TOWARDS A GRAPHICAL DEADLOCK ANALYSIS TOOL

By

BOBBY STEPHEN KOSHY

Bachelor of Technology (Honors)

Regional Engineering College

Calicut, India

1992

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
December 1995

TOWARDS A GRAPHICAL DEADLOCK ANALYSIS TOOL

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

[illegible]

Thomas C. Collins

Dean of the Graduate College

PREFACE

The purpose of this thesis was to implement a graphical tool to model and analyze operating system deadlocks using Process-Resource graphs. The topics covered as background and context for the implementation in this work consist of introductions to: (1) operating system deadlocks, including algorithms for their detection and analysis; (2) graph drawing algorithms, graph editors, and graph browsers; (3) the Sequent S/81 computer system including its architecture and operating systems; (4) the X Window System including its definition, fundamental components, client/server interaction, and software layers; (5) the OSF/Motif toolkit including its architecture, widget set, and programming structure; and (6) the MotifApp application framework.

The programming part of this work consisted of the design and implementation of the modeling and analysis tool referred to as Prograph including its class hierarchy, data structures, widget hierarchy, and interface objects. The Prograph program, coded in the C++ language, has about 15,000 lines of code with 3 major class hierarchies, 56 classes, and 394 member and non-member functions. The Prograph program enables users to model operating system Process-Resource graphs rapidly, analyze the graphs, and then view the different stages of the deadlock analysis. The deadlock representation and analysis tool Prograph was prototypically evaluated by the students in the graduate level

Operating Systems II class as well as a number of graduate students at the Computer Science Department of Oklahoma State University. The feedback obtained from the users of the Prograph program indicated that it was functional and useful for modeling and analyzing Process-Resource graphs.

ACKNOWLEDGMENTS

I wish to express my appreciation and gratitude to my major professor Dr. Mansur H. Samadzadeh. During the course of my graduate studies, his advice, intelligent guidance, assistance, and constructive criticism have been a constant source of inspiration and motivation for me. I also wish to thank Drs. Blayne E. Mayfield and George E. Hedrick for serving on my graduate committee.

Additionally, I wish to thank Dr. Glenn O. Brown at the Department of Biosystems and Agricultural Engineering, Oklahoma State University, for his support through employing me as a Graduate Research Assistant. Also, I appreciate the encouragement given to me by Dr. Marvin Stone of the same department.

Finally, I would like to express my sincere gratitude to my parents, T. K. Koshy and Mary Koshy, for their continued support and encouragement, without which this endeavor would not have been successful.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

One of the tasks of an operating system is to manage the allocation of resources among cooperating and competing processes. An operating system has to manage the "fair" allocation of resources. During this process, unless the dynamic situation is properly managed, there is a possibility of things going awry and resulting in scenarios that can lead to deadlock and starvation. A process is said to be in a state of deadlock if it is waiting for a particular event that is not going to happen. One such possible event might be acquiring control of a certain resource.

An operating system designer needs to consider these aspects, among other things, when building an operating system. A method by which this can be achieved is to build a model of the system before it is actually constructed, and analyze the model for deadlocks. Also, even after the system is constructed, the designer might want to analyze the system dynamically for the presence of deadlocks. One of the means to achieve this is to take occasional snapshots of the state of the running operating system, and analyze the state in order to determine the behavior of the system from the point of view of deadlocks.

Analyzing a system's behavior requires the use of an analysis tool. The tool can construct models of the system states and has the capability to present the states graphically for convenient analysis. In the educational field, such a tool can be useful in helping grasp the concept of deadlocks, as well as their detection and recovery from deadlocks.

The objective of this thesis was to develop a graphical tool to aid in the study and analysis of operating system deadlocks through capturing and representing the associated resource graphs. The tool was implemented using the Motif Toolkit on a Sequent S/81 computer running the DYNIX/ptx operating system. Chapter II of this thesis provides a review of the literature on modeling process-resource graphs and analyzing them for deadlocks. Chapter III provides a discussion on the implementation platform and environment. Chapter IV takes a look at the software architecture and detailed design of the software tool that was developed as part of this thesis. The testing and evaluation of the software tool developed are discussed in Chapter V. This thesis ends with Chapter VI that provides a summary, the conclusions drawn from the study, and some suggestions for future work.

CHAPTER II

LITERATURE REVIEW

2.1 Operating System Deadlocks

According to Isloor and Marsland [Isloor and Marsland 80], deadlocks arise when members of a group of processes which hold resources are blocked indefinitely from access to resources held by other processes within the same group of processes. When no member of the group will relinquish control over its resources until after it has completed its current resource acquisition, deadlock is inevitable and can be broken only by the involvement of some external agent.

A set of processes becomes deadlocked essentially as a consequence of exclusive access and circular wait. The simplest illustration of these conditions involves only two processes, each requesting exclusive access to the resource held by the other. The result is a circular wait which cannot be broken until one of the processes releases its resources or cancels its request.

2.1.1 Conditions for Deadlock

Coffman et al. showed that four conditions must hold for a deadlock to exist [Coffman et al. 71].

1. Mutual Exclusion: Each resource is either currently assigned to exactly one process or is available.

2. Hold and Wait: Processes currently holding resources granted earlier can request new resources.

3. No Preemption: Resources previously granted cannot be forcibly taken away from a process; they must be explicitly released by the process holding them.

4. Circular Wait: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

2.1.2 Deadlock Modeling

Holt showed how these four conditions can be modeled using directed graphs [Holt 72]. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource previously has been requested by, has been granted to, and is currently being held by that process. An arc from a process to a resource means that the process is currently blocked waiting for that resource. Figure 1 illustrates an example.

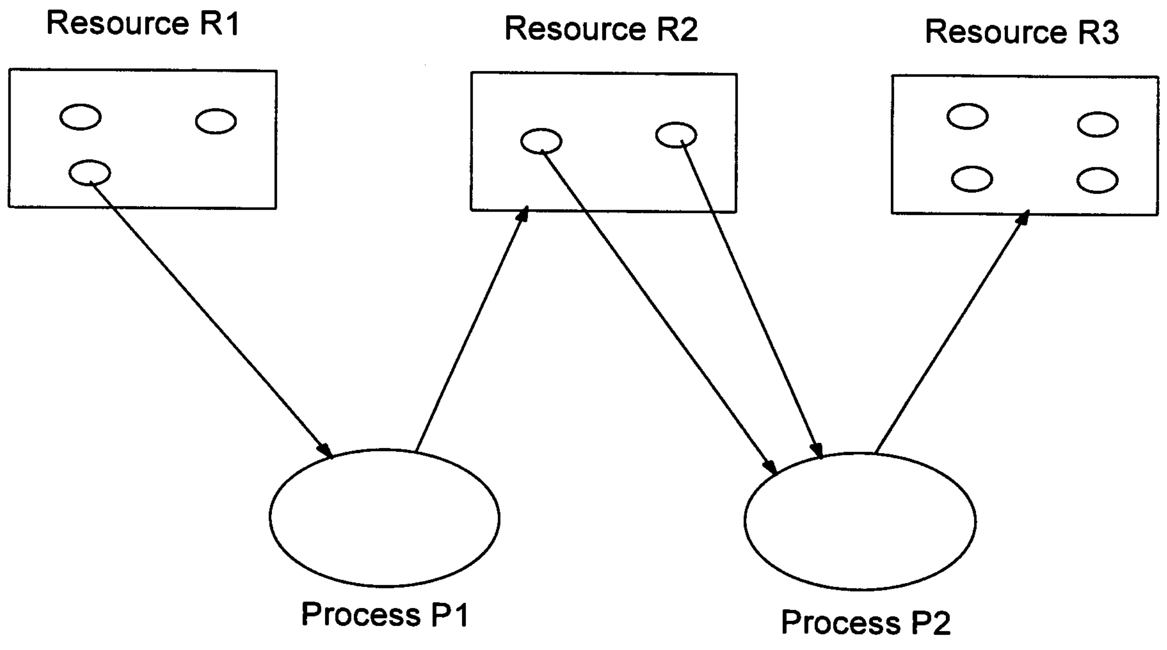


Figure 1. Representing Process-Resource states graphically

2.1.3 Deadlock Detection and Recovery

Since this thesis deals with detecting deadlocks using resource graphs, this section takes a look at the different mechanisms available by which deadlocks can be detected, and how deadlock situations can be removed or recovered from.

2.1.3.1 Detection with One Resource of Each Type

In the case of a system in which only one resource of each type exists, deadlocks can be detected by testing for the existence of cycles in the corresponding resource graph. If the graph contains one or more cycles, deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked. Tanenbaum gives a simple algorithm [Tanenbaum 92] that inspects a graph and terminates either when it has found a cycle or when it has shown that none exists. It uses one data structure called L for a list of nodes. To avoid repeated visiting of the nodes, arcs are marked to indicate that they have already been inspected. The algorithm is given below.

1. For each node N in the graph, perform the following 5 steps with N as the starting node.

2. Initialize L to the empty list and designate all the arcs as unmarked.

3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.

4. From the given node, see it there are any unmarked outgoing arcs. If so, go to Step 5; if not, go to Step 6.

5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to Step 3.

6. We have now reached the end of this path. Go back to the previous node, that is, the node that was current just before this one, make that one as the current node, and go

to Step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

In the worst case this algorithm has complexity O(n!).

2.1.3.2 Detection with Multiple Resources of Each Type

Habermann describes an algorithm [Habermann 69] which handles the case when multiple copies of some of the resources exist. This is a matrix-based algorithm for detecting deadlocks among n processes P_1 through P_n. Let the number of resource classes or types be m, with E_i resources of class i, 1 <= i <= m. E is the existing resource vector. It gives the total number of instances of each resource type.

At any given time, some of the resources are assigned and are not available. Let A be the available resource vector, with A_i giving the number of instances of resource i, 1 <= i <= m, that are currently available (i.e., unassigned). There are two additional arrays: C, the current allocation matrix, and R, the current request matrix. The ith row of C indicates how many instances of each resource class is currently being held by process P_i. Thus C_{ij} is the number of instances of resource j that are held by process i. Similarly, R_{ij} is the number of instances of resource j that process P_i wants.

An important invariant holds for these four arrays. Every resource is either allocated or is available, i.e., if we add up all the instances of resource j that have been allocated, and add to it all the instances of the same resource that are available, the result is the number of instances of that resource class that exist in the system.

The deadlock detection algorithm [Habermann 69] is based on comparing vectors. Let us first define the relation A <= B on two vectors A and B to mean that each component of A is less than or equal to the corresponding element of B. Thus, A <= B holds if and only if $A_i <= B_i$ for $0 <= i <= m$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, the unmarked processes are known to be deadlocked.

The deadlock detection algorithm [Habermann 69] can now be given as follows.

1. Look for an unmarked process P_i, for which the ith row of R is less than A.
2. If such a process if found, add the ith row of C to A, mark the process, and go back to Step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked. The complexity of the deadlock detection algorithm is $O(n^2)$.

2.2 Drawing Directed Graphs

2.2.1 Graph Drawing Algorithms

It is not easy to conceptually grasp the overall structure of a digraph unless vertices are laid out in some regular form (e.g., clustered layout) and unless edges are drawn in such a form that paths can be readily traced visually. A number of researchers have tackled this problem [Warfield 77] [Sugiyama et al. 81] [Tamassia et al. 88] [Gansner et al. 93] and have come up with ways for the aesthetic display of graphs. Some of the aesthetic principles involved in displaying a graph are:

1. Expose the hierarchical structure in the graph. If possible, edges should be aimed in the same general direction. This helps in finding directed paths and highlights source and sink nodes.

2. Try to avoid visual anomalies that do not convey information about the underlying graph. Some examples include edge crossings and sharp bends.

3. As far as possible, an attempt should be made to keep edges short. This makes it easier to find related nodes.

Sugiyama and his colleagues [Sugiyama et al. 81] proposed an algorithm for drawing directed graphs. The algorithm has three basic steps as mentioned below.

1. *Preprocessing*: The first step in the algorithm is to topologically sort the graph with a view towards assigning levels to all nodes. Cycles are eliminated in this step by temporarily reversing edges that cause cycles. To guarantee that each edge spans only one level, dummy nodes are introduced at all intermediate levels for long edges (edges that cross several levels).

2. *Barycentric ordering*: This step attempts to reduces the edge crossings between pairs of nodes on each level. This involves rearranging the nodes at each level. The ordering of the nodes in the top level is fixed. The position of nodes in the next level is determined based upon its barycentre which is the average position of the successors. This downward pass is continued and then a similar upward pass is done. This iteration is done a number of times, until no more improvement in the edge crossings is obtained.

3. *Fine tuning*: This step is responsible for fixing the final x and y coordinates for each node. The y coordinate of each node is the product of the vertical spacing factor and the level number of each node. Determining the x coordinates involves a number of downward-upward iterations. The x coordinates of the nodes in the top level are set initially, so that they are spread apart by the horizontal spacing factor. Then the nodes in the next level are placed based upon the following order. Dummy nodes are placed first followed by nodes with the largest number of incoming and outgoing edges. The x coordinate of each node is calculated as the average position of its predecessors and successors. Each node is placed as close as possible to its desired position without shifting previously placed nodes or changing the relative order of the nodes. Since a dummy node will always have exactly one predecessor and successor, it will always request to be placed at the same x coordinate as its predecessor or successor. Dummy nodes are positioned first to align them and this straightens 'long' edges. Finally the original directions of edges, which were temporarily reversed to accommodate cycles, are restored.

A faster algorithm for drawing directed graphs was proposed by Gansner et al. [Gansner et al. 93]. They describe a four-pass algorithm for drawing directed graphs. The first pass finds an optimal rank assignment using a network simplex algorithm. The second pass sets the vertex order within ranks by an iterative heuristic algorithm incorporating a novel weight function and local transpositions to reduce crossings. The third pass finds optimal coordinates for nodes by constructing and ranking an auxiliary graph. The fourth pass makes splines to draw the edges.

2.2.2 Graph Editors and Graph Browsers

A number of papers having been published detailing various editors for displaying and editing graphs. Paulisch and Tichy [Paulisch and Tichy 90] discuss the implementation of EDGE: an extensible editor kernel for the direct and visual manipulation of graphs. According to the authors, EDGE was designed to solve a number of potential problems faced by any graph editor. Two such problems are mentioned below.

1. Automatic Graph Layout: They discuss how their implementation can integrate application-specific layout requirements, individual preferences, and layout stability with automatic layout algorithms.

2. Graph Abstraction: How can users deal with large graphs containing hundreds of nodes and edges, and thousands of edge crossings? EDGE solves this by using subgraph abstractions and a clustering technique developed by the authors called *edge concentration.*

Vlissides and Linton [Vlissides and Linton 90] discuss a framework called Unidraw for creating graphical editors in various domains such as technical and artistic drawing, music composition, and circuit design. The Unidraw architecture simplifies the construction of these editors by providing programming abstractions that are common across domains. Other research efforts on directed graph editors include the work done by Gansner et al. [Gansner et al. 88] and by Rowe et al. [Rowe et al. 87].

CHAPTER III

IMPLEMENTATION ISSUES

3.1 Implementation Platform

3.1.1 Sequent Symmetry S/81

The Sequent Symmetry S/81, Sequent Computer System, Inc. is a mainframe class computer system with a multiprocessor architecture. The multiprocessing and shared memory architecture consists of the following elements [Sequent 90]:

- A parallel architecture that utilizes multiple industry-standard microprocessors.
- Either the DYNIX v3.0 operating system or the DYNIX/ptx operating system, both UNIX system ports.
- A standard set of network interfaces such as Ethernet, SCSI, VMEbus, and MULTIBUS.

The Sequent Symmetry S/81's operating system has been engineered to incorporate features that support the underlying parallel architecture. In addition to this, software that has been written for the UNIX operating system can run on the Sequent Symmetry S/81 with little or no modification. In the case of multi-user applications, the operating system of the Sequent Symmetry S/81 automatically distributes the tasks to multiple processors in an attempt to reduce response time and increase system throughput [Sequent 90].

The DYNIX v3.0 operating system supports the two major command sets of UNIX, namely, the Berkeley UNIX and UNIX System V. On the other hand, the DYNIX/ptx operating system is compatible with AT&T System V v3.2 only [Sequent 90].

3.2 Implementation Environment

3.2.1 X Window System

The graphical user interface (GUI) for the deadlock analysis tool developed for this research was implemented on top of the X Window System software environment. X supports a device independent graphics system that permits software developers to engineer portable GUIs [Young 90]. The only requirement for complete portability is that the X protocol should be supported by the hardware platforms to which the software is to be ported. The interaction between a client and server is defined by the X protocol. The X Window System follows a client-server architecture. An application acts as a client and the responsibility for all input and output devices is with the server [Young 90].

An application interacts with the X Window System by means of the X library. One of the libraries, which provide access to the device independent graphics of X and interface routines through C language functions, is Xlib [Johnson 90] [Barkakati 91] [Keller 90]. Since Xlib is very low level, programmers often have to deal with a lot of complexity in writing programs using Xlib. Hence programs are written at a layer above the Xlib library. These are the Toolkit libraries. Toolkits are easier to use than Xlib for developing GUIs. The standard toolkit for the X Window System is the X Toolkit. It consists of two modules: Xt Intrinsics, which is the layer that directly interacts with the X Window System, and the widgets, which are a set of user interface building blocks [Johnson 90]. Many popular widget sets are supported by Xt Intrinsics. The Motif widget set by the Open Software Foundation (OSF) is one of the more popular widget sets. The Motif widget set supplies GUI components such as windows, menus, buttons, scroll bars, icons, and bitmaps. The relationship among the various layers in the X Window System is shown in Figure 2.

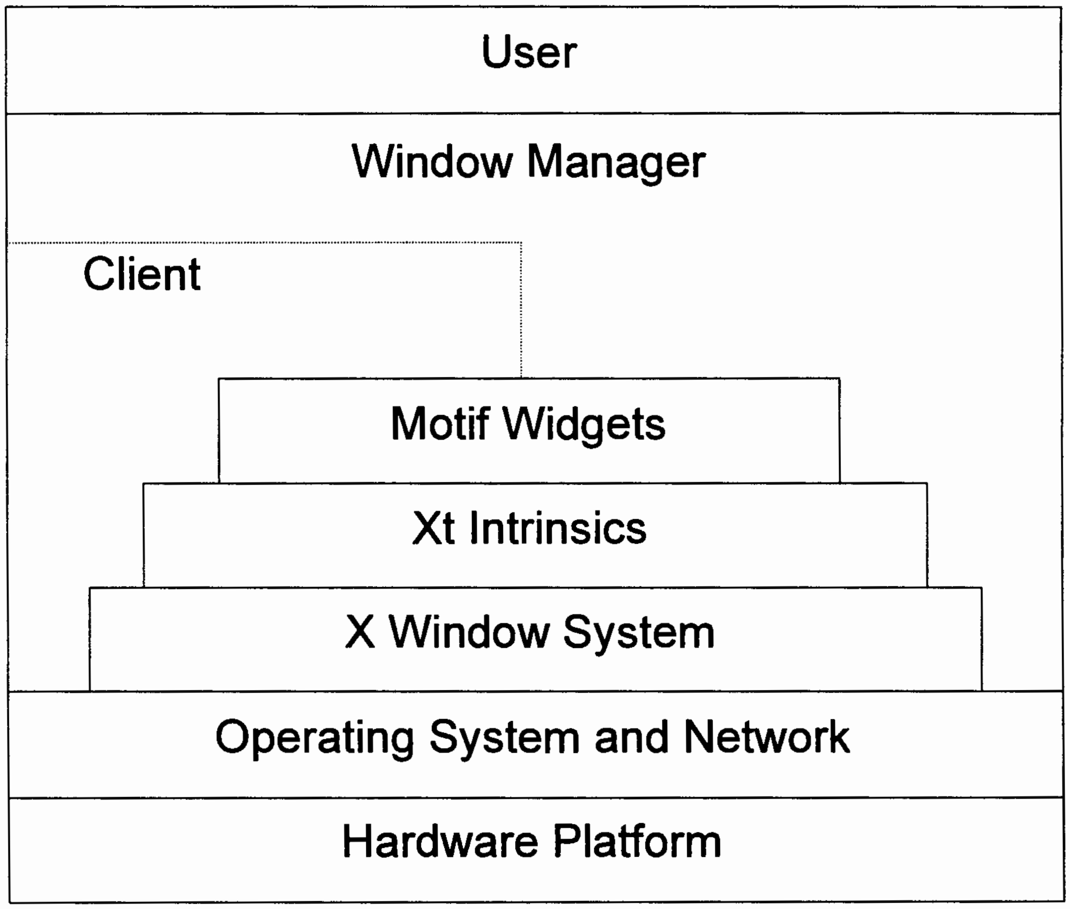


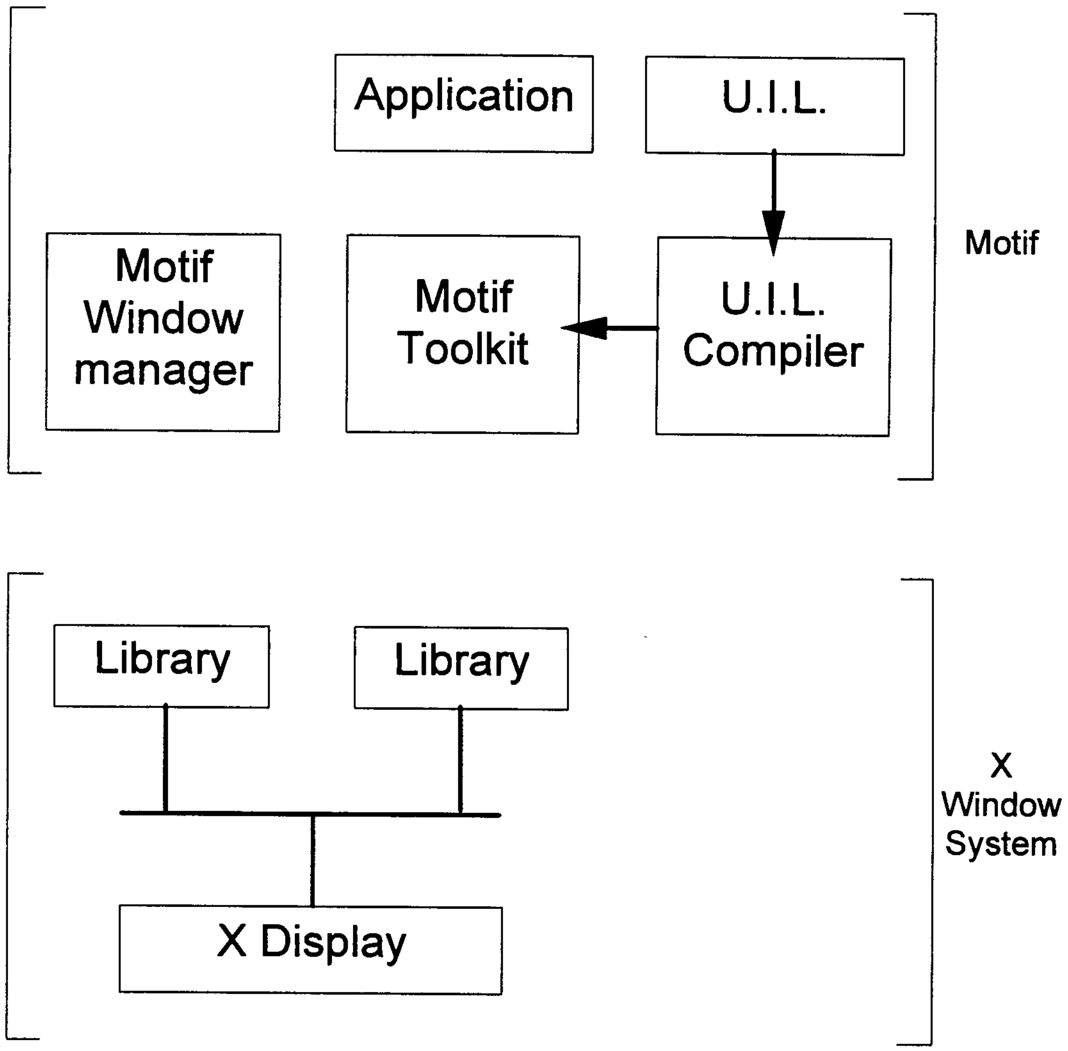
Figure 2. Layers in the X Window System (Source: [Barkakati 91])

3.2.2 OSF/Motif Toolkit

The OSF/Motif Toolkit is a set of functions and procedures that provides quick and easy access to the lower layers of the X Window System. The OSF/Motif Toolkit was designed by the Open Software Foundation (OSF) and is based on the X Toolkit Intrinsics (Xt). OSF/Motif provides a number of user-interface objects known as widgets. These widgets are accessed and manipulated through the various functions and procedures provided by the OSF/Motif library. OSF/Motif is a specification rather than an implementation, making it entirely implementation independent [Heller 91].

