GENERAL PURPOSE VISUAL SIMULATION SYSTEM

by

John Leslie Bishop

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for the degree of

Master of Science

in

Computer Science

APPROVED:

Osman Balci, Chairman

Richard E. Nance

J. Patrick Bixler

June, 1989

Blacksburg, Virginia

GENERAL PURPOSE VISUAL SIMULATION SYSTEM

by

John Leslie Bishop

Osman Balci, Chairman

Computer Science

(ABSTRACT)

The purpose of the research described herein is to prototype a software system that aids a simulationist in developing a general purpose discrete event simulation model.

A literature review has shown the need for an integrated visual simulation system that provides for the graphical definition and interactive specification of the model while maintaining application independence. The General Purpose Visual Simulation System (GPVSS) prototyped in this research meets this need by assisting a simulationist to: (1) graphically design the model and its visualization, (2) interactively specify the model's logic, and (3) automatically generate the executable version of the model, while maintaining domain independence. GPVSS is prototyped on a Sun 3/160C computer workstation using the SunView graphical interface. It consists of over 11,000 lines of documented code. GPVSS has been successfully tested in three different case studies that are described in this work.

ACKNOWLEDGMENTS

I would like to thank my beautiful wife for carrying the not inconsiderable burden of planning our wedding while I completed this work.

Many thanks are owed to Dr. Osman Balci for his undying support and technical guidance. This thesis would not have been possible without him. Dr. Richard E. Nance has provided many useful comments and suggestions that have greatly improved the quality of this work. Also thanks to Dr. Bixler for the graphics expertise he has given me.

I also wish to thank my coworkers and for their friendship and camaraderie. They made the long hours much more bearable.

Finally, I wish to thank the Veterans Administration Vocational Rehabilitation Program and my counselor for the support that made my education possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Statement of the Problem

Many visual modeling tools and packages have become available in recent years. Of the software currently on the market, most tend to be "either collections of FORTRAN subroutines, such as SEE-WHY, or purpose built languages such as Inter_SIM and Xcell" [Bell and O'Keefe 1987] or extensions to existing simulation languages. Examples of this approach include CINEMA which interfaces to the SIMAN simulation language [Johnson and Poorte 1988], and TESS which interfaces with the SLAM II simulation language [Standridge 1986]. RESQME [Gordon et al. 1988] provides a full visual simulation modeling environment, but forces the user to use such primitives as "Sources", "Sinks", "Chains", "Nodes", "Queues", etc. [Sauer 1982].

Clearly there is a need for an integrated visual simulation system that provides for the graphical definition and specification of the model while maintaining application independence.

1.2 Statement of Objectives

The purpose of this research is to prototype a software system that aids a simulationist in developing a general

purpose discrete event visual simulation model. The software system should assist a simulationist to: (1) graphically design the model and its visualization, (2) interactively specify the model's logic, and (3) automatically generate the executable version of the model.

1.3 Overview of Thesis

Chapter 2 presents a state-of-the-art review of Visual Simulation (VS). Chapter 3 describes in detail the conceptual framework and internal logic used in GPVSS. Chapter 4 contains three case studies and the verification/ validation of GPVSS. Chapter 5 fully documents the current use and functionality of GPVSS and contains a simple tutorial illustrating its use. Chapter 6 presents a summary of work accomplished and suggests areas for future research.

CHAPTER 2

LITERATURE REVIEW

2.1 Definition of Terms

Recent years have seen the increasing proliferation of discrete event simulation packages that include some facility for "animation", "Visual Simulation" (VS), "Visual Interactive Simulation" (VIS), "Visual Modeling" (VM), or "Visual Interactive Modeling" (VIM). The meaning of these terms to describe a dynamic display facility for simulation has become relatively ambiguous.

2.1.1 Animation

Animation refers to any graphic display of information where the information to be imparted to the viewer is conveyed by image change [Baeker 1974]. This includes many displays clearly outside the scope of simulation. Therefore it is inappropriate to restrict the definition of "animation tool" to that which provides a display "portraying the dynamic behavior of the system model" with some variable degree of user interaction [O'Keefe 1987].

A more apropos definition is offered by Standridge [1986]. The term animation as it applies to simulation refers to either simulation-concurrent or post-simulation animation. In simulation-concurrent animation, the

simulation is run "based on the state of the system as seen in the animation" [Standridge 1986] and allows the user to interact with and change model parameters. Post-simulation animation "emphasizes the presentation of the dynamics and structure of a system as captured by a simulation" [Standridge 1986] and does not allow the changing of model parameters by the user.

As the above illustrates, the term "animation" has been used to refer to a variety of dynamic simulation displays and degrees of user interaction. According to Mathewson [1985] "animation is a particular use of the topological information in a program generator". For this reason, it is desirable to restrict the scope of animation as it applies to simulation to refer solely to the generated dynamic display. Thus, a "simulation-concurrent animation" is defined as a dynamic display generated by the state of the system model, regardless of whether user interaction with the model is supported. Similarly, "post-simulation animation" is a dynamic display driven by a simulation trace and may or may not include facilities for user interaction.

2.1.2 Visual Simulation

VS is the process of building a model which permits a visual display of the dynamic behavior of the system under study and experimenting with this model on a computer for a specific purpose. The major difference between Discrete

Event Simulation and VS is the fact that the simulation model's input, internal behavior, and/or output are visualized and displayed on a computer screen. VS is a subset of the capabilities provided by VIS.

2.1.3 Visual Interactive Simulation

VIS includes all of the capabilities of VS, but adds the capability of the user to interact with the running model. This interaction may be either model or user determined, depending on who initiates the interaction [O'Keefe 1987].

Prompting the user to make some sort of scheduling decision is one example of model initiated interaction. In fact, the need for a user to know the current state of the modeled system when making such a decision is considered the impetus behind the first VIS efforts [Bell and O'Keefe 1987].

User initiated interaction allows the user to change model parameters and continue execution of the model. This ability to "play" with the model can be crucial for understanding the system. For example, the TRAVIS (Traffic Intersection Visual Interactive Simulator) [White 1988] allows the user to change such parameters as traffic light sequencing, interarrival times of automobiles, etc. and observe the effect on the modeled system produced by these changes.

2.1.4 Visual Modeling

The term Visual Modelling has been used relatively loosely in the literature, and refers generally to the process of building any form of Visual Simulation [Paul 1989]. Therefore VM is a component of VS and a subset of VIM.

2.1.5 Visual Interactive Modeling

The definition of VIM is restricted to the process of building a VIS model. VIM is a component of VIS, and contains all of VM.

2.2 Background

Although GPSS/Norden was created in 1973 and was the first documented work in VIS [Nance 1989], Bell and O'Keefe [1987] report that the first significant work in VIS was conducted by Hurrion at the University of Warwick in England. Hurrion was trying to simulate a job shop manufacturing system. The difficulty in accomplishing this lay in the fact that often a human scheduler had control over the system. Attempts to model the scheduling process with traditional modeling techniques were unsatisfactory. Therefore it was decided to implement a "man-in-the-loop" model. In order for the human scheduler to make realistic decisions, it was necessary for the scheduler to be able to know the current state of the model. To accomplish this, an

iconic visual display with letters depicting entities was created. From this and subsequent work, Hurrion introduced the term "Visual Interactive Simulation".

Since its inception, VS/VIS has become an increasingly popular method of problem solving [Paul 1989]. Graphical symbology, movement depiction, display type, and the degree of user interaction vary greatly from one system to the next.

2.3 Graphical Symbology

Currently there are a variety of methods used to represent the state of a system model visually. These methods consist of keyboard characters, icons, or three dimensional rendering [Paul 1989].

Keyboard characters are the simplest method of representing model objects. The use of keyboard characters is inexpensive in both hardware and development time. Hardware costs are lower because no special graphics hardware is required to display the character image. Development time for creating the object image is lower because the object images (characters) already exist. The tradeoff is that the diversity of model object images that can be employed is extremely limited, and one may feel constrained in adequately conveying the diverse objects in a system visually. In addition, object movement representation is crude at best.

Icons are a more faithful representation of system model objects. Icon images may be pre-defined, defined by the user, or generated automatically [Paul 1989]. The use of iconic representational graphics is typically more hardware- and development-intensive due to the need for special purpose graphics hardware and the time required to create the icon image. Despite the additional overhead, the use of icons to represent system model objects greatly enhances the quality of the visual simulation.

If desired, the visual simulation may be further enhanced by the addition of animation. Animation may be in the form of icon movement, or icon change. In icon movement animation, the icon is dynamically drawn on the screen with time, position, movement, and speed attributes analogous to the object being represented in the system. Icon change animation occurs when the animation is implicit in the varying icon image. Thus it would be possible to show a customer moving from a queue to a server, and show him walking at the same time. The overhead associated with icon change animation is considerable since it requires multiple icon images for each object to be animated and requires substantially more skill and effort of the modeler.

Finally, three dimensional rendering can be used for maximum realism. Techniques range from simple polygon projection to photo-realism. The primary drawback of these methods is that they are currently not able to run in real

time due to hardware considerations. However, with such technical advances such as the Intel 80810 RISC architecture CPU, which features on-chip Phong and Gouraud three dimensional shading, the prevalence of this technique is sure to increase.

2.4 Display Type

Hurrion [1979, 1986] classifies display types into schematic, logical, and null displays. Schematic displays attempt to parrot the system being modeled. Typically they have a detailed blueprint or schematic diagram as a background over which icons representing objects in the system move. This approach is particulary suitable for creating dynamic displays of relationships among objects where a spatial or simple logical relationship exists.

Logical or summary displays take the form of bar charts, histograms, time series, etc.. Use of this type of dynamic display is appropriate when relationships among system model objects are extremely complex or when use of a schematic display is inappropriate. In large complex models it may be more useful to present the user with a logical display containing summary statistics rather than observing a schematic display of the model.

Null displays enable the end user to run the simulation without the performance degradation associated with a dynamic visual display. This method is similar to executing

a conventional simulation model.

2.5 Interaction

The degree of interaction that VIS provides users is perhaps one of its most significant benefits. Within VIS interaction varies greatly in the type, level and methods for incorporating interaction.

2.5.1 Types of Interaction

There are two means by which a user may interact with a running model: model prompted and user prompted. Model prompted interaction occurs when the simulation itself prompts the user for input. An example of this method in Hurrion's seminal visual simulation of a job shop manufacturing system where the scheduler was prompted to make scheduling decisions. User prompted interaction is characterized by the user's ability to specify when the interaction is to occur. This method enables the user to change system model parameters and continue execution of the model.

2.5.2 Levels of Interaction

Originally Hurrion categorized three levels of interaction [Hurrion and Secker 1978]: basic operations, priorities, and algorithms in order of increasing levels of interaction.

Basic operations consist of interactions that effect one-to-one changes in the system model. Any entity or any attribute of any entity on the dynamic display may be modified, deleted, moved, or replicated and the corresponding modification is incorporated into the system model.

Priority interactions modify the priorities of operations, jobs, or entities. This form of interaction gives the user the capability of assigning dynamic priorities to entities for the balance of the simulation run.

Algorithm interactions modify the various driving algorithms of the simulation model. Using algorithm interaction, an analyst would be able to create "algorithmic modules" [Hurrion and Secker 1978] which could be combined in various ways to produce the desired system model.

Hurrion [1980] later combines priority and algorithm interaction into a new level called structural interaction.

2.5.3 Methods for Incorporating Interaction

O'Keefe [1987] outlines three methods for handling both model- and user-initiated interaction: embedded programming, standard interactions, and stopping interpretation.

The embedded programming method incorporates all display generating code into the model itself. This method

provides a great deal of flexibility, but requires much more development effort and offers little reusability.

Providing a library of standard interactions is a step forward. Using this method allows the modeler to pick and choose amongst a set of pre-defined interactions thereby decreasing development time. If need be, some standard interaction libraries may be extended. OPTIK is an example of such a library [O'Keefe 1987].

Stopping interpretation is perhaps the most powerful method by which a user may interact with the simulation model. As the name implies, the underlying code of the simulation model must be interpreted in order to utilize this method. When stopping interpretation, it is possible to actually alter the simulation model code. While this may provide great flexibility, it does require the user to have some knowledge of the code being modified.

2.6 Construction of VS/VIS Models

Two methods of VM/VIM currently exist. The first technique separates design of the model and the dynamic display [Macintosh et al. 1984; Hurrion 1980]. Hurrion [1980] explicitly divides the VM/VIM process into construction of the simulation model and design of the display and interaction facilities. This has the effect of forcing the modeler to construct two models: a model of the system, and a visual display which is in effect a model of

the model.

An alternative method is to provide the user with the capability to create the system model and specify the dynamic display at the same time, so that a one-to-one correspondence exists between objects in the display and objects in the model [Gordon and MacNair 1987; Bell and O'Keefe 1987]. Unfortunately most of these systems have suffered from a distinct lack of application independence.

2.7 Conceptual Frameworks for VS Programming

Activity scan, event scheduling, three-phase, and process interaction approaches have all been used in VS/VIS. Hurrion [1980] advocates the activity scan conceptual framework because it provides significant "advantages for interaction at run time". Provided the model is designed such that a one-to-one relationship exists between objects in the model and objects in the dynamic display, it is a relatively easy task to change the activity tests.

Because the event scheduling and three-phase conceptual frameworks are event based, animation of state changes in the system model is relatively straightforward. However, if user interaction with the model is desired, several difficulties arise. Since the simulation always leads the animation display by some amount, the additional burden is placed on the simulation of continuously keeping track of the model state at the current animation time. Also, any

future events that have been calculated must be re-calculated using the new system model parameters.

Process interaction has the advantages of the activity scan approach and the disadvantages of event based conceptual frameworks. Since the process interaction conceptual framework is object based, changes in attribute values of objects in the dynamic display can be reflected quite easily in changes in the appropriate system model object attributes. On the other hand, objects experiencing delays which are time-based and unconditional must be handled using a future event set [Derrick 1988] which must be handled in a manner similar to that discussed for event-based conceptual frameworks.

Ultimately, the choice of conceptual frameworks is dependent on the degree of user interaction desired and the nature of the system being modeled.

2.8 Perspectives on VS/VIS

O'Keefe [1987] defines five separate views of VS/VIS: statistical, decision support, computer aided design (CAD), gaming, and simulator.

The statistical or traditional perspective treats VS/VIS as a selling aid. Statistical experimentation is the primary means of decision support. Little or no provision is made for the user to interact with the system model as it is running. This perspective is almost mandated by the use

of post-simulation animation, since any significant degree of user interaction is ruled out.

Decision support perspective views VS/VIS as a decision support tool to solve unstructured problems. Consequently much emphasis is placed on providing the user with facilities to interact with the system model. Use of a standard set of interactions may limit the usefulness of a VS/VIS system under this perspective.

The CAD perspective perceives VS/VIS as a tool for designing a system by combining either pre- or user-defined parts. The CAD perspective is particularly appropriate when an object-oriented approach is used. Users can create instances of objects and place them in the system model. The CAD perspective is prevalent with VS/VIS users involved in manufacturing system design.

Gaming perspective views VS/VIS as a tool for learning as opposed to analysis of results.

The simulator perspective incorporates VS/VIS as a rudimentary simulator with the user as the "man-in-the-loop".

The perspectives described above are sufficient to describe the domain of capabilities of VS/VIS when taken as a whole. However, no single perspective currently exists to adequately encompass VS/VIS.

2.9 Benefits of VS/VIS

Many benefits are associated with VS/VIS [Bell and O'Keefe 1986; O'Keefe 1987]. O'Keefe states that VS/VIS gives the client and developer three additional capabilities not found in traditional simulation methods in the areas of selling, gaming, and learning [O'Keefe 1987].

VS/VIS augments selling by providing both a communication and a presentation medium [O'Keefe 1987]. During model development the dynamic display becomes a communication medium that enables the modeler and the client to discuss model validation, development, and experimentation. The dynamic display becomes a presentation medium for the presentation of model results obtained using VS/VIS or by traditional statistical experimentation.

VS/VIS enhances gaming capabilities by making it possible to use gaming with relatively complex system models. The need for user interaction with the running system model was the impetus behind Hurrion's original job shop visual simulation [Bell and O'Keefe 1987].

Learning capabilities are enhanced because the interactive capabilities of VIS allow the client to "play" at managing the system [Bell and O'Keefe 1987]. This allows the client to gain knowledge pertaining to management of the system.

In addition to these augmented capabilities, the following are generally accepted benefits that VS/VIS

provides.

The dynamic display associated with VS/VIS augments model credibility through the enhanced presentation of simulation results. This is due to the fact that the client need not have an extensive simulation background to understand model results [O'Keefe 1987]. In a survey by Bell and Kirkpatrick [1986] seventy percent of the respondents felt that their decisions were implemented more often when VS/VIS was used. Although Bell and Kirkpatrick [1986] themselves admit the validity of their survey is in question, clearly VS/VIS has had some beneficial effect on model credibility for some.

VS/VIS enhances model verification and validation. This is because the analyst can easily see the effect of incorrect behavior when the system model is not working correctly [Paul 1989]. In addition, the communication medium that VS/VIS provides enables the client to participate to a greater degree in the verification/ validation process. In the same survey by Bell and Kirkpatrick [1986] eighty-eight percent of the respondents said that validation was either "faster" or "much faster" when using a VS/VIS system.

One additional benefit of VS/VIS is that the client can be involved in model development at a much earlier stage than is otherwise possible. This allows the developer to tailor the model much closer to the clients needs, and to

possibly avoid costly misunderstandings. In addition, the client may soon know enough about the system to ask questions that would not have previously been thought of [Palme 1977].

2.10 Drawbacks of VS/VIS

VS/VIS has definitely advanced the state of the art in discrete event simulation. However, VS/VIS does have some undesirable characteristics that the analyst and client should be aware of before choosing to construct a VS/VIS.

It is difficult to estimate the cost/benefit ratio of developing a VS/VIS simulation [Bell 1985a]. While increased costs are certain, the benefits of implementing VS/VIS are much less tangible.

VS/VIS tends to be hardware specific [Bell 1985a] due to graphics requirements. Furthermore, the choice of hardware determines the graphics capabilities available to the modeler. While increased complexity may or may not be desirable, this does have the effect of placing a limit on the maximum complexity of the model display.

The use of VS/VIS introduces further requirements for the construction and presentation of model results that would not be present for conventional simulation techniques.

In addition to constructing the system model, the analyst is now responsible for creating a dynamic display. Currently, a good screen design methodology has yet to be

developed within the simulation community. Consequently, the modeler must also be substantially familiar with graphics design and programming. Some aspects of model behavior may be difficult or impossible to portray visually. Paul [1989] presents an example of a port model that easily portrays incoming ships entering their berths. However, the assignment of a given ship to a berth, the primary impetus behind the simulation, is dependent upon many different attributes such as cargo, berth capabilities, and ship type. Paul [1989] states "Any attempt to represent visually such rules would probably be self-defeating" due to the resultant complexity in the dynamic display.

The use of a dynamic display as a presentation medium may require the modeler to add more detail to the system model than actually required to obtain the desired statistical results. Bell [1985a] cites an example of a truck dispatching system model. If the system road network is dense, it may not be material which route a truck actually takes to arrive at a given point. The client, however, may not fully accept the model as credible without actually seeing which route the truck takes.

A dynamic display is an aid in model verification/ validation, not a means in itself. Paul [1989] makes the observation that "undue confidence" may be placed in the model merely because it "looks alright". Bell [1985a] warns that since VS/VIS displays the model transient period in

great detail, and since the client can observe this transient behavior in the system every day, there is a distinct possibility that the client may assess model credibility based on the transient period rather than steady state.

The client must be aware that interacting with the running system model by changing model parameters invalidates the usual assumptions associated with steady-state analysis of model output. The danger exists that the client views a "snapshot" of model behavior and assumes that the system always exhibits these characteristics [Paul 1989]. The use of a dynamic display does not abrogate the need of the modeler to instruct the client about the nature of statistical methods.

Paul [1989] begins to address one area that has been distinctly lacking in VS/VIS literature - human factors. The expressions "seeing is believing" and "a picture is worth a thousand words" immediately come to mind when discussing VS/VIS. Yet within the legal community eyewitness evidence is considered the least reliable form of evidence. Paul [1989] states that "Dreams, wishes, desires and thoughts" can affect an individual's perceptions. Another human factors consideration is the choice of colors used in the dynamic display. Red and green are two of the most widely used colors in VS/VIS but roughly ten percent of the total population is unable to distinguish these due to

color blindness [Paul 1989]. Thus, considerable attention must be paid by the modeler to established human factors guidelines when designing a VS/VIS dynamic display.

2.11 Conclusions

The use of VS/VIS has many positive effects on the processes of model validation, verification, and acceptability of model results. There are some drawbacks associated with VS/VIS however. Not the least of which is the lack of a comprehensive VS/VIS methodology to guide modelers in the construction of a VS/VIS. As the quality and quantity of VS/VIS systems increase, the impact of human factors considerations is increasingly to be felt. The onus is on the modeler to determine when, where and how to implement VS/VIS.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF GPVSS

The General Purpose Visual Simulation System (GPVSS) has been developed to enable a simulationist to interactively define and specify an executable discrete event visual simulation model of the system being studied. It should be noted here that all user interface terminology conforms to the SunView user interface standard [Sun 1988b]. The top-level menu of GPVSS is shown in Figure 3.1.

At the highest level, GPVSS is comprised of three main components: the Image Editor/Model Editor, the Model Generator, and the Model Animation Tool. The purpose of the Image Editor/Model Editor is to provide the user with facilities to graphically define and interactively specify a model of the system being studied. The Model Generator utilizes the model definition and specification provided by the Image Editor/Model Editor to create an executable visual simulation model. Finally, the Model Animation Tool creates a dynamic display of the running simulation.

3.1 Hardware and Software Environment

GPVSS was implemented on a SUN 3/160C[1] color

---

[1] SUN 3/160C is a trademark of Sun Microsystems, Inc.

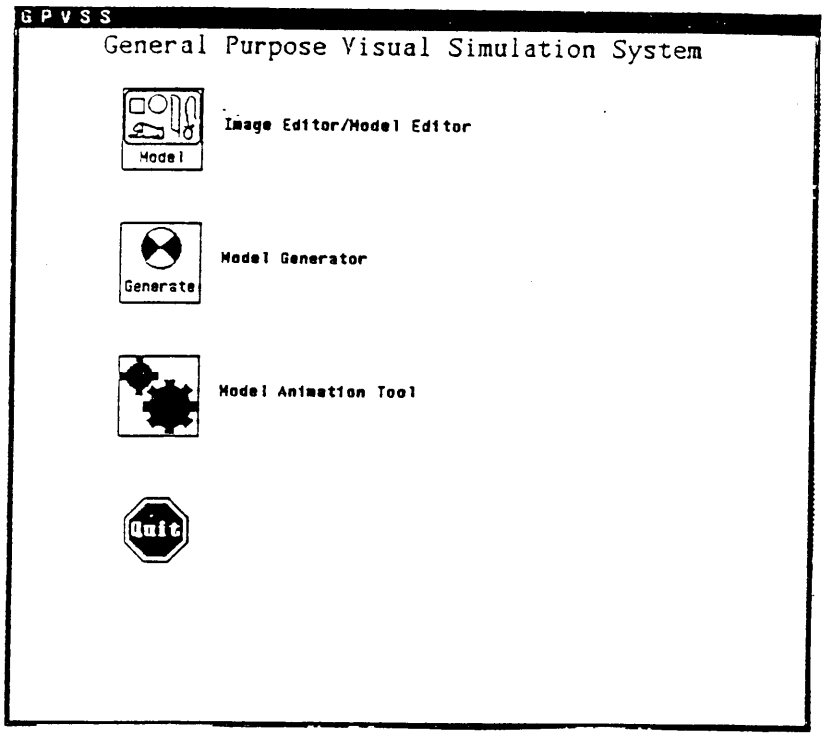

Figure 3.1 GPVSS

workstation running the SUN UNIX operating system. The SUN is a 2-MIPS machine consisting of a 16.67-MHz MC68020 microprocessor with a 16.67-MHz MC68881 floating-point co-processor, a 380-MB Fujitsu Eagle disk subsystem, a 1/4-inch cartridge tape subsystem, 8-MB of main memory, a 19-inch color monitor with a resolution of 1152 X 900 pixels, a pointing device called a mouse, and a connection to the Ethernet network which enables high speed file transfer to and from other University computing systems.

The most significant element of the user interface is the 19-inch bit-mapped display screen. This display technology offers excellent graphics capabilities.

The user communicates with the SUN via keyboard input and the mouse. The mouse may be used to point to or select locations on the screen. The system provides feedback on the current mouse location by continuously updating the position of the mouse pointer. Virtually any material that is displayed on the screen can be pointed to and treated as input.

The windows are the basic building block of the user interface and are roughly analogous to sheets of paper on a desktop. Windows may be resized, overlapped, closed, or quit. Resizing a window allows the user to effectively increase the available workspace [Sun 1988b, p. 35]. Overlapping is useful when the user needs to interact with multiple windows, but there is not enough room to display

all of them. Closing reduces a window to a small 64 X 64 bit image known as an "icon". Icons are useful when a window contains a process that does not require immediate user interaction, but it is still desirable to keep that process running (shell, for example). Quitting a window destroys the window entirely when there is no further use for it.

Menus and buttons are perhaps the most important window features in terms of the user interface. Menus provide a user with the ability to select an option from a finite list of choices. Menus may be "pop-up", requiring the user to depress a mouse button before they are active. Buttons allow the user to interact in a more limited way with the running program. Selecting a button requires the user to position the mouse cursor over the button image and depress a mouse button. This action executes a C function that has been associated with the button by the programmer.

The GPVSS code consists of approximately 11,000 lines of documented SunView, C, and EQUEL/C code. All window features are programmed using the SunView application package, which consists of high-level routines to create a graphical user interface and window management system [Sun Microsystems 1988]. The databases are created using the INGRES relational database management system [Sun Microsystems, 1986]. EQUEL (Embedded QUEry Language)/C was used to access INGRES.

Although many modern systems have the hardware capability to support GPVSS, the SunView interface technology has proved invaluable for its rapid development.

3.2 GPVSS

To define or specify a model, the Image Editor/Model Editor tool is used. Selecting the "Image Editor/Model Editor" Icon executes the pre_objed() function which sets up a blocking requester that queries the user to input the name of the model to be defined and specified (see Figure 3.2). This name must be unique and conform to C identifier conventions. Selecting the "Construct Model" button calls pre_objed() which forks the objed program with the model name obtained from the blocking requester passed as a command line argument (see Figure 3.3).

The user may exit the model_frame without naming a model by selecting the "Exit" button. This event executes the exit_proc() function which merely contains a window_return() call to break out of the blocking requester.

A similar process is used to call the Model Generator. Selecting the "Model Generator" icon executes the pre_gen() function which brings up the frame model_frame mentioned previously (see Figure 3.4) [Sun 1988b, p. 207]. Instead of the "Construct Model" button however, the user is presented with a "Generate Code" button. If selected, this button executes call_gen(). Call_gen() takes the model name found

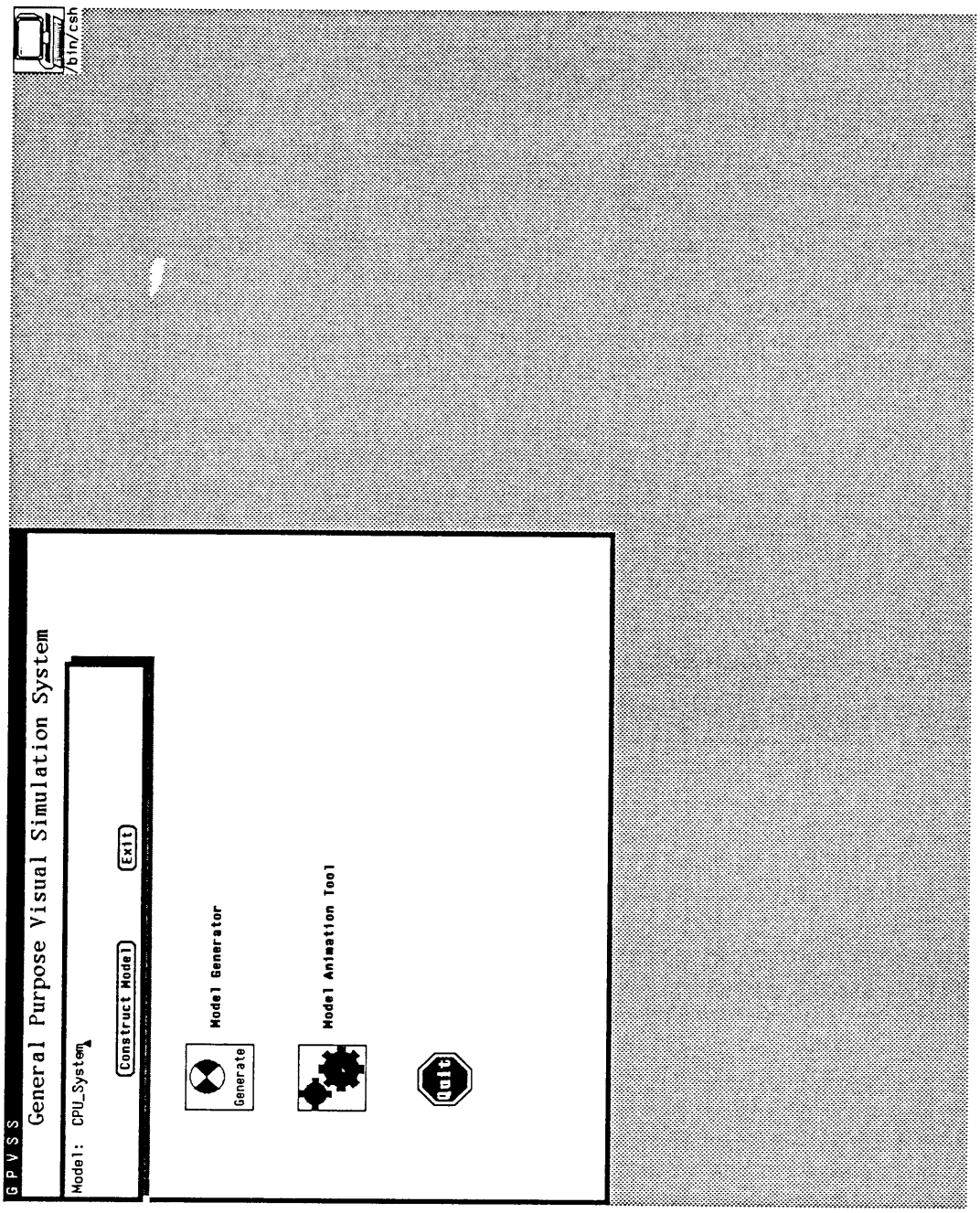

Figure 3.2 Model Name Requester for Editor

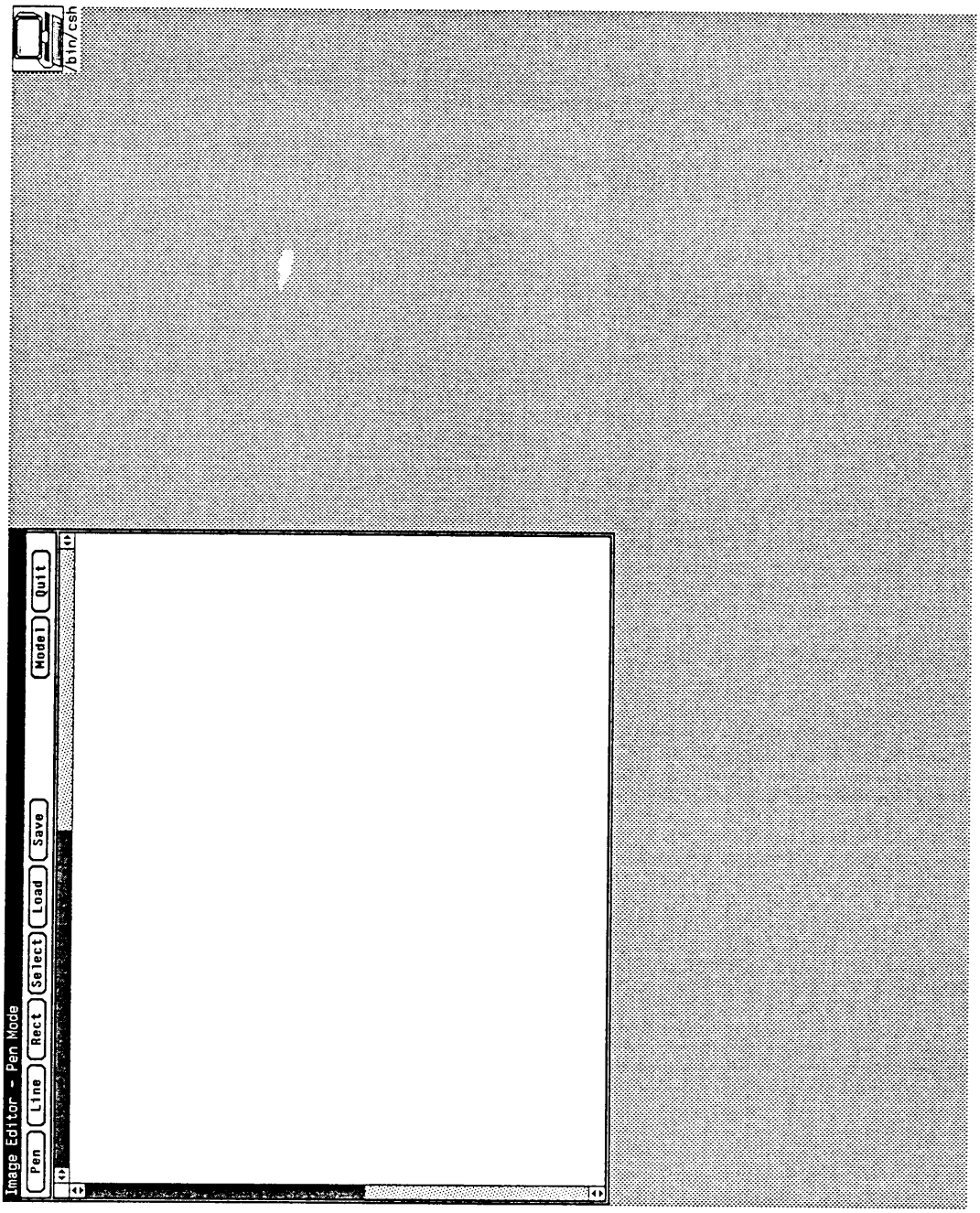

Figure 3.3 Editor Initial Screen

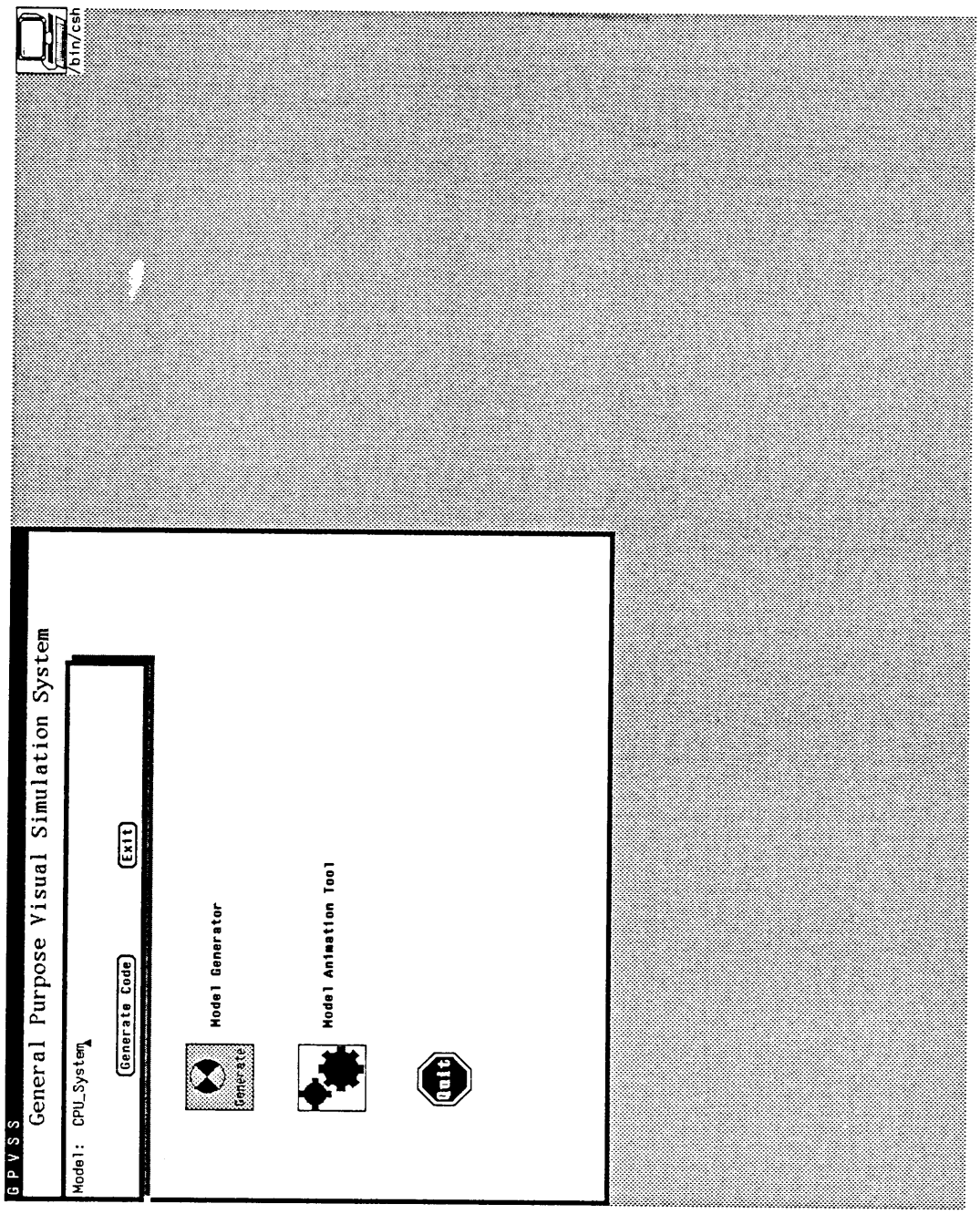

Figure 3.4 Model Name Requester for Generator

in the Panel_item mname_item and passes it as an argument to the gen program. The gen program is called using the system() function and not forked off as is the objed program. The "Generate Code" button remains selected until code generation is complete (see Figure 3.5). This is done because the gen program creates special files in its own directory that are model specific. Therefore, only one model can be executing at any time until the file handling capabilities of the gen program are improved.

The "Exit" button can be used as described previously to exit the model_frame without generating code.

Selecting the "Model Animation Tool" executes the call_anim() function which forks off the pi (process interaction) program created by the model generator tool (see Figure 3.6). It is important to only select this item once as the SunView windowing system queues mouse events and the user could wind up with multiple executing pi programs.

Selecting the "Quit" button calls quit_proc() which closes down the fonts, destroys the Frame base_frame and exits the program.

3.3 The Image Editor/Model Editor

The Image Editor/Model Editor is designed to provide the modeler with the capability of: (1) drawing a graphical image whether it be the model background or a dynamic object image and (2) specifying the model logic interactively.

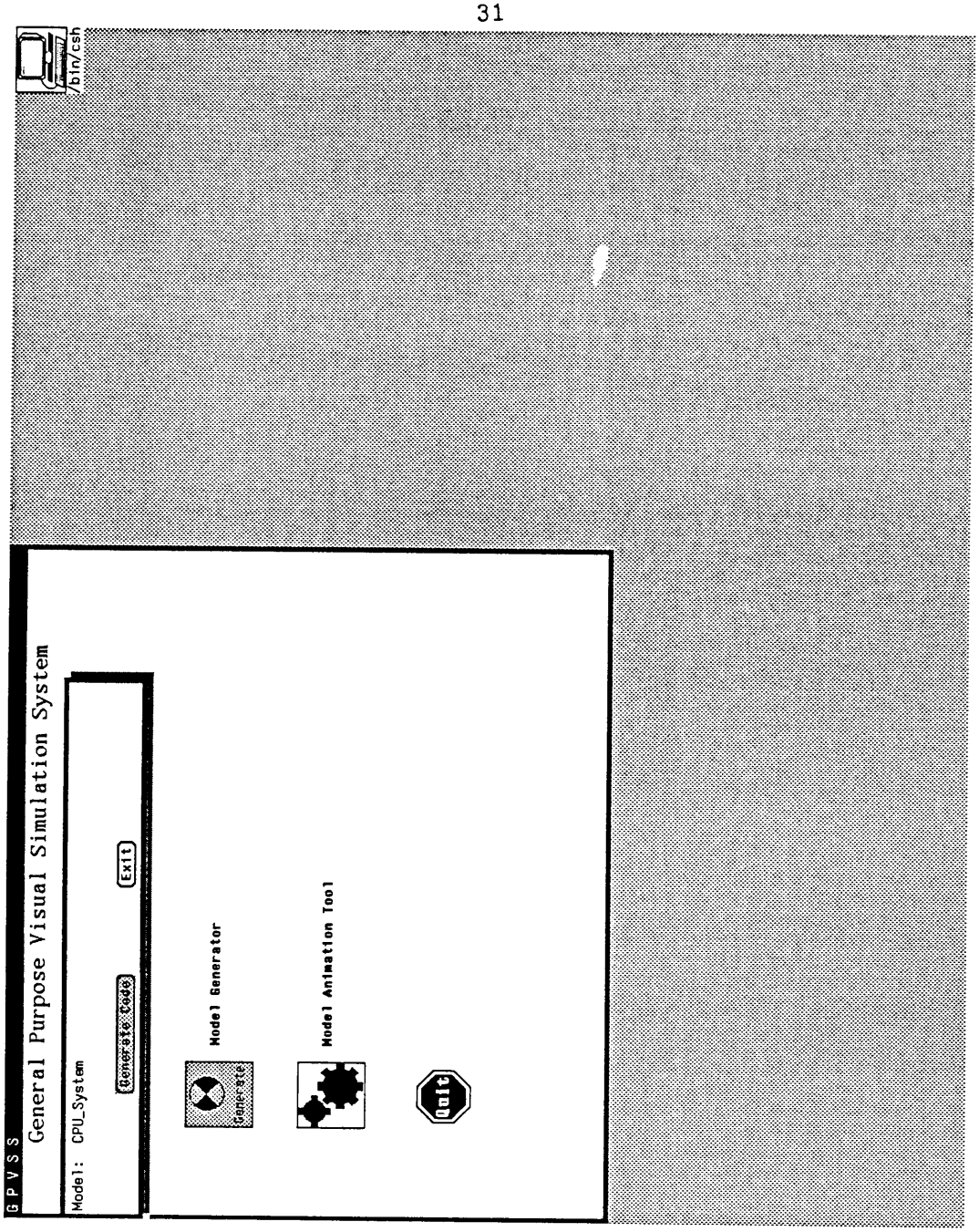

Figure 3.5 Model Generator Execution

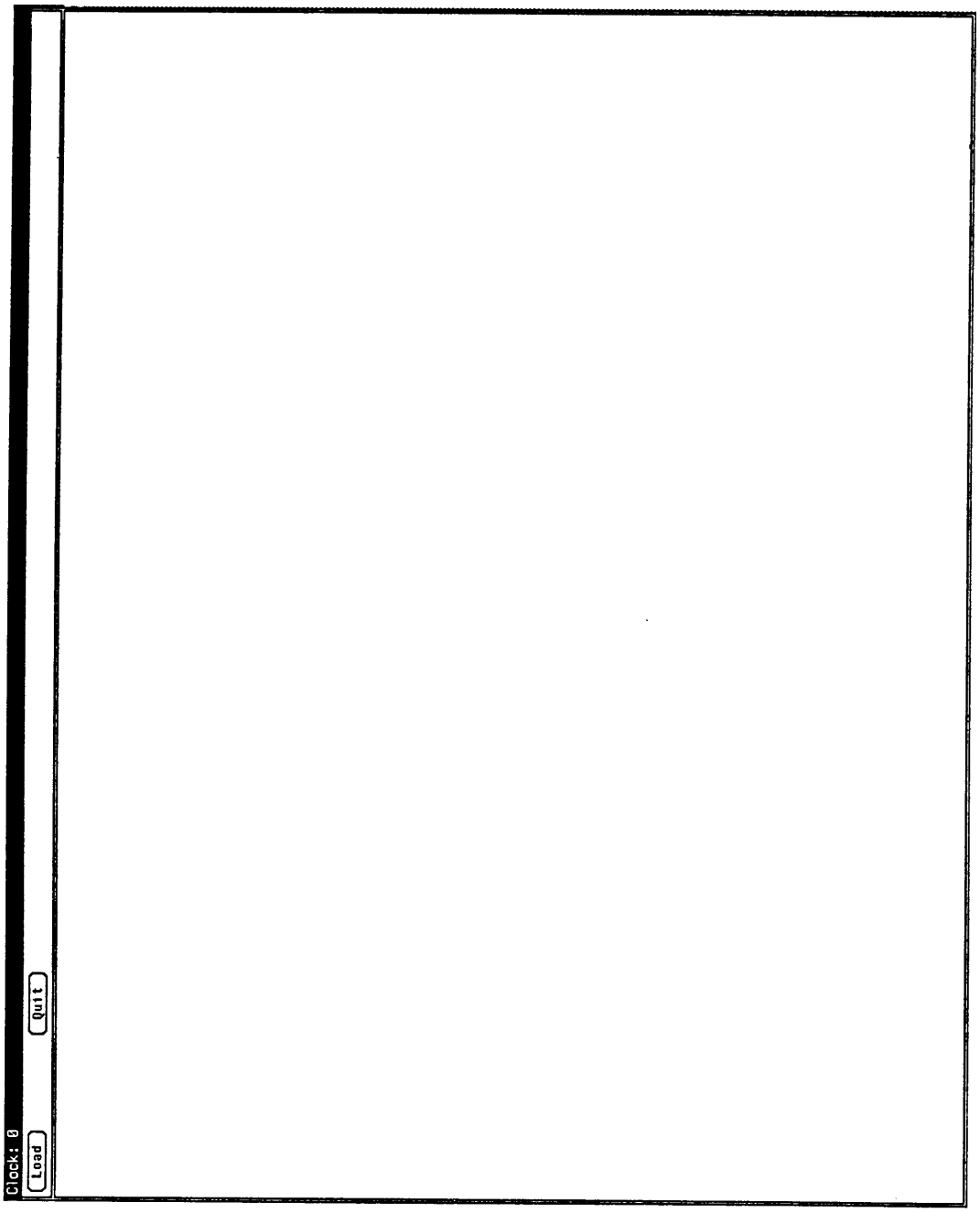

Figure 3.6 The Process Interaction Program Screen

3.3.1 The Image Editor

The Image Editor consists of a 1150x900 scrollable canvas with a row of buttons situated directly above it (see Figure 3.3) [Sun 1988b, p. 207]. The current mode of the Image Editor is displayed as the frame label at the top left corner of the frame. The button panel consists of buttons labelled "Pen", "Line", "Rect", "Select", "Load", "Save", "Model", and "Quit" [Sun 1988b, p. 208]. The following is a detailed discussion of the functionality and implementation that each of these buttons provides.

It should be noted here that the "Quit" button is active any time there is not a blocking requester. Selecting this button calls function quit_proc(). This function de-allocates memory and quits the program.

3.3.1.1 Pen Mode

Pen mode is the default mode of the Image Editor. In this mode it is possible to draw freehand using the mouse with the left mouse button depressed (see Figure 3.7). Pressing the middle mouse button while in pen, line, or rectangle mode erases the pixel currently being pointed to by the mouse pointer. This can be used as a limited erase capability.

Selecting the "Pen" button calls the function pen_proc() of the objed program. Pen_proc() puts the editor into pen mode and changes the SunView event handling

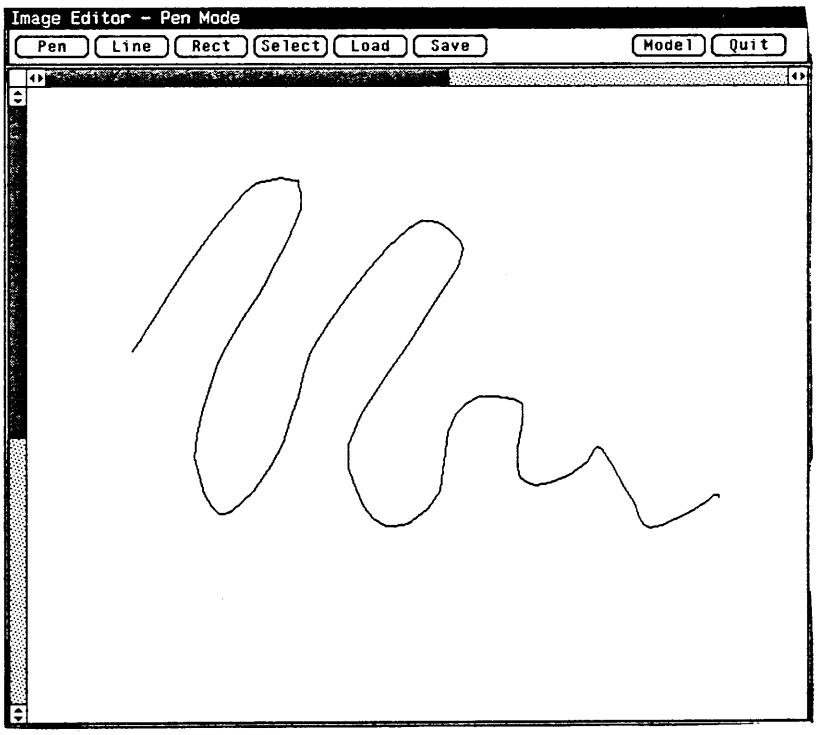

Figure 3.7 Image Editor Pen Mode

function to top_event_proc().

Top_event_proc() executes all events pertaining to pen, line, and rectangle modes. In pen mode, top_event_proc() utilizes pw_put() to set the pixel at the current mouse pointer position when the left mouse button is depressed. Thereafter, pw_vector() draws a line from the last mouse position to the next mouse position as long as the left mouse button remains depressed. This means that curves can be drawn as long as the mouse is not moved very quickly. If the mouse is moved rapidly, the freehand drawing begins to take on a segmented aspect due to the large change in mouse pointer coordinates between SunView events. This design decision was made in order to prevent the occurrence of "dotted" lines when using rapid mouse movements.

The pen, line, and rectangle modes also support an erase feature as mentioned above. While the middle mouse button is depressed, pw_put() is used to clear the color value at the present mouse pointer coordinates. No provision for fast mouse movement is supported as when using the left mouse button. Therefore fast mouse movement results in some pixels being skipped while erasing.

3.3.1.2 Line Mode

Line mode is entered by selecting the "Line" button on the Image Editor control panel and is used to draw lines on the canvas between any two points (see Figure 3.8). The

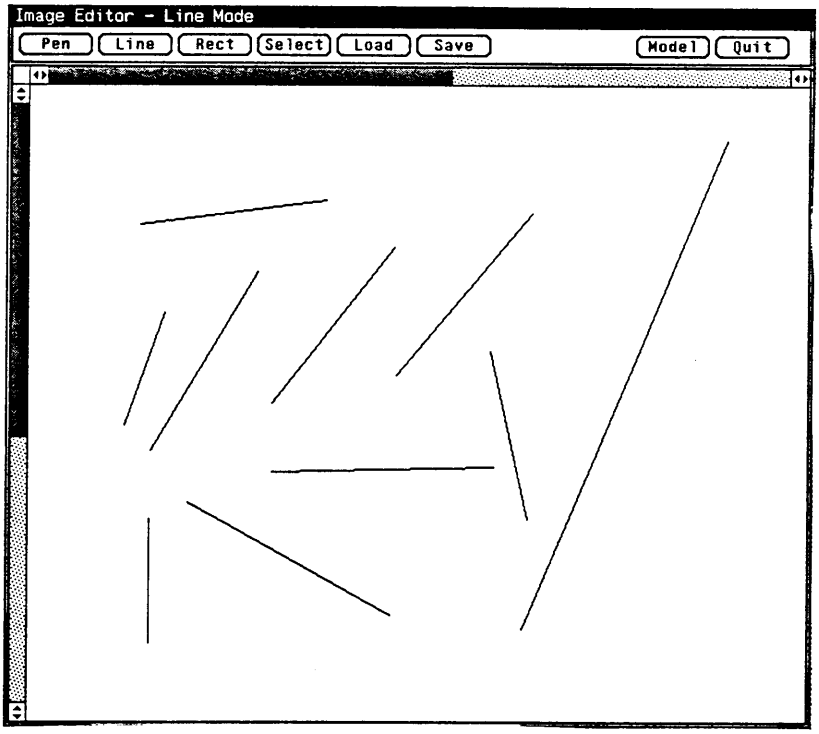

Figure 3.8 Image Editor Line Mode

first point is selected by moving the mouse pointer to the desired position and depressing the left mouse button. While the button remains depressed, the user may move the mouse pointer anywhere on the canvas and a continuously updated line between the first point and the current mouse pointer coordinates is drawn. When the line is positioned correctly, releasing the left mouse button freezes the line in its current position.

The line mode is implemented similarly to pen mode in many respects. Function line_proc() of the program objed is executed when the "Line" button is selected. This function places the editor in line mode and the frame's SunView event handling function to top_event_proc().

When the left mouse button is initially depressed, the pixel at the current coordinates is set with pw_put() just as in pen mode. Mouse drag events are handled differently however. Instead of drawing a line from the mouse's old position to the new position as in pen mode, two lines are drawn using an exclusive-or (XOR) logical mask. The first line is from the original mouse position to the mouse's last reported position. The second line is from the original mouse position to the current mouse coordinates. Since the logical mask being used is XOR, drawing the same line twice has the effect of erasing it. Thus while the mouse is dragged with the left mouse button depressed, the editor continually erases the line at the old mouse position and

draws a new line at the current mouse position.

It should also be noted that attempting to draw two lines on top of each other results in the second erasing the first where they overlap.

3.3.1.3 Rectangle Mode

Rectangle mode is entered by selecting the "Rect" button on the Image Editor control panel and is used to draw rectangles on the canvas between any two opposite corners (see Figure 3.9). The first corner is selected by moving the mouse pointer to the desired position and depressing the left mouse button. This corner remains fixed for the remainder of the rectangle drawing process. While the button remains depressed, the user may move the mouse pointer anywhere on the canvas and a continuously updated rectangle with opposite corners at the first point and the current mouse pointer coordinates is drawn. When the rectangle is positioned correctly, releasing the left mouse button freezes the rectangle in its current position.

