behavior consists of an Event and an ActionPointerDoubleLink. A DoubleLink consists of three pointers: previous, next and element. As illustrated by Figure 3.11, an ActionPointerDoubleLink stores an ActionPointer as its element.

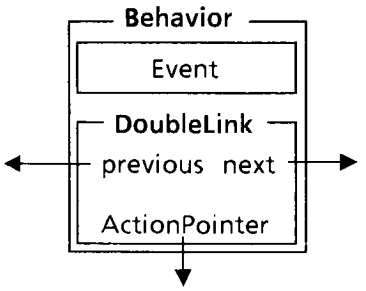


Figure 3.11
ActionPointerDoubleLink

3.3.2.6 The dispatcher

When a state is entered, it Enlists its behaviors with the dispatcher. The dispatcher has an array of ActionLists with one ActionList for each possible event. When a behavior is Enlisted, its action is added to the ActionList for that particular event. When the behavior is DeEnlisted that action is removed from the list. When an event is generated, the dispatcher executes every action in the ActionList at the position corresponding to that event.

One way to implement the dispatch table would be to have an array of ActionPointer arrays. This would make dispatching an event very quick because the dispatcher would just have to step through the ActionPointer array to execute the actions, but adding and removing action pointers from the array would involve searching. In addition there would be the problem of allocating the correct amount of storage for each array.

An other way to implement the dispatch table would be to have an array of singly-linked lists. Stepping through the list to execute actions is still quick, and no storage would be wasted because the lists would grow and shrink as Behaviors were Enlisted and DeEnlisted. The only drawback to this method is that space would continually have to be allocated and deallocated at runtime.

The third alternative is to use an array of doubly-linked lists. Each behavior already has allocated a DoubleLink for its ActionPointer. When a behavior is enlisted, its ActionPointerDoubleLink is pushed on the front of the ActionList, and when the behavior is DeEnlisted, the link is removed from the list by making its two neighbors point to each other. With this implementation, Enlisting requires setting only four pointers, DeEnlisting

requires setting two. This alternative was the one chosen for the first version of Statemaster. Actions are pushed to the front of the ActionList because when several actions are enlisted to the same event, it is usually better to first execute the ones that were enlisted last.

3.3.2.7 The main loop

The main loop of Statemaster is the dispatcher's Run() function. It takes events off the InputEventQueue and dispatches them by executing their associated actions. Transition Actions are treated differently from other actions because they are not executed immediately. If a set of actions must be executed in response to a certain event and one of those actions is a transition, then the transition must be executed last. If a transition were immediately executed at the time when it was encountered in the ActionList, then the following actions might no longer be meaningful in the context of the new state that was transitioned to. ATransition has a member function called DoTransition() in addition to Execute(). Execute() puts the transition on the pending transitions queue. After all the other actions for that event have been executed, then DoTransition() is called to actually do the transition for each transition in the queue.

The dispatch loop uses an additional queue for pending events. This queue is used for events that are broadcast by the ABroadcast action. Initially it might seem that to broadcast an event it would be sufficient to push that event on the front of the input event queue, but in fact that would be incorrect. If a list of actions to be executed includes both broadcast actions and transitions, then Statemaster must make sure to execute the actions enlisted to the events that are broadcast *before* it executes the transitions. Again, this is because those events and actions might not be meaningful in the context of the destination state. Events that are broadcast are therefore put on a separate queue (the pending_events_queue) and dispatched in the order they are broadcast after actions for the current event have been executed. After the pending events queue is empty, then the transitions are done and the input event queue is checked again for more events.

3.3.2.8 The input event queue

Events are pushed on the rear of the input event queue by the function generate_event(). All the input drivers call this function to pass events to

Statemaster. Statemaster is currently implemented on three different operating systems, and some implementations are interrupt driven while others are polled. The input event queue allows the implementation of the dispatcher to be nearly the same in both cases. In the polled implementation poll_input_functions() is called once in every pass through the dispatch loop. This function makes sure that all device drivers call generate_event() if they have any events to give Statemaster and it returns otherwise. In the interrupt driven implementation, each input driver executes asynchronously from Statemaster and blocks until it receives an input at which point it calls generate_event().

3.3.2.9 Soft Enlisting - an alternative implementation

The implementations described above for Enlisting and DeEnlisting of behaviors are all variations of what might be called "hard Enlisting." Hard enlisting is when the dispatcher has an array as large as the number of possible events in the system and this array gets updated as the state of the system changes.

An other approach, "soft Enlisting," would not enlist behaviors with the dispatcher. Rather, States would enlist their BehaviorTables with the dispatcher instead of individual behaviors. A BehaviorTable is a set of Event-Action pairs. For soft enlisting, these tables would be implemented as Hash arrays so that an Action could be quickly found in a table based on an Event key. When an event occurs, the dispatcher would hash all the BehaviorTables that are currently enlisted and execute any actions that were in these tables.

This approach has three significant advantages over the current Hard Enlisted implementation:

1) Behaviors would take less storage because a DoubleLink would not have to be stored for each behavior.

2) There would be no compiled-in limit to the number of events that are possible in the system.

3) Enlisting and DeEnlisting a State would be much faster.

The disadvantage of this approach is that Executing Actions in response to an event would be slower and would depend on the number of states that are currently enlisted.

The current implementation of Statemaster was driven by a requirement to minimize response time. The time between a user input and a visible response had to be as small as possible, so HardEnlisting was chosen.

Another performance metric besides response time is the time it takes between a user input and the when the system can accept the next input. If we use this metric, then SoftEnlisting is faster than HardEnlisting in the cases of transitions between states with large BehaviorTables. With SoftEnlisting, only the tables are enlisted with the dispatcher instead of each individual behavior. This is especially a savings if the user frequently passes through large states without doing much within them. Nevertheless, in these cases response time would still not improve because during a transition, Enlisting and DeEnlisting is only done *after* the entry and exit actions are executed.

The performance and space tradeoffs between hard and soft enlisting are dependent on the specific dialog and usage patterns. A future version of Statemaster may provide both, as well as heuristics and analysis tools to determine which method is best in each case. It might be that high level states with large behavior tables should be SoftEnlisted, especially if the user does not switch in and out of them frequently, while low level states with small behavior tables should use HardEnlisting.

3.3.3 Communication with application software

This section describes how Statemaster communicates with the application software (ASW) at runtime. Some of the goals that this approach attempts to satisfy include the following:

- Flexibility to allow the implementors to optimize the communication according to application and hardware characteristics. For example, in some situations implementors will choose to send frequent, small messages, and in other situations they will want larger, less frequent messages.
- The architecture should be usable on a variety of hardware and operating system configurations allowing maximum reuse of code. For example, the architecture should be usable in each of the following situations:

  The UI and ASW reside in a single process on a single CPU.

  The UI and ASW reside in separate processes under a multitasking OS running on single CPU.

  The UI and ASW reside on separate CPUs communicating through shared memory or a cable.
- The UI should be able to respond immediately with visual feedback to an ASW event, but it should also be able to set values which affect currently inactive states.
- The approach should allow default UI values to be used, changed by the user, and stored for later use.

As described in previous sections, Statemaster interfaces to the outside world through Events and Actions. Communication with the ASW is no exception: Inputs from the ASW must generate Events, and outputs to the ASW must come from executing Actions.

Figure 3.12 illustrates the three layered architecture that Statemaster uses for input and output [Boxes are C + + objects, ovals are functions]. In order to interface Statemaster with an input device, an output device, or application software, one chooses the most appropriate of these layers with which to communicate, and implements a driver for that layer.

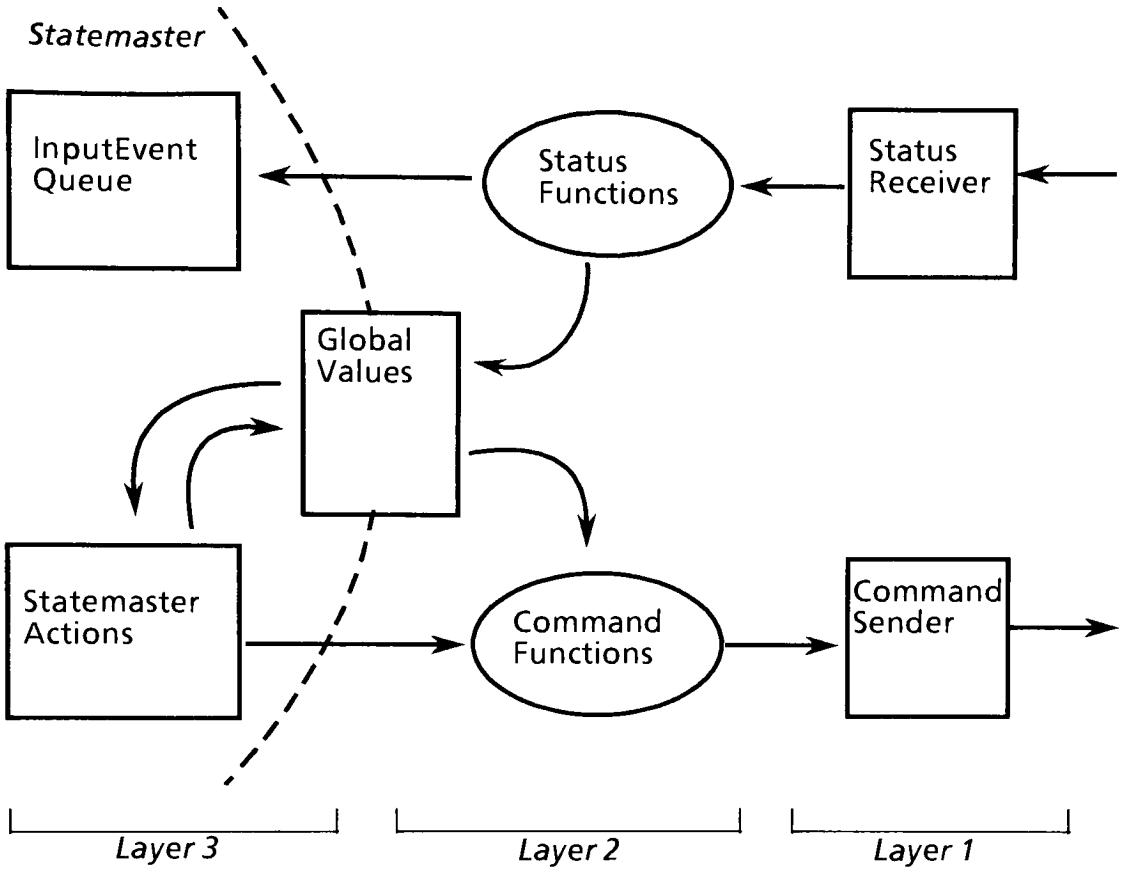


Figure 3.12
Statemaster communication with application software

3.3.3.1 Layer 3

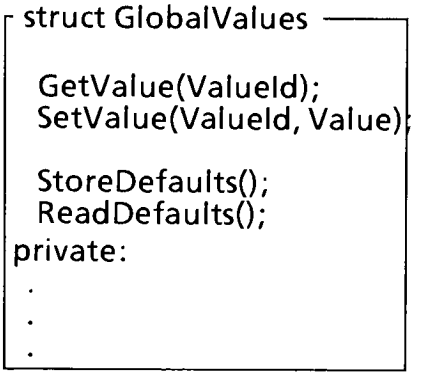
At the top layer and in the dialog specification, Statemaster receives input (or status) from the application through a queue of Events; it sends output (or commands) through a set of Actions. The GlobalValues can be read and set by Statemaster Actions.

Normally, Statemaster executes ACommand() Action by specifying only the command code. If the command requires parameters, then these must be supplied by setting appropriate GlobalValues before executing the command. This is similar to how registers are used in assembly language. The GlobalValues can be set by the entry and exit actions of dialog states. For example:

ACommand(JOB_INTERRUPT) Does not require any parameters

ACommand(Set_CROP_COORDS)	Should only be executed after the GlobalValues X, Y, L, and W have been set.

The C++ struct GlobalValues provides a uniform interface to all values that the dialog specification needs to access. It allows access to the values to be independent of where and how they are stored, according to a #defined ValueId.



igure 3.13
The GolbalValues object structure

Through the GlobalValues, Statemaster has access to all settable ASW parameters, and it has access to all the values needed to monitor the status of the ASW. The GlobalValues are set and read by the status and command functions of layer 2.

3.3.3.2 Layer 2

From UI to ASW: command functions

The command functions of layer 2 are called when ACommand() Action is executed from layer 3. These functions collect all the parameter information required from the GlobalValues, then they use the CommandSender in layer 1 to actually build and send the command record. Every command has its own command function.

From ASW to UI: status functions

The status functions of layer 2 are responsible for updating the GlobalValues to reflect the status of the ASW (if needed), then they must generate a Statemaster Event by pushing it on the back of the InputEventQueue.

3.3.3.3 Layer 1

Layer 1 exists to hide the specifics of the communication data structures. Agreed upon data structures for the StatusRecord and the CommandRecord must exist for the UI and ASW to talk to each other.

There are two C++structs: One to handle incoming data (the StatusReceiver), and one to handle outgoing data (the CommandSender).