The complete architecture of the OSF/Motif Toolkit consists of a number of important modules that are shown in Figure 3 [Berlage 91]. The hardware independent nature of the X Window System is due to the fact that it is built at the lowermost layer of the application hierarchy and hence applications are shielded from any system-dependent input/output variations. The primary window functions such as resizing, closing, moving and iconizing are managed by the Motif Window Manager (MWM). In achieving its tasks, the Motif Window Manager follows the Inter-Client Communication Conventions (ICCC) that enable it to manage X applications developed using different toolkits [Berlage 91]. The Motif Window manager is also responsible for other functions such as maintaining the stacking order of the overlapping windows and controlling the input focus to determine which application window should receive input [Berlage 91].

A Motif user interface can also be developed by using a specification language called the User Interface Language (UIL). Once the specifications of the required user interface are written, the UIL compiler translates the presentation details and loads it into memory at run time. Using the UIL language approach is just another alternative to designing a Motif user interface, in other words it is not necessary for a Motif application to use it [Berlage 91].



Legend

U.I.L.: User Interface Language

Figure 3. Architecture of OSF/Motif (Source: [Berlage 91])

The most important part of the OSF/Motif architecture, as shown in Figure 3, is the Motif Toolkit which provides the set of user interface widgets. The common user-interface objects such as push buttons, menus, labels, dialog boxes, scroll bars, and text entry or display areas are provided by the Motif Toolkit. In addition, there are widgets, known as manager widgets, that perform the function of controlling the layout of the other user interface widgets. A widget operates, to a large extent, independently of the application. A widget's actions are determined by the events dispatched to it by the Xt Intrinsics. A push button, for example, knows how to draw itself, how to highlight itself, and how to respond to a mouse click (or any other user-defined action) by executing an application procedure [Nye 90] [Heller 91].

A class inheritance hierarchy of the Motif widget set is shown in Figure 4. In that inheritance hierarchy, some of the classes are defined by the Xt Intrinsics. These include some of the base classes. Their behavior can be inherited by widgets that are derived directly from the base classes defined in Xt Intrinsics. The Xt Intrinsics classes also provide a common behavior for all widget classes based upon them.

The Core widget class of Xt Intrinsics is the root of the widget class hierarchy. It is the super-class for all widget classes derived from it, and it provides a set of common resources such as size and position which are inherited by all the other classes.

The highest level Motif widget class is the Primitive widget class. It is derived from the Core widget class. Besides inheriting some of its resources from the Core widget class, it also adds some of its own resources such as control of the three-dimensional shadows. The Label widget class is derived from the Primitive widget class. It inherits some resources from the Primitive widget class and adds some features of its own. These include the ability to display a pixmap or a string of text, as well as mechanisms for positioning of a string and displaying the string in a variety of fonts. The sub-classes of the Label widget class are the Push Button class, the Drawn Button class, the Toggle Button class, and the Cascade Button class. These classes, besides inheriting the features of the Label widget class, add features that are necessary to support their unique behavior.

The Composite widget class is again an Xt Intrinsics widget class that is sub-classed from the Core widget class. It adds features that provide it with the capability to manage its geometry. The Constraint widget class is also an Xt Intrinsics widget class that is derived from the Composite widget class. It supports the facility to manage the position and size of the widget. The Manger widget class is provided by Motif and is sub-classed from the Constraint widget class. The Manager widget class is the super-class for all the widgets that manage the geometry of their children such as the Row Column widget class, the Drawing Area widget class, the Scale widget class and the Bulletin Board widget class [Nye 90].

The Shell widget class is yet another Xt Intrinsics widget class, which is a sub-class of the Composite widget class. Shell widgets provide an interface between the window manager and other widgets. The responsibilities of the Shell widget class include handling the window manager protocol for the application and setting the resources required by the window manager. Since the function of interacting with the window manager is very complex, a number of different Shell widget classes are provided. The Override Shell widget class is derived from the Shell widget class and provides a

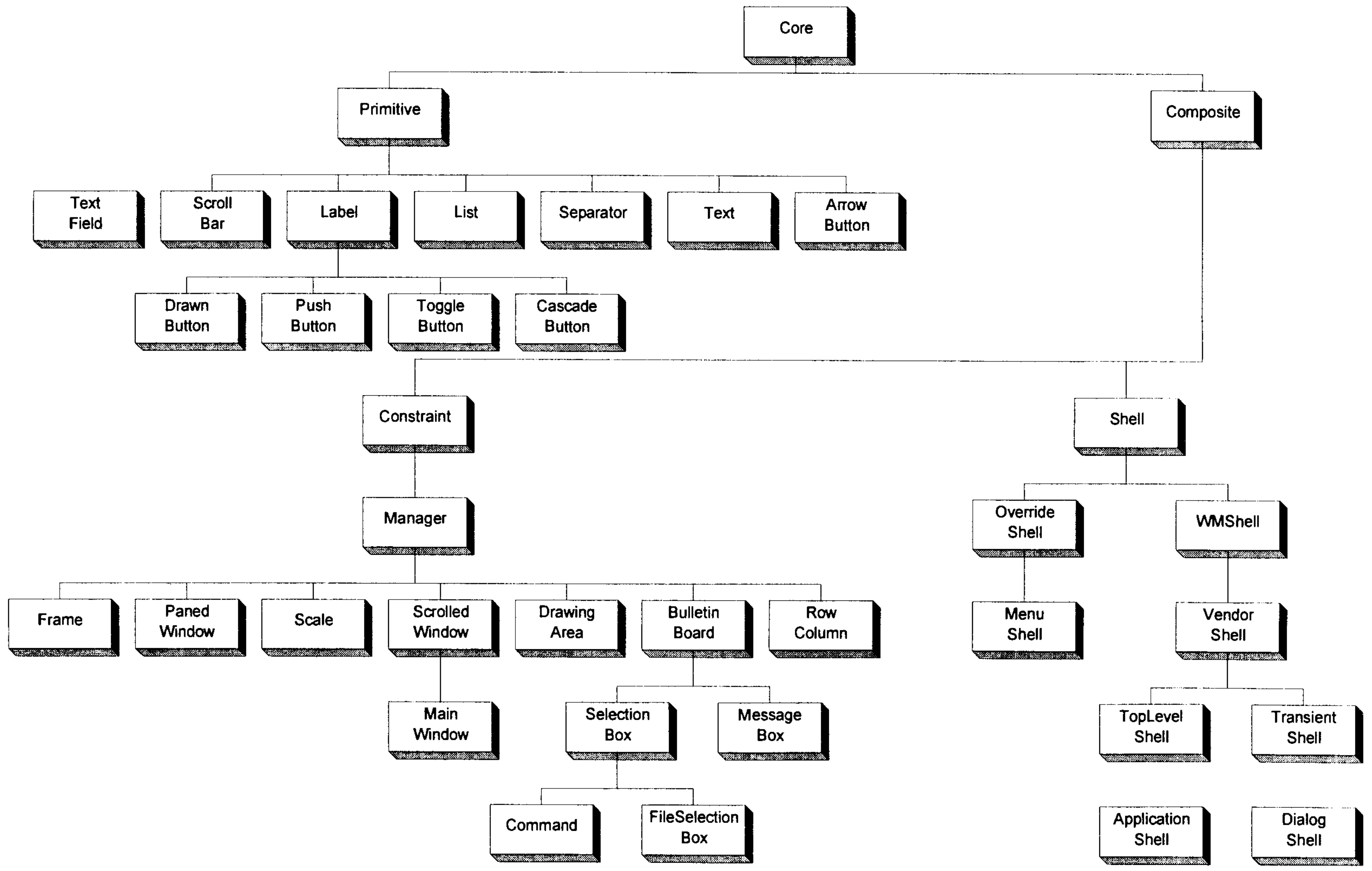


Figure 4. A class inheritance hierarchy for the Motif widget set (Source: [Nye 90])

temporary window that completely bypasses interaction with the window manager. The Menu Shell widget class is derived from the Override widget class and was introduced by Motif to handle the special interface requirements of the OSF/Motif architecture. The WMShell widget class is sub-classed from the Shell widget class. It is a set of simple wire bed-frame widgets that have no special attributes. The Vendor Shell widget class, which is sub-classed from the WMShell widget class, provided features that vendors of window managers can use to define their own attributes that are specific to their own window managers. The Transient Shell widget class is used by Xt Intrinsics to create dialog boxes. The Transient Shell widgets may not be iconified separately by the window manager. They behave in such a way that if an application is iconified, all the child widgets of the application that belong to the Transient Shell widget class are automatically iconified by the window manager. The Dialog Shell widget class is sub-classed from the Transient Shell widget class and was created by OSF/Motif. The functions of the TopLevel Shell widget class and the Application Shell widget class provide various applications with their top-level windows [Nye 90] [Young 90] [Heller 91].

3.2.3 MotifApp Application Framework

There are two main approaches to reducing the amount of code a programmer must write to complete an application.The first and the traditional approach is providing collections of functions or classes that implement common components needed by many programs. The Motif System is an example of a toolkit based upon this approach. When writing programs in Motif, programmers can choose from a collection of off-the-shelf user interface components. Without the toolkit, programmers would have to develop buttons, scrollbars, and so on for each new application. The Toolkit approach is very effective and is a widely-used form of reusing software.

Another approach to code reduction in writing programs is to concentrate less on the individual components needed by various applications, and instead focus on the structure and control flow within a particular type of application. In this approach the programmer does not need to define the architecture of each new application anew. This approach is used by application frameworks and provides a way to capture the organizational characteristics common to many applications. [Young 92].

Similar to a toolkit, an application framework is a library that provides various components needed by a number of programs. Moreover, an application framework also defines most of the connections between these components, and also defines the overall control structure of applications built on the framework. Most application frameworks provide an Application class, which captures the essential behavior of all applications built from the framework. Programmers write new applications by deriving a new subclass of Application that handles application-specific details.

When using an application framework, there is the concept of a generic application. The generic application is the simplest possible program that can be written using the framework, and can be usually written by creating an instance of the Application class, or by declaring and instantiating a trivial subclass. The generic application, though not serving any useful purpose, follows all rules and conventions supported by the framework. The important point to note is that the generic application defines the flow of control used by all similar applications. The application framework has the responsibility to connect the various components of a program, thus relieving the programmer of this responsibility. In creating a new application, only those parts of the program that are unique need to be implemented. The application-specific behavior of a new program can be provided by adding a few new components, defining a few methods, or by deriving new classes from those provided by the framework. So, in other words, writing an application using a framework is similar to deriving a new class from an existing class. In both instances, a new entity is created by specifying only how the new entity is different from the existing one [Young 92].

The MotifApp framework, as described by Young [Young 92], describes a simple application framework that encapsulates a structure that can be useful to applications based on X and Motif. This framework captures many characteristics of typical Motif applications. The MotifApp framework does not capture all the elements common to all Motif applications. Instead, it implements a basic architecture which can be enhanced and expanded with additional classes to form a more powerful framework, if desired.

CHAPTER IV

SOFTWARE ARCHITECTURE AND DETAILED DESIGN

In this chapter, the specification, architecture, and user interface of the software tool Prograph is described. The evaluation of the Prograph deadlock representation and analysis tool is discussed in the next chapter.

4.1 Description

The deadlock analysis tool Prograph enables the user to draw a Process-Resource graph using the tool, and analyze the graph for the presence of deadlocks, safe states, unsafe states, etc. A Process-Resource graph consists of a set of process nodes, a set of resource nodes, and the allocation/request/producer edges joining them. The Prograph software tool provides support for drawing these graph entities. The tool's display is divided into two main parts: a scrollable drawing area, which is used to draw the graph, and a palette of buttons depicting the various graph entities together with the various actions that can be applied to those entities.

A specific graph node can be drawn by selecting the appropriate button from the palette and clicking the mouse on the drawing area. Edges are drawn by selecting the Edge button from the palette and going to the drawing area and connecting a process node to a resource node, or vice versa. The connection is drawn by clicking on the node from which the edge is to be started and terminating the edge by clicking again on the destination node. The tool also allows a user to move a node about the drawing area.

The palette also provides various buttons for editing a graph. This includes deleting a graph entity, editing the specifications of a graph entity, etc. The specifications of a graph entity include its name, the number of resources (if it is a resource node), etc. The palette also contains an undo option which enables the user to undo the previously executed operation.

Once a graph is drawn on the screen, the Prograph software tool can analyze the graph for the presence of deadlocks, safe states, unsafe states, etc. Other features of the tool include saving a graph and other standard file manipulation operations.

4.2 User Interface

The user interface of Prograph is shown in Figure 5. The user interface is divided into three main regions.

- Region 1 of Prograph's interface is the canvas on which a user draws a model. The drawing area is a scrollable viewport into a larger virtual drawing space. The scrollbars can be used to pan the other parts of the drawing area that are not visible on the physical drawing surface. The drawing area also has a title bar at the top displaying the current filename.

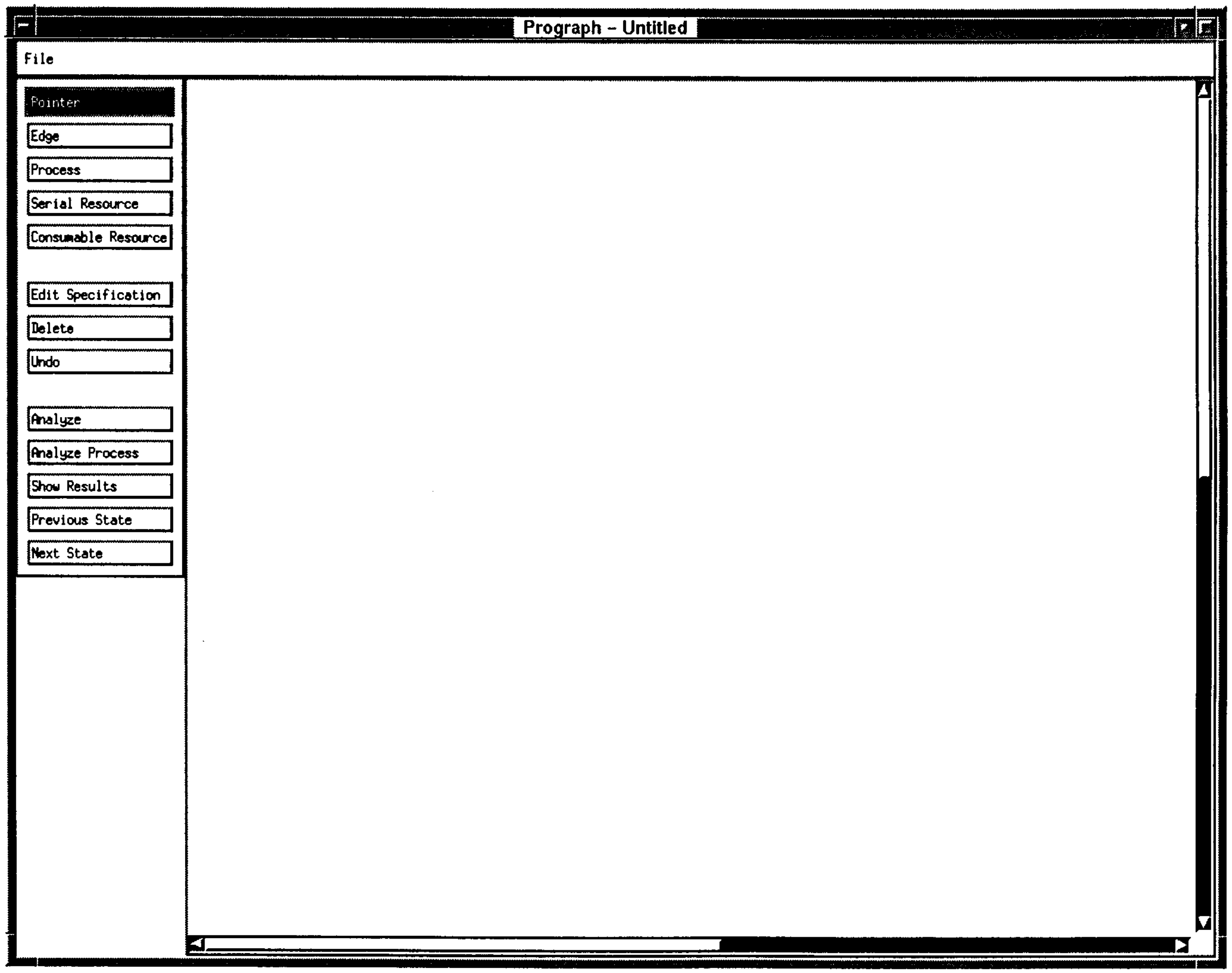


Figure 5. The Prograph User Interface

- Region 2 of Prograph's interface is the Palette containing a number of buttons to set the operating mode of Prograph and also to perform various actions. The buttons for mode setting are "Pointer", "Process", "Serial Resource", "Consumable Resource", and "Edge". The other items in the Palette for performing various actions are: "Edit Specification", "Delete", and "Undo". There are a number of additional buttons for the purposes of analysis. These include: "Analyze", "Analyze Process", "Show Results", "Previous State", and "Next State".
- Region 3 of Prograph's interface is the menu bar. The menu bar has only one item: the "File" Menu. The "File" menu contains the menu items - "Open", "Close", "Save", "SaveAs", and "Exit".

4.3 Software Architecture

Prograph is designed based upon the model-view-controller paradigm [Barkakati 93]. In this paradigm, the users of a program interact with the various controllers. The controllers in turn send messages to the model. The model is the central part of the program. It usually represents the main data structures and the associated code that enable the program to deal with its problem domain. Any changes to the model are displayed in the multiple views. A generic architecture of a model-view-controller based program is shown in Figure 6.

The software architecture of Prograph is based upon the model-view-controller paradigm. Prograph was designed using an object-oriented approach and was implemented in the C++ language [Stroustrup 91].

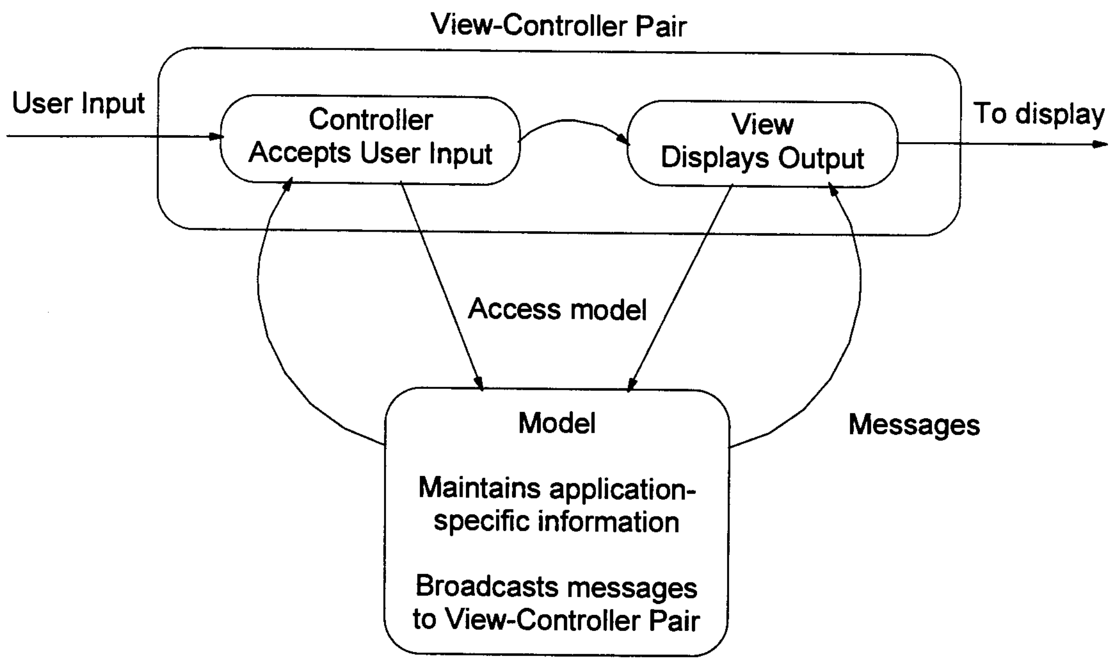


Figure 6. Model-View-Controller architecture (Source [Barkakati 93])

As depicted in Figure 7, Prograph consists of the following architectural elements and assumptions.

1. The view and the controller were combined together as a single module in Prograph. This layer deals with handling all mouse and keyboard input and hiding any system dependencies (X-Windows, MS-Windows, etc.). This layer also deals with screen graphics in a system-independent fashion.

2. An additional layer was placed between the controller and the model. This layer consists of a set of Command objects. Command objects encapsulate knowledge of dealing with certain sequences of user actions. In other words, they execute actions (commands) requested by the user (Moving a graph entity, drawing an edge, etc.).

3. The model of Prograph is a module that handles all aspects of internally storing and manipulating a directed graph.

4. The model layer of Prograph internally consists of a set of shape objects corresponding to various graph entities. The various shape objects embody the representation and behavior of the various items found in a directed graph.