Implementation of the rectangle mode is virtually identical to line mode. Rec_proc() in the objed program puts the editor in line mode and sets the SunView event handling function to top_event_proc(). Instead of drawing single lines however, rec_draw() is called to draw four lines that comprise a rectangle with opposite corners at the specified coordinates.

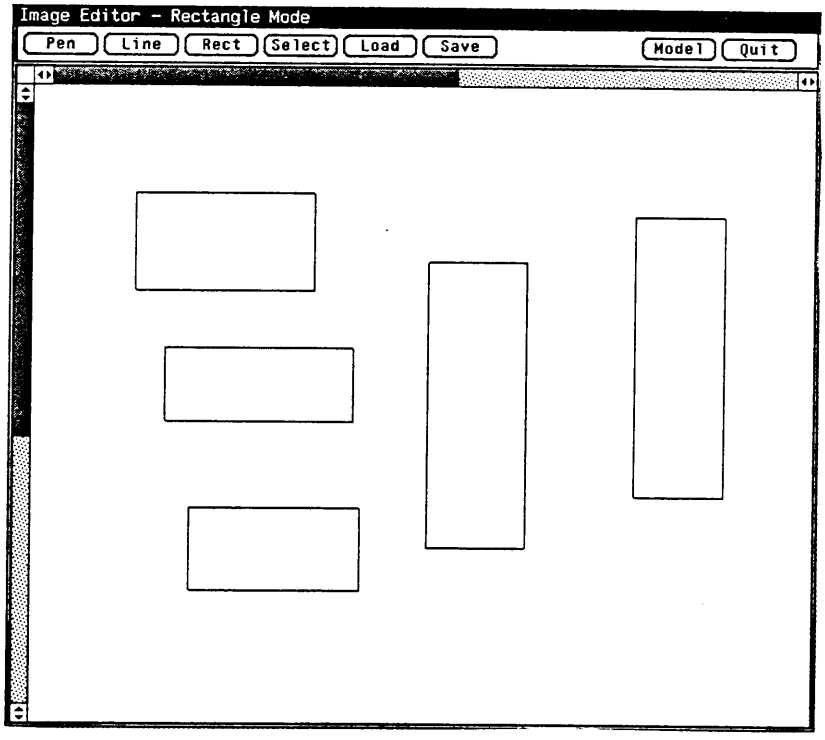

Figure 3.9 Image Editor Rectangle Mode

3.3.1.4 Select Mode

Select mode is entered by designating the "Select" button on the Image Editor control panel (see Figure 3.10). The select mode is provided so that the user can select any portion of the canvas for duplication or in order to copy the selected image to disk. The image selection process is akin to the process of drawing while in rectangle mode. The first corner of the image is selected by moving the mouse pointer to the desired position and depressing the left mouse button. While the button remains depressed, the user may move the mouse pointer anywhere on the canvas and a continuously updated rectangle with opposite corners at the first point and the current mouse pointer coordinates are drawn. When the rectangle is positioned correctly, releasing the left mouse button selects the image within the boundaries of the rectangle (see Figure 3.11). Once this process is complete, the user may duplicate the selected image on the canvas by depressing the middle mouse button (see Figure 3.12). While the button is depressed, the image appears on the canvas and follows the mouse pointer. When the middle button is released, the image is fixed. This process may be repeated as many times as desired until a new image has been selected with the left mouse button.

Selecting the "Select" button on the main control panel executes function select_proc() of the objed program. Select_proc() puts the editor into select mode and changes

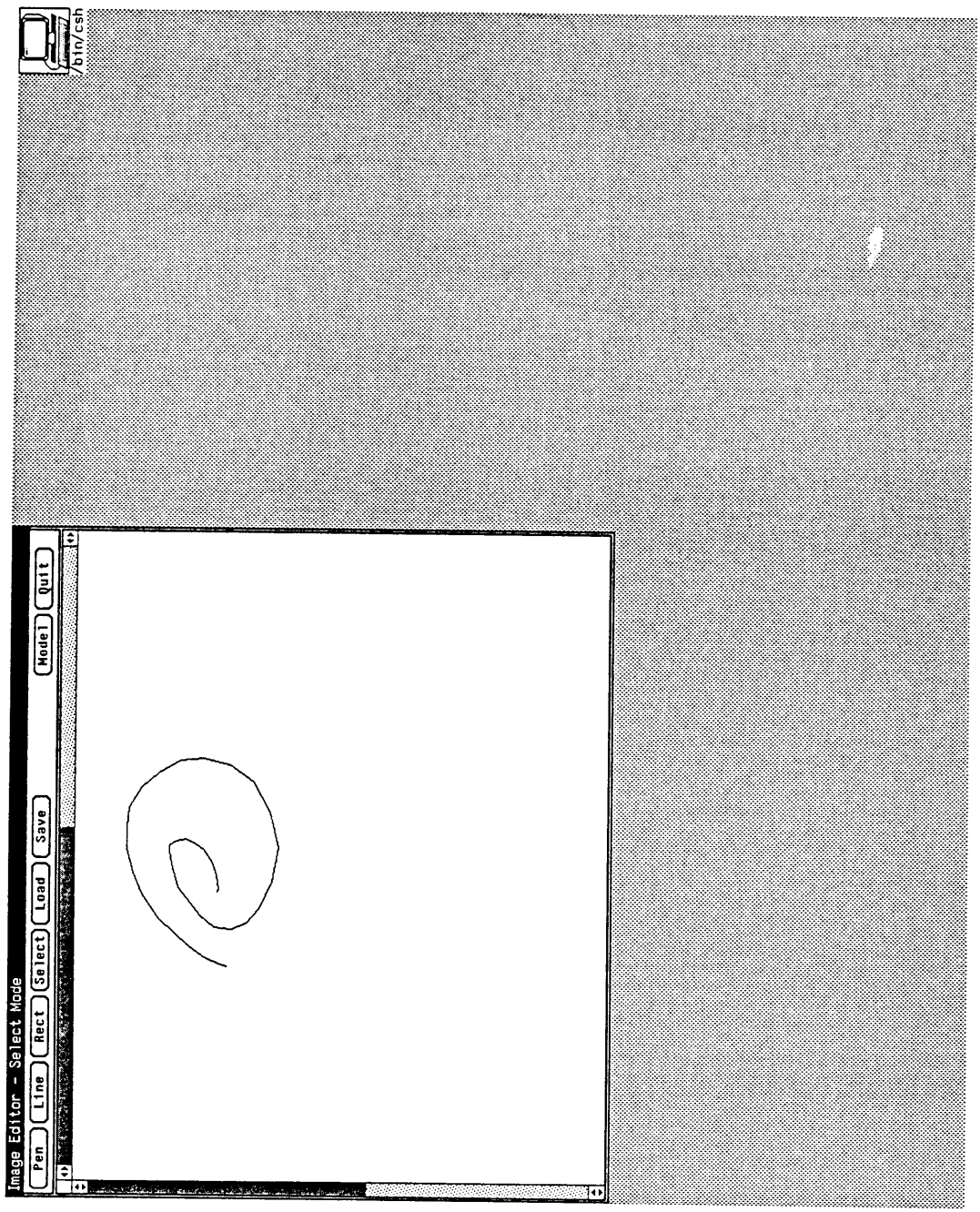

Figure 3.10 Image Editor Select Mode

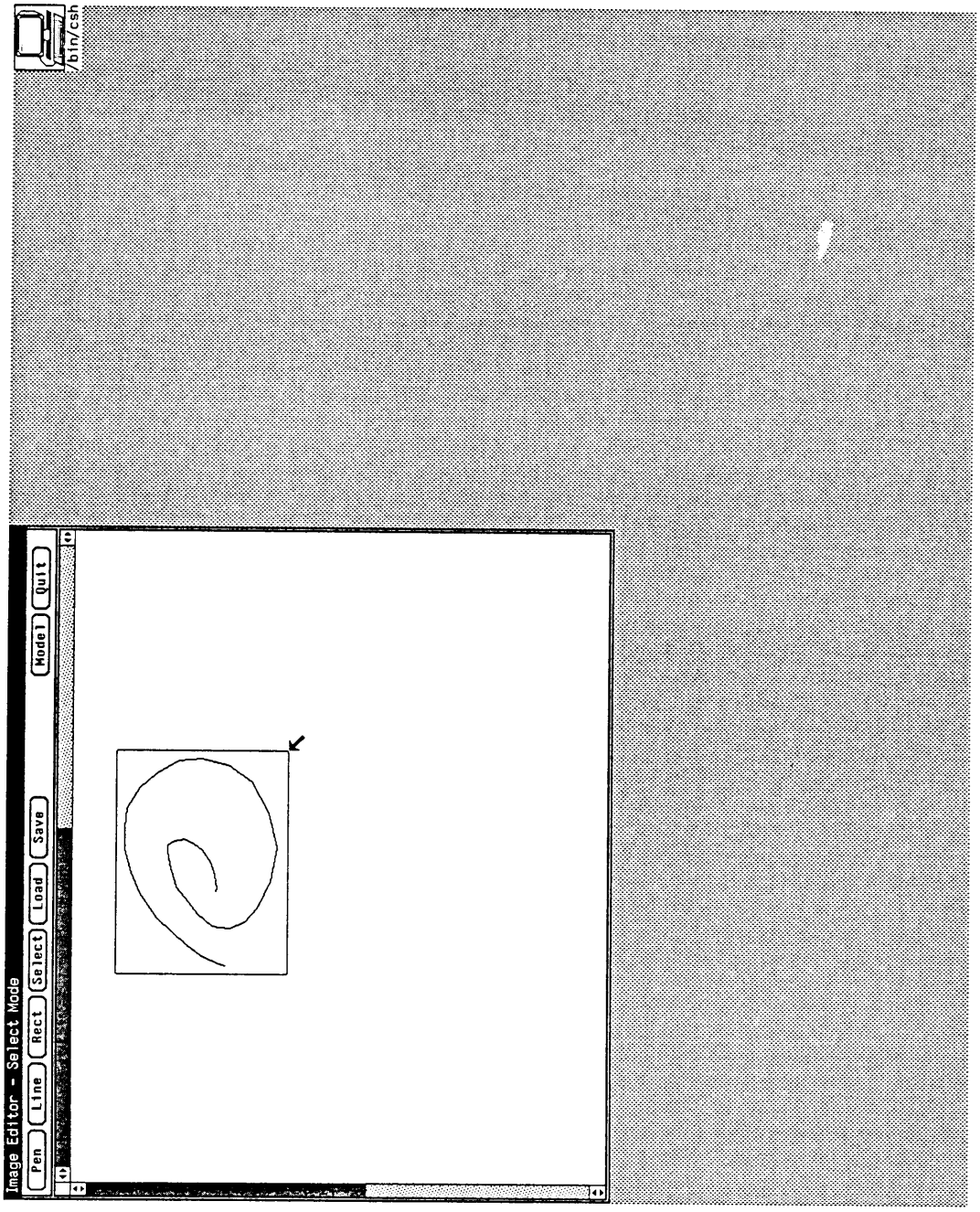

Figure 3.11 Selecting an Image

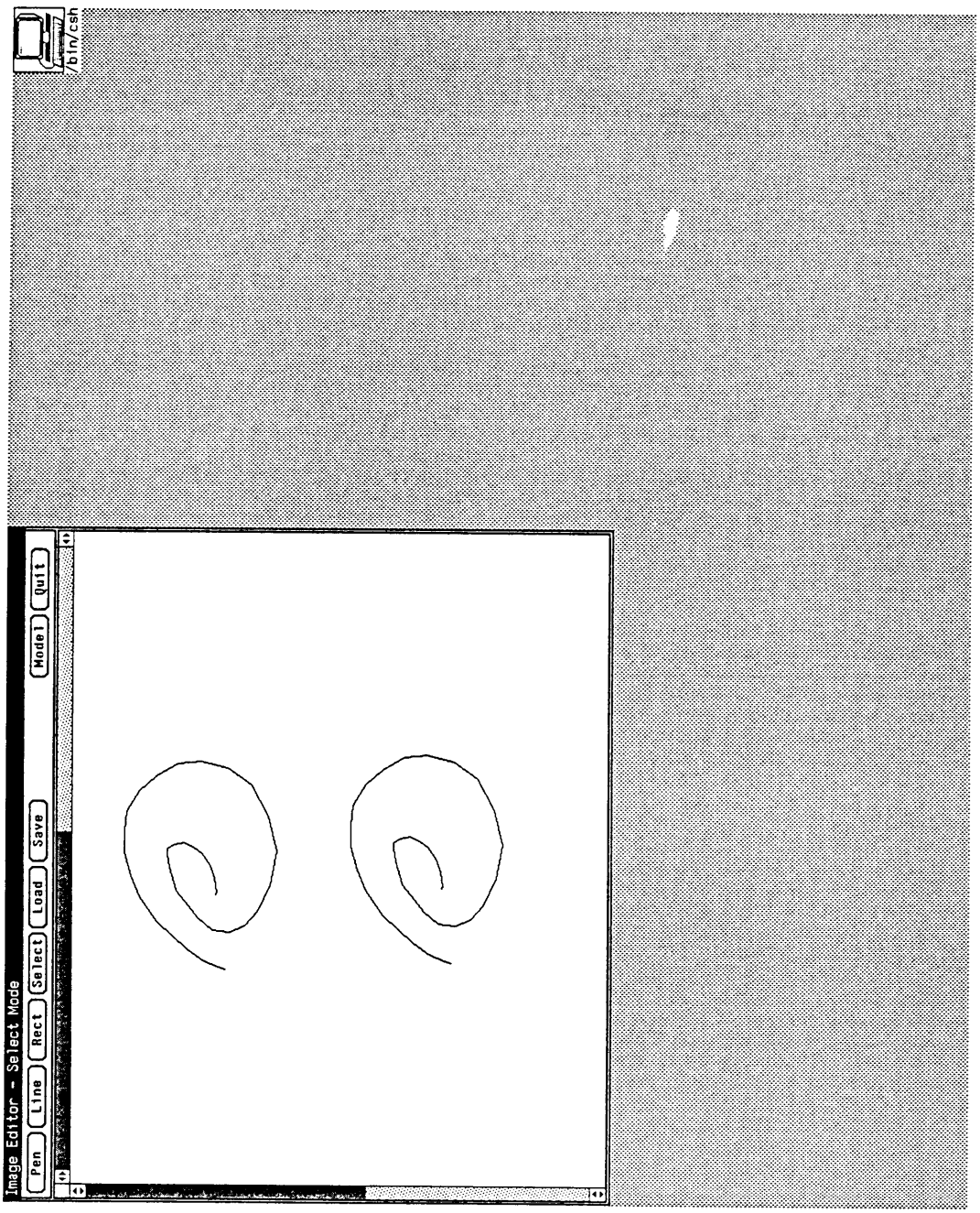

Figure 3.12 Duplicating an Image in Select Mode

the SunView event handling function to select_event_proc().

The select area box is drawn in a manner similar to the rectangle mode. When the dimensions of the selected area have been determined, a pixrect is created in memory using mem_create() and the selected area read into it from the canvas using pw_read(). Image movement is accomplished by writing the stored pixrect to the canvas using an XOR logical mask with the function pw_write().

The same erasure capabilities and restrictions associated with the use of the XOR logical mask discussed previously for line and rectangle modes apply here. Thus it is possible to erase large portions of the screen by selecting the area to be erased and then drawing the selected image directly over its canvas equivalent.

3.3.1.5 Image Load

The image editor has the capability of loading either backgrounds or objects. Background loading restores an entire canvas that has previously been stored to disk. Object loading restores images that have previously been obtained in select mode and saved to disk. In addition, provision has been made to store path names for backgrounds and objects that are "attached" to a specific model.

Selecting the "Load" button brings up a blocking requester with a PANEL_TEXT item lname_item labeled "File/Name:", and buttons labeled "Load Object", "Load

Attached Object", "Load Background", "Load Attached Background", and "No Load" (see Figure 3.13).

The "Load Object" button executes oload_proc(). The purpose of this function is to load a file from disk and allow the user to use it as a selected image in select mode. A previously selected image in memory is destroyed using pr_destroy(). The file specified in the load panel is opened and copied into a pixrect using pr_load(). Finally, the editor is placed in select mode, the SunView event handling function set to select_event_proc(), and the blocking requester cleared.

Selecting the "Load Attached Object" button executes the oload_attach() function. This function is substantially identical to oload_proc() except that the path name containing the image is obtained from the database rather than the user. Within the database provision has been made to reference object images by name rather than by path (see Table 3.1). Thus, an image associated with an object named "job" in the "CPU_System" model could be retrieved by entering "job" at the "File/Name:" prompt and selecting this option provided the modeler had specified "CPU_System" as the model name when entering the editor. Upon completion of loading, the blocking requester is cleared.

Selecting the "Load Background" button executes bload_proc(). This function performs identically to oload_proc() except that the entire canvas is being loaded

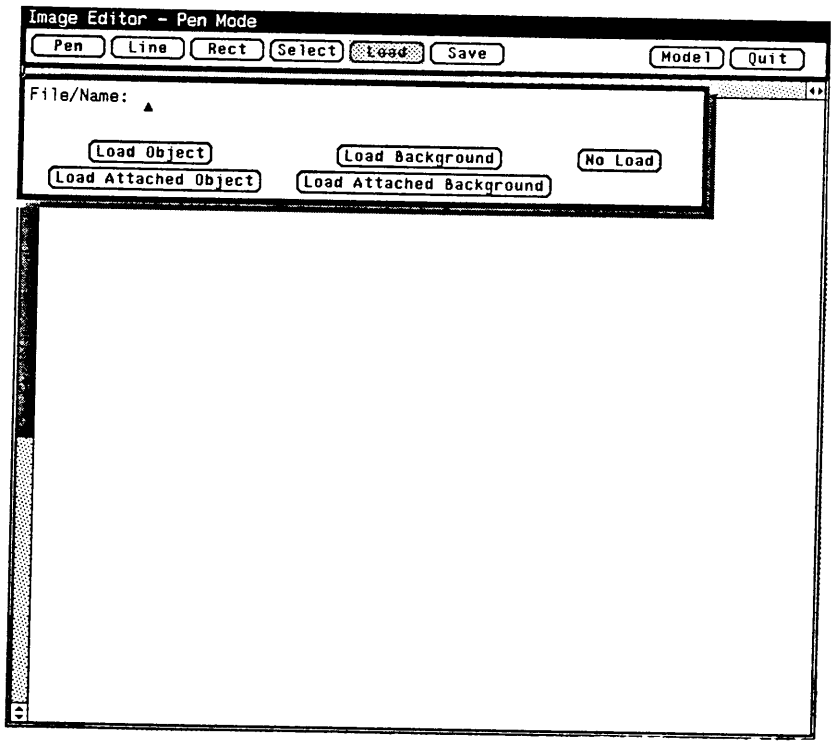

Figure 3.13 Image Editor Load Panel

Table 3.1 GPVSS Database Tables

Table: attr_dat

Name	Type	Bytes	Description
model	text	30	System model name
name	text	30	Name of attribute
brief	text	80	Brief description
attr_type	text	1	See object codes below
type	integer	4	Variable type
iav	text	80	Initial attr. value data
iav_type	integer	4	Initial attr. value type

Table: model_dat

Name	Type	Bytes	Description
mod_name	text	55	System model name
object_type	text	1	See object codes below
name	text	55	Name of object
path	text	55	Path name of file

Table: dynamic_dat

Name	Type	Bytes	Description
model	text	30	System model name
name	text	30	Name of Dynamic Object
brief	text	80	Brief description
submodel	text	80	First sub-model
file	text	30	File containing image
iat	text	80	Interarrival time
iat_type	integer	4	Interarrival type
gen	text	80	Maximum generation data
gen_type	integer	4	Maximum generation type

Object Codes

Model_dat:
- "O" - Object image
- "B" - Background image
- "M" - Model specification file
- "L" - Path to submodel Logic specification files

Attr_dat:
- "S" - Submodel attribute
- "D" - Dynamic Object attribute

from the specified file whose path name was entered at the "File/Name:" prompt. First the image is read into memory using pr_load(), then it is drawn on the canvas using pw_write(). Bload_proc() destroys any previously selected image if the user is in select mode. After loading the background, the blocking requester is cleared.

Selecting "Load Attached Background" executes the bload_attach() function. This function is similar to bload_proc() except the filename containing the image is retrieved from the database (see Table 3.1). There can only be one background for each model. Therefore it is not necessary to enter any name or filename at the "File/Name:" prompt as it is ignored.

Selecting the "No Load" button executes the no_load() function which merely clears the blocking file requester.

3.3.1.6 Image Save

The image editor has corresponding image save functions to those described previously in the image load section.

Selecting "Save" on the image editor control panel executes the save_proc() function, which in turn sets up a blocking requester on the save panel. The save panel contains two PANEL_TEXT items labeled "File:" and "Name:" as well as buttons labeled "Save Object", "Save/Attach Object", "Save Background", "Save/Attach Background", and "No Save" (see Figure 3.14).

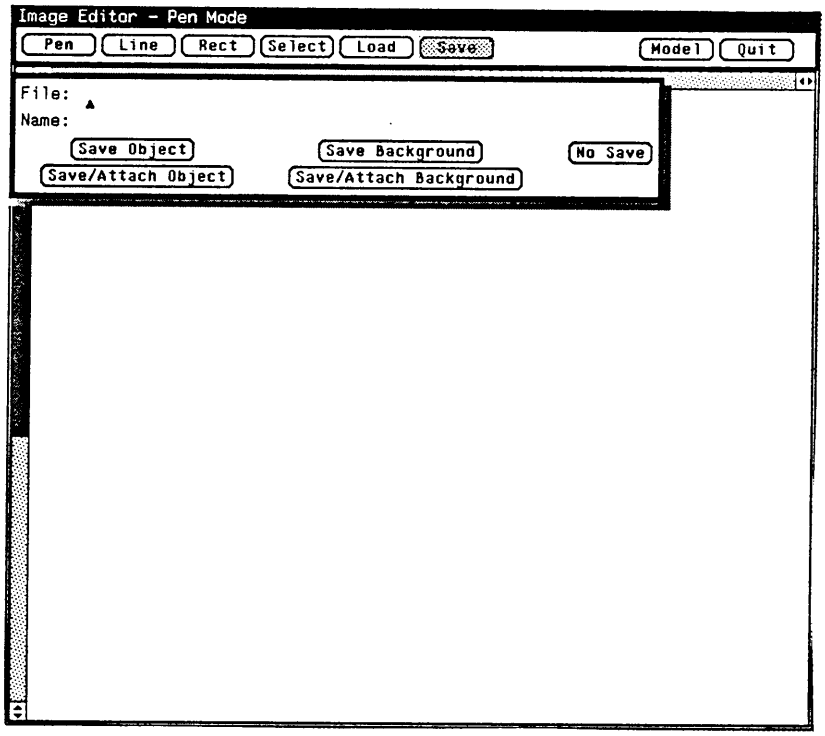

Figure 3.14 Image Editor Save Panel

Selecting the "Save Object" button calls function osave_proc(). Osave_proc() stores a selected object image to disk. Therefore "Save Object" does not work correctly if the editor is not in select mode and an object image selected. Moving an image to disk is a three step process. Storage for the image is allocated using mem_create(), and the image transferred from the canvas to memory using pw_read(). Finally, pr_dump() is used to transfer the image from memory to the path specified in the "File:" panel item. The blocking requester is cleared upon completion of the operation.

Selecting "Save/Attach Object" executes function osave_attach(). This function performs identically to osave_proc(), and also stores information in the database linking an object name to a path containing its image for a given model (see Table 3.1). For this reason, the desired name of the object should be entered at the "Name:" panel item in addition to entering the path name of the image file at the "File:" panel item.

Selecting "Save Background" calls bsave_proc(), which is identical in almost every respect to osave_proc(). There are important differences, however. The primary difference is that the entire canvas is saved to the path specified in the "File:" panel item. Also, osave_proc() may be called while in any mode, but destroys any previously selected image if called in select mode.

Selecting "Save/Attach Background" calls bsave_attach(). This function is functionally equivalent to bsave_proc(), but also updates the database to attach the path name specified in the "File:" panel item as the background image for the current model being worked on. It is not necessary to specify a name as this information is ignored. If the current model has already had a path specified for the background, and the background saved using "Save/Attach Background", the path defaults to the previous entry providing that absolutely nothing was entered at the "File:" panel item, including blank spaces. If a new path name is specified, the database is changed to reflect the change. Thus it is only necessary to specify a path name for the background once.

Selecting "No Save" executes the no_save() function which merely clears the blocking requester without saving.

3.3.2 The Model Editor

The purpose of the model editor is to provide facilities for the interactive definition and specification of the model logic.

The model editor is entered from the image editor by selecting the "Model" button which calls the model_proc() function. Model_proc() draws any paths and submodels that have already been defined, and changes the main panel buttons so that the model editor buttons are shown (see

Figure 3.15). Drawing the model definition is a relatively straightforward process. Submodels are drawn using the rec_draw() function as in rectangle mode with the XOR logical mask. Paths are drawn using the pw_vector() function as in line mode between successive pairs of points also using the XOR logical mask.

The "Image" button takes the place of the "Model" button when in the model editor. Selecting the "Image" button executes the function object_proc(). Object_proc() erases the model definition by re-drawing it with the XOR logical mask and sets the main panel buttons for the image editor.

3.3.2.1 Submodel Definition

This model editor mode provides the user with the capability to partition the background image into multiple discrete rectangular submodel partitions.

Selecting the "SubMod" button on the model editor control panel executes submod_proc(). Submod_proc() merely sets up a blocking requester to input the name of the new submodel (see Figure 3.16).

The submodel name requester contains a "Sub-Model:" PANEL_TEXT item and two buttons labeled "Define Sub-Model" and "Exit". Selecting the "Define Sub-Model" button executes the function model_def_proc(). Model_def_proc() allocates memory for the new sub-model if necessary, updates

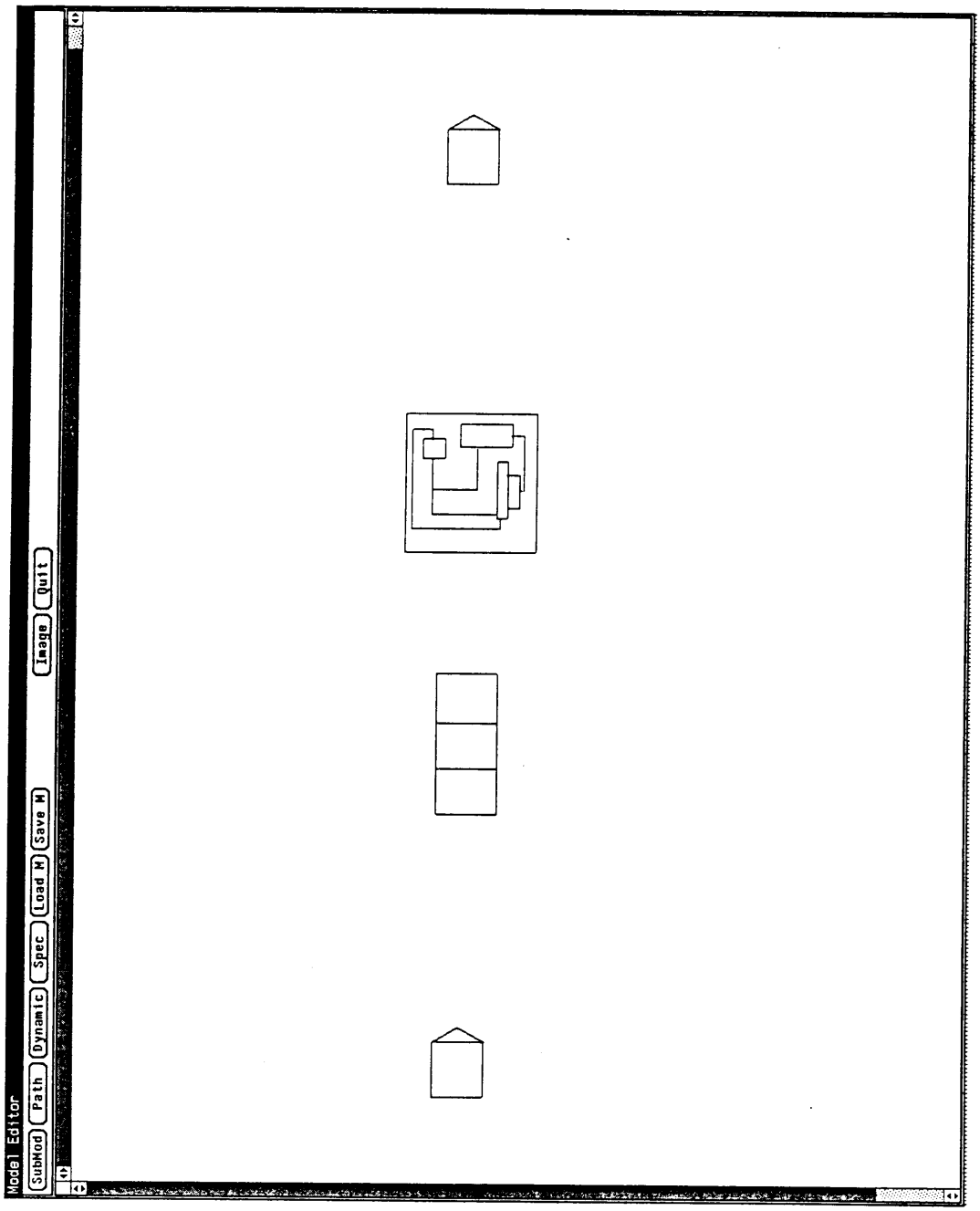

Figure 3.15 The Model Editor

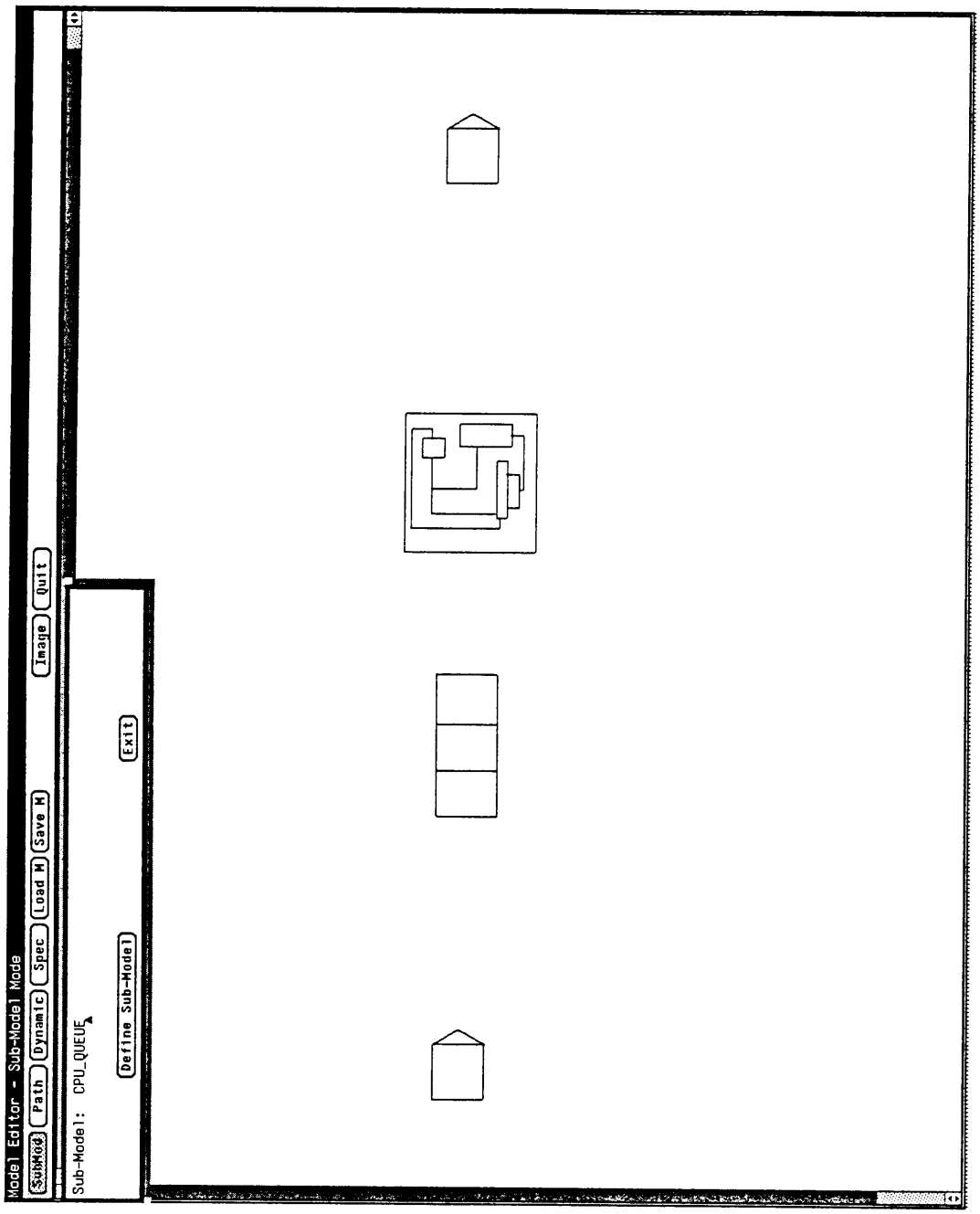

Figure 3.16 Submodel Name Requester

the list of submodels, and changes the SunView event handling function to sub_event_proc().

Sub_event_proc() first obtains the name for the sub-model that is entered at the "Sub-Model:" prompt on the blocking requester. This name must conform to standard C identifier conventions. From this point forward, the procedure for actually drawing the submodel rectangle is very similar to the implementation of the rectangle mode capability of the image editor (see Figure 3.17). The function upper_left() is called to update the submodel coordinates at each SunView event. Note that the internal logic assumes that submodel partitions are discrete, and that the model editor does not enforce this convention.

A limited submodel delete facility is provided. By depressing the middle mouse button, the last submodel to be defined is eliminated. The memory is not de-allocated here and may be used again if defining multiple submodels. The Boolean variable abort_sub keeps track of whether the last submodel in the submodel list has been "deleted" or not. This feature is only active immediately after a sub-model has been defined and the user has not exited the submodel definition mode.

Selecting "Exit" from the submodel blocking requester executes the function exit_proc(). Exit_proc() returns control to the model editor by clearing the blocking requester and not defining a submodel.

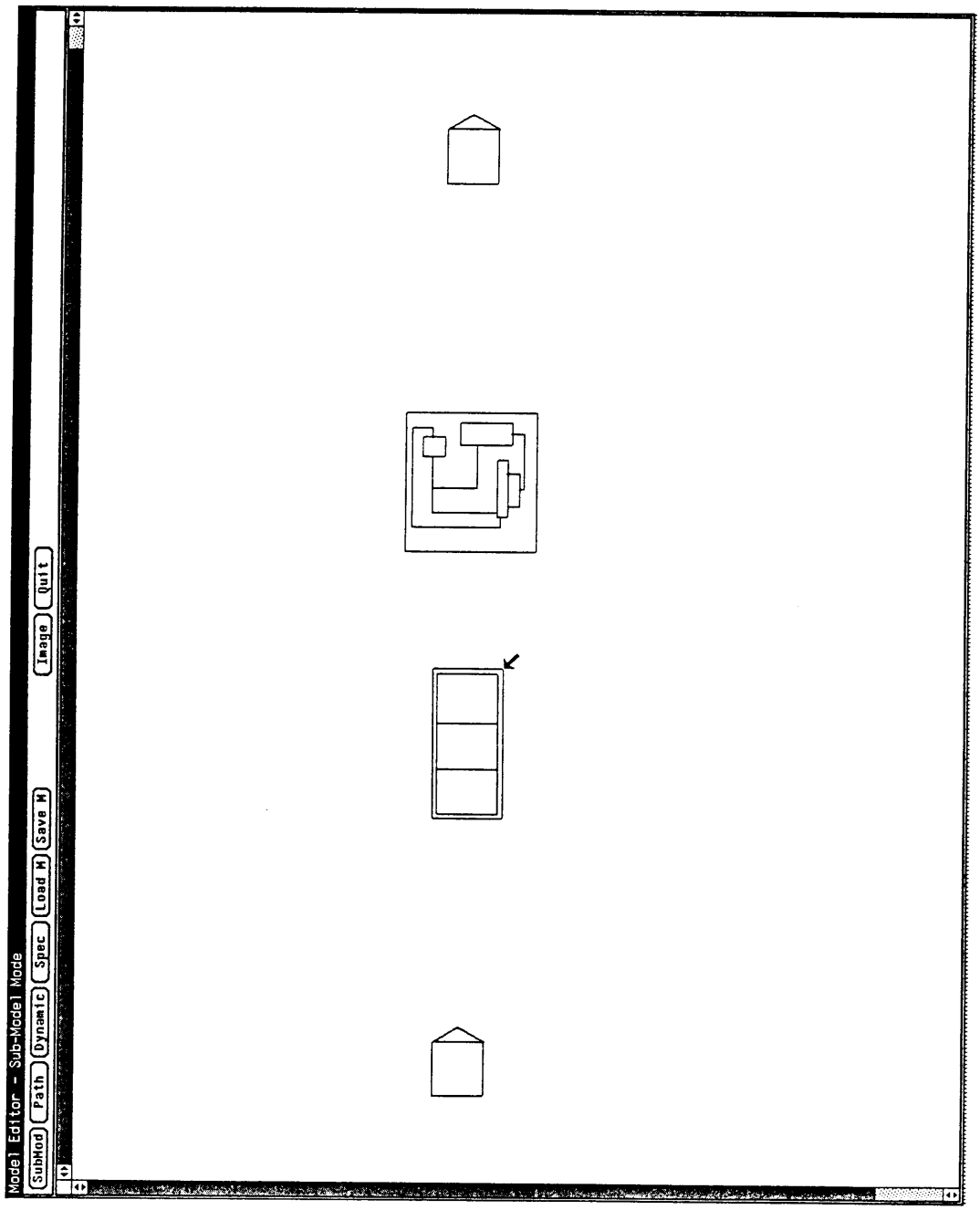

Figure 3.17 Defining a Submodel

3.3.2.2 Path Definition

Path definition facilities are provided to allow the user to define a path between any two submodels. Paths are uni-directional. Currently only one path with a given "from" and "to" submodel is supported. Note that the model editor does not enforce this convention.

Path definition begins by selecting the "Path" button on the main control panel. This action executes the path_proc() function which puts the editor into path mode and sets up a blocking requester for the input of the path name (see Figure 3.18).

The path name requester contains a "Path:" PANEL_TEXT item and two buttons labeled "Define Path" and "Exit". Selecting the "Define Path" button executes the function path_def_proc(). Path_def_proc() initializes some variables and changes the SunView event handling function to sub_event_proc().

The path coordinates can now be defined. Paths must start and end within submodels. First move the mouse pointer to the desired coordinates within the "from" submodel. Depressing the left mouse button designates this as the path starting point. Subsequent points are designated by moving the mouse pointer to the appropriate coordinates and depressing the left mouse button (see Figure 3.19). These intermediate points may be anywhere on the canvas, including within submodels. The termination point

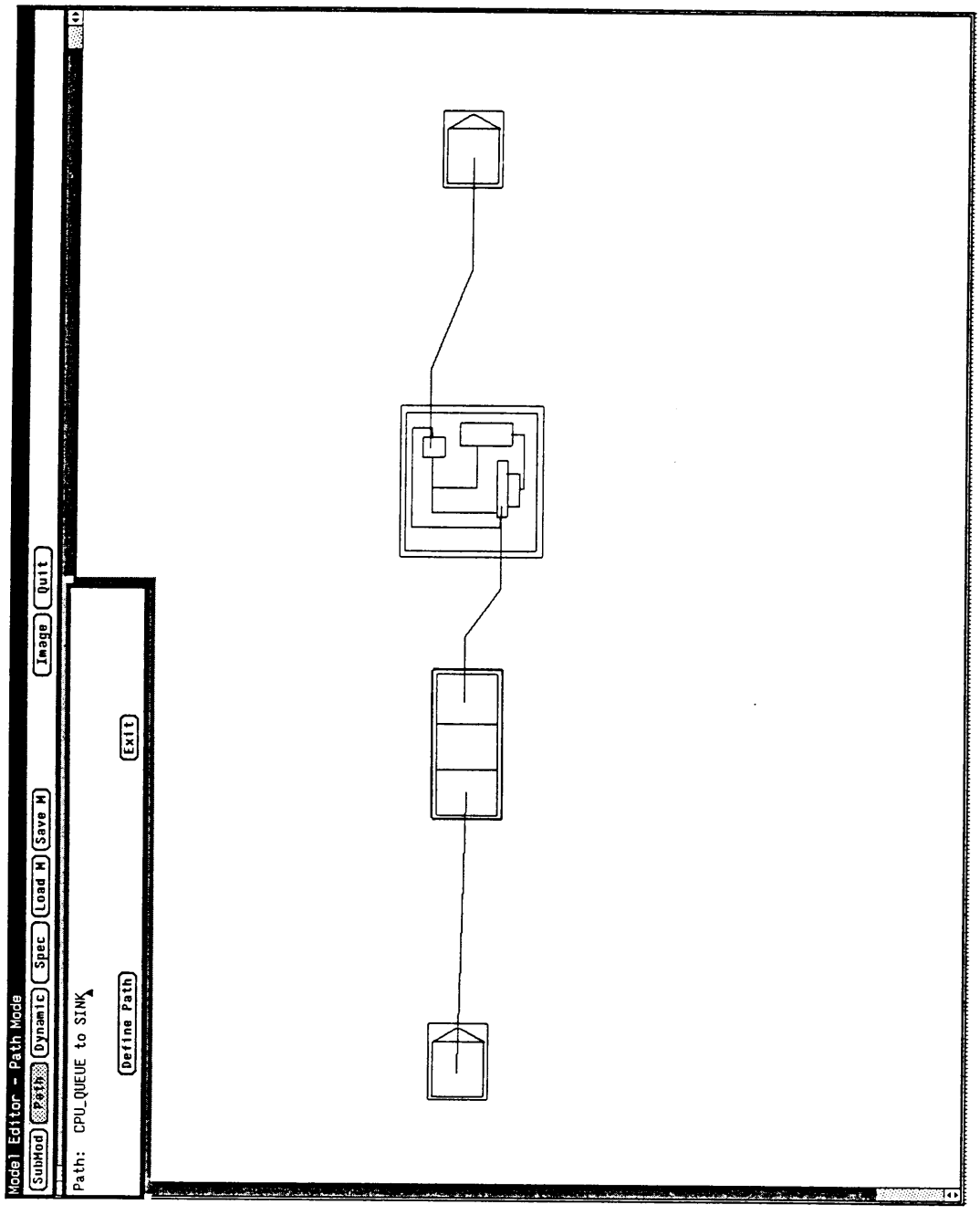

Figure 3.18 Path Name Requester

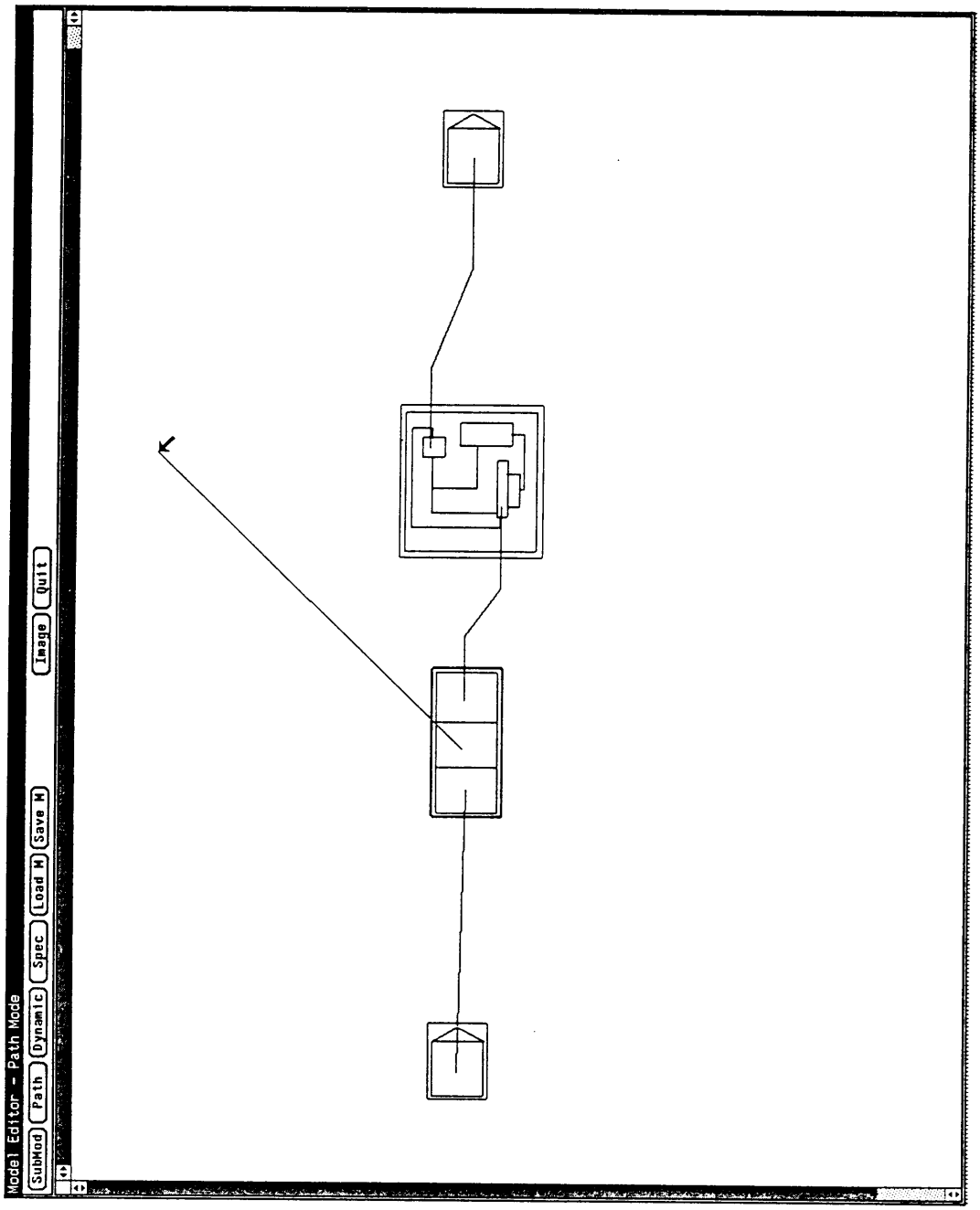

Figure 3.19 Intermediate Path Point Definition

is designated by moving the mouse pointer within a submodel and depressing the middle mouse button (see Figure 3.20).

Sub_event_proc() is the function that handles the above process. Initially, memory is allocated and the path list updated. As start, intermediate, and end points are selected the path definition is updated and memory allocated as necessary. For start and end points, fin_sub() is used to make sure that the coordinates lie within a submodel. In addition, a line is always drawn between the last selected point and the current mouse pointer coordinates in a manner similar to the method used while in line mode in the image editor.

Selecting "Exit" from the path definition blocking requester executes the function exit_proc(). Exit_proc() returns control to the model editor by clearing the blocking requester and not defining a path.

3.3.2.3 Dynamic Object Specification

Dynamic object specification is initiated by selecting the "Dynamic" button on the main control panel to execute the name_dynamic_object() function, which sets up a blocking requester for the input of the dynamic object name (see Figure 3.21). This name must conform to standard C identifier conventions.

The dynamic object name requester contains a "Dynamic Object Name:" PANEL_TEXT item and two buttons labeled "Name

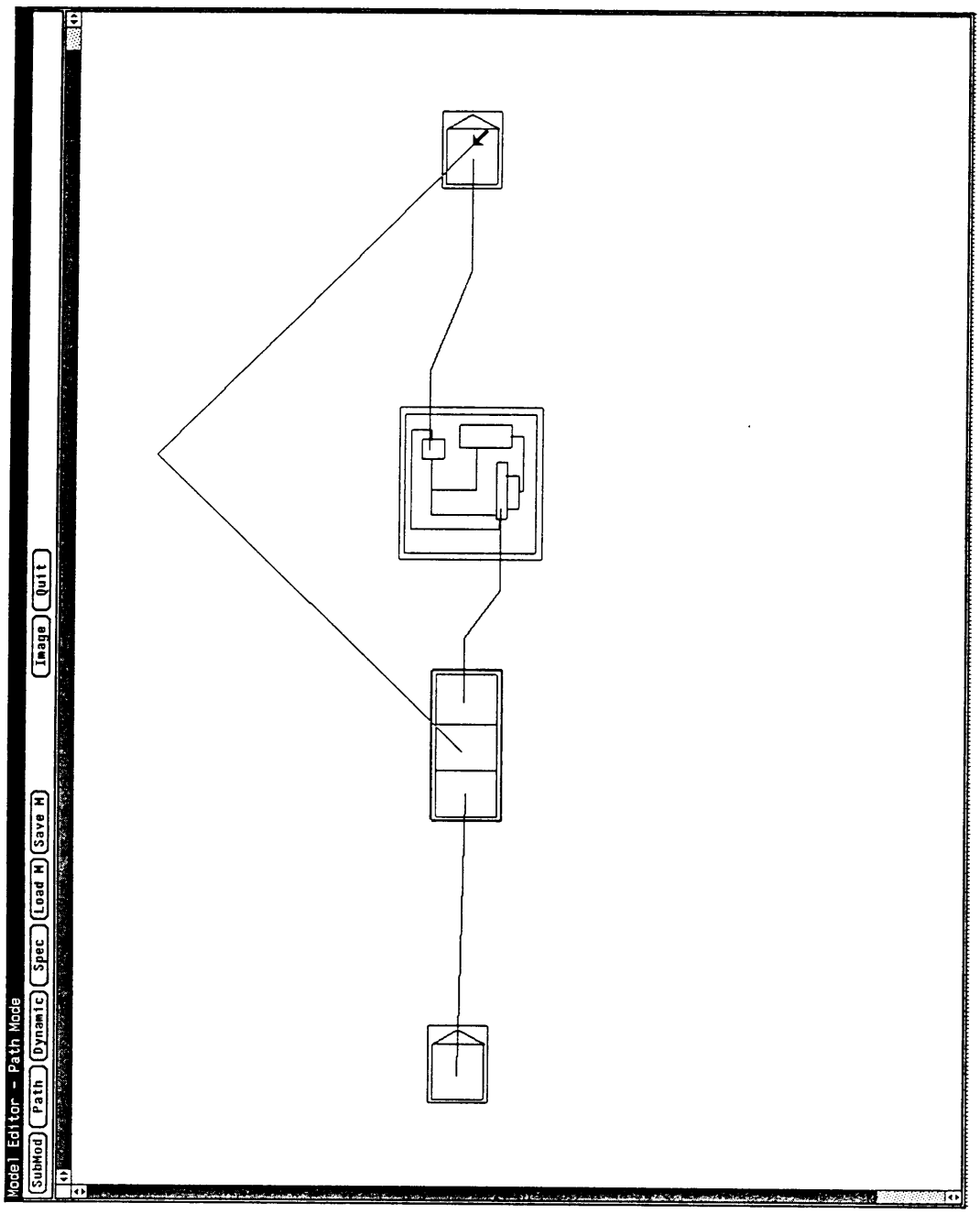

Figure 3.20 Path Termination

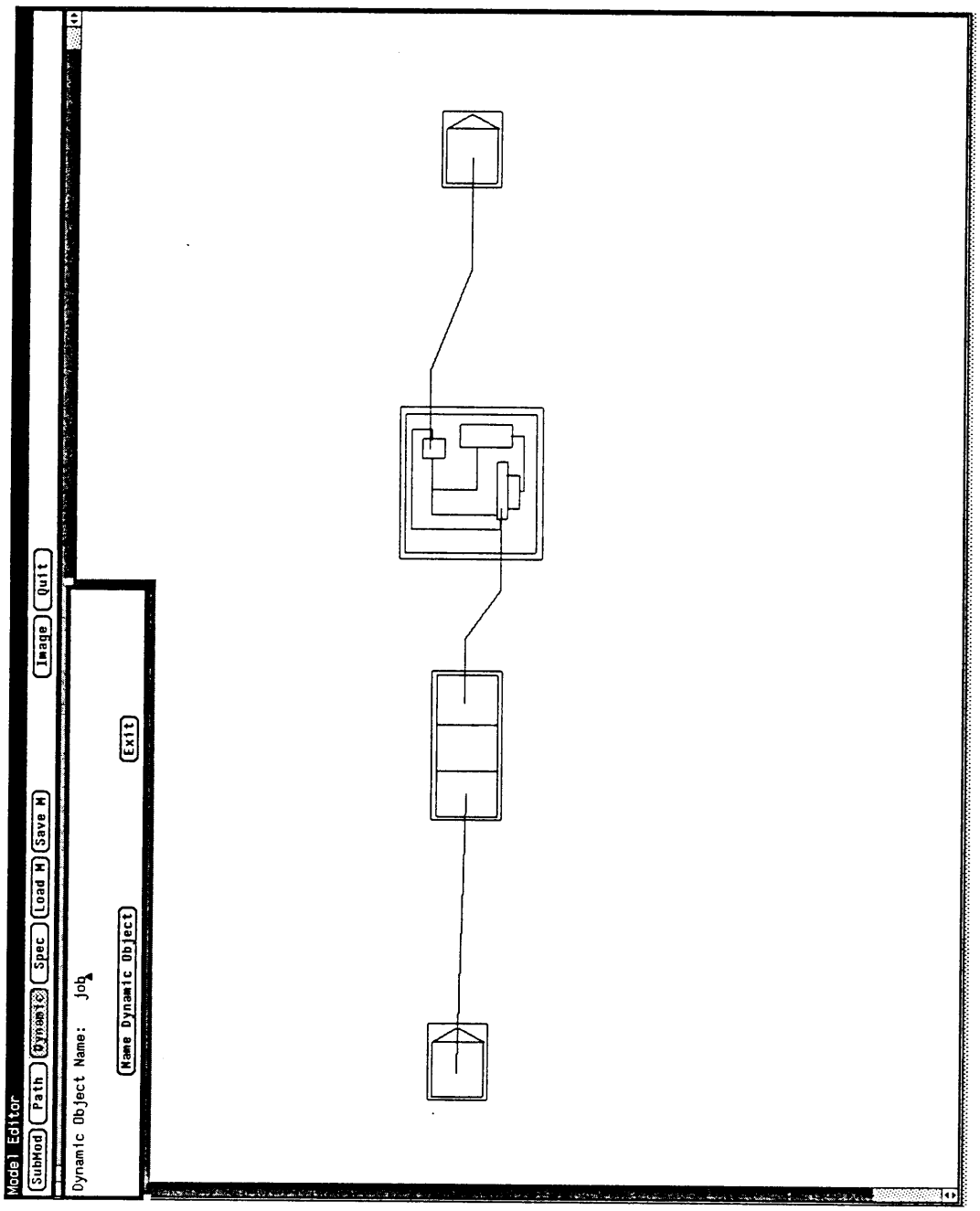

Figure 3.21 Dynamic Object Name Requester

Dynamic Object" and "Exit".

