PADS: A GRAPHICAL INTERFACE FOR SOFTWARE SYSTEMS MODELING

Dana Mark Whiting

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-17 May 1987

To my family
(past, present, and future).

PADS: A GRAPHICAL INTERFACE FOR SOFTWARE SYSTEMS MODELING

BY

DANA MARK WHITING, B.A.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

May, 1987

ACKNOWLEDGEMENTS

My thanks go to Dr. James C. Browne for proposing this thesis topic and supervising its development, to Dr. Douglas M. Neuse for his continuous guidance and numerous helpful recommendations, to Jim Dutton for his development and maintenance of GPSM, and to all the employees of Information Research Associates, especially Mohan Rao and Peter Newton, who helped me work with PAWS and with the various computer systems at IRA. Special thanks go to my loving wife, Sherri, for her moral support and encouragement while I worked on this project through our first year of marriage.

This work was funded in part by the Navy, under SBIR contract number N60921-86-C-0145. The final draft of this thesis was submitted to the Committee on April 9, 1987.

Dana Mark Whiting

The University of Texas at Austin
May, 1987

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

Performance deficiencies have been the single largest cause for the redesign and reimplementation of large software systems. This problem results from a lack of technology for determining the performance of a computer system while it is being designed. Traditionally, performance problems have been detected only when the system enters integration testing or production. By that time, the enormous amounts of time, money, and personnel resources invested in the development and implementation of the programs make them extremely costly to redesign and reimplement.

This thesis describes a method for facilitating the performance evaluation of software systems at design time, which is when design modifications are the easiest and least expensive to implement. The approach is to create a natural, graphical interface for the specification of software systems, and then map this graphical representation onto a simulation model of a computer system, or set of hardware devices. This hardware model then executes a workload that is specified in terms of a software systems definition. The result is a modified performance evaluation package that provides additional capabilities for determining the properties of software systems at the design level.

1.2 Background

For the past six years, the Performance Analyst's Workbench System (PAWS) has been one of the best languages available for simulation modeling of queueing network systems. PAWS provides for the description and performance evaluation of Information Processing Graphs, or IPGs, which are pictorial, directed-graph representations of queueing network systems. It contains many high-level primitive functions pertaining to memory resource management, queueing disciplines, probability distributions, and output statistics, making it ideal for simulating many computer systems [12]. PAWS has been used successfully to model disk subsystems on the Sperry Univac 1100 [13], a retail point-of-sale system [2], a CDC CYBER disk system [6], packet-switched interprocessor communications [22], and many other research and commercial projects.

Recently, a graphical interface to PAWS, known as Graphical Programming of Simulation Models, or GPSM, has been developed to provide an automated translation from IPGs to the equivalent PAWS models. The GPSM system is a tool that allows IPGs to be drawn and modified directly on the graphics screen of any IBM-PC compatible machine, using a mouse as a pointing device, and then automatically translated into simulation programs in the PAWS language. (Currently, the graphical interface is being implemented on other systems as well, such as the Sun and MicroVax workstations.) Thus, GPSM enables modelers to deal directly with the pictorial information in the IPGs in order to design, execute, and refine their simulations [7].

Both PAWS and GPSM represent a hardware system model as

a set of directed graphs. The arcs represent the paths that transactions may take as they flow from node to node, where a transaction may be thought of as a job or task in a computer system. The nodes represent real resources, such as CPUs, disk controllers, and memory and token allocations, or other operations on the transactions in the system, such as altering the variables local to a transaction, creating new transactions, or destroying existing ones. Thus, modeling hardware system execution behavior in PAWS or GPSM is fairly straightforward.

Software system behavior may also be modeled with PAWS or GPSM. However, this modeling is made somewhat more difficult by the fact that both packages integrate the description of execution behavior with the definition of devices. In other words, the usage of a given resource by each transaction is part of the description of that resource, and the path taken by each transaction along the arcs connecting the resources is contained in the definitions of the arcs. Therefore, any patterns of resource usage defined in the virtual device representation (or software) must be mapped manually by the user to usage patterns of real devices (or hardware) [11, p. 3].

To provide an appropriate interface for software modeling in PAWS and GPSM, a new package named Performance Analysis for Designers of Software, or PADS, is being developed. PADS provides the user with a graphical interface similar to that used in GPSM. Using the mouse as a pointing device, the modeler may draw, edit, store, and retrieve graphs of various types that represent physical devices and the software load placed upon them. These graphs are then automatically translated into a program file, written in the PAWS simulation language, and an auxiliary file of routing instructions that can be interpreted by the PAWS program during execution of model simulation.

Although a systems analyst could use GPSM or PAWS to construct models of software system execution behavior directly, the interface presented by PADS gives it a natural advantage over either package for software modeling applications. This makes it more appealing to the software user community.

1.3 Overview of Methodology

When the user invokes the PADS program, the computer will enter the PADS graphics mode, in which the user's actions are initiated by selecting an item from a menu. This selection is accomplished by moving a cursor, or "tool", onto the menu with the mouse, and clicking the mouse when the tool is on the desired portion of the menu. The user may create, edit, or delete files, graphs, nodes, and arcs, depending on the specific tool and object selected. In addition, the user enters specific information about each object from the keyboard; in this way, the objects that make up the set of graphs are annotated and defined. (As used in this thesis, "annotating" refers more precisely to "specifying" the properties of an object by filling out the appropriate fields of its windows.) When the PADS file containing graphs, nodes and arcs has been completely defined, the user then selects the "translate" option to automatically create the PAWS program file and auxiliary file of transaction routing and resource usage information. Together these form an executable version of the model represented in the PADS file, which may then be edited, re-translated and re-executed if the user desires to further refine or alter the model.

An overview of the components of a PADS model and of the PADS modeling methodology is shown in Figure 1-1.

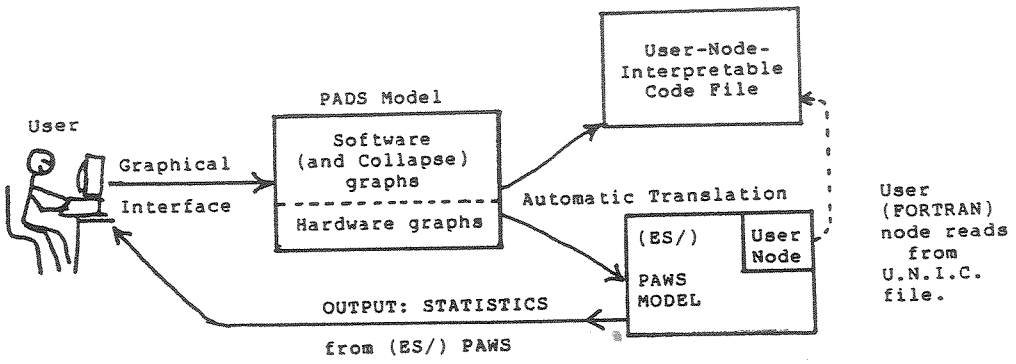


Figure 1-1: Overview of the PADS Modeling Methodology

The PAWS program file is derived from the translation of the set of hardware graphs, which are collections of PAWS resource nodes with no connecting arcs. Thus, it has incomplete information about transaction routing and resource usage. It describes a set of PAWS resource nodes connected bi-directionally in a star topology to a central routing node, which is referred to in PAWS as a USER node. (In PAWS, a USER node allows the modeler to call an external FORTRAN subroutine or set of subroutines, as described in [12].)

The USER node has been implemented in FORTRAN as part of the PADS system. It is the portion of the PAWS program that is able to interpret the transaction routing and resource usage information in the auxiliary file, which is the result of translating the software and collapse graphs. This auxiliary file is referred to in PADS as the User-Node-Interpretable Code, or UNIC.

The software graphs are directed graphs, where each node

represents a portion of software code, and the arcs between nodes represent execution control such as branching and looping. Each node may also contain branching and looping. and software graphs may call each other to an arbitrary degree of nesting or recursion, subject to the size of PADS' run-time and parameter stacks. Therefore, software models may be structured in a hierarchical manner and to any degree of detail.

The collapse graphs provide an elementary and experimental method of model simplification, subject to very narrow constraints. Hierarchical structuring of software models has lessened the need for collapse graphs for model simplification. although they may still be useful for efficiency during the actual simulation, since they could diminish the amount of overhead required by an equivalent number of individual instructions being executed. Collapse graphs are described in more detail in Section 3.5 of this thesis.

1.4 Overview of Related Work

Methods for performance analysis of software systems have been extensively described and discussed during the past decade.

Connie Smith has provided a number of papers on this topic. Her Ph.D. dissertation [16] describes the implementation of ADEPT (A Design-based Evaluation and Prediction Technique). This technique requires definition of the performance goals of a system. the execution environment, the system structure, etc., which are then mapped onto execution paths, which then make it possible to specify the resource requirements and to analyze the resulting system performance. Initially, the analysis uses only the most optimistic or "best-case" projections,

since poor results for the best case are an indisputable indication of a need to change the overall design or to supply better system resources. This technique was used to predict poor performance in the IPAD system in the early stages of its development [18], as well as to verify poor design in the TENEX operating system after it had been completed [15].

Further discussion by Smith of the use of performance goals in software design, of ADEPT and of performance engineering techniques in general is found in [17], which stresses the importance of performance in determining the quality of software systems. A more recent overview of software performance engineering tools and graphical interfaces is in [19], which includes a list of features that tools should have in order to effectively support analysis of software systems. Smith also provides a general overview of the approaches to software performance engineering from the 1960's to the present, as well as suggested future trends, in [20].

Another discussion of systematic design is found in [3], where the authors present an overall methodology for evaluating the performance of computer systems: start with a broad model and refine it as needed. The paper shows how modeling languages such as PAWS can be used with this methodology.

A tool for developing inputs to performance models is described in [5]. This tool requires the software to be already implemented in machine language (or compiled). The modeling technique involves the analysis of "threads" and "paths" through the code.

The current PADS system is very similar in operation to one described by Ed Upchurch in [23]. The system to be modeled was described in that paper as follows:

> It was noticed that the system is so large that hundreds of IPG nodes would be required if each function were implemented separately. A simple IPG was then implemented driven by a table defining each system function graph in terms of system primitive operations....
>
> Entries in the table include: line number, CPU time required, number of times to perform the operation (loops for disk I/O), memory requirements for those memory requests...and a pointer to the next table line to process....
>
> The USER node represents a user-written FORTRAN subroutine that processes the current table entry and sets the routing code (phase) for the next cycle through the IPG. Local variables attached by PAWS to each individual transaction are used to save the current pointer to the table and to save nested macro calls in a small stack. When the transaction leaves the USER node it will be directed to one of the primitive functions: allocate, release, fork or join.... [23]

The last paragraph quoted could be applied to a description of PADS as well. Originally, an attempt was made to use a PAWS COMPUTE node in place of a USER node (to avoid modifying the FORTRAN code of PAWS) and to store the code for the "table" mentioned above in a transaction's local variables. (A COMPUTE node is described in [12] as an "arithmetic" node used to carry out computational steps and modify the variables of a model without requiring the use of an external user-written FORTRAN subroutine.) This approach proved impractical, for several reasons. One was that each transaction has a limited number of local variables available to it, and could not hold enough instructions to perform even a tiny model. Another was the limited amount of space

available for COMPUTE node instructions in a PAWS model. Still another was the fact that COMPUTE node instructions are interpreted by PAWS at run time, a process that is much slower than the execution of FORTRAN code called by a USER node. For these reasons, the decision was made to implement the translation of software graphs as a file of instructions that could be interpreted by a special USER node in order to direct transactions to the appropriate resources.

This approach was also seen as more straightforward than the one studied by Keh-Chiang Yu and described in [8], that of attempting a mapping from Extended Execution Graphs (EEGs) to IPGs. However, a small subset of that approach is found in the implementation of PADS collapse graphs, which may later be expanded or omitted entirely in future implementations of PADS. Although the topic of this thesis was originally inspired by the work of Yu, Connie Smith [16], John Kelly [9] and others, the final product bears little resemblance to the approaches they took. An extensive listing of other works and approaches to the same topic may be found in [21].

The remaining chapters of this thesis present a more detailed description of the visual language interface, an example of a software system modeled with PADS, and the effect of proposed changes to PAWS and GPSM on future versions of PADS. The appendix contains the user's manual for the current version of PADS.

Chapter 2

Visual Language Interface

This chapter outlines the methodology for the creation and alteration of software models using the PADS graphical interface. This methodology and interface are basically the same as those used in GPSM: additional information about them are in [7]. A good overview of the subjects mentioned in this chapter may be found in [11]. The graphical portion of PADS currently runs on the IBM-PC and compatible machines, using any one of the mouse devices and graphics cards listed in [7]. It is also being implemented on the Sun and MicroVax workstations.

2.1 Menus

Upon invoking the PADS program, the user is presented with a menu listing the names of all previously created PADS files in his directory, as shown in Figure 2-1. From this menu, he may choose to edit an existing file, or to create and edit a new one.