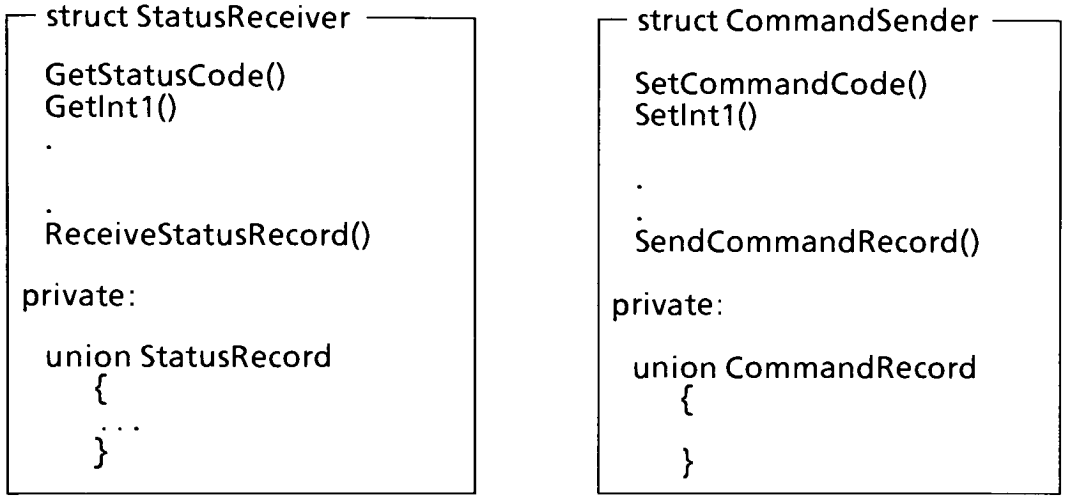


Figure 3.14
The StatusReceiver and CommandSender

From ASW to UI: StatusReceiver:

The StatusReceiver contains a C union into which data from ASW is read byte-for-byte by the low-level communication routines. The class also has a set of "Get" functions to extract meaningful typed data from the union, for example GetStatusCode(), and GetInt1().

When a status code is received, ReceiveStatusRecord() fills up the union, then calls UpdateCurrentStatus() from StatusFunctions. Based on the status code, a status function calls the StatusReceiver's "Get" functions to read relevant parameters from the union in the StatusReceiver and act on them if necessary before generating an Event.

From UI to ASW: CommandSender:

The CommandSender contains a C union that can be copied out to the ASW byte-for-byte to send a command record. The class also contains "Set"

functions for setting the fields of the data structure appropriately. These "Set" functions are called by the command functions described in level 2 based on the values in GlobalValues. After everything has been set, the CommandSender's SendCommandRecord() function is called and the command record is sent on its way to the ASW.

3.3.4 User-settable defaults and start-up system configuration

The GlobalValues described in level 3 are implemented in such a way as to enable user-settable defaults and start-up system configuration.

User-settable defaults:

Before calling an ASW command, it should not be necessary to explicitly set every parameter required by that function. Instead, there should be a set of default parameters already loaded into the UI to be used in case none are explicitly set. This is accomplished by loading the GlobalValues at system start-up with the defaults. To provide user-settable defaults, the user simply sets the values in GlobalValues as he usually does, then StoreValues() is executed so that these new values can be loaded at system startup.

Note: Allowing the user to specify the default state of the UI ("path" information) requires saving and restoring additional state information.

Start-up system configuration:

Machines will be shipped in different configurations. Some will have missing features. The UI's GlobalValues can be used to set up the UI to reflect system configuration. For example, if a machine is configured without the stapler option, then the value of GlobalValues(STAPLER_AVAILABLE) will always be false. So this is the value that would be loaded into the UI at system start-up along with any other UI-relevant configuration variables.

Note: In some cases configuring the dialog in this way will not be adequate, and instead a different version of the dialog specification will have to be shipped with the machine.

3.4 Equipment and Tools

The initial implementation of Statemaster was developed on a Sun 3 / 60 workstation. It was ported to an IBM AT compatible and to an imbedded system based on the Motorola 68000 processor.

Development on the Sun uses Glockenspiel C++, Sun C and SunOS, which is a BSD variant of UNIX. The IBM AT compatible uses Glockenspiel C++ and the Microsoft C 5.0 compiler running under MSDOS on a Sun 386i. The 68000 board uses Glockenspiel C++ and the Microtek C compiler running on the Sun using an Applied Microsystems emulator.

Chapter 4: Experience with Statemaster

To date, Statemaster has been used to implement five different user interface dialogs for reprographic machines that are under development at Xerox. The user interface dialogs are generally designed by human factors people and then handed off to software people who implement them using Statemaster. The dialog delivery process by which interfaces are designed, handed off, formally specified, implemented and reviewed is illustrated in Figure 4.1. The two main components of a user interface dialog are graphics and behaviors.

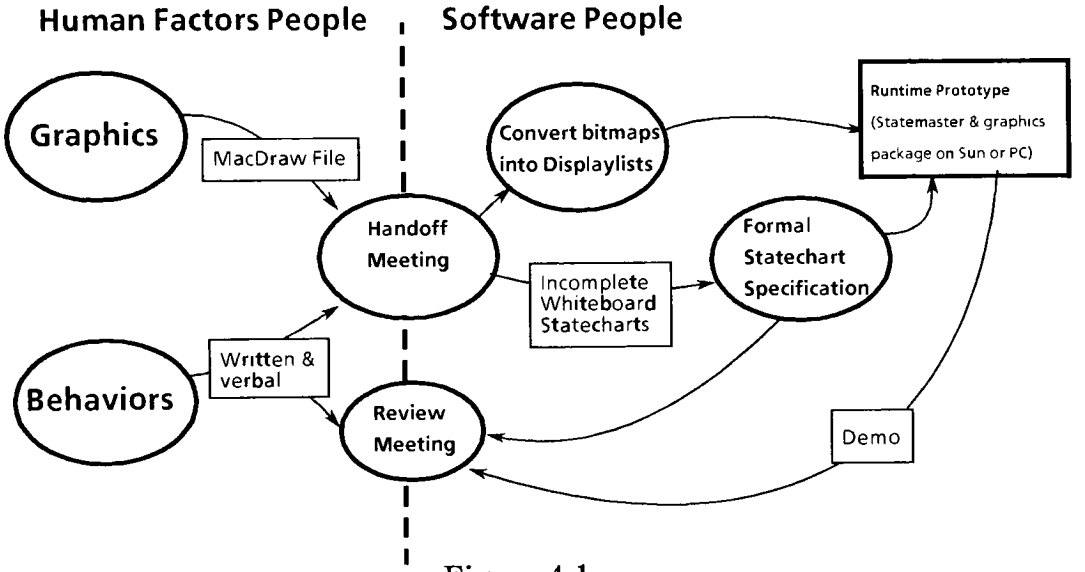


Figure 4.1
Dialog delivery process

4.1 Graphics

Graphic elements are created with a bitmap editor (MacDraw) and converted by hand from bitmaps into a highly compressed display list representation designed by Steve Dacek. Each graphic element is given a label that can be referenced by the Statemaster behavior specification.

4.2 Behaviors

The human factors people were encouraged by the software people to use Statecharts in their own design process, but they preferred to continue using informal text and storyboards to specify behaviors. These dialog specifications are given to the software people with verbal explanation at the

handoff meeting. There, both groups participate in drawing preliminary statecharts on a whiteboard, clarifying ambiguities, omissions, and misunderstandings as they come up. After the meeting, the UI Engineers (Statemaster users) draw up complete formal statecharts of the dialog with an object-oriented graphics editor based on the printouts from the electronic whiteboard. Experience to date indicates that the statechart representation is general enough to specify all the reprographic user interface designs that the human factors people require. Although the human factors people don't design using statecharts, they are able to understand them. They review the formal specification with the engineers at a review meeting before the dialog is finally implemented in Statemaster and integrated with graphics.

The running dialog is ported from the development environment (Sun Workstations) to the target hardware. There, it is reviewed again by the human factors designers and later used for operability testing. Currently the time cycle between a handoff meeting and the implemented dialog running on target hardware is approximately four weeks, although small changes are sometimes made in much less time.

4.3 Using Statemaster

Statemaster currently has only one heavy user and about five casual users. Linda Isaacson is the heaviest user, and although she has a background in computer science, she does not consider herself a programmer. The main experience that she brought to the task of UI Engineering with Statemaster was a very thorough understanding of the dialogs to be implemented.

The first dialog that Linda implemented consisted of approximately 30 statecharts, each one of which might typically consisting of about 5 child states, 15 actions, and 5 behaviors. She designed that dialog, drew the statecharts, typed them in and debugged the running user interface in the course of about four weeks. She says that learning the basics of statecharts does not take long, but there are many different ways of specifying any dialog. The difficulty is not in finding *a way* to specify it, but finding the *best way* to specify it. Most of her statecharts are based on previous charts that she drew for other parts of the dialog. Although Linda designed her own first Statemaster dialog, since then she has been specifying and implementing the

designs of human factors people. She describes her work process as consisting of the following five steps:

1) Understanding the dialog design in detail. This step consists of meeting with human factors and systems people, studying their documentation and calling them up to clarify any questions.

2) Thinking about how to statechart the dialog and drawing rough statecharts with pencil and paper.

3) Formally drawing the statecharts in Viewpoint (the object-oriented graphics editor) in complete detail. She feels that this step is not strictly necessary for her to implement the dialog, but she does it in order to keep a permanent record of the statecharts that can be distributed to others.

4) Typing in the specification. This step "ranges from being completely mindless to almost programming," depending on how modular and efficient she tries to make the specification.

5) Debugging the running dialog. The bugs consist mainly of mistakes in the order of specifying states. In the current implementation a state can not be used as a child or destination for a transition until it has already been specified earlier in the file. The compiler does not catch these errors, so they must be found at runtime.

When asked how she would rate the amount of time spent in each step, Linda rated step 1 as taking the most time and step 4 as taking the least. Step 2 rated as the second most time consuming, and step 5 rated just above step 3.

One comment that surprised the author was that Linda found it necessary to completely understand the dialog before starting to draw statecharts. The author's experience, on the other hand, is that drawing statecharts helps him to understand a dialog better.

4.4 Example Statemaster dialog

The appendix contains a substantial subset from a Statemaster dialog that was specified by Linda Isaacson. It has been slightly modified to remove references to other parts of the original user interface, but aside from that, this example is essentially intact and stands on its own. Please refer to the appendix to see the dialog specification.

Chapter 5: Ideas for the Future of Statemaster and Conclusion

Even though it has been used to implement several dialogs, Statemaster is still a far cry from the ultimate goal of every UIMS: to allow a user untrained in programming to efficiently implement any user interface he or she can dream up. It is clear, however, that no existing UIMS can do this despite advertising claims to the contrary. The following ideas for the future of Statemaster are aimed at bringing Statemaster still closer to the level at which dialog designers express themselves and at improving the turnaround time between dialog design and implementation. They are divided up into three categories: short term, medium term and long term ideas.

5.1 Short term ideas

These ideas could be implemented without making any significant changes to the existing system.

5.1.1 Minor optimizations

All C++ functions should be evaluated in terms of space and time as to whether they should be made inline or not. StatePointers may be able to be stored as short integers that point into an array of states. The ActionList object could store its control link instead of a pointer to its control link. #ifdefs should be used to create Statemaster data structures without debugging information. All ints could be explicitly declared as char, short or long.

5.1.2 Include files

Currently every new file that is added to the system must include a large number of .hxx files, and it is always unclear which include files are needed and which ones can be left out. A partial solution to this problem is to document the include file dependencies along with the objects declared in each .hxx file with a dependency chart. This would make it easier for programmers to work with Statemaster, but an automatic solution would be preferable. Such a system for automatically managing include files was described at the 1988 C++ Conference in Denver, Colorado and subsequently posted on the C++ usenet distribution list. Statemaster could be modified to use this scheme.

5.1.3 User documentation

The first Statemaster users learned to use it from working closely with its author. Subsequent users will not always be able to do this, so documentation must be provided to enable new users of Statemaster to learn it. Training materials may be necessary, as well as reference material to use while specifying dialogs

5.1.4 A library of C functions

Statemaster could be made into a library of functions that could simply be linked to without any need for recompilation. It would be beneficial to make this a library of ordinary C functions, so that users don't need to have C + + in order to use Statemaster. This would make the specification functions less intuitive, but with the use of macros (or a specification tool) they would still be usable. Files then would not have to be broken up for compiling on MSDOS, and porting Statemaster dialogs would be made easier.

5.1.5 Soft enlisting and dynamic events

The current implementation has a fixed number of possible events equal to the size of the dispatcher's table. This approach requires the number of events to be compiled into the code. Using #define to name events makes it difficult to add a new event names and numbers without recompiling the dispatcher. If Statemaster is to be made available to dialog specifiers as a compiled library, then it should not need to be recompiled for specific dialogs. Soft enlisting, an alternative implementation for the dispatcher, could achieve this aim.

5.1.6 Commands

The Value structures currently used for global variables could be organized in groups within structures called Commands. Each command would have all the values required for a specific command to the underlying machine or application. The dialog specification could set the values of the command , it could load it, save it, or reset it. This would simplify saving and restoring sets of machine commands, and could localize all the machine interface code to be part of these objects.

5.2 Medium term ideas

These ideas would require more significant changes or additions to the current system and would have a greater impact on how the system is used.

5.2.1 Storable & retrievable objects

Currently, Statemaster specifications are stored as the C++ function specify(), which constructs all dialog objects using C++ object constructors. This function is called by the run-time system before entering the dispatch loop. An alternative is to use storable and retrievable objects to allow completely separate specification and run-time environments. This would also allow specification-time objects and run time objects to be implemented differently from each other. Specification objects need to contain debugging information and data structures that are easily modified by an interactive specification tool. The run-time objects, on the other hand, should be highly compressed and optimized. The current implementation of Statemaster is a compromise between the flexibility requirements of specification, and the efficiency requirements of implementation, though the compromise was made heavily in favor of implementation requirements. In the current implementation, calls to state constructors tend to be deeply nested and cause the compiler to run out of stack space under MSDOS. If specification-time objects are implemented in a way that allows interactive editing, then these deeply nested calls to constructor objects will no longer be necessary.