Selecting "Exit" from the dynamic object name blocking requester executes the function exit_proc(). Exit_proc() returns control to the model editor by clearing the blocking requester and not naming a dynamic object.

Selecting the "Name Dynamic Object" button executes the function doname_proc(), which forks the dynamic program with the model name and system name as command line arguments.

The dynamic program is designed as an input tool for the definition and specification of dynamic objects (see Figure 3.22). All dynamic object (DO) information input through the dynamic program is stored in the database in the attr_data table (see Table 3.1 in Section 3.3.1.5).

The DO name previously specified heads the display. Underneath are provisions for entering (1) a brief description of the DO, (2) the origin submodel, (3) the inter-arrival time, and (4) the maximum generated DOs.

The brief description of the DO is primarily for the use of the modeler. It holds a single line of 80 characters describing the significance of the DO within the model.

The origin submodel is the first submodel the DO attempts to enter.

The interarrival time may be "Constant", "Returned", or "Sampled" (see Figure 3.23). "Constant" interarrival times never vary. "Returned" interarrival times are determined by a user defined function. "Sampled" interarrival times may

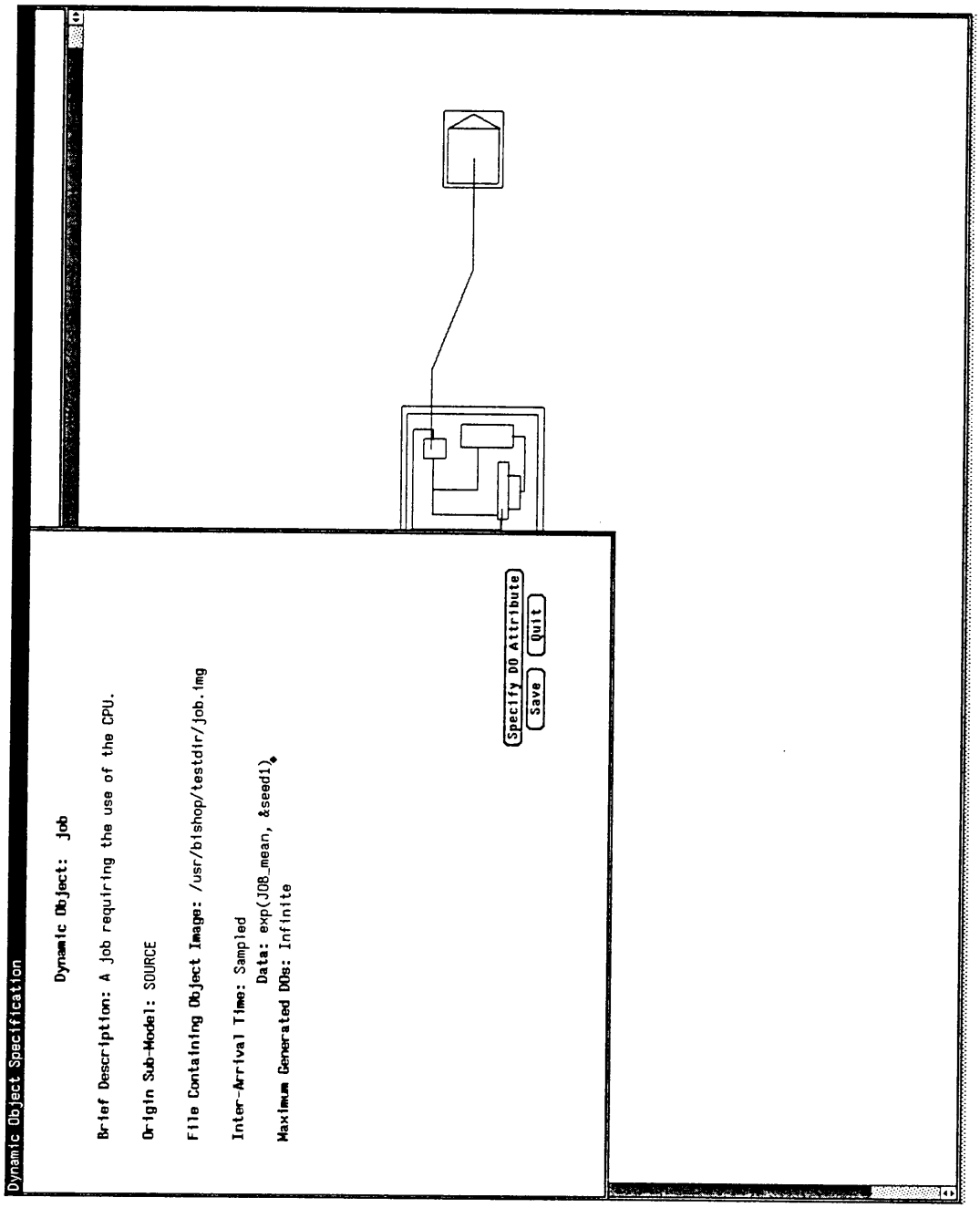

Figure 3.22 DO Definition and Specification Tool

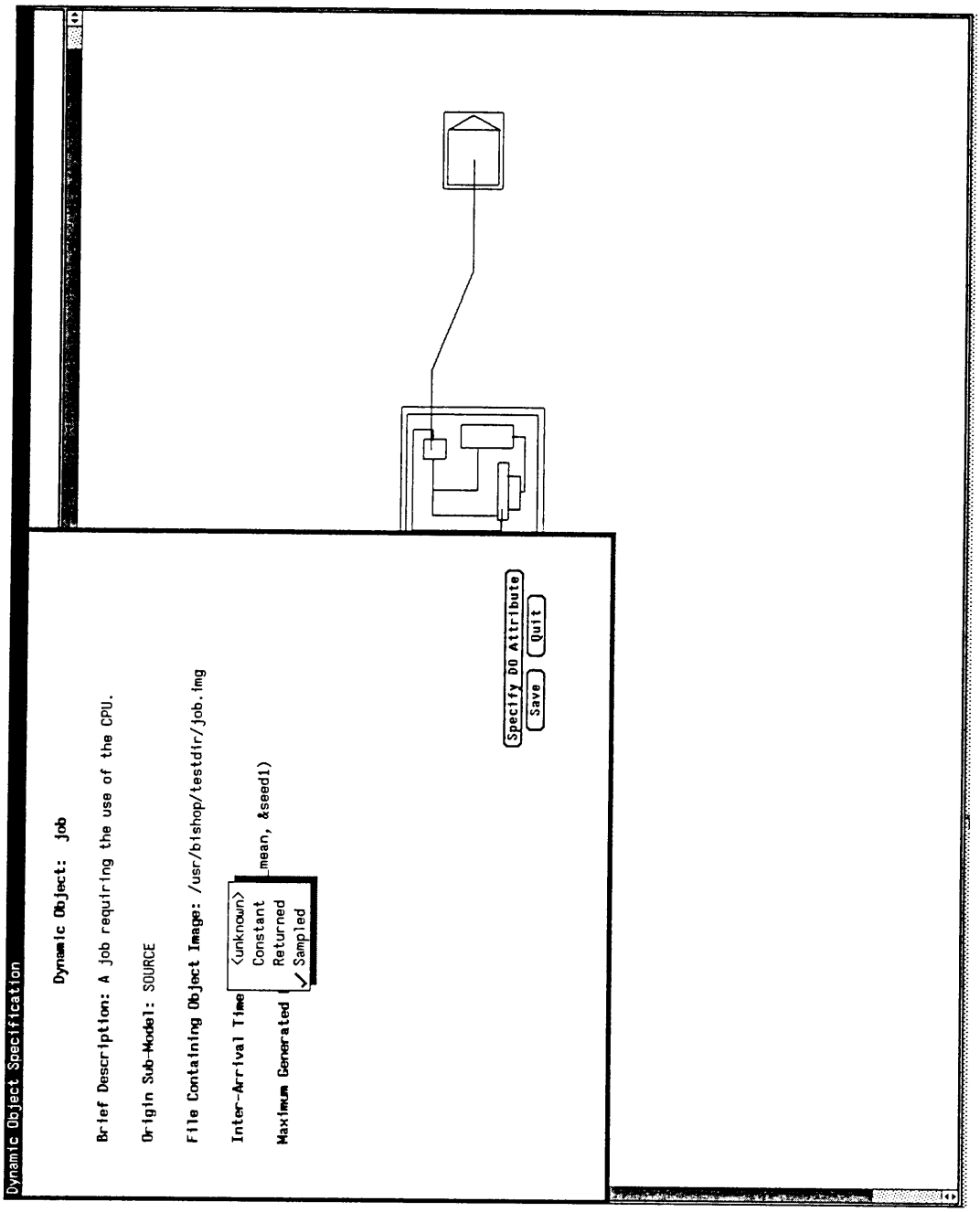

Figure 3.23 DO Interarrival Time Menu

be chosen from the probability distribution screen (see Figure 3.24). Currently only the exponential distribution function is implemented.

The maximum generated DOs may be either "Infinite" or "Finite" (see Figure 3.25). Selecting "Infinite" creates a never ending stream of DOs into the model system. If "Finite" is chosen, the user must specify an integer value for the number of DOs entering the model system at the "Data:" prompt. The user may specify a function to determine the number of DOs to enter the system, as long as that function returns an integer value.

Located at the bottom of the display are buttons labeled "Save", "Quit", and "Specify DO Attribute".

The "Save" button calls save_proc(). Save_proc() stores the currently displayed DO specification in the "attr_dat" table in the database (see Table 3.1 in Section 3.3.1.5).

The "Quit" button calls quit_proc() which quits the dynamic program without saving the DO specification.

Selecting the "Specify DO Attribute" allows the user to define an attribute that becomes part of the DO structure definition. This button executes the function spec_proc(). Spec_proc() sets up a blocking requester for the input of an attribute name (see Figure 3.26). This panel has an "Attribute Name:" PANEL_TEXT item and two buttons labeled "Name Attribute" and "Exit".

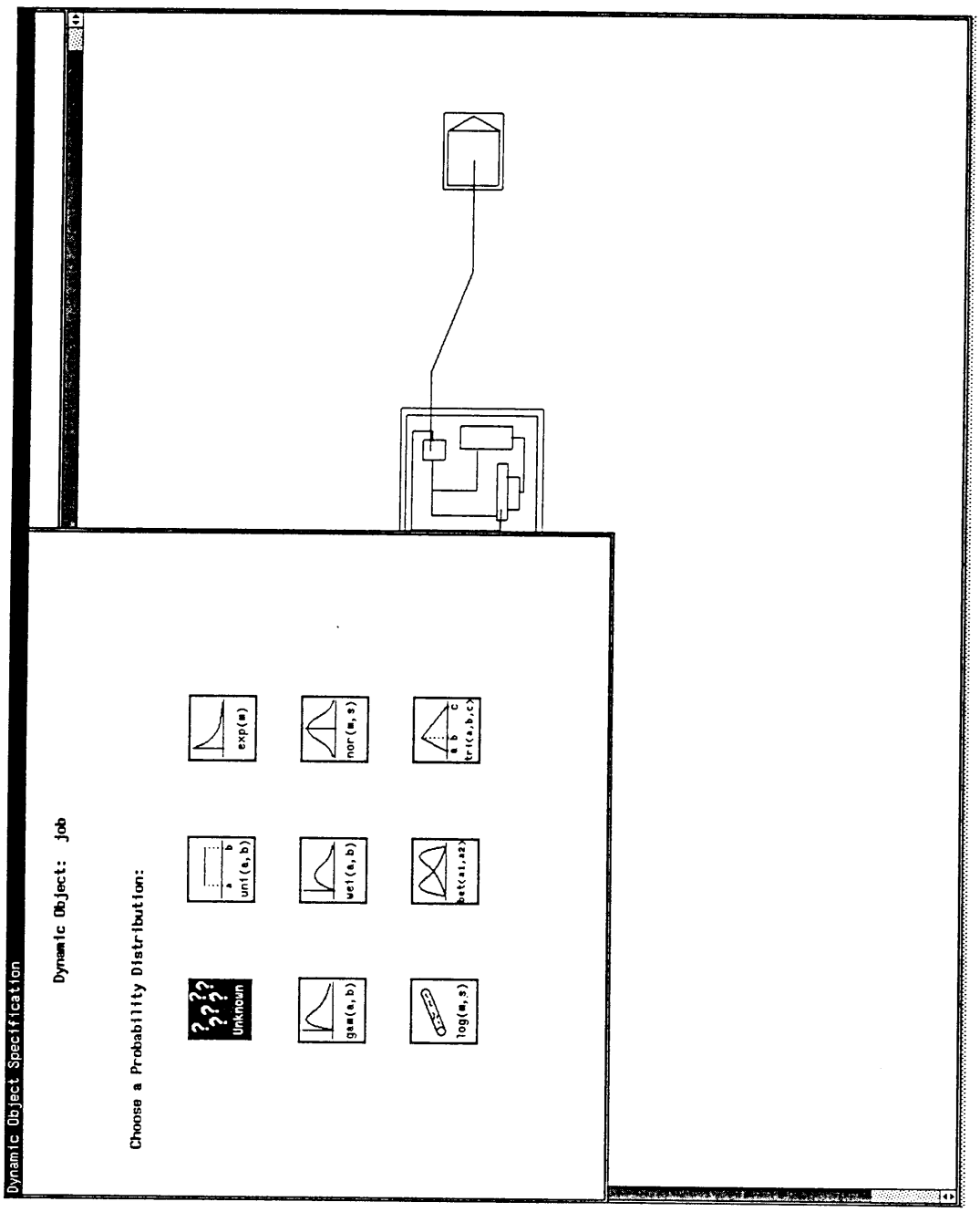

Figure 3.24 Interarrival Probability Distributions

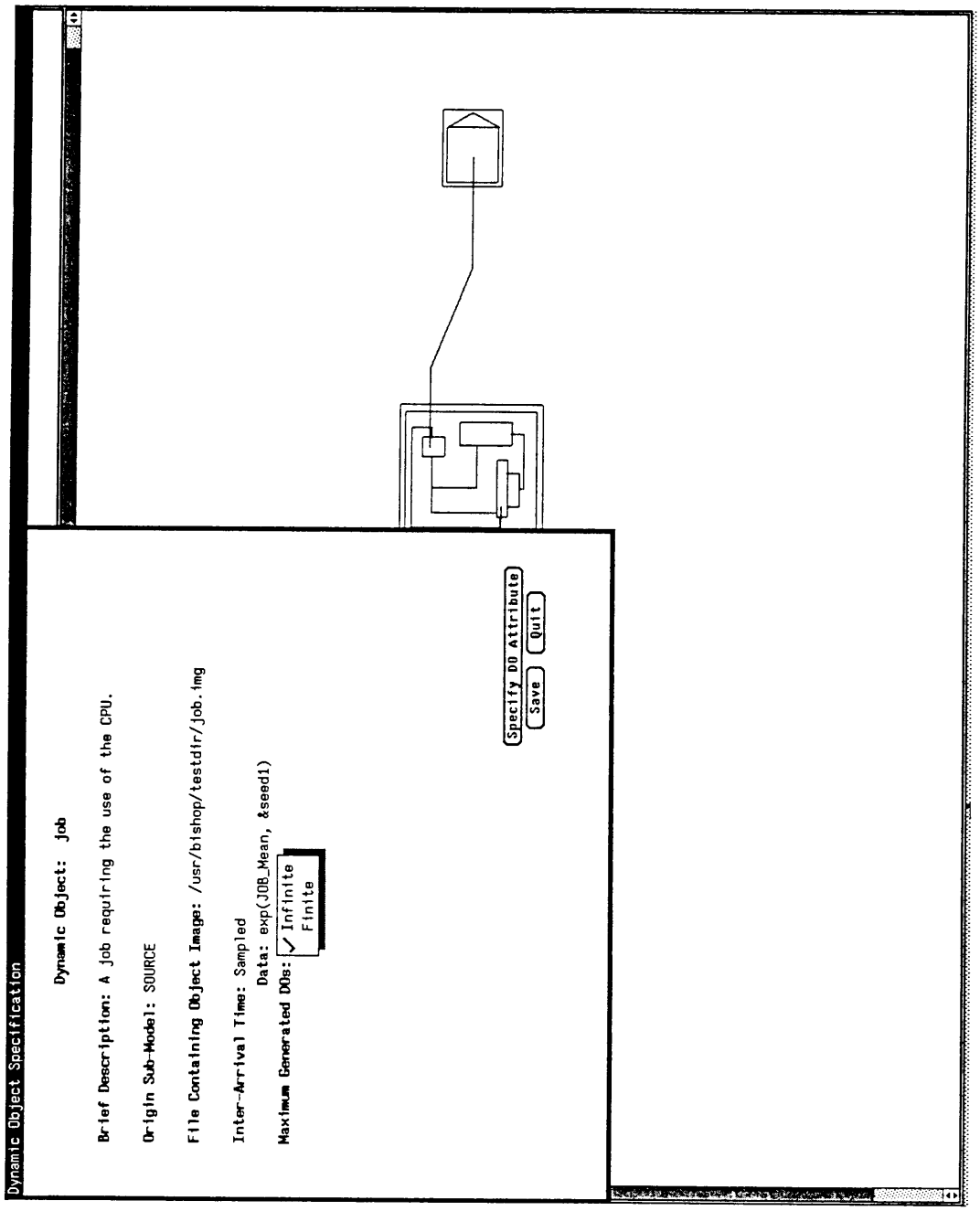

Figure 3.25 Maximum Generated DO Menu

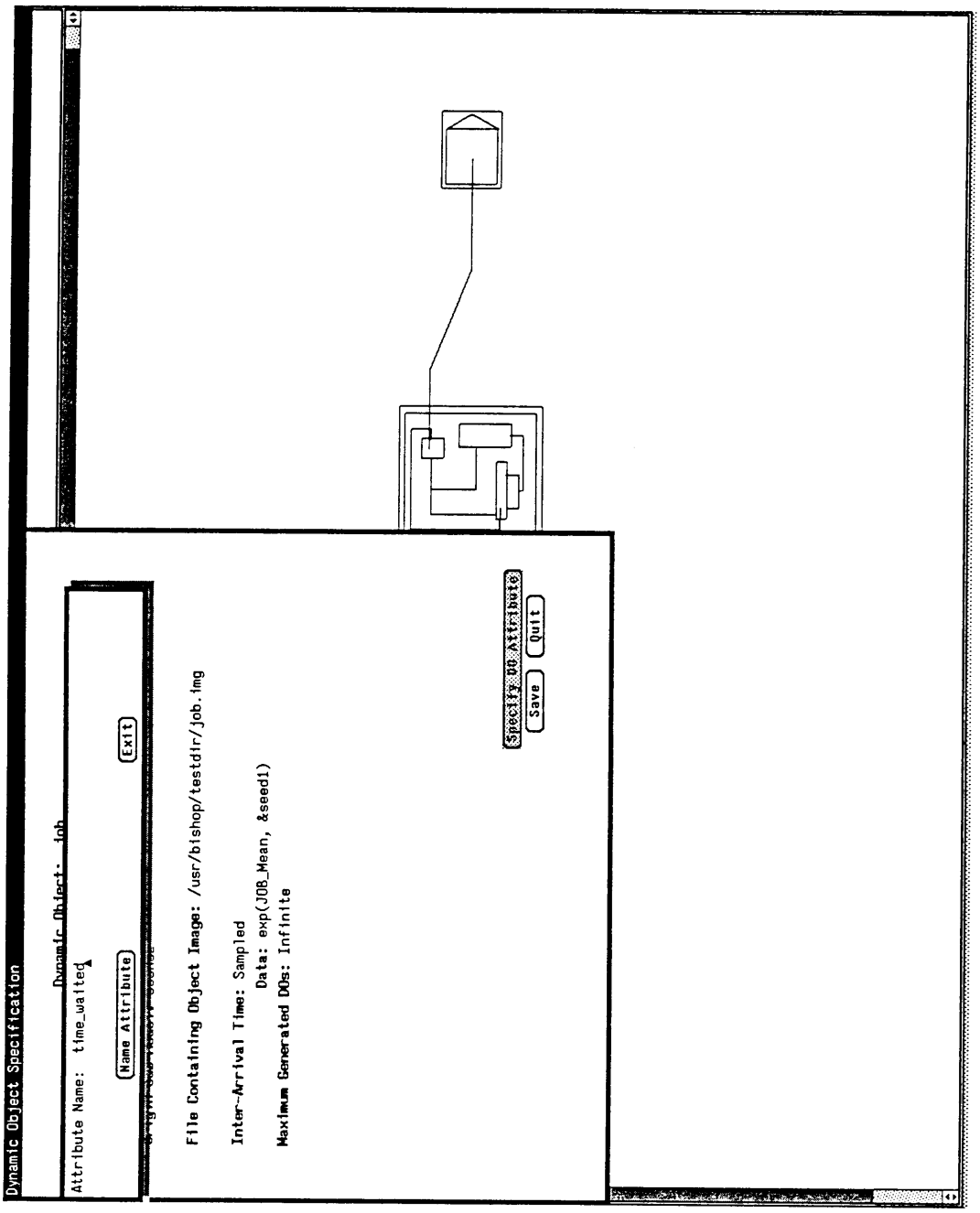

Figure 3.26 Attribute Name Requester

The "Exit" button executes the exit_proc() function which clears the blocking requester without defining an attribute.

The "Name Attribute" button forks the attribute program with the model name, attribute name, and "D" (for DO) as command line arguments. The purpose of the attribute program is to provide facilities for the definition, specification, and storage of both DO attributes and submodel attributes. DO information is stored in the "attr_dat" table of the database (see Table 3.1 in Section 3.3.1.5). The attribute panel contains: (1) the attribute name, (2) a brief description of the attribute, (3) the type of attribute value, and (4) the initial attribute value (see Figure 3.27).

The brief description describes what aspect of the DO or model system the attribute depicts and the type of attribute value describes what values the attribute may take. Attribute types may be (1) "int", (2) "long", (3) "float", (4) "double", (5) "char", or (6) "time" (see Figure 3.28). These classifications correspond to C variable types except for "time", which in this case is type double.

The initial attribute value may be (1) "Constant", (2) "Returned", (3) "Sampled", or (4) "Evaluated" (see Figure 3.29). "Constant" may be any value of the same type as the attribute. "Returned" allows the initial value to be set by a user defined function. In addition, the pre-defined

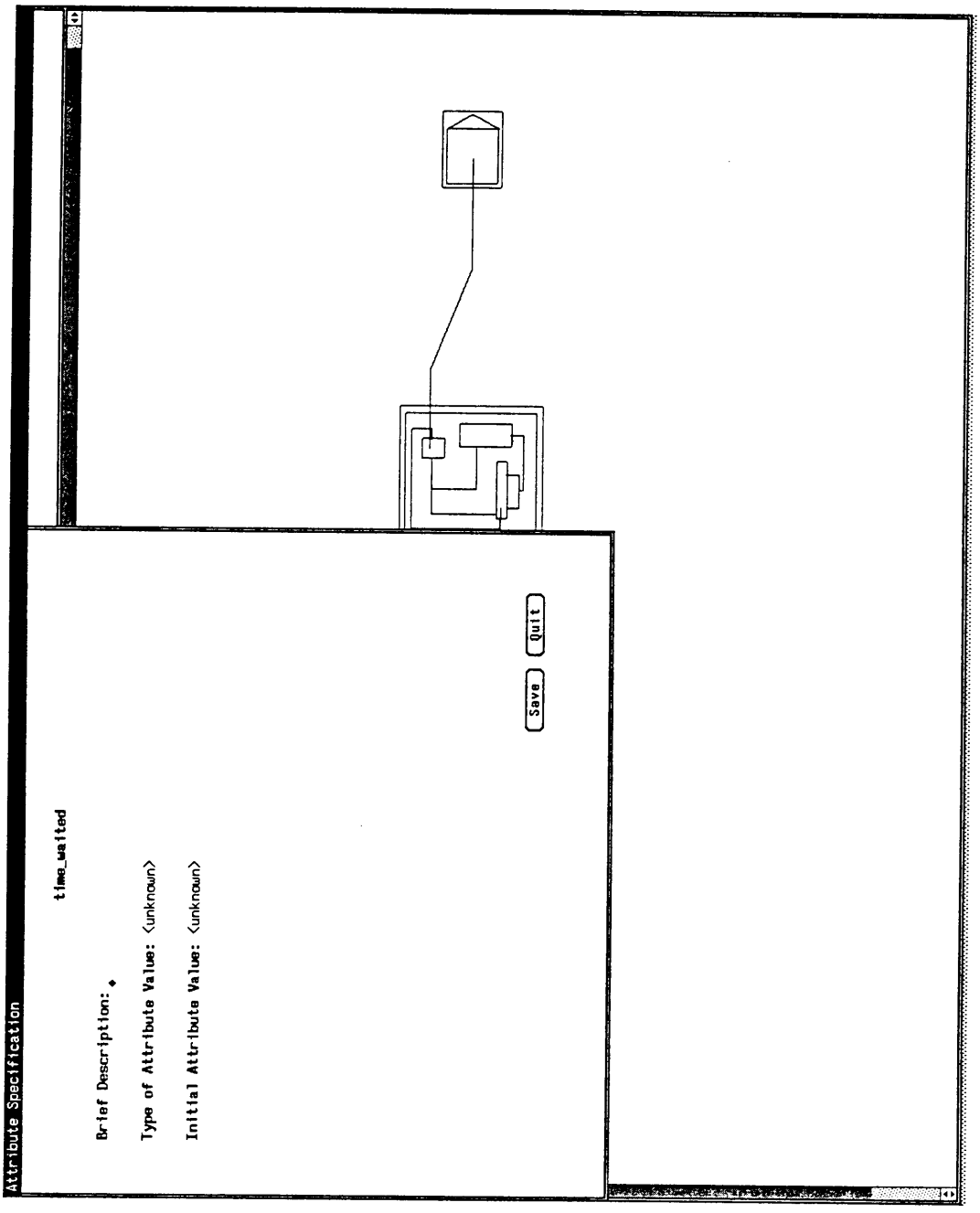

Figure 3.27 Attribute Definition/Specification Tool

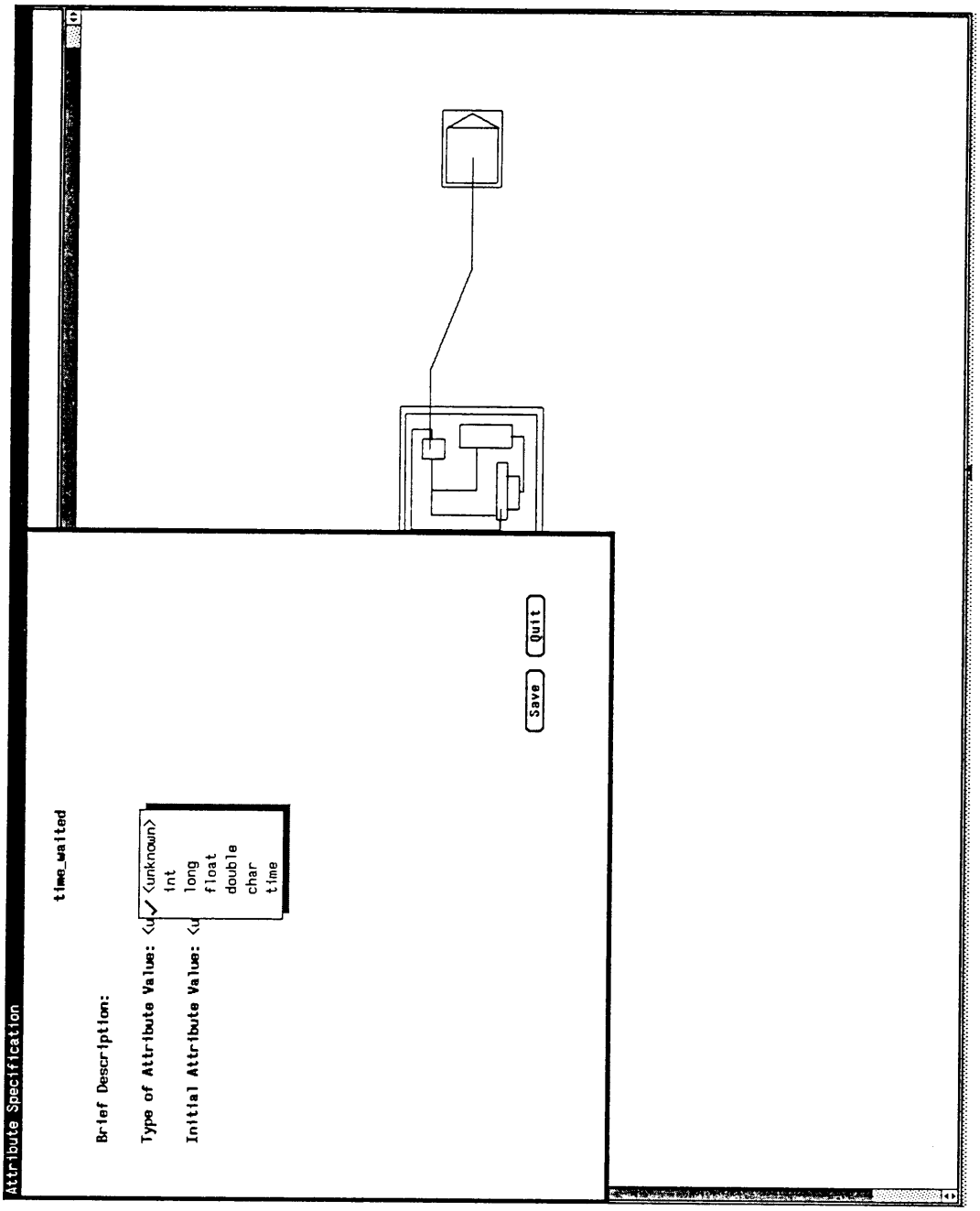

Figure 3.28 Attribute Variable Type Menu

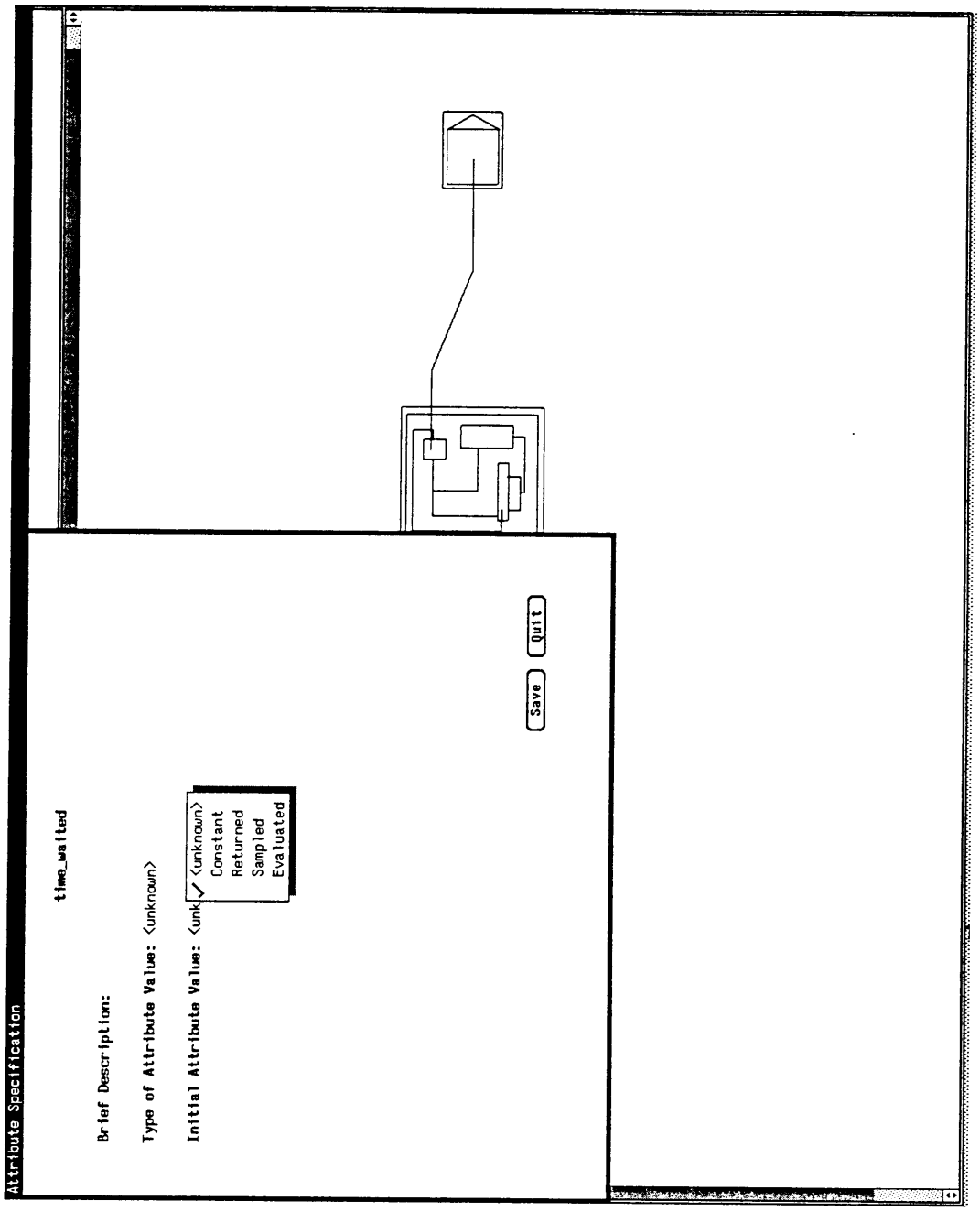

Figure 3.29 Initial Attribute Value Menu

function seed_init() is available if the user wishes to have a random number seed initialized from the pre-defined global seed "seed". It is merely necessary to pass seed_init() the address of the integer seed it generates a random number from (i.e. seed_init(&seed)). "Sampled" allows the attribute to be initialized from a probability distribution (see Figure 3.30). Note that currently only the exp() function is implemented. "Evaluated" allows the user to set the initial value of an attribute to true or false depending on the expression or function supplied at the "Data:" prompt.

It should also be noted here that DOs have a number of attributes pre-defined in GPVSS as shown in Table 3.2. Perhaps the most useful of these attributes is the "q_sm_et" attribute which is usually set to be the queue submodel entry time.

Two buttons labeled "Save" and "Quit" are placed at the bottom of the panel. The "Save" button executes the save_proc() which saves the attribute data in the database and quits the program (see Table 3.1 in Section 3.3.1.5). The "Quit" button executes the function quit_proc() which quits the program without saving.

3.3.2.4 Specification Mode

The specification mode allows the user to specify submodel attributes and submodel logic. The specification

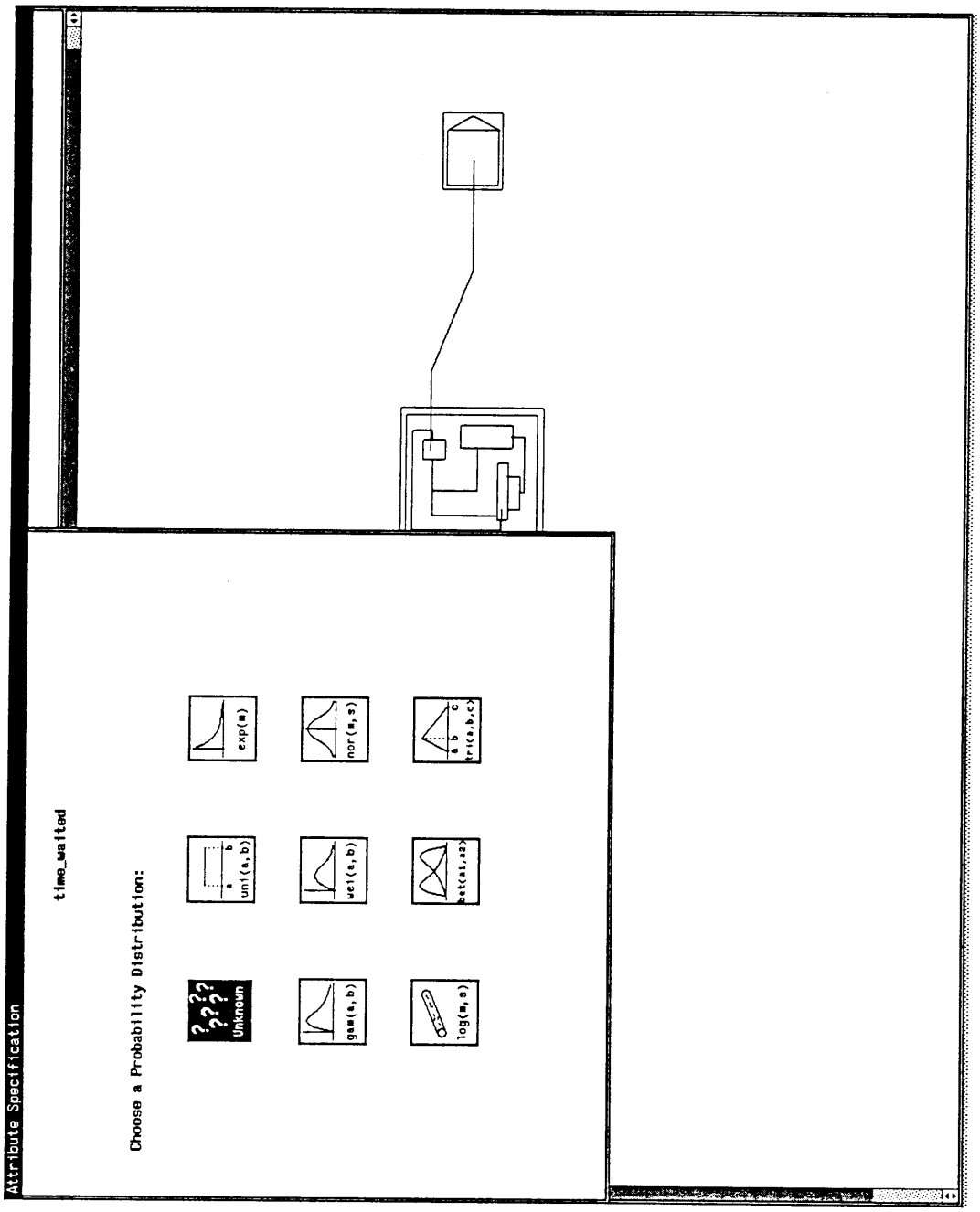

Figure 3.30 IAV Probability Distributions

Table 3.2 Dynamic Object Standard Attributes

Type	Name	Description
double	tag	The unique DO tag
int	type	The type of the DO
int	max_gen	The maximum generated DOs
time	occurence_time	Time of the activity
time	q_sm_et	Queue submodel entrance time
int	first	If DO just entered the system
int	sub_exec	Sub-model sect. to process
char	origin_name[56]	The prev. submodel holding DO
char	dest_name[56]	The next submodel holding DO
int	submodel	The submodel containing the DO
time	entry_time	DO entry time to the system
int	job_origin	Where the DO came from
Pixrect	*DO_image	What the DO looks like
struct	DO_list *next	A pointer to the next DO
struct	DO_list *prev	A pointer to the prev. trans.

mode is entered by selecting the "Spec" button on the main control panel, which executes function ms_proc(). Ms_proc() changes the frame label to reflect the specification mode and the SunView event handler to ms_event_proc().

Ms_event_proc() handles the model specification events. Precisely, it forks off both the submodel attribute and submodel logic specification programs. This is accomplished by means of a menu that is active only when the mouse pointer is within a defined submodel rectangle and the right mouse button is depressed. Available menu choices include "Sub-Model Attribute" and "Sub-Model Logic" (see Figure 3.31).

Selecting the "Sub-Model Attribute" menu item first executes the name_attr() function. Name_attr() brings up the same blocking requester for the input of the attribute name that was used for DOs (see Figure 3.26). The procedure is identical to the process of DO attribute specification described earlier except for the fact that an "S" (for Sub-model attribute) is passed to the attribute program in the command line instead of a "D" (for Dynamic object attribute).

Selection of the "Sub-Model Logic" menu calls function spec_path() which sets up a blocking requester for the input of the specification path (see Figure 3.32). The path is needed to store the multiple files that the spec_log program generates.

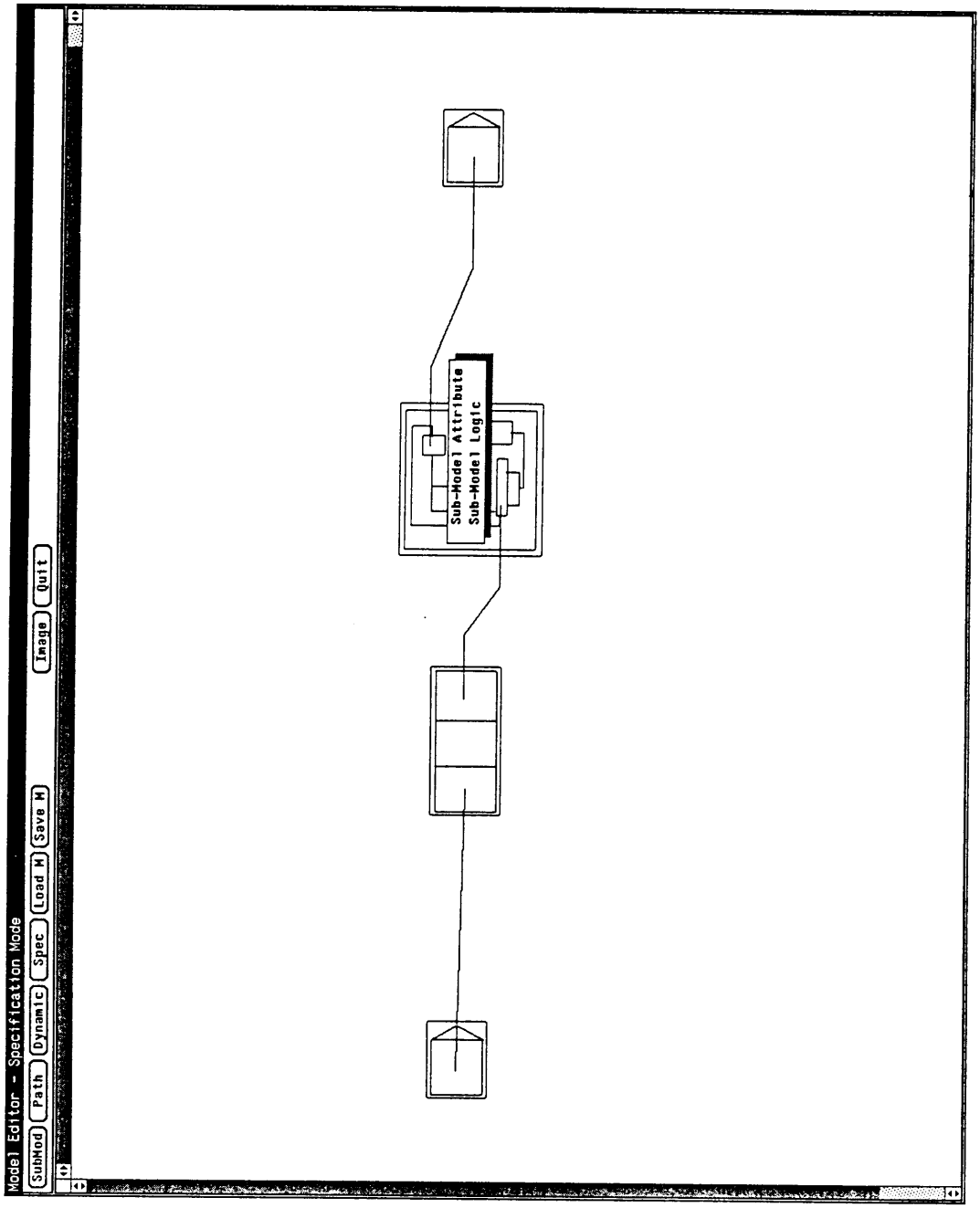

Figure 3.31 Specification Menu

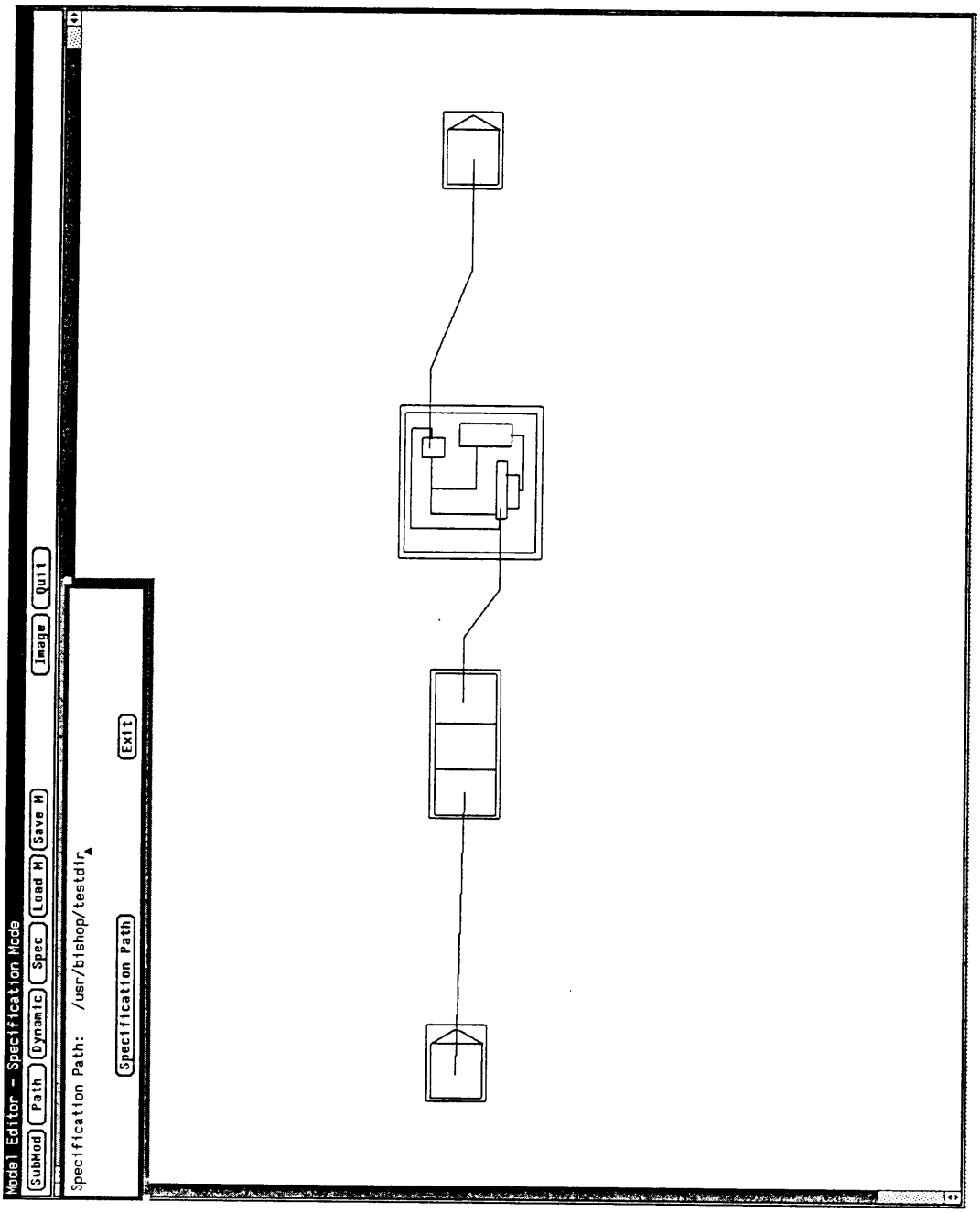

Figure 3.32 Specification Path Requester

The specification path requester has a PANEL_TEXT item labeled "Specification Path:" and two buttons labeled "Specify Logic" and "Exit". The "Exit" button clears the blocking requester without specifying submodel logic. The "Specify Logic" button executes the function spec_proc().

Spec_proc() takes the specified path and store it in the database for future use (see Table 3.1 in Section 3.3.1.5). If no path is specified, then the path defaults to the database value. Note that absolutely nothing must be typed at the "Specification Path:" prompt, including blanks. Finally, spec_proc() clears the blocking requester so that the spec_log program can be called with the system model name and submodel name, and specification path as arguments.

Program spec_log provides the actual submodel specification facilities. Each submodel is broken down into multiple components [Balci 1986]. In addition, a "Global Definition" file is provided for the inclusion of user defined functions.

Spec_log opens a number of SunView Textsw's for each submodel that is specified (see Figure 3.33 and Table 3.3). The Textsw files are stored in the path passed to the program as a command line argument. Two files for the same Textsw but different submodels are distinguished by the submodel name appended to the standard filename. Thus, the entrance condition file ("EC") for the "CPU" submodel would be saved in the "EC_CPU" file in the specified directory.

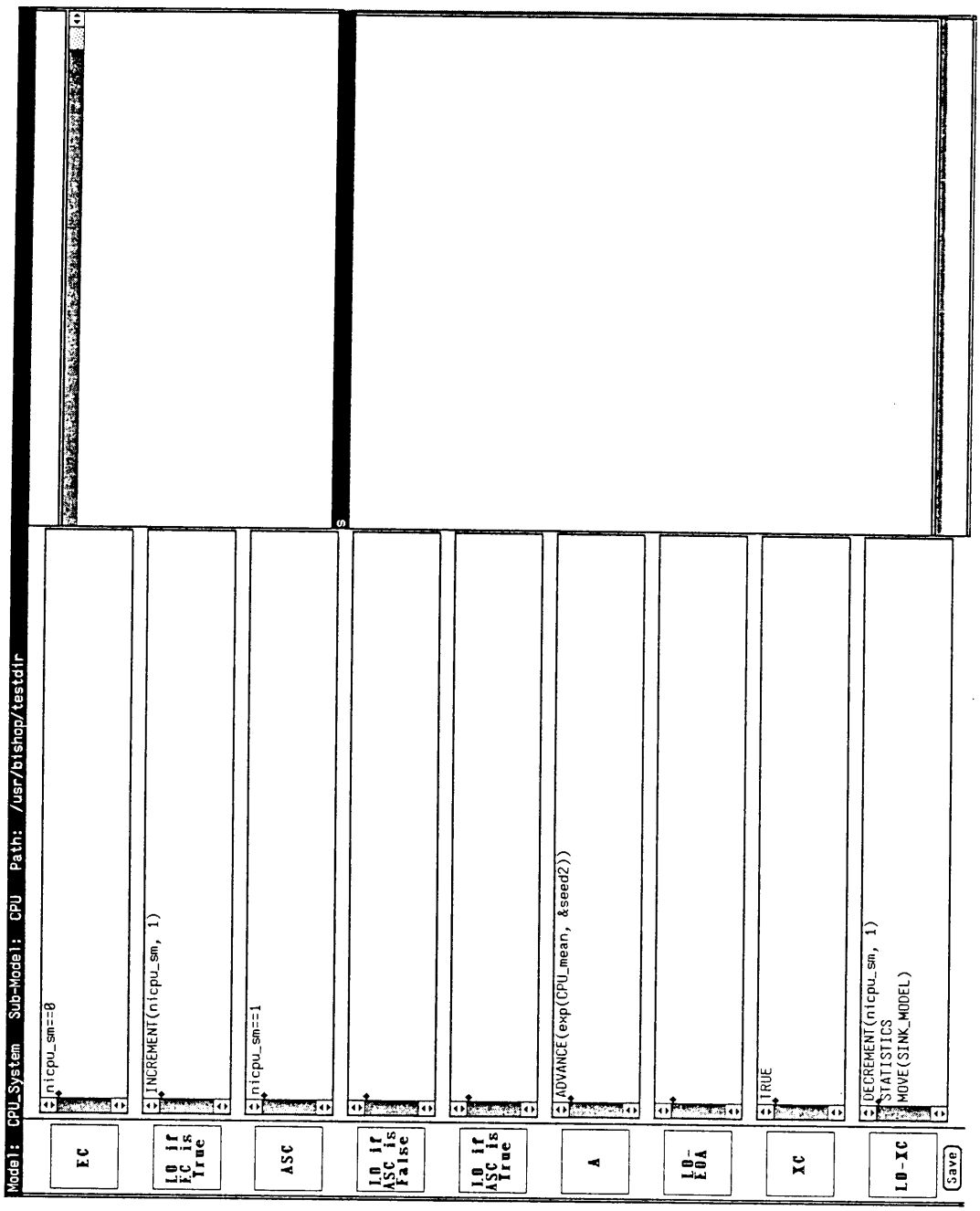

Figure 3.33 Submodel Logic Specification Tool

Table 3.3 Submodel Logic Specification

EC: Entrance conditions that must be true in order to enter the submodel. This may consist of an expression or a function as long as the function evaluates to either true or false.

LO if EC is True: Logical operations to be completed if the entrance condition evaluates to true. These operations must be in standard C code. However, for this and other Textsw's, more user friendly macros have been defined that bring the logical specification up to a higher level.

ASC: Activity start condition that must evaluate to true before the activity in the submodel may begin.

LO if ASC is False: Logical operations that are executed if the activity start condition evaluates to false.

LO if ASC is True: Logical operations that are executed if the activity start condition evaluates to true.

A: The activity of the submodel. Provision must be made for advancing the DO to "LO_EOA" after the activity is completed if not using a pre-defined macro.

LO_EOA: Logical operations to be performed at the end of the activity.

XC: Textsw contains the exit condition that must evaluate to true before a DO may leave the submodel.

LO_XC: Logical operations to be executed if the "XC" condition evaluates to true.

A "Save" button is located at the bottom left of the panel that calls the ms_save function. This saves the contents of each Textsw in the appropriate file.

Note that only one "Global Def" file is used. Therefore it is important to edit only one version of this file when multiple sub-window logic specifications are running, and to save the edited version last.