After a file to be edited has been chosen, the PADS banner appears at the top of the screen, showing the names of the current file, graph, and tool (also known as the cursor). The banner acts as a sort

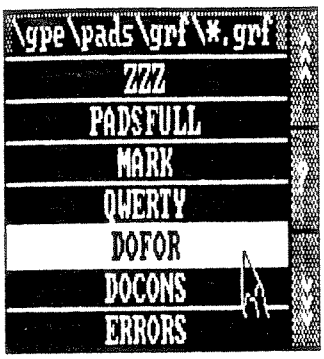


Figure 2-1: Menu of Existing PADS Models.

of "meta-menu" which allows other menus to be selected; for example, clicking the mouse when the cursor is on the "Tool" portion of the banner opens the menu for selecting a different tool, and clicking it on the "Graph" portion of the banner opens one of several menus for performing operations on graphs, depending on the current tool.

In addition, each tool has a menu associated with it. For example, the "Erase" tool, when used to select a node, arc, or graph, has a simple "confirmation" menu as shown in Figure 2-2, from which the user may change his decision to delete an object. Furthermore, the Draw tool, when clicked on an existing node, displays a "secondary" menu to allow that node to be changed to another node or to change its orientation in the graph, as shown in Figure 2-3. Figure 2-4 shows a list of the tools available, which are discussed further in Section 2.4.

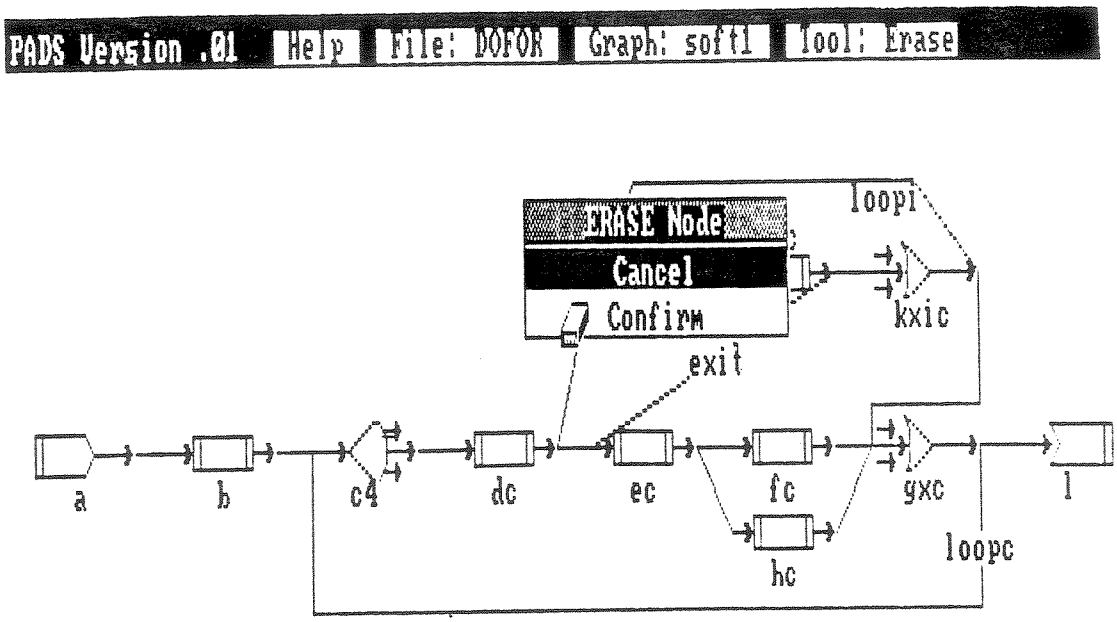


Figure 2-2: A Confirmation Menu

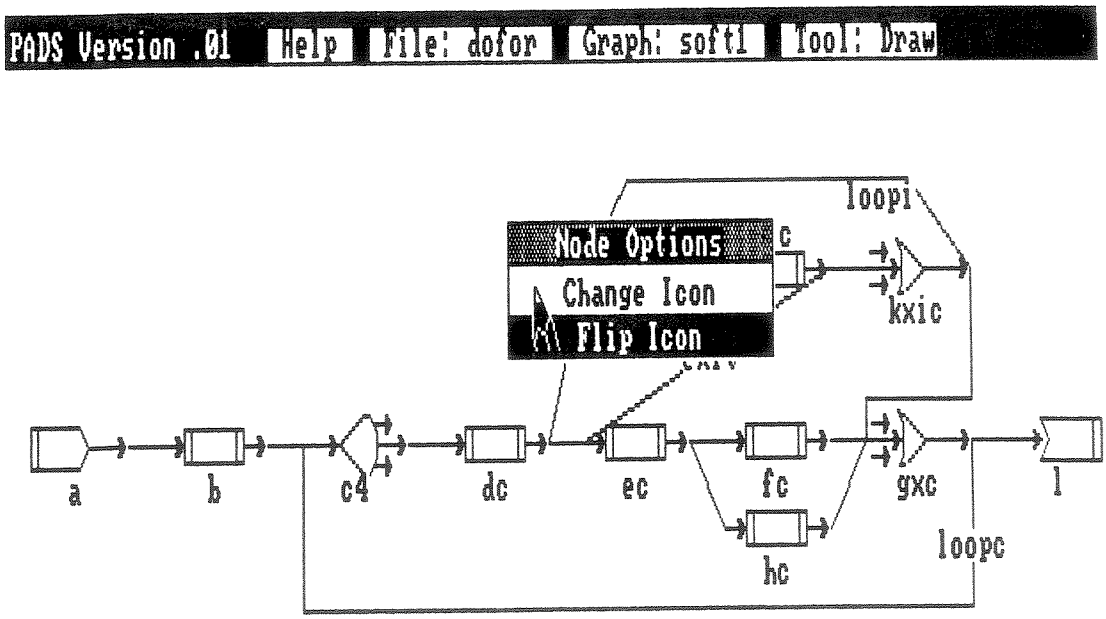


Figure 2-3: A Secondary Menu

Help menus are available to the user, either through clicking the "Help" tool on the object about which more information is desired,

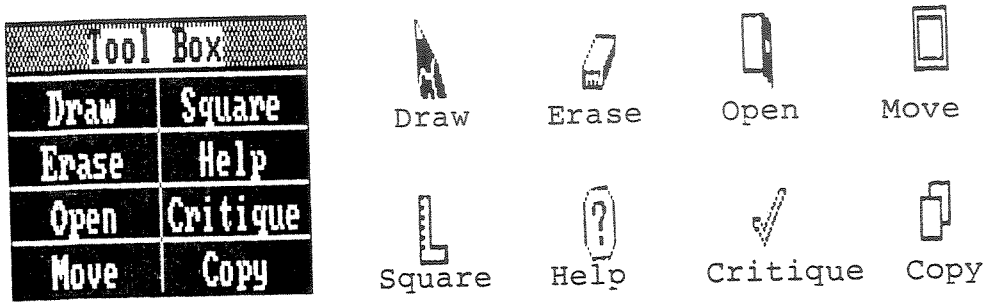


Figure 2-4: The Tool Box Menu and Associated Cursors

or through the use of the function keys to list the correct syntax for entering the definitions of objects.

Finally, there is an "Icon" menu that appears when the "Draw" tool is clicked on an empty portion of the screen, as shown in Figure 2-5. This is used to place the appropriate components (or "nodes") of the graphs on the screen, and is described further in the next section.

2.2 Icons

Each type of graph (hardware, software, and collapse) has its own set of icons, which represent the nodes of a graph. The icons are chosen from a menu, or palette, that appears on the screen when the "Draw" tool is clicked on an empty portion of the screen. Selecting an icon causes a copy of that icon to appear at the place on the screen (i.e., in the graph) where the tool was clicked.

The icons for PADS hardware graphs are, for the most part, identical to those currently used in GPSM graphs, as shown in Figure 2-6. However, the icons for Fork, Join, Split, Submodel Entry (ENTER),

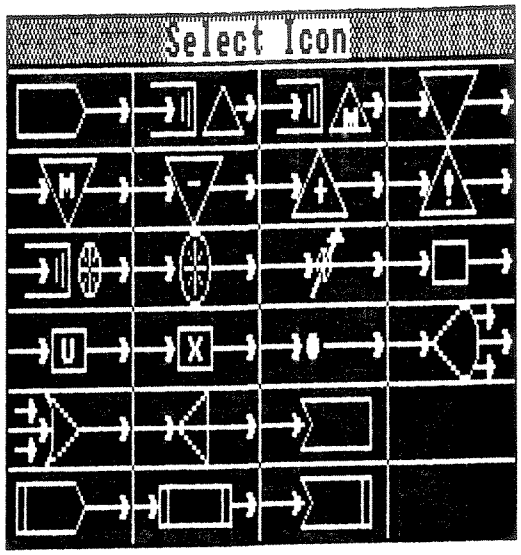


Figure 2-5: The Icon Menu

Submodel Call (CALL), and Submodel Exit (RETURN) are not used in PADS hardware graphs, since those icons do not correspond to actual individual hardware resources. The user gives each hardware node a unique name, which is used throughout the model (including the software graphs) to refer to the resources they represent. The definitions of hardware nodes are written using a syntax very similar to that currently used for defining nodes in GPSM; this facilitates the translation of the PADS model into PAWS. A sample hardware graph is shown in Figure 2-7.

PADS software graphs require a relatively small set of icons. Each software graph has exactly one ENTER and one RETURN node; these look like the icons of the same name currently used in GPSM. There is also a LOOP node (also known as a FOR node), signifying the beginning of a FOR-loop, and an ENDLOOP (or ENDFOR) node, which

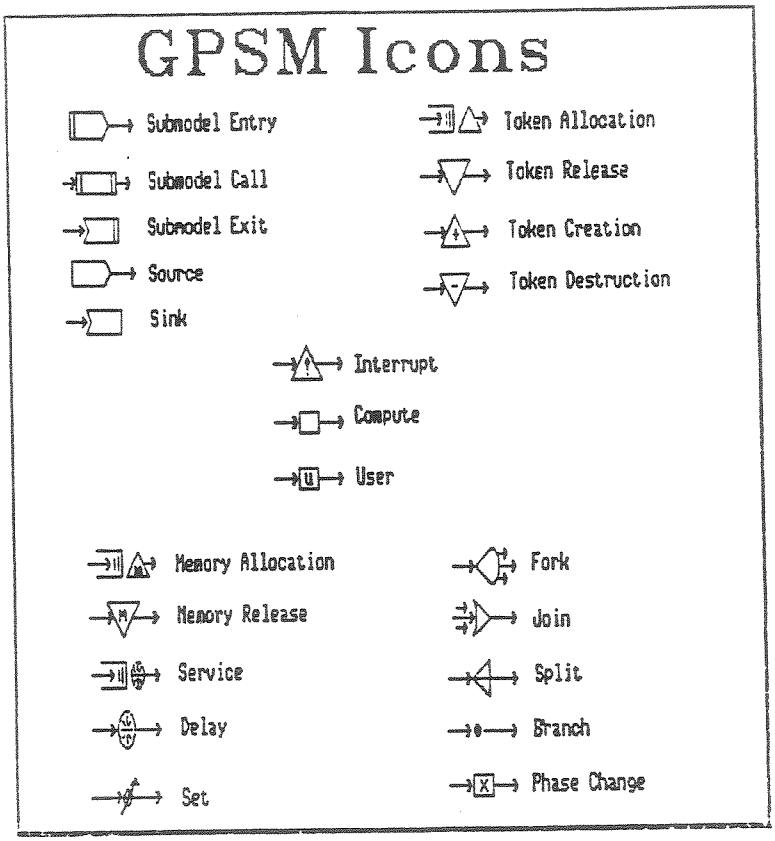


Figure 2-6: GPSM Node Icon Summary

signifies the end of a FOR-loop. Finally, there is a CODE node, which looks like a CALL node in GPSM, and holds the software instructions in the PADS language corresponding to the software portion of the system being modeled.

The current version of PADS reuses the FORK node of GPSM as a FOR node, and the JOIN node as an ENDFOR node. However, these nodes should be redesigned in future versions of PADS, especially if those versions are to include the ability to model parallelism, for which FORK and JOIN are essential.