5.2.2 Modular interaction techniques

The value of a UIMS is largely determined by its "toolkit" of dialog building blocks that designers and UI Engineers can draw from. Often, UIMS toolkits cannot be created or modified by the UI Engineer. Instead, the toolkits are programmed by people very knowledgable about the inner workings of the UIMS. Statemaster is currently still at this stage. Although the interface between Statemaster and the graphics package is very clearly defined, the interface between the interaction techniques and Statemaster as well as the interface between the interactors and the Graphics package are still not defined clearly enough (see Figure 5.1). As a result, adding an interaction technique (*e.g.* scrollable queues of text fields) leads to much confusion as to who does what. A clear relationship must be established between all these components that is general enough to handle any variety of interaction

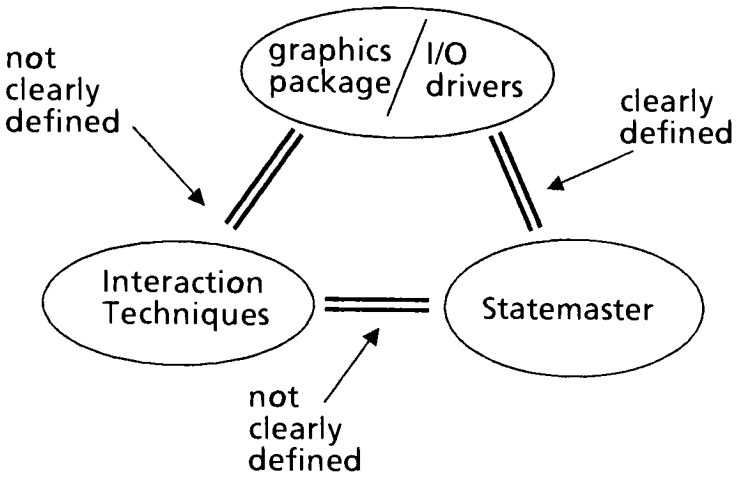


Figure 5.1

techniques, including dynamic user interface techniques and text fields. Ideas for this interface can be borrowed from work done with MacApp, the Andrew Toolkit, InterViews, Open Dialog, and also the X11 and NeWS window servers.

5.2.3 Building blocks and container objects

To make specification of UI dialogs easier, it would be very helpful if the UI Engineer could easily reuse work from previous dialogs (or other parts of the current dialog) and enable building of user interfaces from parts of other dialogs or building blocks. This facility is available in a limited form for the current implementation of Statemaster (through Adopt), but it can be utilized only if the element to be reused is exactly the same in every instance. There is no facility for specifying a primitive dialog element that can be reused with slight modifications in different parts of the dialog. Statecharts is well suited for doing bottom up design. The only thing that is lacking is a way of making a statechart building block or container that specifies a piece of dialog with parameters. For example, most XorGroupers of toggle buttons are essentially the same: each button puts up a selected bitmap on entry and a deselected bitmap on exit, and each sets the same global variable to a different value. The statechart for describing this appears many times in many dialogs, but every time it is slightly different. What is needed is a way to specify this statechart once, but with parameters: the number of buttons, the set of

bitmaps to use, the global variable and the values. This could be implemented without too much difficulty by allowing states to have local variables, but it requires a corresponding representation for specifying these building blocks within the statecharts.

5.2.4 Implicit linking of states with graphics

In a typical dialog, virtually every state displays a bitmap as one of it's entry actions. Specifications could be made much simpler if an entry bitmap were an implicit part of every state. Implicit sensitization and desensitization of bitmaps would also be useful.

5.2.5 Statechart specification tools

The current implementation of Statemaster requires the statecharts to be typed into a text file by hand from the graphic drawings. Not only is this tedious and mechanical, but since both the graphics and the text are maintained by hand they do not always exactly reflect each other as they should. Sometimes changes are made to the text file without updating the graphics or vice versa. These problems are manageable when a few people work on small dialogs, but when projects become larger this problem will become more and more significant.

A good editor can also prevent the UI engineer from making syntactic statechart errors. It could check semantic errors by making sure all states are reachable, all events are possible, and no conflicting behaviors are simultaneously active. It should incorporate visual representations for the building blocks and container objects described above. Ideally this editor should be able to communicate with the graphics editor, providing two views on the same dialog: the behavioral view and the graphic view. While the dialog is running on the development system, the statechart representation of behaviors could be animated to illustrate the running dialog at the same time as the visual appearance of the dialog is changing. Access to both views running at the same time would be a very effective dialog debugging tool.

5.2.6 Translation tools

Tools must be provided to aid in the task of translating a user interface dialog. Correct translation can only be done when text messages are presented to the translator in the context that they appear in the user interface. The

translation tools must therefore provide a way of looking at each text string of the dialog in its full context. Because the dialog is implemented in terms of Statecharts, the translation tool can keep track of which states have been visited by the translator, and verify that no state has been missed. This tool should provide special access to text messages, so that they can be sent to external translation systems and then read back in, reformatted, and incorporated into the message database.

5.2.7 Action language

The current implementation has a limited set of actions, and each action is implemented as a separate object that can be constructed and then executed. As more complex user interfaces are developed with Statemaster, actions are needed to provide most features of a general purpose programming language. Virtually every UIMS needs to have a language for specifying conditional expressions, simple arithmetic, and subroutines. Some UIMS's have their own interpreted programming languages *e.g.* Demo II and Hypercard. UIMSs that are themselves implemented in an interpreted language often use that language *e.g.* Lisp or Smalltalk. The rudimentary action language that is currently provided with Statemaster is an example of a language created just for that purpose. There is no good reason why Statemaster's action language should be incompatible with existing languages. Instead, an implementation compatible with an existing language should be provided with Statemaster. The most convenient languages to use for this purpose are interpreted, and two languages that seem good candidates are Hypertalk and Postscript: Hypertalk because it is widely used as a behavior specification language for Hypercard, and Postscript because it is in widespread use as a graphics specification language for printers and window systems. It would be advantageous to use compatible action and graphics specification languages. Users would not have to learn two different languages and the common superset would have a single implementation.

These considerations must also keep in mind how Statemaster will interface with interactors, and how UI Engineers can most naturally use the actions. The integration of the language with Statemaster and its interaction techniques must also fit in with an object-oriented architecture, even if the language itself is not fundamentally object-oriented.

5.3 Long term ideas

The following long term ideas would require significant work to implement and would substantially change the way that the system is used.

5.3.1 Integrated system vs. set of tools

One common approach to implementing a UIMS is to supply a fully integrated environment that allows behavior specification, graphics editing, text editing, translation, simulation of the underlying machine and any other desired functions in one completely integrated package. This approach is very good when a single user is developing all aspects of the user interface because he can easily modify any aspect of the interface from within the same environment. This approach leads to problems, however, when working in an organizational setting where a large group of people are involved in developing the user interface. Each person in the group has a set of responsibilities dealing with specific parts of the user interface. Each person or organization has its own needs as to what kind of information is required about the user interface, and how to modify it. Another important consideration is that the user interface development system will not remain static. There will always be new things it will be required to do, and new people that will want to modify the way it works. It is important to keep this in mind when planning the architecture and organization of the system.

5.3.2 The centralized data approach

The monolithic, integrated tool approach is not so good for development of large, complex user interfaces because complex interfaces developed in a corporate environment are put together by a team consisting of many people and organizations. Each person involved has different requirements from the information in the user interface. For example:

- Behavior specifiers need to work with statecharts. Different people will need to look at the statecharts at different levels of detail and look at the associated requirements and specification documents.
- Graphics designers must be able to see and edit graphic elements.
- Interaction-technique implementors must be able to test and modify interaction techniques.

- Some users will have the responsibility to keep track of all system messages.
- Some users will keep track of all fault and exception conditions.
- Other users will track and maintain the UI / Machine interface. (some with commands, some with status messages)
- Some users require text and translation data.
- The runtime system requires integrated data structures for all the system information.
- There must be a version control and archiving process for all aspects of system.

These requirements are very similar to those of CASE (Computer Aided Software Engineering) systems, so the way in which CASE systems are implemented may be the best approach for implementing a large scale UIMS. Most software engineering tools that claim to be CASE tools are built around a centralized "data repository" [McCl89]. A CASE-style implementation of Statemaster would have a database that keeps track of all the states, behaviors, actions, messages, display lists, translated strings, etc. There would be an associated set of tools for editing these elements. Additional tools could run consistency checks on the database and verify that all components are being used and properly referenced. Development tools would extract their data from the database along with tracing and debugging information. Runtime environments would extract optimized versions of the same data. The entire system could be integrated within a commercial software engineering environment such as Sun's NSE (Network Software Environment) in order to take advantage of its facilities for version control of data files as well as software modules [see Figure 5.2].

5.3.3 Managing the evolution a UIMS

A user interface development system that is tightly integrated can be very difficult to modify. When a certain aspect of the system changes, then the entire system can be impacted. Experience at Xerox shows that different versions of the system proliferate each one for its own specific set of users and development branches off in directions that are never reconciled.

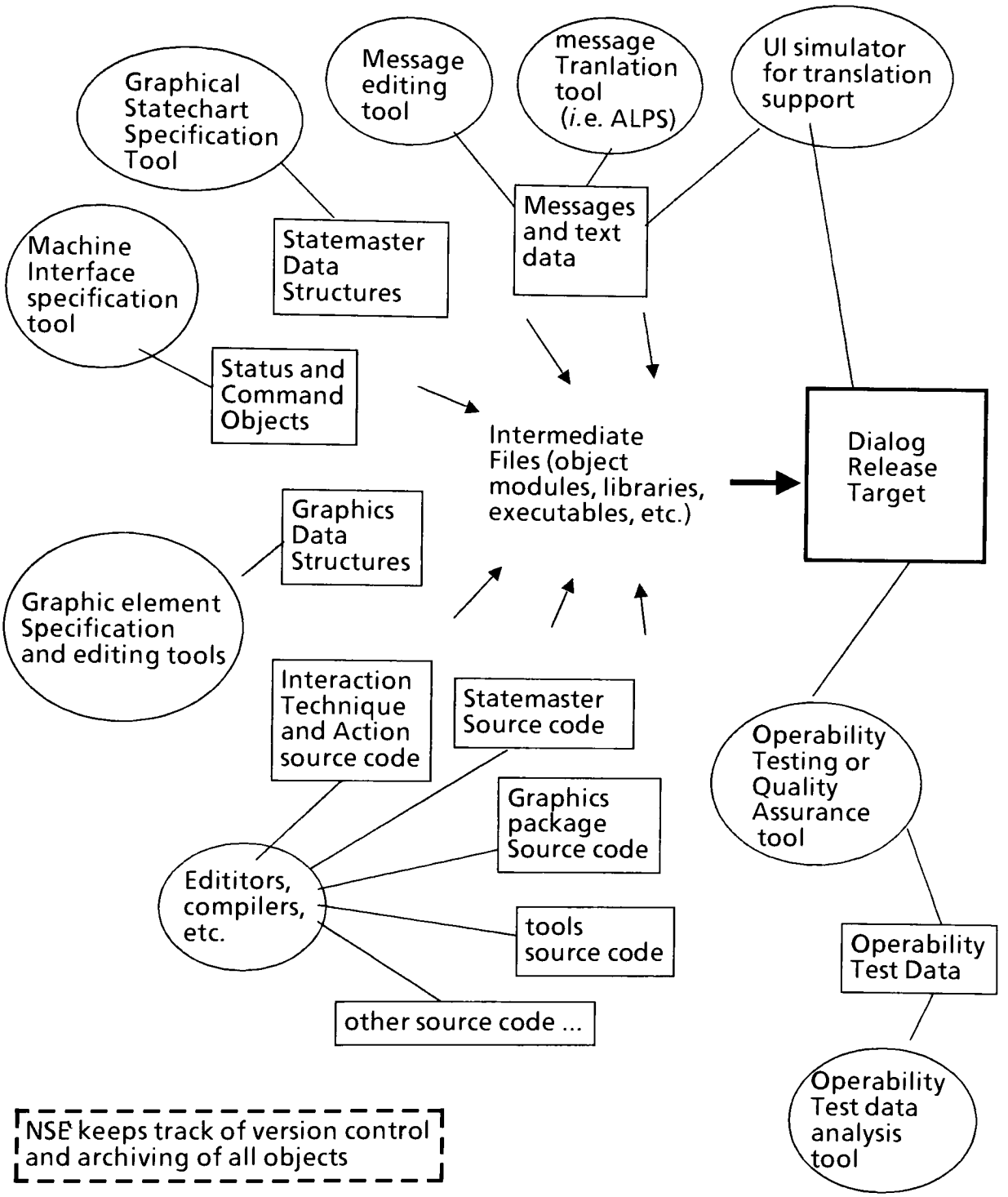


Figure 5.2
Ideas for Statemaster tools and objects within the Network Software Environment

If tools are defined in terms of queries to a centralized database, then the format of the database could change without introducing incompatibilities with other tools, just as long as new queries did not change the results of the old queries.

The database approach could help to insulate the parts of the system from each other. As features (or data items) are added for one tool, the interface of the database to the other tools can be made to remain consistent through the database query language. This would allow different components of the user interface development system to evolve more independently and it would make it easier to coordinate enhancements being made by different people or organizations. As requirements change, new information and queries can be added to the database to accommodate new tools or new tool features.

The data oriented approach can also make the problem of backwards compatibility easier. Old versions of dialogs and tools may still be in use by some people, but they can continue to run correctly in the new environment as long as the database queries yield the same results.

Stand-alone tools can share information mainly through a shared database, but they could also be made to communicate with each other while running simultaneously through UNIX pipes or inter process communication. If tools interface through UNIX, that may help to standardize their interfaces, and it becomes easier to develop these tools independently. Integration is done by the operating system instead of by linking everything together in one big executable.