A number of macros have been defined to expedite the submodel specification process. See Table 3.4 for a presentation of their availability and proper use.

In addition, the function "least(DO, submodelname_MODEL)" is provided to determine if a DO is at the head of a given queue in the specified model.

These macros and functions bring the submodel specification process up to a higher level than exclusively using the standard C programming language, while still maintaining a high degree of flexibility.

3.3.2.5 Load Model

The load model mode is offered to provide the modeler with the ability to load previously defined submodel and path definitions. These definitions are stored in a text file with a unique format.

Submodel definitions are at the beginning of the file. For each submodel a string containing the submodel name, the coordinates of the upper left corner, the width, and the

Table 3.4 Macros

Purpose	Use
ADVANCE(x)	
Prevents a DO from exiting the "A" section of a submodel until time x has passed.	Should only be used in the "A" submodel specification section. This macro calls the function Advance( ).
BRANCH(seed, x, y, z)	
Uses seed to generate a random number. If the generated number is less than x, the macro will MOVE the DO to submodel y. Otherwise the DO will MOVE to submodel z.	Seed must be an integer variable. X must be a floating point value. Submodels are specified by their names followed by the "_MODEL" extension.
COND_BRANCH(x, y, z)	
MOVEs a DO to submodel y if x evaluates to true, or to submodel z otherwise.	X may be a logical argument or a function that returns true or false. Submodels are specified by their names followed by the "_MODEL" extension.
DECREMENT(x, y)	
Decrements the variable x by amount y.	X must be a variable.
DESTROY	
Used when a DO leaves the model.	This macro calls destroy_proc( ), which destroys the DO and tells the program that the DO made a logical advance.
EXIT_SUBMODEL	
Advance the DO directly to the "XC" section.	Used only in the "A" submodel section. The use of this macro is appropriate when the DO spends zero time in a submodel activity.

Table 3.4 Macros Continued

Purpose	Use
INCREMENT(x, y)	
Increments the variable x by amount y.	X must be a variable.
MOVE(x)	
Moves the DO to the "EC" section of the specified submodel x.	The submodel must be specified by the name of the submodel followed by a "_MODEL" extension.
SET_DO(x, y)	
Sets the DO attribute x to the value y.	X and Y must be compatible variable types.
STATISTICS	
Collects the proper statistics when a DO leaves the system.	This macro calls statistics_proc(), and its use is optional.
SYS_ENTRY	
Used to collect the appropriate statistical information when a DO enters the model system.	Initializes DO statistics variables. Use of this macro is optional.
UNTIL(x)	
Prevents a DO from exiting the "A" section of a submodel until condition x evaluates to true.	This macro should only be used in the "A" section of a submodel. X may be a logical argument or a function that returns true or false.

height of the submodel rectangle are stored. "END_OF_MODELS" is placed directly after the last submodel definition in the file.

Path definitions follow the submodel definitions beginning with a string containing the path name. This is followed by two strings containing the "from" and "to" submodel names that define the end points of the path. Next, the x and y coordinates of the path are listed. A arbitrary number of points can make up a path. Therefore, the coordinates -1,-1 denote the end of each path. Note that negative coordinates are impossible otherwise, since the x and y values correspond to actual pixel positions on the screen and the Sun color monitor does not have negative coordinates. "END_OF_PATHS" marks the end of path definitions.

The "Load M" button on the main control panel starts the process of loading a model definition by calling loadm_proc(). This action brings up a blocking requester that has a PANEL_TEXT item labeled "Are you sure?" and two buttons labeled "Load Model" and "Exit" (see Figure 3.34). As discussed previously, the "Exit" button executes exit_proc() which clears the blocking requester without loading a model.

The "Load Model" button executes the model_load() function. This function looks up the filename containing the model definition in the database. Therefore, any text

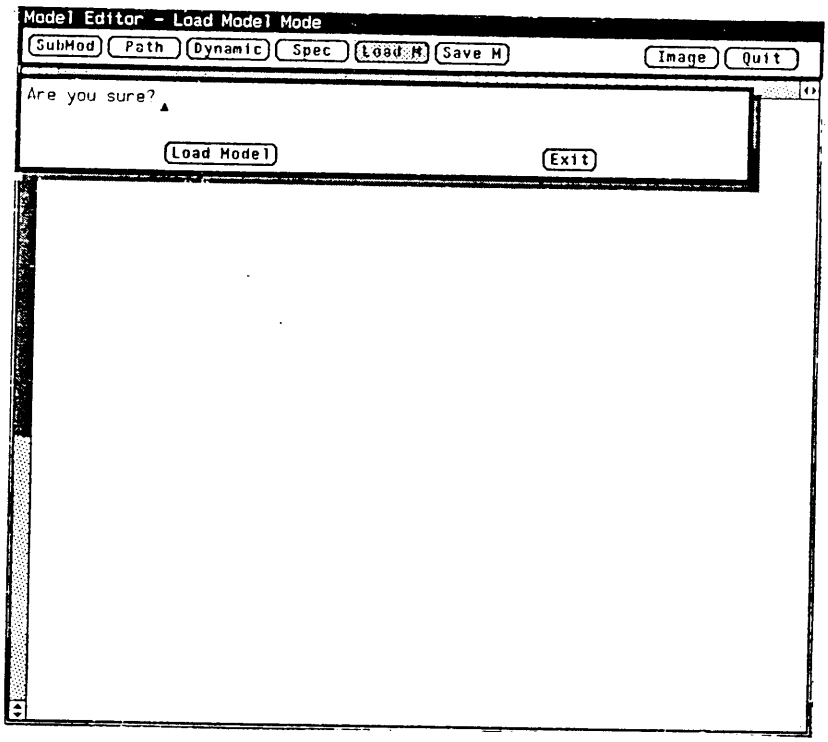

Figure 3.34 Model Load Requester

previously entered at the "Are you sure?" on the blocking requester is ignored. Next any existing path and/or submodel definitions are erased using the XOR logical mask and their memory de-allocated. Finally, the new model definition is loaded into memory and drawn onto the canvas, again using the XOR logical mask.

If the model definition is saved frequently using the "Save M" button, the "Load M" button can also be used as a limited path and submodel definition deletion facility.

3.3.2.6 Save Model

The purpose of the save model mode is to create the file containing submodel and path definitions. The save file format follows that previously discussed in the load model section.

The "Save M" button on the main control panel starts the process of saving a model definition by calling savem_proc(). This brings up a blocking requester with a PANEL_TEXT item labeled "Model:" and buttons labeled "Save Model", "Save/Attach Model" and "Exit" (see Figure 3.35). As mentioned previously, the "Exit" button executes exit_proc() which clears the blocking requester without saving a model.

The "Save Model" button executes function model_save(). Model_save() saves the submodel and path definitions in the file specified at the "Model:" prompt in the blocking

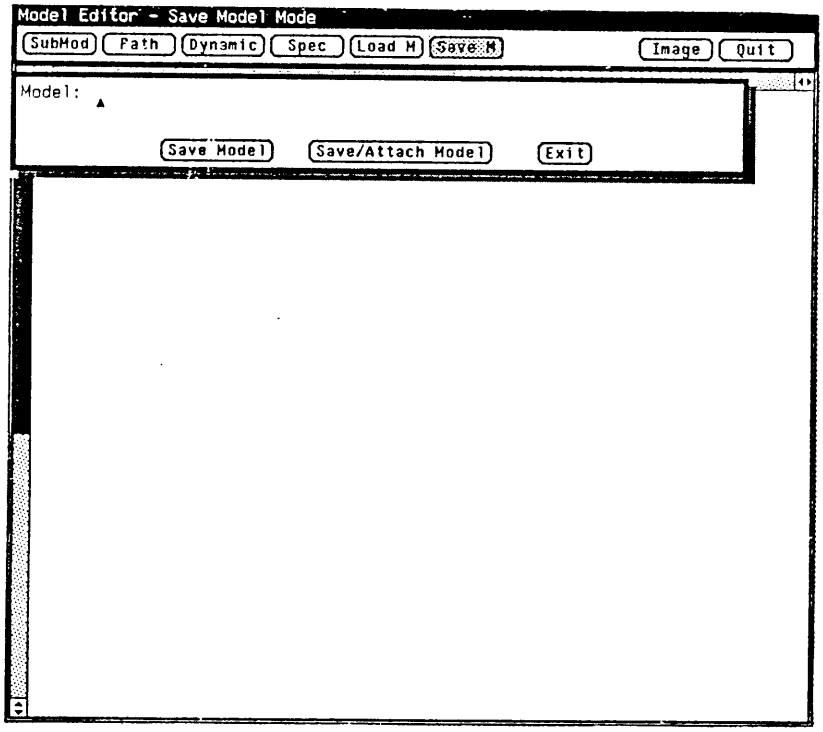

Figure 3.35 Save Model Requester

requester. The usefulness of this option is limited due to the fact that model definition path names are retrieved only from the database.

The "Save/Attach Model" button executes function model_attach(). This saves the submodel and path definitions in a user defined file as mentioned above, but also stores the path in the database (see Table 3.1 in Section 3.3.1.5). If no path is specified, then the path defaults to the database value. Note that absolutely nothing must be typed at the "Model:" prompt, including blanks.

3.4 The Generator

The generator creates the executable code for the visual simulation by invoking the gen program. Selecting the "Model Generator" icon on the main menu serves as the invocation. The gen program accesses several files both in the generator directory and in the directory specified for submodel logic specification as well as making extensive use of the database (see Table 3.1 in Section 3.3.1.5).

The "MASTER_INCLUDE.h" file is created in the same directory as the submodel logic specifications. This file tells the pi (process interaction) program where to find the DO structure definition, the GLOBAL_DEF file, and the function definitions created for each submodel.

For each submodel, a function is created and stored in

a file named for the submodel with a ".h" extension. Thus the function handling the "CPU" submodel would be in the CPU.h file in the submodel logic directory. The created function has the name "CPU_proc()". The various files created in the submodel specification mode of the model editor are opened, and the code is inserted in the proper place.

Next, the pi.h file is created. This file contains all macro definitions and global declarations.

The pi.pre_h is copied into the pi.h file. The purpose of this file is to allow convenient addition and deletion of code without the need to program the changes into the gen program.

Submodel constants are defined for each submodel. These take the form of the model name followed by a "_MODEL" extension. The "CPU" submodel would have a "CPU_MODEL" unique integer constant defined for it. These constants are used by the DOs to determine which submodel function to execute.

Following the submodel constant definitions all user-defined attributes are declared and initialized. The required information to accomplish this is queried from the "attr_dat" table in the database (see Table 3.1 in Section 3.3.1.5). The system model name is defined in the "MODEL_NAME" constant.

Table 3.5 contains descriptions of the additional files

Table 3.5 Program Generation Files

pi.h: Contains all global definitions.

DO_ATTRIBUTE.c: This file is automatically included in the DO_DEF.h file which declares the entire DO structure. Thus user defined attributes apply to all DOs as there is only one DO structure.

DOINIT.c: This file contains the initialize() function. Initialize() is used to initialize all variables between simulation runs, whether they are user defined or not. Each DO specified in the database has it's attributes initialized and is assigned a unique integer identifier which is used for the determination of such things as inter-arrival times, priorities, etc..

DOINIT1.c: This file is used by the initialize() function to determine a DOs inter-arrival time based on its DO type. The code contained in this file is designed to be inserted within a C switch statement on the DO type.

DOINIT2.c: Initializes the user defined DO attributes when the DOs are created.

DOSWITCH.c: Contains code that is designed to be inserted in a C switch statement on the current submodel of a DO. This determines which submodel function the program will execute.

ANIMSWITCH.c: This file contains code designed to be inserted in a C switch statement on the submodel constant. The purpose of this code is to convert the numerical constant submodel names to the string submodel names the animator understands.

required by the gen program to transform the model definition and specification into the programmed model.

To transform the programmed model into the executable model, the gen program issues a system command to "make -f pi_makefile" which compiles and links the pi program.

3.5 The Animation Tool

The capabilities offered by a GPVSS generated simulation include the ability to show moving bitmap images on the dynamic display, and to freeze/restart the animation through user initiated interaction. When the dynamic screen first appears, two buttons labeled "Load" and "Quit" are shown (see Figure 3.6).

The "Quit" button calls quit_proc() which merely quits the program. The "Load" button calls function attach_load() which executes attach_load(). This function retrieves the path name of the background image from the database (see Table 3.1 in Section 3.3.1.5), loads it into a pixrect memory structure, and then draws the image on the dynamic display.

The dynamic display utilizes a technique known as "double buffering" to draw the dynamic objects on the screen [Sun 1988a]. Double buffering provides smooth animation by hiding erasing and redrawing until the next frame is ready to be viewed. Even with rapid regeneration systems, these operations are visible. Double buffering enables images to

be drawn behind the currently visible frame. When drawing is complete, the buffers are switched, thereby making all the objects on the new frame appear to have been drawn instantaneously.

Double buffering is implemented by restricting read and write access to a specific buffer. Limiting write access to a buffer is achieved by dividing in half the bitplanes associated with the canvas using the pw_putattributes() function. Display or read access to a buffer is limited by manipulating the colormap. Two colormaps are constructed: one with colors associated with the low-order bits, the other with the high-order bits. This effectively halves the maximum available bits for color definition. Thus, a 256-color canvas is capable of supporting two 16-color buffers. Display of either the high- or low-order buffer is accomplished by changing the colormap appropriately.

Due to the use of double buffering, the background image must be loaded into the dynamic display twice, once for each buffer. Because write access to the canvas has been masked, any attempt to resize the window results in the elimination of the background image in that buffer. This results in an undesirable flickering effect when the dynamic display is run.

If the background image is successfully loaded, attach_load() performs the remainder of the setting up tasks.. Submodel and path definitions are loaded into

memory from a path name retrieved from the database (see Table 3.1 in Section 3.3.1.5). If no errors were encountered, initialize() is called to initialize all simulation variables. Finally, attach_load() eliminates the "Load" button and brings up the "Animate" and "Run" buttons (see Figure 3.36).

3.5.1 Animation Mode

The "Animate" button starts and stops the dynamic display. It executes the anim_proc() function which enables or disables the repeated execution of the draw_objs() function and whether or not the "Run" button is shown. The calls to draw_objs() are enabled by using notify_set_itimer_func() to make the system repeatedly call draw_objs() as often as possible. The "Run" button is removed due to the fact that the dynamic display must necessarily halt in order for the simulation to execute properly in "Run" mode. This is because both "Animate" and "Run" modes require the use of some of the same global variables, thereby making it impossible to initialize the "Run" model properly while producing a dynamic display. Therefore, making the "Run" button and the dynamic display mutually exclusive avoids this conflict.

Draw_objs() is the heart of the dynamic display facility. It first checks to see if any animation objects on the animation list have a start time less than or equal

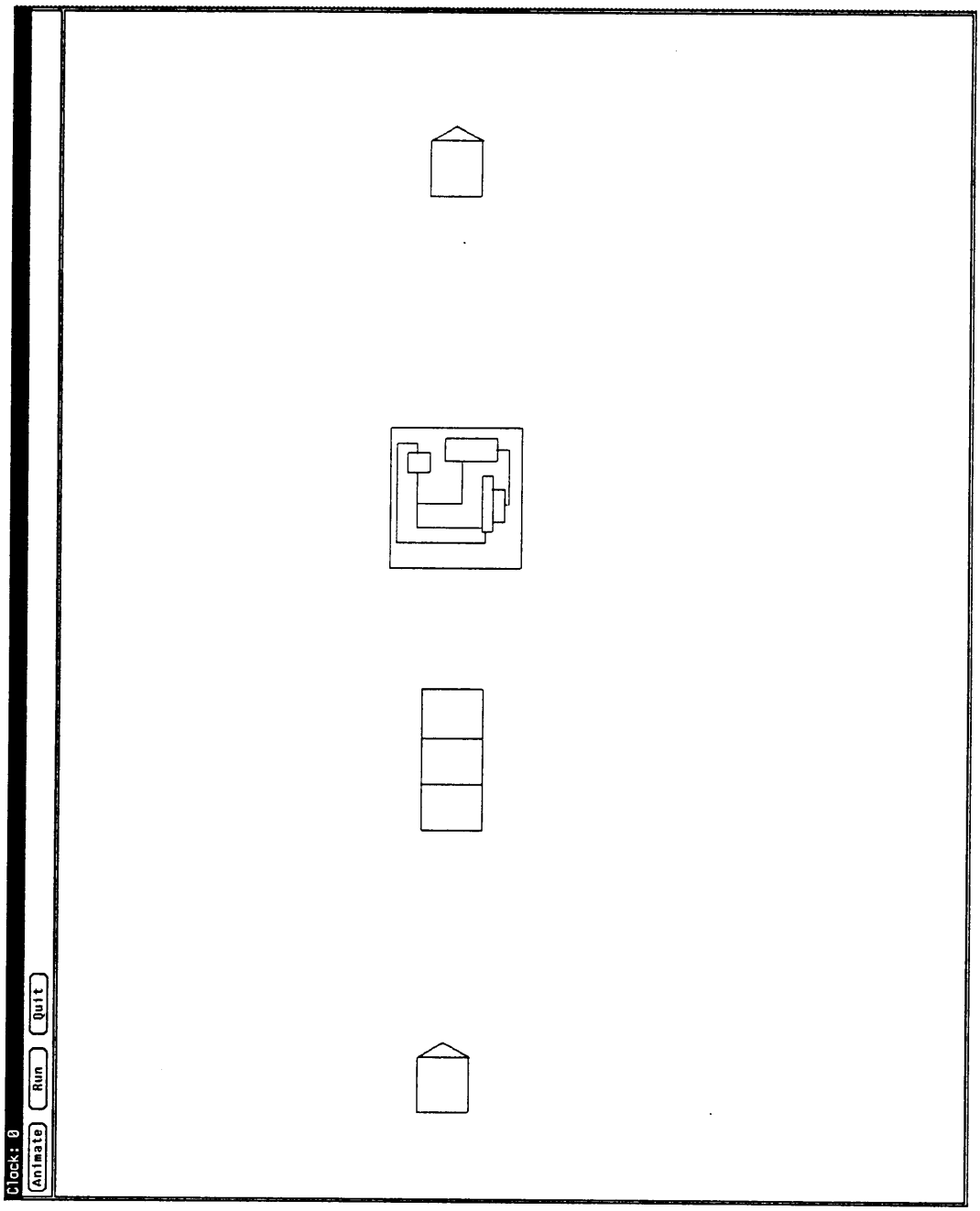

Figure 3.36 Animation Tool with Loaded Model

to the animation clock. If there are, these objects are inserted into the draw list. These animation objects are classified as "EXTEND" objects due to the fact that all movement between submodels is assumed to be instantaneous. Therefore, the animation clock must halt during their movement.

Since all movement between submodels is assumed to be instantaneous in simulation time, a problem arises when an object spends zero time in a submodel. Assume an object moves from submodel A to submodel B, spends zero time in B, and then moves to submodel C. Since both movements occur at identical times, care must be taken so that the same DO is not seen moving on two different paths at the same time. This preserves the sequential logic of the model. Thus, instead of seeing a DO move from A to B and from B to C at the same time, the DO moves first from A to B then from B to C.

This solution was implemented by giving each DO a unique integer tag. If the draw list contains a DO with the same tag, the animation object is placed on a temporary list. When all objects currently on the screen arrive at their respective destinations, the temporary animation object list becomes the draw list again and the above cycle continues.

Code exists to animate object movements that begin at a specified animation clock time, but are otherwise unbounded.

The animation clock does not halt during the animation as is the case for instantaneous object movement. "EXTEND" and "CONCURRENT" animation objects are maintained on separate lists in a manner similar to that described above. No "CONCURRENT" objects move until all the "EXTEND" objects have reached their destination. This code is currently not utilized, but could be used to animate arrivals or departures from the model system. Note that the animation of system arrivals should be led by the proper number of animation time units so that the animation object arrives at its destination at the designated time. The function path_length() computes the number of animation steps required to traverse a given path.

Once all current animation objects have been removed from the animation list and place on the appropriate draw list, the function simulate() is called.

Simulate() contains all code pertaining to the system model. The simulation code was created under the process interaction conceptual framework as described in Figure 3.37 reprinted from [Balci 1988, p. 294]. Simulate() updates the simulation clock to the first event on the Future Object List (FOL) and moves all DOs with occurrence times less than or equal to the simulation clock (which is distinct from the animation clock) from the FOL to the Current Object List (COL). It then attempts to advance each DO as far as possible until no DO may move any further.

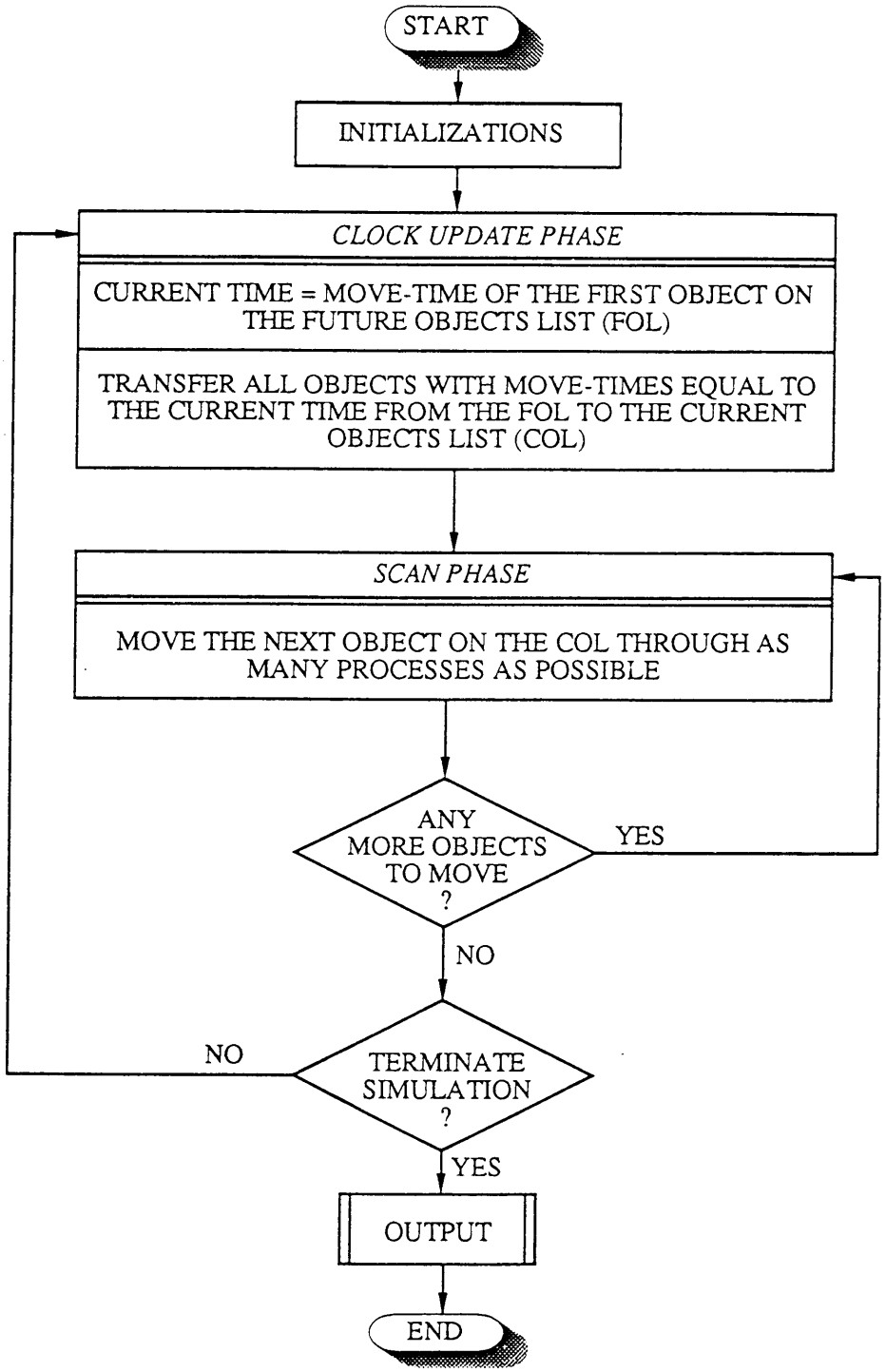

Figure 3.37 Process Interaction Conceptual Framework [Balci 1988, p. 294]

Since the ability to animate "CONCURRENT" animation objects was originally incorporated, simulate() is called until the last simulation DO movement leads the animation by the constant "INTERVAL". "INTERVAL" must be greater than or equal to the longest path length used for "CONCURRENT" animation objects in order to start the animation object early enough to arrive at the correct time.

The FRAME_LABEL of the dynamic display is updated to reflect the current animation clock.

Lastly draw_objs() computes the new positions of the animation objects, draws them in the background buffer, swaps the buffers, and erases the old animation objects.

3.5.2 Run Mode

The run mode provides facilities for the gathering of statistical data using the method of replications. GPVSS reports the average number of DOs in the system, and the average waiting time for DOs.

Selecting the "Run" button executes stat_proc() which brings up the run panel (see Figure 3.38). The run panel provides an interface to enter: (1) "Random Number Seed", (2) "Number of Runs", (3) "Transient Length", and (4) "Steady State Length". Two buttons labeled "Run Simulation" and "Exit" are located at the bottom of the panel.

The "Exit" button calls exit_proc() which clears the blocking requester without running the simulation.

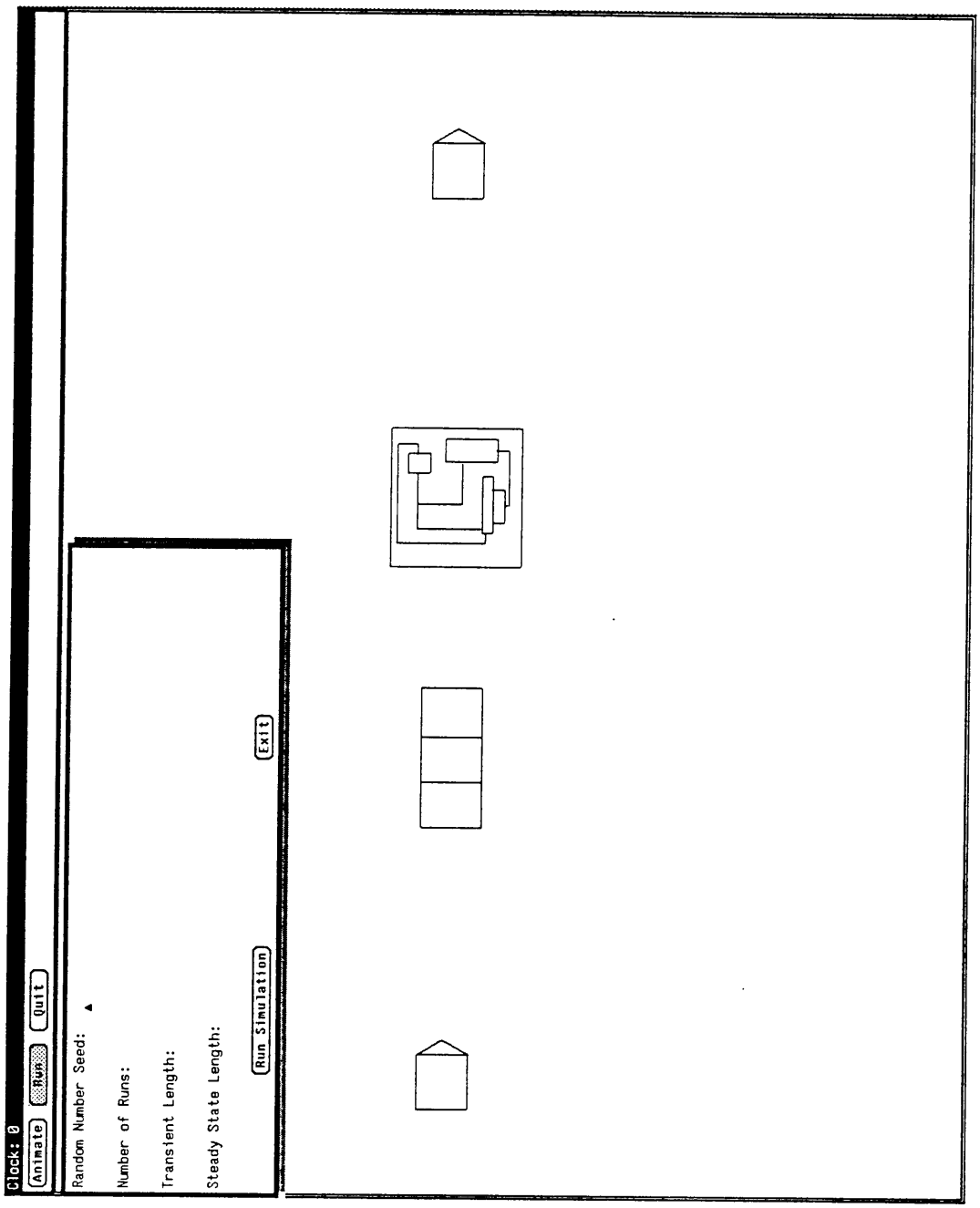

Figure 3.38 The Animation Tool Run Panel

The "Run Simulation" button executes function run_sim(). This function retrieves the data entered by the user on the run panel, initializes all variables, and runs the simulation. The output heading, the average number of DOs in the system for each replication, and the average waiting time of DOs for each replication are all output to standard output. Instead of the animation clock being displayed as the frame label, the number of the current replication is shown. After all replications are complete, run_sim() closes down all windows and quits the program.

CHAPTER 4

CASE STUDIES AND TESTING

4.1 M/M/1 System

A simple computer system has one Central Processing Unit (CPU). Users access the system from a single keyboard device and submit jobs to the computer.

Interarrival times of jobs to the system are exponentially distributed with mean 200 seconds. Service times of the CPU are exponentially distributed with mean 160 seconds.

A GPVSS model of the system was constructed using the facilities described in Chapter 3 (see Figure 4.1). Figure 4.2 shows the running animation of the CPU_System. A job can be seen leaving the CPU, while another is on its way from the CPU_QUEUE to the CPU.

4.1.1 The Experimental Model and Results

Statistics were gathered using the "Run" mode of the animation tool (see Figure 4.3). The system model was assumed to enter steady state after 3000 completed jobs. After reaching steady state, the model was allowed to process 15000 jobs. The method of replications was used to gather 20 values for the average time a DO spent in the system and the average number of DOs in the system.

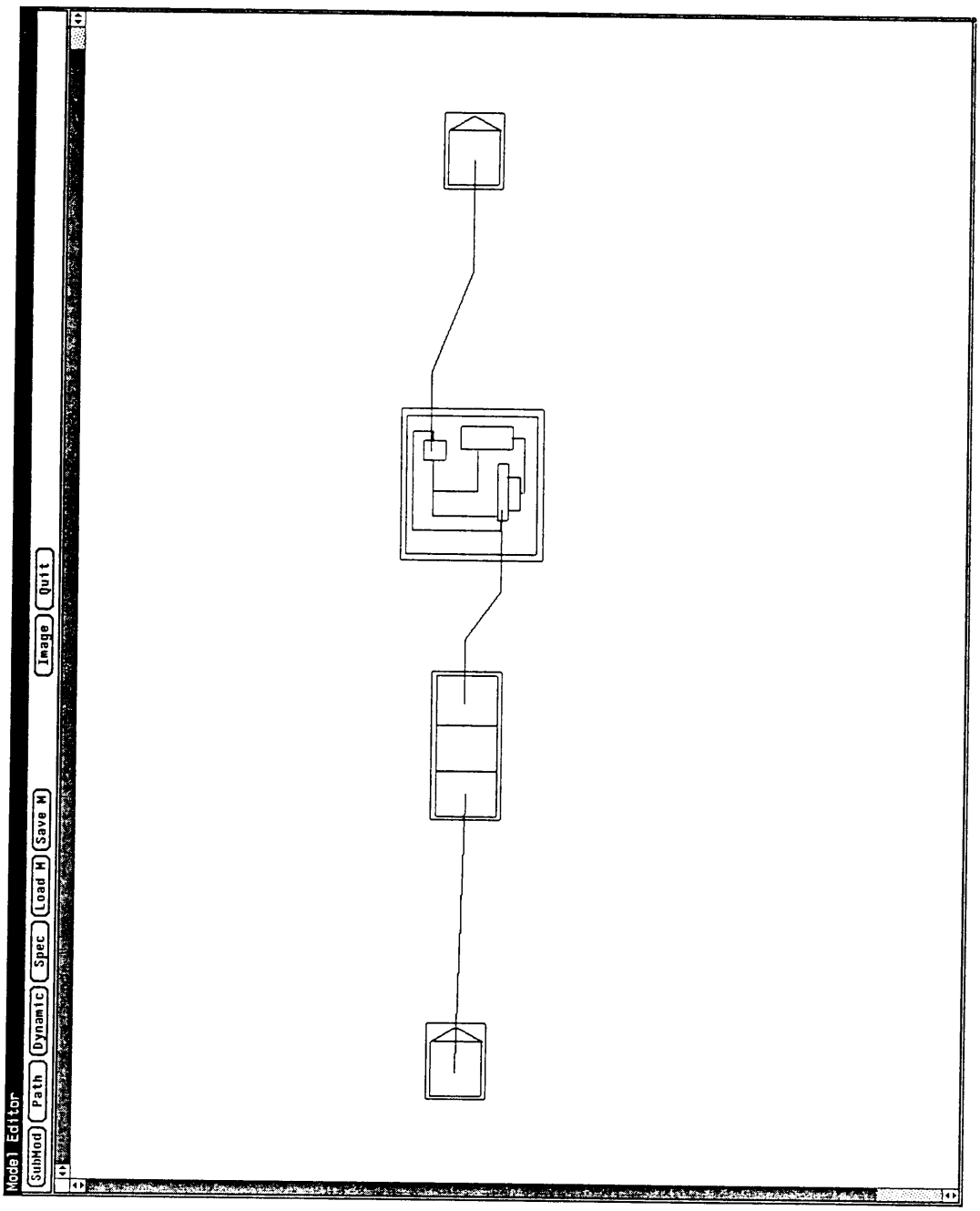

Figure 4.1 M/M/1 Model

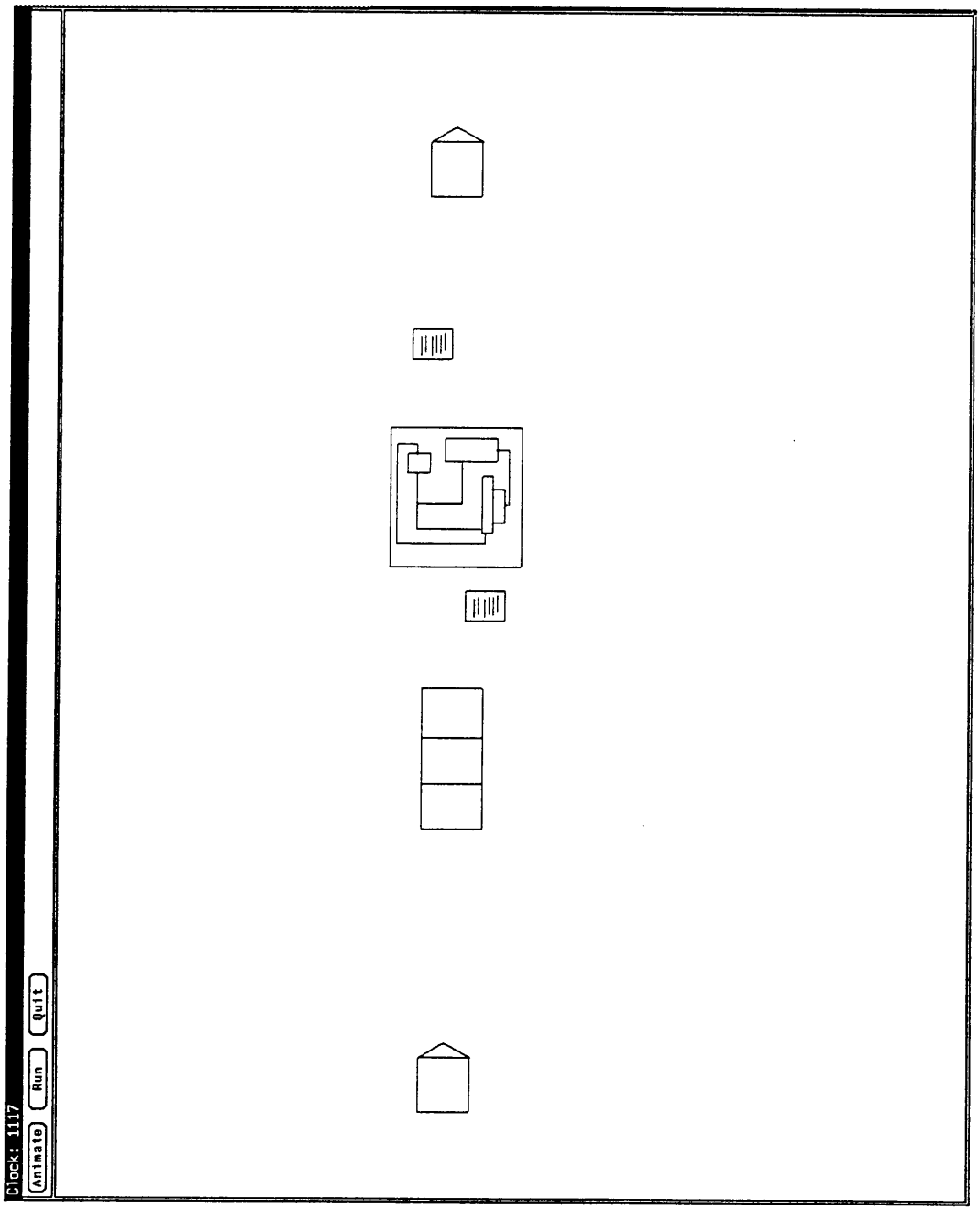

Figure 4.2 M/M/1 Dynamic Display

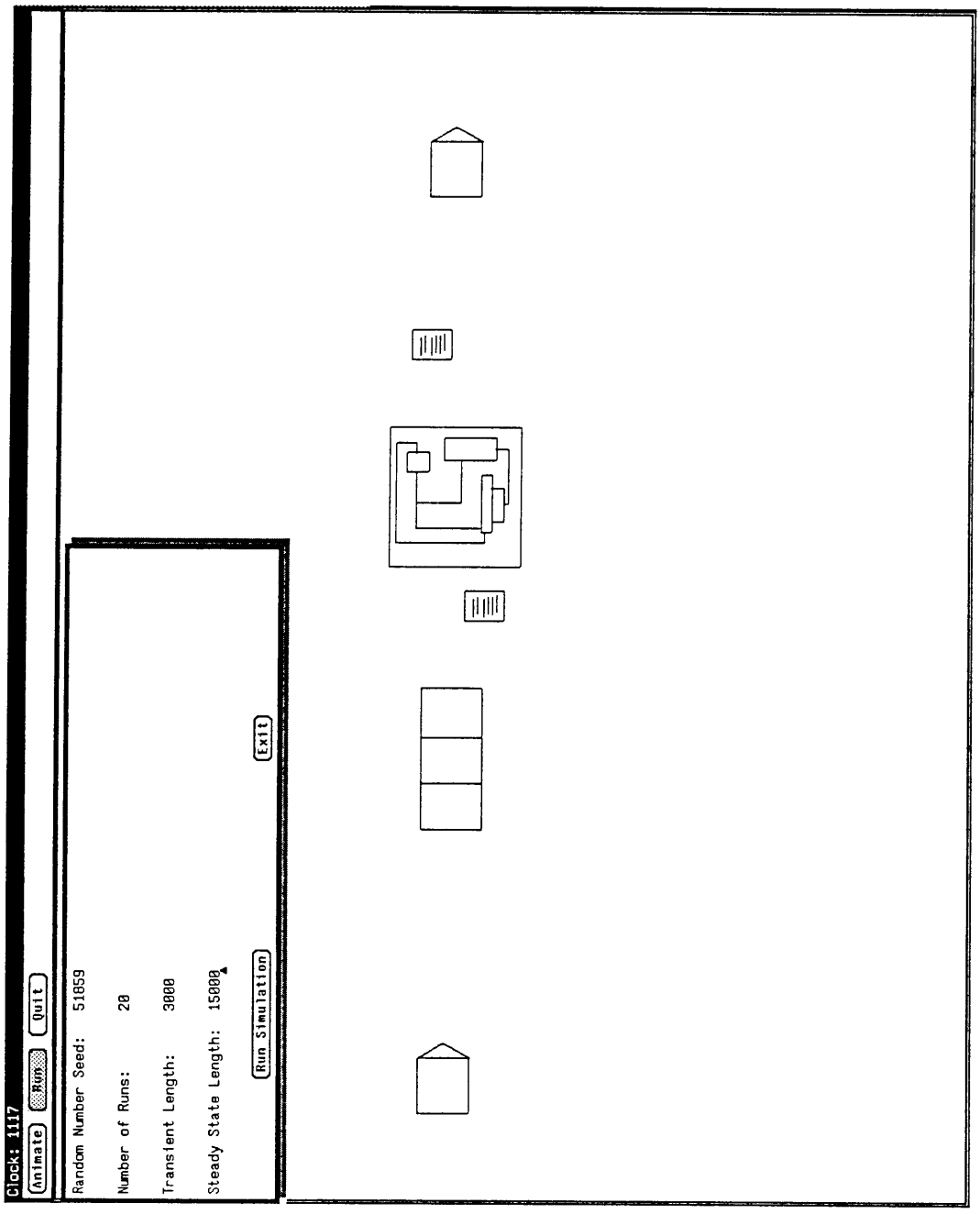

Figure 4.3 M/M/1 Experimental Model

The results of the simulation are listed in Table 4.1. The average time a job spent in the system was between 778.998 and 821.214 seconds at the 95% confidence interval. The average number of jobs in the system was between 3.896 and 4.116 at the 95% confidence interval.

4.2 MVS System

A Multiple Virtual Storage (MVS) batch computer system has two Central Processing Units [Balci 1988]. Users access MVS through an interactive Virtual Memory (VM) computer system running the CMS operating system. Users are classified as follows: (1) 300 baud modem users, (2) 1200 baud modem users, (3) 2400 baud modem users, and (4) LAN users.

Each user creates a batch program under VM/CMS and submits it to the MVS with the distributions listed below.

User Type	Interarrival Times	Mean
300 baud	Exponential	3200 Seconds
1200 baud	Exponential	640 Seconds
2400 baud	Exponential	1600 Seconds
LAN	Exponential	266.67 Seconds

The four facilities in the MVS system are the Job Entry Subsystem (JES) scheduler, processor 1 (CPU1), processor 2 (CPU2), and the printer (PRT). Listed below are the distributions of service times.

Table 4.1 M/M/1 System Experimental Results

Average Waiting Time

VALUES GIVEN:

851.207	825.170	878.756	747.024	707.498
727.253	822.011	851.601	812.012	781.124
826.138	804.737	795.171	710.398	724.980
823.250	798.654	795.861	908.723	810.554

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 800.106140
Sample Variance = 2980.889404

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	783.893	778.998	774.554	769.109	765.178
Upper limit	816.319	821.214	825.658	831.103	835.034

Average Number of Jobs in the System

VALUES GIVEN:

4.290	4.148	4.422	3.722	3.522
3.687	4.143	4.346	4.052	3.896
4.119	4.037	3.964	3.531	3.626
4.081	4.011	3.928	4.545	4.053

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 4.006063
Sample Variance = 0.080510

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	3.922	3.896	3.873	3.845	3.825
Upper limit	4.090	4.116	4.139	4.167	4.188

Facility	Processing Times	Mean
JES	Exponential	112 Seconds
CPU1	Exponential	226.67 Seconds
CPU2	Exponential	300 Seconds
PRT	Exponential	160 Seconds

When a batch job is first submitted by a user, it goes to the JES scheduler. If both queues are empty, the JES scheduler assigns the job to CPU1. This is because CPU1 has the lowest average service time. If one of the queues for CPU1 or CPU2 is not empty, the JES scheduler assigns the job to the shortest queue.

After being serviced by a CPU, all jobs either use the printer (PRT) with probability of 0.2, or leave the system with probability of 0.8.

All jobs are assumed to have equal priority and all queues are handled using the first-come-first-served scheduling discipline.

A GPVSS model of the system was constructed using the facilities described in Chapter 3 (see Figure 4.4). Figure 4.5 shows the running animation of the CPU1_System. A 1200 baud job can be seen leaving the printer, while a 9600 baud job is on its way from the printer queue to the printer.

4.2.1 The Experimental Model and Results

Statistics were gathered using the "Run" mode of the animation tool (see Figure 4.6). The system model was assumed to enter steady state after 3000 completed jobs.

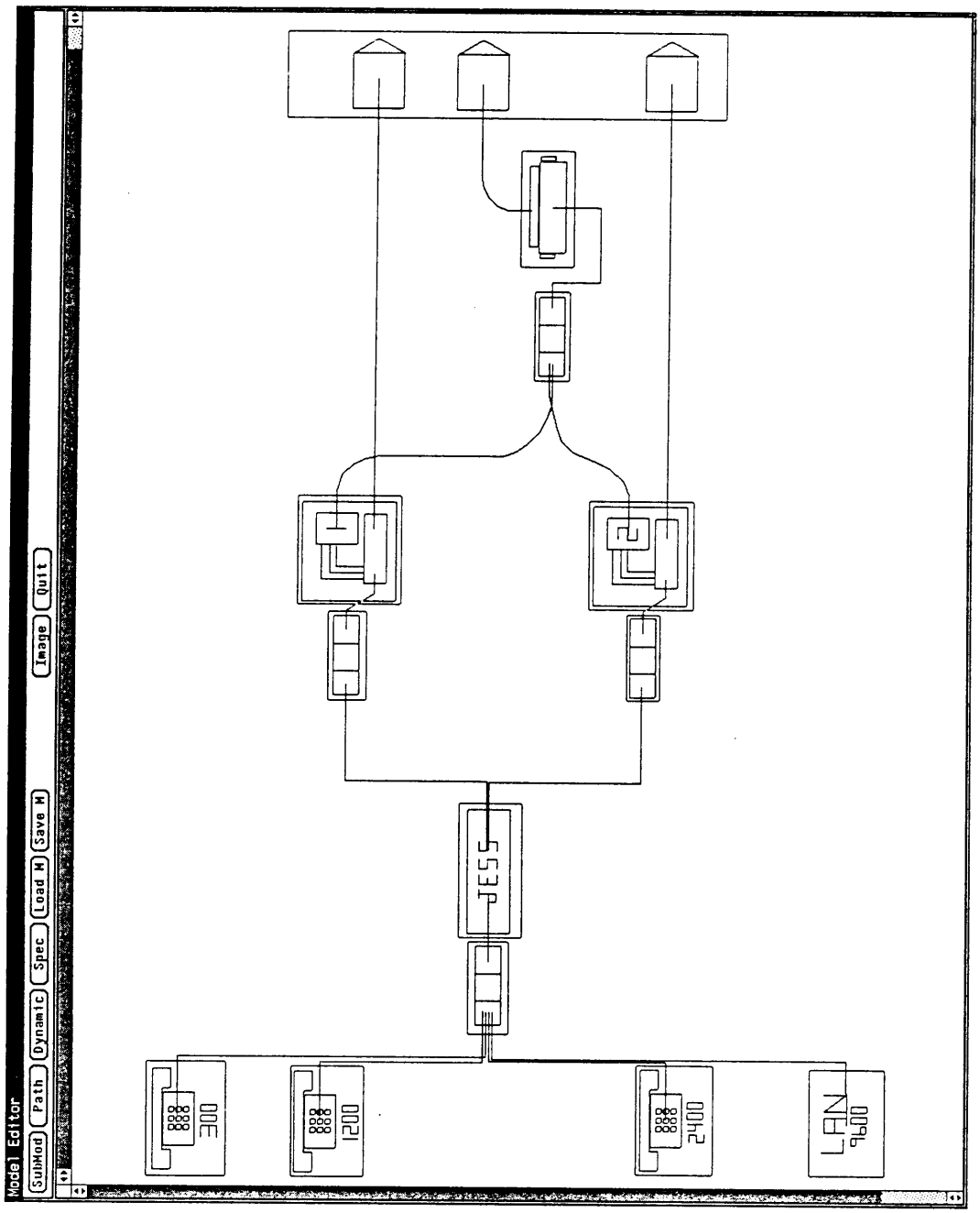

Figure 4.4 MVS Model

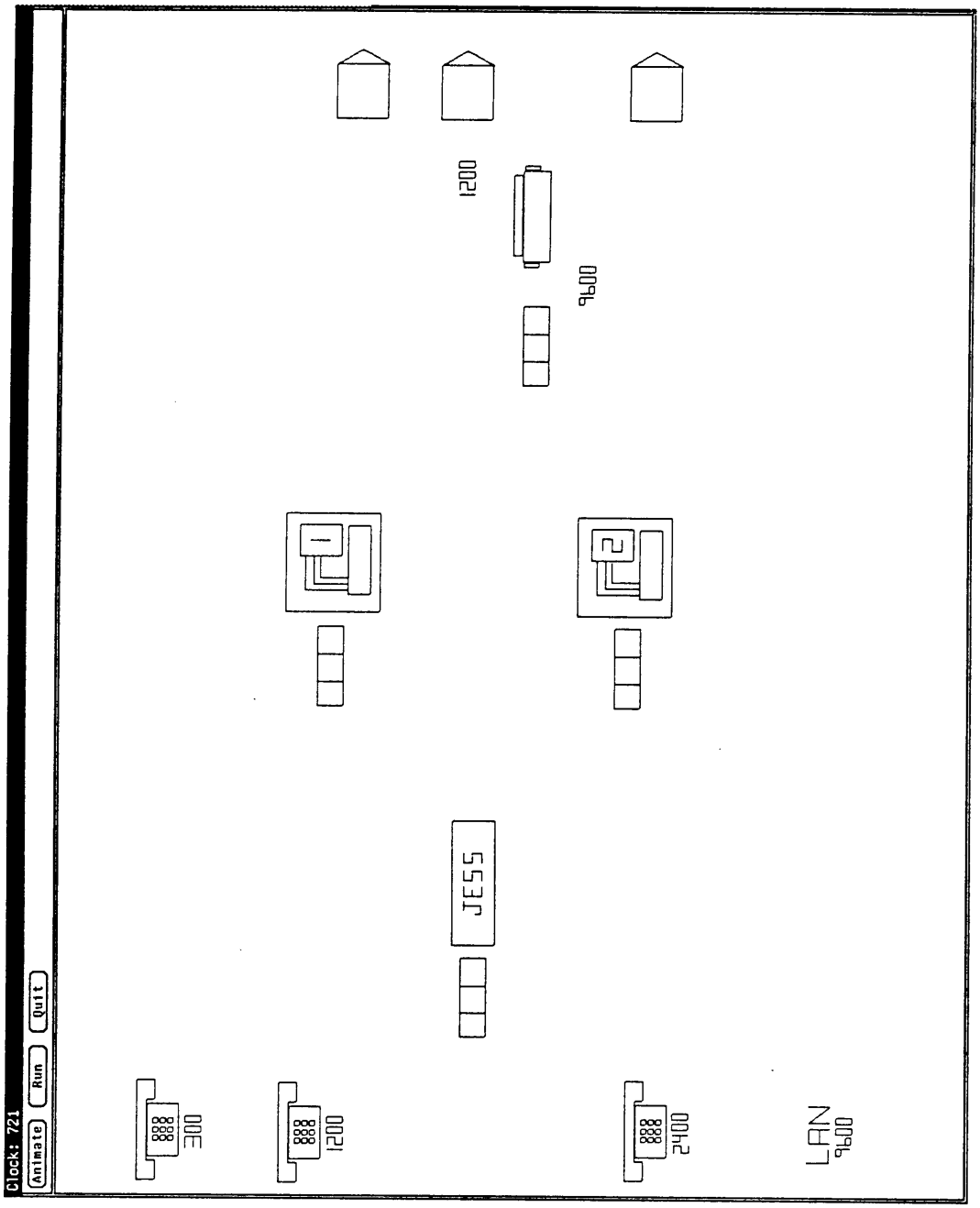

Figure 4.5 MVS Dynamic Display

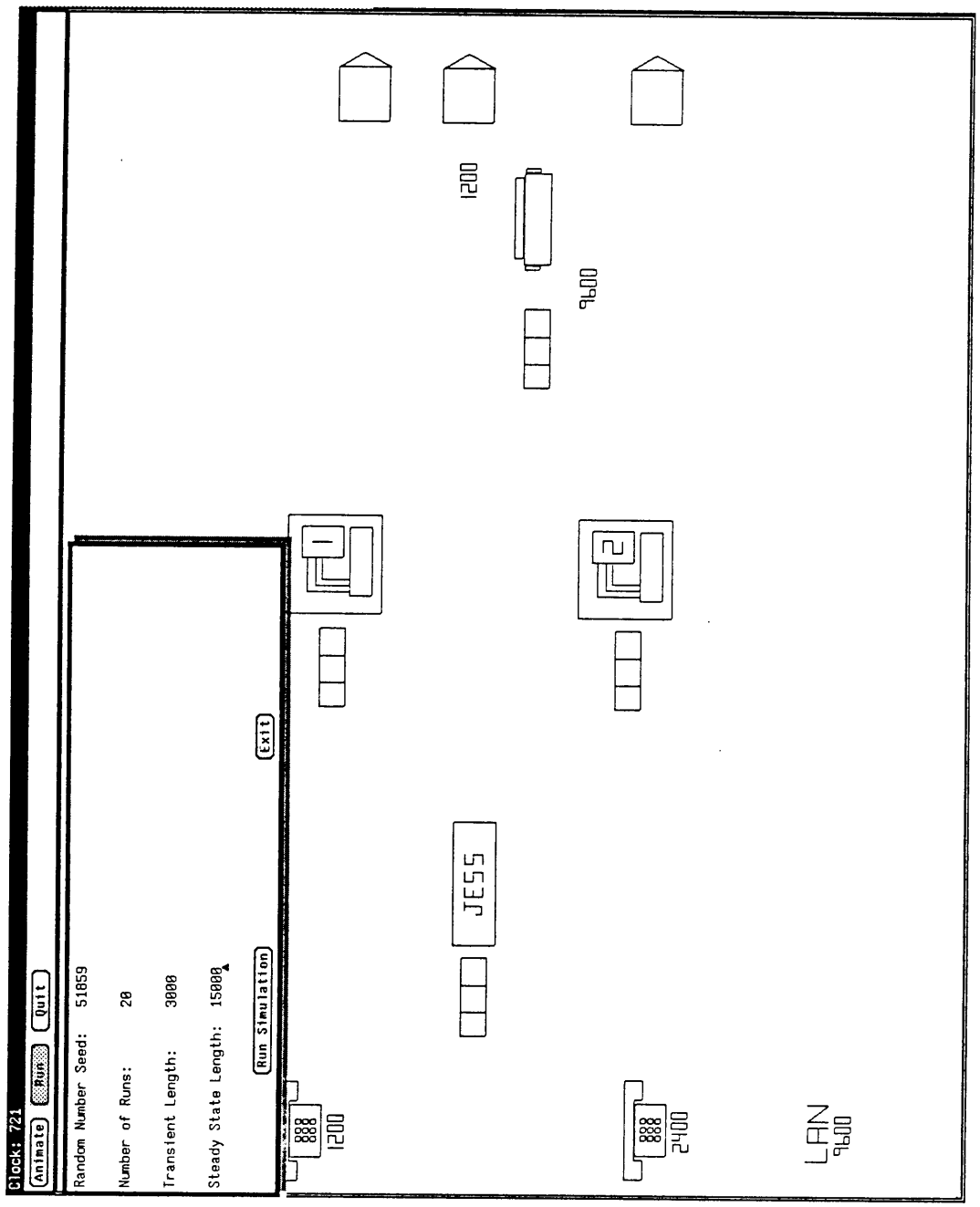

Figure 4.6 MVS Experimental Model

After reaching steady state, the model was allowed to process 15000 jobs. The method of replications was used to gather 20 values for the average time a DO spent in the system and the average number of DOs in the system.