5. A module exists to analyze a graph for deadlocks, safe/unsafe states, etc.

6. A set of dialog classes to represent the various kinds of dialogs required by an application.

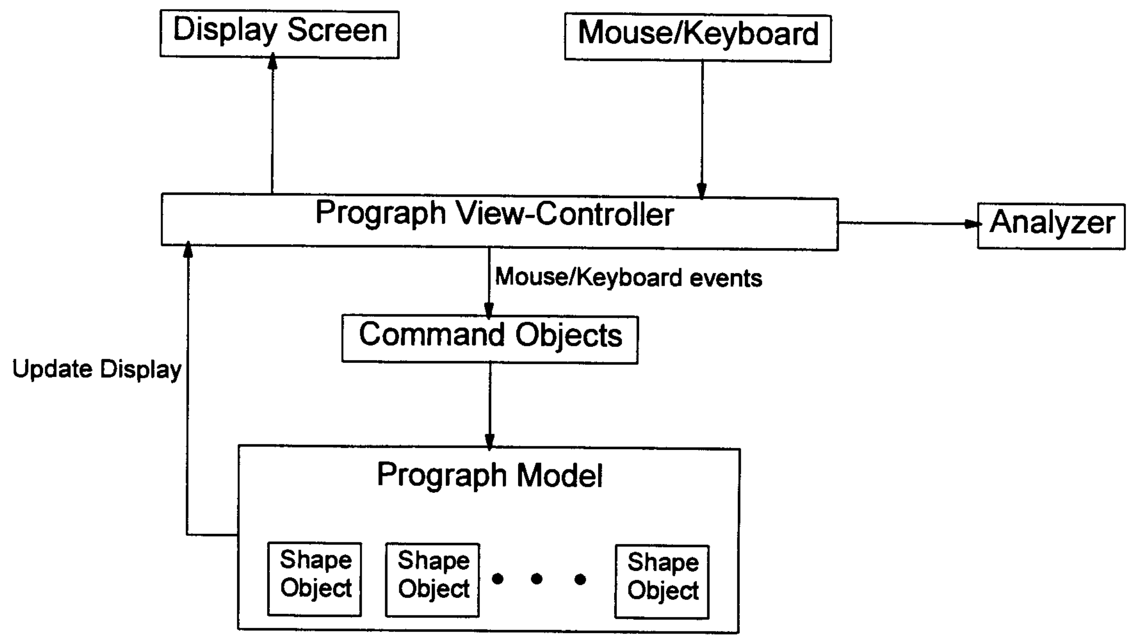


Figure 7. Software Architecture of Prograph

All the major class names in Prograph are prefixed with the letter 'T'. This was done to differentiate the classes in Prograph from the classes in the application framework that was used to build Prograph. The particular choice of the letter 'T' does not have any significance. The following six subsections describe six main classes comprising Prograph.

4.3.1 Prograph View Class

The controller and the view modules in the model-view-controller design are packaged together into one class called TPrographView. This class serves as a wrapper around the system-dependent input/output library calls so as to make the library portable across implementations. As seen in the architecture (Figure 7), events arriving at the TPrographView object (the Prograph View-Controller) are redirected to the Command Objects, which in turn cause changes to the Prograph model. The Prograph model reflects all changes to the TPrographView object.

4.3.2 Shape Classes

The Shape classes handle the display and behavior of the various graph entities (See Figure 8 for an illustration of the Shape class hierarchy). Basically, a Process-Resource graph consists of three different types of graph objects: Process objects, Resource objects, and Edge objects. All these shapes are compound objects, which are in turn composed of simpler shapes. A Process shape is composed of a Circle object and a Text object. The Text object displays the name of the process, and is placed at the center of the Circle. A Resource shape is composed of a Square shape, and three Text objects that represent the type of the Resource, the name of the Resource, and the number of units of the Resource, respectively. An Edge object is composed of a Polyline and an Arrowhead.

The compound shapes (i.e., Process, Resource, and Edge) are classified as Graph shapes since they are shapes occurring as basic graph entities. The Circle, Square, Polyline, and Arrow shapes are classified as Pure shapes. Both Graph shapes and Pure shapes are obviously in the general category of shapes. Thus we have an inheritance hierarchy (Figure 8) with the top class being named as TShape having sub-classes TGraphShape and TPureShape. TGraphShape has the sub-classes TProcess, TResource, and TEdge, while TPureShape has the sub-classes TCircle, TSquare, TPolyline, and TArrow. The TText object is classified as a subclass of TShape.

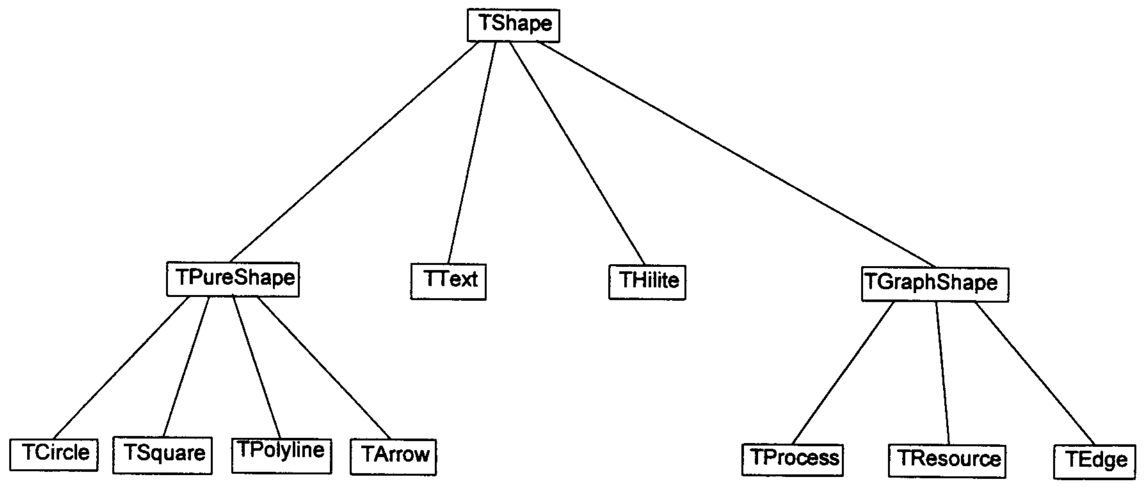


Figure 8. Shape class hierarchy

When an object in the graph is selected, it is distinguished from other objects by the presence of a number of solid squares placed at the boundaries of the object. These set of squares are called the Hilite object and they are represented by the THilite class which is classified as a subclass of TShape.

4.3.3 Command Classes

The Command objects are responsible for responding to the mouse and keyboard input from the user, and manipulating the shape objects accordingly. From the specifications, a number of commands can be readily identified, they include creating a new process or resource, moving an object (process or resource), drawing an edge connecting a process and a resource, deleting an object, etc. Each of these commands consists of a series of mouse events having a specific start event and a set of end events.

Corresponding to each command, we have a Command object. Thus we have the corresponding set of classes TMoveCommand, TNewProcessCommand, TNewResourceCommand, TEdgeCommand, TDeleteCommand, etc. (Figure 9).

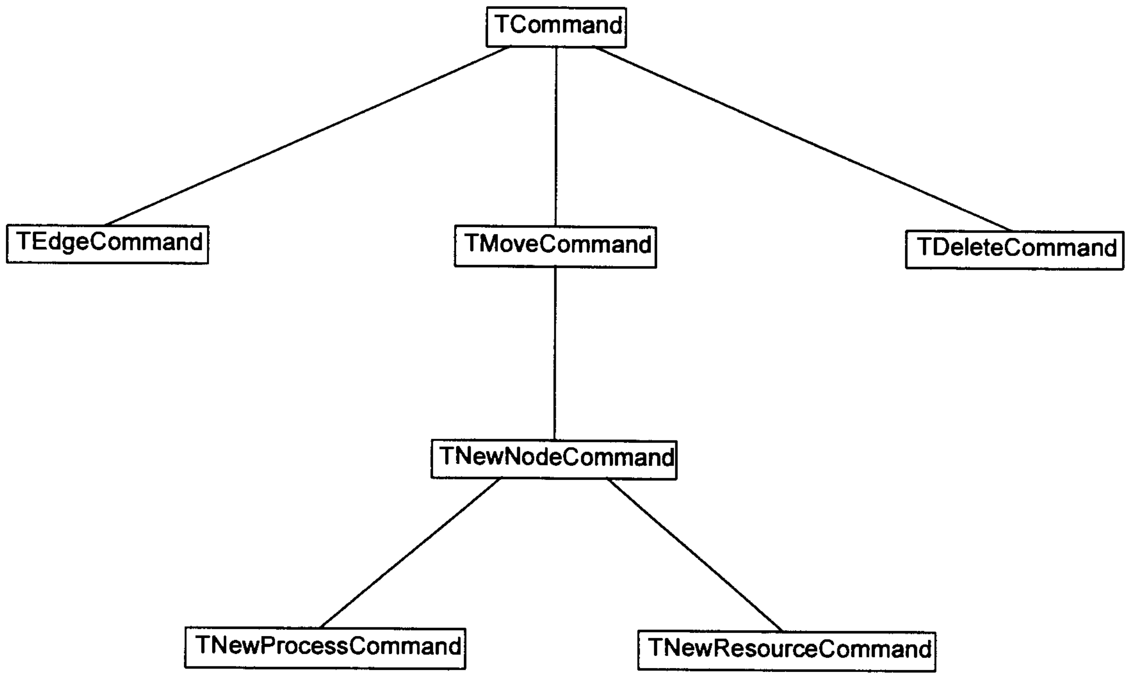


Figure 9. Command class hierarchy

A Command object is created usually on a specific mouse click or a keyboard press (in the case of the delete command). The TPrographView object (responsible for input/output) decides which Command object to create depending upon the mode the program is in. Once a Command object is created, all further input events are sent to that Command object. The Command object performs the required user action. A Command object terminates upon a specific event (e.g., a mouse-up event in the case where the user is moving a shape object). Internally, each Command object behaves as a finite state automaton. Each user input event (mouse-down, mouse-up, mouse-move, key-press, etc.)

results in the Command object's internal automaton making a transition to a different state. A Command object is terminated when a series of user input events results in a final state of the automaton.

The Command class inheritance hierarchy (as depicted in Figure 9) was designed to abstract properties common to all command classes. The class TCommand is the root class of the hierarchy. Its direct descendants are TMoveCommand, TDeleteCommand, and TEdgeCommand. When creating a new node, once the node is created and placed in the viewing area, every successive user event is similar to that encountered when moving an object (a user can create a new process or resource by clicking on the viewing area, and in the same sequence, before releasing the mouse, the user can move the newly-created object to any position on the screen). So the TNewNodeCommand class was made a sub-class of the TMoveCommand class, and the TNewProcessCommand class along with the TNewResourceCommand class were made sub-classes of the TNewNodeCommand class.

4.3.4 Prograph Model Class

In Prograph terminology, any Process-Resource graph is internally represented as what is called a Prograph Model. So, in effect a Prograph model is a set of process objects, resource objects, and edge objects. A Prograph model created by a user can be saved in a file, a previously saved model can be read from a file, and of course a Prograph model can be analyzed.

4.3.5 Prograph Analyzer Class

The analysis functionality of Prograph is encapsulated in the TPrographAnalyzer object. The input to the analysis phase is a Prograph model. The output is a list of Prograph models which is the set of reduction sequences of the input model (each state in the reduction sequence is represented by a separate model in the list.). If the input Prograph model is not reducible, the resulting list will contain just the input model. Besides the reduction sequence, the analyzer also generates a report of the results of the analysis.

4.3.6 Dialog Classes

The dialog objects are wrappers for the various types of dialogs required for Prograph. A number of common elements were observed when constructing the various dialog classes, so most of these elements were abstracted and formed into a hierarchy (Figure 10).

The TDialog class is at the top of the dialog inheritance hierarchy. Two major subcategories of dialogs were identified: Predefined Motif dialogs and user-defined Custom dialogs. This resulted in two sub-classes of the TDialog class: TMotifDialog and TCustomDialog. The dialogs already available in Motif were made as individual classes, and were made sub-classes of the TMotifDialog class: TGetFileDialog, TQuestionDialog, and TGeneralPurposeDialog. The various custom dialogs which were created, were made sub-classes of TCustomDialog. These include the classes TAnalysisResultsDialog and the TProcessTextDialog.

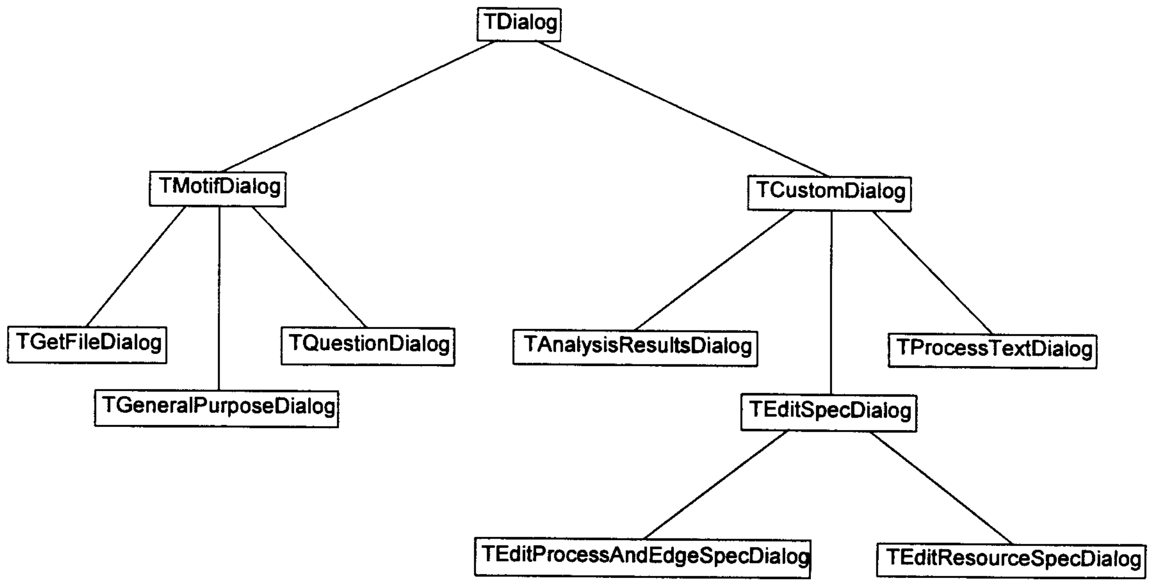


Figure 10. Dialog class hierarchy

The TEditProcessAndEdgeSpecDialog and TEditResourceSpecDialog were created as sub-classes of TEditSpecDialog which in turn is a subclass of TCustomDialog.

4.4 Design of the Prograph View Class

As described earlier, the TPrographView class serves as the input/output layer of Prograph. This class provides a system-independent interface for input and output. This class gets all the mouse and keyboard events and presents them to the other modules in a multiplatform portable format. The TPrographView class also provides a graphical drawing library that is called by the model layer. In the current X-Window implementation of Prograph, the graphical library hides various X Windows dependent information. The TPrographView object is also responsible for creating the various

Command objects in response to various mouse and keyboard events. After each command finishes executing, they are immediately deleted.

4.4.1 Invoking Command objects

The decision as to when to create a Command object and which Command object to create, is made by the TPrographView object depending upon the mode the user is in. There are five different modes: "Pointer" mode, "Edge" mode, "Process" mode, "Serial Resource" mode, and "Consumable Resource" mode. In the "Pointer" mode, the user can select objects on the drawing area and move them around. In the "Process" mode, clicking on the drawing area results in the creation of a new process. In the "Serial Resource" mode or the "Consumable Resource" mode, the user can create a serial resource or a consumable resource, respectively. The "Edge" mode enables the user to draw an edge connecting either a process to a resource or vice versa. The five mode buttons (counting the resource modes as two) are handled by a separate view object called the TPaletteView object. At any given time, Prograph will be in any one of the modes. The current mode will be shown highlighted. Whenever the user changes any of the modes, the TPaletteView object sends a message to the TPrographView object. The TPrographView object has an internal variable storing the mode which is currently active. When Prograph start up, the default mode is the "Pointer" mode.

Consider the case where the program is in the "Pointer" mode. On a mouse click, the TPrographView object first tries to find out if the mouse has hit on any of the shapes on the screen. For this a message is sent to the TPrographModel object requesting a hit check. If the TPrographModel object returns with no hit, the mouse click is ignored and the TPrographView object continues to wait for another click. If a hit has occurred on a shape, then it means that the user wants either to select the shape or move the shape. Both operations are handled by the TMoveCommand object. So a new TMoveCommand object is created and the mouse click (down) event is sent to the TMoveCommand object. All further mouse events are sent to the new Command object. In fact, once a new Command object is created, all mouse and keyboard events are sent to it. A Command object, after it has dealt with all the required mouse events, will specifically inform the TPrographView object (by a return value) whether it has finished its processing. The TMoveCommand object finishes its processing on a mouse-up event. Note that mouse-move events could have occurred in between the starting mouse-down and the mouse-up, but the TPrographView object never has to bother about this, it simply has to route them to the Command object until the Command object indicates it does not want any more mouse events. After the mouse-up event, the TMoveCommand object sends an event complete message (signifying termination of the user action) back to the TPrographView object, and the Command object is destroyed. The TPrographView object goes back and waits for the next mouse down and the whole process is repeated. This goes on till the user exits Prograph.

To be precise the Command object is not destroyed immediately. Instead, it is stored separately as the previous command. This is essential to support undo, since only the specific Command object will have the necessary knowledge to undo what it has done.

If the user is in the "Process" mode (on a mouse-down event anywhere in the drawing area), a new TNewProcessCommand object is created. The TNewProcessCommand object will create a new process shape. The same sequence of steps as above is followed. In this case also, the Command object terminates on the next mouse-up event. Likewise, if the user is in either the "Serial Resource" mode or the "Consumable Resource" mode, a new TNewResourceCommand object is created. This will result in a new resource shape.

As compared to the other modes, event handling in the case of the "Edge" mode is more complicated. A new TEdgeCommand object is created only if a mouse-down occurs with the mouse being within a process or a resource. As before, the TPrographView object has to request the TPrographModel object to check for a hit. Unlike the previous Command objects, a TEdgeCommand object need not terminate on a mouse-up event. This is because the user can draw a multi-line (also called a polyline) edge, which will have multiple end points. This will involve a lot of mouse clicks within the drawing area. However, this is the responsibility of the TEdgeCommand object and the TPrographView object need not concern itself with the different possibilities. All that the TPrographView object has to do, is redirect all further mouse and keyboard events to the TEdgeCommand object. When the user finishes drawing the edge (after the user clicks on a destination shape), the TEdgeCommand object will return an event complete message back to the TPrographView object.

4.4.2 Hit Testing a Line

Normally, hit testing is done by the TPrographModel which sends the coordinates where the mouse hits the drawing area to each of the individual shapes that make up the graph. In the case of a process or a resource, which are depicted as a circle or a square, respectively, hit testing is simply done by checking whether the mouse coordinates fall within the inscribing rectangle of the object. This check can be done by a simple coordinate comparison.

However, in the case of a line (or a polyline), this is not possible. Another approach to hit testing is required. The hit testing algorithm used in Prograph is by DiLascia [DiLascia 92]. When the user clicks on a shape, a tolerance of a couple of pixels on either side is given. The mouse click need only be a couple of pixels (usually 2 or 3) near the shape on each side. The main resource used by this algorithm is a simple drawable pixmap of size twice the tolerance on each side. In our present implementation, the algorithm uses a 5x5 pixmap. The algorithm for hit testing is as follows.

1. Translate the line to a coordinate system with the mouse click spot as the origin.
2. Draw the line with the new coordinate system on the pixmap with a new width equaling twice the number of tolerance pixels.
3. Check if any of the pixels in the pixmap is set. If so, the mouse click has hit the polyline. If no pixels are set, then there is no hit. The significance of testing the pixmap is that, if there is a hit, part of the wider line will pass through the pixmap, since the pixmap can be considered to be a magnified view of the area surrounding the mouse click spot. Obviously, the thicker the line, the more tolerance is obtained.

4.5 Design of Shape Classes

4.5.1 TShape class

The TShape class is an abstract class that embodies behavior common to all shapes. If one examines the functionality of each shape, a couple of things immediately come to mind.

1. Each shape should be able to draw and erase itself. Thus the draw function is implemented as a pure virtual function to be handled by the appropriate actual shape subclass (say the TCircle shape class).

2. A shape should be able to determine whether a mouse click has hit on it. This is called the hit test. Hit testing is quite easy for rectangular shapes (just check whether the mouse coordinates lie inside the shape region) and a generic hit testing member function is implemented in the abstract TShape class. However, in the case of a TPolyline shape, checking for a hit is more complicated and hence the hit test function is set as a simple virtual function in case a sub-class down the inheritance hierarchy wishes to override the default hit test.

3. A shape should be able to determine the extent of the area occupied by itself on the viewing surface. This is used by the hit test member function, as well as by the edge drawing command.

4.5.2 TPureShape class

The TPureShape class is an abstract sub-class of the TShape class. The TPureShape class abstracts behavior generic to the "Pure Shapes": Circle, Square, Polyline, and Arrow. A pure shape is the defining shape of all the compound shapes (otherwise called Graph Shapes). For instance, the Circle is the defining shape of the Process shape, the Square is the defining shape of the Resource shape, and the Polyline is the defining shape of an Edge shape.

A common operation in Prograph is dragging a shape on the drawing area. When a shape (specifically a Graph Shape) is dragged, an outline of the shape follows the mouse. The outline is also referred to as the ghost of the shape. Once the outline is dragged and placed at a specific point, the original shape is erased from its previous position and placed at the new position. The outline shape for all the Graph Shapes is their defining Pure Shape. So, in order to support the dragging operations, all Pure Shapes should be able to change their line drawing styles (from solid to dash-dot-dash). To support this, the TPureShape class has member functions to change the line drawing style. The outline dragging of a shape is accomplished by the use of the XOR drawing mode, where the outline is successively XOR'ed to follow the mouse, thus resulting in a drag effect. So a mode setting member function is available in the TPureShape class. The mode of a Pure Shape is set depending upon whether the shape is being dragged or simply is being drawn in solid mode. A translate member function is also needed as part of the TPureShape class. The translate member function is responsible for translating the coordinates of the shape to follow the mouse coordinates.

Another important operation supported by the TPureShape class is the concept of cloning. When an outline shape is created for dragging, it is actually a copy (clone) of the original Pure Shape member of the to-be-dragged Graph Shape. Cloning an outline shape is referred to as cloning a ghost shape. The other type of cloning is the normal cloning or copy of a Pure Shape. A normal clone is required when a copy of an entire graph model is to be generated during the analysis of a graph. The graph clone is used as one of the successive states in a reduction sequence.