5.3.4 Metaphors for statecharts, actions, behaviors

Statecharts are relatively simple, but they are intimidating to people who have not been exposed to computer science. It may be possible to find a metaphor for states and groupers that would help non-technical users easily understand how to construct dialogs. The card and link button in Hypercard, for example, are good metaphors for simple states and transitions. Perhaps these metaphors can be extended to include the functionality of XorGroupers and AndGroupers. Statemaster would be made more accessible to non-programmers if it had a consistent set of familiar metaphors that represent its constructs.

5.3.5 Use Statemaster to implement its own specification tools

It is not unreasonable to expect a UIMS to be capable of generating the user interfaces of its own specification tools. It is a good test of the system and improvements to one system help the other. Experience gained and problems solved in building a specification tool with Statemaster can be used to implement other, simpler dialogs. If specification tools (*e.g.* a statechart or display list editor) are to be developed for Statemaster, they should be implementable using Statemaster itself.

5.3.6 Undo support

Every action could be implemented with an associated undo function. The AUndo() action would undo the last action executed, and it could be executed a specified number of times. Automatic undo would of course require extra storage, because most actions would have to store something when executed in order for their corresponding undo actions to be able to work.

5.3.7 Default user interface

The MIKE UIMS [Olse86] generates an entire user interface automatically simply from a specification of the machine interface. This feature is very useful to get a quick start in developing a UI for a specific application.

5.3.8 User tailorable user interfaces

If the UIMS that was used to create a user interface is very easy to use then the possibility arises of integrating the UIMS into the user interface itself, allowing users to modify the UI according to their tastes.

5.3.9 Automatic macro detection

A UI implemented in terms of statecharts can log user events and notice sequences of actions that often get executed as a group. The UI could be made to be smart enough to detect this and generate a macro (button or key) that would execute that set of commands with one keypress.

5.4 Conclusion

Although Statemaster is not usable by a designer with no programming skills, it has been successfully used by "UI engineers" to implement a variety of user interface dialogs on several different hardware platforms. Some of the dialogs implemented with Statemaster include buttons, menus, text fill-in fields and

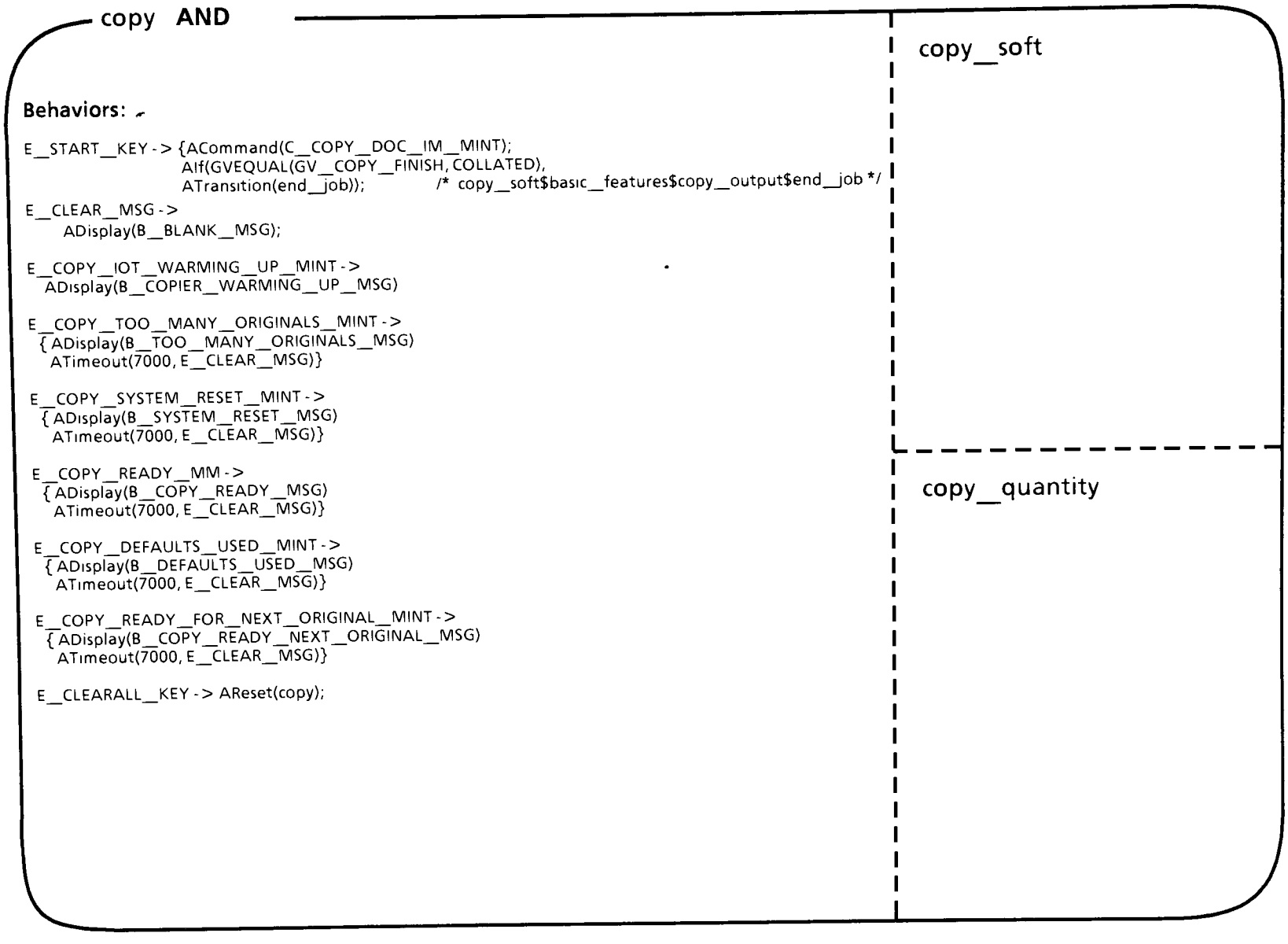
list editing. Dialog specifications are hardware independent so that the same dialog can run both on a Sun using a mouse with the SunWindows package, or on an IBM PC using a touch screen and a PC graphics package. Significant implementation effort is saved because Statemaster is portable and efficient enough to be used on low cost target hardware. Prototyping can begin on Sun workstations and PCs before custom target hardware is ready, then, when the final UI software is implemented, very little of the prototyping work is lost.

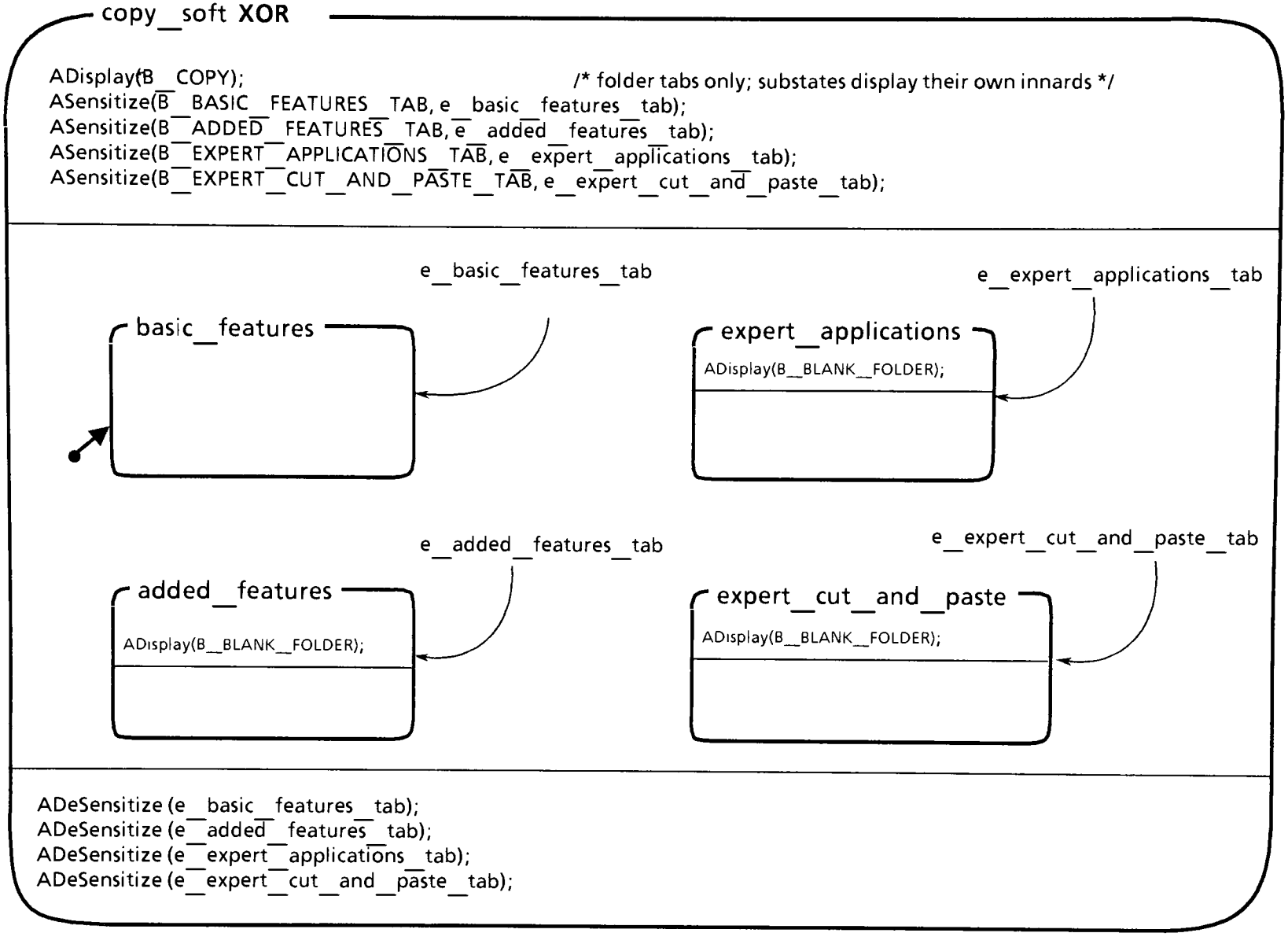
Initial experiences show that statecharts are an effective representation for communicating dialog designs and modifications within a user interface implementation team because they are formal, compact, and can be understood by all members of the team. Statecharts are easily drawn and modified on a whiteboard during a meeting, then printed or distributed electronically. They can represent virtually any user interface dialog, they help to reduce misunderstandings, and they improve collaboration. The direct mapping from statecharts to Statemaster's data structures enables a quick turnaround time between a UI design and its implementation.