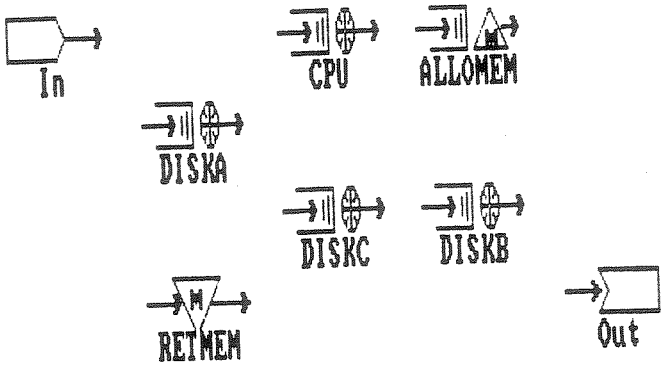


Figure 2-7: A PADS Hardware Graph

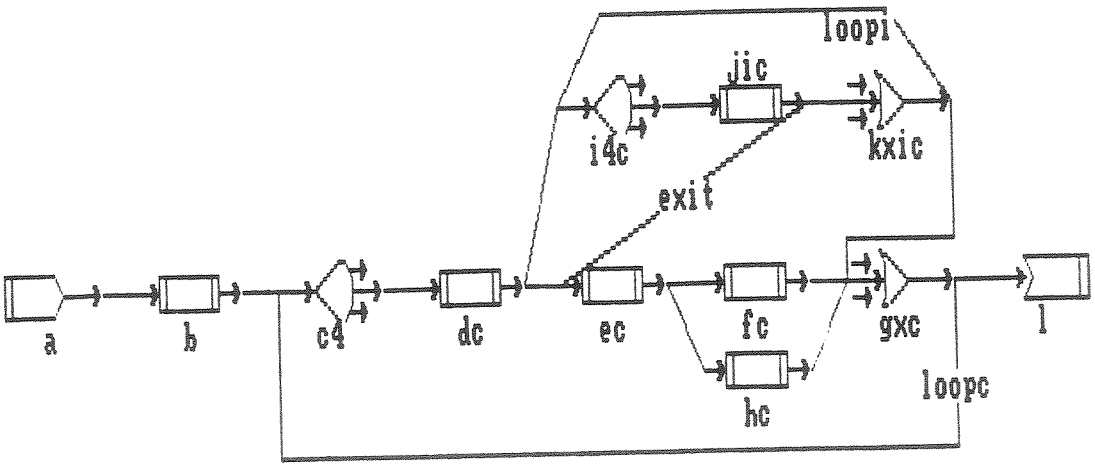


Figure 2-8: A PADS Software Graph

Figure 2-8 shows a sample software graph, in which the node labeled **a** is an ENTER node, and the node labeled **l** is a RETURN node. The graph contains two LOOP nodes, labeled **c4** and **i4c**, and two ENDLOOP nodes, labeled **gxc** and **kxic**. In this case, the loop between **i4c** and **kxic** is totally within the loop between **c4** and **gxc**; these are referred to as **nested loops**. All the other nodes are CODE nodes, which contain instructions in the PADS software description language.

The PADS language itself contains FOR-loops and GOTOs, as well as other constructs for controlling the flow of execution (IF, RAND, etc.), so that the PADS code in the description field of each CODE node may syntactically represent a miniature software graph. The PADS language also contains a CALL construct, from which another software graph may be called; this allows arbitrary nesting of the structure of PADS software models. (A list of the constructs available in the PADS language is in Appendix A.) Each CODE node may therefore represent a single software instruction, an entire set of instructions, or any level of detail in between.

2.3 Arcs

Arcs are used in the PADS graphical interface to connect the nodes in software and collapse graphs. They represent the execution paths taken by transactions between portions of a software graph. No connectivity is shown by the user between any nodes in PADS hardware graphs, since the patterns of usage of hardware resources are determined solely within the software and collapse nodes. Therefore, hardware graphs do not contain any arcs.

Each arc has exactly one source node (known as its "from-node") and one destination node (known as its "to-node"). They are created by clicking the mouse device on both the from-node and the to-node, as described in [7].

Several types of branching (unconditional, conditional, and probabilistic) are available in software graphs, and are described, along with the arcs used to define loops and premature exits, in Appendix A of this thesis.

2.4 Cursors

As mentioned previously, several different types of cursors, or "tools", are available, each with a different function. The tools used in PADS are identical to those currently used in GPSM. There are eight tools available, of which four are used more frequently and are referred to as "top drawer tools," while the four used less frequently are called "bottom drawer tools."

The top drawer tools are: "Draw," which allows objects (i.e., arcs, nodes, graphs and files) to be created or altered, "Erase," which deletes objects, "Move," which allows the user to reposition nodes and arcs on the screen or to change to another graph or file, and "Open," which allows the user to view and edit the text in the definition fields of objects. The user may select any of the four top drawer tools by cycling through them with one of the mouse buttons.

The bottom drawer tools are: "Copy," which permits the copying of PADS objects (nodes, arc specifications, graphs, and files),

"Square," which causes the arcs drawn between nodes to become more square and regular in appearance, "Help," which provides general user assistance about the objects used in PADS, and "Critique," which is intended to check the correctness and consistency of objects and their definitions, although the function of this last cursor is not currently implemented in either PADS or GPSM. These bottom drawer tools are selected with the "Tools" portion of the banner line.

2.5 Windows

2.5.1 Object Windows

Each object in PADS may be annotated, i.e., its properties may be specified when the appropriate fields of its windows are filled out. The user performs this annotation by clicking the "Open" cursor on the object to be annotated, and then entering from the keyboard the appropriate information in the various fields of the window which appears when the object is opened.

Each object type has a window with a specific set of fields. The windows that appear when a file, graph, node and arc are opened are found in Figures 2-9 through 2-12, respectively.

Each field in a window may be edited independently of the others. The "Specification" field can contain several lines of text, which may extend beyond the lower boundary of the window; hence, the window editor permits scrolling up or down in this field. All other fields consist of a single line.

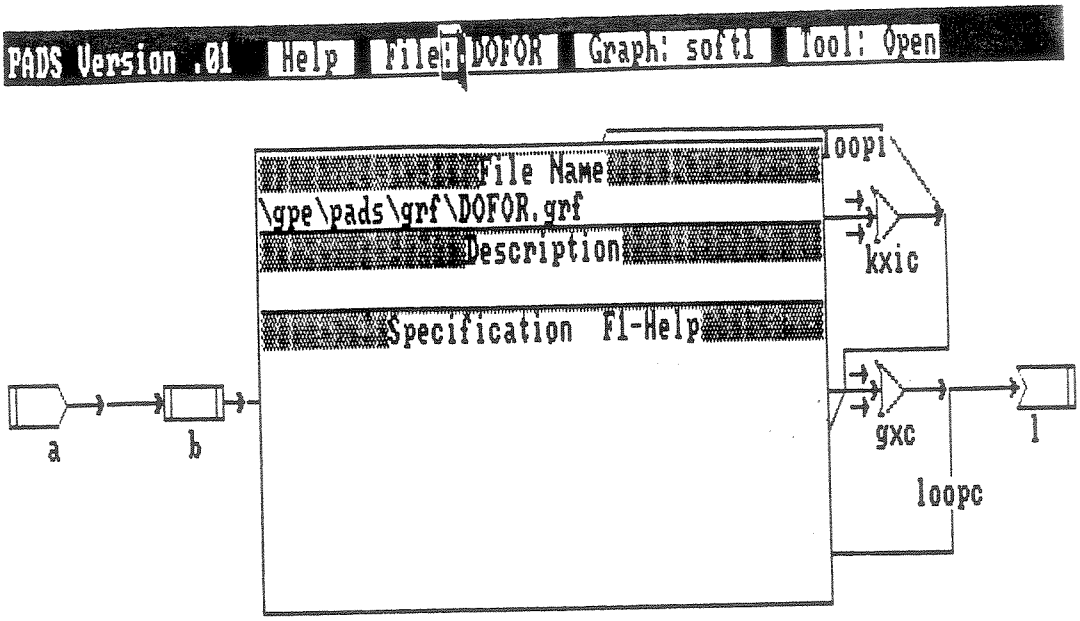


Figure 2-9: Opening a File

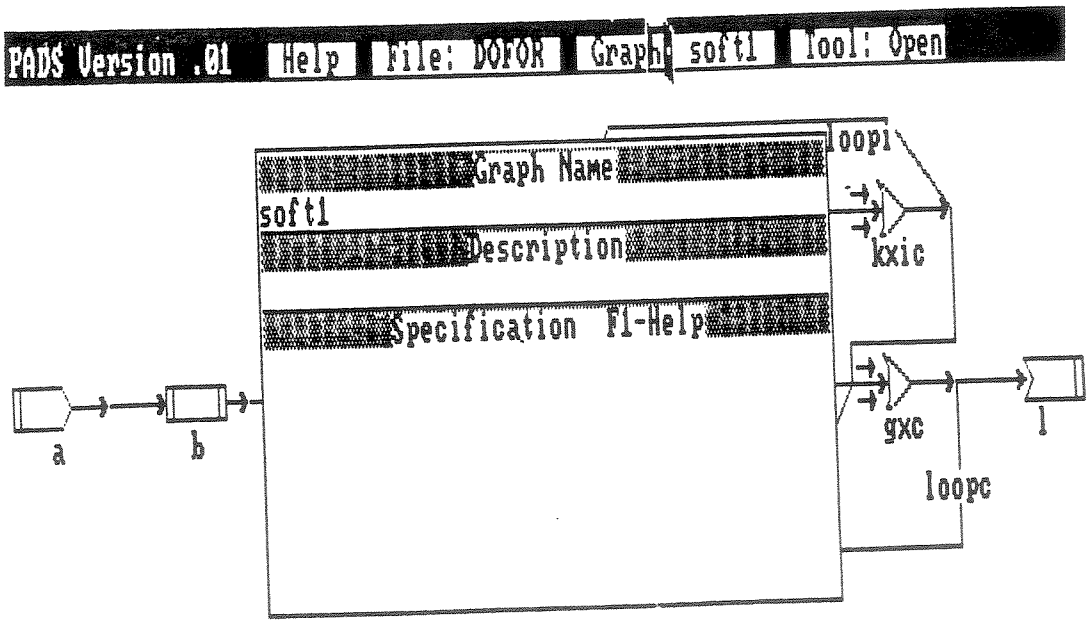


Figure 2-10: Opening a Graph

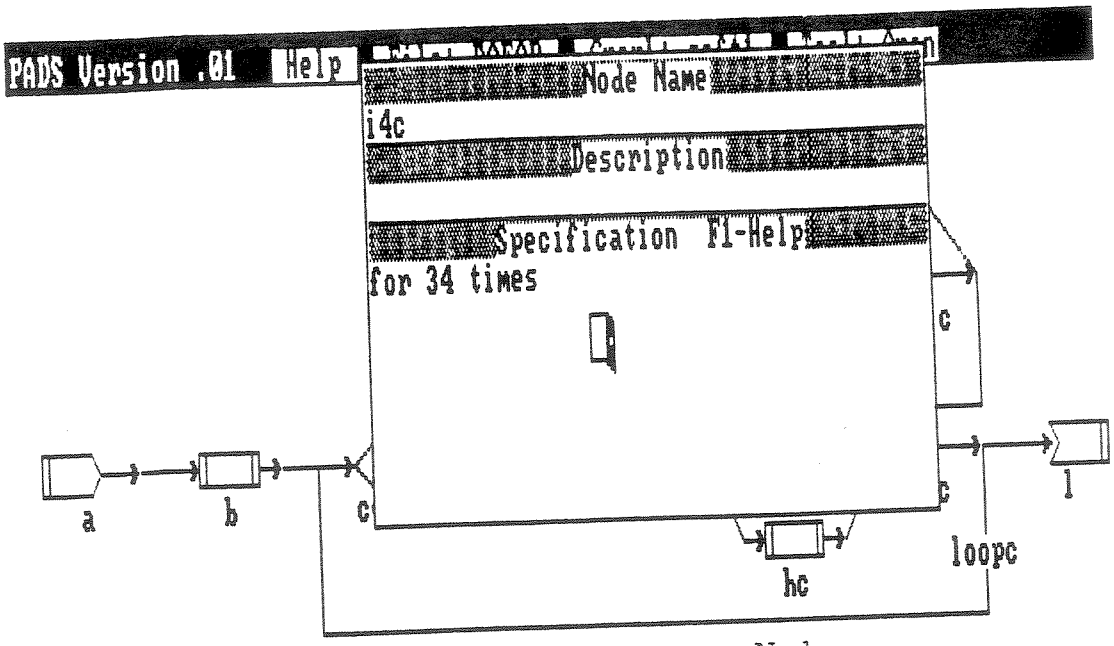


Figure 2-11: Opening a Node

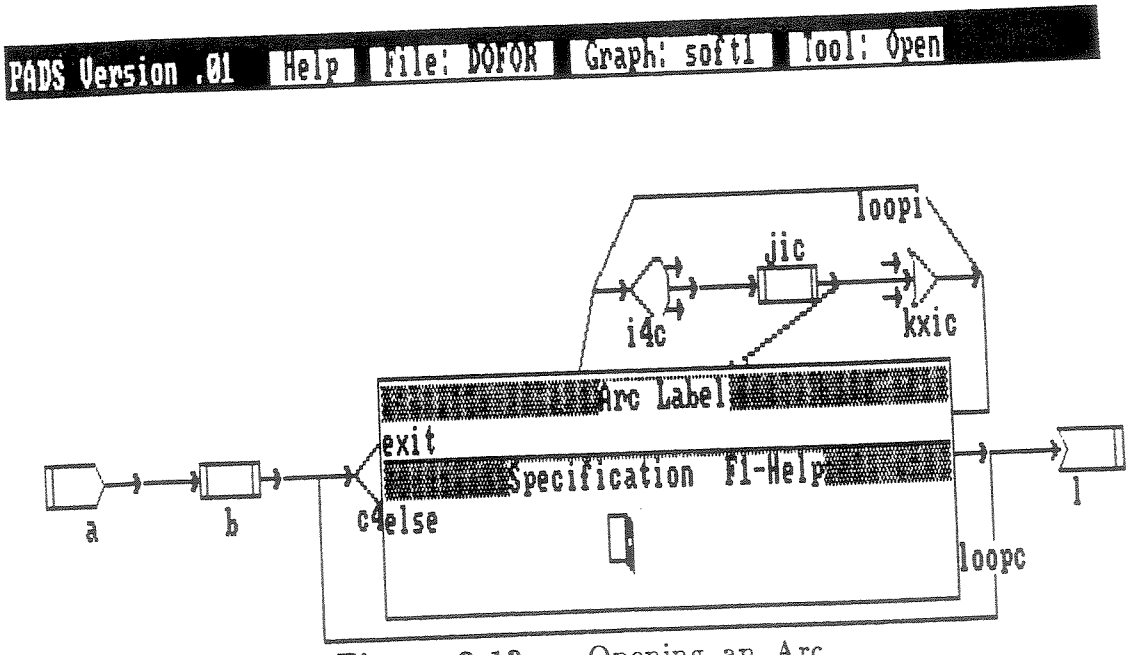


Figure 2-12: Opening an Arc

The "Description" field of an object is simply an English-language comment about that object, entered by the user for his own benefit.