4.5.3 TGraphShape Class

The TGraphShape class, which is a sub-class of TShape, is an abstract super-class of the compound shapes TProcess, TResource, and TEdge. One of the elements common to all graph shapes is their name. The name member of a graph shape is implemented as an instance of the TText object. The TGraphShape class supports member functions to set or retrieve a Graph Shape's name. Each type of Graph Shape is distinguished by its distinctive Pure Shape main components. The TProcess graph shape has the TCircle Pure Shape as its main component. The TResource graph shape has a TSquare Pure Shape main component. Likewise, the TEdge graph shape has a TPolyline Pure Shape as its main component. Hence the TGraphShape class has a member variable to hold the appropriate Pure Shape component. This member variable is called the main shape variable.

Earlier it was explained that, when a Graph Shape is being dragged, it is actually a clone (a ghost clone) of its main component that is being dragged. At the end of the dragging, the Graph Shape is redrawn at the new position. This is actually accomplished by cloning a normal version of the ghost clone and replacing the original main component of the Graph Shape by the latest clone. Thus, a replace main shape member function is also an integral part of the TGraphShape class. When this replacement is done, the position of the Graph Shape is automatically updated to its new position, since the new clone would already be at that new position. The TGraphShape class also supports a cloning operation of its own. This cloning is used in the analysis procedure to generate copies of the Graph Shape for duplicating an entire graph.

The TGraphShape class supports a hit test member function which is the actual hit test called when a user clicks on a shape in the drawing area. A Graph Shape does a hit test by simply asking its Pure Shape main component to perform the hit checking, since the Pure Shape is the dominating component of a Graph Shape. This might not sound like a complete hit test since, in the case of a TEdge object, we will be performing the hit test only on the TPolyline component and not on the TArrow component. The hit test however, is complete for a TProcess object or a TResource object, since the Pure Shape main component completely encloses the Graph Shape's other components.

4.5.4 THilite Class

A hilite is the set of solid squares that appear at the boundaries of a shape when it is selected (usually by clicking the mouse on a shape). The THilite object is composed of a set of TSquare objects. Every time the user creates a new shape or selects a shape, a new THilite object is created (the previous one, if any, is destroyed). A new THilite object on creation asks the shape to be hilited, for a set of boundary points on which hilites should appear. This function is implemented by the TPureShape class and its sub-classes. A generic "get hilite positions" function is implemented in the TPureShape class. This function is used by the TCircle and TSquare sub-classes, and returns eight boundary points namely the northwest, north, northeast, west, east, southwest, south, and southeast corners of the Pure shape. This function is a virtual function and is overridden in the TPolyline class, since a Polyline has no concept of specific corners. The TPolyline class returns the end points of all its lines as the set of hilite positions.

A THilite object checks for a mouse click hit on itself, by interrogating each of its component TSquare objects for hit testing.

4.5.5 TText class

The TText class is responsible for representing all the different sorts of labels that appear on different Graph Shapes. These labels include the name of the Graph Shapes, the number of units of the TResource shape, and the type of the TResource shape. These labels appear at the top of the respective instances for each type of shape. In the current implementation of Prograph, a TText object can appear only in one of three different positions on a Graph Shape: at the center line of the shape, at the midpoint of the upper half of the shape, or at the midpoint of the lower half. The center position is used as the location for the name of the shape. The upper midpoint is used for the position of the type of a TResource object, and the lower midpoint is used as the position for the number of units of a TResource object.

4.6 Design of Command Classes

The Command objects are responsible for actually responding to user input events and performing the appropriate action. As explained earlier, the Command objects are created and destroyed by the input/output layer (the TPrographView object). A Command object is created in response to a specific user action, say dragging an object. Once a Command object is created, it has total control of the mouse and keyboard. After performing its job, the Command object decides to terminate upon a predetermined input event (say a mouse-up event). The TPrographView object creates a Command object on each user action and, once the action is complete, the Command object is deleted. Some of the common user actions are moving a Shape Object, drawing an edge, connecting a process object and a resource object, deleting an object, etc.

The decision of what Command object to create and when to create it are made by the TPrographView object. This decision depends upon various factors including the drawing mode which the user is in (e.g., "Pointer" mode, "Edge" mode, etc.). As mentioned above, once a Command object is created, only the Command object can decide when it is time for the user action to end and return mouse and keyboard control back to the TPrographView object.

4.6.1 Event Handling Using Finite State Automata

Once a Command object starts up, it will be getting a continuos stream of different input events such as mouse-down events, mouse-move events, and mouse-up events. On each input event, the Command object has to perform some action. The action to be performed depends upon the set of all previous events received. In other words, a Command object has to change its state upon receipt of an input event. This behavior is ideally modeled by means of a finite state automaton. Each input event causes the Command object to change its state. The state change is dependent upon the previous state and the current input event. A number of different finite state automata were designed for the various kinds of user actions. A finite state automaton is implemented in actual practice using a transition table. Each element of the table has three entries. One entry for the current state, one entry for an input event, and one entry for the next state.

All Command objects on creation start in a special state called the Start State. Upon receiving an event, a dispatch routine looks up the next state from the transition table and executes code corresponding to that state. Since the dispatch routine is generic to all Command objects, the routine is placed in the root class of the hierarchy TCommand.

To get a feel of how an actual Command works using a finite state automaton, a simplified example of a subset of the TMoveCommand object is explained here. The TMoveCommand object is created by the TPrographView object when a user starts moving a shape object. In this example, assume for simplicity that the user can drag the shape only within the boundaries of the drawing area. Three different input events are possible. The user initiates the move with a mouse-down event. Then successive mouse-move events are generated when the user drags the shape object. Finally, the user places the shape object at a specific position by releasing the mouse, resulting in a mouse-up event being sent to the TMoveCommand object. In this example, the transition table for the TMoveCommand object is quite simple and an outline is given in TABLE I.

TABLE I. SAMPLE TRANSITION TABLE

Current State	Input Event	Next State
Start state	Mouse-Down	Mouse-Down state
Mouse-Down State	Mouse-Move	Mouse-Move state
Mouse-Move state	Mouse-Move	Mouse-Move State
Mouse-Move State	Mouse-Up	Mouse-Up State

A better picture can be obtained by a transition graph representation of the table. Once the dispatch function finds out the next state, it calls a routine to execute the code for the next state. In the case of the TMoveCommand object, the code for the "mouse-down state" will create a ghost of the to-be-dragged shape and sets up various internal variables to prepare for dragging. On each successive mouse-move event, the code for the mouse-move state is executed. This code is responsible for generating the drag effect, i.e., erasing the previous outline of the ghost (by an XOR draw) and drawing a new outline at the new mouse position. Finally when the mouse-up event occurs, the code for the mouse-up state erases the last drawn outline ghost, and replaces the main component of the original shape with a normal clone of the ghost, resulting in the position of the original shape to be updated to the new position.

The actual implementation is far more complicated than the above explanation. In the middle of a Command, the mouse can leave the drawing area resulting in mouse-enter and mouse-leave events. Mouse-down and mouse-up events have to be distinguished as to whether they are coming from inside or outside the drawing area. During a mouse-move, the mouse buttons can be either depressed or undepressed leading to different kinds of mouse-move events and coupled with mouse-moves outside and inside complexity is increased further.

4.6.2 AutoScrolling

A significant feature implemented as part of Prograph is a concept called AutoScrolling. Typically, autoscrolling is useful when drawing an edge between a process and a resource which are far apart, such that they are not visible on the screen at the same time. With autoscrolling, after the user clicks on the source node and drags the mouse towards the destination node, if the mouse leaves the boundary of the drawing area, the drawing area will scroll towards the direction of the mouse movement. Hence the source node will scroll off the screen and the destination node will scroll onto the screen, enabling the user to click the destination node and thus ending the edge drawing. Autoscrolling is not only useful for drawing edges, it is also useful in case a user wants to move a shape to a part of the drawing area not currently visible on the screen. Without autoscrolling, the user would have to do it in a series of steps, at each step using the vertical or horizontal scroll bars. In autoscrolling, the screen will start to scroll automatically immediately after the mouse leaves the drawing area. The scrolling will continue till the end of the virtual drawing area is reached. When the screen is being autoscrolled, the object being moved (an edge or a node shape), will appear under the mouse. If the mouse is off the edge of the drawing area, then obviously the object may not be visible. For scrolling to work, it is simply enough that the mouse be outside the drawing area boundary, the mouse need not be physically moving.

Since the mouse need not be moving for autoscrolling to work, there had to be a way for the program to periodically receive events once the mouse was outside the drawing area. This problem was solved by making use of the timer facility in X Windows. Autoscrolling was implemented as follows.

1. When the mouse leaves the boundary of the drawing area, a mouse-leave event is received. On receipt of this event, the Command object starts a timer to send timing signals at regular intervals.

2. For each timer signal received, the Command object scrolls the screen towards the direction the mouse is currently at.

3. If the mouse happens to also move outside the drawing area boundary, then the Command objects will receive additional mouse-move events. These events are processed the same way as if the object is inside the drawing area.

4. When the mouse reenters the drawing area, the Command object will receive a mouse-enter event. The event handler for the mouse-enter event will stop the timer thus ending the scrolling.

4.6.3 TNewNodeCommand Class and Its Subclasses

The TNewNodeCommand class abstracts behavior pertaining to creating a new process or resource shape on the screen. To create a new shape, the user selects the appropriate palette button and moves the mouse onto the drawing area and clicks, resulting in the appearance of the specified shape. The shape will actually appear on the drawing area during the mouse-down event. In the same action and before releasing the mouse button, the user can move the newly-created shape around on the screen and place it at an appropriate place by releasing the mouse button (thus resulting in the final mouse-up event to the Command object).

From the above description of the creation of a new node (shape), the only difference from the TMoveCommand object is in the reaction to the initial mouse-down event. While the TMoveCommand object hilited the clicked-upon-object on a mouse-down, the TNewNodeCommand object created a new object on a mouse-down. The remaining part of the automaton is the same for both classes. Hence the same automaton could be used for both classes. So it was decided to make the TNewNodeCommand class as a subclass of the TMoveCommand class. The mouse-down event handling member function is overridden by the TNewNodeCommand class.

The TNewNodeCommand class is only an abstract class. The actual Command objects that create the shapes on the drawing area are instances of either the TNewProcessCommand class or the TNewResourceCommand class. Both these classes are sub-classes of the TNewNodeCommand class. These new classes only needed to provide a new constructor. In the constructor for each class, the appropriate shape (process or resource) is created. After that, the remaining events are handled appropriately by member functions higher up the hierarchy (i.e., member functions in TNewNodeCommand and TMoveCommand).

4.6.4 Aborting a Command

A command can be aborted at any time by using the "Esc" key. When the Esc key is pressed, a special event handler is called. This event handler is responsible for reversing any action done up to that point by the command. The abort event handler has to be implemented separately by each command class.

4.6.5 Implementing Undo

The undo facility enables the user to reverse the effect of the previous user action. Prograph currently supports one level of undo. When a user does an undo, the system reverts back to the state it was in, prior to the previous command. If the user does an undo again, the system reverses the effect of the undo. So the undo can also act as sort of a toggle between two consecutive system states. In the current implementation, the second undo is called a "reverse undo".

To support an undo, some changes had to be made in the Command object creation and deletion layer. Without undo, once a Command object finishes, it was immediately deleted. For undo support, a Command object on completion is not immediately deleted, instead it is saved in a previous-command variable. The previous-command variable is updated each time a new command terminates execution. Each Command object has an undo member function. When the user does an undo, the undo member function of the previous Command object is called. If the user successively does undo operations (for whatever reason!), the undo member function in turn maintains a Boolean variable that calls the reverse undo function each alternate time it is called. Implementing the undo member function for a Command object is dependent upon what that Command object does.

Consider the case of a TMoveCommand object. This command moves an object from one position to another. As explained in the section on the Shape hierarchy (Section 4.5), an object is moved from one place to another by replacing its main shape component (a TPureShape) by a new TPureShape (specifically a TCircle or a TSquare) positioned at the destination position. To support undo, the TMoveCommand object, before replacing the main shape, saves the old main shape internally. When the user does an undo, the TMoveCommand object simply replaces the new main shape with the saved old main shape resulting in the effect of the shape jumping back to its old position. During the undo, the TMoveCommand object saves the new main shape internally, so that when the user does an undo again, the old main shape can be replaced again by the new main shape.

In the case of the TNewNodeCommand object (specifically, a TNewProcessCommand or a TNewResourceCommand), the Command object saves a pointer to the newly-created object internally. When the user does an undo, the Command object asks the TPrographModel object (which represents the graph structure) to remove the newly-created node. For a reverse undo, the TNewNodeCommand object adds the removed new node back into the TPrographModel object. The same logic applies to the TEdgeCommand object, which is responsible for drawing an edge.

The TDeleteCommand object works in the reverse way when compared to the TNewNodeCommand and TEdgeCommand objects. When a shape is deleted, the TDeleteCommand object saves the deleted object internally. When the user does an undo, the deleted object is added back into the TPrographModel object. For another undo, the object is again removed from the TPrographModel.

4.7 Design of Prograph Model Class

The TPrographModel class is responsible for the internal representation of a Process-Resource graph. The class has member functions for adding a new node to the graph, removing a node from the graph, and adding and removing edges. The class is also responsible for reading and writing an instance of itself from a file. The class also has support for generating unique names for processes, resources, and edges.

A digraph is normally represented by a matrix. The value of the (i, j)th entry of a matrix denotes the number of edges from node i to node j in the graph, where each node has been assigned a specific number. A Process-Resource graph is a special form of a digraph (a bi-partite graph), in which edges are permitted only from a process to a resource or from a resource to a process. This fact leads to a design where the process entries are considered to be along the rows of the matrix and the resource entries are considered to be along the columns of the matrix. An element of the matrix, say (i, j), is implemented as a list whose members are the set of edges between process i and resource j. In fact the matrix is implemented by the TEdgeListMatrix class. This class is designed to encapsulate all operations pertaining to the matrix. Each element of the matrix (i.e., a list) is implemented as a list class, in this case the TGraphShapeList class., which, as the name implies, implements a list consisting of Graph Shape objects. Also, all the processes in the graph were put on a separate list, likewise for the resources. The edges in the graph, besides being a member of a matrix element, were also put on a duplicate list, for ease of access for certain operations. Despite the extra storage, there is no change in the order of the space complexity.

When a new node is added to the graph, it has to be added either to the process list or to the resource list, depending upon whether it is a process or a resources. Also, when the node is added, either the number of rows or the number of columns of the matrix has to be increased. So the TPrographModel object sends a request to the TEdgeListMatrix object to add an additional row or column to the matrix. All the elements (which are lists) of the additional row or column are set to empty lists. Obviously some particular situations have to taken are of, such as when there are only processes in the graph and no resources, and vice versa. In such cases, the internal matrix is not created by the TEdgeListMatrix object. It comes into existence only when there is at least one process or one resource.

When a node has to be removed from the graph, all the edges associated with that node also have to be removed. These edges are not immediately deleted, instead they are put on a separate list and returned to the Command object that does the deletion (the TDeleteCommand object). The TDeleteCommand object will store this list in case the user decides to do an undo. To remove a node, it is first removed from either the process list or the resource list. Then the row (or column) in the matrix associated with that node is removed. This operation is done by the TEdgeListMatrix object. Finally, the removed node is returned back to the TDeleteCommand object for possible undo later on. The actual deletion of the memory occupied by the node and the corresponding edges is done in the destructor of the TDeleteCommand object after the user executes a new command, and the previous TDeleteCommand object is no longer needed. So in effect the actual node removal is a staggered removal.

When an edge is added to the graph, it is added to the appropriate matrix entry corresponding to the source and destination nodes of the edge. The edge shape is also added to the separate list of edges. The removal of an edge is the exact reverse process and the edge is returned to the TDeleteCommand object for supporting undo. One important fact to be taken care of when adding an edge is whether the edge is permitted to be added. This occurs if for example an assignment edge is drawn from a resource having no more resource units. So, before the edge drawing is finalized, the TPrographModel object asks the source node for permission. If permission is denied, the TPrographModel object aborts the operation and returns, and in turn the TEdgeCommand object, which is responsible for drawing the edge, also aborts. Similarly, when removing an edge, the

TProgaphModel object sends the message to the source node, which might want to increase its number of resource units.

After a user moves a node from one position to another, all the edges associated with that node have to be redrawn to reflect the new position of the node. This is initiated by the TMoveCommand object. After a move, the Command object sends a replace node main shape request to the TPrographModel object. After the main shape of the moved node is replaced (thus resulting in the movement of the node), all the edges associated with that node are found, and the requisite end point of each of those edges is changed to reflect the new node position. Finally, the whole graph is updated.

The TPrographModel object also stores the current hilite object. Whenever a new shape is created, or clicked upon, or moved, that shape has to be hilited and the previous hilited shape, if any, has to be unhilited. All such hilite requests are handled by the TPrographModel object. This object is responsible for deleting the old hilite object and creating a new hilite object for the requested shape.

For generating unique names, the TPrographModel object traverses the current list of nodes and edges, and assigns a name with the next free sequential number with the appropriate string appended. Process names start with the letter "p", and resource names and edge names start with "r" and "e", respectively. When a user clicks on the drawing area, the input/output layer (which is the TPrographView class) requests the TPrographModel object to do a hit test on all the nodes and edges on the graph. The TPrographModel object in turn traverses its internal node and edge lists and asks the node or edge to perform the requested hit test.

4.8 Design of Prograph Analyzer Class

The TPrographAnalyzer class is responsible for analyzing a model and generating a list of models (if the state of the original model is safe) or the single original model (if the state of the original model is unsafe). Prograph supports two types of analyses.

1. A general analysis that tries to find out if there is any way of reducing the state represented by the model.

2. A process-specific analysis that is almost the same as the first type except that the analysis starts with a specific process.

After the analysis, the analyzer returns a list of models as explained above, as well as a small report containing the results of the analysis. The report is immediately displayed in a dialog box on the screen.

4.8.1 Main Algorithm

The analysis is done using an exhaustive search strategy, as described below.

1. Initialize all the internal variables and matrices required for the analysis.

2. Assign each process (in the set of processes of the graph to be analyzed) a sequential number. Generate all possible permutations of processes. E.g., if there are 3 processes - 1, 2, and 3, then the permutations generated are 1-2-3, 1-3-2, 2-1-3, 2-3-1, 3-1-2, and 3-2-1.

3. Analyze each permutation. Specifically, check to see if reducing the initial state by the process list specified by a permutation results in a state where all processes in that combination are reducible.

4. If a permutation that can be reduced is found, generate a new state diagram (model) for reduction by each process. Also generate a report of the results of the analysis.

5. Finally, return back the list of models as well as the report.

The complexity of this algorithm is O(n!). This can be improved by the use of heuristics to speed up the generation of the permutations, and can be done as part of future work.

For analysis based on a specific process, the only difference from the above scheme is in the generation of the permutations. For instance, if we want to reduce by process 2, the permutations that are analyzed are 2-1-3 and 2-3-1.

To analyze a permutation, each process in that permutation, is taken and the state of the model is reduced by that process, if possible. If the whole permutation is reducible, the main algorithm stops and the list of models is generated. If not, the algorithm proceeds to the next permutation until all permutations are analyzed.

4.8.2 Reducing a State by a Process

Before explaining the algorithm, data structures used by the analyzer are described. Three matrices and two vectors are required by the analyzer. The matrices are the request matrix, the allocation matrix, and the producer matrix. Each matrix entry denotes a connection between a process and a resource. Processes are considered to be along the rows of the matrix and resources along the columns. The two vectors are the available resource vector and the total resource vector.

The request matrix stores information about the requests that each process has made for each of the resources. A specific row of the request matrix represents the request vector of a particular process, and the value of the (i, j)th entry in the matrix represents the number of requests process i has made for resource j. Similarly, the allocation matrix stores information about the number of resources of each resource type allocated to each process. The (i, j)th entry of the allocation matrix represents the number of units of resource j assigned to process i. The producer matrix is a binary matrix that serves to denote which of the processes are producers of which of the resources (consumable resources to be specific). An entry (i, j) in the producer matrix denotes whether a process i is the producer of a resource j.

The available resource vector stores the number of units of all resources that are free. The total resource vector stores the total number of units of each resource. All the matrices and vectors are initialized before the start of the analysis. The algorithm to reduce a state by a particular process is given below.

1. Use the request matrix and check if the request row vector of that process is less than the available resource vector. If so, continue, else return.

2. Now we have to take steps to change the system state to reflect reduction by that process. Add the allocation vector of the process (from the allocation matrix) to the available resources vector. Check if the process is a producer of any of the resources. If it is, set the number of available units of that resource to infinity (i.e., w).

This algorithm mainly performs a set of vector comparisons and additions, and hence the time complexity of the algorithm is O(n). The space complexity of the algorithm is O(mn), where m is the number of processes and n is the number of resources.

Duplicate copies of all the three matrices and two vectors are kept separately, since after each combination is analyzed, these matrices and vectors would have been modified. Before the analysis of the next permutation, the original state has to be restored.

4.8.3 Generating a New Intermediate Model

Once a combination is found to be reducible, the analyzer restores the original state and repeats the whole reduction process again; this time, after reducing by each process, an intermediate graph (model) is generated. Generating a new graph involves duplicating the entire graph, leaving out edges associated with processes that have been reduced. If there are m processes, n resources and p edges, then the complexity of duplicating the process and resource nodes in the graph is O(m + n). The complexity of finding edges that are to be copied to the new graph is O(mp).

The intermediate states' graphs differ from the pre-reduction graph only in the number of edges and the number of resource units of each resource. The positioning and the number of all the other processes and resources remain the same. In generating a new graph, all the nodes in the previous graph are cloned and added to the new graph. Now we have to add only the edges that were not connected to any of the processes freed in this stage or any of the preceding stages. For example, assume the permutation to be reduced is 2-1-3 and we have already generated the graph state for reduction by process 2, and we are now reducing by process 1. In this case, the freed processes are 2 and 1, so the edges connected to processes 2 and 1 are not added to the new intermediate graph.

4.9 Prograph Module Listing

The program listing of Prograph consists of 20 C++ source files and 22 header files. The 20 source modules are listed below.

- ProgrAphApp.C: This module is the startup module. The module is responsible for instantiating sub-classes of the MotifApp application framework's Application and MainWindow classes.

- PrographWindow.C: This module contains code to initialize and setup the main window of Prograph. It also creates the palette, the drawing area, and the menu items.

- PaletteView.C: This source code file contains code to setup the different buttons found on the palette of Prograph. The palette is initialized to the "Pointer" button.

- PrographView.C: This source file implements the "View" member functions of the TPrographView object. Besides that, this file also contains member functions to initialize the drawing area.

- ViewMain.C: This source file implements the "Controller" member functions of the TPrographView object. All the Command objects are created and handled in this file.

- Shape.C: This module implements the member functions of the TShape class, as well as the THilite and TText classes.

- PureShape.C: This module implements the member functions of the TPureShape class and its sub-classes namely, TCircle, TSquare, TPolyline, and TArrow.