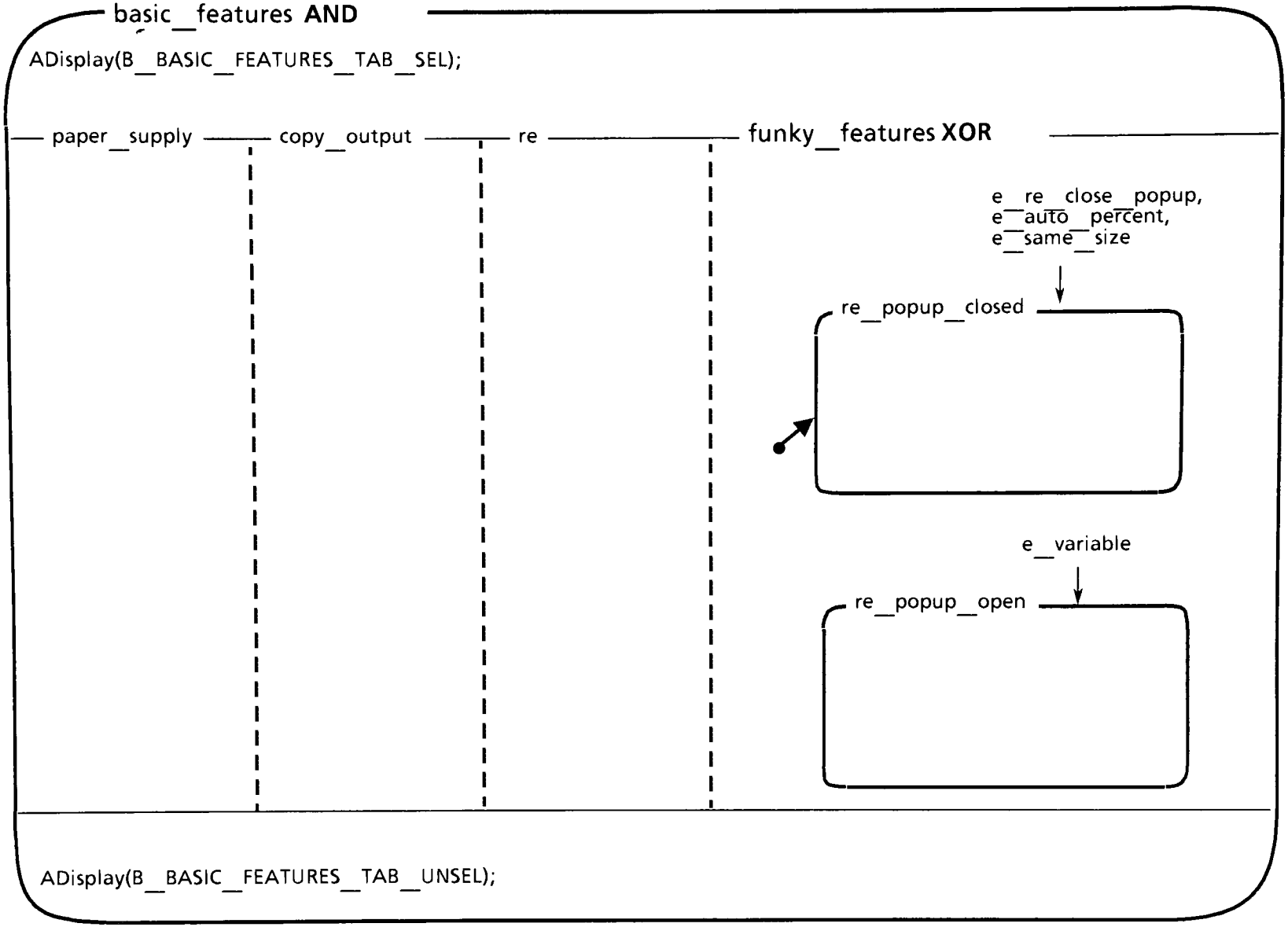
Appendix: Example Statemaster dialog

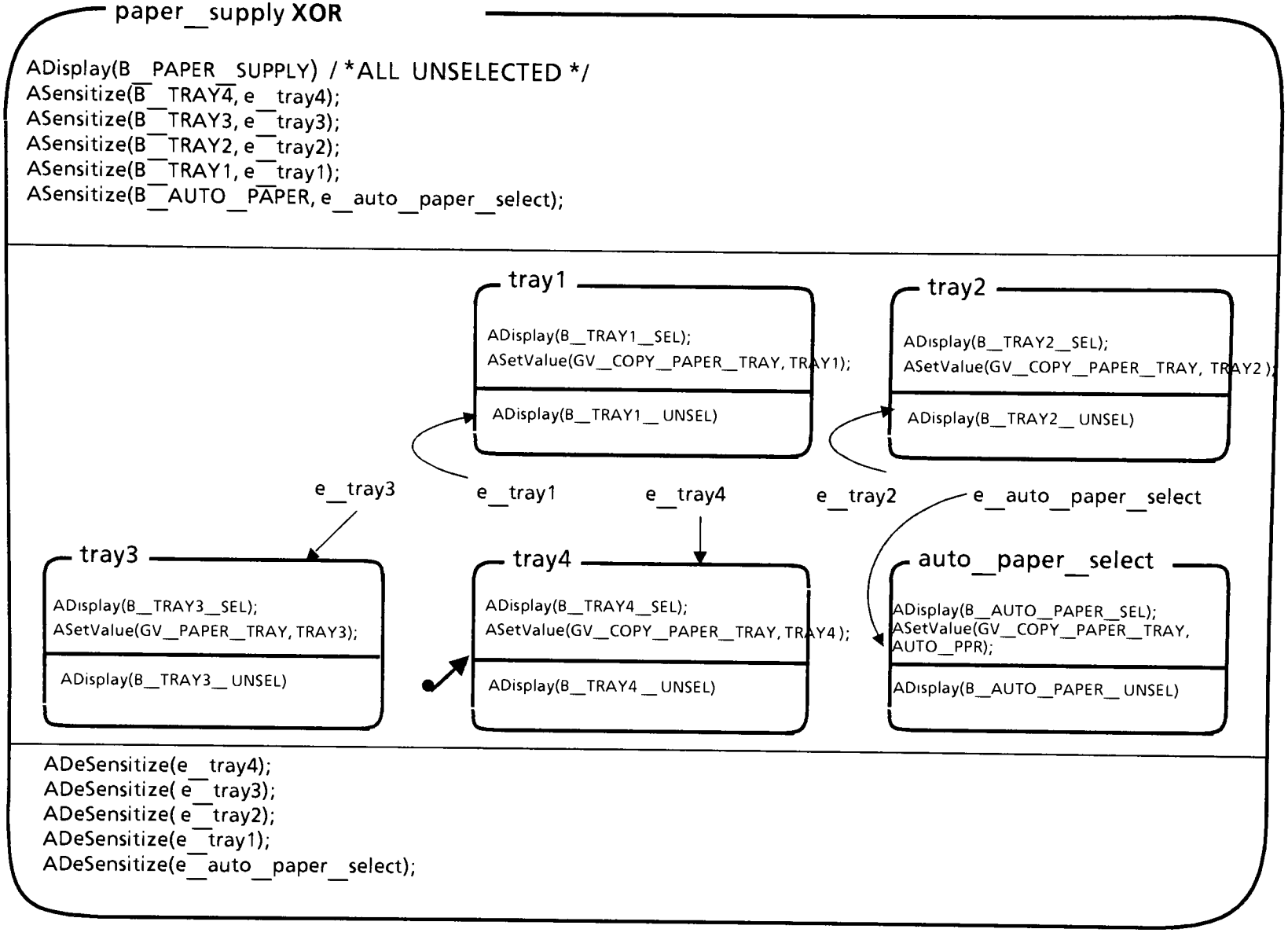
This appendix contains a substantial subset from a Statemaster dialog that was specified by Linda Isaacson. It has been slightly modified to remove references to other parts of the original user interface, but aside from that, this example is essentially intact and stands on its own.

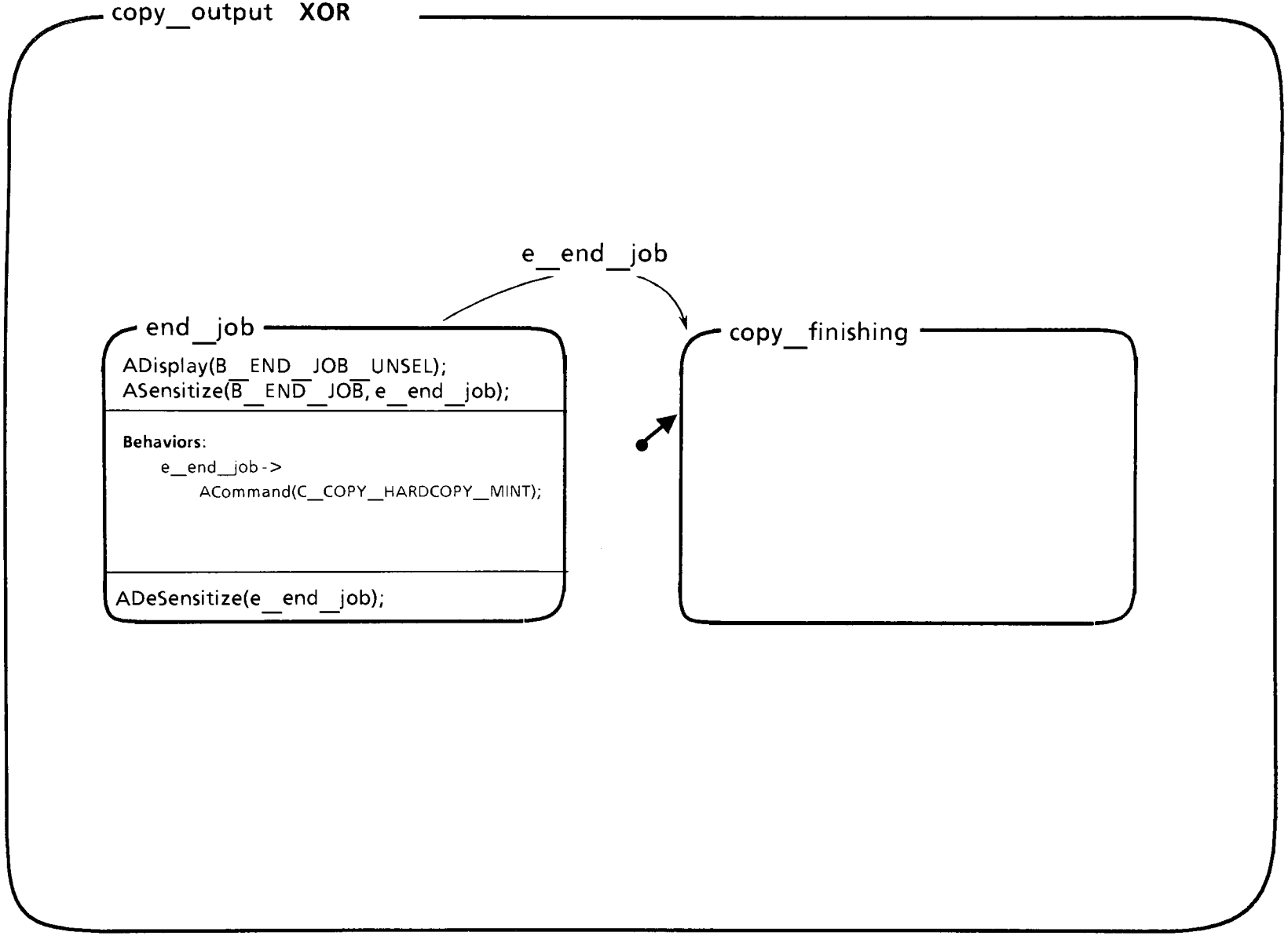
In reading these Statecharts you will notice some conventions. Event identifiers begin with the letter e or E, and Bitmap identifiers begin with a B. Two actions that are frequently used are ADisplay() and ASensitize(). ADisplay takes a bitmap identifier as its argument and displays it when executed. ASensitize() takes a bitmap and event identifier as arguments. After ASensitize() has been executed, and before ADeSensitize() has been executed, the specified event will be generated any time the user touches the sensitized bitmap. Most other actions used in the following pages are either unimportant or self-explanatory.

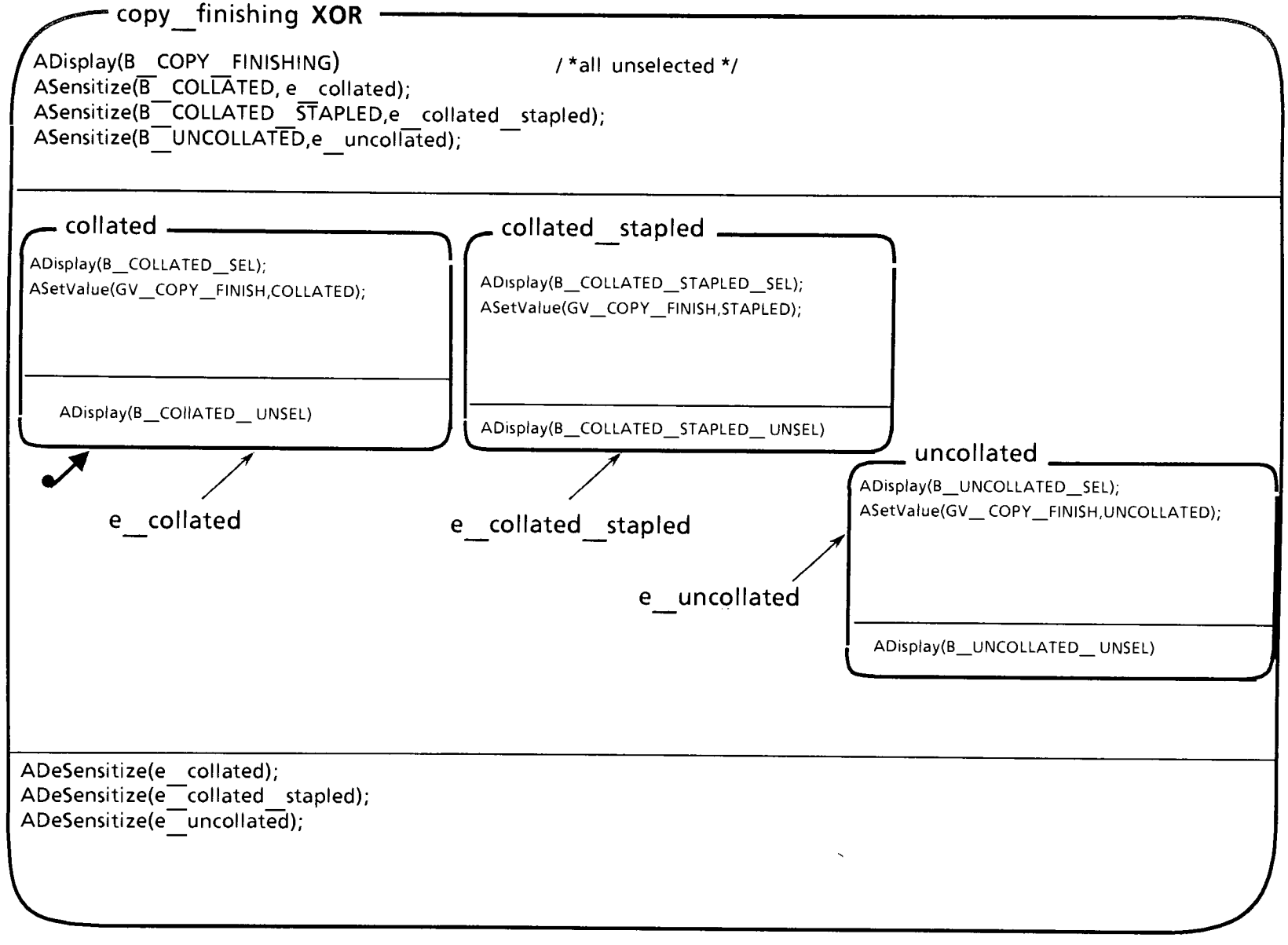

copy AND
Behaviors:
E_START_KEY -> {ACommand(C_COPY_DOC_IM_MINT);
AIf(GVEQUAL(GV_COPY_FINISH, COLLATED),
ATransition(end_job));
/* copy_soft$basic_features$copy_output$end_job */
E_CLEAR_MSG ->
ADisplay(B_BLANK_MSG);
E_COPY_IOT_WARMING_UP_MINT ->
ADisplay(B_COPIER_WARMING_UP_MSG)
E_COPY_TOO_MANY_ORIGINALS_MINT ->
{ADisplay(B_TOO_MANY_ORIGINALS_MSG)
ATimeout(7000, E_CLEAR_MSG)}
E_COPY_SYSTEM_RESET_MINT ->
{ADisplay(B_SYSTEM_RESET_MSG)
ATimeout(7000, E_CLEAR_MSG)}
E_COPY_READY_MM ->
{ADisplay(B_COPY_READY_MSG)
ATimeout(7000, E_CLEAR_MSG)}
E_COPY_DEFAULTS_USED_MINT ->
{ADisplay(B_DEFAULTS_USED_MSG)
ATimeout(7000, E_CLEAR_MSG)}
E_COPY_READY_FOR_NEXT_ORIGINAL_MINT ->
{ADisplay(B_COPY_READY_NEXT_ORIGINAL_MSG)
ATimeout(7000, E_CLEAR_MSG)}
E_CLEARALL_KEY -> AReset(copy);
copy_soft
copy_quantity


copy__soft XOR
ADisplay(B__COPY);
/* folder tabs only; substates display their own innards */
ASensitize(B__BASIC__FEATURES__TAB, e__basic__features__tab);
ASensitize(B__ADDED__FEATURES__TAB, e__added__features__tab);
ASensitize(B__EXPERT__APPLICATIONS__TAB, e__expert__applications__tab);
ASensitize(B__EXPERT__CUT__AND__PASTE__TAB, e__expert__cut__and__paste__tab);
e__basic__features__tab
basic__features
e__expert__applications__tab
expert__applications
ADisplay(B__BLANK__FOLDER);
e__added__features__tab
added__features
ADisplay(B__BLANK__FOLDER);
e__expert__cut__and__paste__tab
expert__cut__and__paste
ADisplay(B__BLANK__FOLDER);
ADeSensitize (e__basic__features__tab);
ADeSensitize (e__added__features__tab);
ADeSensitize (e__expert__applications__tab);
ADeSensitize (e__expert__cut__and__paste__tab);


basic__features AND
ADisplay(B__BASIC__FEATURES__TAB__SEL);
paper__supply
copy__output
re
funky__features XOR
e__re__close__popup,
e__auto__percent,
e__same__size
re__popup__closed
e__variable
re__popup__open
ADisplay(B__BASIC__FEATURES__TAB__UNSEL);