The results of the simulation are listed in Table 4.2. The average time a job spent in the system was between 2370.111 and 2508.026 seconds at the 95% confidence interval. The average number of jobs in the system was between 14.801 and 15.750 at the 95% confidence interval.

4.3 Capacity System

The operating system of a computer installation controls the processing of jobs sent by three independent terminals [Balci 1988].

When a job arrives from a terminal, it enters a queue in front of the Input-Output unit 1 (IO1) and waits for processing. All of the queues in the system are handled in arrival order (First-Come First-Served). After the job is processed at IO1, it enters the queue in front of the Central Processing Unit (CPU). Then, after the job is executed at the CPU, it joins the queue in front of the Input-Output unit 2 (IO2). No limit is placed on the number of jobs waiting in the queue in front of IO1 and CPU, but only ten jobs are allowed at IO2 (*including the one in processing*). If ten jobs are at IO2 (9 in queue and 1 in processing), then any subsequent arrivals to IO2 are

Table 4.2 MVS System Experimental Results

Average Waiting Time

VALUES GIVEN:

2371.152	2296.923	2716.885	2487.397	2303.139
2441.727	2371.648	2681.721	2441.719	2294.878
2372.147	2961.429	2533.762	2381.045	2180.745
2382.213	2344.331	2296.519	2522.246	2399.740

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 2439.068359
Sample Variance = 31812.566406

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	2386.104	2370.111	2355.594	2337.806	2324.964
Upper limit	2492.032	2508.026	2522.543	2540.331	2553.173

Average Number of Jobs in the System

VALUES GIVEN:

14.733	14.319	17.175	15.677	14.308
15.390	14.902	17.006	15.216	14.228
14.764	18.831	15.732	14.888	13.548
14.784	14.705	14.256	16.026	15.029

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 15.275742
Sample Variance = 1.507309

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	14.911	14.801	14.701	14.579	14.490
Upper limit	15.640	15.750	15.850	15.973	16.061

diverted to another I/O unit that is not included in our study.

Statistics of the distributions of interarrival times between jobs and the processing times at each service center are tabulated below. All values are stated in milliseconds.

Terminal	Interarrival Times	Mean
1	Exponential	2000
2	Exponential	4000
3	Exponential	1250

Facility	Processing Times	Mean
IO1	Exponential	450
CPU	Exponential	500
IO2	Exponential	580

A GPVSS model of the system was constructed using the facilities described in Chapter 3 (see Figure 4.7). Figure 4.8 shows the running animation of the CPU2_System. A terminal 1 job can be seen leaving the IO1 device, while a terminal 3 job is on its way from the IO1 queue to IO1.

4.3.1 The Experimental Model and Results

Statistics were gathered using the "Run" mode of the animation tool (see Figure 4.9). The system model was assumed to enter steady state after 3000 completed jobs. After reaching steady state, the model was allowed to process 15000 jobs. The method of replications was used to gather 20 values for the average time a DO spent in the system and the average number of DOs in the system.

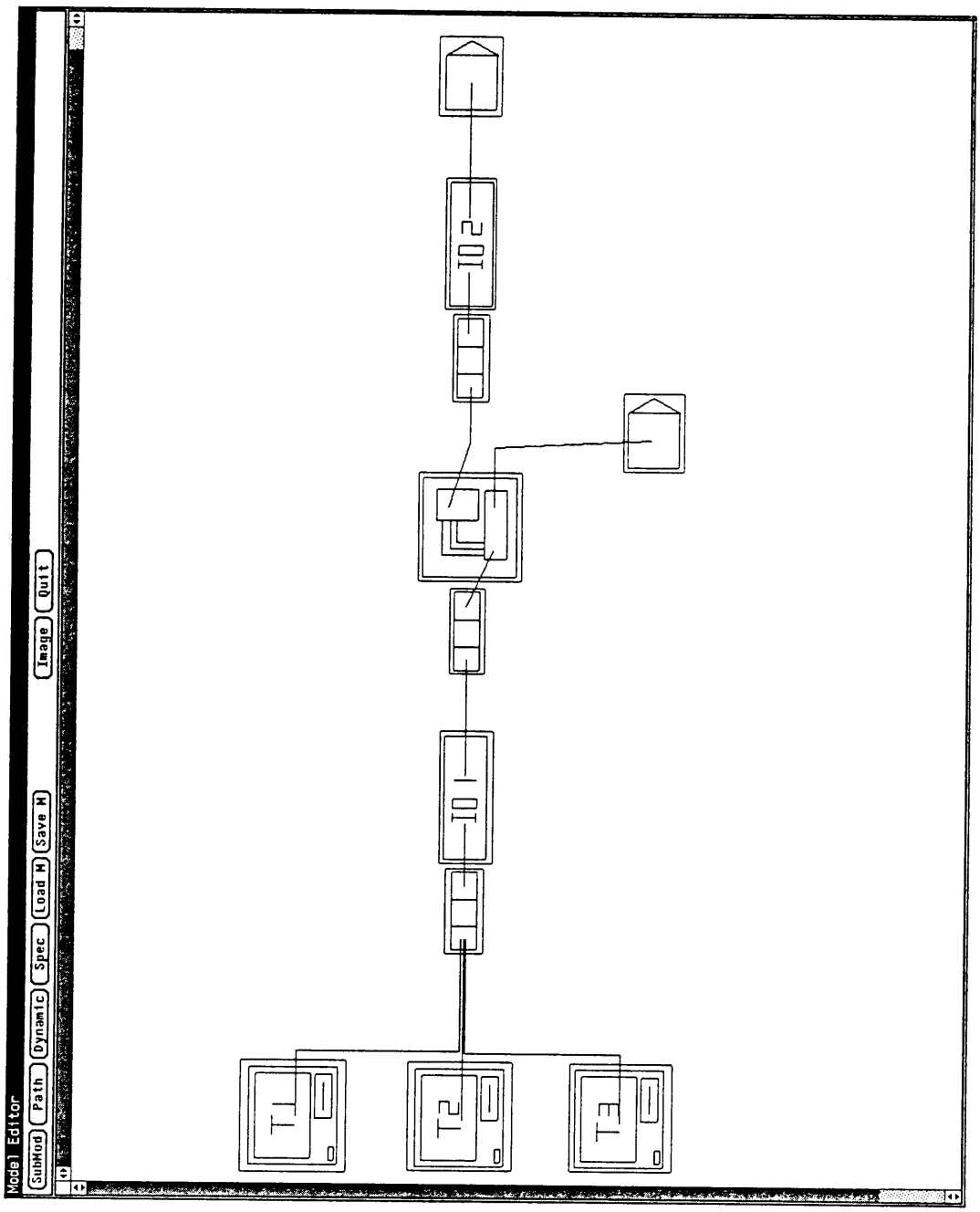

Figure 4.7 Capacity Model

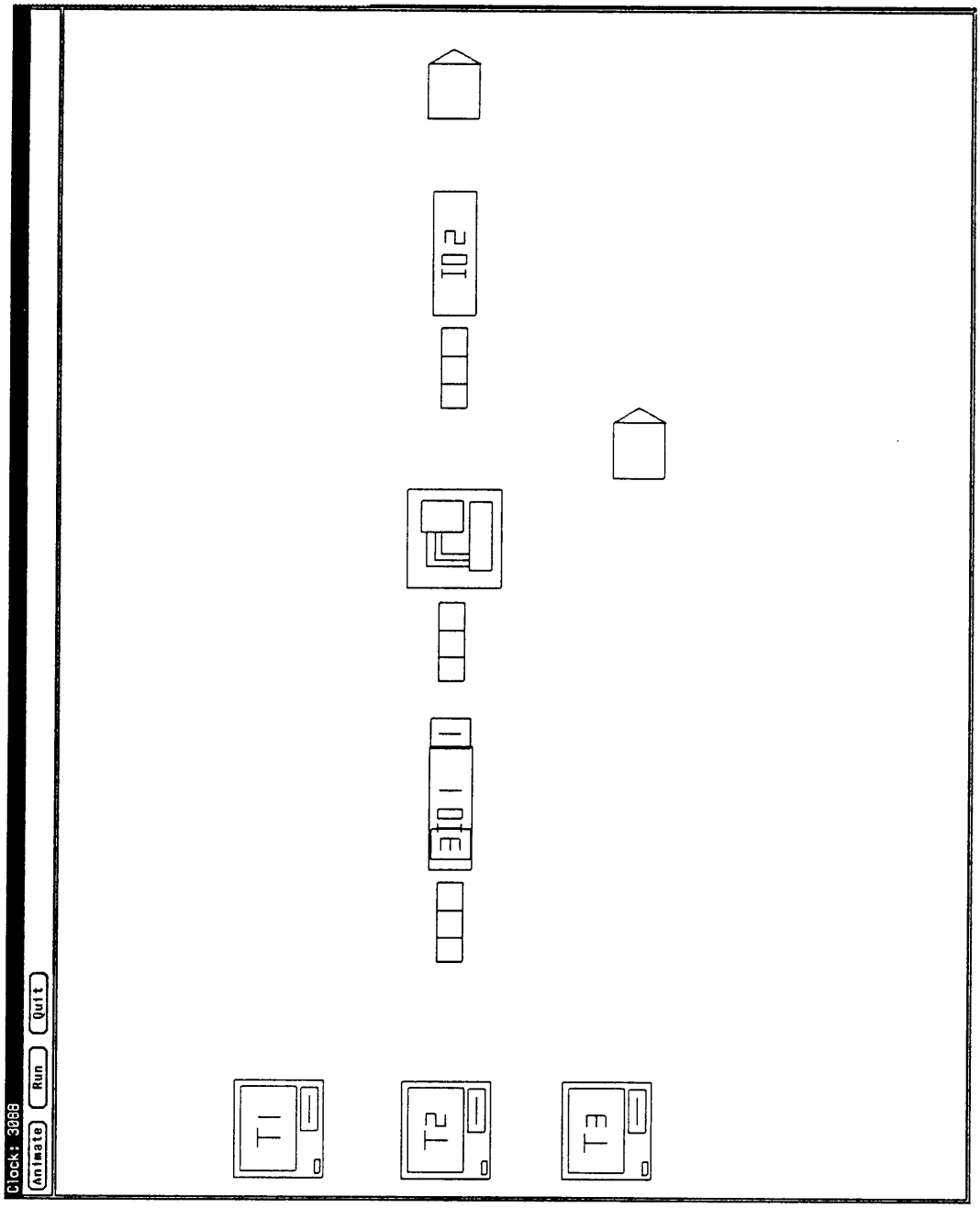

Figure 4.8 Capacity Dynamic Display

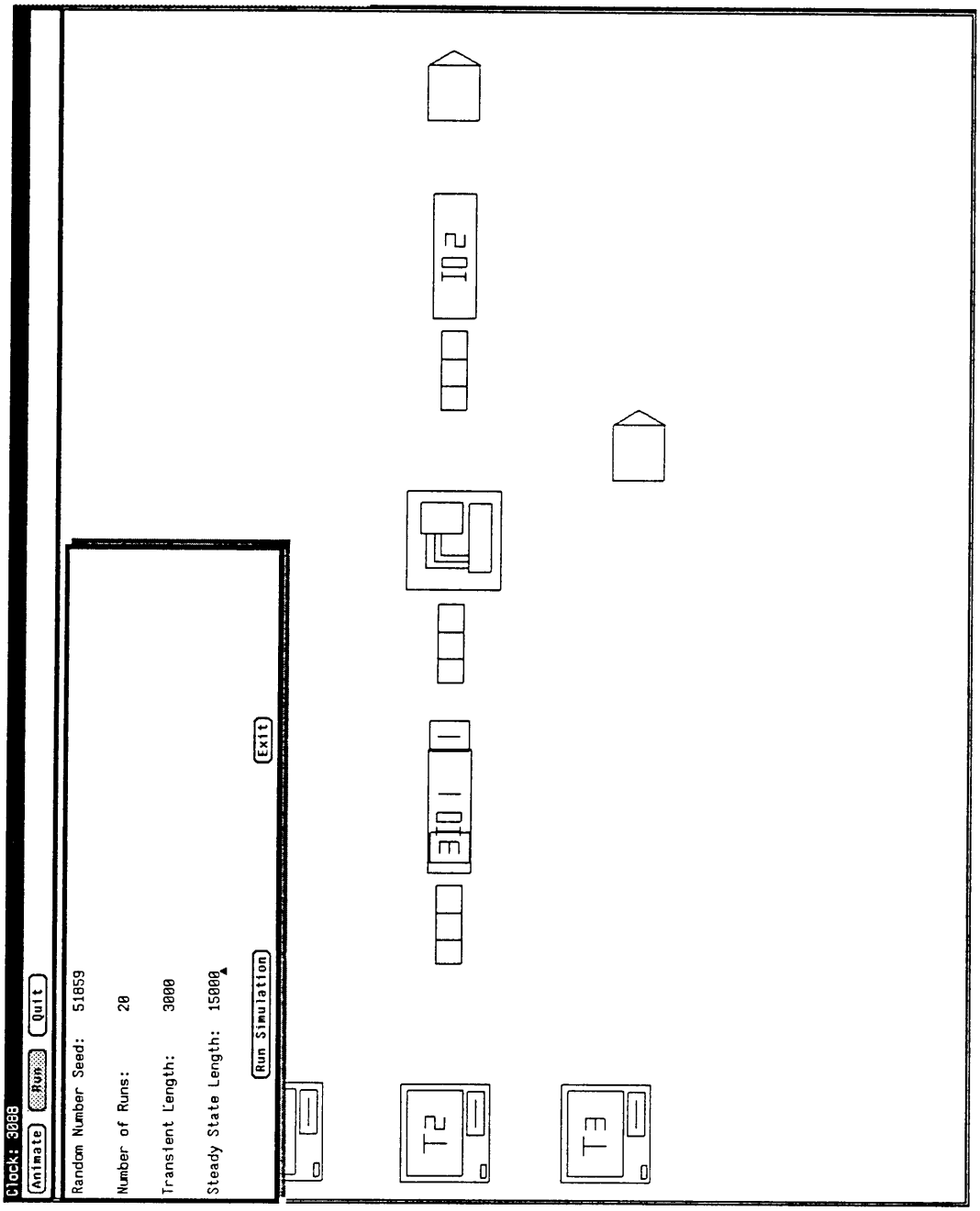

Figure 4.9 Capacity Experimental Model

The results of the simulation are listed in Table 4.3. The average time a job spent in the system was between 6175.095 and 6352.312 seconds at the 95% confidence interval. The average number of jobs in the system was between 9.546 and 9.855 at the 95% confidence interval.

4.4 Testing

A simulation model is only useful it produces sufficiently credible results. The internal logic of GPVSS and the logic of the model must be shown to be sufficiently accurate. Testing refers to both verification and validation. Model Verification refers to building the model right. Model Validation refers to building the right model. Model verification and validation are therefore required if the user is to have confidence in the statistical results.

4.4.1 Verification

Both *desk checking* and *tracing* techniques were used to achieve model verification. Desk checking consists of carefully and thoroughly examining the GPVSS code for errors. Debugging the code was an incremental process. As each new feature was added, as many bugs as possible were eliminated. Additions to the code incorporated as much previously debugged code as possible, thereby minimizing the number of errors.

Tracing was accomplished by following all DOs as they

Table 4.3 Capacity System Experimental Results

Average Waiting Time

VALUES GIVEN:

6391.875	6225.573	6215.359	6033.398	6324.733
5928.592	5964.760	6224.833	6168.832	6198.975
6384.520	6423.599	6275.756	6630.325	6633.741
6708.255	6113.443	6406.245	6121.348	5899.883

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 6263.703125
Sample Variance = 52527.960938

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	6195.645	6175.095	6156.440	6133.583	6117.082
Upper limit	6331.761	6352.312	6370.966	6393.823	6410.325

Average Number of Jobs in the System

VALUES GIVEN:

9.871	9.730	9.663	9.365	9.903
9.094	9.152	9.602	9.529	9.626
9.918	9.918	9.602	10.422	10.268
10.490	9.386	9.923	9.428	9.120

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 9.700429
Sample Variance = 0.159356

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	9.582	9.546	9.514	9.474	9.445
Upper limit	9.819	9.855	9.887	9.927	9.956

moved from one submodel to another. The "DEBUG" variable is used to toggle the GPVSS verification facilities on and off. These remain in the code, but are currently inactive. If the "DEBUG" constant is defined as true, information regarding the DO type, from submodel, to submodel, start time, and DO tag is printed to standard output each time a DO movement occurs. The standard output can be re-directed to a file so that careful analysis of relatively long traces may be performed.

4.4.2 Validation

Validation was performed by comparing the results obtained from the three case studies discussed above with known analytical model results.

Actual values for the M/M/1 computer system described in Section 4.1 are 800 seconds for the average waiting time and 4.0 for the average number of jobs in the system. Comparison of these analytical results obtained from the GPVSS implementation shows that these values fall within all confidence intervals listed in Table 4.1.

Actual values for the MVS batch computer system described in section 4.2 are 2400 seconds for the average waiting time and 15 for the average number of jobs in the system. Comparison of these analytical results obtained from the GPVSS implementation shows that these values fall within all confidence intervals listed in Table 4.2.

Actual values for the capacity computer system described in Section 4.3 are 6264.047 milliseconds for the average waiting time and 9.709 for the average number of jobs in the system. Comparison of these analytical results obtained from the GPVSS implementation shows that these values fall within all confidence intervals listed in table 4.3.

Thus the statistics produced by the GPVSS models of each of the three previously discussed systems compare favorably with the analytical results substantiating sufficient model validity.

CHAPTER 5

USER'S MANUAL

This chapter illustrates the usage of GPVSS. For a case study problem described in the next section, it shows how to develop and execute a visual simulation model step by step.

5.1 Round Robin System

The operating system of a computer installation with a single CPU controls the processing of jobs sent by three independent terminals in a round-robin manner.

The user of each terminal "thinks" for an amount of time and sends a job to the computer system for execution. Think times are exponentially distributed with mean 250.00 milliseconds. The user is inactive until the computer's response starts appearing on the terminal. Thereupon, the user goes through a "thinking" process again and sends the next job. The arriving jobs join a single queue, with first come first served (FCFS) discipline on CPU access. Repeated usage of the CPU is accomplished in a round-robin manner. The CPU allocates to each job a maximum service quantum of length QUANTUM (100.00) milliseconds (not including overhead). If the job is finished within this service time period, it spends a fixed overhead time OVERHEAD (15.00)

milliseconds at the CPU after which a response is sent to the originating terminal. If the job is not finished within QUANTUM, its remaining service time is decremented by QUANTUM and it is placed at the end of the queue after spending a fixed overhead time of OVERHEAD. Job service times are exponentially distributed with mean 500.00 milliseconds. All transmission times between the terminals and the host are assumed to be negligible.

5.2 Starting GPVSS

Before starting GPVSS, it is a good idea to have a directory already created in which the various files associated with the model are stored. In this case, the directory used is "/usr/bishop/test3dir".

To activate GPVSS, make sure that "/usr/bishop/work/top_level" is the current directory and type "GPVSS<CR>". This brings up the GPVSS main menu (see Figure 5.1).

5.3 Drawing Images

The first step in constructing the model is to draw the images that are used in the dynamic display. Two types of images must be created: the system background image and the Dynamic Object (DO) images.

To create the system background image, enter the model editor by first selecting the "Image Editor/Model Editor"

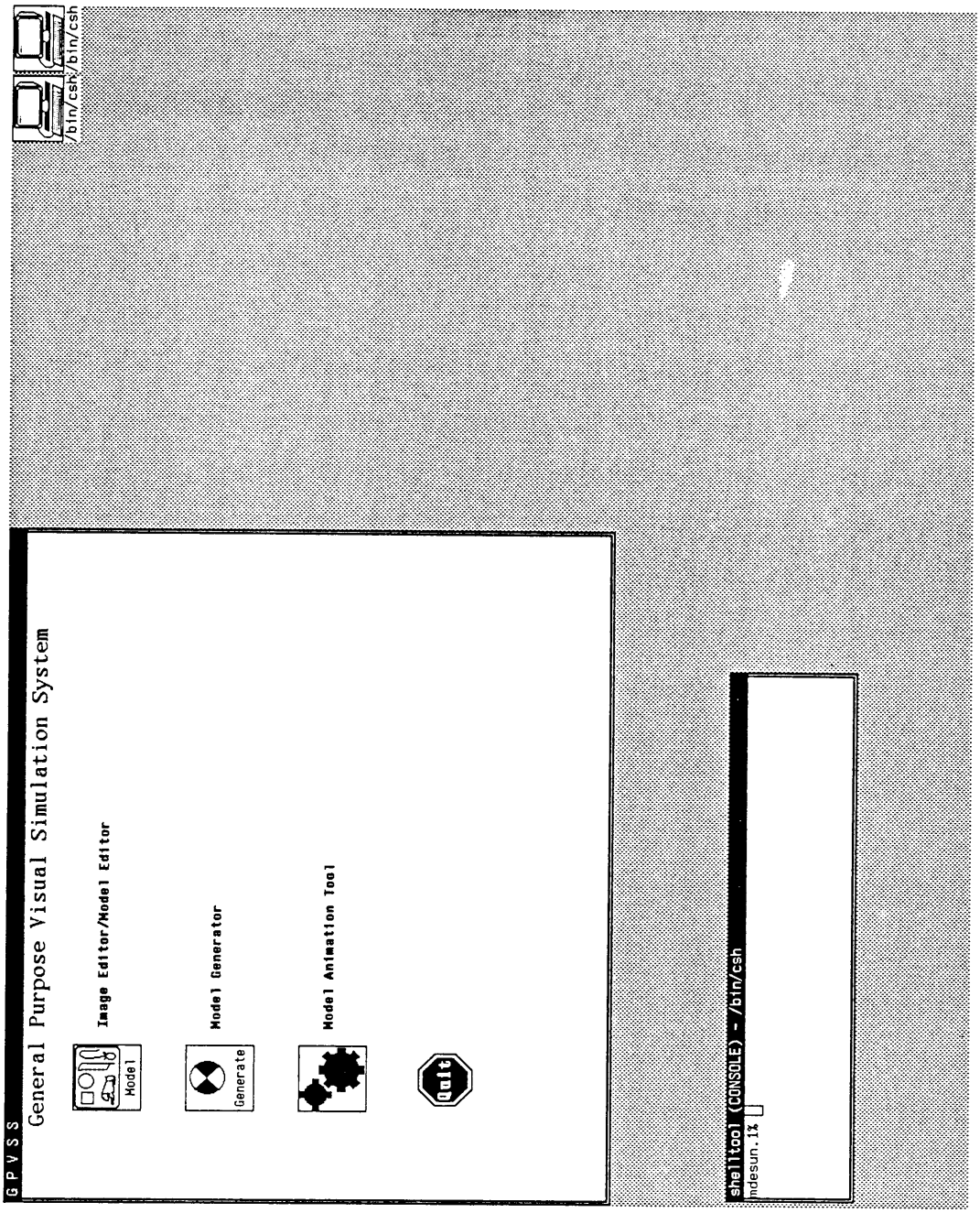

Figure 5.1 GPVSS Main Menu

Icon on the main menu. This brings up a blocking requester asking the user to input the name of the system model (see Figure 5.2). In this case, type "CPU3_System" and then select the "Construct Model" button. This starts the Image Editor/Model Editor as a separate process (see Figure 5.3).

Resize the window as desired (see Figure 5.4). If help with the SunView interface is needed, see the SunView Programmers Guide [Sun 1988a].

The background is now drawn using the pen, line, rectangle, and select modes as described in Chapter 3. The quality of the background is limited only by the patience and artistic capability of the modeler. Be sure to read carefully Section 3.3.1.4 as it describes uses of the select mode that come in extremely handy when creating/erasing images. Once the background is completed, it should look something like Figure 5.5.

Since we now have a completed background that we wish to associate with this model, select the "Save" button on the image editor panel. This brings up a blocking requester for the filename (see Figure 5.6). Enter a path name at the "File:" item and Select the "Save/Attach Background" button. This saves the background in the specified file and place an entry in the "model_dat" relation in the INGRES database (see Table 3.1 in Section 3.3.1.5) associating this file with the background image of

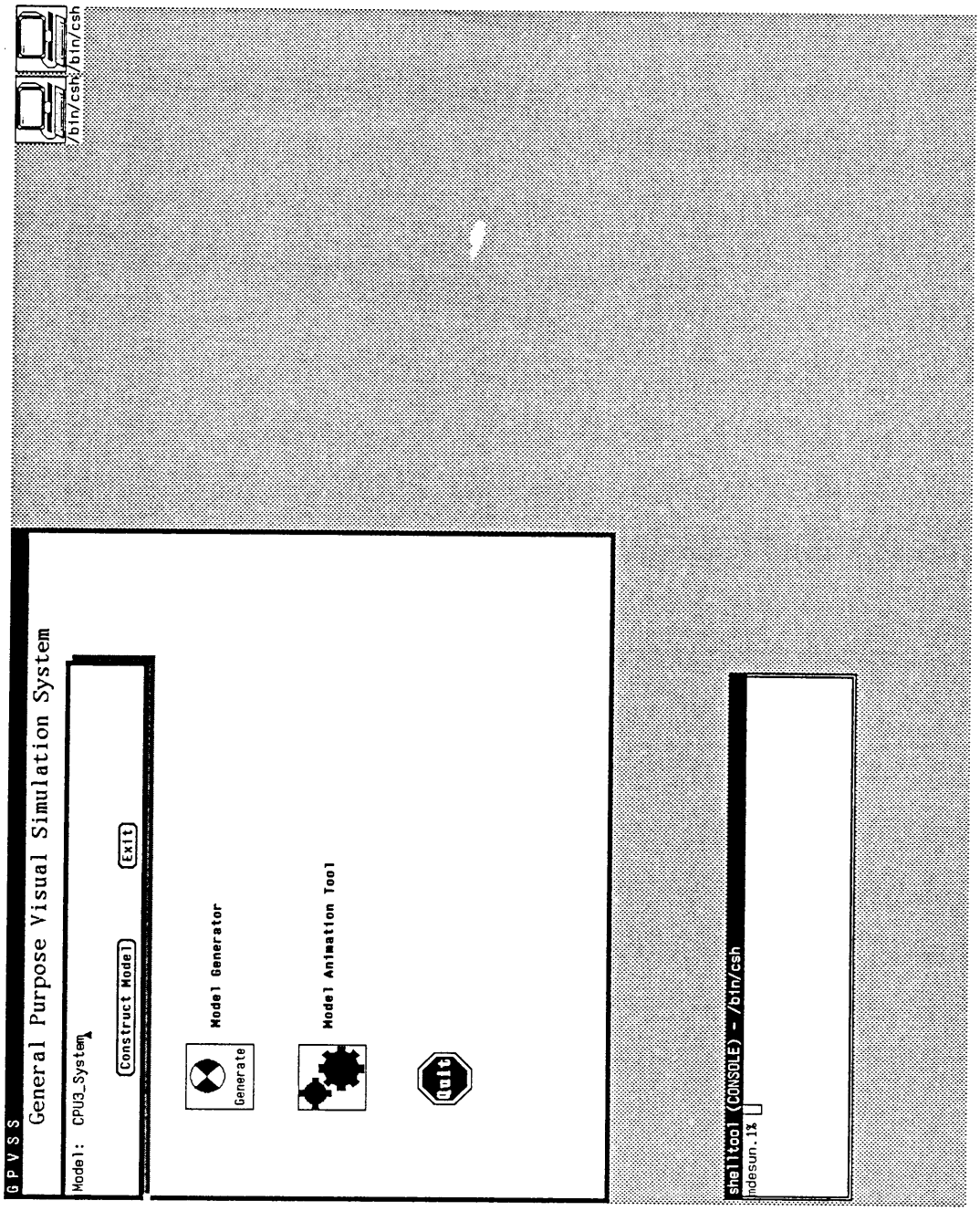

Figure 5.2 System Model Name Requester

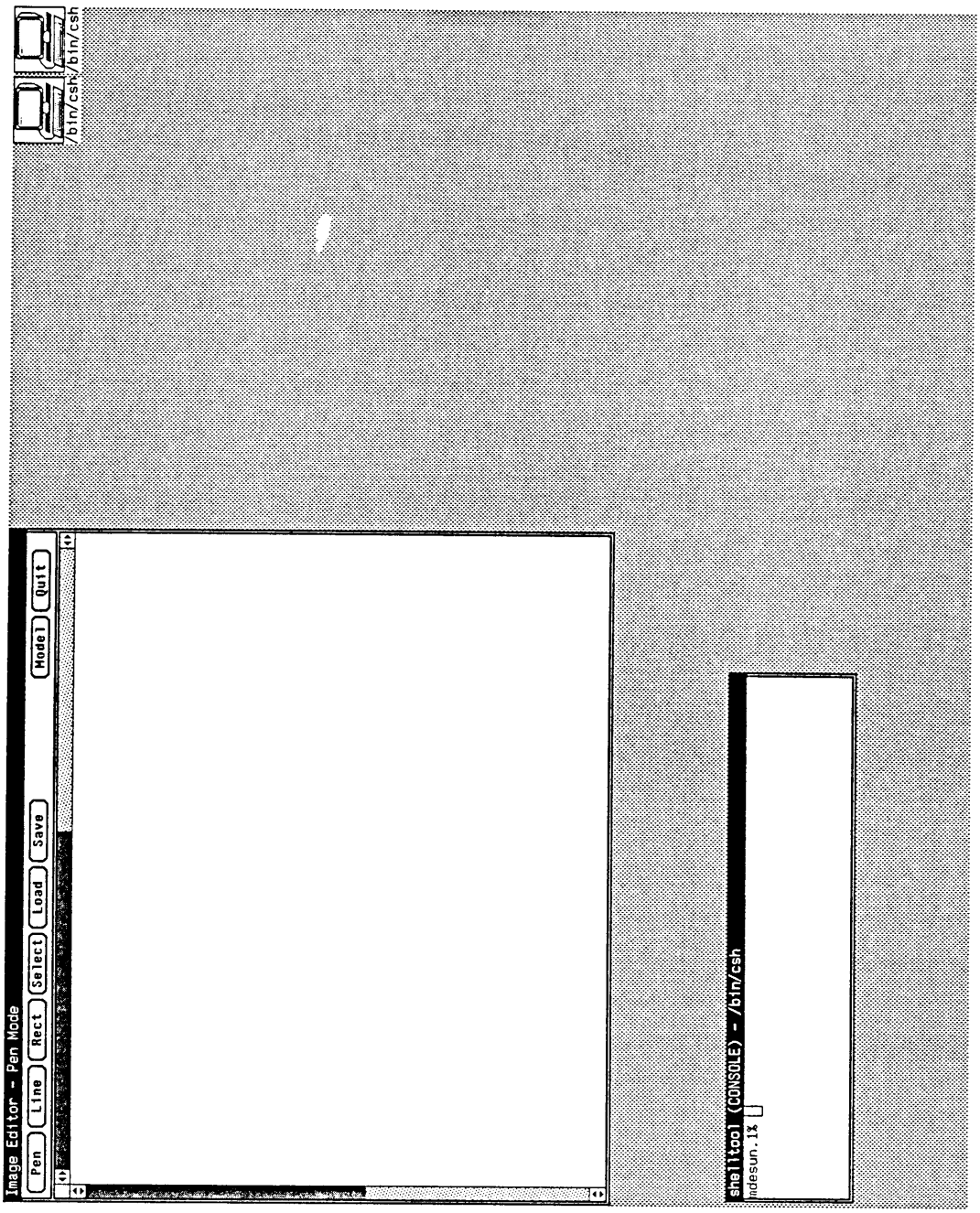

Figure 5.3 Image Editor/Model Editor

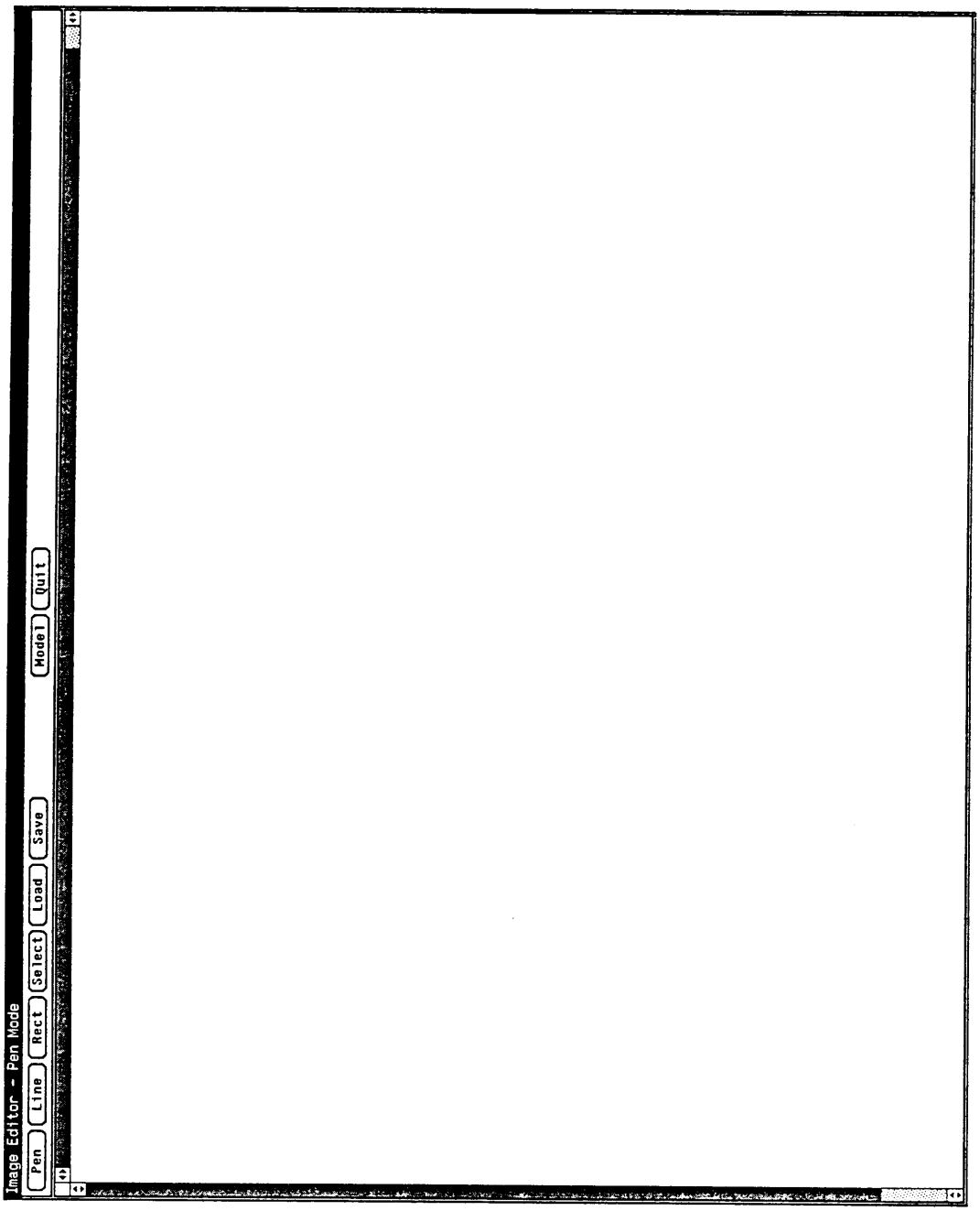

Figure 5.4 Re-sized Image Editor

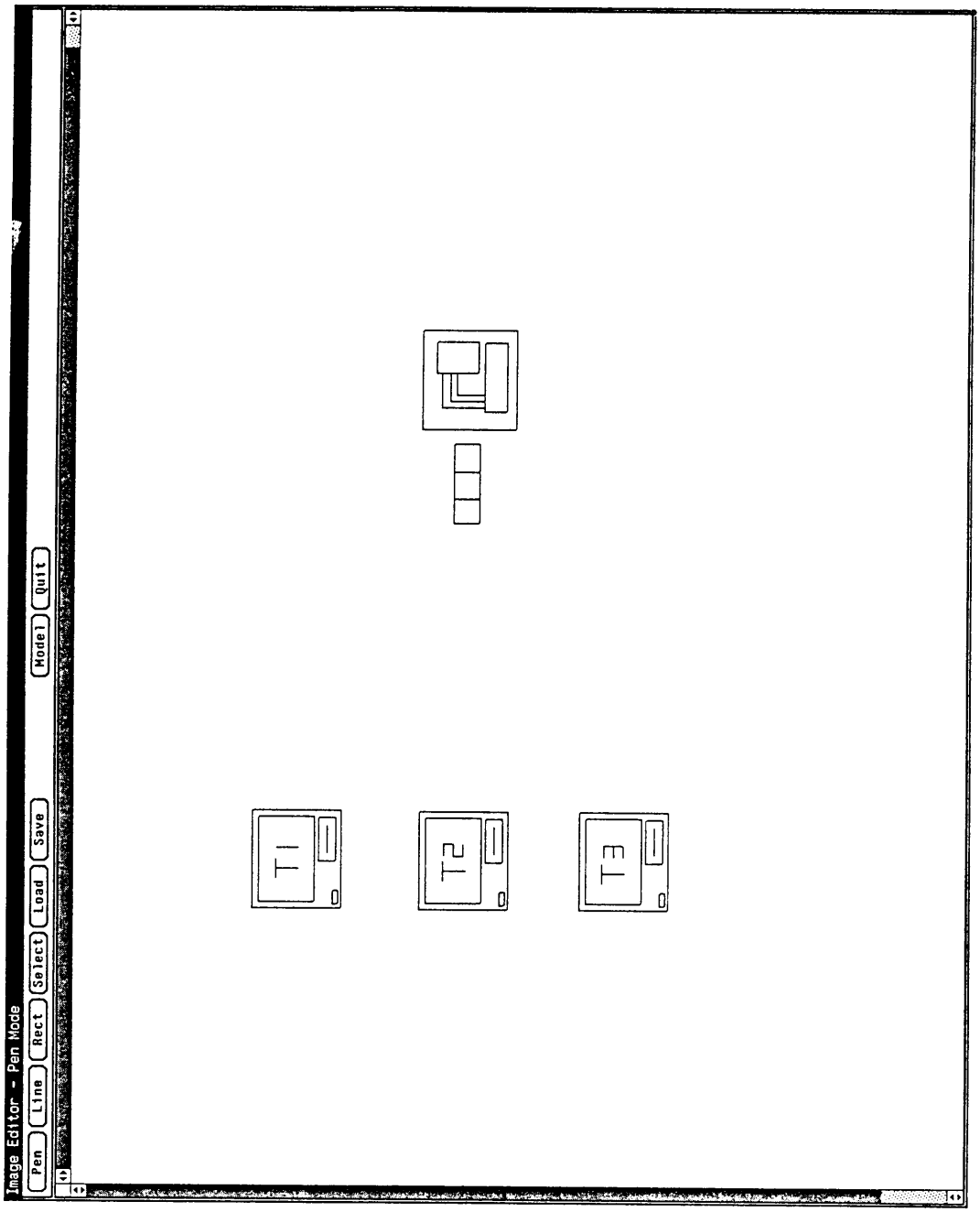

Figure 5.5 Completed Background Image

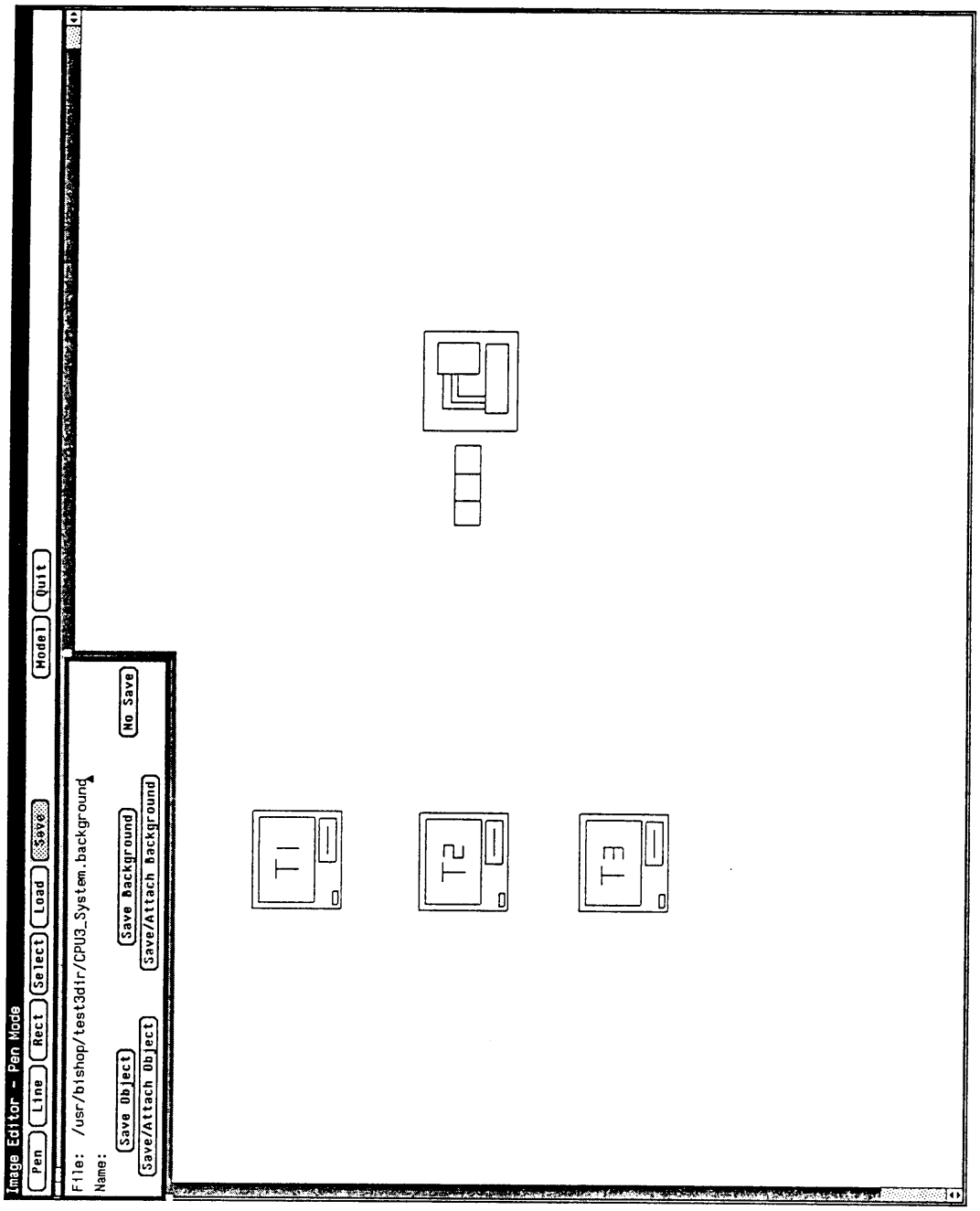

Figure 5.6 Saving the Background Image

this model.

Now design the DO images. To clear the screen, select the "Quit" button and re-enter the image editor. Once the DO images are completed they should look like something resembling the images in Figure 5.7. The images must now be saved as objects. Select the image of a DO originating from Terminal 1 using procedures discussed in section 3.3.1.4 (see Figure 5.8). Select the "Save" button again and fill in the appropriate file name and object name (see Figure 5.9). Select the "Save/Attach Object" button to store the DO image in the specified file and to place an entry in the database (see Table 3.1 in Section 3.3.1.5) so that the DO image may be referenced by name. Repeat this process for the DOs originating from Terminal 2 and Terminal 3 using "job2" and "job3" for the DO names.

5.4 Defining the Model

Once these tasks have been accomplished, resize the window and enter the model editor. To enter the model editor, first erase the DO images and re-load the background image. Select "Load" button from the menu, then select the "Load Attached Background" button (see Figure 5.10). This results in the restoration of the previously saved background image. Note that the editor is still in select mode, but any previously selected image has been destroyed by the image loading process (see Figure 5.11). Now select