The "File Name" field may contain any name permitted by the operating system and by the limitations of PADS. It stores the name of the entire PADS model, as a file is a collection of graphs.

The "Graph Name" determines the type of a PADS graph: hardware graphs have names starting with "HARD", collapse graphs have names starting with "COLL", and software graphs have names starting with neither. Each graph within the model must have a unique name.

The "Node Name" field is required to contain a unique identifier only when the node is in a hardware graph, although the user may want to attach names to nodes in software and collapse graphs for his own benefit.

The "Arc label" field is significant only if its first four characters are "LOOP", in which case it is a LOOP arc, or "EXIT", in which case it is an EXIT arc (see Appendix A). In all other cases, it treated simply as a comment.

The specification of an object contains most of its definition, and is entered by the user in that object's "Specification" field.

The file specification contains all the declarations required by a PAWS model that cannot be deduced from the rest of the file, such as

those for INTEGER and REAL variables, the INITIAL and RUN sections, and so on. The optional CONSTANTS section permitted by PADS appears here as well.

The specification for a hardware node (i.e., a node in a hardware graph) contains language very similar to that currently used for nodes in GPSM; it is basically the definition field for a node in PAWS with the TYPE field omitted and a few PADS extensions added. The specification for a software node contains text written in the PADS software language, which describes the patterns of usage of hardware resources by that software node. The specification for a collapse node contains a highly constrained, limited syntax description of the usage pattern of hardware SERVICE resources by that node.

The specifications of arcs (and of all other objects mentioned above) are described in Appendix A.

2.5.2 Help Windows

Help windows, which describe the syntax expected in an object's specification, are available when that object's window has been opened by pressing the appropriate function keys. For example, if the banner over an object's specification reads "Specification F1-Help", then pressing F1 produces a help window outlining the syntax expected for that object's specification. This outline may contain references to other function keys, which, if pressed, produce additional help windows detailing some aspect of the syntax outlined in the previous window.

An example of this feature is in Figures 2-13 and 2-14. Here,

the CPU node in the hardware graph of Figure 2-7 has been opened, and the F1 key has been pressed, resulting in the picture shown in Figure 2-13. Pressing the F8 key (to receive more information about the "dist" or service distributions available) produces a second help screen as shown in Figure 2-14. A partial specification for this node (as it is being entered or edited by the user) is in Figure 2-15: notice the cursor at the end of the last line.

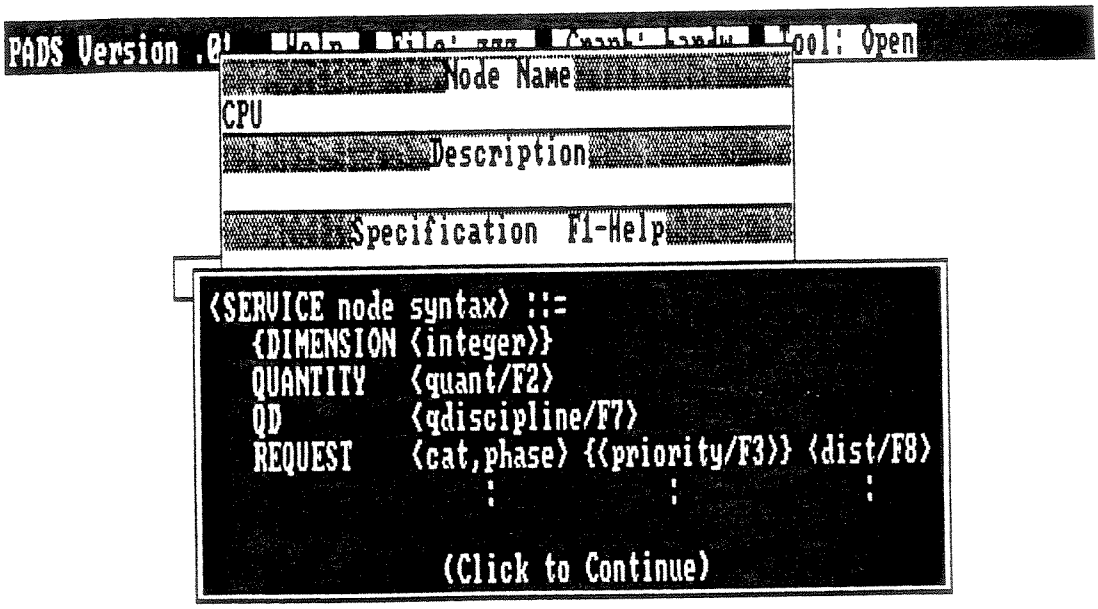


Figure 2-13: A Help Screen

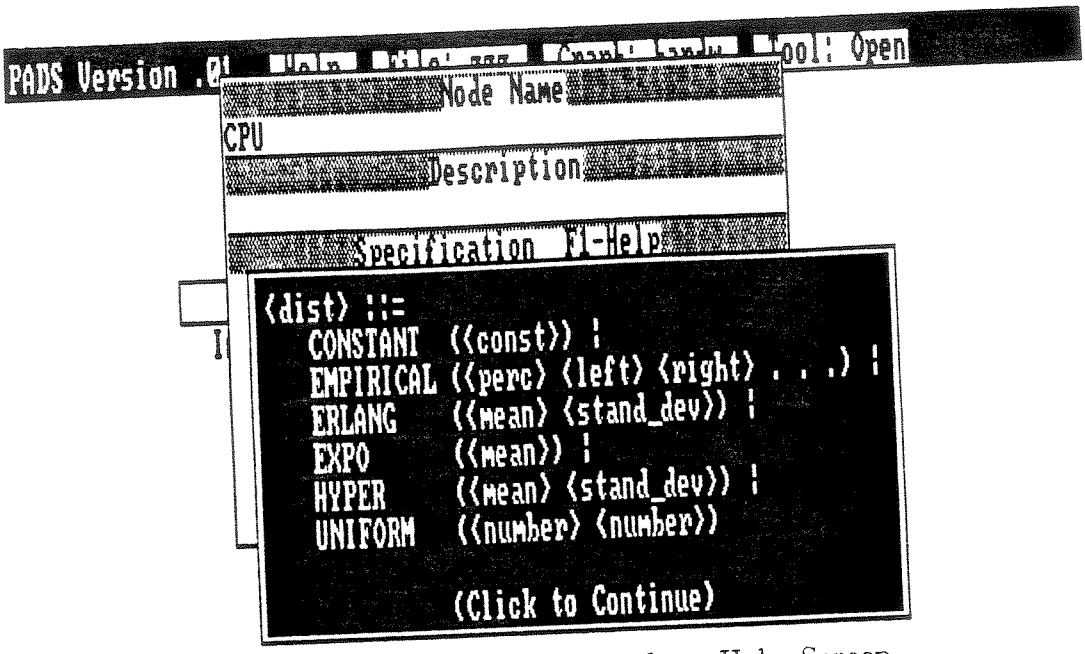


Figure 2-14: A Secondary Help Screen

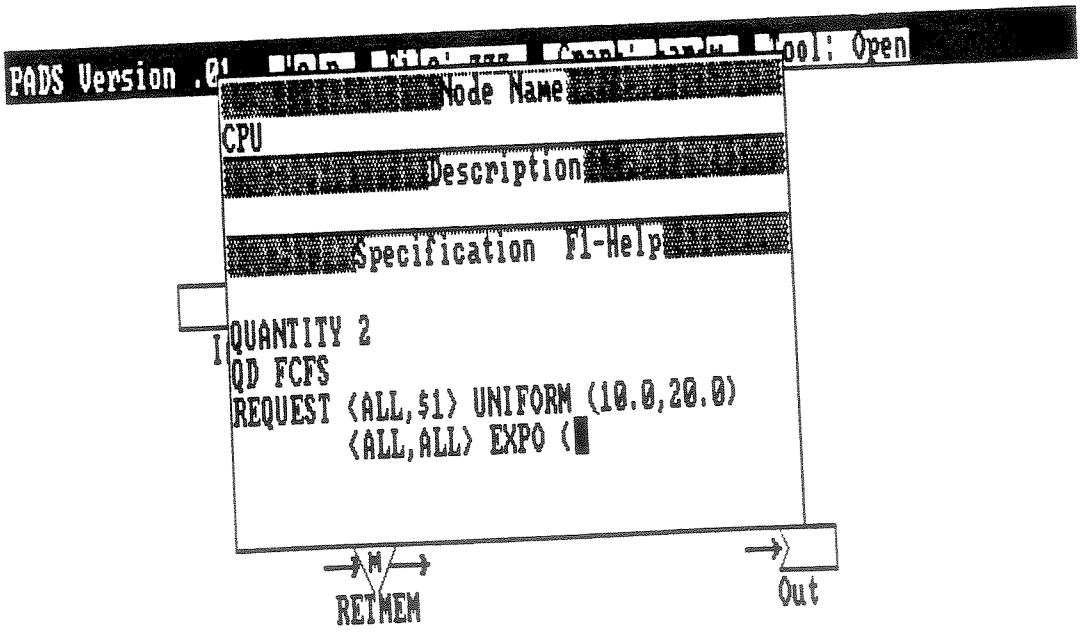


Figure 2-15: Entering a Node's Specification

Chapter 3

An Extended Example

This chapter illustrates the use of PADS in software systems modeling by describing the translation and execution of a software model. The details provided in this example are designed to illustrate certain features of PADS; they are not intended to denote the most efficient or accurate modeling methodology for the given system, nor to reflect the most accurate figures for the execution of any existing system.

3.1 Description of the Software System

The system to be modeled has two disks, one CPU, and twenty-five "blocks" of memory available for temporary files, where a "block" represents any arbitrary, constant amount desired by the modeler. For this example, it is assumed that all requests for disk, CPU, and memory resources are satisfied using a first-come-first-served queueing discipline, and that memory space is allocated according to a "best fit" strategy.

The software using these resources consists of several sets of processes running concurrently on the same hardware.

There is a set of background processes entering the system every eleven seconds. One third of these jobs uses a single burst of CPU, exponentially distributed with a mean value anywhere between one and one hundred milliseconds (ms). (The time unit chosen, ms, is arbitrary, but must remain constant throughout the model.) The remaining two thirds require between ten and twenty cycles of CPU-I/O usage, as follows: each cycle begins with fifty ms of CPU, followed by either (a) twenty ms of I/O to the first disk, or (b) an amount of I/O to the first disk, exponentially distributed with mean value thirty ms, or (c) an amount of I/O to the second disk, exponentially distributed with mean value thirty-five ms. For each cycle, the probability that (a) occurs is ten percent, that (b) occurs is fifteen percent, and that (c) occurs is seventy-five percent.

There is also a set of interactive jobs entering the system at an exponentially distributed rate with a mean of 625 ms. Each job performs the following actions in a loop. First, the job uses an amount of CPU time with mean equal to thirty ms and writes a record to the first disk for an average twenty-five ms, where both times are exponentially distributed. When some random number of records between two and twelve has been written in this way, then an appropriate number of blocks of main memory (between one and the number of records written) is requested. The records are read from the first disk and inserted into memory in some sorted order, then written to the second disk as blocks, each block requiring twenty-five ms to write. (For each record, this [insertion] sorting requires fifteen ms seek time on the CPU, fifteen ms read time on the first disk, and 10 ms CPU time to insert into memory.) Finally, the job relinquishes its blocks of memory, and the job has an eighty-three percent probability of repeating the loop

again; if it does not, any records accumulated but not yet sorted and written out are sorted and written as described, and the job leaves the system.