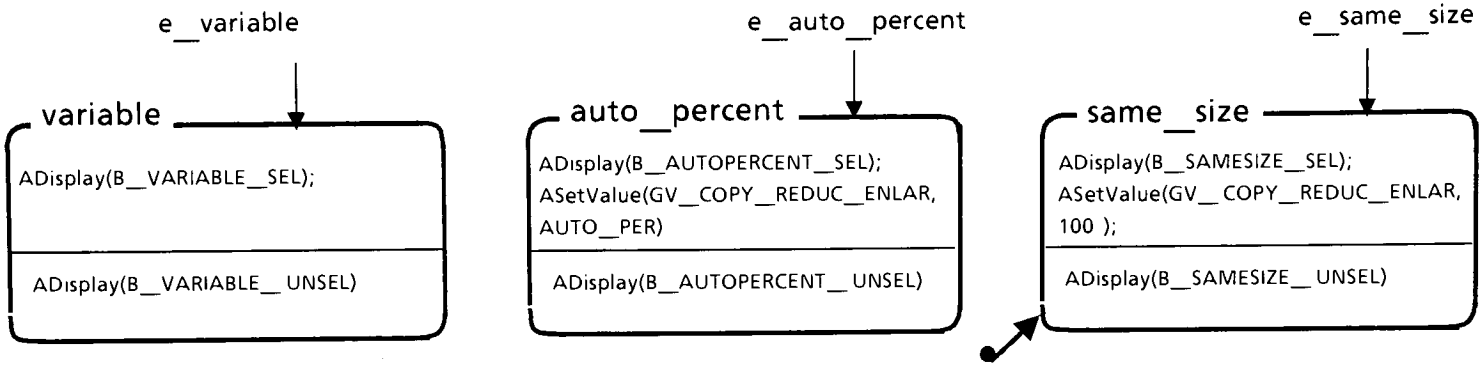

paper__supply XOR
ADisplay(B__PAPER__SUPPLY) /*ALL UNSELECTED */
ASensitize(B__TRAY4, e__tray4);
ASensitize(B__TRAY3, e__tray3);
ASensitize(B__TRAY2, e__tray2);
ASensitize(B__TRAY1, e__tray1);
ASensitize(B__AUTO__PAPER, e__auto__paper__select);
tray1
ADisplay(B__TRAY1__SEL);
ASetValue(GV__COPY__PAPER__TRAY, TRAY1);
ADisplay(B__TRAY1__UNSEL)
tray2
ADisplay(B__TRAY2__SEL);
ASetValue(GV__COPY__PAPER__TRAY, TRAY2);
ADisplay(B__TRAY2__UNSEL)
e__tray3
e__tray1
e__tray4
e__tray2
e__auto__paper__select
tray3
ADisplay(B__TRAY3__SEL);
ASetValue(GV__PAPER__TRAY, TRAY3);
ADisplay(B__TRAY3__UNSEL)
tray4
ADisplay(B__TRAY4__SEL);
ASetValue(GV__COPY__PAPER__TRAY, TRAY4);
ADisplay(B__TRAY4__UNSEL)
auto__paper__select
ADisplay(B__AUTO__PAPER__SEL);
ASetValue(GV__COPY__PAPER__TRAY, AUTO__PPR);
ADisplay(B__AUTO__PAPER__UNSEL)
ADeSensitize(e__tray4);
ADeSensitize(e__tray3);
ADeSensitize(e__tray2);
ADeSensitize(e__tray1);
ADeSensitize(e__auto__paper__select);


copy_output XOR
e_end_job
end_job
ADisplay(B_END_JOB_UNSEL);
ASensitize(B_END_JOB, e_end_job);
Behaviors:
e_end_job ->
ACommand(C_COPY_HARDCOPY_MINT);
ADeSensitize(e_end_job);
copy_finishing

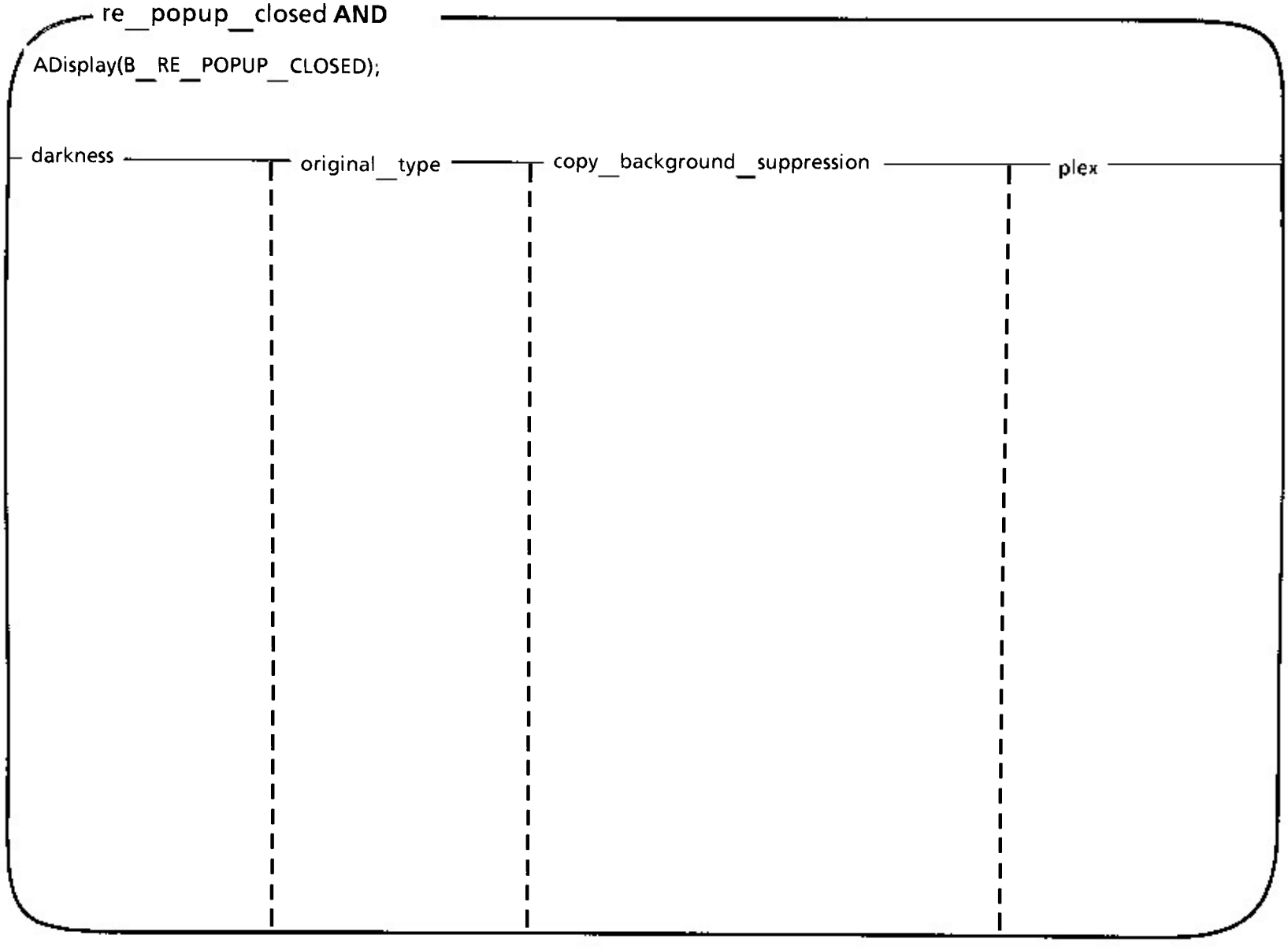

copy_finishing XOR
ADisplay(B_COPY_FINISHING) /*all unselected */
ASensitize(B_COLLATED, e_collated);
ASensitize(B_COLLATED_STAPLED,e_collated_stapled);
ASensitize(B_UNCOLLATED,e_uncollated);
collated
ADisplay(B_COLLATED_SEL);
ASetValue(GV_COPY_FINISH,COLLATED);
ADisplay(B_COLLATED_UNSEL)
collated_stapled
ADisplay(B_COLLATED_STAPLED_SEL);
ASetValue(GV_COPY_FINISH,STAPLED);
ADisplay(B_COLLATED_STAPLED_UNSEL)
uncollated
ADisplay(B_UNCOLLATED_SEL);
ASetValue(GV_COPY_FINISH,UNCOLLATED);
ADisplay(B_UNCOLLATED_UNSEL)
e_collated
e_collated_stapled
e_uncollated
ADeSensitize(e_collated);
ADeSensitize(e_collated_stapled);
ADeSensitize(e_uncollated);