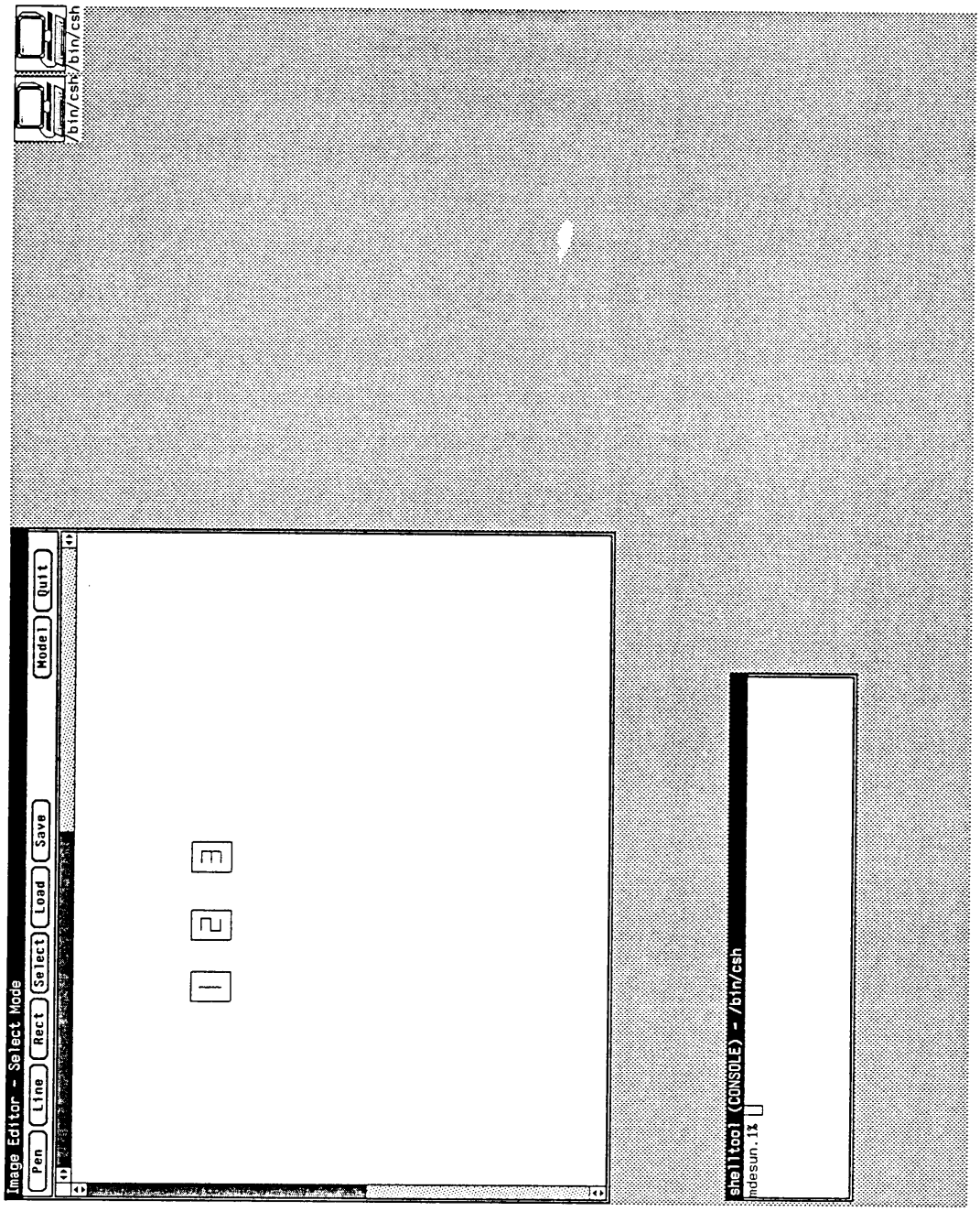

Figure 5.7 Dynamic Object Images

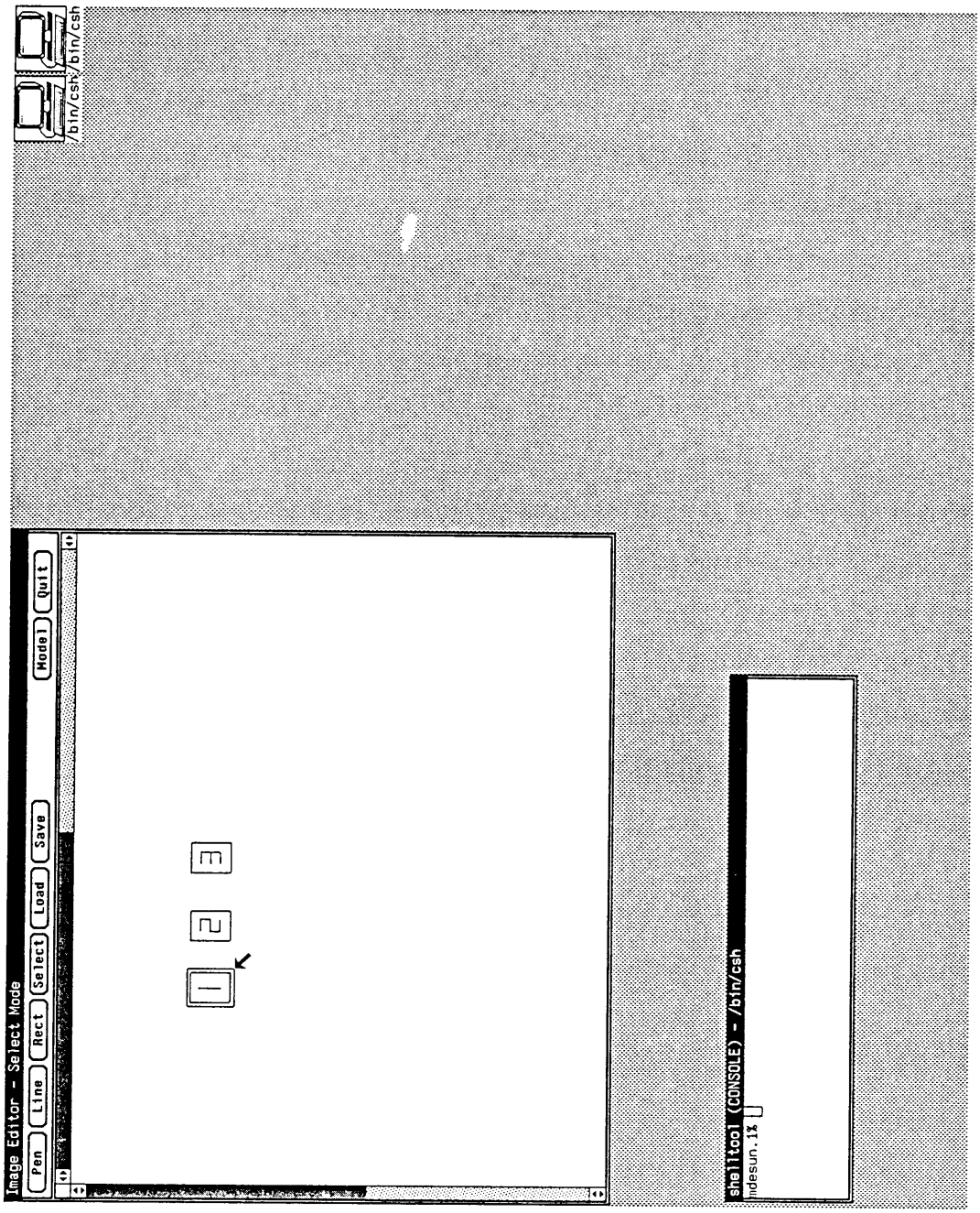

Figure 5.8 Selecting the DO Image

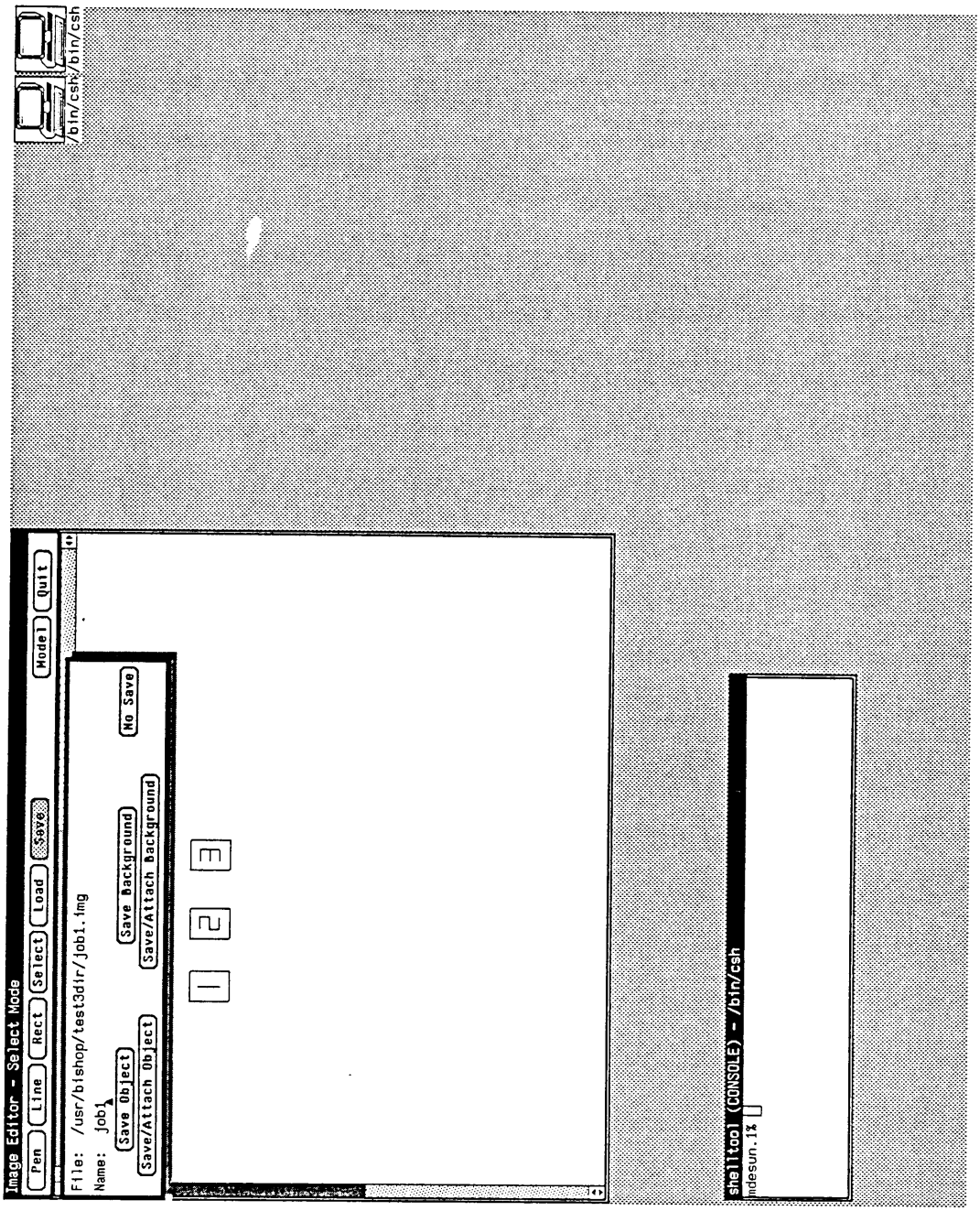

Figure 5.9 Saving a DO Image

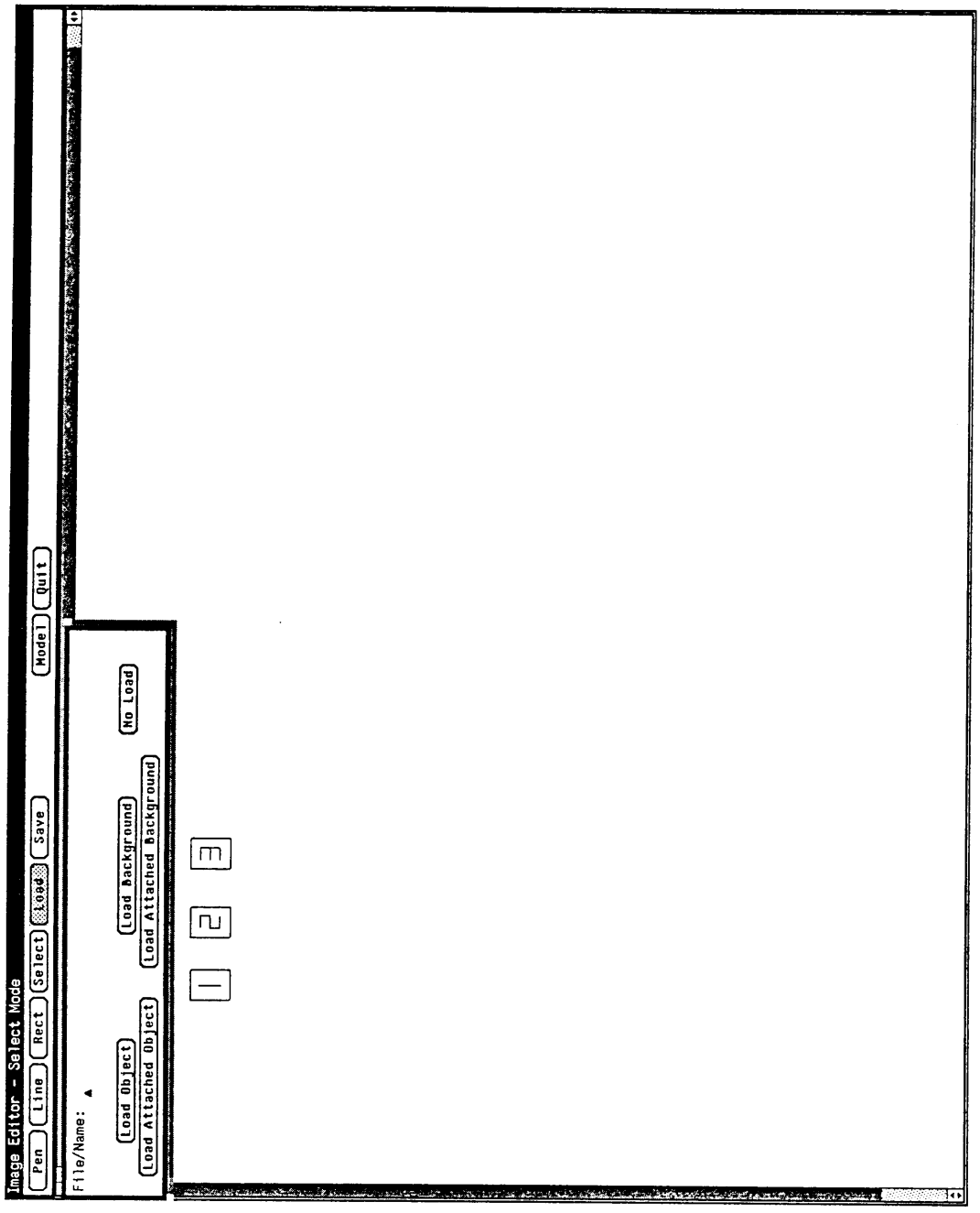

Figure 5.10 Loading the Background Image

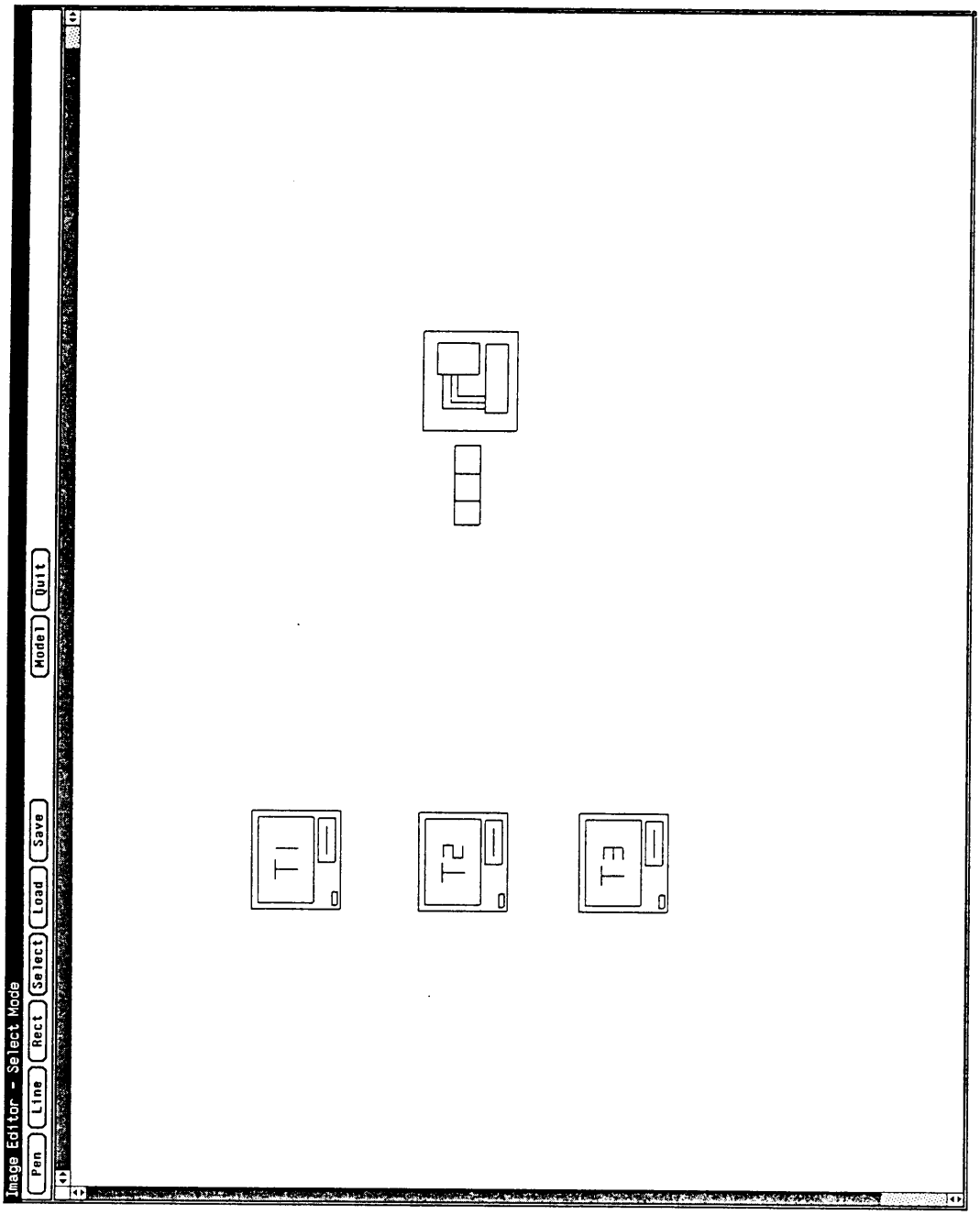

Figure 5.11 Image Editor After Background Load

the "Model" button to enter the model editor (see figure 5.12).

5.4.1 Defining Submodels

It is now time to define all submodels in the model. To define the "Terminal1" submodel select the "SubMod" button on the model editor panel. This brings up a blocking requester for the input of the submodel name (see Figure 5.13). Enter "Terminal1" at the prompt and select "Define Sub-Model". The editor is now ready to draw a rectangle that describes the boundaries of the "Terminal1" submodel for logic specification and path definition purposes. This process is identical to drawing a rectangle (see Section 3.3.1.3).

It should also be noted that if the results of the last submodel definition were not as desired, it is possible to delete the last submodel defined by depressing the middle mouse button. This feature becomes inactive if another editor mode is entered.

Repeat the above process for the "Terminal2", "Terminal3", "CPU_QUEUE", and "CPU" submodels. The model should then look like Figure 5.14.

5.4.2 Defining Paths

The paths connecting the submodels should now be defined. Paths are uni-directional. If a DO moves both

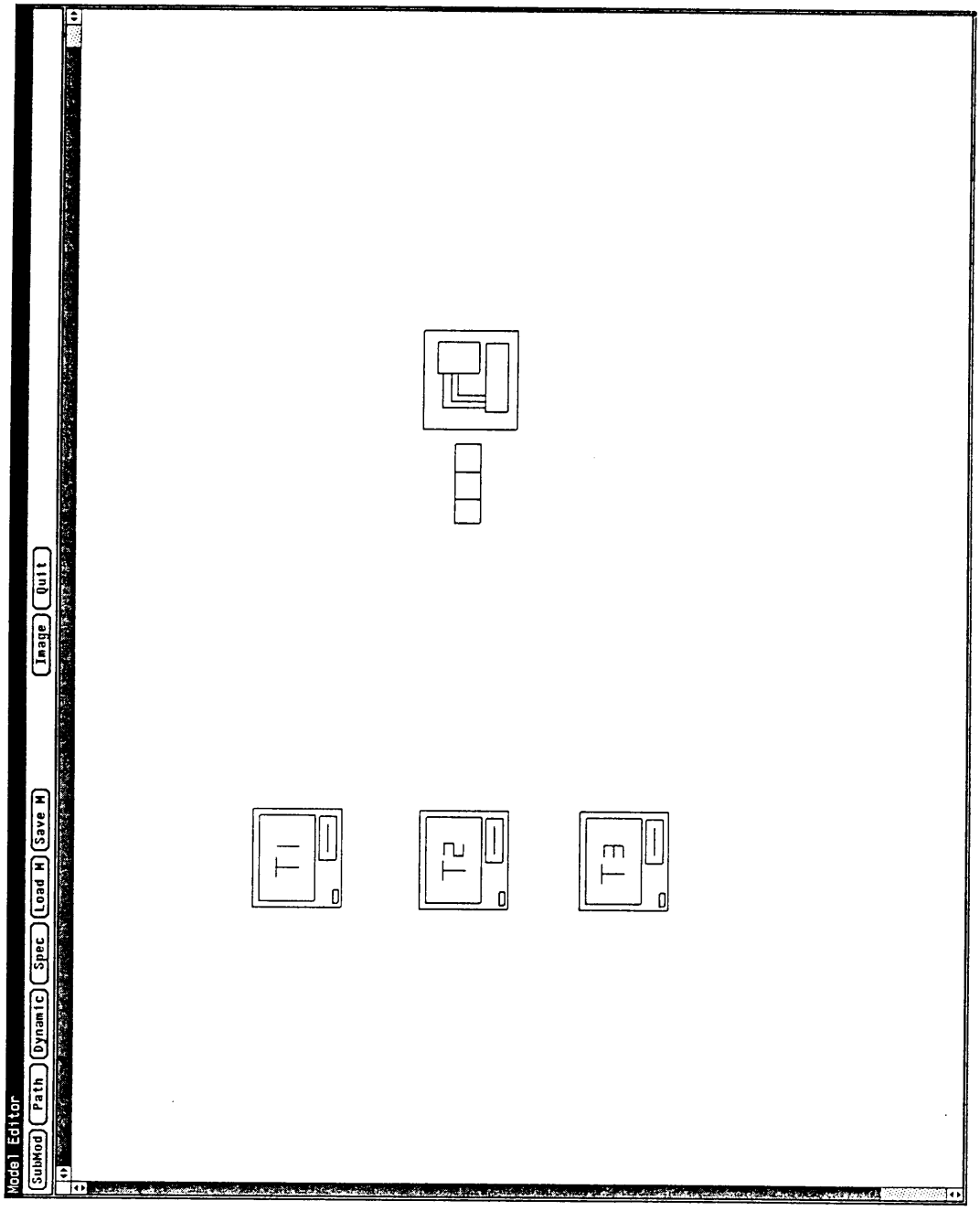

Figure 5.12 The Model Editor

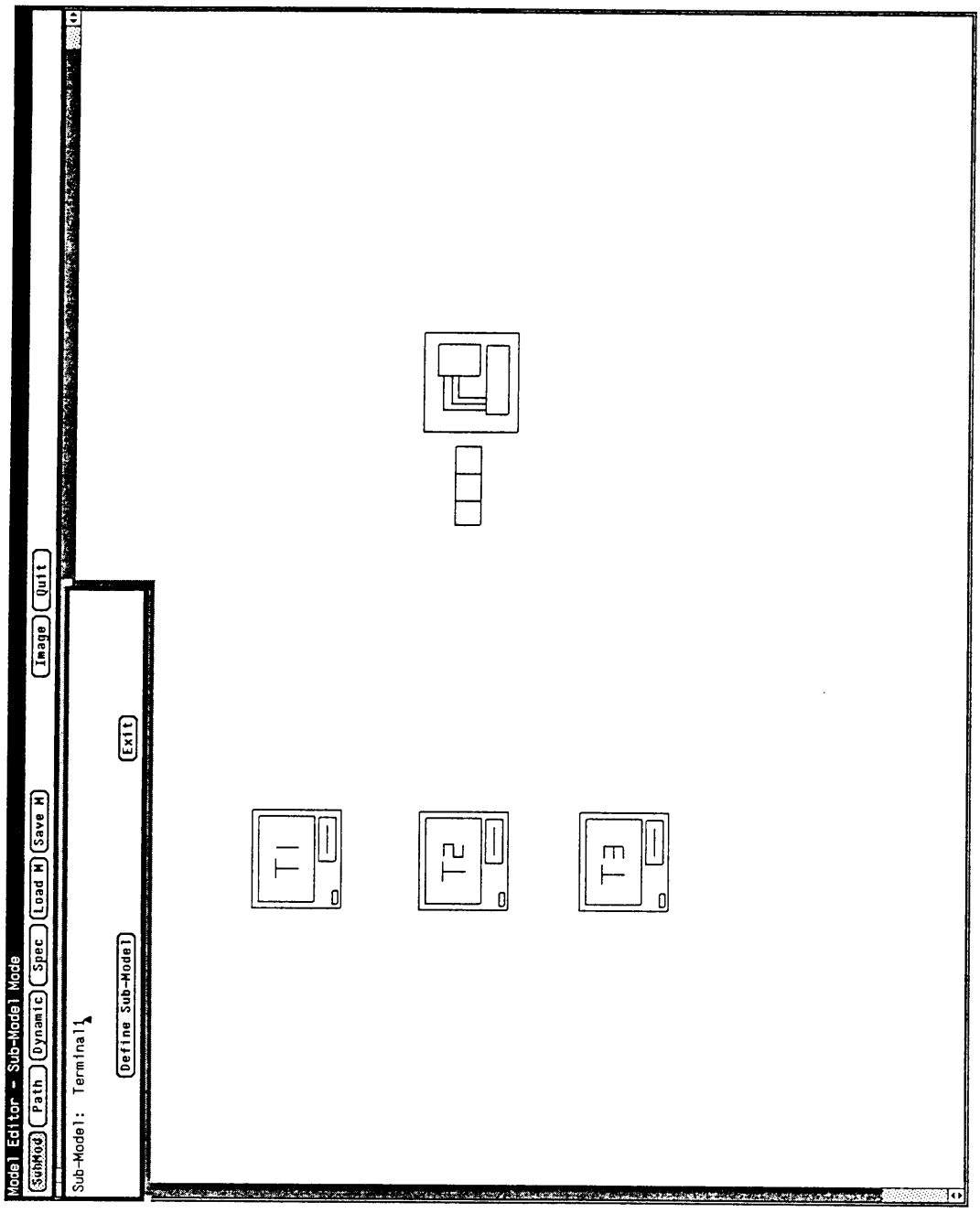

Figure 5.13 Submodel Definition Name Requester

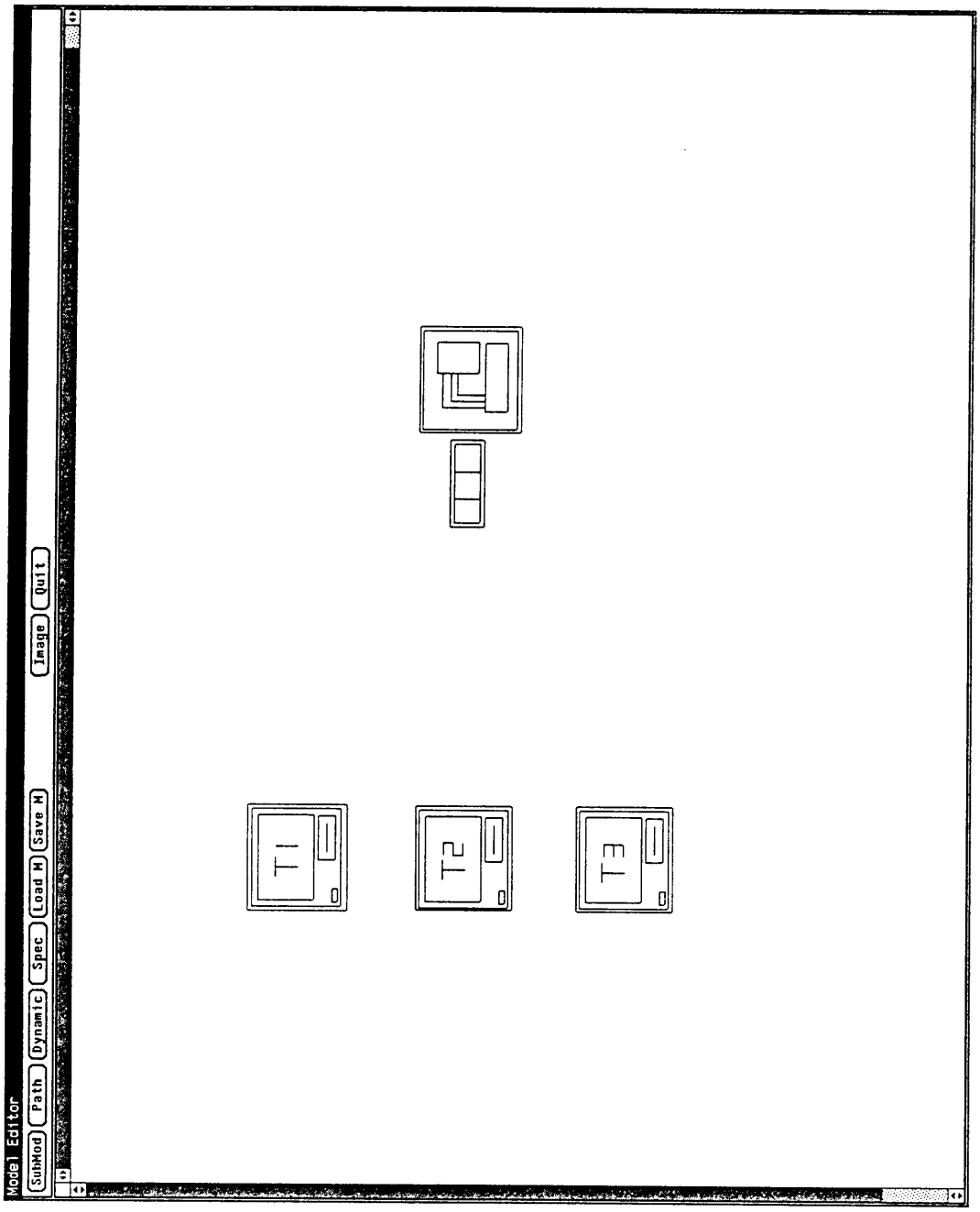

Figure 5.14 Completed Submodel Definition

ways between two submodels, two paths must be specified. Not all submodels must be interconnected, but there must be a path between any two submodels in the same direction a DO could possibly travel. If a DO tries to move to and from submodels that are not connected by a path, an error occurs.

To define a path, first select the "Path" button on the model editor panel. This brings up a blocking requester for the path name (see Figure 5.15). Currently this information is not used. Therefore anything maybe entered here. Select the "Define Path" button to start the path definition.

Locate the mouse cursor at that point the path is to begin. This point must begin within a submodel boundary (see Figure 5.16). Press the left mouse button to select the first point on the path. If the mouse cursor is outside a submodel, this has no effect. Subsequent intermediate points may be specified irregardless of submodel boundaries by depressing the left mouse button (see Section 3.3.2.2). The path termination point must be in a submodel and is selected with the middle mouse button (see Figure 5.17). Repeat the above process until all paths are defined. The model should then resemble Figure 5.18.

5.4.3 Saving a Model

Since the model is now completely defined, it is a good idea to save it. Select the "Save M" button on the model editor panel. Enter the desired file name and Select

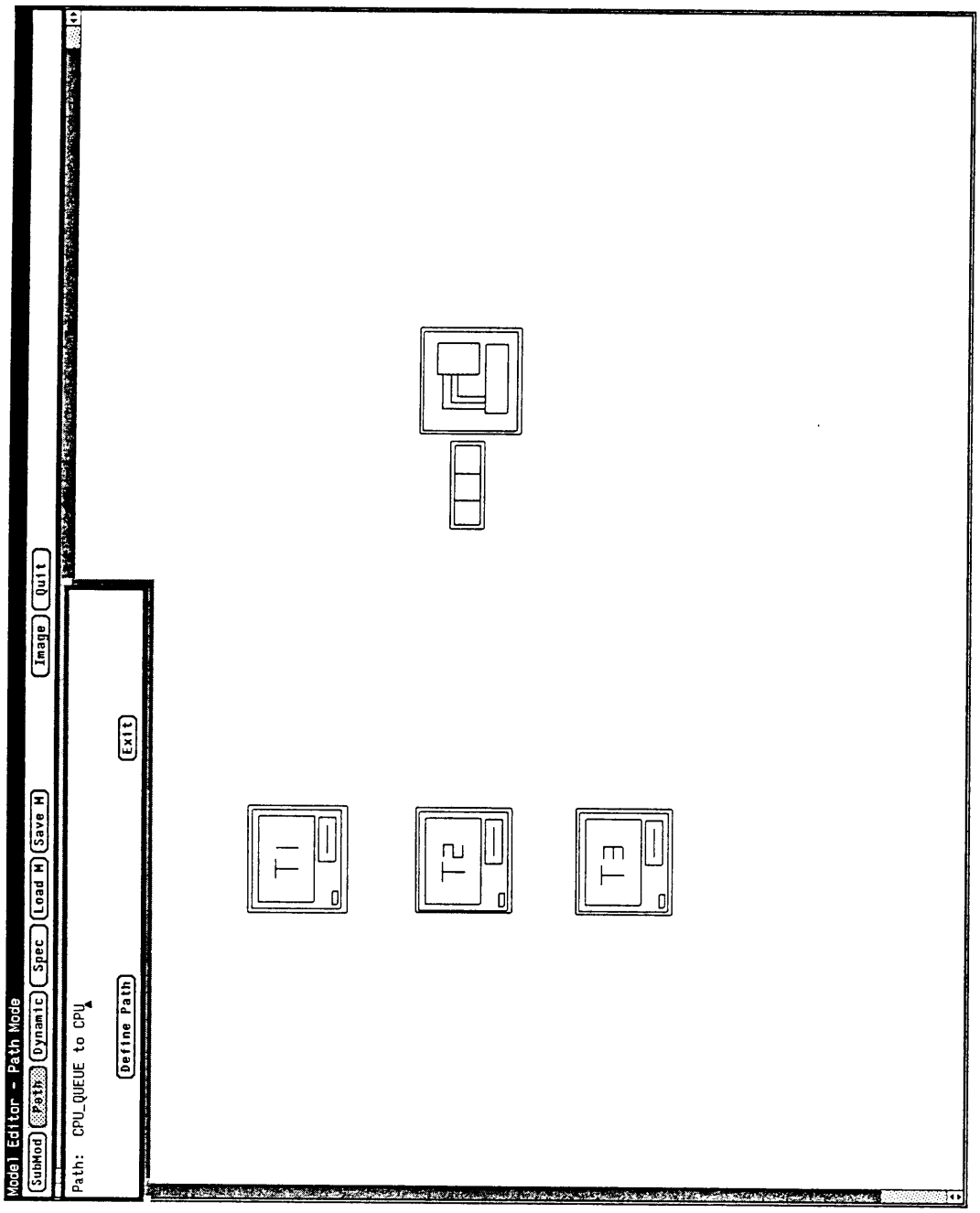

Figure 5.15 Path Name Requester

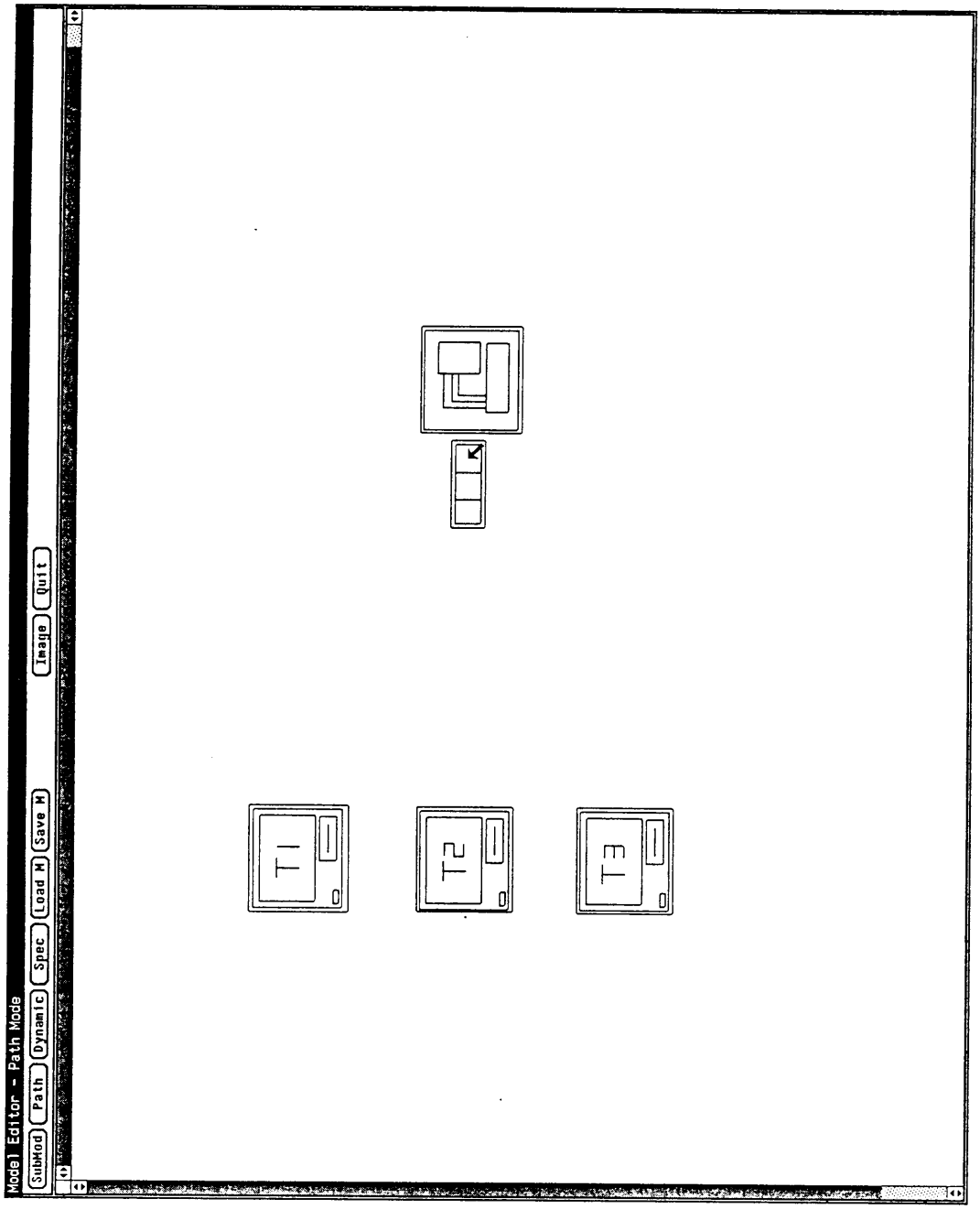

Figure 5.16 Path Starting Point

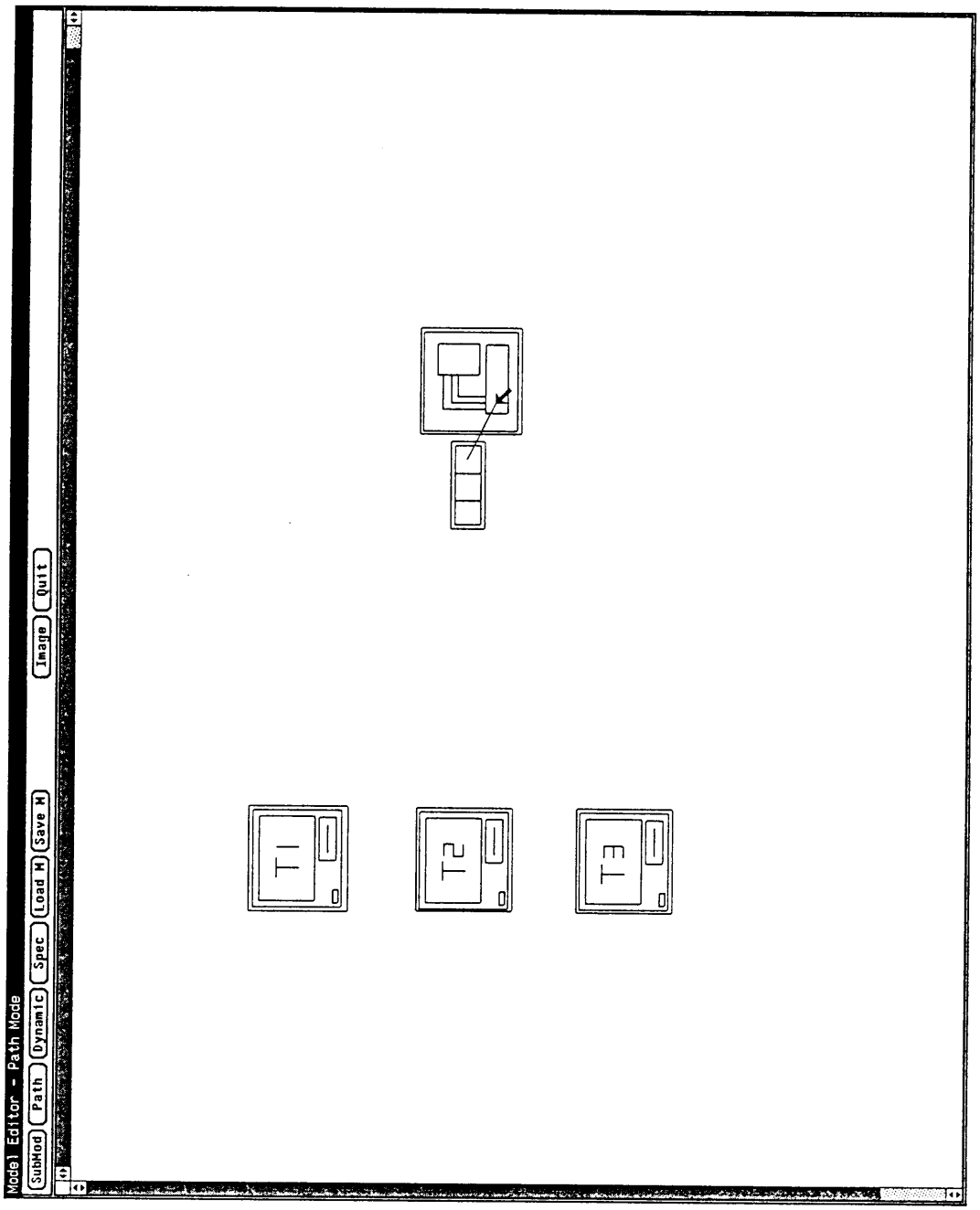

Figure 5.17 Path Termination

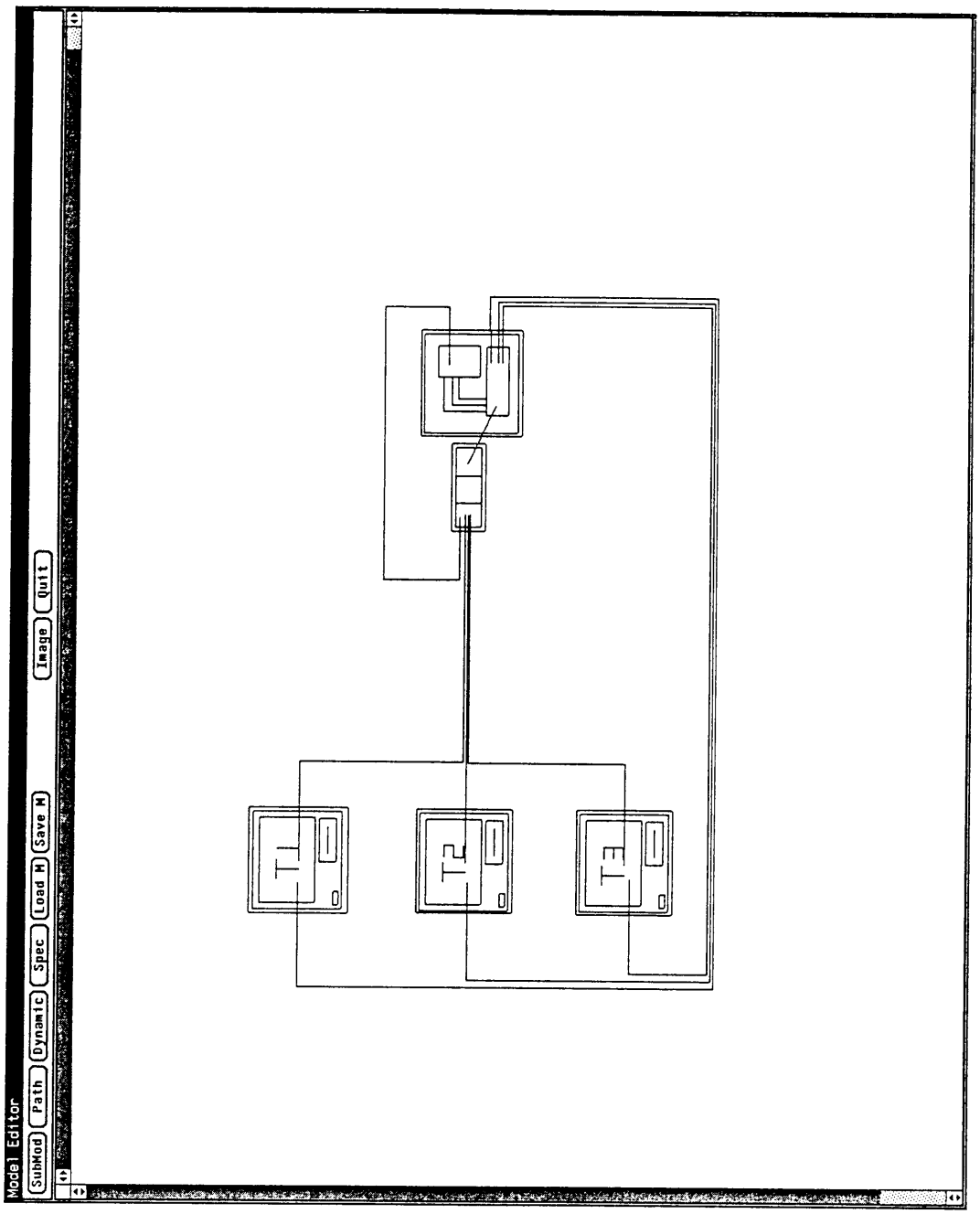

Figure 5.18 Completed Path Definitions

"Save/Attach Model" (see Figure 5.19). This stores the model and make an appropriate entry in the "model_dat" table of the database (see Table 3.1 in Section 3.3.1.5). When future model saves are desired, the file name defaults to this value if nothing is typed at the prompt, including blanks. Saved models may be loaded by selecting the "Load" button (see Section 3.3.1.5).

5.5 Submodel Specification

The next step is submodel specification. Select the "Spec" button on the model editor panel to enter specification mode. A pop-up menu is activated by depressing the right mouse button while within a submodel boundary (see Figure 5.20). Depressing the right mouse button outside of a submodel boundary has no effect.

5.5.1 Submodel Logic

Select the "Sub-Model Logic" menu item while within "Terminal1" to specify the internal logic of this submodel. This brings up a blocking requester for the path name of the directory all created files are stored in (see Figure 5.21). This path should only be entered once for all submodels. Subsequent specifications should merely select the "Specification Path" button without typing anything at the prompt, including blanks. This is similar to the process described previously for saving a model definition. In any

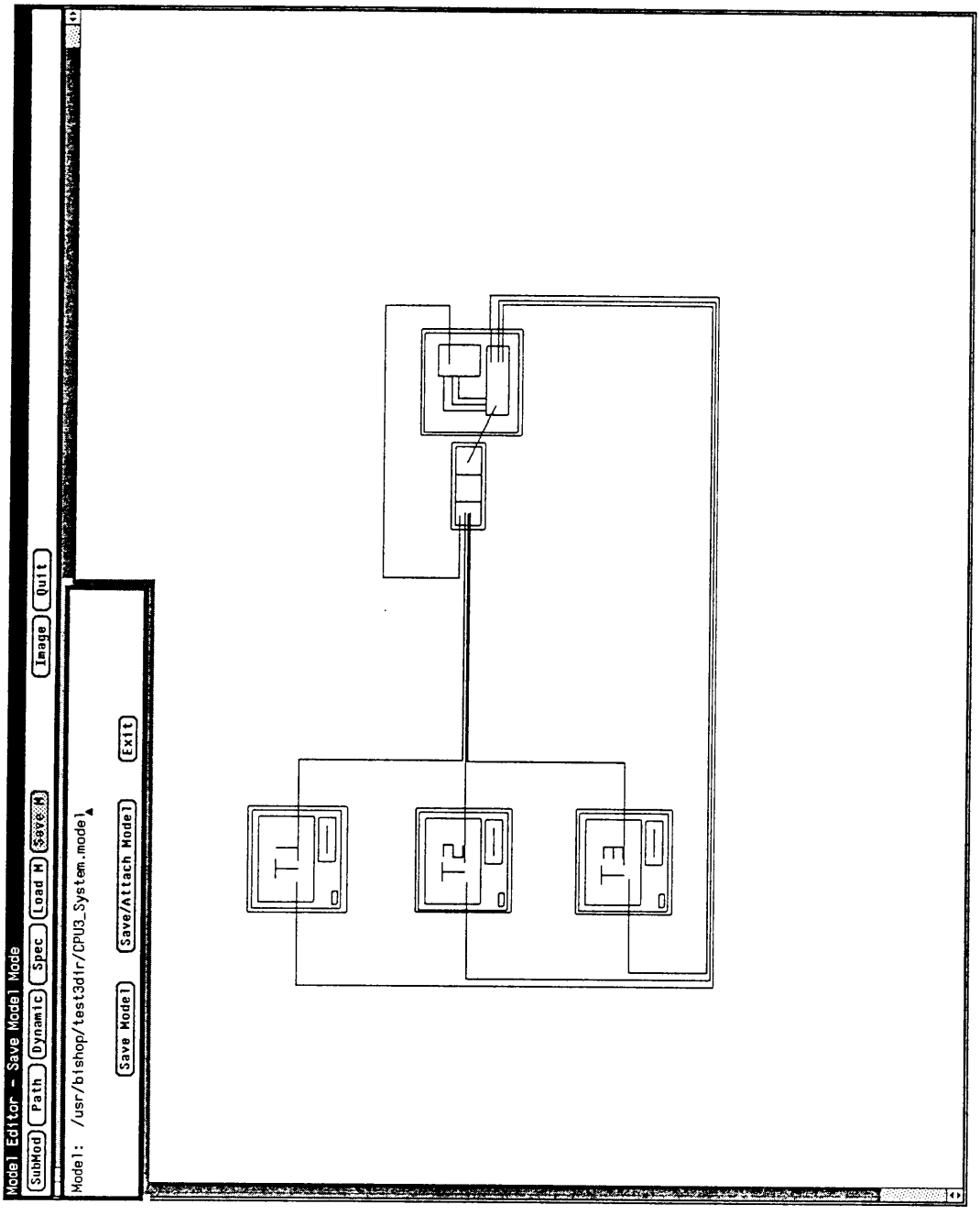

Figure 5.19 Save Model Requester

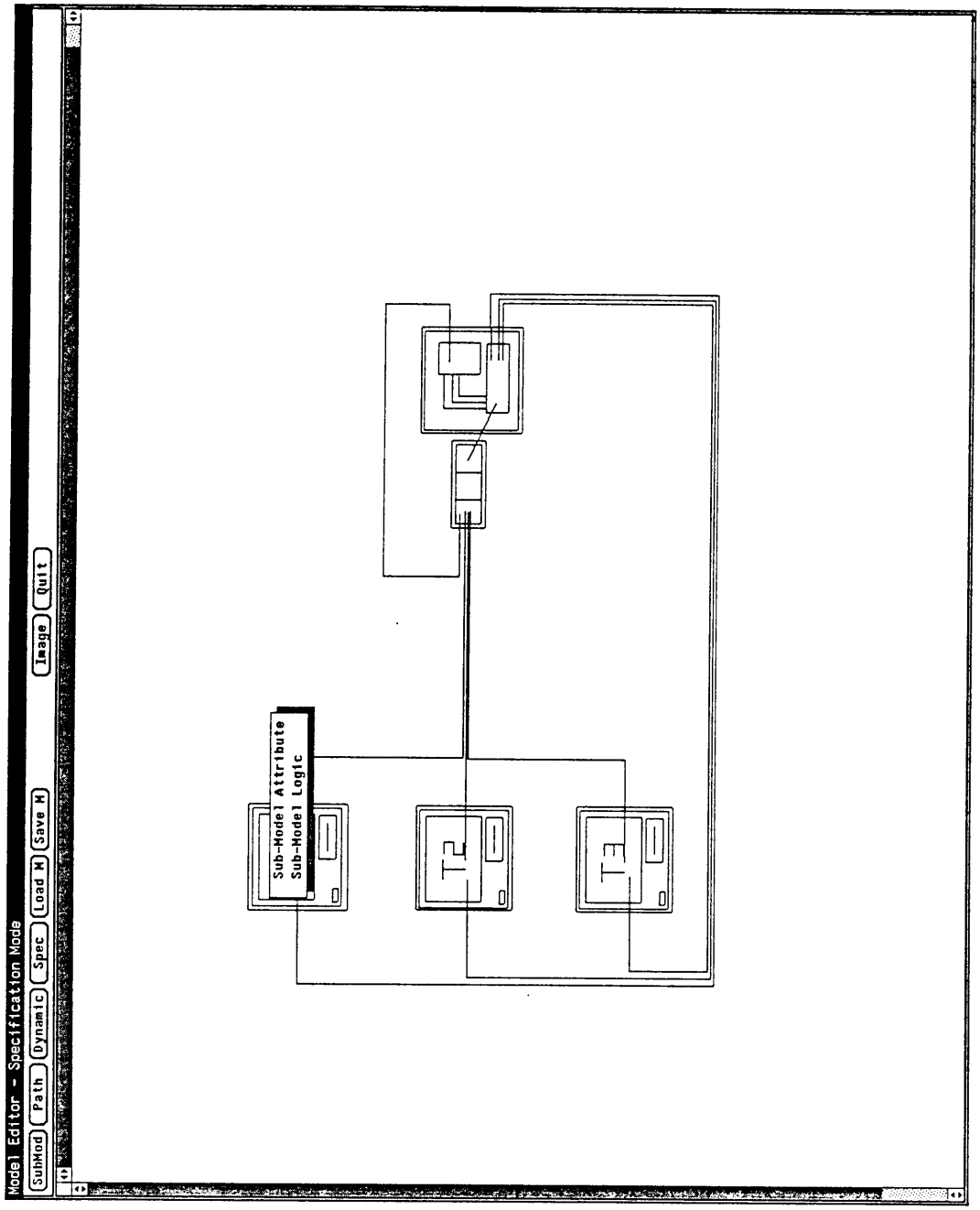

Figure 5.20 Specification Menu

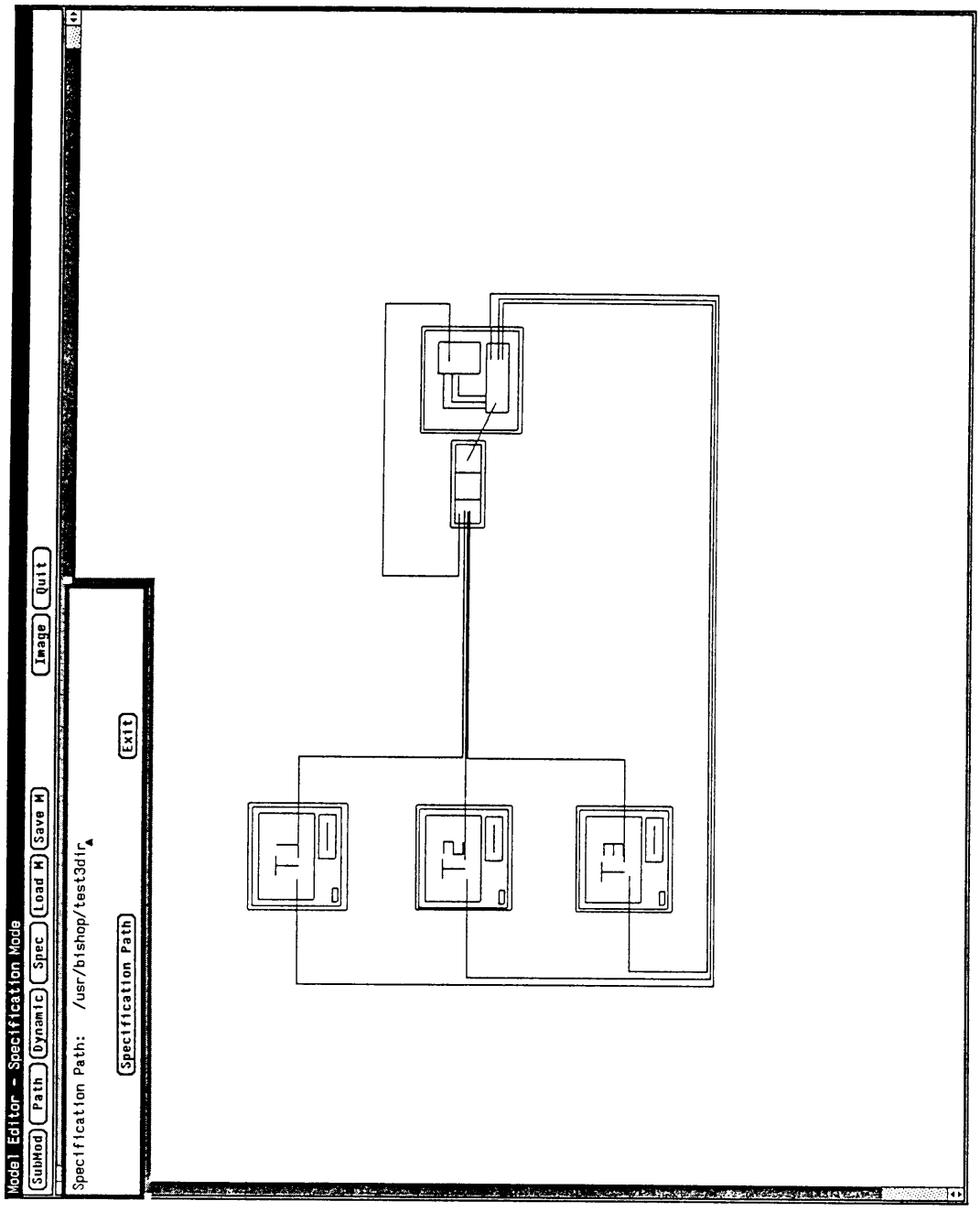

Figure 5.21 Specification Path Requester

event, selection of the "Specification Path" brings up the submodel specification tool.

The specification of the "Terminal1" submodel is shown in Figure 5.22. The Entrance Condition (EC) for "Terminal1" is true. This means DOs can always get into this submodel. Once the DO has entered the submodel, its origin is set to 1, and rservice_time (remaining service time) is sampled from an exponential distribution with mean 500.00. "Origin" and "rservice_time" DO attributes are not pre-defined and must be defined and specified along with the DOs. For now it is best to keep a list of attributes used and define them all at once later. "Seed" is a pre-defined integer seed for a random number generator. The two SET_DO statements constitute the Logical Operations (LO) if the EC holds true.

Since the Activity Start Condition (ASC) is true, the DO may engage in the activity right away. The activity duration is sampled from an exponential distribution with mean 250.00. This is the user "think" time. The DO remains engaged in the Activity until this time has passed.

After the activity is completed, the "SYS_ENTRY" macro records the DO entering the model.

Since the eXit Condition (XC) is true, the DO leaves the submodel and moves to the CPU_QUEUE submodel. Note the "_MODEL" extension to the CPU_QUEUE submodel name.

Finally, when the specification is complete, select the "Save" button located at the bottom left hand side of the

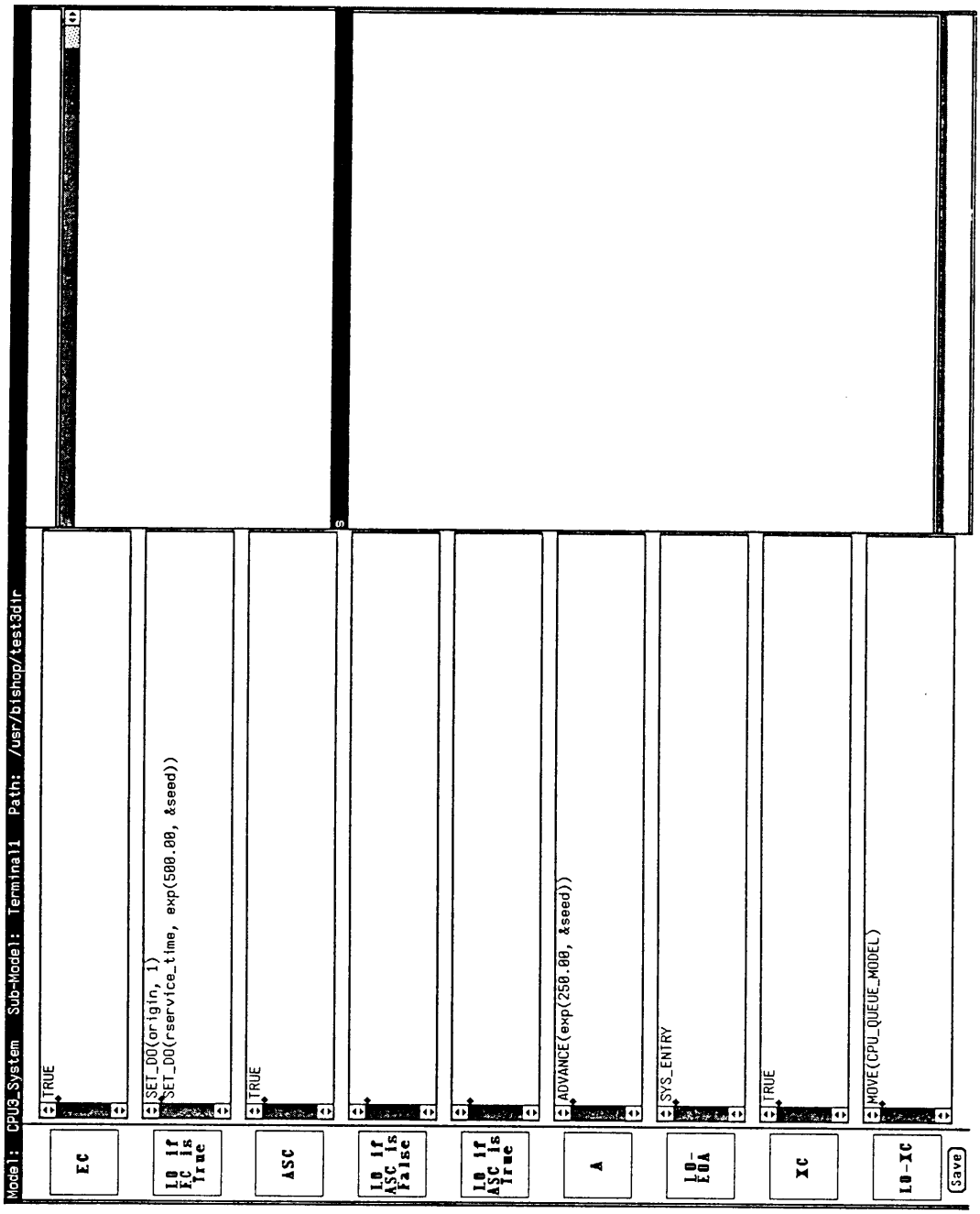

Figure 5.22 Terminal1 Internal Logic Specification

specification tool to exit the tool and save the specification.