- GraphShape.C: This module implements the member functions of the TGraphShape class and its sub-classes namely, TProcess, TResource, and TEdge.

- Command.C: This module implements the member functions of the TCommand class.

- MoveCommand.C: This module implements the member functions of the TMoveCommand class.

- NewNodeCommand.C: This module implements the member functions of the TNewNodeCommand class, the TNewProcessCommand class and the TNewResourceCommand class.

- EdgeCommand.C: This module implements the member functions of the TEdgeCommand class.

- PrographModel.C: The TPrographModel object's member functions are implemented in this class.

- PrographAnalyzer.C: This module is the analysis module and is responsible for implementing the member functions of the TPrographAnalyzer class.

- Document.C: The TDocument object's member functions are implemented in this class.
- Dialog.C: This module implements the member functions of the TDialog class, which is the root class of the Dialog class hierarchy.
- MotifDialog.C: This module implements the member functions of the TMotifDialog class and its sub-classes namely, TGetFileDialog, TGeneralPurposeDialog, and TQuestionDialog.
- CustomDialog.C: This module implements the member functions of the TCustomDialog class and that of two of its sub-classes: TAnalysisResultsDialog and TProcessTextDialog.
- SpecDialog.C: This module implements the member functions of the TEditSpecDialog and its sub-classes TEditResourceSpecDialog and TEditProcessAndEdgeSpecDialog.
- Rectangle.C: This module implements the member functions of the TRectangle class.

CHAPTER V

EVALUATION OF THE TOOL

5.1 User Evaluation

The testing and evaluation of the software tool Prograph, developed as part of this thesis, are discussed in this chapter along with some comments on the drawbacks of this tool and suggestions for its improvement based on the experience gathered in using the tool. About 25 students of the graduate-level Operating Systems II class of Spring 1995 used Prograph for a class assignment to design two Process-Resource graphs and analyze them for the presence of deadlocks.

Useful feedback was received during the evaluation process. The feedback was used to improve the tool. One deficiency of the tool was the lack of an on-line help system. Instead of an on-line help, the users were supplied with a hard copy of the README file for Prograph. It was observed that very few of the participants/users were utilizing the README file, and instead most of them were going ahead and directly experimenting with the tool. Also, it was felt that a live demonstration could prove useful for the effective use of the tool. Another observed problem was that if the tool was run without the presence of a window manager, the title of the current file being manipulated would not be visible. This created some confusion in the minds of the users about where in the program they were, since they could not distinguish between a model they drew and the analysis file generated by Prograph.

Also some users did not effectively use the multiple line edge drawing feature of Prograph. From the answer files to the design assignment problem, and based on observing the peculiar shapes of the edges, it was obvious that some users were unaware of the existence of such a facility. Another problem was that the contents of the file dialog boxes are not updated immediately after the user saves a file. The workaround currently used, is to click on the "Filter" button of the file dialog box, to refresh its contents.

5.2 Sample Systems Modeled by Prograph

As a limited experiment in the usability of Prograph, a number of different Process-Resource graphs were modeled, including the assignment problems given to the students of the graduate-level Operating Systems II course in Spring 1995. One of the systems is described by Nutt [Nutt 92] in Figure 5.8 of his book. The graph is shown in Figure 11 and contains only serial resources. There are three process and three resources in the graph. Figures 12 to 14 depict the various stages in the reduction sequence of the graph. This reduction sequence is obtained as a result of the analysis of the graph.

Another graph modeled by the tool is shown in Figure 15. The graph represents a Process-Resource model having both serial and consumable resources and is described in Figure 5.13 in [Nutt 92].

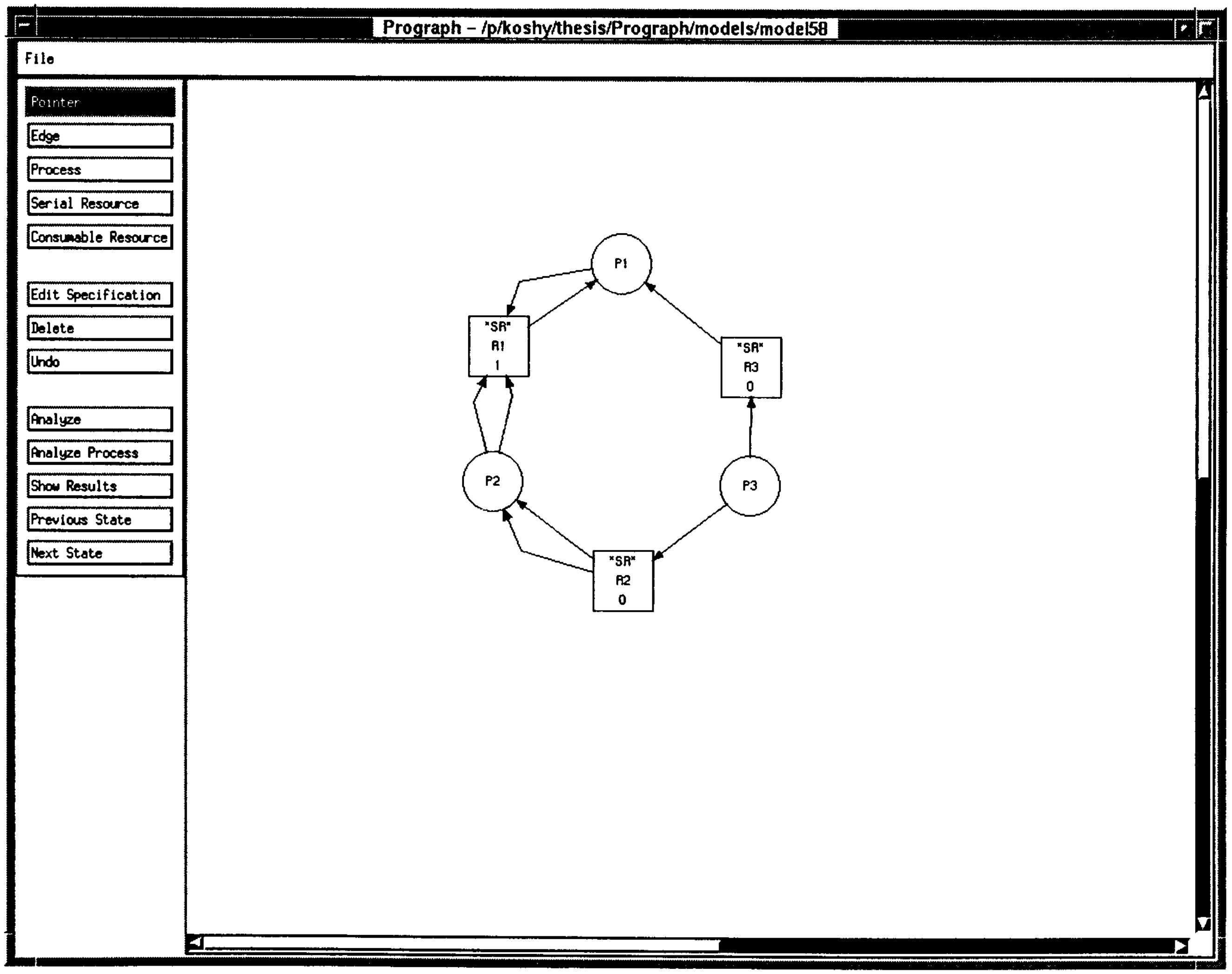


Figure 11. A Sample Process-Resource Graph (Source [Nutt 92], Figure 5.8)

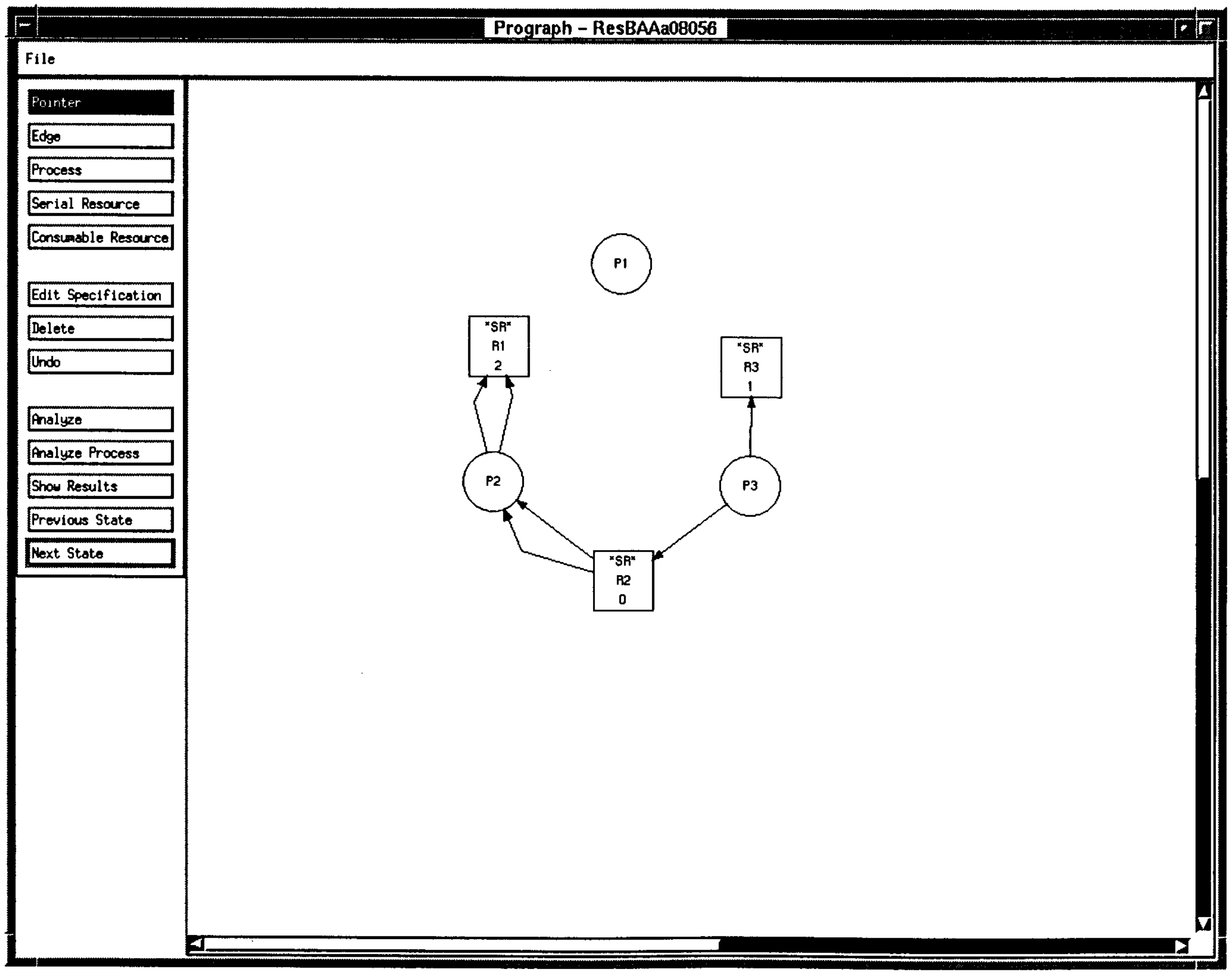


Figure 12. Step 1 in the reduction sequence for Figure 11

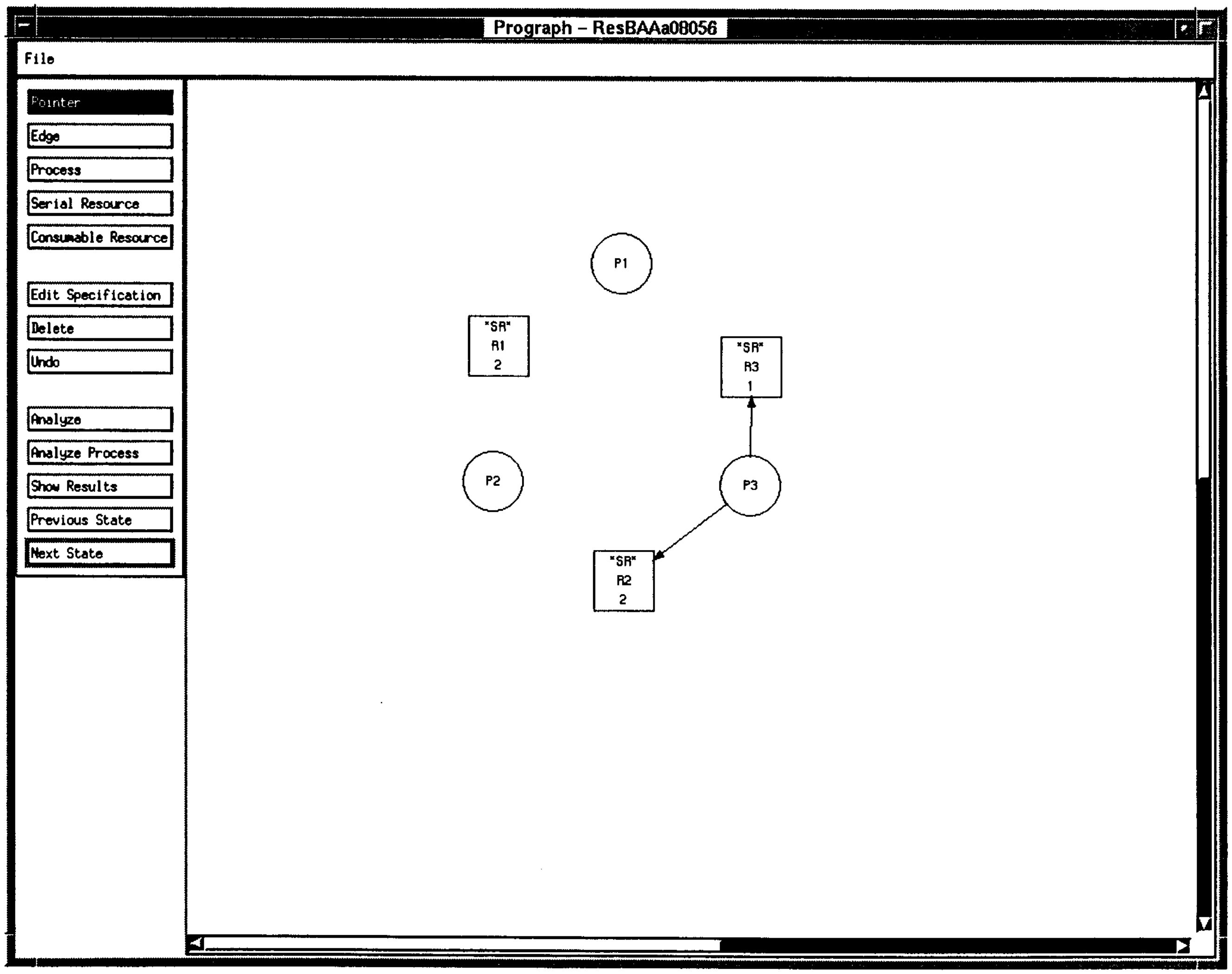


Figure 13. Step 2 in the reduction sequence for Figure 11

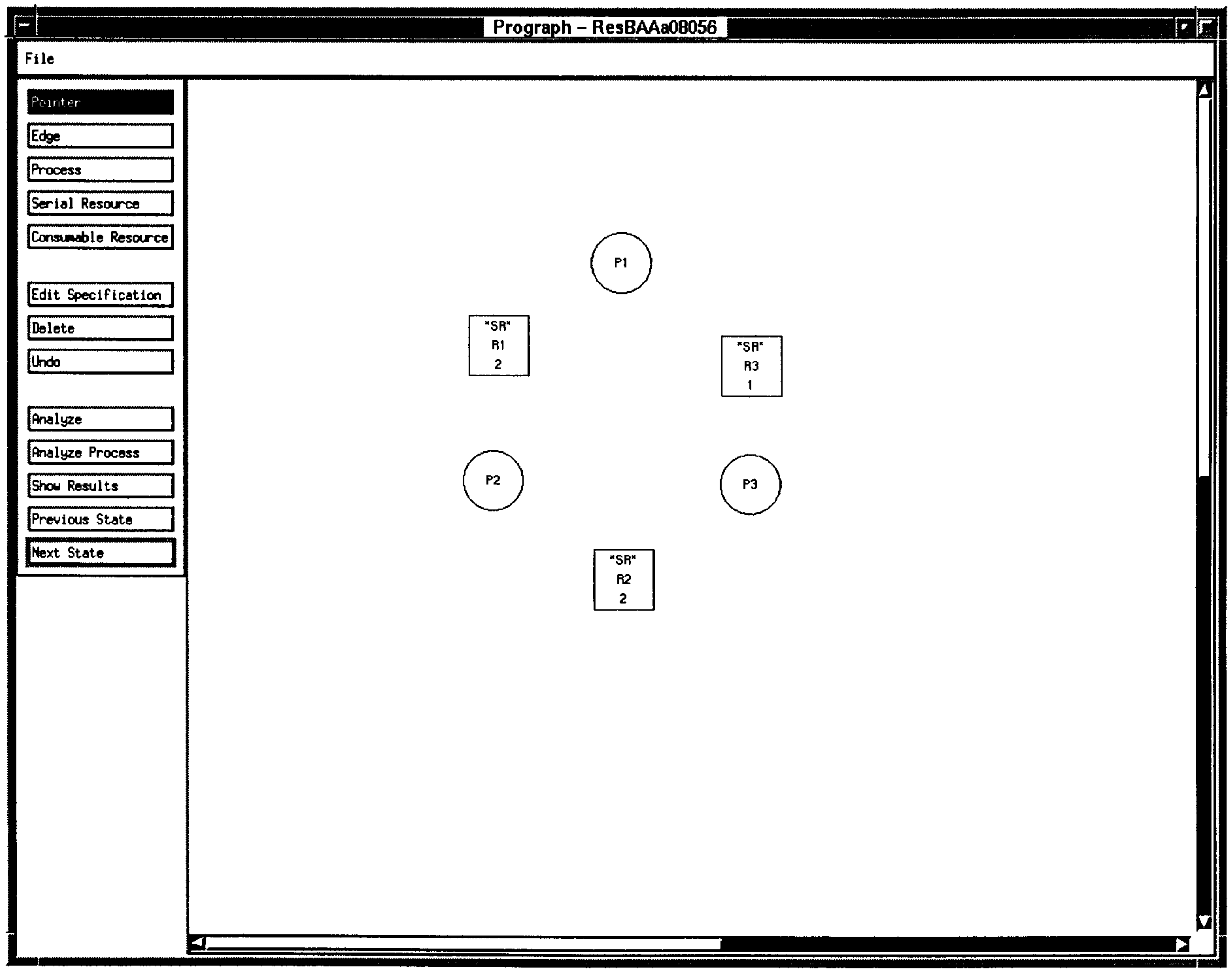


Figure 14. The final state of the reduction sequence for Figure 11

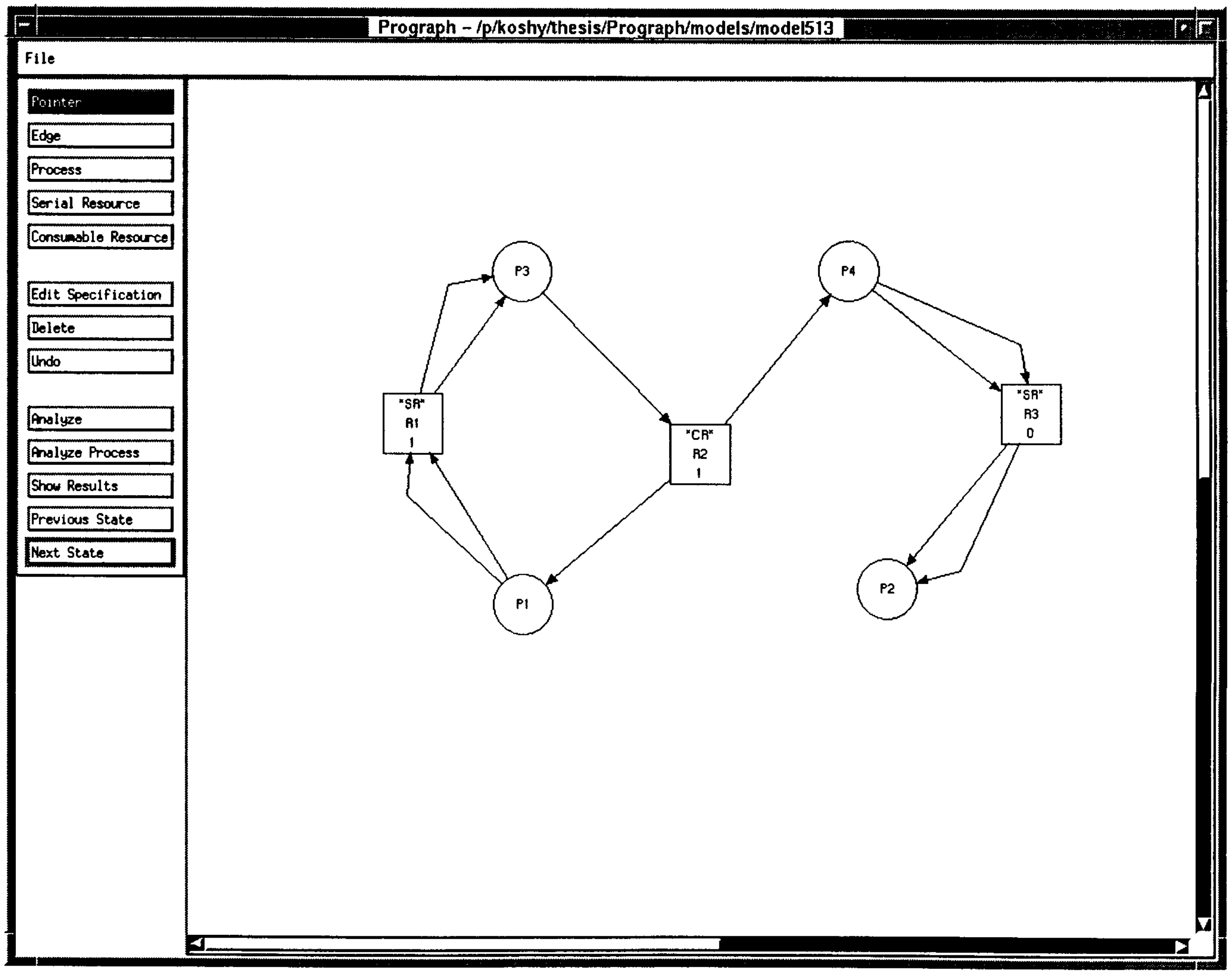


Figure 15. Another Sample Process-Resource Graph (Source [Nutt 92], Figure 5.13)

CHAPTER VI

SUMMARY AND FUTURE WORK

6.1 Summary

The importance of modeling Process-Resource graphs for deadlock analysis and the main objective of this thesis were discussed in Chapter I. Chapter II presented a survey of the current literature on Process-Resource graphs and deadlock analysis . This chapter presented an introduction to deadlocks and their detection. Also, the methods by which an operating system Process-Resource graph can be modeled were generally discussed in that chapter. Chapter II also contained a discussion of a number of different algorithms used for analyzing and detecting deadlocks. The various implementation issues of the software tool developed as part of this thesis, called Prograph, were discussed in Chapter III. The implementation platform and run-time environment, including an introduction to the Sequent Symmetry S/81, the X Window System, and the OSF/Motif widget set, were discussed in the various sections of Chapter III. Chapter IV of the thesis dealt with the software architecture and the detailed design of Prograph. Also, a description of the various parts of the user interface was presented in that chapter. Chapter V contained the results of the initial testing and evaluation of Prograph.