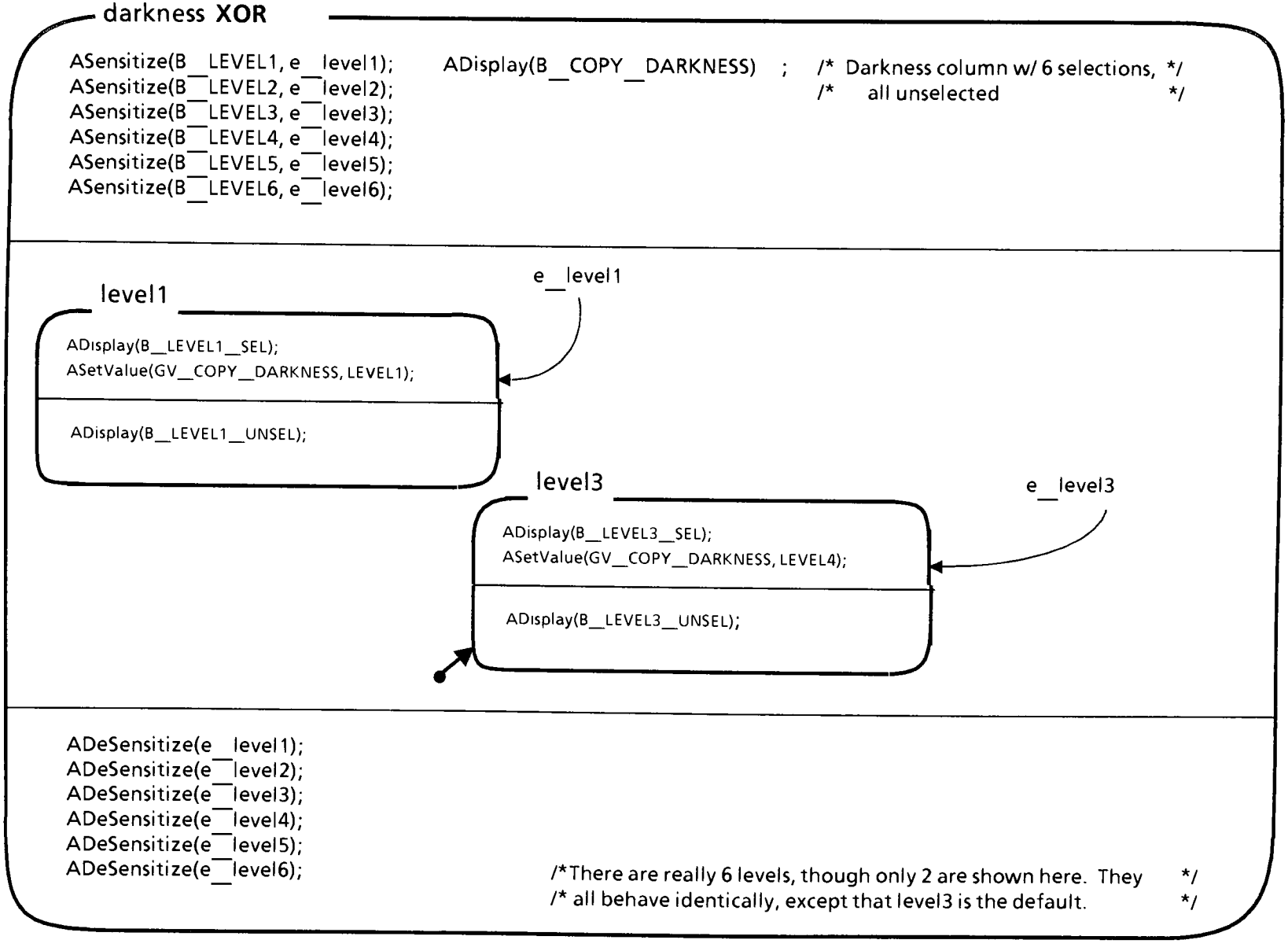
re **XOR**

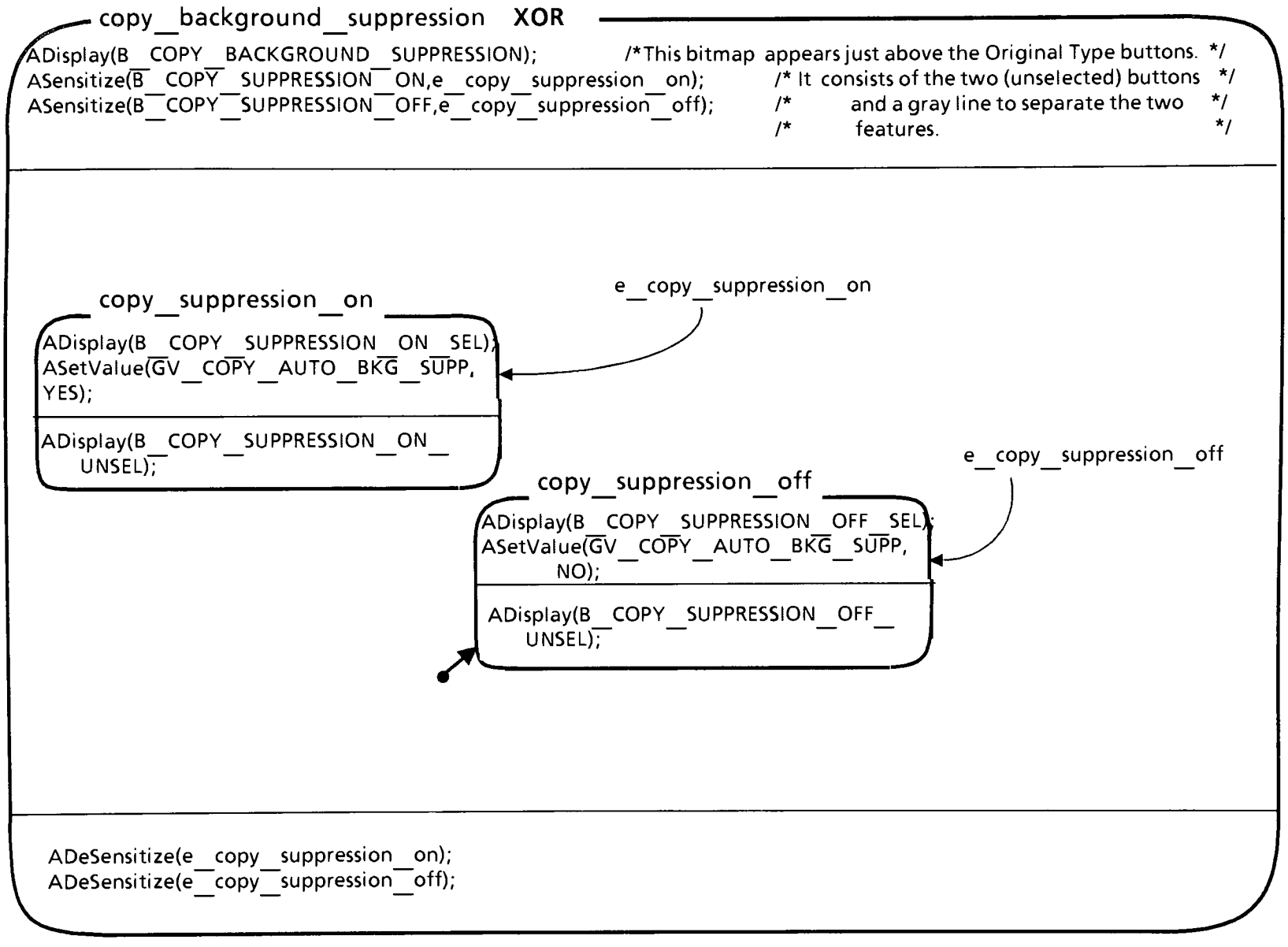
```
ADisplay(B__RE)                                  /* RE column containing 3 selections, all unselected */
ASensitize(B__VARIABLE, e__variable);
ASensitize(B__AUTOPERCENT,e__auto__percent);
ASensitize(B__SAMESIZE,e__same__size);
```

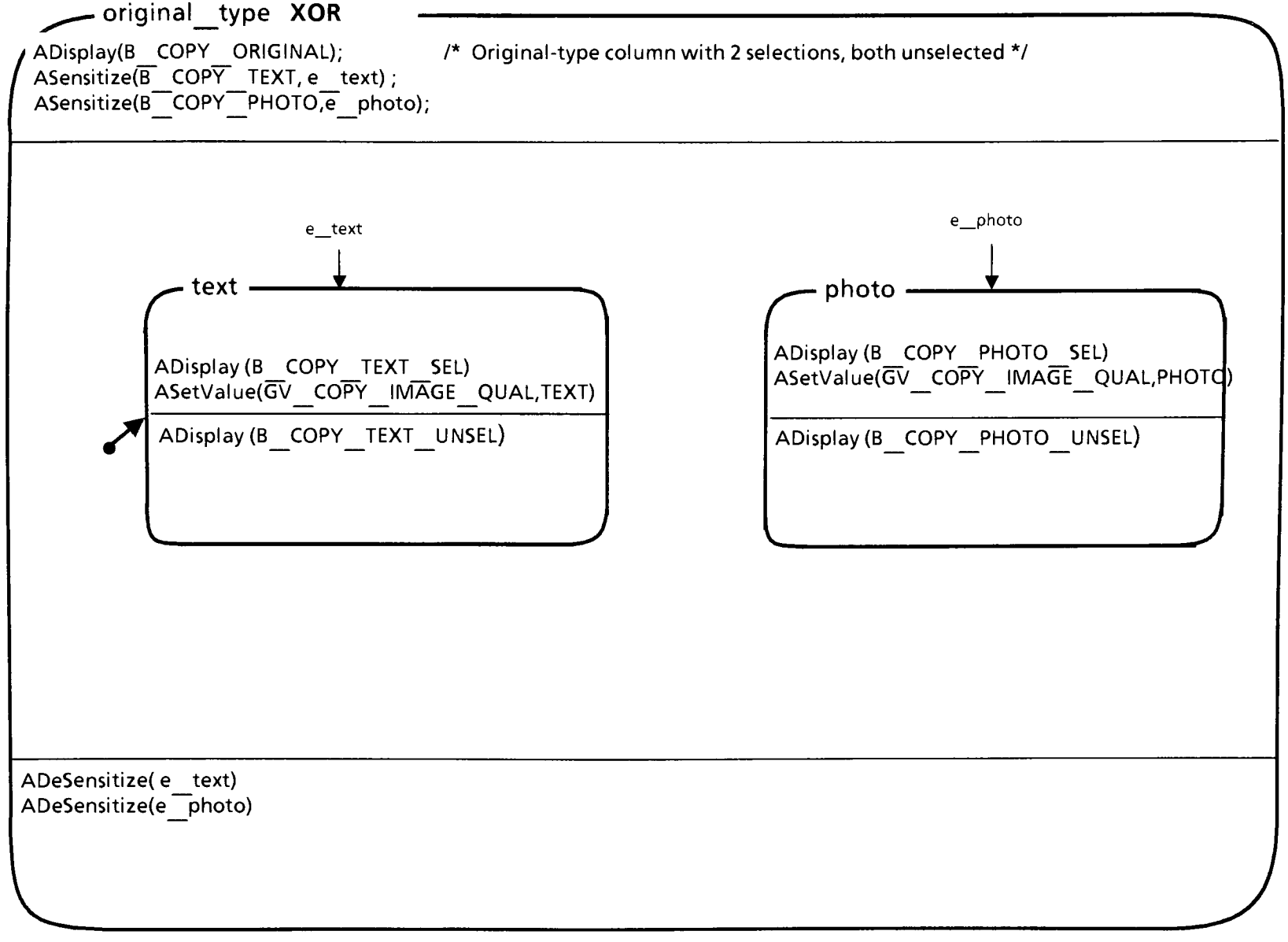


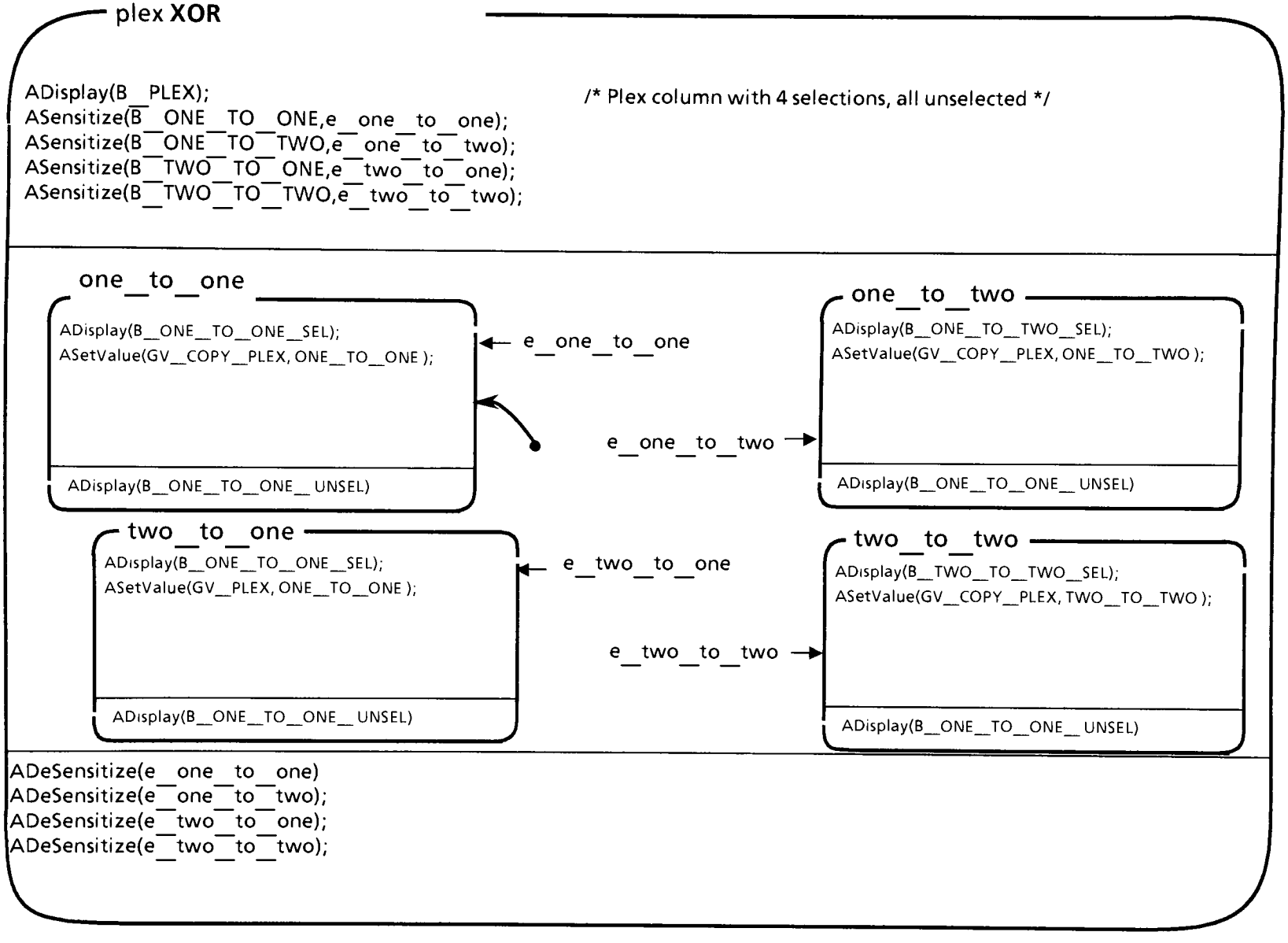

```
ADeSensitize(e__re__open__pop__up);
ADeSensitize(e__autopercent);
ADeSensitize(e__samesize);

/*  Note that the e__re__open__popup event also causes a transition to  */
/*       $basic__features$funky__features$re__popup__open                */
```