Finally, there is a set of PAWS jobs being executed on the system, entering at a mean time of six seconds, exponentially distributed. Thirty percent of them perform all of their I/O operations on the first disk; the remainder perform all I/O on the second disk. Most PAWS jobs contain only one RUN section, or "batch", although some may contain more than one (the exact distribution is shown in the specification for node DOPAWS in Figure 3-6). The time required to compile a job is eight hundred ms, exponentially distributed: twenty percent of that is spent in the CPU, the rest in the appropriate disk. Each "batch" in the job requires between five hundred and fifteen hundred ms to simulate, uniformly distributed: only five percent of that time involves the disk, the rest being used for the CPU. The reports generated for each batch require two hundred ms, ten percent of which involves the CPU, the rest being used for the appropriate disk.

In addition to the usual statistics describing the usage of hardware resources, the following statistics are needed: a distribution of the times required to execute the PAWS jobs as modeled above, and of the times spent between memory allocation and memory release in the interactive jobs.

3.2 The PADS Representation

The system described in the preceding section may be represented in PADS as a collection of graphs. In this example, there is one hardware graph, depicted in Figure 3-3, and five software graphs, as shown in Figures 3-4 through 3-8. The entire collection of graphs is referred to as a file or model. In a PADS model, each object (i.e., arc, node, graph, or the file itself) may have a specification defining its purpose or behavior in the model, although the specification of a graph carries no meaning in the current version of PADS. (More detail on the meaning of the specifications of various objects may be found in Appendix A.)

The specification of an object, along with its name and one-line English description, may be viewed and edited by clicking the "Open" cursor on that object, as shown in the illustrations for opening a software and hardware node in this model (Figures 3-1 and 3-2). The specifications of the nodes and arcs in the various graphs are listed below those graphs in each Figure; the specification for the file is shown in Figure 3-9.

The hardware graph contains the nodes describing the hardware resources mentioned in the system, along with a few other nodes. The source node, NEWTRANS, determines at what rate the various types of jobs (transactions) will enter the system, and at what places in the software code (UNIC file) the jobs will begin executing. The branch nodes, PAWSIN, PAWSOUT, CHECKA and CHECKB, do not correspond to any hardware resources, but are called from within the software code to enable PAWS to gather certain statistics (such as response times) while the system model is running.

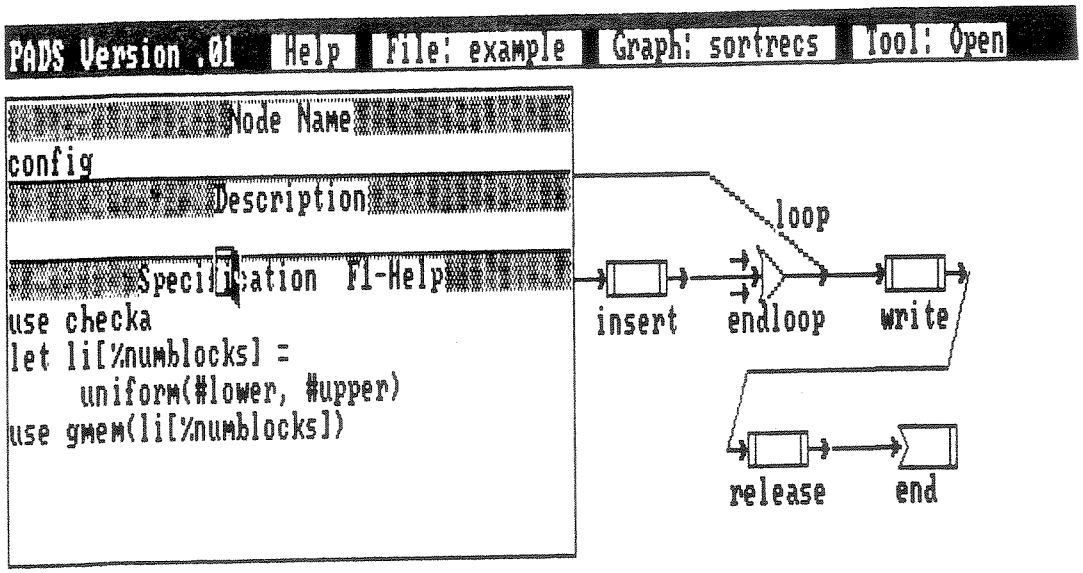


Figure 3-1: Opening a Software Node (CONFIG)

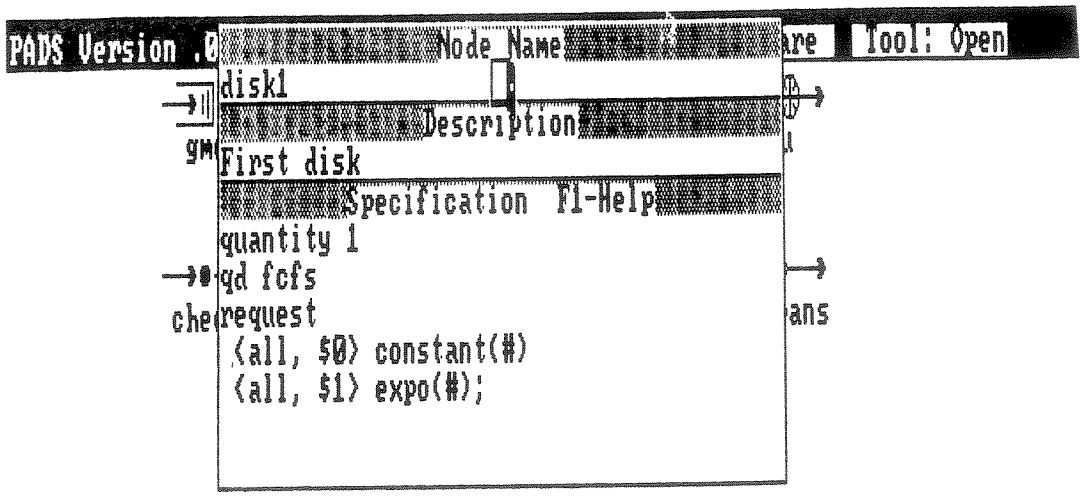
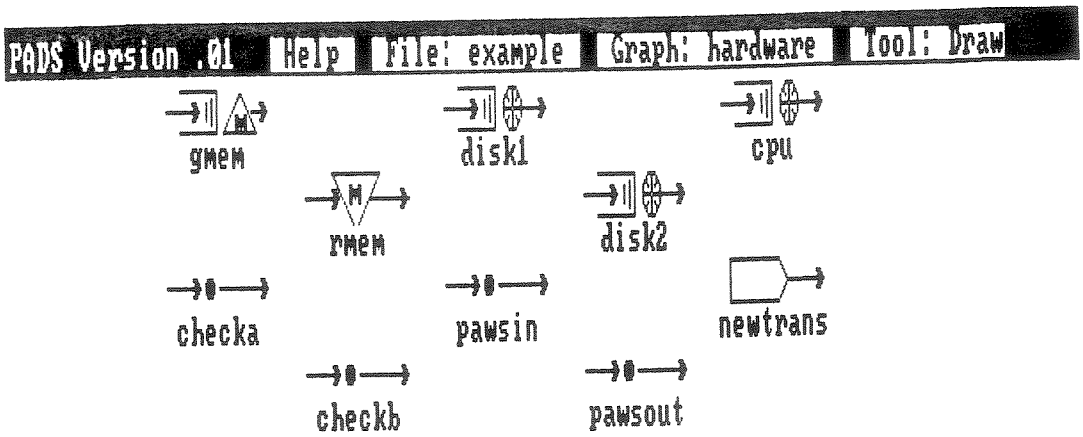


Figure 3-2: Opening a Hardware Node (DISK1)



Specification for GMEM:

```
quantity 25 mainmem bestfit
qd fcfs
request
 <all,all> constant (#) mainmem li[%memadrs]
```

Specification for RMEM:

```
request <all,all> all
```

Specification for DISK1:

```
quantity 1
qd fcfs
request
 <all, $0> constant(#)
 <all, $1> expo(#)
```

Specifications for DISK2 and CPU
are the same as for DISK1.

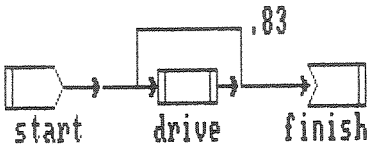
Specification for NEWTRANS:

```
request <inter,%idtrans> expo (625.0)
        <batch,%inbatch> constant(11000.0)
        <paws,%inpaws> expo(6000.0)
```

Specifications for CHECKA, CHECKB,
PAWSIN, and PAWSOUT are blank (empty).

Figure 3-3: The Hardware Graph and its Node Specifications

PADS Version .01 | Help | File: example | Graph: driver | Tool: Draw



Specification for the node START:

```
source %idtrans
let li[8] = 0
```

Specification for the node DRIVE:

```
use cpu($1, %thinktime)
use disk1($1, %writetime)
let li[8] = li[8] + 1
let limit = random*10
let limit = limit + 2
if li[8] >= limit then
  call sortrecs(uniform(1,limit),limit,li[8])
  let li[8] = 0
endif
```

Specification for the node FINISH:

```
call sortrecs(2,3,li[8])
sink
```

Specification for the arc labeled ".83":

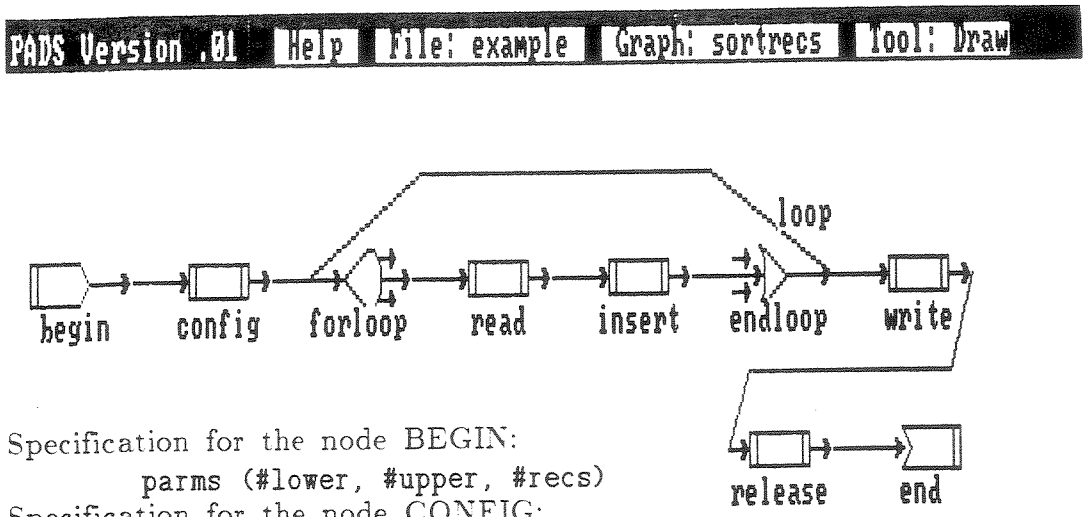
```
if random < .83
```

Specification for the arc from node DRIVE to node FINISH:

```
else
```

All other specifications are empty.

Figure 3-4: The Graph DRIVER and its Specifications



Specification for the node BEGIN:

```
parms (#lower, #upper, #recs)
```

Specification for the node CONFIG:

```
use checka
let li[%numblocks] =
     uniform(#lower, #upper)
use gmem(li[%numblocks])
```

Specification for the node FORLOOP:

```
for #recs
```

Specification for the node READ:

```
use cpu($1, %seektime)
use disk1($1, %readtime)
```

Specification for the node INSERT:

```
use cpu($1, %inserttime)
```

Specification for the node ENDLOOP:

```
endfor
```

Specification for the node WRITE:

```
use cpu($1, %seektime)
let writeout =
     li[%numblocks] * %writetime
use disk2($0, writeout)
```

Specification for the node RELEASE:

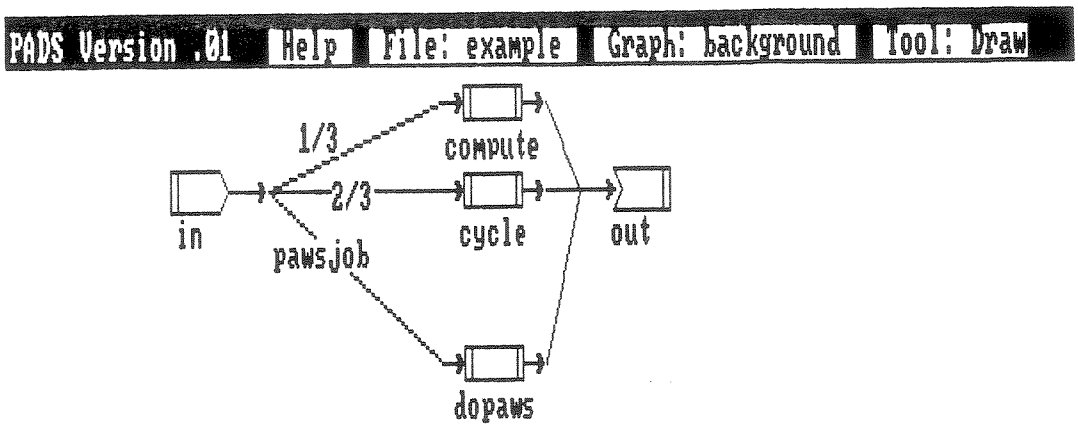
```
use rmem
use checkb
```

Specification for the node END:

```
exit
```

Specification for the arc labeled "loop" is empty.
All other specifications are empty.