The main objective of this thesis was the development of a graphical software tool that can aid in building a Process-Resource model of the dynamic state of an operating system, analyze the graph for safe/unsafe states, and detect the presence of deadlocks.

This tool can be used to design and construct Process-Resource system states of virtually any size and complexity. The graphical user interface was implemented using the OSF/Motif widget set (see Chapter III for a discussion of the X Window System and the Motif Toolkit). This tool was used to design graphs of reasonable size and complexity in an academic environment.

A disadvantage of the tool is that, it is currently dependent on the Motif user interface. Also the algorithm used for deadlock analysis has an exponential time complexity and hence cannot be practically used to analyze large graphs.

6.2 Future Work

The improvement mentioned below should be incorporated into the future versions of Prograph.

- Automatic repositioning of the nodes and edges is not implemented currently. This feature can be helpful when designing large graphs, when it becomes difficult to control the web of edges traveling criss-cross across the graph.
- An on-line help system can be implemented in a future release of the tool.
- The analysis algorithm which has O(n!) complexity, can be made more efficient by incorporating heuristic searching.
- The cut, copy, and paste options can be implemented to provide additional flexibility in designing Process-Resource graphs.
- Currently, Prograph does not support the resizing of objects. This can be implemented in a future version of the tool.

- A grid background can be shown as a backdrop behind the drawing area, to assist in the placement of nodes.

- Related to the above item, support can be added to the tool for a snap effect, where nodes once moved should be allowed to be placed at discrete positions at predetermined and equidistant locations in the graph.

- Support can be added for aligning a group of nodes in various positions (e.g., center aligned).

- The tool can be extended to provide support for hierarchical Process-Resource graphs.

- Prograph can be redesigned to act as a tool to assist in situations requiring deadlock prevention. Similarly this concept can be used to provide support in environments requiring a deadlock avoidance approach or a deadlock detection and recovery approach.

- An extensive evaluation of the tool needs to be done, to ensure the robustness of the tool under stress conditions.

REFERENCES

[Barkakati 91] N. Barkakati, *X Window System Programming*, Macmillan Computer Publishing, Carmel, IN, 1991.

[Barkakati 93] Nabjyoti Barkakati, *Imaging and Animation for Windows*, Sams Publishing, Carmel, IN, 1993.

[Berlage 91] Thomas Berlage, *OSF/Motif: Concepts and Programming*, Addison-Wesley Publishing Company, Reading, MA, 1991.

[Coffman et al. 71] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System Deadlocks", *ACM Computing Surveys*, Vol. 3, No. 7, pp. 67-78, June 1971.

[Dannenberg 90] Roger B. Dannenberg, "A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors", *Software-Practice and Experience*, Vol. 20, No. 2, pp. 109-132, February 1990.

[DiLascia 92] Paul DiLascia, *Windows++: Writing Reusable Windows Code in C++*, Addison-Wesley Publishing Company, Reading, MA, 1992.

[Gansner et al. 88] E. R. Gansner, S. C. North, and K. P. Vo, "DAG - A Program that Draws Directed Graphs", *Software-Practice and Experience*, Vol. 18, No. 11, pp. 1047-1062, November 1988.

[Gansner et al. 93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo, "A Technique for Drawing Directed Graphs", *IEEE Transactions on Software Engineering*, Vol. 19, No. 3, pp. 214-230, March 1993.

[Habermann 69] A. N. Habermann, "Prevention of System Deadlocks", *Communications of the ACM*, Vol. 12, No. 7, July 1969.

[Heller 94a] Dan Heller, *Motif Programming Manual, Volume 6A*, O'Reilly & Associates, Inc., 1994.

[Heller 94b] Dan Heller, *Motif Reference Manual, Volume 6B*, O'Reilly & Associates, Inc., 1994.

[Holt 72] R. C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, Vol. 4, No. 9, pp. 179-196, September 1972.

[Isloor and Marsland 80] Sreekaanth S. Isloor and T. Anthony Marsland, "The Deadlock Problem: An Overview", *IEEE Computer*, Vol. 13, No. 11, pp. 58-78, November 1980.

[Johnson 90] E. F. Johnson and K. Richard, *Advanced X Window Application Programming*, Advanced Computer Books, Management Information Source Inc., Portland, OR, 1990.

[Keller 90] B. J. Keller, *A Practical Guide to X Window Programming*, CRC Press Inc., Boca Raton, FL, 1990.

[Lippman 91] Stanley B. Lippman, *C++ Primer*, Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1991.

[Nutt 92] G. J. Nutt, *Centralized and Distributed Operating Systems*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1992.

[Nye 90] Adrian Nye, *X Protocol Reference Manual for Version 11 of the X Window System*, O'Reilly and Associates Inc., Sebastapol, CA, 1990.

[Paulisch and Tichy 90] Frances N. Paulisch and Walter F. Tichy, "EDGE: An Extendible Graph Editor", *Software-Practice and Experience*, Vol. 20(S1), pp. 63-88, June 1990.

[Rowe et al. 87] Lawrence A. Rowe, Michael Davis, Eli Messinger, and Carl Meyer, "A Browser for Directed Graphs", *Software-Practice and Experience*, Vol. 17, No. 1, pp. 61-76, June 1987.

[Sequent 90] *DYNIX/ptx User's Guide*, Sequent Computer, Inc., 1990.

[Stroustrup 91] Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley Publishing Company, Reading, MA, 1991.

[Sugiyama et al. 81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda, "Methods for Visual Understanding of Hierarchical System Structures", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 11, No. 2, pp. 109-125, February 1981.

[Tamassia et al. 88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini, "Automatic Graph Drawing and Readability of Diagrams", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 18, No. 1, pp. 61-79, January 1988.

[Tanenbaum 92] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Inc. Englewood Cliffs, NJ, 1992.

[Vlissides and Linton 90] John M. Vlissides and Mark A. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors", *ACM Transactions on Information Systems*, Vol. 8, No. 3, pp. 237-268, July 1990.

[Warfield 77] J. N. Warfield, "Crossing Theory and Hierarchy Mapping", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 7, No. 7, pp. 505-523, July 1977.

[Wisskirchen 90] P. Wisskirchen , *Object-Oriented Graphics*, Springer Verlag, Berlin, 1990.

[Young 90] D. A. Young, *The X Window System: Programming and Applications with Xt*, OSF/Motif Edition, Prentice Hall Inc., Englewood Cliffs, NJ, 1990.

[Young 92] Douglas A. Young, *Object-Oriented Programming with C++ and OSF/Motif*, Prentice Hall Inc., Englewood Cliffs, NJ, 1992.

APPENDICES

APPENDIX A

GLOSSARY AND TRADEMARK INFORMATION

GLOSSARY

accelerator: Single keystrokes that are the equivalent of certain application functionalities, most commonly associated with menu selections.

application window: The window where an application resides with its complete user-interface.

background: The area on which a widget resides.

background color: The color from which all widgets generate their top and bottom shadows and their select color, and against which labels and bitmaps are created with the foreground color.

bitmap: An image created using only the foreground and background colors of the screen.

button: Either a physical button on the mouse or a widget that simulates a real button on the screen.

callback: A function or a procedure that is to be executed when a specific event occurs within a widget that is in a particular state.

class: A common description for a set of similar objects with the same structure but different attribute values. Each class has unique characteristics and any number of instances of the class may be created.

class hierarchy: A logical ordering of classes, in which each class lower in the hierarchy (sub-class) is a specialization of the class directly above it (super-class). Sub-classes may inherit, add, delete, or modify the attributes.

click: Pressing and immediately releasing a mouse button without moving the mouse in between.

client-server model: A server process in a client-server model provides some services to the other processes. These other processes are known as clients. In the X Window System, the server controls all input and output devices. An application is a client process that utilizes the services provided by the server.

composite widget: A widget that contains one or more widgets as its children, and controls their geometry.

Deadlock: A process in a multiprogramming system is said to be in a state of deadlock (or deadlocked) if it is waiting for a particular event that will not occur.

dialog box: A collection of widgets that are displayed by an application in response to an event when detailed information needs to be provided to the user or when input needs to be obtained from the user.

Directed Graph: A directed graph (or a digraph) G consists of a set of vertices $V = \{v_1, v_2, ..., v_n\}$, a set of edges $E = \{e_1, e_2, ..., e_n\}$, and a mapping that maps every edge in E onto some ordered pair of vertices (v_i, v_j) in V.

event: A message from the X server to an application.

event handler: A procedure that is executed in response to one or more predefined events for a widget.

geometry management: The process of automatic negotiation of the size and relative position of all child widgets.

graphical user interface (GUI): A visual representation of some of the functionality of a system that can be manipulated in a friendly, easy-to-use, and non-programmatic manner.

graphics context (GC): A data structure that contains various information necessary for drawing graphic objects on a window such as the foreground pixel, the background pixel, line width, line style, and clipping region. A graphics context is applicable only to drawables that have the same depth and root window as the graphics context.

GUI: See graphical user interface.

icon: A graphical symbol of an object or an action. Selecting an icon typically results in either selecting the object or performing the action.

inheritance: A mechanism that makes use of the characteristics of a super-class in a sub-class without the need for duplication.

inter-client communication conventions (ICCC): A set of protocols that govern the interaction among the clients as well as between a client and the window manager.

intrinsics: The base library of functions on which the Motif widget set has been built. It implements the fundamental procedures for building new widget classes.

MULTIBUS: An industry standard for buses that may be used to connect a variety of peripheral devices.

pixel: A single identifiable point on the screen or in a pixmap. A pixel may have different color values, or may be white or black in the case of a monochrome monitor.

pixel values: An n-bit value, where n is the number of bit planes in a window or a pixmap. In other words, n is the depth of the pixmap or window. In the case of a window, it indexes a colormap to derive the actual color to be made visible.

pixmap: A three-dimensional array of bits that can be considered as a two dimensional array of pixels. The value of each pixel can range from 0 to 2^{n-1}, where n is the depth of the pixmap. Alternately, a pixmap may be viewed as a stack of n bitmaps.

pointer: A synonym for the mouse cursor.

Resource: A component managed by an operating system for which there is competition among different processes.

server (X Window System): It offers the basic windowing mechanism. It is responsible for handling inter-process communication connection between clients, graphic requests, and demultiplexes and screens. It is also responsible for multiplexing input back to the appropriate clients.

Starvation: A condition in which a process in a multiprogramming system waits for a resource for an unbounded period of time.

toolkit: A low-level library of objects and functions that are available for use to the application programmer and upon which the intrinsics are built.

widget: A user interface mechanism comprising data structures and the associated procedures that can be displayed in different ways such as menus, dialog boxes, or windows.

window: A rectangular area on the screen that belongs to a particular application.

window manager: The program that manages the display of windows and their manipulation on the screen.

X: A networked, portable, and transparent windowing system.

X client: An application program that makes use of the services of the X server for input and output.

Xm: The prefix for any value assigned to a widget resource. This convention differentiates the X Window system and the Motif widget set values from values assigned to other variables in the source code.

XmN: The prefix for any resource attribute whose value needs to be specified.

X protocol: The protocol by which X clients communicate with the X server.

X server: A set of C language routines that exclusively control the display hardware and service client requests.

Xt Intrinsics: A synonym for X toolkit intrinsics.

X toolkit intrinsics: A library of functions, procedures, and data structures built on top of Xlib that makes application programming much easier compared to working with Xlib functions.

X Window System: A network-transparent and hardware-independent base layer that provides services to graphical user interfaces.

TRADEMARK INFORMATION

DEC is a registered trademark of Digital Equipment Corporation.

DYNIX, DYNIX/ptx, Sequent S/81, and Symmetry are registered trademarks of Sequent Computer Systems, Inc.

Motif, OSF, and OSF/Motif are registered trademarks of the Open Software Foundation.

The X Window System is a registered trademark of the Massachusetts Institute of Technology.

UNIX is a registered trademark of AT&T.

APPENDIX B

USER GUIDE FOR PROGRAPH

1. General Overview

This is a step-by-step explanation of how to use Prograph to model and analyze a Process-Resource graph. Start up Prograph by typing /p/koshy/Prograph/Prograph or the path name for the new location of Prograph. When the program loads up, the initial Prograph screen appears (See Figure on the following page). In the center of the screen is the drawing area. On the top left hand corner is the menu bar containing the *File* menu, which contains the menu items: *New*, *Open*, *Save*, *Save As*, *Close,* and *Quit.* The use of these menu items should be obvious if you have previous experience with any graphical drawing or editing package. You will also notice scroll bars along the edge of the drawing area. On the left end of the screen is a toolbar containing a number of buttons organized into three sets.

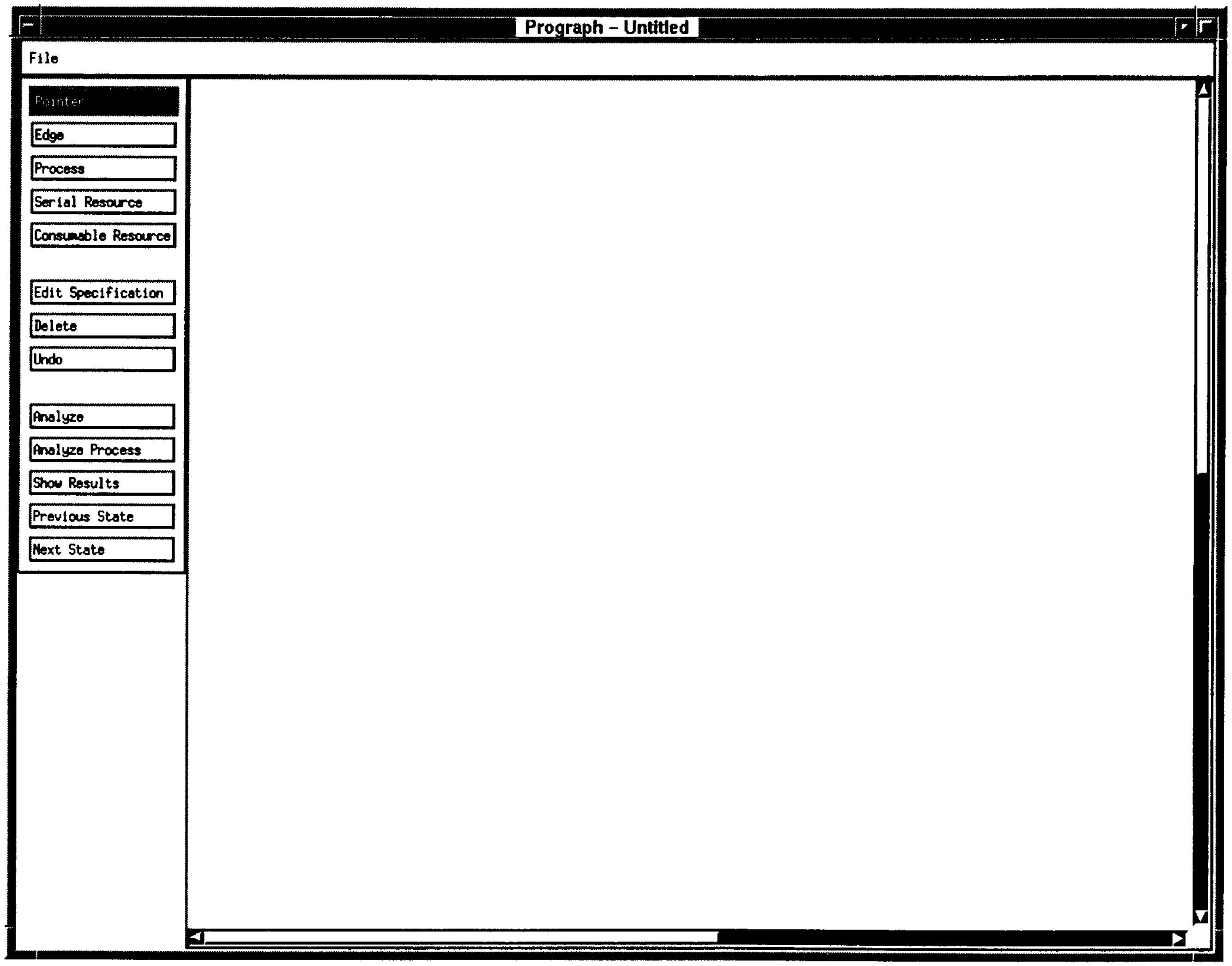
Set1: [Pointer] [Edge] [Process] [Serial Resource] [Consumable Resource]
Set2: [Edit Specification] [Delete] [Undo]
Set3: [Analyze] [Analyze Process] [Show Results] [Previous State] [Next State]

The interface to Prograph is a point and click interface. You can select objects in the drawing area by clicking on them, move them about by dragging them, etc. Another feature of the tool is the auto-scrolling facility. In the middle of a drawing or dragging operation, if you chance to move the pointer out of the drawing area, the drawing area will automatically scroll to provide a virtual screen effect. The scrolling will continue until you reach the edge of the virtual drawing area, which is currently set to about four times the size of the physical drawing area. Of course, you can also use the scroll bars, but the auto-scrolling facility comes in handy if you want to scroll in the middle of a drawing or dragging operation.

The rest of this User Guide explains how to proceed to analyze a Process-Resource graph.

2. Step 1: Describe the Problem on the Screen

In the first stage you will need to construct a graphical representation of the problem. For this, you will be using the buttons in Set1 and Set2.



Initial Prograph Screen

Set1 is known as the "mode" set. One of the items in Set1 will be highlighted at all times. If you click on another item in Set1, the previous item's highlight will disappear and the new item will be highlighted. The highlighted item denotes the current mode you are in. One common problem while constructing a model on the screen, is forgetting which mode you are in. When in doubt, switch to the [Pointer] mode.

The [Pointer] mode is used for selecting, dragging, and just about all other editing operations. When you select an object by clicking on it, the object will be highlighted by the presence of 8 hilite handles on its corners, except for Edge objects which will have hilites for each endpoint of each segment of the edge.

If you switch to the [Process] mode, any time you click in the drawing area, you will be getting a new process object. Each process object will be displayed with its default name in the center of the object. In the [Serial Resource] or the [Consumable Resource] mode clicking at a point on the screen will give you a new serial or consumable resource object, respectively. When created, a new serial resource or consumable resource object is set by default to contain one resource unit. Each resource object will have 3 items displayed on it. On the top is either an *SR* denoting a Serial Resource or a *CR* denoting a Consumable Resource. At the center of the object is its default name. At the bottom of the object is the number of available units in the object. When using any of the three modes [Process], [Serial Resource], or [Consumable Resource] in generating a new object, you can move the object to an appropriate place in the same operation (before releasing the mouse button). A newly-created object will appear hilited.

The [Edge] drawing mode enables you to draw edges from a process object to a resource object or vice versa. Once you have switched to the [Edge] drawing mode, you can start drawing edges. To draw an object from, say, process p1 to resource r1, click the mouse on p1 and move the mouse outside p1, at this stage you can decide to directly connect to r1 or have a few temporary intermediate points. Clicking anywhere on the drawing area will set the line at that point, Finally, click on r1 to finish the line drawing (actually, the whole operation is straightforward and works similarly to line-drawing operations in some of the commercial packages). When an edge is drawn from or to an object, the edge will have an affinity towards the center of the object. In the example above, when you enter r1, the edge will be captured temporarily by r1, and r1 will release the edge when you move outside of r1. Note that you cannot draw an edge from a process to a process or from a resource to a resource.

There are three types of edges: Request edges, Assignment edges, and Producer edges. When an edge is drawn from a process to a resource, it is a request edge. When an edge is drawn from a serial resource to a process, it is an assignment edge. When you draw an edge from a consumable resource to a process, you are denoting that the process is a producer of that consumable resource, and hence the edge is called a producer edge. If you try to draw an assignment edge from a serial resource containing no available resource units, the edge drawing operating will abort on completion, and a message box will popup. To prevent this, you will have to increase the number of resource units of that resource type, and we will discuss how to do it in the next section.

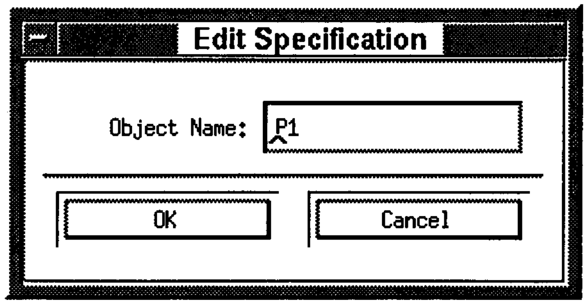
Set2 buttons contain the edit set. It contains various items useful for editing a graph. The items in Set2 are [Edit Specification], [Delete], and [Undo].

Using [Edit Specification] you can edit the various properties of an object. The only property of a process or an edge is its name. For a resource, the additional properties are its type and the number of units it holds. To edit an object's specification, it has to be hilited. This can be done by switching to the [Pointer] mode and clicking on the object. Once there is a hilited object, clicking on [Edit Specification] will popup a specification dialog box. For a process or an edge, the only item on the dialog box will be a name text entry box. The edit specification dialog box for a process or an edge is shown in the figure on the following page.

In the case of an edit specification dialog box for a resource, there will be a label denoting the type of the resource. Also, there will be a text entry box to enter the number of units. The edit specification dialog box for a resource is shown in the figure on the following pages. Consider the case of a serial resource. When a serial resource is created, it starts up with 1 unit. Suppose the unit is assigned to a process (by way of drawing an assignment edge), the unit count reduces by one and the resource display will show 0 available units. This 0 is the number of available units, while the number of total units of the resource is still 1. So the number of total units is the number of units assigned to a resource, while the available units is that which is remaining. In the case of a consumable resource, the number of available units is the same as the number of total units, since once a consumable resource unit is assigned to a process, it is gone forever.