Figures 5.23 and 5.24 show similar specifications for the "Terminal2" and "Terminal3" submodels respectively.

The specification of the "CPU_QUEUE" submodel is shown in Figure 5.25. Since the Entrance Condition is TRUE, a DO can always enter the "CPU_QUEUE" submodel. When it does, the q_sm_et (queue submodel entry time) is set to the simulation clock. This attribute is a standard DO attribute (see Table 3.2 in Section 3.3.2.3).

The activity (waiting for the CPU) is started if there is a DO in the CPU, or if there are DOs already waiting in the queue. The "nicpu_sm" and "nicpu_queue_sm" attributes must be defined using the submodel attribute definition tool. As before, make a list of attributes to be specified so that they may be done all at once later.

If there is no need to wait (LO if ASC is FALSE), the EXIT_SUBMODEL macro takes the DO to the CPU_QUEUE XC. Otherwise, the number of DOs waiting in the CPU_QUEUE is incremented by one using the INCREMENT macro.

Activity duration for a DO in the CPU_QUEUE is until the CPU is free and the DO has the earliest q_sm_et for this submodel. The least() function is provided for the evaluation of the q_sm_et. Again, notice the "_MODEL" extension to the submodel name to be checked.

On completion of the activity, the number in the queue

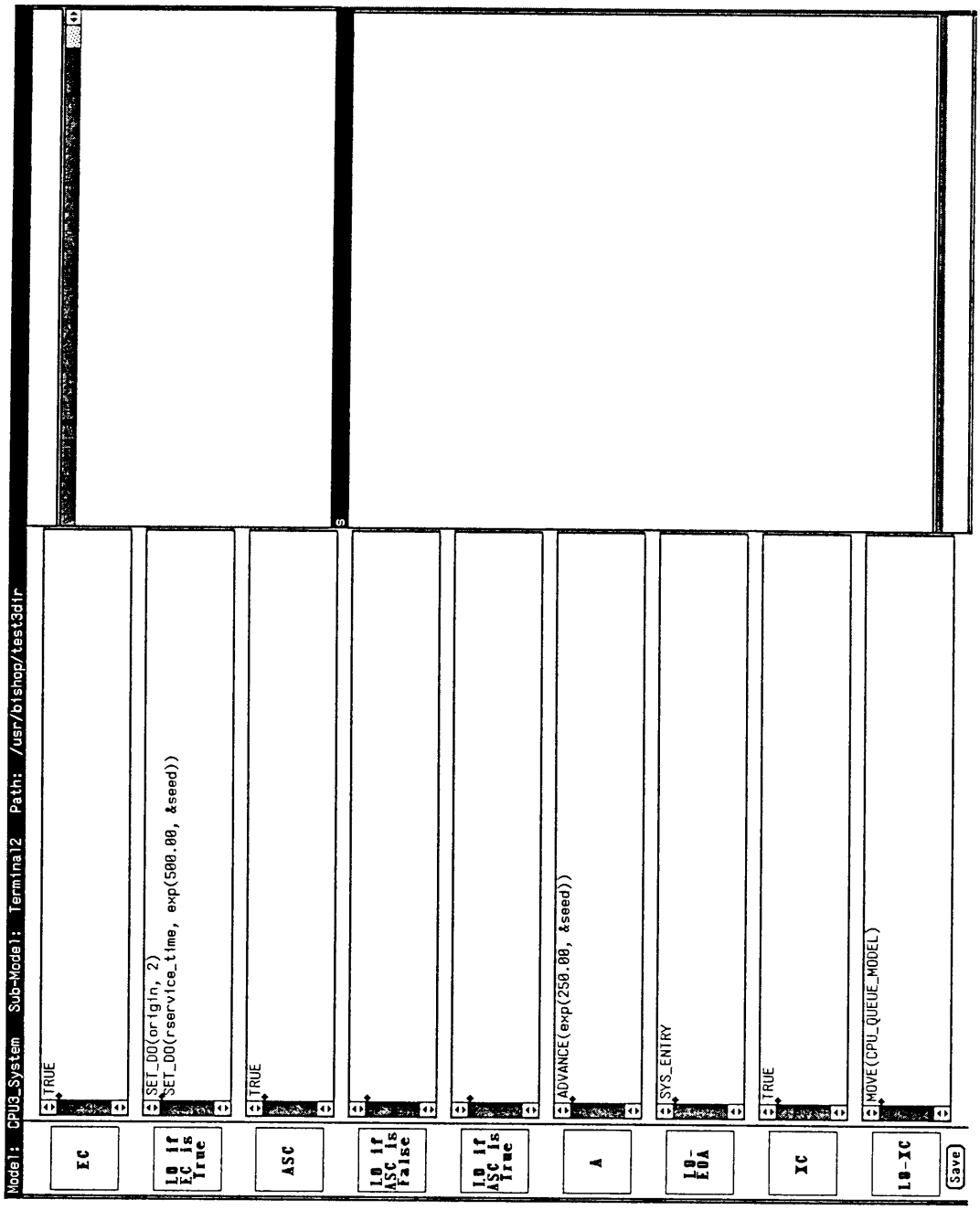

Figure 5.23 Terminal2 Internal Logic Specification

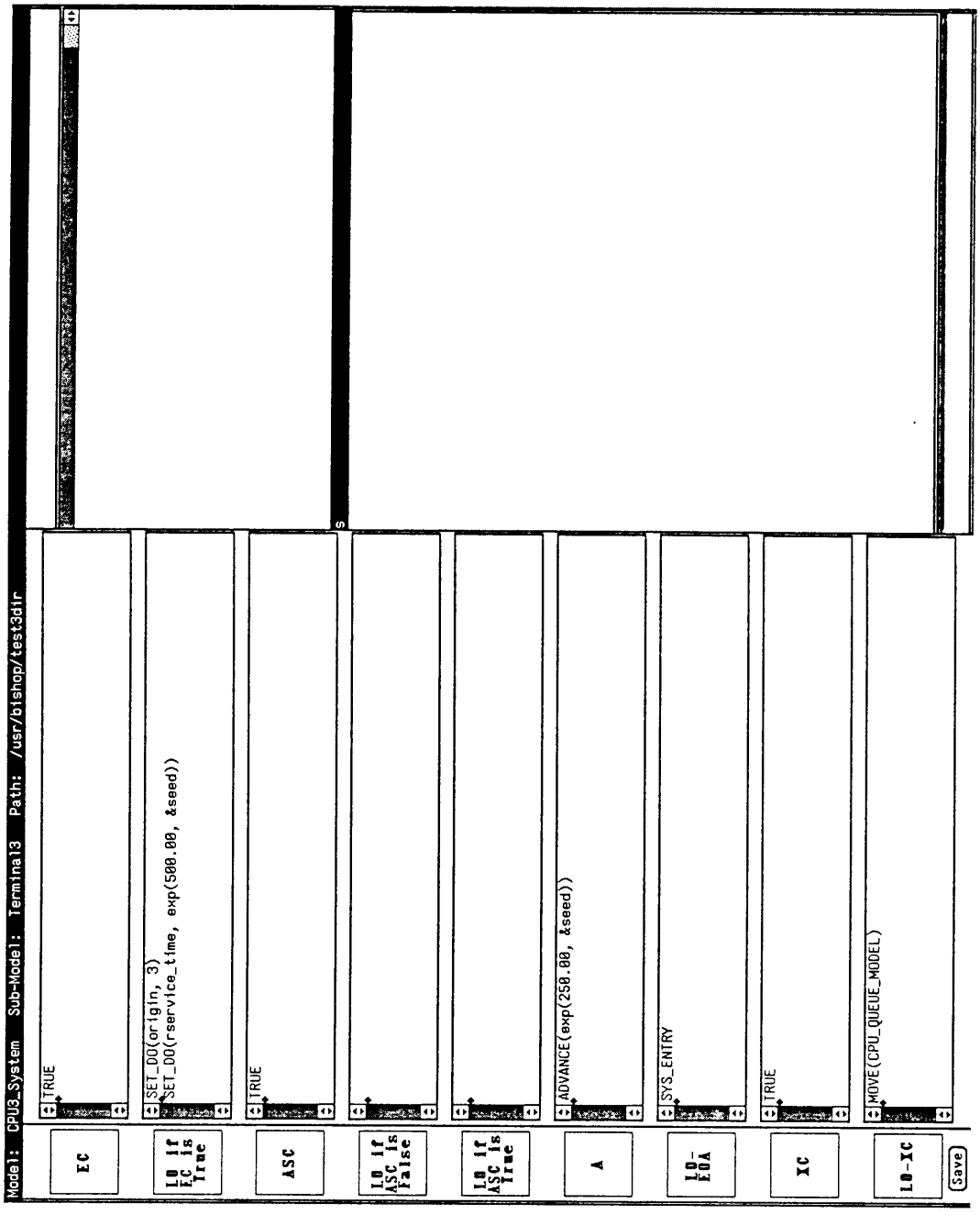

Figure 5.24 Terminal3 Internal Logic Specification

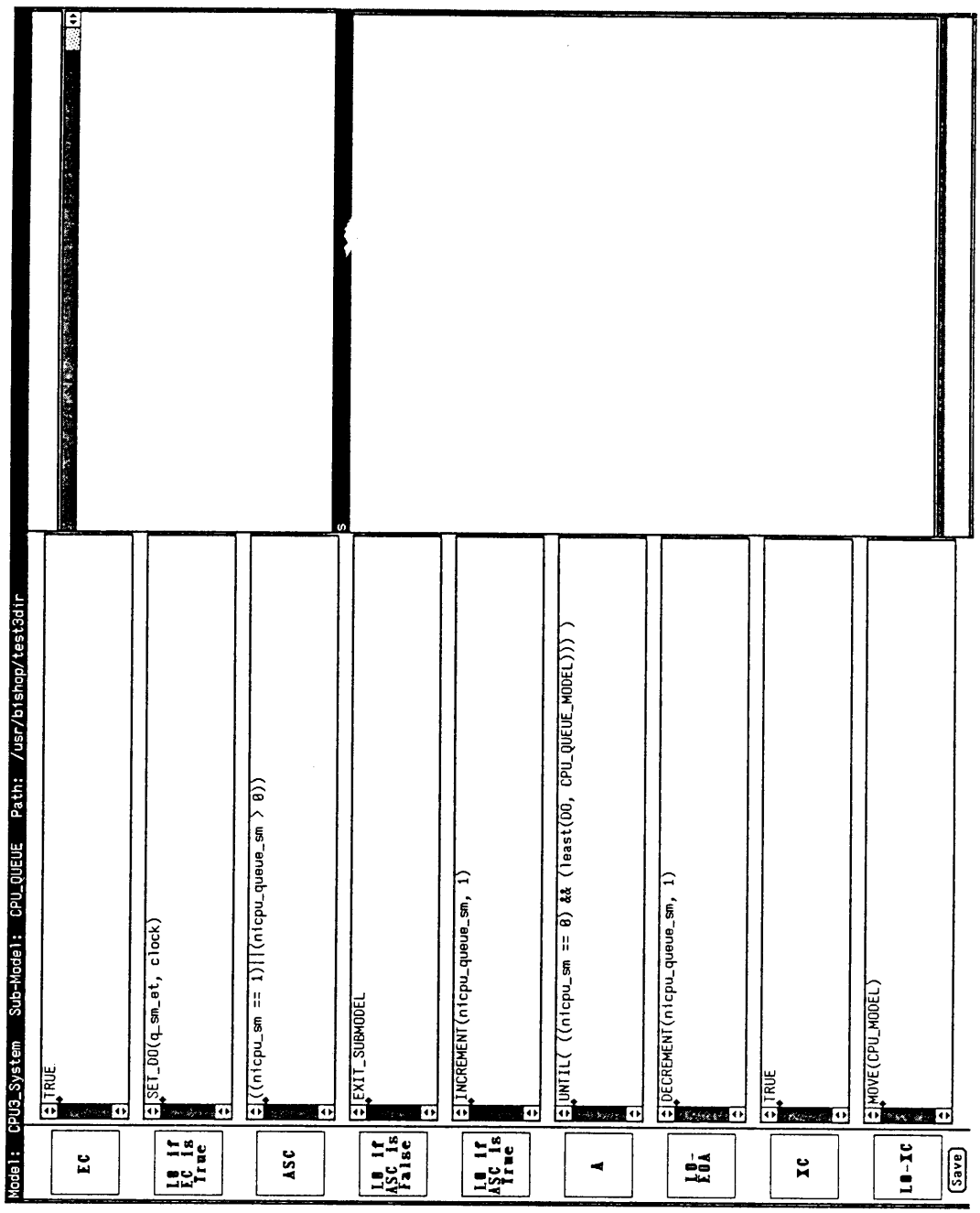

Figure 5.25 CPU_QUEUE Internal Logic Specification

is decremented by one using the DECREMENT macro.

Once jobs get to the XC submodel section, they are moved to the "CPU" using the MOVE macro. Figures 5.26 and 5.27 show the CPU submodel specification. The Entrance Conditions enforce the fact that a DO may only enter this submodel if the CPU is empty. Upon entrance, the number of DOs in the CPU is incremented by 1 using the INCREMENT macro.

If there is one DO in the CPU, the activity may start. The "LO if ASC is TRUE" section introduces the "QUANTUM", "execution_time", and "OVERHEAD" attributes. Make a note of these submodel attributes so that they are specified later. These attributes are used to determine how much processing time a CPU spends on a given DO. If the remaining service time is greater than the maximum quantum size, the CPU processes the DO for the execution time equal to the quantum plus scheduling overhead time. Otherwise, the execution time is set to the remaining service time plus scheduling overhead time. The DO then remains in the Activity until the execution time has elapsed.

Upon completion of the Activity, the remaining service time of the DO is decremented as much as the execution time excluding the overhead time.

Since the eXit Condition is true, all DOs may always exit the CPU submodel.

When leaving, the number of DOs in the CPU is

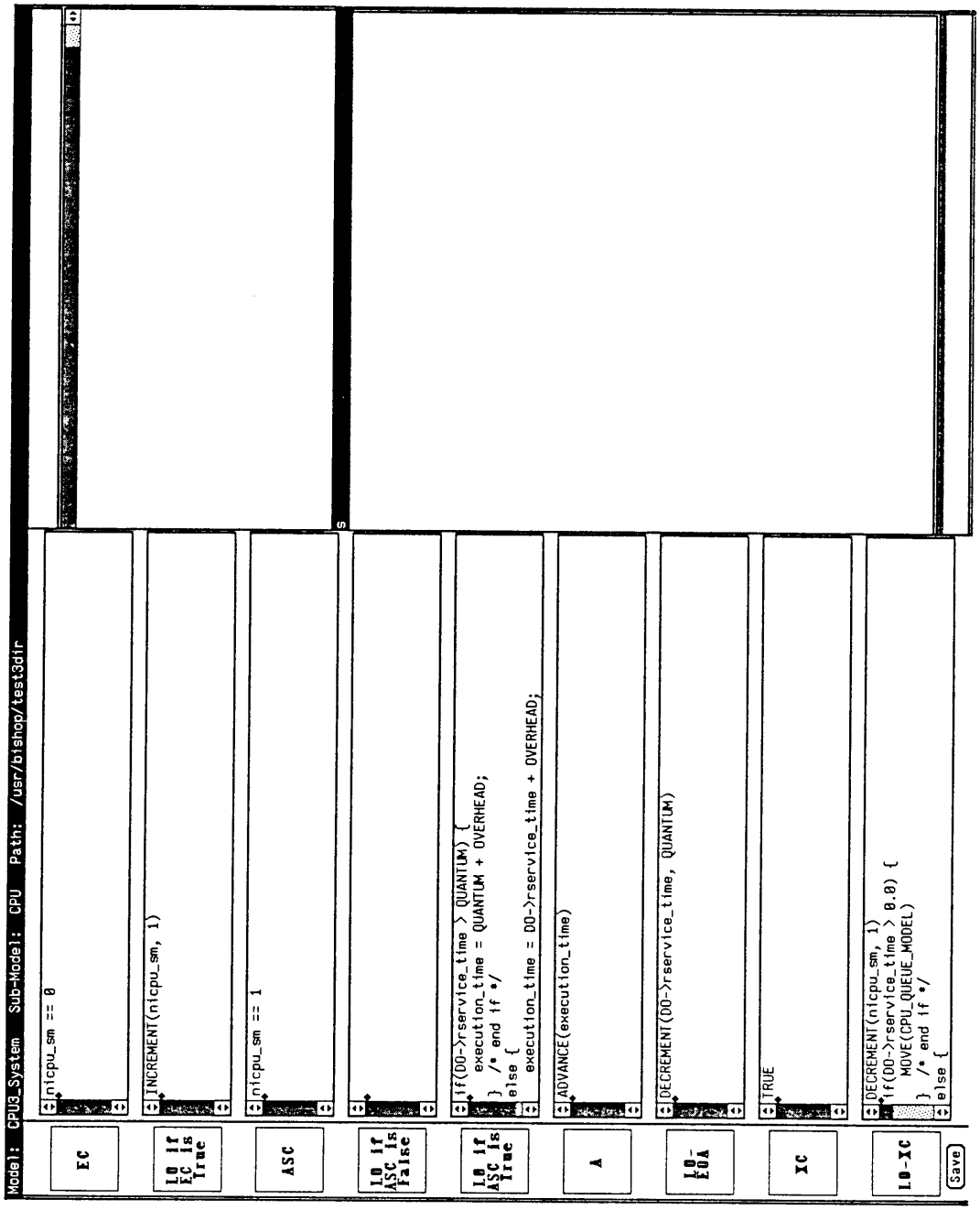

Figure 5.26 CPU Internal Logic Specification

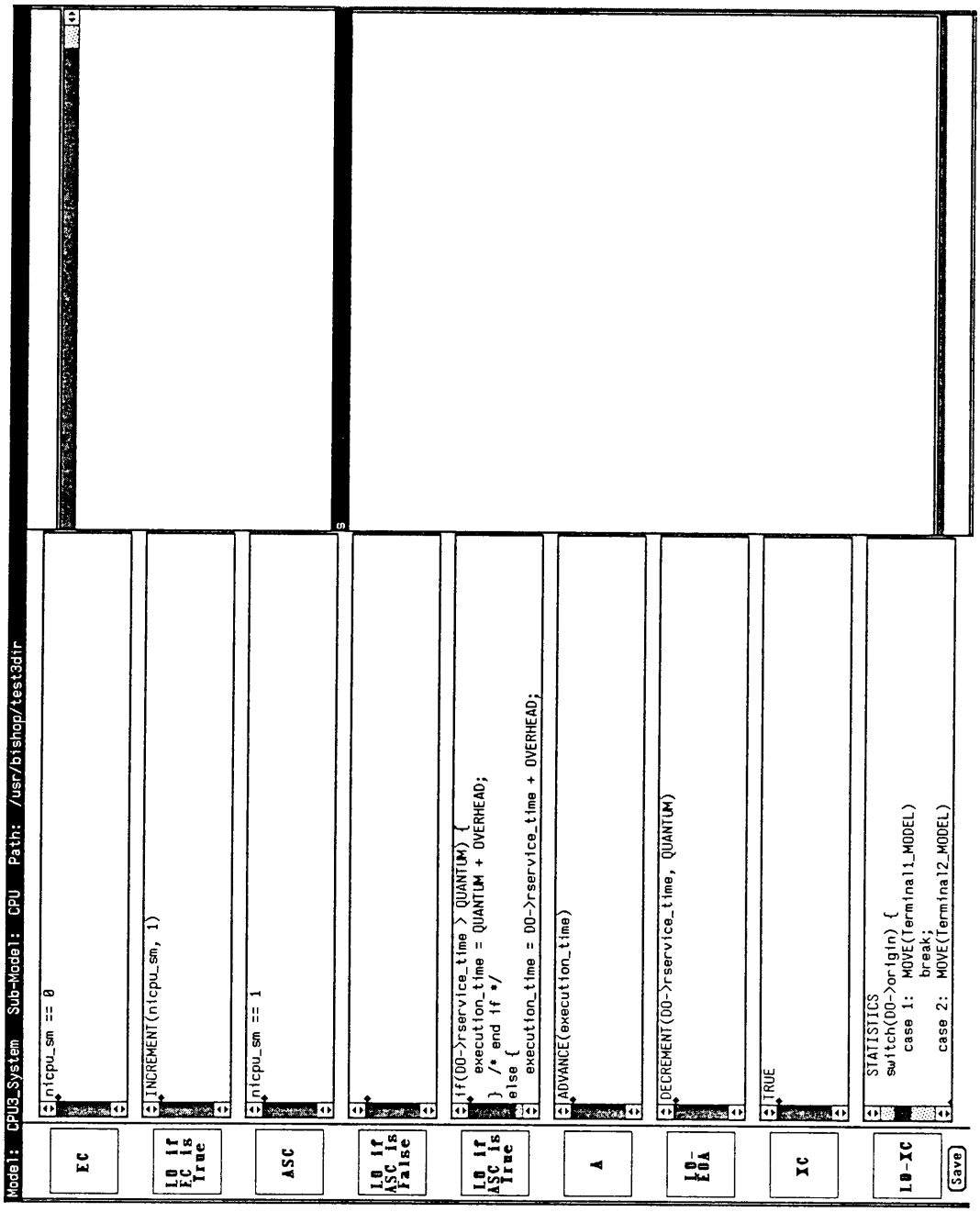

Figure 5.27 CPU Internal Logic Specification

decremented by one using the DECREMENT macro. If the remaining service time is greater than zero, the DO is moved back to the CPU_QUEUE. Otherwise, the "STATISTICS" macro is called to record a system departure and the DO is moved to its terminal of origin. This information is stored in the DO "origin" attribute (see Figure 5.27).

When finished, select the "Save" button on the lower left panel to exit the program and save the specification.

5.5.2 Submodel Attributes

Now that the submodel specifications are complete, it is necessary to define and specify the submodel attributes that are used in those specifications. To accomplish this, select the "Sub-Model Attribute" item from the pop-up menu in the specification mode shown previously in Figure 5.20. The submodel in which this is done in unimportant since all submodel attributes are global and are therefore accessible by all. This brings up a blocking requester for the input of the attribute name (see Figure 5.28). Select the "Name Attribute" button to enter the attribute specification tool.

As can be seen in Figure 5.29, "nicpu_sm" is an integer constant initialized to zero that describes the number of jobs in the CPU. Select the "Save" button to save the attribute information in the "attr_dat" table of the database (see Table 3.1 in Section 3.3.1.5).

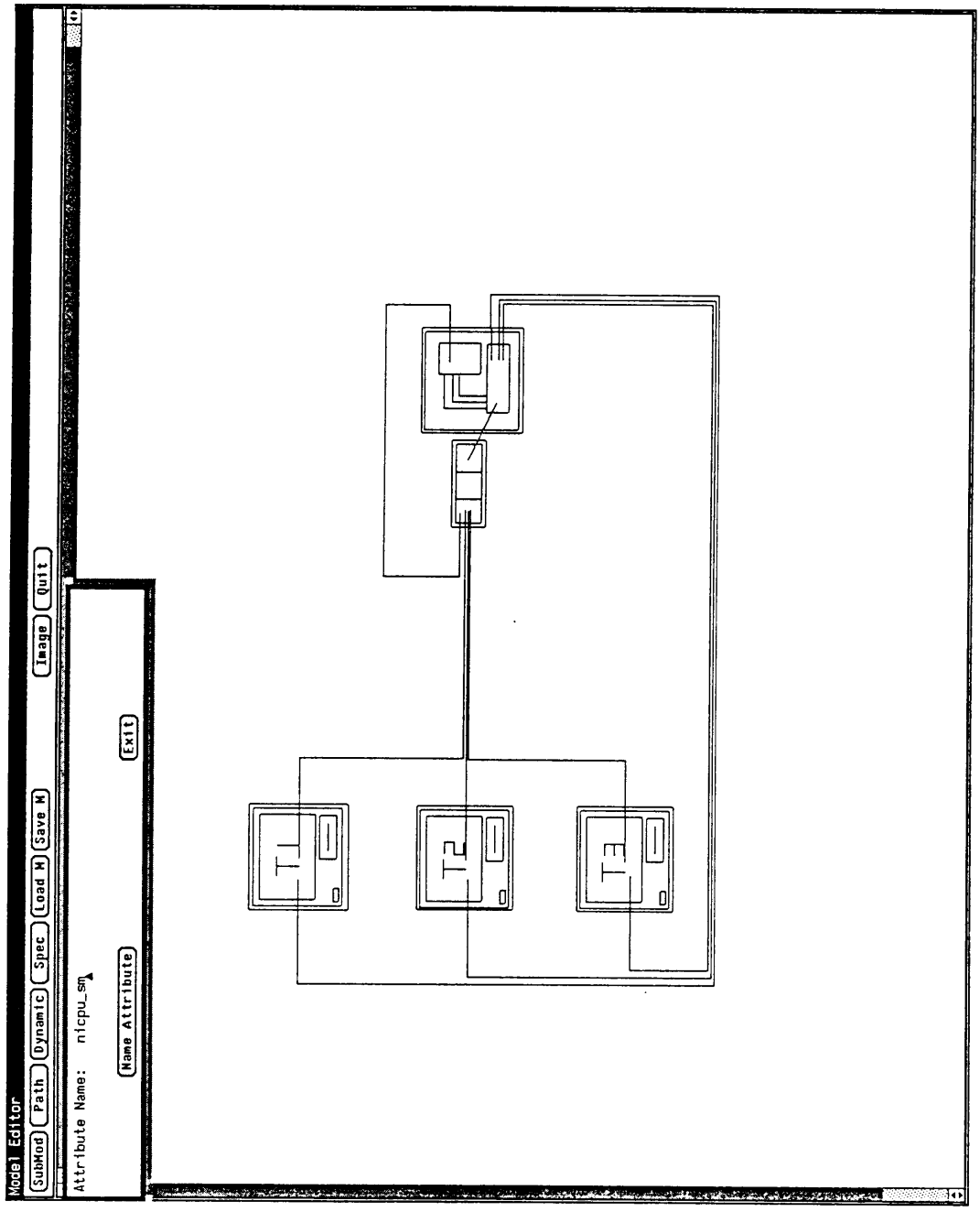

Figure 5.28 Attribute Name Requester

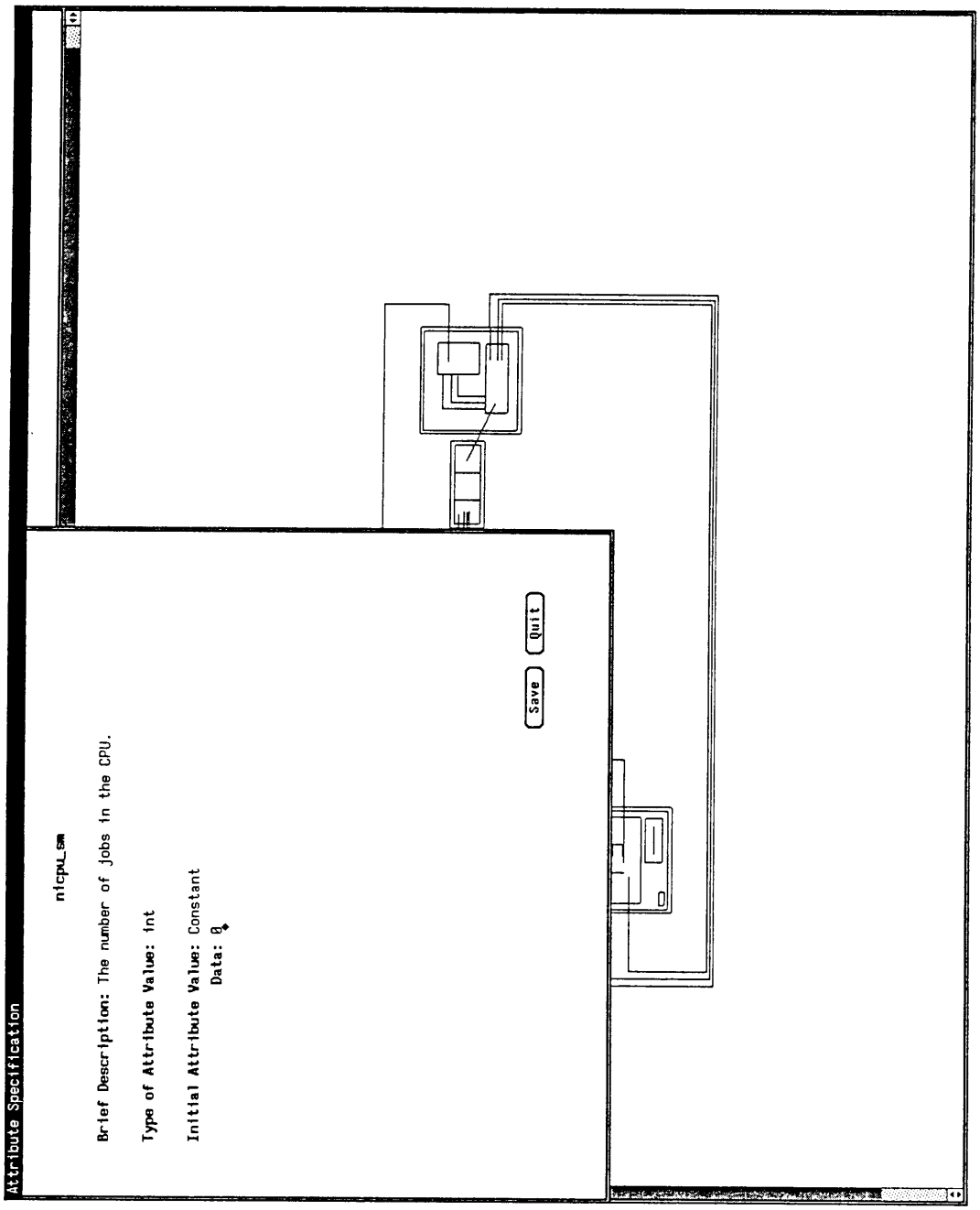

Figure 5.29 Nicpu_sm Attribute Specification

A similar process is performed for the "nicpu_queue_sm" (see Figure 5.30), "QUANTUM" (see Figure 5.31), "OVERHEAD" (see Figure 5.32), and "execution_time" (see Figure 5.33) attributes.

5.6 Dynamic Object Definition and Specification

Once this has been accomplished, we define the DOs. Select the "Dynamic" button on the model editor panel. This brings up a requester for the Dynamic Object name (see Figure 5.34). To enter the DO specification tool, select the "Name Dynamic Object" button.

The DO specification tool for job1 then appears on the screen (see Figure 5.35). Job1 is described as a job originating from Terminal 1. The name of the first submodel the DO tries to enter is "Terminal1". The DO image created earlier for a job originating from Terminal 1 is listed as the Object Image file. The interarrival time is constant and equal to zero. Lastly, the maximum number of generated DOs of this type is one.

The DO attributes "origin" and "rservice_time" used previously in the submodel specification should now be defined and specified. DO attributes are global to all DOs. Thus a given attribute need only be defined once. Select the "Specify DO Attribute" button to bring up a blocking requester for the DO attribute name (see Figure 5.36). Select the "Name Attribute" button to enter

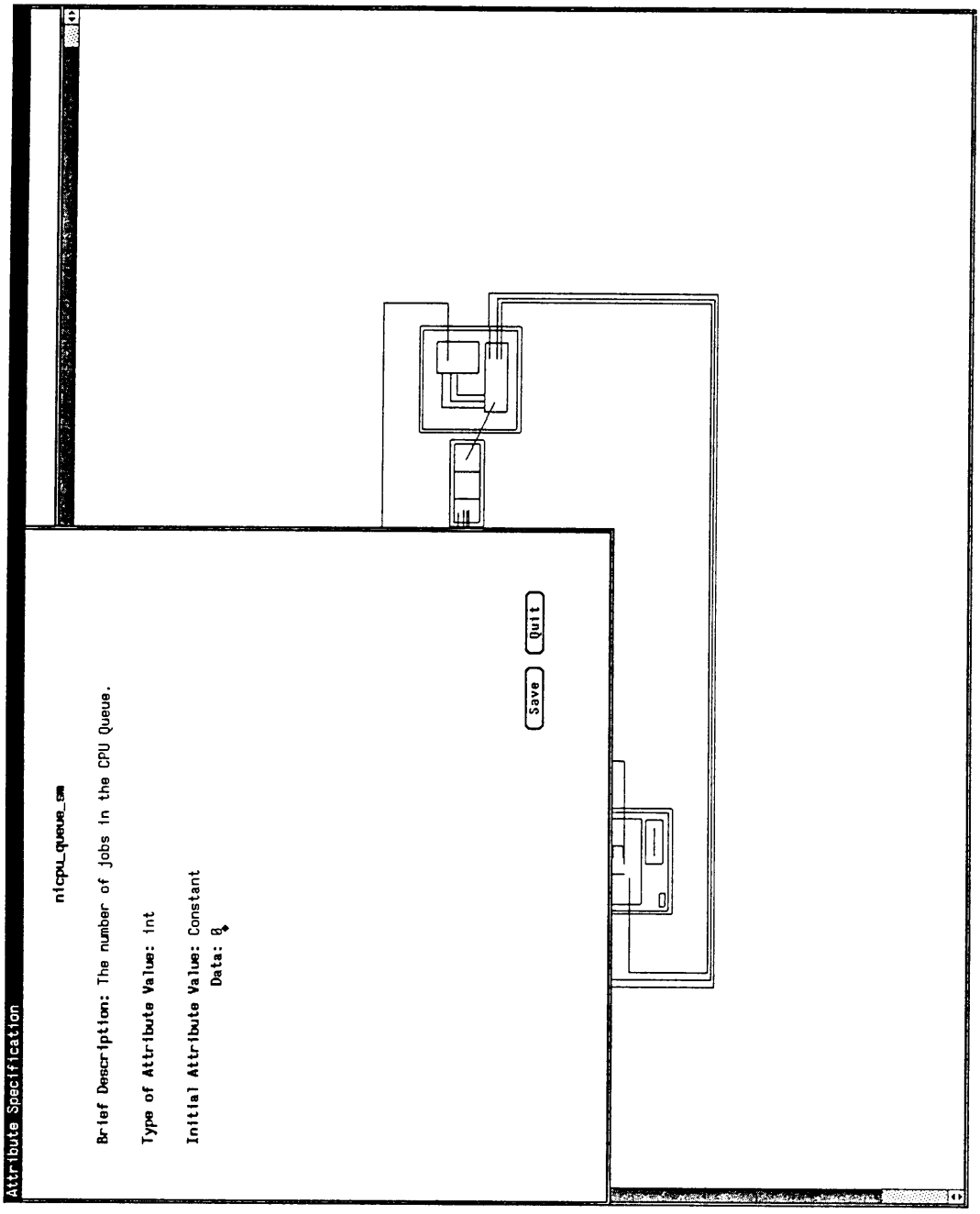

Figure 5.30 Nicpu_queue_sm Attribute Specification

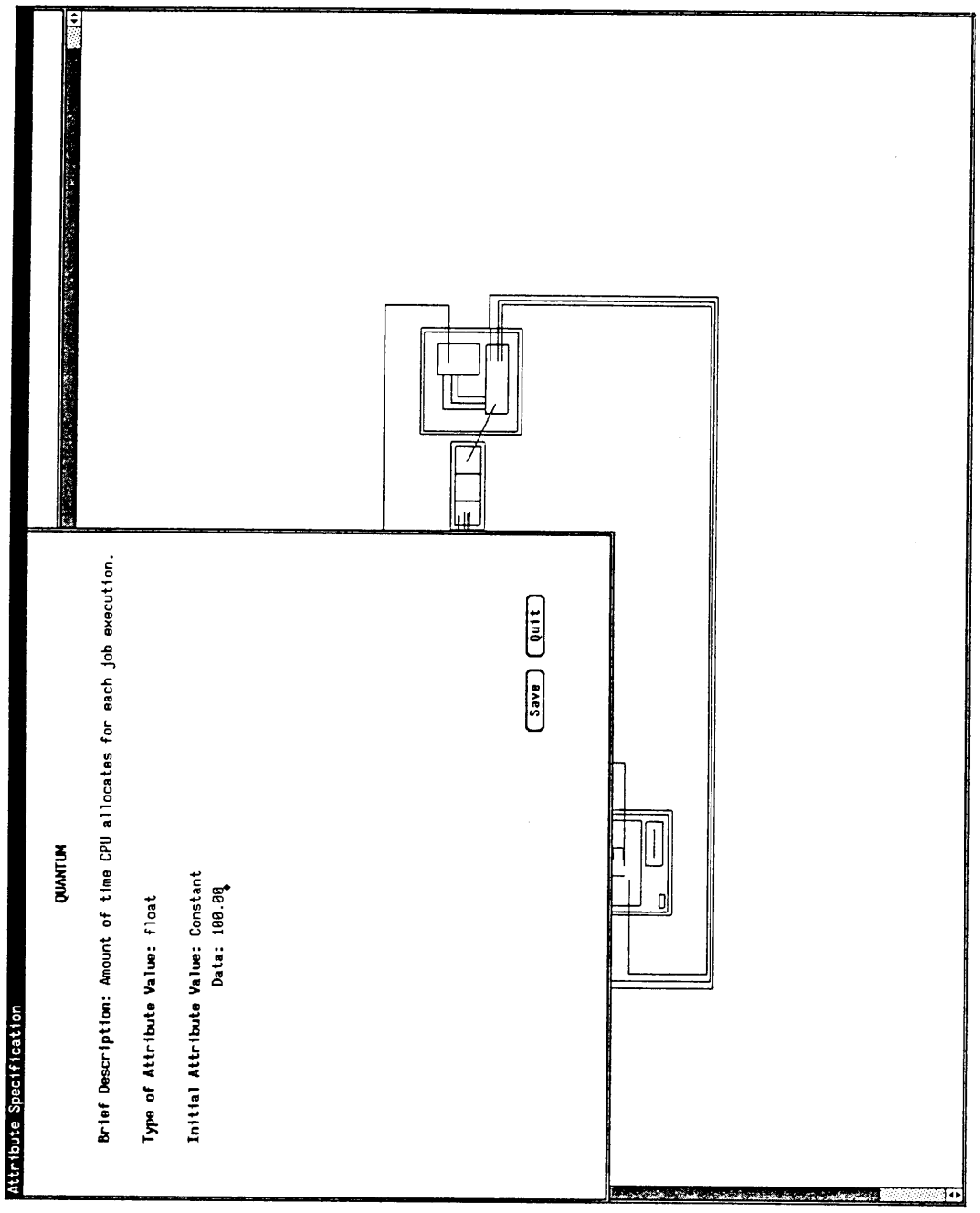

Figure 5.31 QUANTUM Attribute Specification

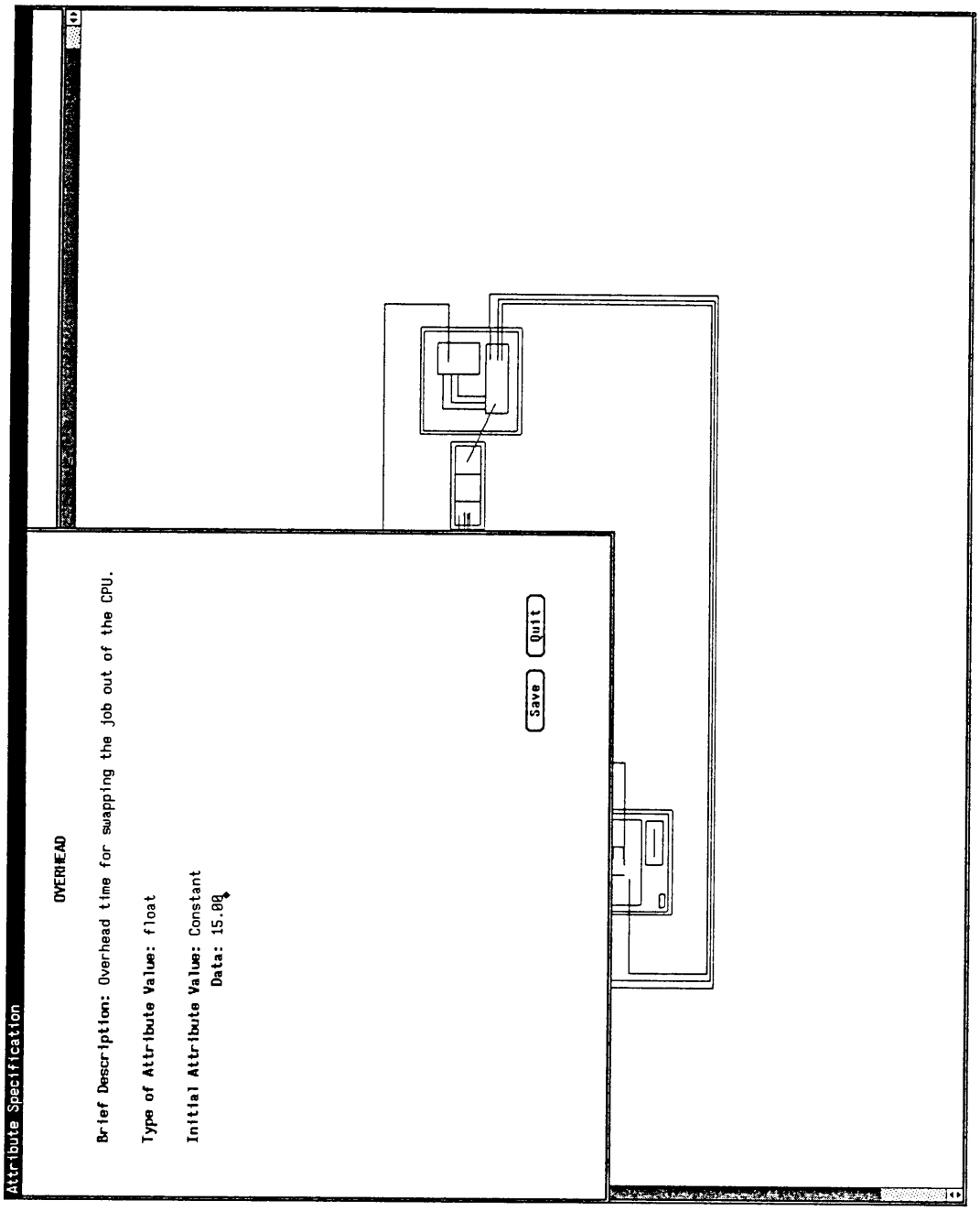

Figure 5.32 OVERHEAD Attribute Specification

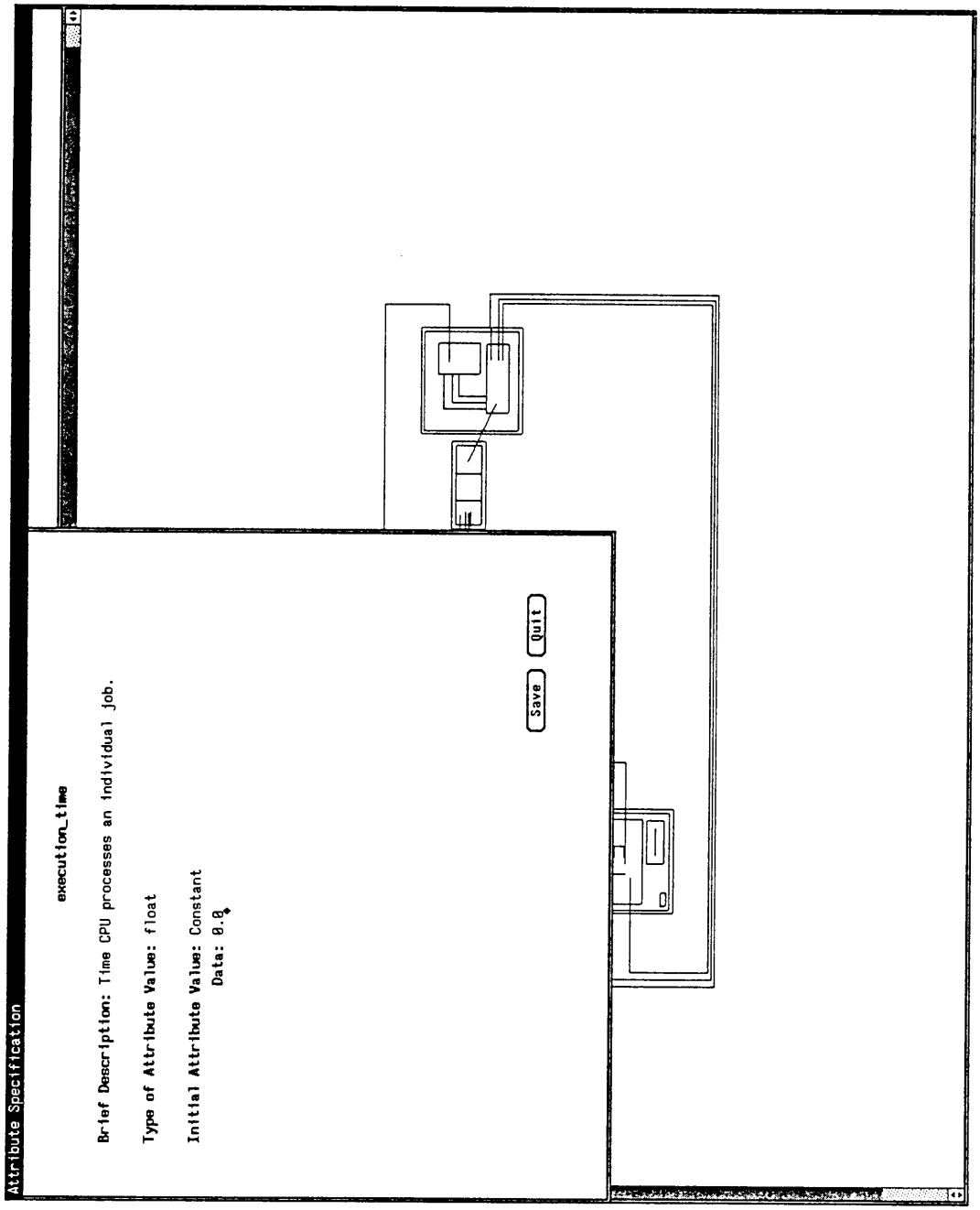

Figure 5.33 Execution_time Attribute Specification

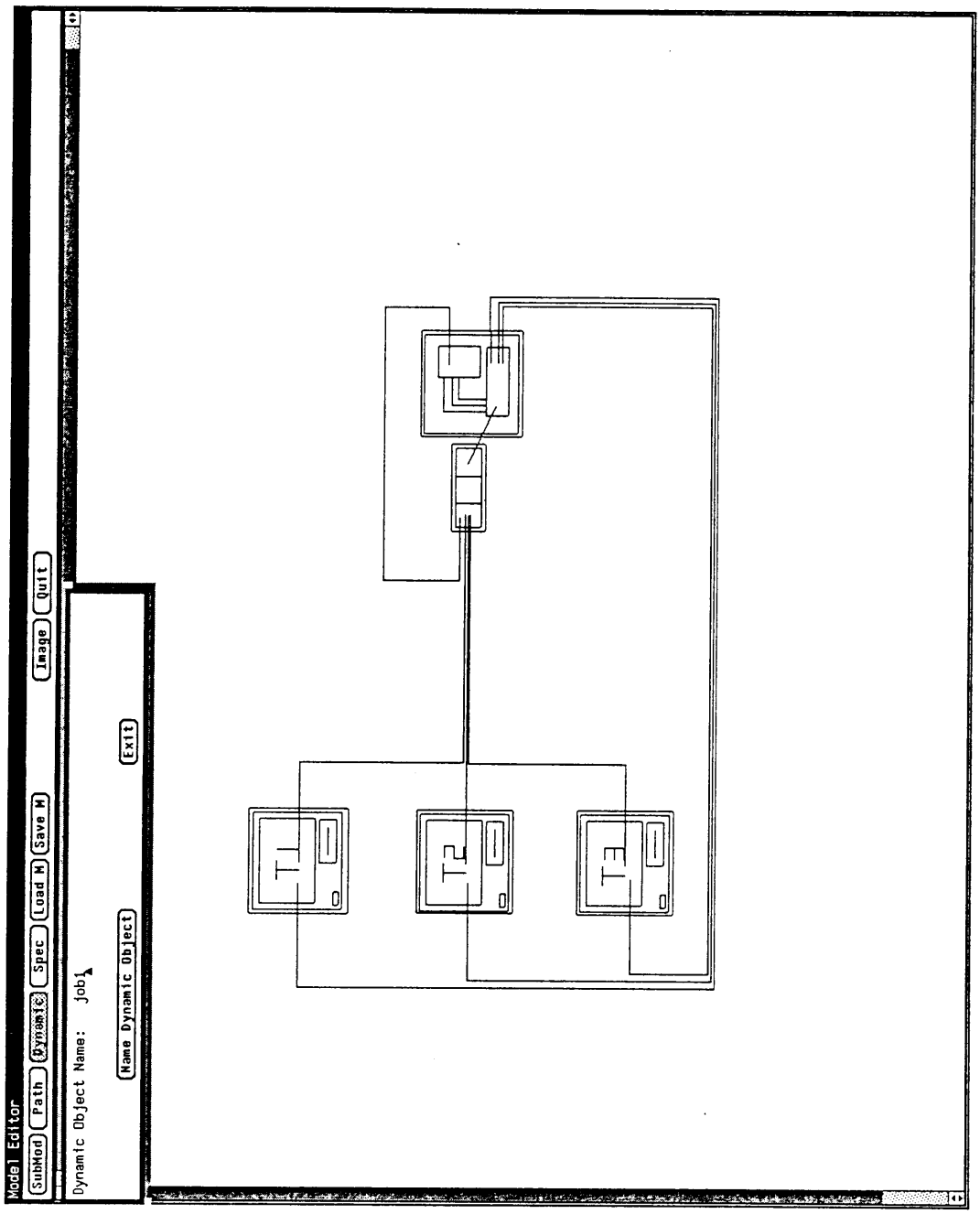

Figure 5.34 DO Name Requester for Job1

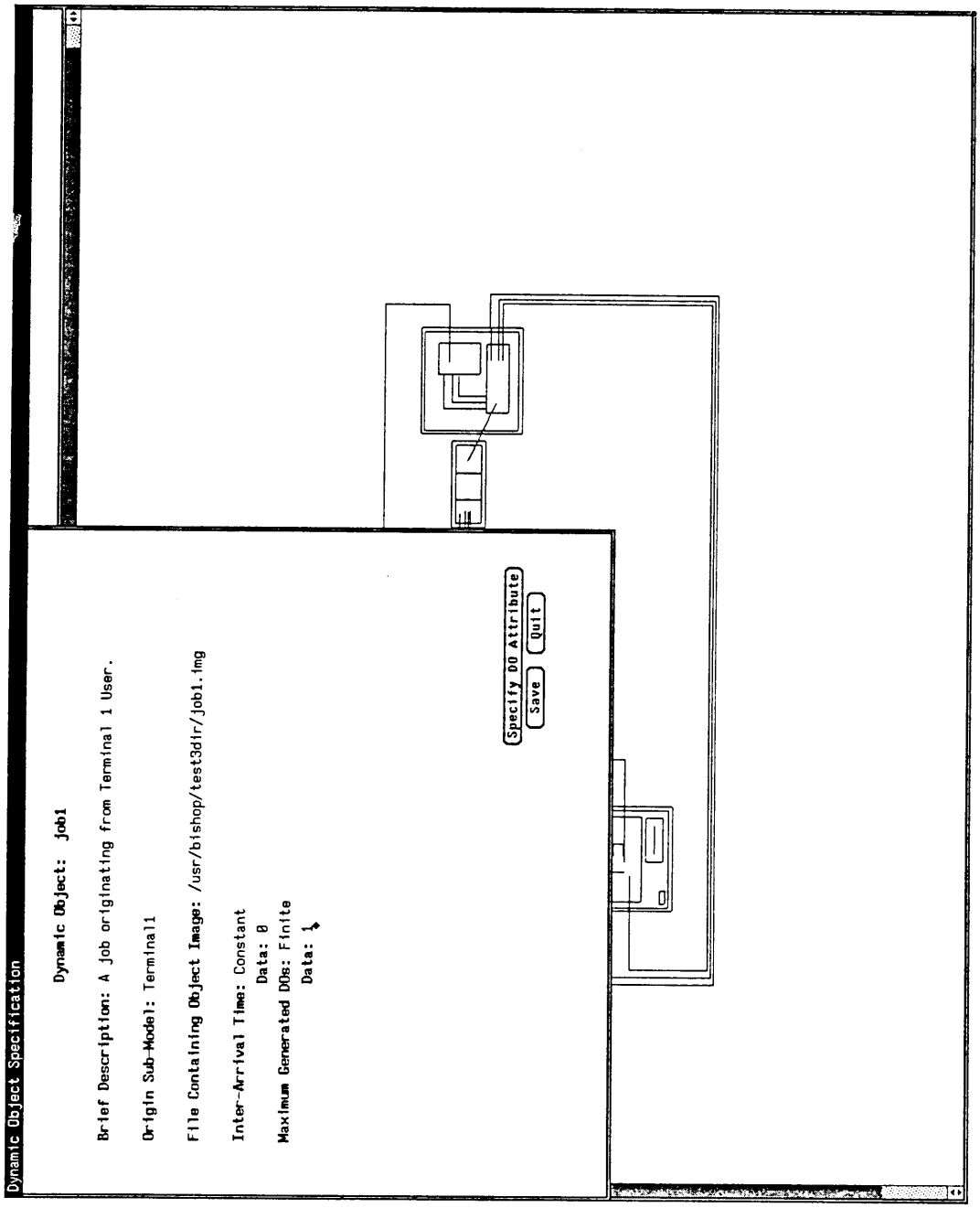

Figure 5.35 Job1 DO Specification Tool

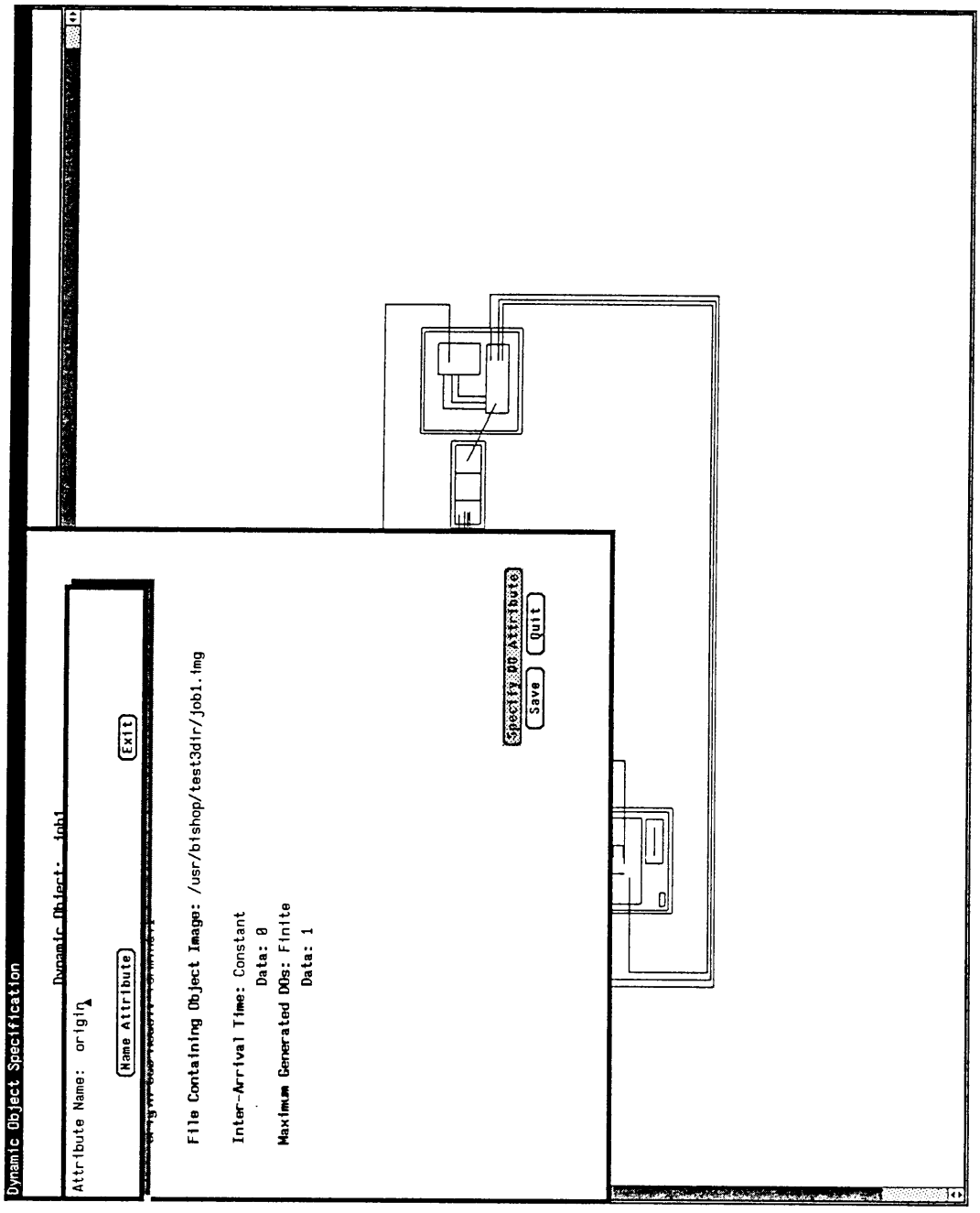

Figure 5.36 DO Attribute Name Requester for Origin

the attribute specification tool (see Figure 5.37). DO attribute specification is identical to submodel attribute specification from this point forward.