Figure 3-5: The Graph SORTRECS and its Specifications



Specification for the node IN:

```
source %inbatch
source %inpaws
```

Specification for the node COMPUTE:

```
let limit=random*100
use cpu($1,limit)
```

Specification for the node CYCLE:

```
for uniform(10,20)
 use cpu($0,50)
 rand
  0.75 : use disk2($1,35)
  0.10 : use disk1($0,20)
  others :use disk1($1,30);
 endrand
endfor
```

Specification for the node DOPAWS:

```
if random < .3 then
  let disknum=1
else let disknum=2
endif
let batchnum = 1+erlang(0.8,0.5)
let size= expo(800)
let runtime = uniform(500,1500)
call runpaws (disknum,batchnum,runtime,size)
```

(More specifications are on the next page.)

Figure 3-6: The Graph BACKGROUND and its Specifications

Specification for the node OUT:

```
sink
```

Specification for the arc labeled "1/3":

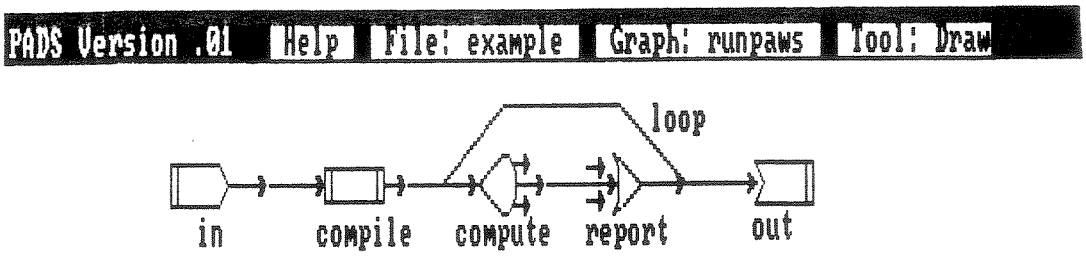
```
cat batch 0.3333
```

Specification for the arc labeled "2/3":

```
cat batch .6667
```

All other specifications are empty.

Figure 3-6, continued.



Specification for the node IN:

```
parms(#disknum,#batchnum,#runtime,#size)
use pawsin
let li[8] = #runtime / #batchnum
```

Specification for the node COMPILE:

```
let size=#size
call cpuio(#disknum,size,0.2,0.8)
```

Specification for the node COMPUTE:

```
for #batchnum
  call cpuio(#disknum,li[8],0.95,0.05)
  !(for-loop continued on next node)
```

Specification for the node REPORT:

```
  !(for-loop continued from prev. node)
  call cpuio(#disknum, 200, 0.1, 0.9)
endfor
use pawsout
```

Specification for the node OUT:

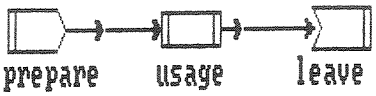
```
exit
```

Specification for the arc labeled "loop" is empty.

All other specifications are empty.

Figure 3-7: The Graph RUNPAWS and its Specifications

PADS Version .01 | Help | File: example | Graph: cpuio | Tool: Draw



Specification for the node PREPARE:

```
parms(#disknum,#runitme,#cpu,#io)
let li[6]=#runtime * #cpu
let li[7]=#runtime * #io
```

Specification for the node USAGE:

```
let li[6]=li[6]/2
use cpu(li[6])
if #disknum=1 then
  use disk1(li[7])
else use disk2(li[7])
endif
use cpu(li[6])
```

Specification for the node LEAVE:

```
exit
```

All other specifications are empty.

Figure 3-8: The Graph CPUIO and its Specifications

```
constants
 %memadrs = 5 ! index of li[] for memory
 %numblocks = 6 ! index of li[] : # of mem. blocks
 %readtime = 15 ! avg. ms to read record fm disk
 %writetime = 25! avg. ms to write block to disk
 %inserttime = 10! ms for cpu to insert record in memory
 %seektime = 15! avg. ms for seek on disk
 %thinktime = 30! avg compute time between record writings to disk
 %inbatch = 1! labels source of batch transactions
 %inpaws = 2! where paws jobs enter
 %idtrans = 3! labels source of inter transactions
 ;

 integers
  writeout ! time to write set of blocks to memory
  limit ! spare integer variable
  disknum ! Disk number to run paws job on
  batchnum ! How many batches in PAWS run section
  runtime ! Total run time (ms) of PAWS modeling job
  size ! Total compile time (ms) of PAWS model
   ;

 categories inter
            batch
            paws
 ; !transaction categ.

 memories mainmem; !type of memory avail.

 statistics report
   response checka checkb gran 600.0
   response pawsin pawsout gran 1000.0
 ;

 run
  go 50000.0 100.0
  dump;
```

Figure 3-9: The File Specification for the PADS Model

The software graphs are similar to flowcharts, and represent the code executed by the jobs. They determine which resources a job uses, in what order they are used, and in what fashion (e.g., how long). The PADS software language used in the specifications of the nodes and arcs in the software graphs is explained in detail in Appendix A.

3.3 Translation

As described in Chapter 1, the model is translated by PADS into a PAWS program and a file of routing instructions, or USER-node-interpretable code.

The first step in the translation process alters the specifications of all nodes and arcs in the model by replacing all references to user-defined constants (which begin with a percent sign) with their corresponding constant values as defined in the CONSTANTS section of the file specification. For example, the expression "mainmem li[%memadrs]" in the specification of node GMEM in the hardware graph is replaced by "MAINMEM LI[5]", since the line "%memadrs = 5" appears in the CONSTANTS section. (The CONSTANTS section and other PADS language constructs mentioned in this chapter are described more fully in Appendix A.) Furthermore, all lower-case letters in the model (except those following the comment symbol, "!") are replaced by the corresponding upper-case letters.

3.3.1 Hardware Translation

The specifications of the hardware nodes are further modified in several ways to bring their syntax into conformity with that required by PAWS nodes. The language used in the hardware node specifications is very similar to that used in PAWS (and GPSM), in order to make this transformation fairly straightforward.

First, each semicolon not inside a comment is deleted, and a single semicolon is placed at the end of the specification, in order to bring the specification language into conformity with the equivalent node definition in PAWS.

Next, the pound signs (which represent a simple form of parameterization of a hardware node) are replaced by actual references to the local real variables (LR[i]) of a transaction, since that is how those parameter values are transmitted in PAWS. For example, the expression "constant (#)" in the node GMEM mentioned above becomes "CONSTANT (LR[1])".

Finally, the "modes" in a node definition are replaced by actual phase values (which is accomplished in the current version by merely deleting the dollar signs preceding the mode numbers), and appropriate lines are added to the specification of SERVICE nodes that have been declared GEXPO. (GEXPO means that service time distributions may use mean and coefficient of variation as parameters in place of the usual PAWS distributions.)

Definitions and examples of the use of pound signs, "modes,"

and GEXPO are in the PADS User's Manual located in Appendix A of this thesis. An explanation of transaction categories and phases is in [12]. Briefly, each transaction in PAWS has associated with it a category, which remains constant throughout the lifetime of the transaction, and a phase, which is a number which can change as directed by the modeler. Both the category and the phase are used in PAWS to control the routing and behavior of transactions; in PADS, the category of a transaction and the mode in which it "uses" a resource determines its behavior while within the resource, as explained in Appendix A.

Other hardware nodes (such as the default SINK node and the intermediate nodes needed to set the phases of resources using "modes") are created automatically by the translator and integrated into the final PAWS model. These nodes have names beginning with the letters "QQ". The nodes beginning with "QQ0000..." are needed to initialize the phase of a transaction before it is sent on to a PAWS resource node that uses "modes," since those modes correspond to PAWS phases.

Figure 3-10 shows the PAWS model that results from the translation of the hardware graph in Figure 3-3.

The hardware graphs are translated into an executable model written in PAWS 2.0. This model has a "star" topology, where each hardware resource is bi-directionally connected to a central USER node. This central node, named "QQUSER", routes all incoming transactions to the appropriate hardware resources, each of which was represented by a node in the hardware graphs.

```
!Example of PADS for thesis illustration

DECLARE

INTEGERS
 WRITEOUT ! time to write set of blocks to memory
 LIMIT ! spare integer variable
 DISKNUM ! Disk number to run paws job on
 BATCHNUM ! How many batches in PAWS run section
 RUNTIME ! Total run time (ms) of PAWS modeling job
 SIZE ! Total compile time (ms) of PAWS model
 ;

NODES

 QQSINK ! Default sink node

! From hardware graph HARDWARE:

GMEM
RMEM
DISK1
  QQ00000004
DISK2
  QQ00000005
CPU
  QQ00000006
NEWTRANS
CHECKA
CHECKB
PAWSOUT
PAWSIN

!Default nodes:
QQINIT QQUSER;

CATEGORIES INTER
           BATCH
           PAWS
; !transaction categ.

MEMORIES MAINMEM; !type of memory avail.
```

Figure 3-10: The PAWS Model Translated from the Hardware Graph

```
TOPOLOGY

QQUSER QQSINK <ALL, 1> 1.0;
QQINIT QQUSER <ALL,ALL> 1.0;

!From hardware graph HARDWARE:

GMEM QQUSER <ALL,ALL> 1.0;
QQUSER GMEM <ALL,   2> 1.0;

RMEM QQUSER <ALL,ALL> 1.0;
QQUSER RMEM <ALL,   3> 1.0;

DISK1 QQUSER <ALL,ALL> 1.0;
QQUSER QQ00000004 <ALL,   4> 1.0;
QQ00000004 DISK1 <ALL,ALL> 1.0;

DISK2 QQUSER <ALL,ALL> 1.0;
QQUSER QQ00000005 <ALL,   5> 1.0;
QQ00000005 DISK2 <ALL,ALL> 1.0;

CPU QQUSER <ALL,ALL> 1.0;
QQUSER QQ00000006 <ALL,   6> 1.0;
QQ00000006 CPU <ALL,ALL> 1.0;

NEWTRANS QQINIT <ALL,ALL> 1.0;

CHECKA QQUSER <ALL,ALL> 1.0;
QQUSER CHECKA <ALL,   8> 1.0;

CHECKB QQUSER <ALL,ALL> 1.0;
QQUSER CHECKB <ALL,   9> 1.0;

PAWSOUT QQUSER <ALL,ALL> 1.0;
QQUSER PAWSOUT <ALL,  10> 1.0;

PAWSIN QQUSER <ALL,ALL> 1.0;
QQUSER PAWSIN <ALL,  11> 1.0;
```

Figure 3-10, continued.

```
DEFINE

QQSINK ! Default sink node
 TYPE SINK;

QQUSER ! Central user node
 TYPE USER
 REQUEST <ALL,ALL> 1;

QQINIT ! Initializes new processes
 TYPE USER
 REQUEST <ALL,ALL> 2;

!From hardware graph HARDWARE:

GMEM
!memory is allocated here
TYPE  GETMEM
QUANTITY 25 MAINMEM BESTFIT
QD FCFS
REQUEST
 <ALL,ALL> CONSTANT (LR[1]) MAINMEM LI[ 5 ]
 ;

RMEM
TYPE  RELMEM
REQUEST <ALL,ALL> ALL
 ;

DISK1
!First disk
TYPE  SERVICE
QUANTITY 1
QD FCFS
REQUEST
 <ALL, 0> CONSTANT(LR[1])
 <ALL, 1> EXPO(LR[1])
 ;

QQ00000004
 TYPE COMPUTE
 REQUEST <ALL,ALL> LETEQ TPHASE LI[4];
```

Figure 3-10, continued.

```
DISK2
!Second disk
TYPE  SERVICE
QUANTITY 1
QD FCFS
REQUEST
 <ALL, 0> CONSTANT(LR[1])
 <ALL, 1> EXPO(LR[1])
;

QQ00000005
 SAME QQ00000004;

CPU
!Central processor
TYPE  SERVICE
QUANTITY 1
QD FCFS
REQUEST
 <ALL, 0> CONSTANT(LR[1])
 <ALL, 1> EXPO(LR[1])
;

QQ00000006
 SAME QQ00000004;

NEWTRANS
!Source of new transactions
TYPE  SOURCE
REQUEST <INTER, 3 > EXPO (625.0)
        <BATCH, 1 > CONSTANT(11000.0)
        <PAWS, 2 > EXPO(6000.0)
;

CHECKA
TYPE  BRANCH
;

CHECKB
TYPE  BRANCH
;
```

Figure 3-10, continued.

```
PAWSOUT
TYPE  BRANCH
;

PAWSIN
TYPE  BRANCH
;


STATISTICS REPORT
  RESPONSE CHECKA CHECKB GRAN 600.0
  RESPONSE PAWSIN PAWSOUT GRAN 1000.0
;

RUN
 GO 50000.0 100.0
 DUMP;

!CONSTANTS
 %MEMADRS = 5 ! index of li[] for memory
 %NUMBLOCKS = 6 ! index of li[] : # of mem. blocks
 %READTIME = 15 ! avg. ms to read record fm disk
 %WRITETIME = 25! avg. ms to write block to disk
 %INSERTTIME = 10! ms for cpu to insert record in memory
 %SEEKTIME = 15! avg. ms for seek on disk
 %THINKTIME = 30! avg compute time between record writings to disk
 %INBATCH = 1! labels source of batch transactions
 %INPAWS = 2! where paws jobs enter
 %IDTRANS = 3! labels source of inter transactions
;

END;

!   0 errors detected.
```

Figure 3-10, continued.