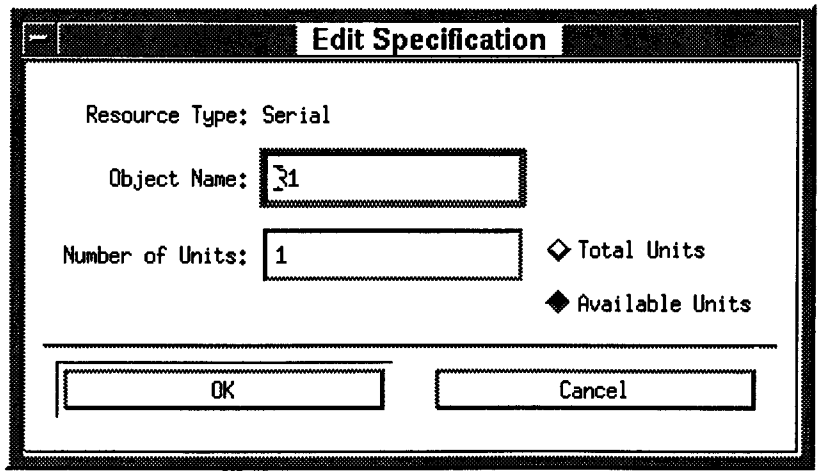
When the edit specification dialog for a resource pops up, beside the entry area for the number of units, you will notice two toggle buttons, one labeled {Total Units} and the other one labeled {Available Units}. By default when popping up, the toggle is on {Available Units} and the number displayed on the Units entry area is the number of available units for that resource. Clicking on the {Total Units} will result in the total number of units held by the resource being displayed in the Units entry area.

It is preferable to edit the number of units with the toggle on {Available Units}. When you increase the number of available units, the number of total units also increases by the same amount. Decreasing the number of available units likewise decreases the number of total units. You cannot decrease the number of available units below zero. The changes can immediately be observed by toggling to {Total Units}. If you prefer to edit the number of total units (i.e., the toggle is on {Total Units}), increasing it will increase the number of available units by the same amount. When decreasing the number of total units, you can only decrease it so that the corresponding decrease in the number of available units will not make the number of available units less than zero. If you enter an



Object Name: P1

Edit Specification Dialog Box for a Process or Edge



Resource Type: Serial

Object Name: 31

Number of Units: 1

◇ Total Units

[✓ ◆ Available Units]

Edit Specification Dialog Box for a Resource incorrect value for the available units and toggle to {Total Units}, the toggle will remain in the {Available Units}, and vice versa.

If the discussion in the previous paragraphs seem confusing to you, then the easiest way to approach the editing is to edit the specification of the resources before starting to draw any edges. In other words, after placing all the resources on the screen, simply set the number of available units of each resource to that mentioned in the problem statement.

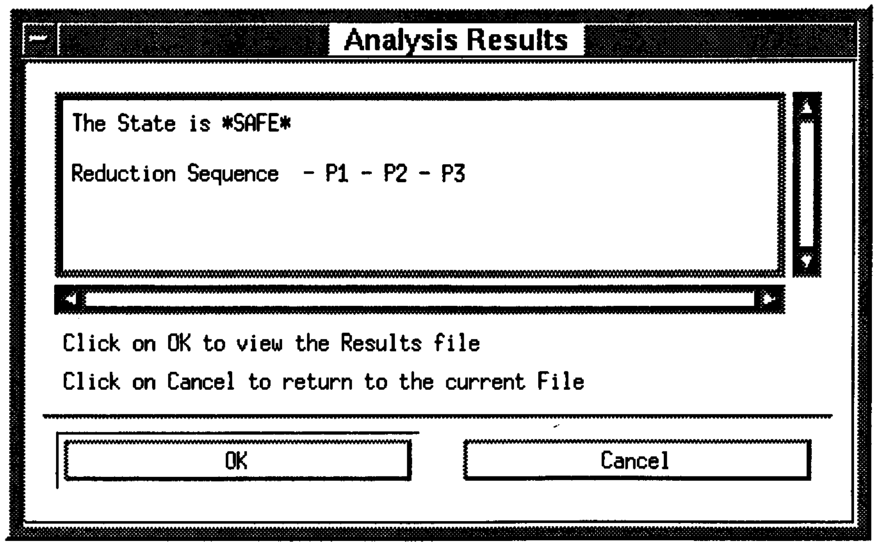
Clicking the [Delete] key will delete the currently hilited object. The [Undo] key will undo the previously completed operation. For example, you can undo deletion of objects, movement of objects, drawing new objects, etc. One thing to note is that selecting an object is also considered an operation; so, if you delete an object and then select another object, the undo command will not undelete the deleted object, since the deletion operation would have become the previous to previous operation.

3. Step 2: Analyze the Problem

Ok, by now you must have a model on the screen exactly representing the problem statement, recheck to make sure the figure meets all the specifications and all the assignment, and the request and producer edges (if any) are correctly in place. If you haven't saved the file by now, save it under an appropriate name. This brings us to the buttons in Set3.

Set3 is the analysis set, it consists of various items for analysis. After a model is drawn or displayed on the drawing area, selecting the [Analyze] button will analyze the model state to see if it is safe or deadlocked. After the analysis, a dialog box pops up, displaying the results. The Analysis Results dialog box is shown on the following page. Clicking OK on the dialog box, will load up the Results file (the file currently on the screen, which is the model you had drawn, will be closed; if you haven't saved that file, the system will prompt you to do so). The Results file is now loaded up on the screen. It will be assigned an automatically generated unique file name. The Results file will not be automatically saved, you will have to save it explicitly if necessary.

If the analysis of the model resulted in a safe state, then the analysis dialog box which you saw earlier would have mentioned a reduction sequence. The Results file contains the reduction sequence. The first figure which you see in the Results file is the start state, Clicking on the [Next State] button, will take you to the next state obtained by reducing the first process in the reduction sequence. When you do so, you may observe that some of the edges have disappeared, the number of resource units updated, etc. These changes correspond to the fact that the first process' requested resources have been assigned and it has released any resources it was holding, or it has generated an infinite number of units if it is the producer of a consumable resource. In Prograph, following the classical deadlock analysis conventions, infinity is represented by the symbol w.



```
The State is *SAFE*

Reduction Sequence  - P1 - P2 - P3
```

Click on OK to view the Results file

Click on Cancel to return to the current File

Analysis Results Dialog Box

Continue stepping through the reduction sequence by clicking the [Next State] button each time. If at any point you want to go back, click on the [Previous State] button. So you can step back and forth through the Results file. Clicking on the [Show Results] button will popup a dialog box, which will show you the analysis information pertaining to the Results file (this is the same information you saw earlier).

Save the Results file using an appropriate file name by using the *Save As* menu item (or you can use the automatically-generated filename itself, if you wish). If you want to do additional analysis on the model, first close the Results file and then open the previously-saved model file and proceed.

The [Analyze Process] is similar to [Analyze] except that the analysis starts with a particular process which you specify. When you select [Analyze Process], you will be asked to specify a process to reduce by. The system will analyze whether that process is safe or deadlocked. The same dialog box will popup as before. You can click OK if you wish to see the Results file.

After an analysis, if the state is deadlocked, the Results file will contain a single model. In that case, clicking on [Next State] or [Previous State] will simply redisplay that one model.

4. Aborting a Drawing Operation

You can abort a drawing operation at any time by pressing the Esc key. This might be useful when doing edge drawing.

APPENDIX C

SYSTEM ADMINISTRATOR GUIDE FOR PROGRAPH

1. Maintenance

Most of the configurable parameters of Prograph are stored in the header file "Types.h".

The title of the Program is currently "Prograph". This string is appended to the name of all files shown at the title bar on top of the user interface. The title can be changed by redefining the string variable PROGRAPH_TITLE.

The maximum number of points allowed in a Polyline is set currently to 100. This can be changed by redefining the variable MAX_POINTS. Also, the length of an arrow for an edge is set by default to 10 pixels and the half width of the arrow is set at 4 pixels. These are defined in the variables ARROW_LENGTH and ARROW_HALF_WIDTH, respectively, and can be changed as required.

The maximum length of a file name is currently set at 100 characters. This can be changed by redefining the variable MAX_FILE_NAME. Similarly, it is expected that the report string of an analysis will be no longer than 4000 characters. This limit can be changed by modifying the variable MAX_ANALYSIS_RESULTS_STR_LENGTH. The font for displaying the various labels on a node can be changed by redefining the variable FONT_NAME.

When a user clicks on a line, a tolerance of a few pixels is allowed. The default value of the tolerance level is 3 pixels on either side, or a total width of 6 pixels. This is defined in the variable HIT_LINE_WIDTH.

Currently, the virtual drawing area is divided into a 30 rows and 30 columns, each row or column being 50 pixels high and 50 pixels wide, respectively. A row or a column represents the minimum amount that the screen will scroll due to a scroll event. The NUM_CELL_ROWS variable holds the number of rows, and the NUM_CELL_COLUMNS variable holds the number of columns. The height of a row is defined in CELL_HEIGHT, and the width of a column is defined in CELL_WIDTH. Changing these parameters will result in the changing of the virtual screen area as well as the scrolling parameters.

When a new node is created, the size of the node is set by default to 50 pixels. This can be changed by redefining the variable DEFAULT_NODE_SIZE.

When a shape is clicked, a number of handles appear on the corners or end points of the shape. The default size of the square handles is 5 pixels. This size can be changed by modifying the variable HANDLE_SIZE.

The text that appears on a resource shape, to depict whether it is a serial resource or a consumable resource, is currently "*SR*" or "*CR*", respectively. These can be changed by redefining the variables SERIAL_RESOURCE_TEXT for serial resources and the variable CONSUMABLE_RESOURCE_TEXT for consumable resources.

The prefix of all automatically generated result files is currently set to "Res". This can be modified by redefining the variable RESULT_PREFIX.

2. Options

Prograph is a software tool based on the Motif widget set. As mentioned earlier, the Motif widget set is built upon the X Toolkit. Hence, Prograph accepts all the standard X Toolkit command line options [Young 90]. The more popular options accepted by applications written using the X Toolkit are briefly described below.

-bg color	This option can be used to change the background color of an application window. The default background color is white.
-display display	This option is used to define the X server to which an application is to be connected.
-fg color	This option can be used to change the foreground color of an application window. The default foreground color is black.
-iconic	This option ensures that Prograph is started by the window manager as an icon rather than as a normal window
-rv	This option is used to swap the background and foreground colors of an application.
-title string	This option sets the title of an application window. This option may be ignored by the window manager. The default window title is the string specified after the -e command line option. If none is specified, the application name is used.