A similar process is used to define the DO attribute rservice_time (see Figure 5.38).

When all DO attributes have been defined, exit the DO specification tool by selecting the "Save" button. This exits the program and put appropriate entries in the "dynamic_dat" table of the database (see Table 3.1 in Section 3.3.1.5).

A similar process is used to define the DOs job2 (see Figure 5.39) and job3 (see Figure 5.40).

The image editing/model editing process is now complete. Quit the Image Editor/Model Editor by selecting the "Quit" button on the main panel.

5.7 The Model Generator

The model generator is used to convert the model definition and specification into an executable simulation. Select the "Model Generator" icon on the top level menu (see Figure 5.1) to bring up a blocking requester that should already have the system model name entered (see Figure 5.41). Select the "Generate Code" button to create an executable visual simulation model (see Figure 5.42). The blocking requester clears after the completion of the generation process.

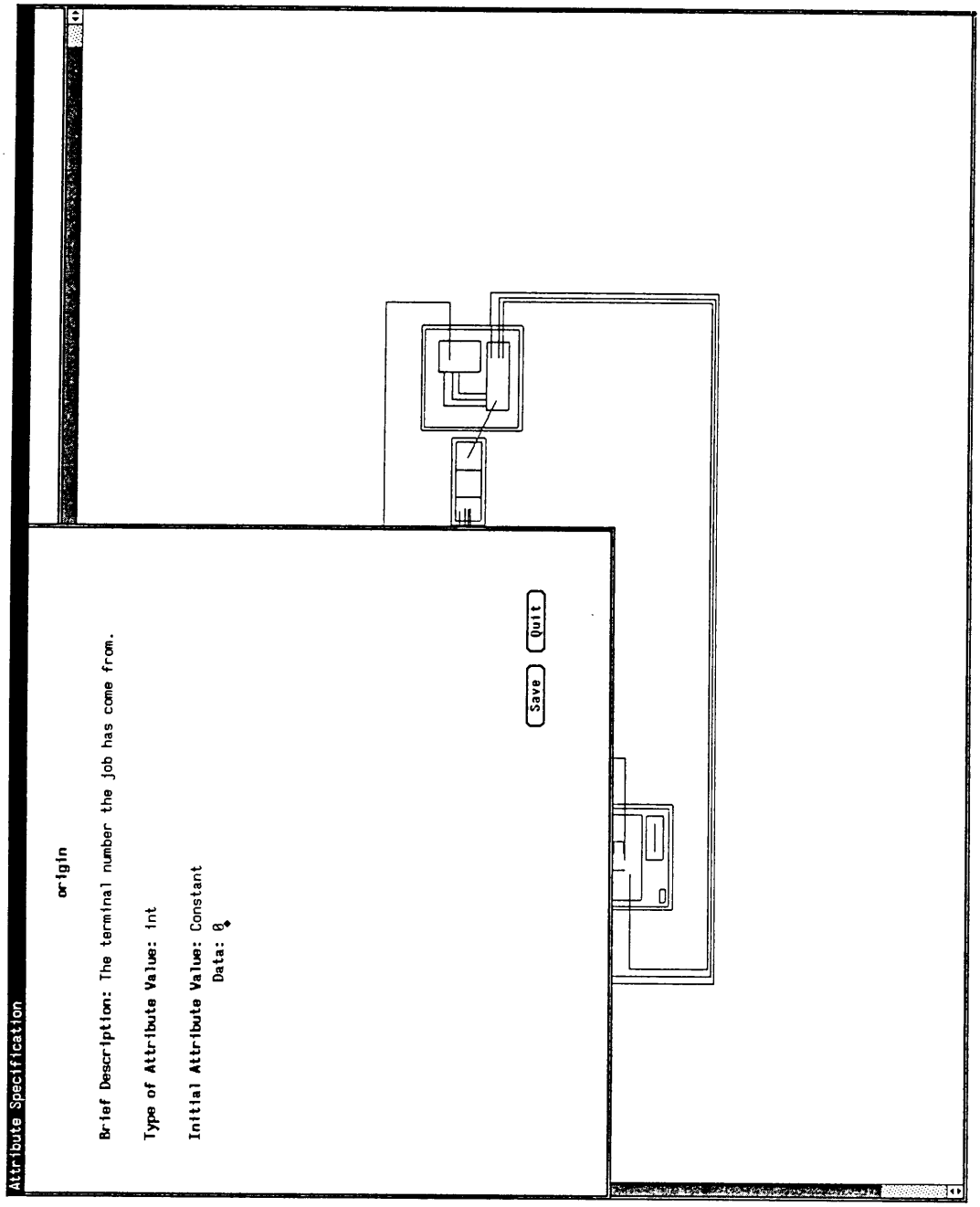

Figure 5.37 Origin DO Attribute Specification Tool

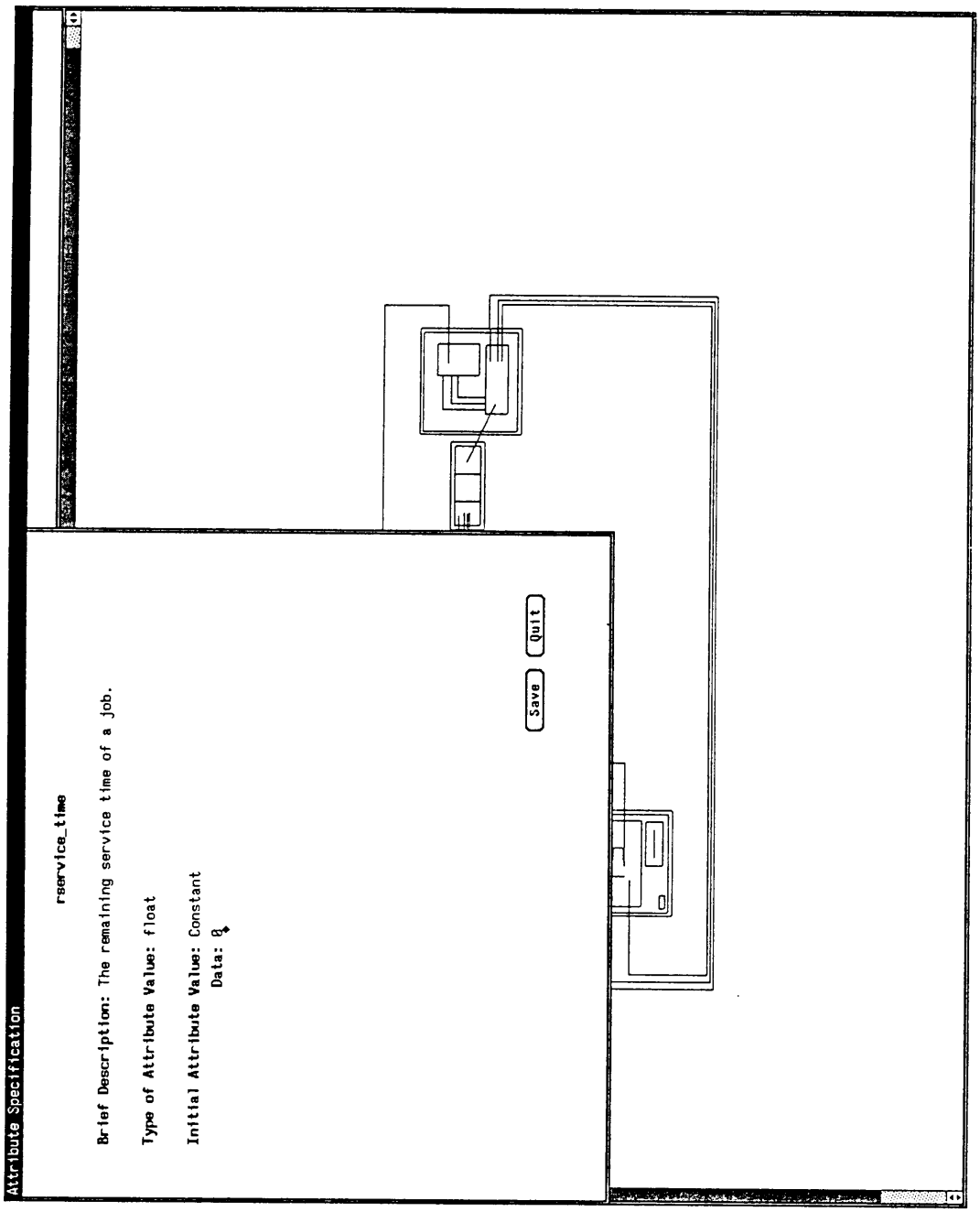

Figure 5.38 Rservice_time DO Attr. Spec. Tool

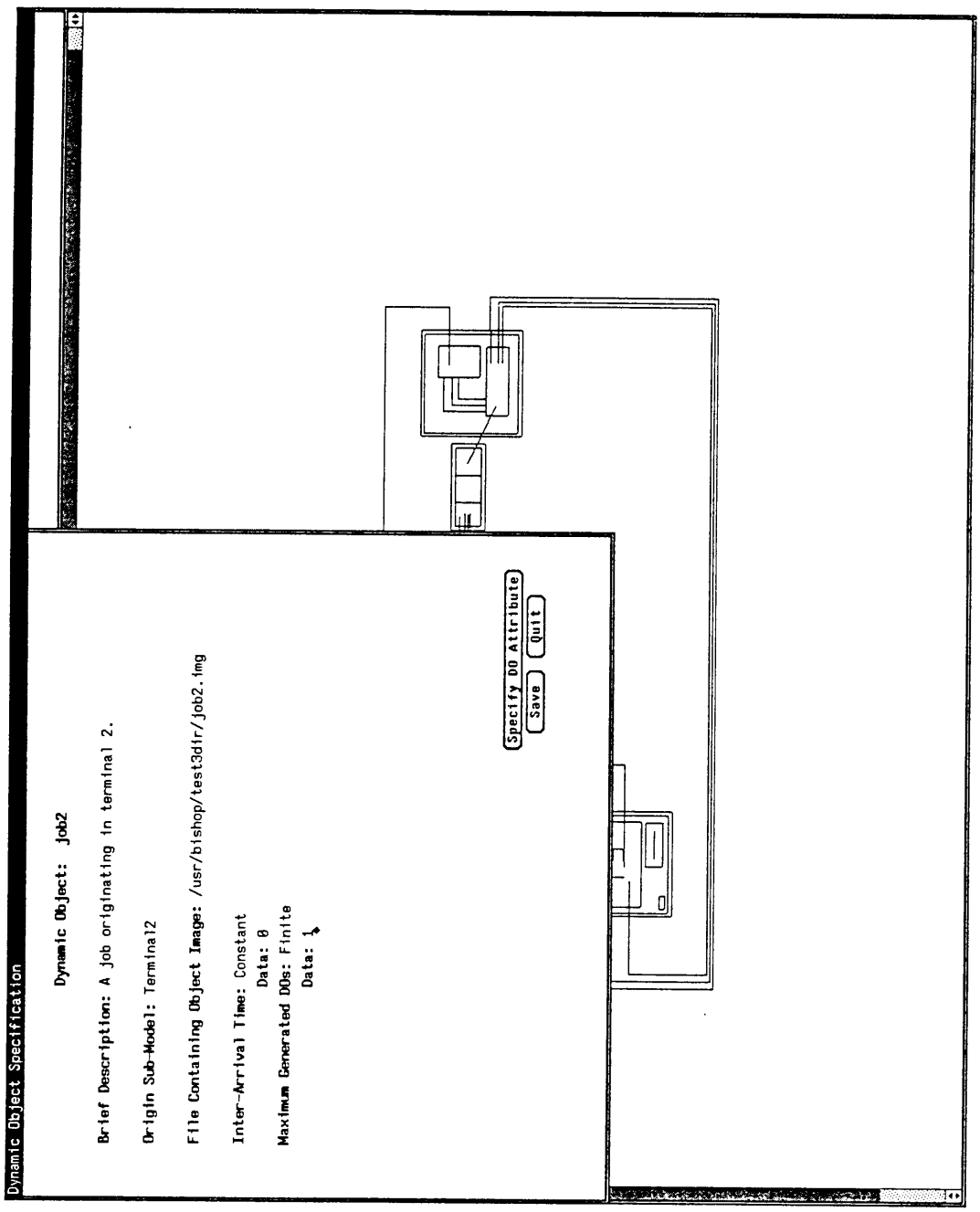

Figure 5.39 Job2 DO Specification Tool

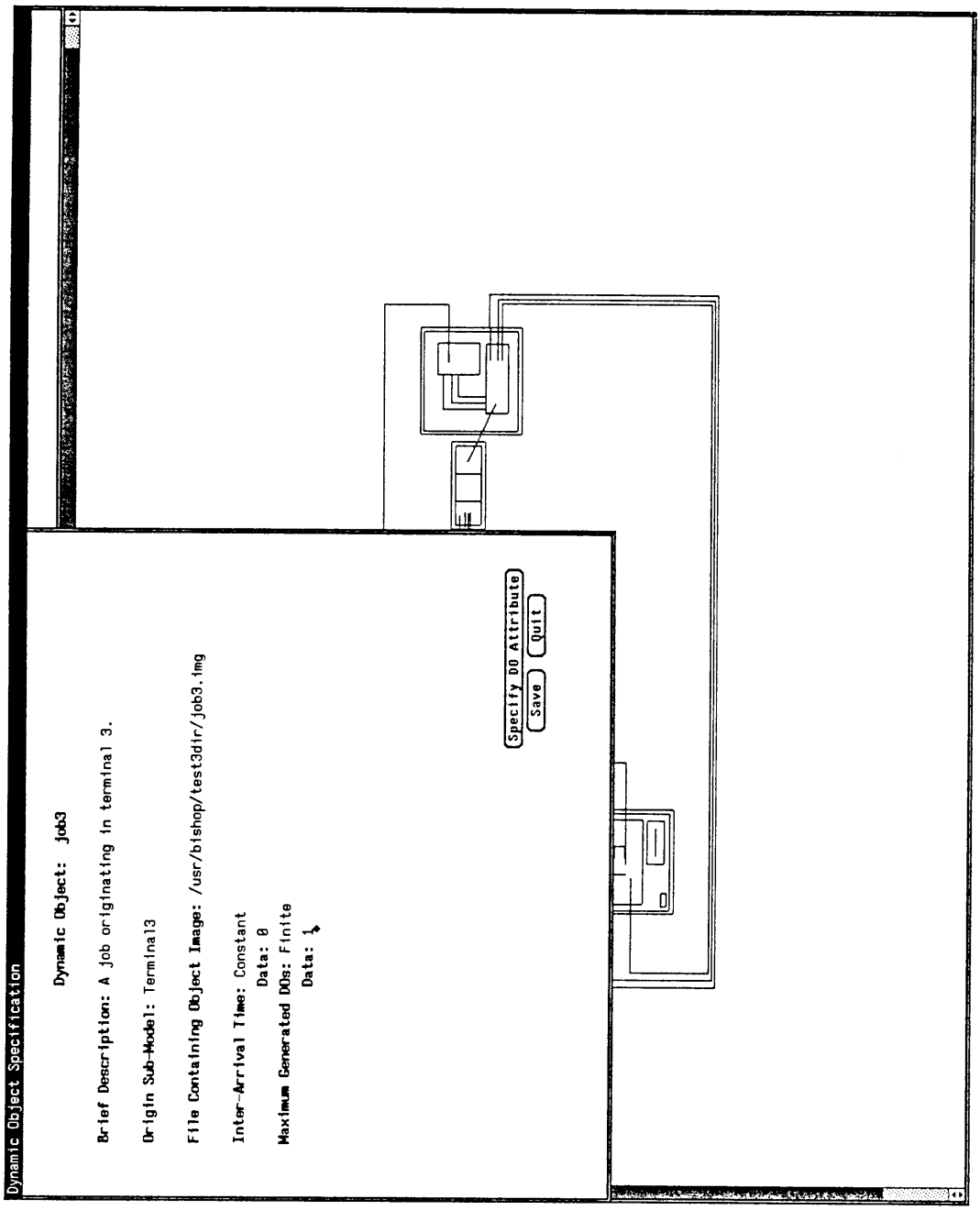

Figure 5.40 Job3 DO Specification Tool

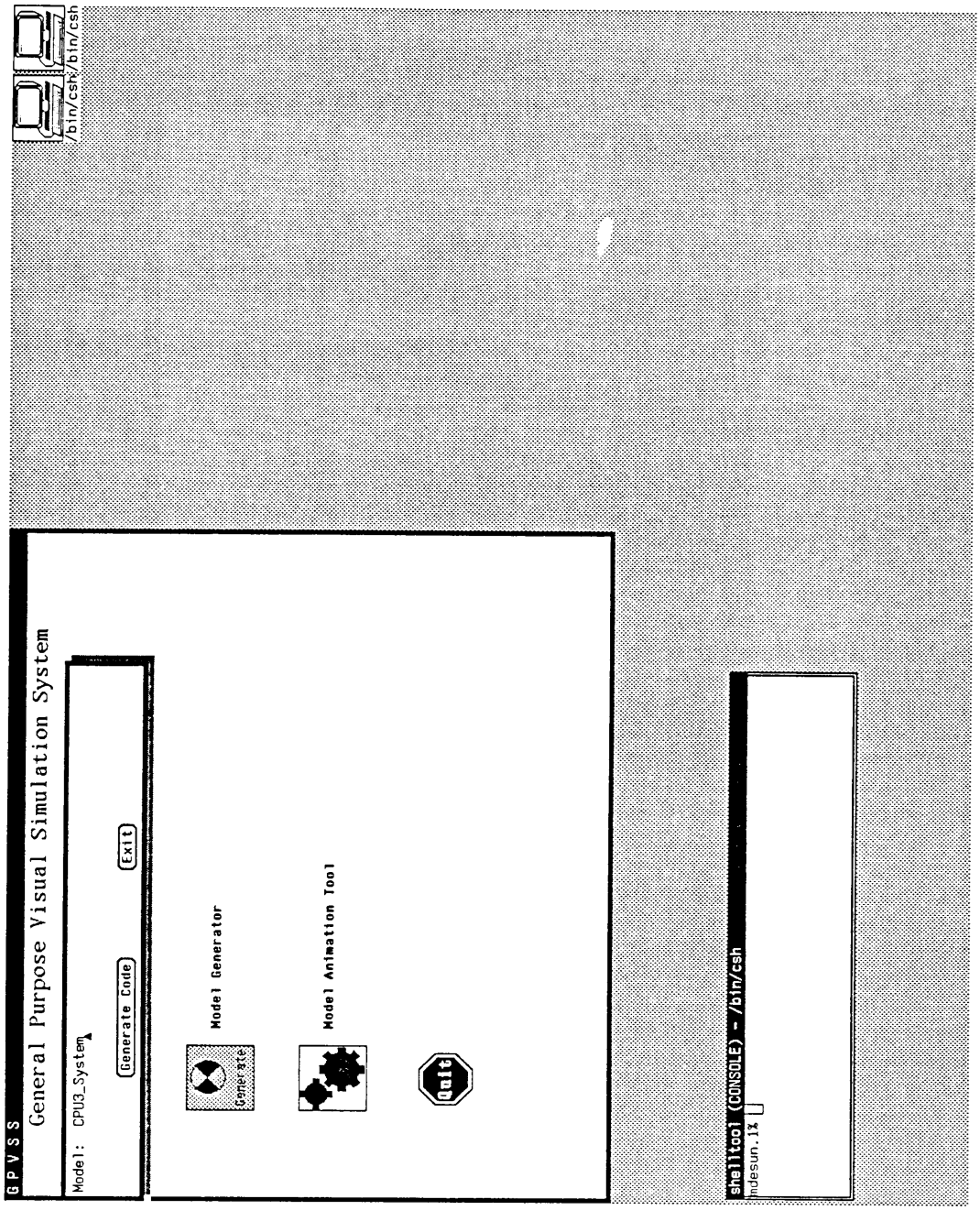

Figure 5.41 Model Generator Requester

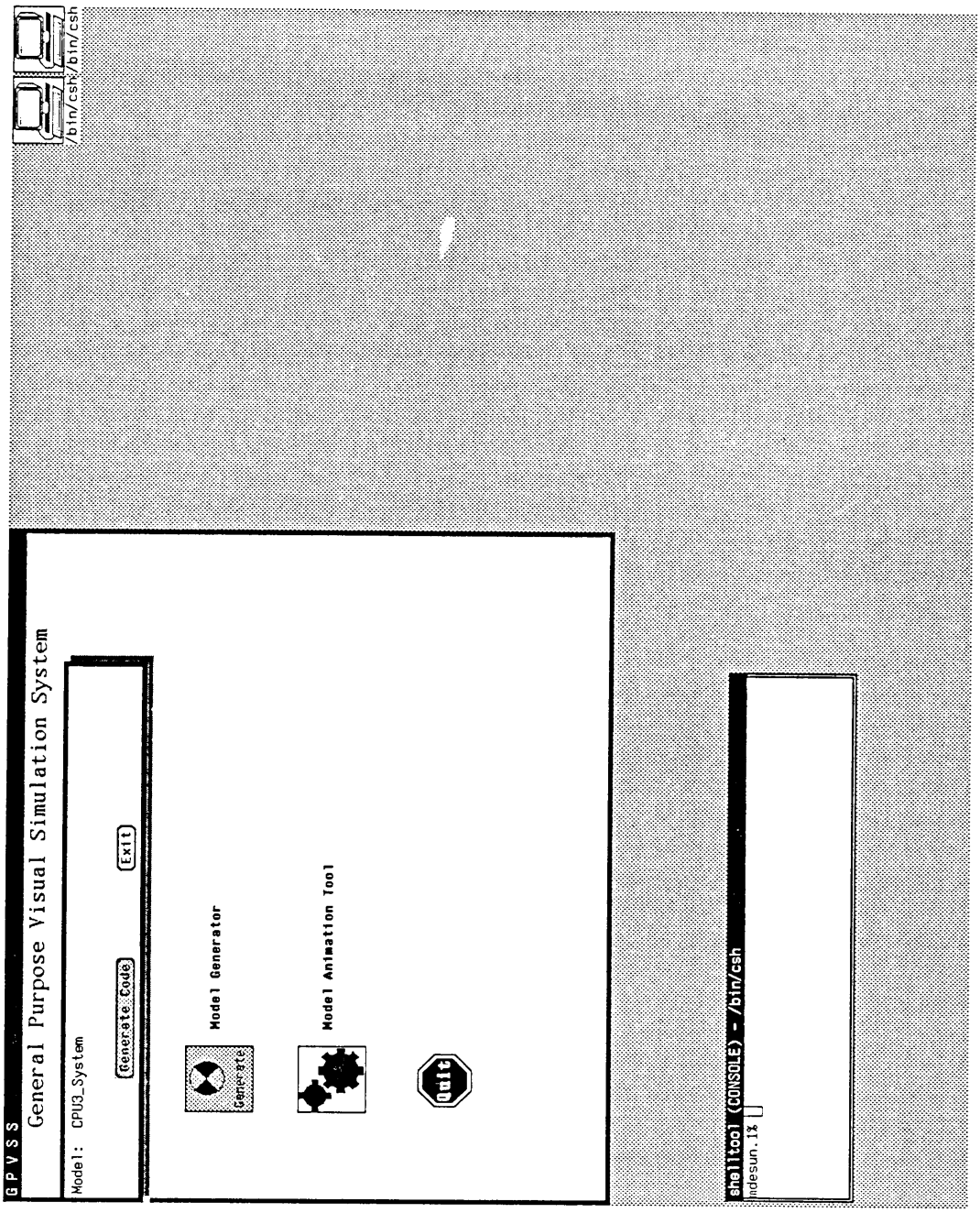

Figure 5.42 Generating the Model

5.8 The Model Animation Tool

To run the Visual Simulation, select the "Model Animation Tool" icon and wait for the dynamic display screen to come up (see Figure 5.43). Select the "Load" button to load the background image and initialize variables (see Figure 5.44). Select the "Animate" button to toggle the VS on and off (see Figure 5.45). Selecting the "Run" button brings up the statistics panel (see Figure 5.46). The "Run Simulation" button runs the model without the dynamic display (see Section 3.5.2) and is used for the gathering of statistical information. Results obtained from this run are listed in Table 5.1.

Finally, exit the top_level menu by selecting the "Quit" button. This quits GPVSS entirely.

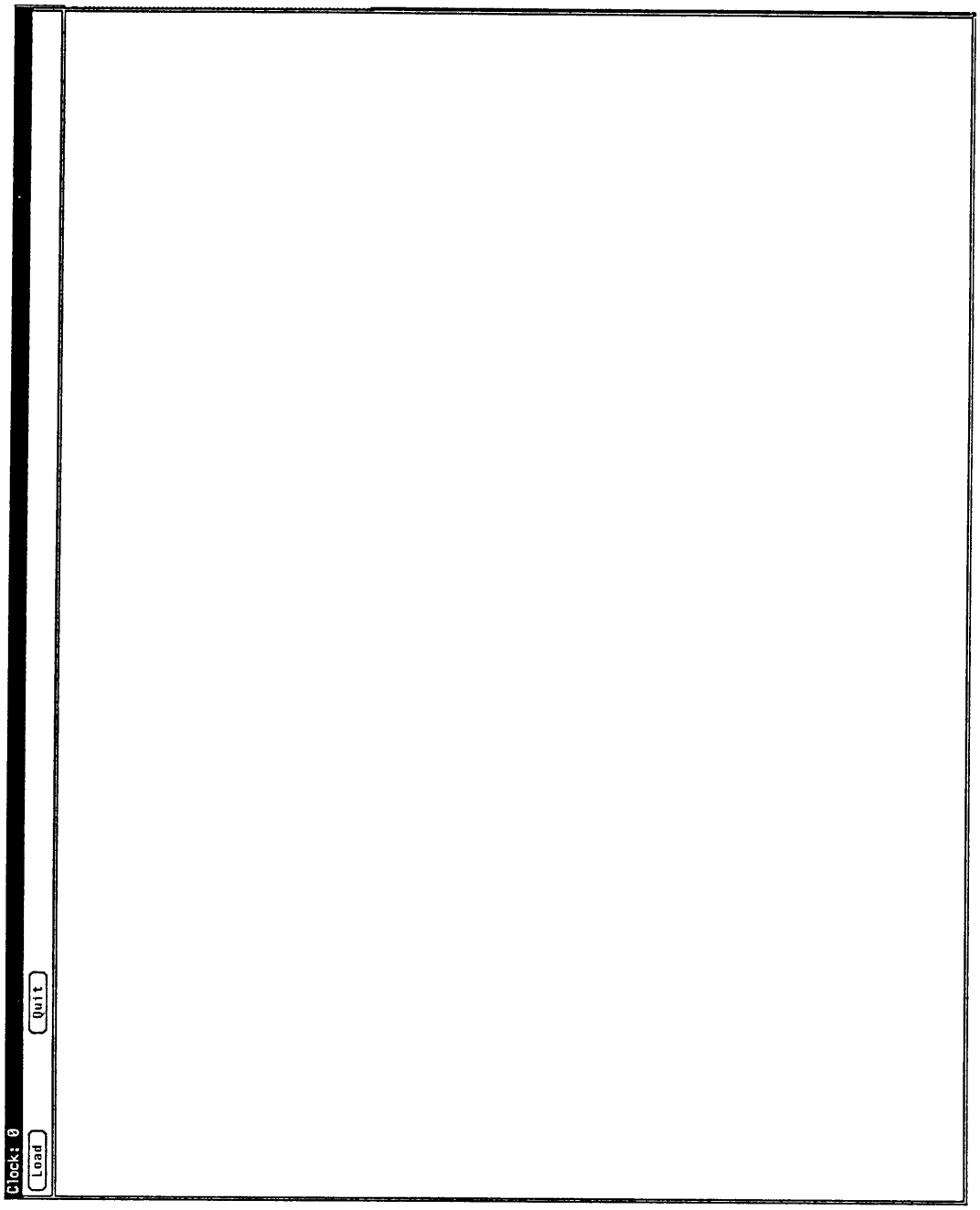

Figure 5.43 Model Animation Tool Screen

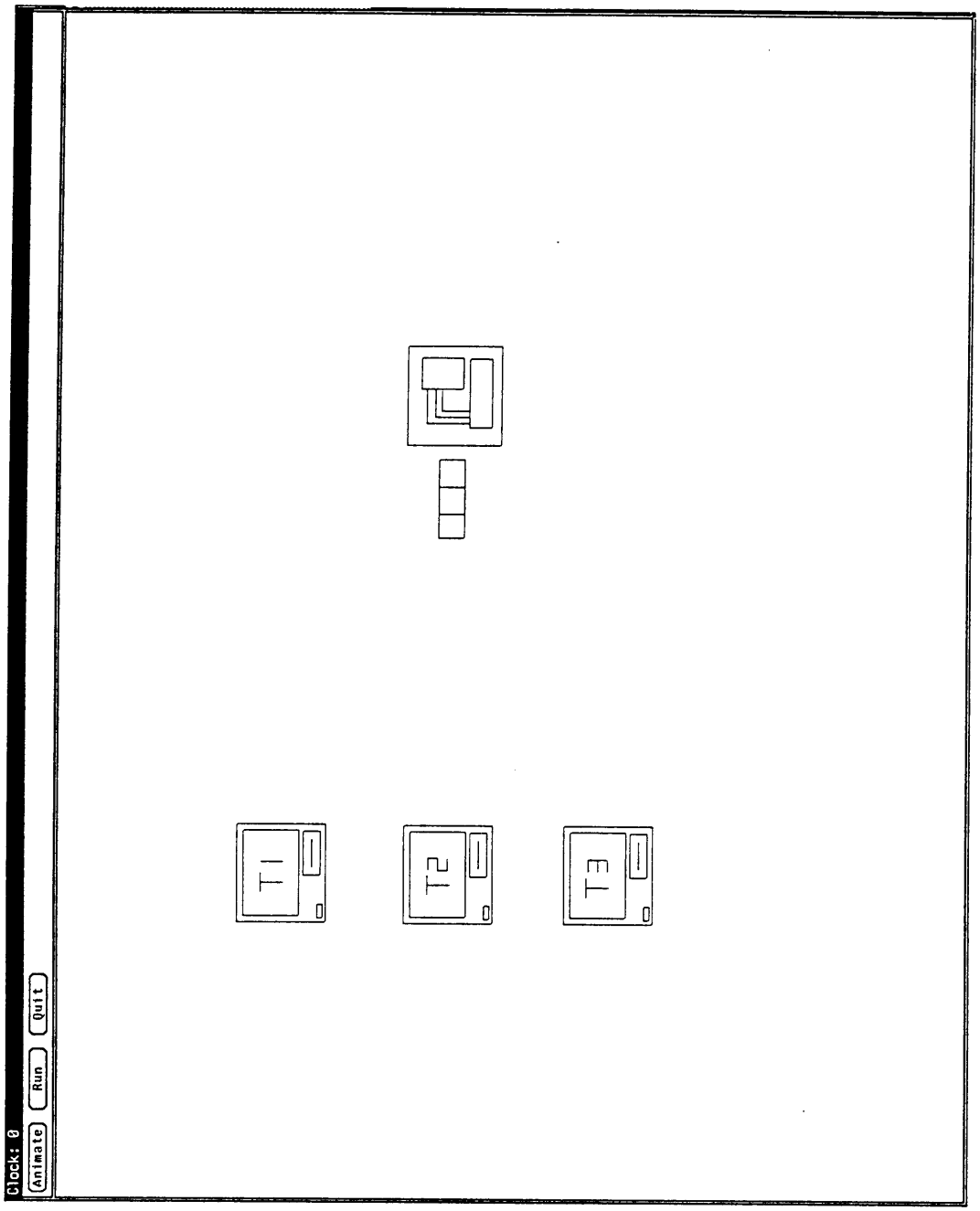

Figure 5.44 Animation Tool After Load Operation

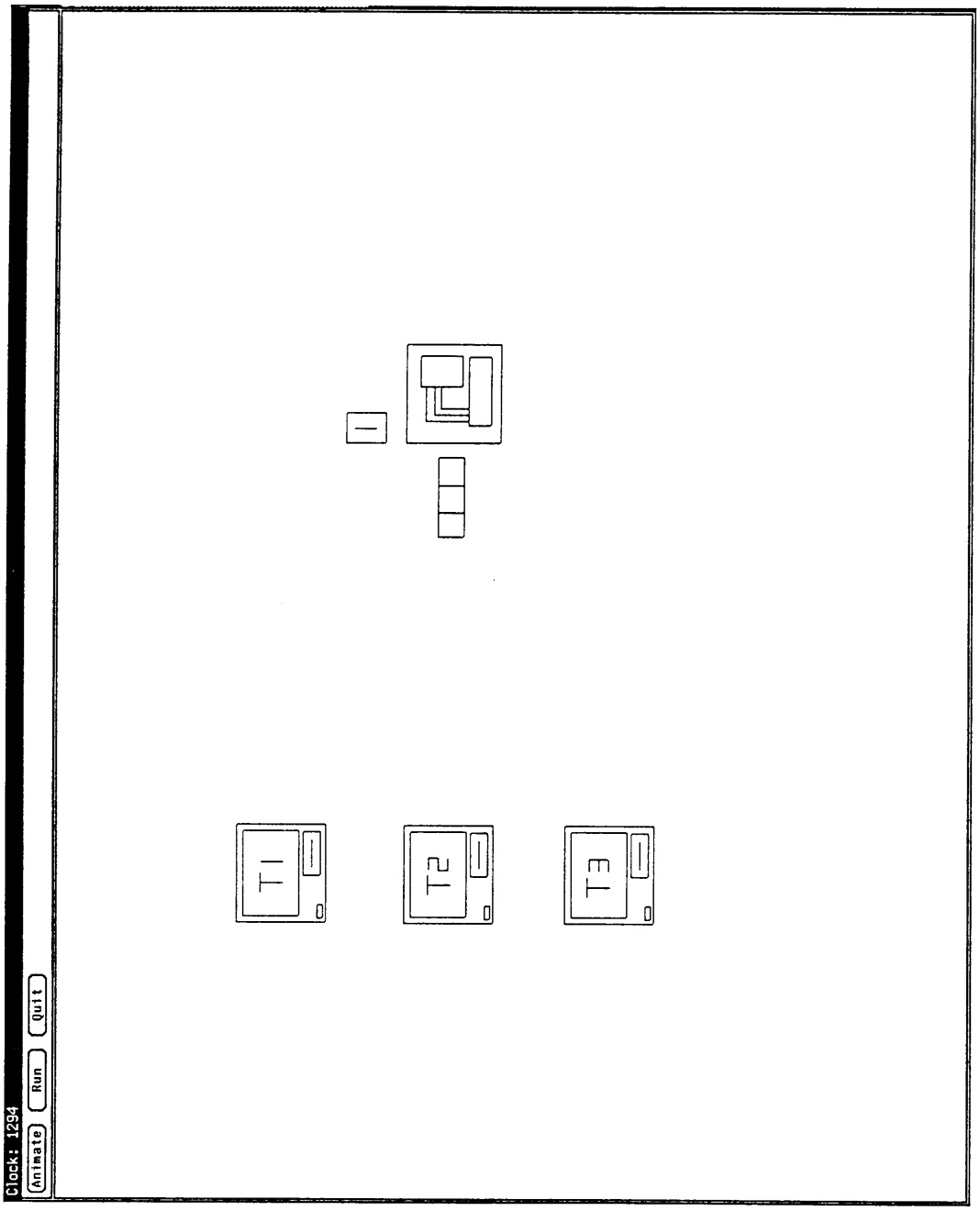

Figure 5.45 The Dynamic Display

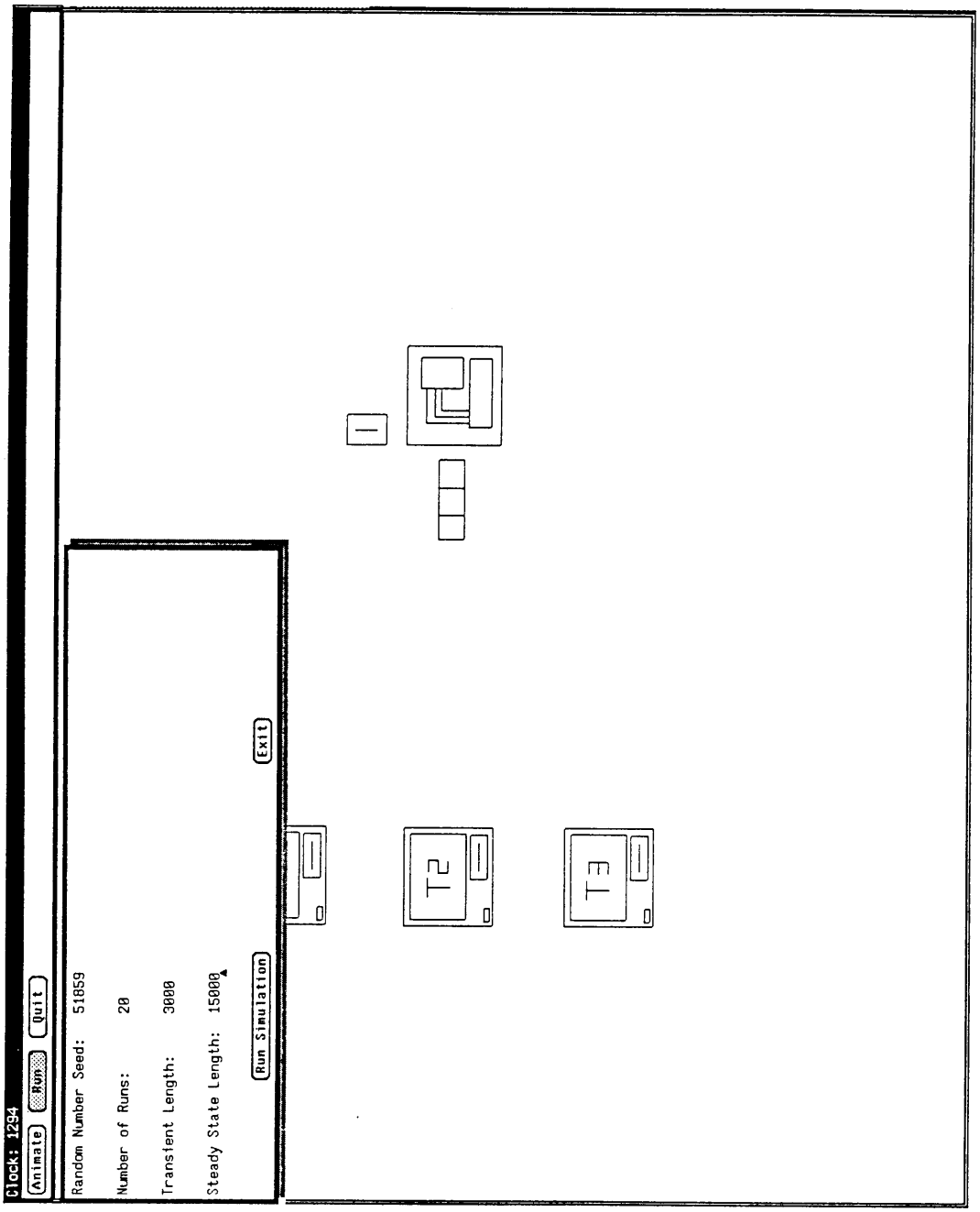

Figure 5.46 The Experimental Model

Table 5.1 Round Robin System Experimental Results

Average Waiting Time

VALUES GIVEN:

1507.162	1515.372	1483.607	1509.918	1498.039
1526.684	1513.152	1516.152	1496.858	1518.415
1510.644	1517.421	1516.233	1512.524	1496.327
1476.913	1517.480	1502.854	1498.681	1488.792

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 1506.161377
Sample Variance = 169.530167

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	1502.295	1501.127	1500.068	1498.769	1497.832
Upper limit	1510.028	1511.195	1512.255	1513.554	1514.491

Average Number of Jobs in the System

VALUES GIVEN:

2.572	2.575	2.560	2.575	2.568
2.584	2.569	2.574	2.572	2.575
2.575	2.579	2.575	2.575	2.569
2.571	2.577	2.567	2.574	2.567

CONFIDENCE INTERVALS FOR THE DATA

Number of data points = 20
Sample Mean = 2.572650
Sample Variance = 0.000023

Confidence Intervals:

Level	0.100	0.050	0.025	0.010	0.005
Lower limit	2.571	2.571	2.570	2.570	2.570
Upper limit	2.574	2.575	2.575	2.575	2.576

CHAPTER 6

SUMMARY AND FUTURE RESEARCH

6.1 Summary

The literature review presented in Chapter 2 clearly indicates the need for an integrated visual simulation system that provides for the graphical definition and interactive specification of the model while maintaining application independence. GPVSS is an interactive graphics oriented general purpose simulation system designed to produce a running visual simulation from a graphical model design. GPVSS provides facilities to: (1) graphically design the model and its visualization, (2) interactively specify the model's logic, and (3) automatically generate the executable version of the model, while maintaining domain independence.

Chapter 3 provides an extensive detailed description of the internal logic of GPVSS. GPVSS is prototyped on a Sun 3/160C computer workstation using the SunView graphical interface. The Image Editor/Model Editor provides the facilities for the graphical design of the model and its visualization as well as interactive specification of the model logic. The Generator automatically creates the Animation Tool which is the executable version of the model.

Chapter 4 presents three case studies and shows that

the internal logic of GPVSS and the logic of the models are sufficiently accurate. Verification of GPVSS is achieved through the use of *desk checking* and *tracing* techniques. Validation is achieved by modeling three systems, each admitting a closed-form analytical solution, and comparing the experimental and analytical results. In every case the analytical estimate of the mean falls within the 90% confidence interval.

Chapter 5 illustrates the usage of GPVSS. Using a case study of a round robin system, the user is shown step by step how to develop and execute a visual simulation.

The case studies presented clearly show that GPVSS is an adequate visual simulation system for a wide variety of problems. Thus, the goal of prototyping a software system that aids a simulationist in developing a general purpose discrete event visual simulation model has been achieved.

6.2 Recommendations for Future Research

GPVSS has been prototyped by using the evolutionary software development approach. Because of this, a variety of characteristics of GPVSS could stand improvement.

During GPVSS development, database tables were created as the needs for them arose. Subsequent development often rendered attributes of these tables obsolete, or duplicated information unnecessarily. While there are no unused attributes in the GPVSS databases currently, there remains

considerable duplication of information. A useful modification would be to consolidate the database as much as possible to maximize the sharing of information between the various GPVSS components. For instance, although a DO image may be stored with its name as a key, the DO specification tool requires the user to know the name of the file containing this image.

The drawing environment of GPVSS is relatively crude. A circle primitive would be helpful. Improved erase modes are also desirable. Also, provision for the incorporation of color images would increase the communicative capabilities of the dynamic display. Many commercial "paint" programs could provide this capability.

Currently, limited macro, function and image libraries assist the modeler in the creation of the system model. In addition, only the exponential distribution on the distribution selection panel is implemented. Code implementing a variety of statistical distributions would be both easy and useful.

Finally, the animation facilities of GPVSS are basic. Code exists to animate "concurrent" animation events but is not currently used. A very useful addition to GPVSS would be the introduction of VIS capabilities.

Future improvements to GPVSS are needed. GPVSS is an early prototype; however, GPVSS has proved to be a useful visual modeling system for a significant class of systems.

Future improvements and implementations can gain ever more of the potential for computer assisted modeling offered by VS and VIS.

BIBLIOGRAPHY

Alemparte, M., D. Cheda, D. Seely, and W. Walker (1975), "Interacting With Discrete Simulation Using On Line Graphic Animation," *Comput. & Graphics 1*, 309-318.

Baeker, R.M. (1974), "Genesys Interactive Computer-Mediated Animation", In *Computer Animation*, J. Halas, Ed. New York, N.Y., pp. 97-115.

Balci, O. (1986), "A Conceptual Framework for Simulation Modeling," MDE Project Working Paper, Department of Computer Science, Virginia Tech, Blacksburg, Va.

Balci, O. (1988), "CS 4214 Simulation & Modeling Lecture Notes," Department of Computer Science, Virginia Tech, Blacksburg, Va.

Bell, P.C. (1985a), "Visual Interactive Modeling as an Operations Research Technique," *Interfaces 15*, 4 (July-Aug.), 26-33.

Bell, P.C. (1985b), "Visual Interactive Modeling in Operational Research: Successes and Opportunities," *Journal of the Operational Research Society 36*, 11 (Nov.), 975-982.

Bell, P.C. (1986), "Visual Interactive Modelling in 1986," School of Business Administration, The University of Western Ontario, London, Canada.

Bell, P.C. and P.F. Kirkpatrick (1986), "Questionnaire on the Practice of Visual Interactive Modeling," School of Business Administration, The University of Western Ontario, London, Canada.

Bell, P.C. and R.M. O'Keefe (1987), "Visual Interactive Simulation - History, Recent Developments, and Major Issues," *Simulation 49*, 3 (Sept.), 109-115.

Biles, W.E. and S.T. Wilsor (1987), "Animated Graphics and Computer Simulation," In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta, Ga., Dec. 14-16). IEEE, Piscataway, N.J., pp. 472-476.

Birtwistle, G., J. Joyce, and B. Wyvill (1984), "ANDES - An Environment for Animated Discrete Event Simulation," In *Proceedings of the U.K. Simulation Conference* (Bath, U.K.), Sept.

Cox, S. (1987), "Interactive Graphics in GPSS/PC," *Simulation 49*, 3 (Sept.), 117-122.

Gordon, R.F. and E.A. MacNair (1987), "A Visual Programming Approach to Manufacturing Modeling," In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta, Ga., Dec. 14-16). IEEE, Piscataway, N.J., pp. 465-470.

Gordon, R.F., E.A. MacNair, K.J. Gordon, and J.F. Kurose (1988), "The RESearch Queuing Package Modeling Environment (RESQME)," Research Report RC 13428 (#60092), Research Division, IBM, Yorktown Heights, N.Y., Jan.

Grant, J.W. and S.A. Weiner (1986), "Factors To Consider In Choosing A Graphically Animated Simulation System," *Industrial Engineering*, (Aug.), 37-68.

Grant, M.E. and D. W. Starks (1988), "A tutorial on TESS: The Extended Simulation Support System," In *Proceedings of the 1988 Winter Simulation Conference* (San Diego, Ca., Dec. 12-14). IEEE, Piscataway, N.J., pp. 136-240.

Hollocks, B.W. (1987), "Practical Benefits of Animated Graphics in Simulation," In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta, Ga., Dec. 14-16). IEEE, Piscataway, N.J., pp. 323-328.

Hurrion, R.D. (1980), "An Interactive Visual Simulation System for Industrial Management," *European Journal of the Operational Research Society 5*, 2 (Aug.), 86-93.

Hurrion, R.D. (1986), "Visual Interactive Modeling," *European Journal of the Operational Research Society 23*, 3 (Mar.), 281-287.

Hurrion, R.D. and R.J. Secker (1978), "Visual Interactive Simulation: An Aid to Decision Making," *Omega 6*, 5, 419-426.

Johnson, M.E. and J.P. Poorte (1988), "A Hierarchical Approach to Computer Animation in Simulation Modeling," *Simulation 50*, 2 (Jan.), 30-36.

Kurose, J.F., K.J. Gordon, R.F. Gordon, E.A. MacNair and P.D. Welch (1986), "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queuing Package (RESQ)," In *1986 Proceedings of Fall Joint Computer Conference* (Nov.). ACM, New York, N.Y., pp. 719-728.

Macintosh, J.B., R. W. Hawkins, and C. J. Shepard (1984), "Simulation on Microcomputers: The Development of a Visual Interactive Modelling Philosophy," In *Proceedings of the 1984 Winter Simulation Conference* (Dallas, Tex., Nov. 28-30). IEEE, Piscataway, N.J., pp. 531-537.

Mathewson, S.C. (1985), "Simulation Program Generators: Code and Animation on a P.C.," *Journal of the Operational Research Society 36*, 7 (June), 583-589.

Melamed, B. and R.J.T. Morris (1985), "Visual Simulation: The Performance Analysis Workstation," *IEEE Transactions on Computers 18*, 2 (Aug.), 87-94.

Miles, T., R.P. Sadowski, and B.W. Werner (1988), "Animation with CINEMA," In *Proceedings of the 1988 Winter Simulation Conference* (San Diego, Ca., Dec. 12-14). IEEE, Piscataway, N.J., pp. 136-240.

Nance, R.E. (1989), Personal Communication, Department of Computer Science, Virginia Tech, Blacksburg, Va.

O'Keefe, R.M. (1987), "What is Visual Interactive Simulation? (And is There a Methodology for Doing it Right?)," In *Proceedings of the 1987 Winter Simulation Conference* (Atlanta, Ga., Dec. 14-16). IEEE, Piscataway, N.J., pp. 461-464.

Palme, J. (1977), "Moving Pictures Show Simulation to User," *Simulation 29*, 6 (Dec.), 204-209.

Sauer, C.H., E.A. MacNair, and J.F. Kurose (1982), "The Research Queuing Package Version 2: Introduction and Examples," Research Report RA 138 (#41126), Research Division, IBM, Yorktown Heights, N.Y., Apr.

Smith, R.L. and L. Platt (1987), "Benefits of Animation in the Simulation of a Machining and Assembly Line," *Simulation 48*, 1 (Jan.), 28-30.

Standridge, C. R. (1986), "Animating Simulations Using TESS," *Computers and Industrial Engineering 10*, 2, 121-134.

Sun Microsystems (1986), *SunINGRES*, Volumes I, II, and III, Sun Microsystems, Inc., Mountain View, CA, May.

Sun Microsystems (1988a), *SunView Programmer's Guide*, Sun Microsystems, Inc., Mountain View, CA, May.

Sun Microsystems (1988b), SunView 1 Beginner's Guide, Sun Microsystems, Inc., Mountain View, CA, May.

White, R.A. (1988), "TRAVIS: A General Purpose Traffic Intersection Visual Interactive Simulator," M.S. Thesis, Department of Computer Science, Virginia Tech, Blacksburg, Va., Sept.

Wyvill, B. and B. MacDonald (1987), "State of the Art in Computer Graphics and Animation," In *Proceedings of the Conference on Methodology and Validation* (1987 Eastern Simulation Conferences, Orlando, Fla., Apr. 6-9). SCS, San Diego, Calif., pp. 1-6.

The vita has been removed from the scanned document