The USER node accomplishes this routing by reading instructions from a file that represents the translation of the software and collapse graphs, and by using some of the variables local to each transaction to keep track of which instruction in the file each transaction is currently executing. This file is known as the "USER-

Node-Interpretable Code" file, which is abbreviated as "UNIC." Its creation from those graphs is described in the next section.

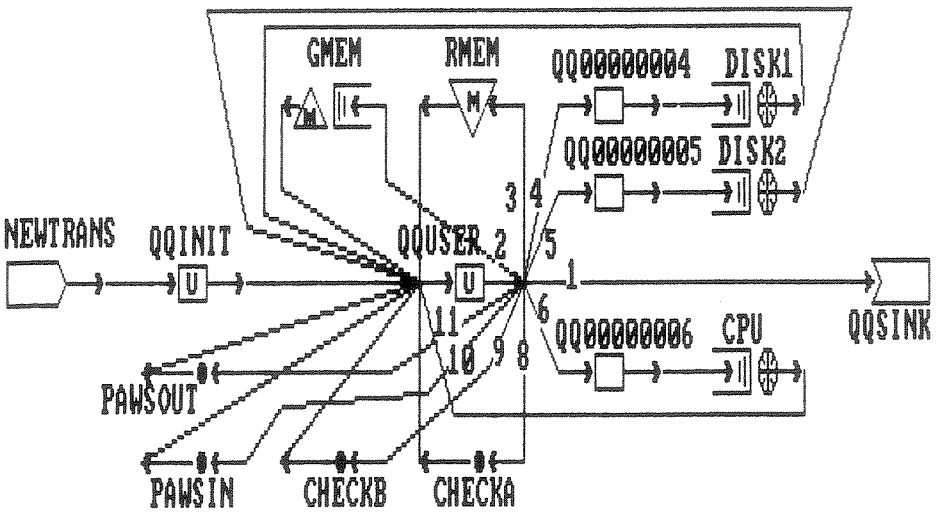


Figure 3-11: A Conceptual View of a Translated PADS Model

As an illustration, Figure 3-11 depicts the graphical equivalent in GPSM of the PAWS model in Figure 3-10. The transactions are created at the source node NEWTRANS, initialized at QQINIT, and sent to QQUSER, where their phases and other local variables are altered before they are sent along the appropriate arc to the correct resource (sometimes via an intermediate node). Here, the number on each arc from QQUSER corresponds to the PAWS phase that a transaction must have in order to follow that arc. After using a resource, transactions return to QQUSER, unless they were directed to QQSINK, in which case they leave the model.

3.3.2 Software Translation

Figure 3-12 is a dataflow diagram presenting an overview of the software translation process.

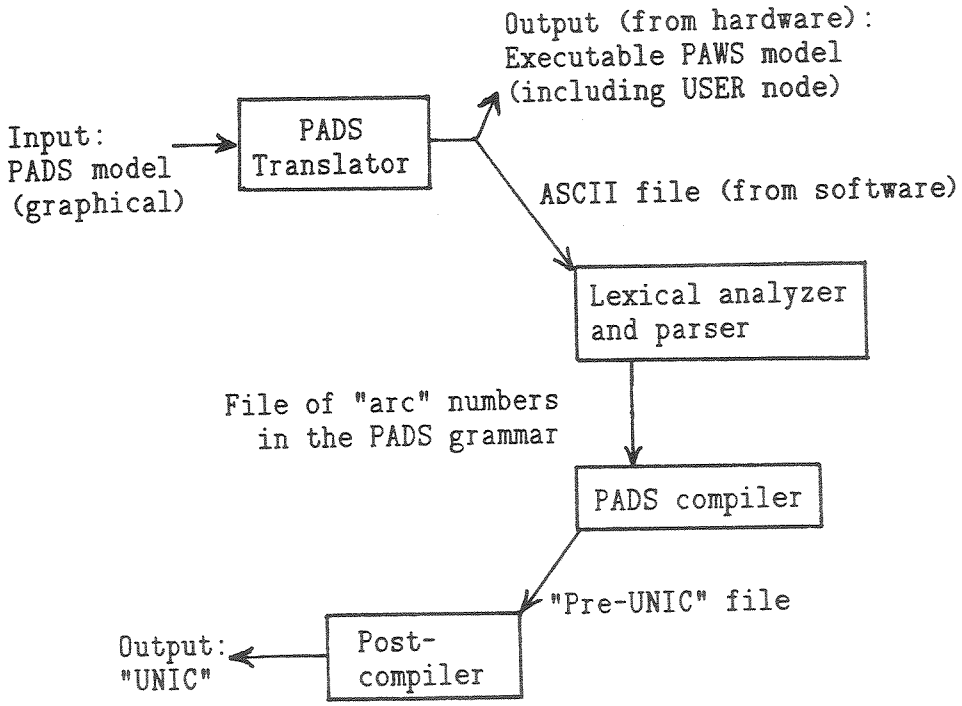


Figure 3-12: A Dataflow Diagram of the Translation Methodology

The software graphs are translated first into an ASCII file representing the code and calls found in each software and transformed collapse graph. The ASCII file duplicates the structure and code content of the graphs, using special symbols to indicate the arcs connecting the nodes and the boundaries of software graphs and nodes. Each software graph is translated separately from the others and given its own section in the file; similarly, each node within a graph is given its own section within that software graph's section. Connections (arcs) between nodes are represented by pieces of PADS code containing translator-generated GOTOs, and FOR-loops of nodes are arranged in the translation so that the nodes within their scope are correctly traversed.

The file is then parsed using a recursive descent technique similar to that described in [1], and translated into a second file which consists of formatted instructions for the central USER node mentioned above. The second file is referred to as the "pre-UNIC" file, since it contains many of the same instructions as the final UNIC file, but implements GOTO and other branching instructions by referring to label numbers instead of to actual locations in the code.

The third and final phase of the software graph translation alters these instructions by replacing references to labels with actual UNIC instruction locations. Sample portions of the files produced at each step of the translation of the software graphs in this example are listed in Figures 3-13 through 3-16.

3.4 Simulation

After the PAWS model and UNIC file have been produced, they may be transferred to any computer that runs the PAWS simulation language. The PADS simulation system is the same as the one for PAWS, but has additional FORTRAN code in subroutine "uss" to allow the UNIC to be interpreted during the simulation. When the model is simulated, the output is a set of statistical results produced by PAWS.

Portions of the statistics files produced by PAWS when our example model is executed are in Figure 3-17. (Some of this output has been compressed horizontally, in order to allow it to fit on the pages of this thesis.) The throughput statistics for some of the hardware resources are listed by transaction category, followed by lists of the queue length

```
?S DRIVER
!:start
?N NO802
SOURCE  3
LET LI[8] = 0
?E NO802
?C ALLLLLLLLL
?G NO803
!:drive
?N NO803
USE CPU($1,  30 )
USE DISK1($1,  25 )
LET LI[8] = LI[8] + 1
LET LIMIT = RANDOM*10
LET LIMIT = LIMIT + 2
IF LI[8] >= LIMIT THEN
  CALL SORTRECS(UNIFORM(1,LIMIT),LIMIT,LI[8])
  LET LI[8] = 0
ENDIF
?E NO803
?C ALLLLLLLLL
IF    RANDOM < .83
THEN ?G NO803
ENDIF
?G NO804
!:finish
?N NO804
CALL SORTRECS(2,3,LI[8])
SINK
?E NO804
?X DRIVER
?S SORTRECS
!:begin
?N NO701
PARMS (#LOWER, #UPPER, #RECS)
?E NO701
?C ALLLLLLLLL
?G NO702
!:forloop
?N NO703
FOR #RECS
?E NO703
```

Figure 3-13: A Portion of the ASCII File Created by "Translating" the Software Graphs

```
0|?S DRIVER
 94
161*S
162*DRIVER
0|!:start
0|?N NO802
 11
  4
  8
 94
161*N
162*NO802
0|SOURCE  3
 11
  4
  8
  3
  6/3
0|LET LI[8] = 0
  8
 83
 95
108
110/8
111
113
114
 62/0
 65/0
193/0
 74/0
0|?E NO802
199
115
120
 11
  4
  8
 94
161*E
162*NO802
```

Figure 3-14: A Portion of the First Parse of the ASCII File

```
! 0 |LET LI[8] = 0
! 0 |?E NO802
0==LI1+#IO          2
      8      0
LENNO802            1
! 0 |?C ALLLLLLLLL
! 0 |?G NO803
GTNNO803           -1
      2
! 0 |!:drive
! 0 |?N NO803
LBNNO803            2
! 0 |USE CPU($1,  30 )
0==PI$+#IO          2
      0      6
0==LI1+#IO          2
      4      1
0==LR1+#IO          2
      1     30
! 0 |USE DISK1($1,  25 )
FRE                 0
0==PI$+#IO          2
      0      4
0==LI1+#IO          2
      4      1
0==LR1+#IO          2
      1     25
! 0 |LET LI[8] = LI[8] + 1
FRE                 0
! 0 |LET LIMIT = RANDOM*10
0ADLI1+LI1;+#IO     3
      8      8      1
! 0 |LET LIMIT = LIMIT + 2
0MUSI1+RR$;+#IO     3
      2      0     10
! 0 |IF LI[8] >= LIMIT THEN
0ADSI1+SI1;+#IO     3
      2      2      2
0GEAB$+LI1;+SI1     3
      0      8      2
GTF                -1
   1001
```

Figure 3-15: A Portion of the "Pre-UNIC" File

```
! 0 |LET LI[8] = 0
! 0 |?E NO802
0==LI1+#IO          2  #4
      8      0
QENNO802            0  #5     (1:      1)
! 0 |?C ALLLLLLLLL
! 0 |?G NO803
GTONO803           -1  #6
      7
! 0 |!:drive
! 0 |?N NO803
QBNNO803            0  #7     (1:      2)
! 0 |USE CPU($1,  30 )
0==PI$+#IO          2  #8
      0      6
0==LI1+#IO          2  #9
      4      1
0==LR1+#IO          2  #10
      1     30
! 0 |USE DISK1($1,  25 )
FRE                 0  #11
0==PI$+#IO          2  #12
      0      4
0==LI1+#IO          2  #13
      4      1
0==LR1+#IO          2  #14
      1     25
! 0 |LET LI[8] = LI[8] + 1
FRE                 0  #15
! 0 |LET LIMIT = RANDOM*10
0ADLI1+LI1;+#IO     3  #16
      8      8      1
! 0 |LET LIMIT = LIMIT + 2
0MUSI1+RR$;+#IO     3  #17
      2      0     10
! 0 |IF LI[8] >= LIMIT THEN
0ADSI1+SI1;+#IO     3  #18
      2      2      2
0GEAB$+LI1;+SI1     3  #19
      0      8      2
GTF                -1  #20
     27
```

Figure 3-16: The Corresponding Portion of the "UNIC" File

and queueing time for those resources. (Interestingly, when the initial amount of memory is increased from twenty-five to thirty blocks, the queueing time for memory becomes zero in this example.) These are followed by the utilization statistics for the "servers" (DISK1, DISK2 and CPU) and for the memory. A distribution of the times required to execute the PAWS jobs is in the statistics report of the "RESPONSES" from node PAWSIN to node PAWSOUT. A similar report of the times spent between memory allocation and memory release in the interactive jobs is found in the "RESPONSES" from CHECKA to CHECKB.

3.5 Expansion and Contraction

An experimental function of the current version of PADS is the ability to perform a very limited form of the operations known as "collapse" and "serialize" on a special form of software graph known as a collapse graph. The collapse operation refers to the "contraction" of a graph into a simpler, more generalized list of the resources it uses and the amount of use it requests for each; the serialize operation takes the results of a collapse and "expands" it onto a generic pattern of usage for those resources. The desired result is an approximation of the original graph which does not seriously affect its accuracy and is simpler to simulate. As the precise functions involved in performing a collapse and serialize are not known at this time, it is difficult to provide a more detailed description of those operations than the one just mentioned. However, the current version of PADS does attempt to implement a limited version of the type of operations one would expect from collapse and serialize; these are described in this section.