APPENDIX D

SAMPLE PROGRAM LISTINGS

Due to space constraints the entire code listing could not be included in the appendix. Only the code for the Analyzer module is listed in this appendix. The Analyzer module contains the algorithms for analyzing a graph for deadlock, and hence was considered the most important module of the entire program. The module has two files: a header file, PrographAnalyzer.h and a source file, PrographAnalyzer.C. The listing is given below.

```
////////////////////////////////////////////////////////////////////////////////
//
//                    P r o g r a p h A n a l y z e r . h
//                    -----------------------------------
//
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
//
//  Header file for the Analyzer module
//
////////////////////////////////////////////////////////////////////////////////

#ifndef PROGRAPHANALYZER_H
#define PROGRAPHANALYZER_H

// Forward reference
//
class TPrographView;

////////////////////////////////////////////////////////////////////////////////
//  class TPrographAnalyzer
////////////////////////////////////////////////////////////////////////////////
//
class TPrographAnalyzer {

private:

    TPrographView *_theView;

    TGraphShapeList *_processList;
    TGraphShapeList *_resourceList;
    TGraphShapeList *_edgeList;

    int _numProcesses;
    int _numResources;
    int _numEdges;

    int **_allocMatrix;
    int **_requestMatrix;
    int **_producerMatrix;  // For producer process of consumable resources

    int *_availVector;      // Vector containing the number of units of
                            // each resource that are available
    int *_totalVector;      // Vector containing the total number of units
                            // of each resource available
    ResourceType *_typeVector;  // Vector holding the types of the various
                                // resources

    // Duplicate matrices and vectors
    int **_orgAllocMatrix;
    int **_orgRequestMatrix;
    int **_orgProducerMatrix;
    int *_orgAvailVector;
    int *_orgTotalVector;

    // Functions private to the Analyzer
    //
    void initializeInternalState(TPrographModel *aModel);

    void restoreOriginal();
    BOOL isCombinationReducible(int *comb, int setSize);
    void reduceCombination(int *comb, int setSize,
        TGraphShapeList *freedProcessList, TPrographModelList *modelList);
    ReduceByProcessType reduceByProcess(int process, int *tempAvailVector);

    void generateAnalysisResults(BOOL thereIsACannotReduce,
        TGraphShapeList *freedProcessList, char *analysisResultsStr);
    TPrographModel *generatePrographModel(TGraphShapeList *freedProcessList);
    void addAllEdgesToModelExcept(TGraphShapeList *freedProcessList,
        TPrographModel *newModel, TGraphShapeList *newProcessList,
        TGraphShapeList *newResourceList);

    BOOL isLessThanOrEqualTo(int *lhsVector, int *rhsVector, int size);
    void addVector(int *vector, int *vectorToBeAdded, int size);
    void subtractVector(int *vector, int *vectorToBeSubtracted, int size);
    void addProducerVector(int *vector, int *producerVector, int size);

    ResourceType *allocateResourceTypeVector(int size);

    // Function for debugging
    void printInternalState();


public:

    TPrographAnalyzer(TPrographView *theView);
    ~TPrographAnalyzer();

    TPrographModelList *analyze(TPrographModel *aModel, ReduceType reduceType,
        TGraphShape *startProcess = NULL);

};

#endif
```

```
///////////////////////////////////////////////////////////////////////////
//
//                       P r o g r a p h A n a l y z e r . C
//                       -----------------------------------
//
///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////
//
//  Source file for the Analyzer module
//
///////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include "Types.h"
#include "Point.h"
#include "Rectangle.h"

#include "Shape.h"
#include "PureShape.h"
#include "GraphShape.h"
#include "PrographModel.h"
#include "PrographAnalyzer.h"

#include "Command.h"
#include "MoveCommand.h"
#include "NewNodeCommand.h"
#include "SizeCommand.h"
#include "EdgeCommand.h"

#include "PrographView.h"
#include "PaletteView.h"
#include "Document.h"
#include "PrographWindow.h"


// Other useful functions
//
int **generateAllCombinations(int setSize, int &numCombs);
int **generateSpecialCombinations(int specialNumber, int setSize,
    int &specialNumCombs);
void generateCombinations(int numCombs, int setSize, int **combs);
BOOL isCombination(int setSize, int *perm);
void incrementPermutation(int setSize, int *perm);
int power(int x, int y);
int factorial(int x);

BOOL isZeroVector(int *vector, int size);
void setToZero(int *vector, int size);
int **allocateIntegerMatrix(int numRows, int numColumns);
void freeIntegerMatrix(int **matrix, int numRows);
int *allocateIntegerVector(int size);

void copyMatrix(int **srcMatrix, int **destMatrix, int numRows, int numColumns);
void copyVector(int *srcVector, int *destVector, int size);
void printMatrix(int **matrix, int numRows, int numColumns);
void printVector(int *vector, int size);

///////////////////////////////////////////////////////////////////////////
//  Member functions for class TPrographAnalyzer
///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////
//  T P r o g r a p h A n a l y z e r
```

```
//
//  Description:
//
//  Constructor for the TPrographAnalyzer object. Initializes the view
//
TPrographAnalyzer::TPrographAnalyzer(TPrographView *theView)
{
    _theView = theView;
}

////////////////////////////////////////////////////////////////////////////////
//  ~T P r o g r a p h A n a l y z e r
//
//  Description:
//
//  Destructor for the TPrographAnalyzer object. Here we free the matrices
//  and vectors used for the analysis.
//
TPrographAnalyzer::~TPrographAnalyzer()
{
    freeIntegerMatrix(_allocMatrix, _numProcesses);
    freeIntegerMatrix(_requestMatrix, _numProcesses);
    freeIntegerMatrix(_producerMatrix, _numProcesses);
    free (_availVector);
    free (_totalVector);
    free (_typeVector);

    // Free duplicate matrices
    freeIntegerMatrix(_orgAllocMatrix, _numProcesses);
    freeIntegerMatrix(_orgRequestMatrix, _numProcesses);
    freeIntegerMatrix(_orgProducerMatrix, _numProcesses);
    free (_orgAvailVector);
    free (_orgTotalVector);
}

////////////////////////////////////////////////////////////////////////////////
//  a n a l y z e
//
//  Description:
//
//  The member function which is called externally for analyzing a model.
//  A TPrographModel object is passed as an argument. The function after the
//  analysis returns a list of models representing the successive states of
//  the reduction sequence.
//
TPrographModelList *
TPrographAnalyzer::analyze(TPrographModel *aModel, ReduceType reduceType,
    TGraphShape *startProcess)
{
    int comb;
    int process;
    int **combs;
    int numCombs;
    BOOL thereIsACannotReduce;
    TGraphShapeList *freedProcessList;
    TPrographModel *newModel;
    TPrographModelList *modelList;
    char analysisResultsStr[MAX_ANALYSIS_RESULTS_STR_LENGTH];

    initializeInternalState(aModel);
    modelList = new TPrographModelList(_theView);
    freedProcessList = new TGraphShapeList;
    // Generate the existing model
    //
    newModel = generatePrographModel(freedProcessList);
    modelList->add(newModel);
    if (reduceType == REDUCE_SPECIFIC)
    {
        process = _processList->getIndex(startProcess);
        combs = generateSpecialCombinations(process, _numProcesses, numCombs);
    }
    else
    {
        combs = generateAllCombinations(_numProcesses, numCombs);
    }

    thereIsACannotReduce = TRUE;
    for (comb = 0; comb < numCombs; comb++)
    {
        restoreOriginal();

        if (isCombinationReducible(combs[comb], _numProcesses) == TRUE)
        {
            restoreOriginal();
            reduceCombination(combs[comb], _numProcesses, freedProcessList,
                modelList);
            thereIsACannotReduce = FALSE;
            break;
        }
    }

    // Finishing touches
    //
    generateAnalysisResults(thereIsACannotReduce, freedProcessList,
        analysisResultsStr);
    modelList->setAnalysisResults(analysisResultsStr);
    modelList->setChangesMade();

    // We will have to popout each entry of the freedProcessList
    free (freedProcessList);
    freeIntegerMatrix(combs, numCombs);
    return modelList;

}

////////////////////////////////////////////////////////////////////////////////
//  r e s t o r e O r i g i n a l
//
//  Description:
//
//  Restore the orignal values of the matrices and vectors. This function
//  is called before starting to analyze a new combination, since the
//  analysis of the previous combination would have modified the matrices
//  and vectors.
//
void
TPrographAnalyzer::restoreOriginal()
{
    // Restore the original state
    copyMatrix(_orgAllocMatrix, _allocMatrix, _numProcesses, _numResources);
    copyMatrix(_orgRequestMatrix, _requestMatrix, _numProcesses,
                                                      _numResources);
    copyMatrix(_orgProducerMatrix, _producerMatrix, _numProcesses,
                                                      _numResources);
    copyVector(_orgAvailVector, _availVector, _numResources);
    copyVector(_orgTotalVector, _totalVector, _numResources);
}

////////////////////////////////////////////////////////////////////////////////
//  i s C o m b i n a t i o n R e d u c i b l e
//
//  Description:
//
```

```
// Check if a particular combination (Say 2-1-3, in the case of a system
// containing 3 processes) is reducible.
//
BOOL
TProgramAnalyzer::isCombinationReducible(int *comb, int setSize)
{
    int i, process;
    int *tempAvailVector;
    ReduceByProcessType reduce;

    for (i = 0; i < setSize; i++)
    {
        process = comb[i];
        tempAvailVector = allocateIntegerVector(_numResources);
        reduce = reduceByProcess(process, tempAvailVector);

        switch (reduce)
        {
        case CAN_REDUCE:
            addVector(_availVector, tempAvailVector, _numResources);
            break;

        case CANNOT_REDUCE:
            free (tempAvailVector);
            return FALSE;

        case NEED_NOT_REDUCE:
            break;
        }

        free (tempAvailVector);
    }

    return TRUE;
}

//////////////////////////////////////////////////////////////////////////////////
// r e d u c e C o m b i n a t i o n
//
// Description:
//
// Reduce the original state by a particular combination (Say 2-1-3). This is
// done after checking if the above combination is found to be reducible.
//
void
TPrographAnalyzer::reduceCombination(int *comb, int setSize,
    TGraphShapeList *freedProcessList, TPrographModelList *modelList)
{
    int i, process;
    int *tempAvailVector;
    ReduceByProcessType reduce;
    TGraphShape *processShape;
    TPrographModel *newModel;

    for (i = 0; i < setSize; i++)
    {
        process = comb[i];
        tempAvailVector = allocateIntegerVector(_numResources);
        reduce = reduceByProcess(process, tempAvailVector);
        assert (reduce != CANNOT_REDUCE);

        if (reduce == CAN_REDUCE)
        {
            addVector(_availVector, tempAvailVector, _numResources);
            processShape = _processList->getShapeAtIndex(process);
            freedProcessList->addToBack(processShape);
            newModel = generatePrographModel(freedProcessList);
```

```
                modelList->add(newModel);
            }

            free (tempAvailVector);
        }
}

////////////////////////////////////////////////////////////////////////////////
//  r e d u c e B y P r o c e s s
//
//  Description:
//
//  Given a combination, this function reduces the orignal state of the graph
//  by a particular process in the combination (passed as an argument).
//  So this function is repeatedly called by the reduceCombination function
//  to reduce a complete combination.
//
ReduceByProcessType
TPrographAnalyzer::reduceByProcess(int process, int *tempAvailVector)
{
    int resource;

    // If the request matrix is less than the available matrix
    // then we can reduce by this process, if not return FALSE
    //
    if (isLessThanOrEqualTo(_requestMatrix[process], _availVector,
                                            _numResources) == FALSE)
    {
        return CANNOT_REDUCE;
    }

    // However now we have to check if the process is already
    // isolated from the rest of the graph (no edges, i.e., all matrix
    // rows of the process will be zeros). If so, then we dont need
    // to consider this process
    //
    if ((isZeroVector(_requestMatrix[process],  _numResources)) &&
        (isZeroVector(_allocMatrix[process],    _numResources)) &&
        (isZeroVector(_producerMatrix[process], _numResources)))
    return NEED_NOT_REDUCE;

    addVector(tempAvailVector, _allocMatrix[process],
        _numResources);

    // Now our aim is to free up as many processes's claims as we
    // can. So we will be continuing with the for loop. So the
    // next process down the line, will see a reduced _availVector
    // (due to the subtraction), if it can satisfy itself with that,
    // then that process also will be freed. We add the request
    // vector to the tempVector, so that it can be added later on
    // Note this whole process is a virtual operation
    //
    subtractVector(_availVector, _requestMatrix[process],
        _numResources);

    // To handle a consumable resource, say Rx, we set the
    // _requestMatrix[process][Rx] to 0, because we should prevent that
    // entry being added to tempAvailVector (which will be added later
    // to _availVector). Note that this operation is done after
    // subtracting the _requestMatrix[process] vector from the
    // _availVector. I dont know whether this is an elegent solution
    // or not
    // Because a consumable resources unit will be gone after the allocation
    //
    for (resource = 0; resource < _numResources; resource++)
    {
        if (_typeVector[resource] == RESOURCE_CONSUMABLE)
```

```
            _requestMatrix[process][resource] = 0;
        }

        addVector(tempAvailVector, _requestMatrix[process],
            _numResources);

        addProducerVector(tempAvailVector, _producerMatrix[process],
            _numResources);

        setToZero(_allocMatrix[process], _numResources);
        setToZero(_requestMatrix[process], _numResources);
        setToZero(_producerMatrix[process], _numResources);

        // Since we can reduce by the process, return CAN_REDUCE
        //
        return CAN_REDUCE;
}


////////////////////////////////////////////////////////////////////////////////
//  g e n e r a t e P r o g r a p h M o d e l
//
//  Description:
//
// This routine uses the current internal state of the Analyzer object
// to create a model
//
TPrographModel *
TPrographAnalyzer::generatePrographModel(TGraphShapeList *freedProcessList)
{
    int process;
    int resource;
    TGraphShape *processShape;
    TGraphShape *resourceShape;
    TProcess    *newProcessShape;
    TResource   *newResourceShape;
    TGraphShapeList *newProcessList;
    TGraphShapeList *newResourceList;
    TPrographModel *newModel;

    newModel = new TPrographModel(_theView);

    newProcessList = new TGraphShapeList;
    newResourceList = new TGraphShapeList;

    // Add all the processes and resources to the model.
    //
    processShape = _processList->first();
    while (processShape != NULL)
    {
        newProcessShape = (TProcess *) processShape->clone();
        newModel->addNode(newProcessShape);

        // We construct this so that when we add the edges we can send its new
        // source and destination shapes. Note - addToBack is very important
        //
        newProcessList->addToBack(newProcessShape);
        processShape = _processList->next();
    }

    resourceShape = _resourceList->first();
    while (resourceShape != NULL)
    {
        resource = _resourceList->getIndex(resourceShape);
        newResourceShape = (TResource *) resourceShape->clone();
        newResourceShape->initializeResourceCounts();
```

```
            // Now in the new model, we set the number of avaliable resources
            // to the total (in the case of SERIAL resources). Later on when we
            // add edges. this will automatically reduce
            //
            if (_typeVector[resource] == RESOURCE_SERIAL)
            {
                newResourceShape->setNumTotalResources(_totalVector[resource],
                    FALSE); // Dont refresh text display
            }
            else
            {
                newResourceShape->setNumAvailableResources(_availVector[resource],
                    FALSE); // Same explanation as above
            }

            newModel->addNode(newResourceShape);
            newResourceList->addToBack(newResourceShape);
            resourceShape = _resourceList->next();
        }

        addAllEdgesToModelExcept(freedProcessList, newModel, newProcessList,
            newResourceList);

        // Clean up
        free(newResourceList);
        free(newProcessList);

        return newModel;
}

////////////////////////////////////////////////////////////////////////////////
//  g e n e r a t e A n a l y s i s R e s u l t s
//
//  Description:
//
//  Generate a report of the results of the analysis.
//
void
TPrographAnalyzer::generateAnalysisResults(BOOL thereIsACannotReduce,
    TGraphShapeList *freedProcessList, char *analysisResultsStr)
{
    TGraphShape *process;
    char shapeName[MAX_STR_LENGTH];

    // If the results of the analysis exceeds MAX_ANALYSIS...STR_LENGTH
    // then the program will core dump. I am not expecting such a long
    // result. It can happen only if there are hundreds of nodes and edges in
    // the model to be analyzed

    strcpy(analysisResultsStr, "");

    switch (thereIsACannotReduce)
    {
    case TRUE:
        strcat(analysisResultsStr, "The State is *DEADLOCKED* or *UNSAFE*");
        break;
    case FALSE:
        strcat(analysisResultsStr, "The State is *SAFE*");
        break;
    }

    if (freedProcessList->size() != 0)
    {
        strcat(analysisResultsStr, "\n\n");
        strcat(analysisResultsStr, "Reduction Sequence ");

        process = freedProcessList->first();
        while (process != NULL)
        {
            process->getShapeName(shapeName);
            strcat(analysisResultsStr, " - ");
            strcat(analysisResultsStr, shapeName);
            process = freedProcessList->next();
        }
    }

    strcat(analysisResultsStr, "\n\n");

}

////////////////////////////////////////////////////////////////////////////////
//  a d d A l l E d g e s T o M o d e l E x c e p t
//
//  Description:
//
//  This function is part of generating a new graph. To the new graph, this
//  function adds the edges connected to all processes except those of the
//  processes passed as an argument in the freedProcessList list.
//
void
TPrographAnalyzer::addAllEdgesToModelExcept(
    TGraphShapeList *freedProcessList, TPrographModel *newModel,
    TGraphShapeList *newProcessList, TGraphShapeList *newResourceList)
{
    int process;
    int resource;
    TEdge *edgeShape;
    TEdge *newEdgeShape;
    TGraphShape *newSourceShape;
    TGraphShape *newDestShape;

    // Construct the newEdgeList which will consist of only edges that
    // do not have any process in the freedProcessList as an end point shape
    // i.e., theoretically the freed processes are giving up the resources
    // allocated to them or giving up the resource requests they made, or
    // if the freed processes are producers, removing their producer edges
    //
    edgeShape = (TEdge *) _edgeList->first();
    while (edgeShape != NULL)
    {
        // If any of the end points of the edge are one among the freed
        // processes, then we should not add that edge to the new model,
        // hence we continue
        //
        if ((freedProcessList->isMember(edgeShape->source())) ||
            (freedProcessList->isMember(edgeShape->destination())))
        {
            edgeShape = (TEdge *) _edgeList->next();    // Duplication
            continue;
        }

        // Find the clone shapes of the source and dest, so that we can
        // be up to date
        //
        if ((edgeShape->source())->shapeType() == SHAPE_PROCESS)
        {
            process = _processList->getIndex(edgeShape->source());
            newSourceShape = newProcessList->getShapeAtIndex(process);
            resource = _resourceList->getIndex(edgeShape->destination());
            newDestShape = newResourceList->getShapeAtIndex(resource);
        }
        else if ((edgeShape->source())->shapeType() == SHAPE_RESOURCE)
        {
            resource = _resourceList->getIndex(edgeShape->source());
            newSourceShape = newResourceList->getShapeAtIndex(resource);
            process = _processList->getIndex(edgeShape->destination());
            newDestShape = newProcessList->getShapeAtIndex(process);
        }
        else
            assert (0);

        // Now clone the edge shape and add the clone to the
        // newModel
        //
        newEdgeShape = (TEdge *) edgeShape->clone(newSourceShape, newDestShape);

        // FALSE in the next statement signifies not to update the resource
        // shape numUnits *display* (of course the numUnits will be updated)
        //
        newModel->addEdge(newEdgeShape, newSourceShape, newDestShape, FALSE);

        edgeShape = (TEdge *) _edgeList->next();
    }

}

//////////////////////////////////////////////////////////////////////////////
//  i n i t i a l i z e I n t e r n a l S t a t e
//
//  Description:
//
//  Initializes all the internal matrices and vectors.
//
void
TPrographAnalyzer::initializeInternalState(TPrographModel *aModel)
{
    int edge;
    int process;
    int resource;
    TGraphShape *sourceShape;
    TGraphShape *destShape;
    TProcess *processShape;
    TResource *resourceShape;
    TEdge *edgeShape;

    _processList = aModel->processList();
    _resourceList = aModel->resourceList();
    _edgeList = aModel->edgeList();

    _numProcesses = _processList->size();
    _numResources = _resourceList->size();
    _numEdges = _edgeList->size();

    _allocMatrix = allocateIntegerMatrix(_numProcesses, _numResources);
    _requestMatrix = allocateIntegerMatrix(_numProcesses, _numResources);
    _producerMatrix = allocateIntegerMatrix(_numProcesses, _numResources);

    _availVector = allocateIntegerVector(_numResources);
    _totalVector = allocateIntegerVector(_numResources);
    _typeVector = allocateResourceTypeVector(_numResources);

    // Allocate duplicate matrices
    _orgAllocMatrix = allocateIntegerMatrix(_numProcesses, _numResources);
    _orgRequestMatrix = allocateIntegerMatrix(_numProcesses, _numResources);
    _orgProducerMatrix = allocateIntegerMatrix(_numProcesses, _numResources);

    _orgAvailVector = allocateIntegerVector(_numResources);
    _orgTotalVector = allocateIntegerVector(_numResources);

    for (resource = 0; resource < _numResources; resource++)
    {
            resourceShape = (TResource *) _resourceList->getShapeAtIndex(resource);
            _availVector[resource] = resourceShape->getNumAvailableResources();
            _totalVector[resource] = resourceShape->getNumTotalResources();
            _typeVector[resource] = resourceShape->resourceType();
        }

        // Now fill the matrices and Vector
        for (edge = 0; edge < _numEdges; edge++)
        {
            edgeShape = (TEdge *) _edgeList->getShapeAtIndex(edge);
            sourceShape = edgeShape->source();
            destShape = edgeShape->destination();

            if (sourceShape->shapeType() == SHAPE_PROCESS)
            {
                processShape = (TProcess *) sourceShape;
                resourceShape = (TResource *) destShape;

                process = _processList->getIndex(processShape);
                resource = _resourceList->getIndex(resourceShape);
                _requestMatrix[process][resource]++;
            }
            else if (sourceShape->shapeType() == SHAPE_RESOURCE)
            {
                resourceShape = (TResource *) sourceShape;
                processShape = (TProcess *) destShape;

                resource = _resourceList->getIndex(resourceShape);
                process = _processList->getIndex(processShape);

                switch (_typeVector[resource])
                {
                    case RESOURCE_SERIAL:
                        _allocMatrix[process][resource]++;
                        break;

                    case RESOURCE_CONSUMABLE:
                        _producerMatrix[process][resource]++;
                        break;
                }
            }
        }

        // Now save the original state in the duplicate matrices
        copyMatrix(_allocMatrix, _orgAllocMatrix, _numProcesses, _numResources);
        copyMatrix(_requestMatrix, _orgRequestMatrix, _numProcesses, _numResources);
        copyMatrix(_producerMatrix, _orgProducerMatrix, _numProcesses,
                                                                 _numResources);
        copyVector(_availVector, _orgAvailVector, _numResources);
        copyVector(_totalVector, _orgTotalVector, _numResources);
}

////////////////////////////////////////////////////////////////////////////////
//  a d d V e c t o r
//
//  Description:
//
//  This function adds two vectors. Since the vectors can contain an
//  INFINITE value as one of its elements, we have to take specifically
//  take care of that.
//
void
TPrographAnalyzer::addVector(int *vector, int *vectorToBeAdded, int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        if ((vectorToBeAdded[i] == INFINITY) || (vector[i] == INFINITY))
            vector[i] = INFINITY;
        else
            vector[i] = vector[i] + vectorToBeAdded[i];
    }
}

////////////////////////////////////////////////////////////////////////////////
//  s u b t r a c t V e c t o r
//
//  Description:
//
//  Subtract two vectors, keeping in mind that the vectors can have INFINITY
//  as one of its elements.
//
void
TPrographAnalyzer::subtractVector(int *vector, int *vectorToBeSubtracted,
    int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        // The vectorToBeSubtracted will be a request vector
        //
        assert(vectorToBeSubtracted[i] != INFINITY);
        if (vector[i] == INFINITY)
            vector[i] = INFINITY;
        else
        {
            vector[i] = vector[i] - vectorToBeSubtracted[i];
            assert (vector[i] >= 0);    // Should not go negative. if it does
                                        // then there's a bug in the program
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
//  a d d P r o d u c e r V e c t o r
//
//  Description:
//
//  Called to add a vector in the producer matrix to a general resource
//  vector.
//
void
TPrographAnalyzer::addProducerVector(int *vector, int *producerVector, int size)
{
    int i;

    // Each element of the producerVector denotes the number of producer edges
    // particular process might be having with a particular resource
    // If it is at least 1, then we set the resulting vector element to
    // INFINITY, denoting that the process released the resource, and thus
    // resulted in an INFINITE number of resources being placed in the resource
    // Note - this addition cannot be called an addition in the true sense of
    // the term
    //
    for (i = 0; i < size; i++)
    {
        if (producerVector[i] > 0)
            vector[i] = INFINITY;
        else
            vector[i] = vector[i];
    }
}
```

```
////////////////////////////////////////////////////////////////////////////
// i s L e s s T h a n O r E q u a l T o
//
// Description:
//
// Compare two vectors for <=
//
BOOL
TPrographAnalyzer::isLessThanOrEqualTo(int *lhsVector, int *rhsVector, int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        // lhsVector will be a request vector - No term of it cannot be infinity
        //
        assert (lhsVector[i] != INFINITY);

        if (rhsVector[i] == INFINITY)    // Of course rhs is greater
            continue;

        if (lhsVector[i] > rhsVector[i])    // Aha!, lhs is greater, so NO!!!
            return FALSE;
    }

    // If we reach here, then all lhs is less than or equal to all of rhs
    //
    return TRUE;
}

////////////////////////////////////////////////////////////////////////////
// a l l o c a t e R e s o u r c e T y p e V e c t o r
//
// Description:
//
// Allocate memory for a Resource vector
//
ResourceType *
TPrographAnalyzer::allocateResourceTypeVector(int size)
{
    ResourceType *newVector;

    newVector = (ResourceType *)malloc(size * sizeof(ResourceType));
    assert (newVector != NULL);

    return newVector;
}

////////////////////////////////////////////////////////////////////////////
// p r i n t I n t e r n a l S t a t e
//
// Description:
//
// Print the internal state of the analyzer (Used for debugging)
//
void
TPrographAnalyzer::printInternalState()
{
    printf("\n\n");
    printf("*****************************************************************\n");
    printf("Num Processes = %d\n", _numProcesses);
    printf("Num Resources = %d\n", _numResources);
    printf("Num Edges = %d\n", _numEdges);

    printf("Alloc matrix\n");
    printMatrix(_allocMatrix, _numProcesses, _numResources);
    printf("Request matrix\n");
    printMatrix(_requestMatrix, _numProcesses, _numResources);

    printf("Producer matrix\n");
    printMatrix(_producerMatrix, _numProcesses, _numResources);

    printf("Avail vector");
    printVector(_availVector, _numResources);
}

////////////////////////////////////////////////////////////////////////////////
//  Other miscellaneous functions
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
//  g e n e r a t e A l l C o m b i n a t i o n s
//
//  Description:
//
//  Given a particular maximum value (denoted by setSize), generate all
//  combinations possible for number below that value.
//  Say for 3 processes in the system, the combinations generated are
//  0-1-2, 0-2-1, 1-2-0, 1-0-2, 2-1-0, 2-0-1
//
int **
generateAllCombinations(int setSize, int &numCombs)
{
    int **combs;

    numCombs = factorial(setSize);
    combs = allocateIntegerMatrix(numCombs, setSize);
    generateCombinations(numCombs, setSize, combs);
    return combs;
}

////////////////////////////////////////////////////////////////////////////////
//  g e n e r a t e S p e c i a l C o m b i n a t i o n s
//
//  Description:
//
//  Generate combinations with a specific number as its first element.
//  For a setSize of 3, and the initial element 2, the combinations
//  generated are 2-0-1 and 2-1-0
//
int **
generateSpecialCombinations(int specialNumber, int setSize,
    int &specialNumCombs)
{
    int i;
    int **combs;
    int **specialCombs;
    int numCombs;
    int combsFilled;

    // We generate the combinations having only the specialNumber as the
    // first element of the combination. This is to support reduction by a
    // particular process

    numCombs = factorial(setSize);
    specialNumCombs = numCombs / setSize;
    combs = allocateIntegerMatrix(numCombs, setSize);
    specialCombs = allocateIntegerMatrix(specialNumCombs, setSize);

    generateCombinations(numCombs, setSize, combs);
    combsFilled = 0;
    for (i = 0; i < numCombs; i++)
    {
        if (combs[i][0] == specialNumber)
        {
            copyVector(combs[i], specialCombs[combsFilled], setSize);
            combsFilled++;
        }
    }
    assert (specialNumCombs == combsFilled);

    free (combs);
    return specialCombs;
}

////////////////////////////////////////////////////////////////////////////////
//  g e n e r a t e C o m b i n a t i o n s
//
//  Description:
//
//  The core routine to generate the combinations.
//
void
generateCombinations(int numCombs, int setSize, int **combs)
{
    int i, j, maxPerms;
    int combsFilled;
    int *perm;

    perm = allocateIntegerVector(setSize);
    maxPerms = power(setSize, setSize);
    combsFilled = 0;

    for (i = 0; i < maxPerms; i++)
    {
        incrementPermutation(setSize, perm);
        if (isCombination(setSize, perm) == TRUE)
        {
            for (j = 0; j < setSize; j++)
                combs[combsFilled][j] = perm[j];

            combsFilled++;
        }
    }

    assert (numCombs == combsFilled);
    free (perm);
}

////////////////////////////////////////////////////////////////////////////////
//  i s C o m b i n a t i o n
//
//  Description:
//
//  Check if whether a set of numbers is a combination (or is it a permutation)
//
BOOL
isCombination(int setSize, int *perm)
{
    int i, j;
    int num;

    for (i = 0; i < setSize; i++)
    {
        num = perm[i];
        for (j = 0; j < setSize; j++)
        {
            if (i == j)
                continue;
            if (perm[j] == num)
                return FALSE;
        }
    }

    return TRUE;
}

////////////////////////////////////////////////////////////////////////////////
//  i n c r e m e n t P e r m u t a t i o n
//
//  Description:
//
//  Increment the permutation and get the next permutation in the sequence
//
void
incrementPermutation(int setSize, int *perm)
{
    int i;

    for (i = (setSize - 1); i >= 0; i--)
    {
        if (perm[i] < (setSize - 1))
        {
            perm[i] = perm[i] + 1;
            return;
        }

        perm[i] = 0;
    }
}

////////////////////////////////////////////////////////////////////////////////
//  p o w e r
//
//  Description:
//
//  Find x^y
//
int
power(int x, int y)
{
    int i, pow;

    pow = 1;
    for (i = 0; i < y; i++)
        pow = pow * x;

    return pow;
}

////////////////////////////////////////////////////////////////////////////////
//  f a c t o r i a l
//
//  Description:
//
//  Find the factorial of a number
//
int
factorial(int x)
{
    int fact;

    if (x == 0)
        return 1;

    fact = x * factorial(x - 1);
    return fact;
}

////////////////////////////////////////////////////////////////////////////
//  i s Z e r o V e c t o r
//
//  Description:
//
//  Ascertain if a vector consists of all zero elements
//
BOOL
isZeroVector(int *vector, int size)
{
    int i;

    for (i = 0; i < size; i++)
    {
        if (vector[i] != 0)
            return FALSE;
    }

    // If we reach here then all the elements in the vector were zeros. i.e.,
    // the condition check could not find any non zero element, so this
    // vector consists of all zeros
    //
    return TRUE;
}

////////////////////////////////////////////////////////////////////////////
//  s e t T o Z e r o
//
//  Description:
//
//  Set all the elements of a vector to zero
//
void
setToZero(int *vector, int size)
{
    int i;

    for (i = 0; i < size; i++)
        vector[i] = 0;
}

////////////////////////////////////////////////////////////////////////////
//  a l l o c a t e I n t e g e r M a t r i x
//
//  Description:
//
//  Allocate a matrix of integer elements
//
int **
allocateIntegerMatrix(int numRows, int numColumns)
{
    int i, j;
    int **newMatrix;

    newMatrix = (int **) malloc(numRows * sizeof(int *));
    assert (newMatrix != NULL);

    for (i = 0; i < numRows; i++)
    {
        newMatrix[i] = (int *) malloc(numColumns * sizeof(int));
        assert (newMatrix[i] != NULL);
    }

    // Zero it out
    for (i = 0; i < numRows; i++)
        for (j = 0; j < numColumns; j++)
            newMatrix[i][j] = 0;

    return newMatrix;
}

////////////////////////////////////////////////////////////////////////////////
//  f r e e I n t e g e r M a t r i x
//
//  Description:
//
//  Free a matrix containing integer elements
//
void
freeIntegerMatrix(int **matrix, int numRows)
{
    int i;

    for (i = 0; i < numRows; i++)
        free (matrix[i]);

    free(matrix);
}

////////////////////////////////////////////////////////////////////////////////
//  a l l o c a t e I n t e g e r V e c t o r
//
//  Description:
//
//  Allocate a vector of integer elements
//
int *
allocateIntegerVector(int size)
{
    int i;
    int *newVector;

    newVector = (int *)malloc(size * sizeof(int));
    assert (newVector != NULL);

    // Zero out the vector
    for (i = 0; i < size; i++)
        newVector[i] = 0;

    return newVector;
}

////////////////////////////////////////////////////////////////////////////////
//  c o p y M a t r i x
//
//  Description:
//
//  Copy function for a matrix
//
void
copyMatrix(int **srcMatrix, int **destMatrix, int numRows, int numColumns)
{
    int i, j;

    for (i = 0; i < numRows; i++)
        for (j = 0; j < numColumns; j++)
            destMatrix[i][j] = srcMatrix[i][j];
}

////////////////////////////////////////////////////////////////////////////////
//  c o p y V e c t o r
```

```
//
// Description:
//
// Copy function for a vector
//
void
copyVector(int *srcVector, int *destVector, int size)
{
    int i;

    for (i = 0; i < size; i++)
        destVector[i] = srcVector[i];
}

////////////////////////////////////////////////////////////////////////////////
// p r i n t M a t r i x
//
// Description:
//
// Print out a matrix (for debugging)
//
void
printMatrix(int **matrix, int numRows, int numColumns)
{
    int i, j;

    for (i = 0; i < numRows; i++)
    {
        for (j = 0; j < numColumns; j++)
            printf("%d\t", matrix[i][j]);
        printf("\n");
    }
    printf("\n");
}

////////////////////////////////////////////////////////////////////////////////
// p r i n t V e c t o r
//
// Description:
//
// Print out a vector (for debugging)
//
void
printVector(int *vector, int size)
{
    int i;

    printf("\n");
    for (i = 0; i < size; i++)
        printf("%d\t", vector[i]);
    printf("\n");
}
```

VITA

Bobby Stephen Koshy

Candidate for the Degree of Master of Science

Thesis: TOWARDS A GRAPHICAL DEADLOCK ANALYSIS TOOL

Major Field: Computer Science

Biographical:

Personal Data: Born in Kolenchery, Kerala, India, October 6, 1970, son of Thavalathil Kocheepen Koshy and Mary Koshy.

Education: Graduated from Christ College, Irinjalakuda, Kerala, India, in June 1988; received Bachelor of Technology (Honors) Degree in Computer Engineering from Regional Engineering College, Calicut, Kerala, India, in June 1992; completed the requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in December 1995.

Professional Experience: Software Engineer, Wipro Infotech Ltd., Bangalore, India, July 1992 to July 1993; Programmer, Department of Biosystems and Agricultural Engineering, Oklahoma State University, September 1993 to December 1993; Graduate Research Assistant, Department of Biosystems and Agricultural Engineering, Oklahoma State University, January 1994 to August 1995.