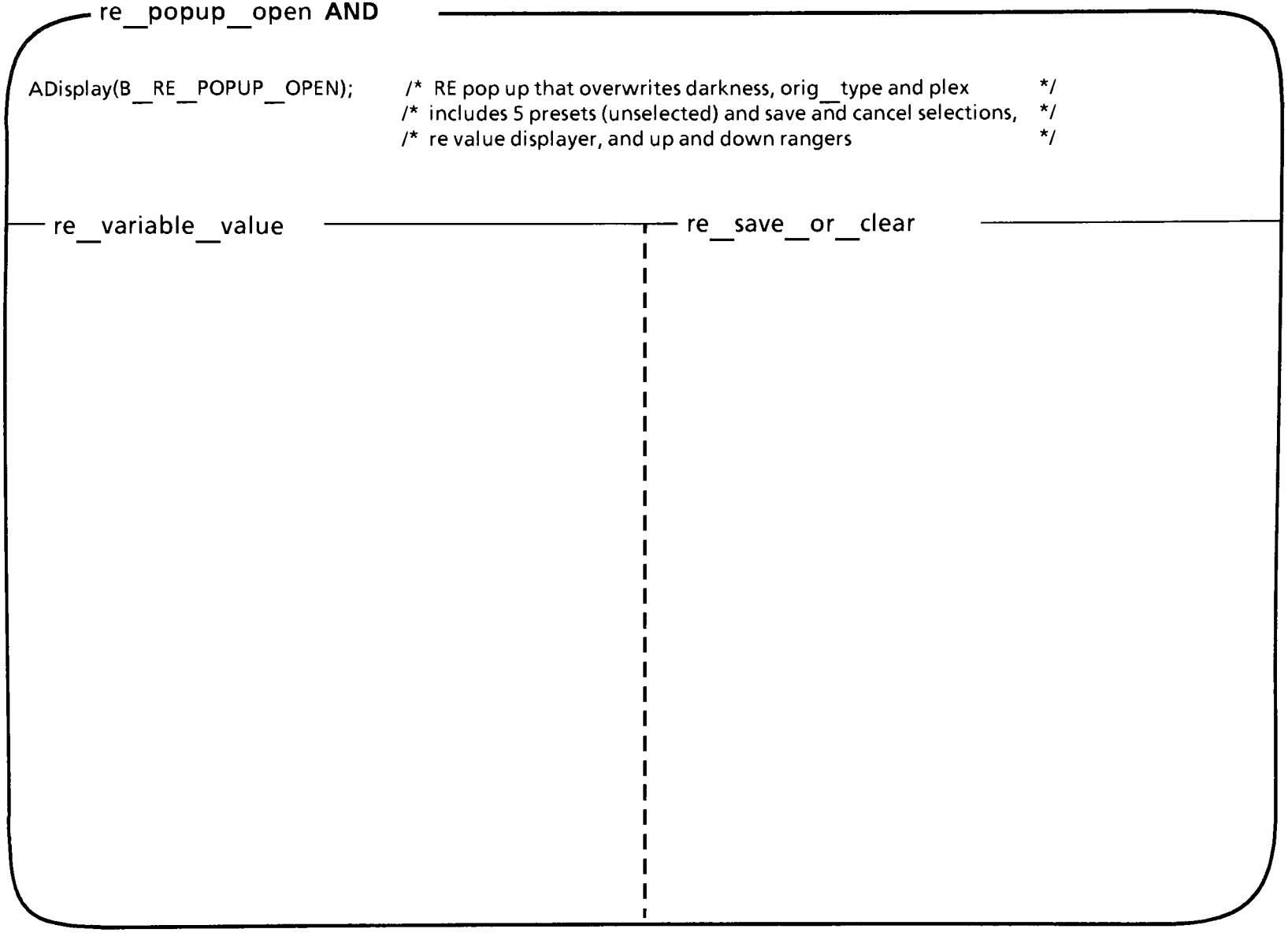

re_popup_closed AND
ADisplay(B_RE_POPUP_CLOSED);
darkness
original_type
copy_background_suppression
plex


darkness XOR
ASensitize(B_LEVEL1, e_level1);
ASensitize(B_LEVEL2, e_level2);
ASensitize(B_LEVEL3, e_level3);
ASensitize(B_LEVEL4, e_level4);
ASensitize(B_LEVEL5, e_level5);
ASensitize(B_LEVEL6, e_level6);
ADisplay(B_COPY_DARKNESS) ; /* Darkness column w/ 6 selections, */
/* all unselected */
e_level1
level1
ADisplay(B_LEVEL1_SEL);
ASetValue(GV_COPY_DARKNESS, LEVEL1);
ADisplay(B_LEVEL1_UNSEL);
level3
e_level3
ADisplay(B_LEVEL3_SEL);
ASetValue(GV_COPY_DARKNESS, LEVEL4);
ADisplay(B_LEVEL3_UNSEL);
ADeSensitize(e_level1);
ADeSensitize(e_level2);
ADeSensitize(e_level3);
ADeSensitize(e_level4);
ADeSensitize(e_level5);
ADeSensitize(e_level6);
/*There are really 6 levels, though only 2 are shown here. They */
/* all behave identically, except that level3 is the default. */


copy_background_suppression XOR
ADisplay(B_COPY_BACKGROUND_SUPPRESSION); /*This bitmap appears just above the Original Type buttons. */
ASensitize(B_COPY_SUPPRESSION_ON,e_copy_suppression_on); /* It consists of the two (unselected) buttons */
ASensitize(B_COPY_SUPPRESSION_OFF,e_copy_suppression_off); /* and a gray line to separate the two */
/* features. */
e_copy_suppression_on
copy_suppression_on
ADisplay(B_COPY_SUPPRESSION_ON_SEL);
ASetValue(GV_COPY_AUTO_BKG_SUPP, YES);
ADisplay(B_COPY_SUPPRESSION_ON_ UNSEL);
e_copy_suppression_off
copy_suppression_off
ADisplay(B_COPY_SUPPRESSION_OFF_SEL);
ASetValue(GV_COPY_AUTO_BKG_SUPP, NO);
ADisplay(B_COPY_SUPPRESSION_OFF_ UNSEL);
ADeSensitize(e_copy_suppression_on);
ADeSensitize(e_copy_suppression_off);


original_type XOR
ADisplay(B_COPY_ORIGINAL);
/* Original-type column with 2 selections, both unselected */
ASensitize(B_COPY_TEXT, e_text) ;
ASensitize(B_COPY_PHOTO,e_photo);
e_text
text
ADisplay (B_COPY_TEXT_SEL)
ASetValue(GV_COPY_IMAGE_QUAL,TEXT)
ADisplay (B_COPY_TEXT_UNSEL)
e_photo
photo
ADisplay (B_COPY_PHOTO_SEL)
ASetValue(GV_COPY_IMAGE_QUAL,PHOTO)
ADisplay (B_COPY_PHOTO_UNSEL)
ADeSensitize(e_text)
ADeSensitize(e_photo)


plex XOR
ADisplay(B__PLEX);
ASensitize(B__ONE__TO__ONE,e__one__to__one);
ASensitize(B__ONE__TO__TWO,e__one__to__two);
ASensitize(B__TWO__TO__ONE,e__two__to__one);
ASensitize(B__TWO__TO__TWO,e__two__to__two);
/* Plex column with 4 selections, all unselected */
one__to__one
ADisplay(B__ONE__TO__ONE__SEL);
ASetValue(GV__COPY__PLEX, ONE__TO__ONE);
ADisplay(B__ONE__TO__ONE__ UNSEL)
e__one__to__one
e__one__to__two
one__to__two
ADisplay(B__ONE__TO__TWO__SEL);
ASetValue(GV__COPY__PLEX, ONE__TO__TWO);
ADisplay(B__ONE__TO__ONE__ UNSEL)
two__to__one
ADisplay(B__ONE__TO__ONE__SEL);
ASetValue(GV__PLEX, ONE__TO__ONE);
ADisplay(B__ONE__TO__ONE__ UNSEL)
e__two__to__one
e__two__to__two
two__to__two
ADisplay(B__TWO__TO__TWO__SEL);
ASetValue(GV__COPY__PLEX, TWO__TO__TWO);
ADisplay(B__ONE__TO__ONE__ UNSEL)
ADeSensitize(e__one__to__one)
ADeSensitize(e__one__to__two);
ADeSensitize(e__two__to__one);
ADeSensitize(e__two__to__two);


re__popup__open AND
ADisplay(B__RE__POPUP__OPEN);
/* RE pop up that overwrites darkness, orig__type and plex */
/* includes 5 presets (unselected) and save and cancel selections, */
/* re value displayer, and up and down rangers */
re__variable__value
re__save__or__clear

re_variable_value **XOR**

```
ASensitize(B_RE_PREn, e_re_pren);      /* for n = 1 to 5 */
ASensitize(B_RE_UPARROW, e_re_uparrow);
ASensitize(B_RE_DNARROW, e_re_dnarrow);
AVDRedisplay(re_popup_displayer);
```

/* Note: parent displays unselected bitmaps */

```
e_re_uparrow → new ActionTable(4,
    ADisplay(B_RE_UPARROW_SEL);
    ARangerUp(re_popup_ranger, re_popup_displayer);
    ADisplay(B_RE_UPARROW_UNSEL););
e_re_dnarrow → new ActionTable(4,
    ADisplay(B_RE_DNARROW_SEL);
    ARangerDn(re_popup_ranger, re_popup_displayer);
    ADisplay(B_RE_DNARROW_UNSEL););
```

re_pre1

```
ADisplay(B_RE_PRE1_SEL);
AVDSet(re_popup_displayer,400);
```

```
ADisplay(B_RE_PRE1_UNSEL);
```

e_re_pre1

e_re_uparrow, e_re_dnarrow

re_nopre

re_pre3

```
ADisplay(B_RE_PRE3_SEL);
AVDSet(re_popup_displayer,77);
```

```
ADisplay(B_RE_PRE3_UNSEL);
```

e_re_pre3

```
ADeSensitize(e_re_pren); /* for n = 1 to 5 */
ADeSensitize(e_re_uparrow);
ADeSensitize(e_re_dnarrow);
```

```
/* Note: There are really five similar preset states, but only two are shown */
/* Their associated values are:                                              */
/* pre1 = 400, pre2 = 200, pre3 = 77, pre4 = 65, pre5 = 25                   */
```

re_save_or_clear

```
ASensitize(B_RE_SAVE, e_re_save);          /* Note: Parent displays bitmaps */
ASensitize(B_RE_CLEAR, e_re_clear);
```

Behaviors:

```
e_re_save_exit →   { AGetValue(GV_COPY_REDUC_ENLAR, val);
                     AVDSet(re_column_displayer, val);
                     ABroadcast(e_re_close_popup)
                   }
e_re_clear_exit →  ABroadcast(e_same_size);
```

```
ADeSensitize(e_re_save);
ADeSensitize(e_re_clear);
```

/* Selection of Save/Exit closes the popup and copies the value in the popup displayer to the R/E column displayer. Selection of Clear/Exit closes the popup and sets the R/E value back to the default, 100%. It does NOT set the R/E value back to what it was when the popup was opened. */

References

[Adca86] Ad-Cad, Ltd., "The Languages of STATEMATE," internal report, Weizmann Science Park, Rehovot, Israel, 1986.

[Baek87] Baecker, Ronald M., and Buxton, William A. S. "Readings in Human-Computer Interaction A Multidisciplinary Approach." Morgan Kaufmann, California (1987) 555-560.

[Bric87] Bricklin, Dan. "Dan Bricklin's Demo II Program User Manual," Software Garden Massachusetts (1987)

[Bett87] Betts, Bill, et. al. "Goals and Objectives for User Interface Software," Computer Graphics Vol 21, no 2 (1987) 73-78.

[Bigh87] Bigham, *et. al.* "The Trillium User Study," Xerox Accession No. X8700086 (1987).

[Gett86] Gettys, James, "Problems Implementing Window Systems in Unix," Usenix Proceedings, January 1986.

[Gold83] Goldberg, Adele, and Robson, Dave. "Smalltalk-80: The Language and its Implementation." Addison-Wesley, Massachusetts (1983).

[Good87] Goodman, Danny. "The Complete Hypercard Handbook." Bantam Books, New York (1987).

[Hare87a] Harel, David. "On the Formal Semantics of Statecharts," Proceedings of the 2nd Symp. on Logic in Computer Science (1987).

[Hare87b] Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, 8 (1987).

[Hend86] Henderson, Austin. "The Trillium User Interface Design Environment," in Proceedings of SIGCHI '85, April (1986) 221-227.

[Hill86] Hill, Ralph D. "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction The Sassafras UIMS." *ACM Transactions on Graphics*, 5, 3 (1986) 179 - 210.

[Jaco84] Jacob, Robert J. K. "An Executable Specification Technique for Describing Human-Computer Interaction," in Advances in Human-Computer Interaction, H. R. Hartson, ed. Ablex Publishing Co. (1984).

[Jaco85] Jacob, Robert J. K. "A State Transition Diagram Language for Visual Programming," *IEEE Computer* Vol 18, no. 8 (1985) 51-59.

[Jaco86] Jacob, Robert J. K. "A Specification Language for Direct Manipulation User Interfaces,"*ACM Transactions on Graphics*, 5, 4 (1986) 283-317.

[Kier83] Kieras, D., and Polson, P. G. "A Generalized Transition Network Representation for Intractive Systems." in Proceedings of CHI '83, Dec. (1983) 103 - 106.

[McCl89] McClure, Carma. "CASE is Software Automation." Prentice Hall, Englewood cliffs, New Jersey (1989).

[Myer87] Myers, Brad A. "Creating User Interfaces by Demonstration," PhD Thesis, University of Toronto May (1987).

[New86] Newmann, William M. "A System for Interactive Graphical Programming," Proceedings of the AFIPS Spring Joint Computer Conference (1986) 47-54.

[Olse86] Olsen, Dan R. Jr. "Mike: The Menu Interaction Kontrol Environment," *ACM Transactions on Graphics*, 5,4 (1986) 318-344.

[Schm86] Schmucker, Kurt J. "MacApp: An Application Framework," *Byte* August (1986) 189-193.

[Stou86] Stroustrup, Bjarne. "The C++ Programming Language," Addisson-Wesley (1986).

[Tane81] Tanenbaum, Andrew S. "Computer Networks," Prentice-Hall, NJ (1981).

[Wate86] Waterman, Donald A. "A Guide to Expert Systems," Addison-Wesley Reading, MA (1986).

[Wass85] Wasserman, Anthony I. "Extending State Tranition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 11,8 August 1985. 699-713.