Currently, a collapse graph has exactly one ENTER node and

		THROUGHPUT			INPUT	
		RATE	COUNT		RATE	COUNT
NODE:	/GMEM (1)			*		
CAT:	/INTER	0.003	152.	*	0.003	152.
CAT:	/BATCH	0.000	0.	*	0.000	0.
CAT:	/PAWS	0.000	0.	*	0.000	0.
CAT:	/ALL	0.003	152.	*	0.003	152.
NODE:	/RMEM (1)			*		
CAT:	/INTER	0.003	149.	*		
CAT:	/BATCH	0.000	0.	*		
CAT:	/PAWS	0.000	0.	*		
CAT:	/ALL	0.003	149.	*		
NODE:	/DISK1 (1)			*		
CAT:	/INTER	0.017	848.	*	0.017	848.
CAT:	/BATCH	0.300e-03	15.	*	0.300e-03	15.
CAT:	/PAWS	0.300e-03	15.	*	0.300e-03	15.
CAT:	/ALL	0.018	878.	*	0.018	878.
NODE:	/QQ00000004(1)			*		
CAT:	/INTER	0.017	848.	*		
CAT:	/BATCH	0.300e-03	15.	*		
CAT:	/PAWS	0.300e-03	15.	*		
CAT:	/ALL	0.018	878.	*		
NODE:	/DISK2 (1)			*		
CAT:	/INTER	0.003	149.	*	0.003	149.
CAT:	/BATCH	0.460e-03	23.	*	0.460e-03	23.
CAT:	/PAWS	0.240e-03	12.	*	0.240e-03	12.
CAT:	/ALL	0.004	184.	*	0.004	184.
NODE:	/QQ00000005(1)			*		
CAT:	/INTER	0.003	149.	*		
CAT:	/BATCH	0.460e-03	23.	*		
CAT:	/PAWS	0.240e-03	12.	*		
CAT:	/ALL	0.004	184.	*		
NODE:	/CPU (1)			*		
CAT:	/INTER	0.028	1416.	*	0.029	1419.
CAT:	/BATCH	0.780e-03	39.	*	0.780e-03	39.
CAT:	/PAWS	0.108e-02	54.	*	0.108e-02	54.
CAT:	/ALL	0.030	1509.	*	0.030	1512.

Figure 3-17: Portions of the Statistical Results

```
QUEUE AT NODE:          /GMEM      (  1)
  CATEGORY:           /INTER
             QUEUE-LENGTH
               SUMMARY
                     MEAN:       0.153 2ND MOMENT:       0.401
                     VAR:        0.378 STNDRD DEV:       0.615

             QUEUEING-TIME
               SUMMARY
                     MEAN:      50.440 2ND MOMENT:   25593.113
                     VAR:    23048.904 STNDRD DEV:     151.819
  CATEGORY:            /ALL
             QUEUE-LENGTH
               SUMMARY
                     MEAN:       0.153 2ND MOMENT:       0.401
                     VAR:        0.378 STNDRD DEV:       0.615

             QUEUEING-TIME
               SUMMARY
                     MEAN:      50.440 2ND MOMENT:   25593.113
                     VAR:    23048.904 STNDRD DEV:     151.819

QUEUE AT NODE:          /DISK1     (  1)
  CATEGORY:           /INTER
             QUEUE-LENGTH
               SUMMARY
                     MEAN:       1.924 2ND MOMENT:      15.296
                     VAR:       11.592 STNDRD DEV:       3.405

             QUEUEING-TIME
               SUMMARY
                     MEAN:     113.472 2ND MOMENT:  239232.391
                     VAR:   226356.422 STNDRD DEV:     475.769
  CATEGORY:            /BATCH
             QUEUE-LENGTH
               SUMMARY
                     MEAN:       0.012 2ND MOMENT:       0.012
                     VAR:        0.011 STNDRD DEV:       0.107

             QUEUEING-TIME
               SUMMARY
                     MEAN:      38.305 2ND MOMENT:    3808.602
                     VAR:     2341.313 STNDRD DEV:      48.387
```

Figure 3-17, continued.

```
CATEGORY:          /PAWS
          QUEUE-LENGTH
            SUMMARY
                  MEAN:       0.153 2ND MOMENT:       0.169
                  VAR:        0.146 STNDRD DEV:       0.382

          QUEUEING-TIME
            SUMMARY
                  MEAN:   0.510e+03 2ND MOMENT:   0.151e+07
                  VAR:    0.125e+07 STNDRD DEV:   0.112e+04

CATEGORY:          /ALL
          QUEUE-LENGTH
            SUMMARY
                  MEAN:       2.089 2ND MOMENT:      17.925
                  VAR:       13.561 STNDRD DEV:       3.682

          QUEUEING-TIME
            SUMMARY
                  MEAN:     118.964 2ND MOMENT:  256893.266
                  VAR:   242740.766 STNDRD DEV:     492.687


QUEUE:          /DISK1     (  1) NUMBER OF SERVERS:        1
  CATEGORY                MEAN SERVICE TIME        UTILIZATION
          /INTER                20.145                  34.17
          /BATCH                21.587                   0.65
          /PAWS                473.133                  14.19
          /ALL                  27.909                  49.01
QUEUE:          /DISK2     (  1) NUMBER OF SERVERS:        1
  CATEGORY                MEAN SERVICE TIME        UTILIZATION
          /INTER                46.644                  13.90
          /BATCH                21.916                   1.01
          /PAWS                184.750                   4.43
          /ALL                  52.560                  19.34
QUEUE:          /CPU       (  1) NUMBER OF SERVERS:        1
  CATEGORY                MEAN SERVICE TIME        UTILIZATION
          /INTER                18.616                  52.72
          /BATCH                49.248                   3.84
          /PAWS                163.037                  17.61
          /ALL                  24.576                  74.17
```

Figure 3-17, continued.

```
MEMORY:            /MAINMEM
  CATEGORY                      UTILIZATION
             /INTER                  33.11
             /BATCH                   0.00
             /PAWS                    0.00
  ALL                                33.11


RESPONSES FROM       /CHECKA     ( 1)  TO       /CHECKB     ( 1)
  CATEGORY:      /INTER
          RESPONSE-TIME
           INTERVAL          NUMBER IN    % IN          HISTOGRAM
                              INTERVAL  INTERVAL  0   10   20   30
   0.000 <= X <    600.000    46.000     30.87   I***************<
 600.000 <= X <   1200.000    34.000     22.82   I***********<
1200.000 <= X <   1800.000    22.000     14.77   I*******<
1800.000 <= X <   2400.000    14.000      9.40   I****<
2400.000 <= X < *INFINITY*    33.000     22.15   I***********<
                             .........
                  TOTAL:     149.000

          SUMMARY
                MEAN:  0.153e+04 2ND MOMENT:  0.501e+07
                VAR:   0.266e+07 STNDRD DEV:  0.163e+04

RESPONSES FROM       /PAWSIN     ( 1)  TO       /PAWSOUT    ( 1)
  CATEGORY:      /PAWS
          RESPONSE-TIME
           INTERVAL          NUMBER IN   % IN          HISTOGRAM
                              INTERVAL INTERVAL 0   10   20   30   40
   0.000 <= X <   1000.000    0.000      0.00  I<
1000.000 <= X <   2000.000    1.000     14.29  I*******<
2000.000 <= X <   3000.000    2.000     28.57  I**************<
3000.000 <= X <   4000.000    3.000     42.86  I*********************<
4000.000 <= X < *INFINITY*    1.000     14.29  I*******<
                             ........
                  TOTAL:      7.000

          SUMMARY
                MEAN:  0.388e+04 2ND MOMENT:  0.203e+08
                VAR:   0.526e+07 STNDRD DEV:  0.229e+04
```

Figure 3-17, continued.

one RETURN node, both of which have blank specifications and serve only to mark the entry and exit points of the graph. The other nodes are all of type CODE and have specifications containing lines of the form:

freq \ *resource-id* (*total-req*)

where *resource-id* is the name of the hardware resource being used (which must be of type SERVICE), *total-req* is the total time that a transaction requests service from that resource while (conceptually) within the current node, and *freq* is the number of times the transaction requests the use of that resource during that time. The letter C may be used in place of a number for *freq*, in which case the resource is treated like a CPU and is requested n+1 times, where n is the sum of the other *freq*s (not including any other C's) in that node. The arcs connecting the nodes in a collapse graph all have specifications consisting of simple probabilities (literal real numbers between 0.0 and 1.0); a blank arc specification defaults to 1.0.

The collapse is performed as follows: the average number of times each node is invoked is calculated analytically using the visit ratio equations described in [4], and a Gaussian elimination routine as shown in [14]. The values of *freq* and *total-req* are then multiplied by the visit ratio of each node and used to form a nested FOR-RAND construct.

As is the case with software graphs, each collapse graph is given its own section in the ASCII file. Unlike software graphs, however, collapse graphs are converted into a nested FOR-RAND construct instead of having their nodes translated separately. An example of a collapse graph and its conceptual equivalent, as well as the construct resulting from its translation, are in Figures 3-18 through 3-23.

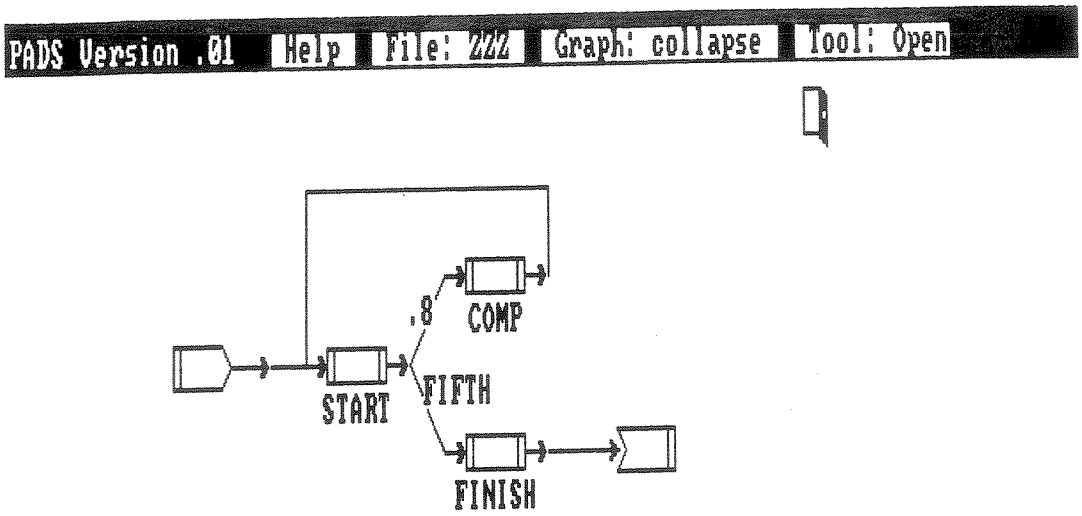


Figure 3-18: A PADS Collapse Graph

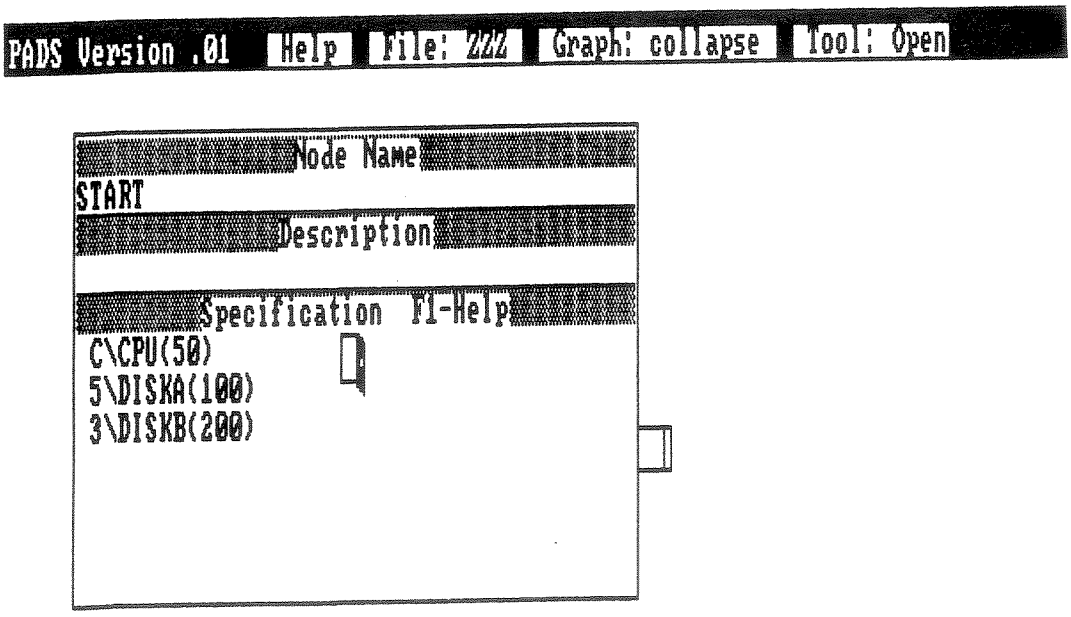


Figure 3-19: The Definition of Node "Start"

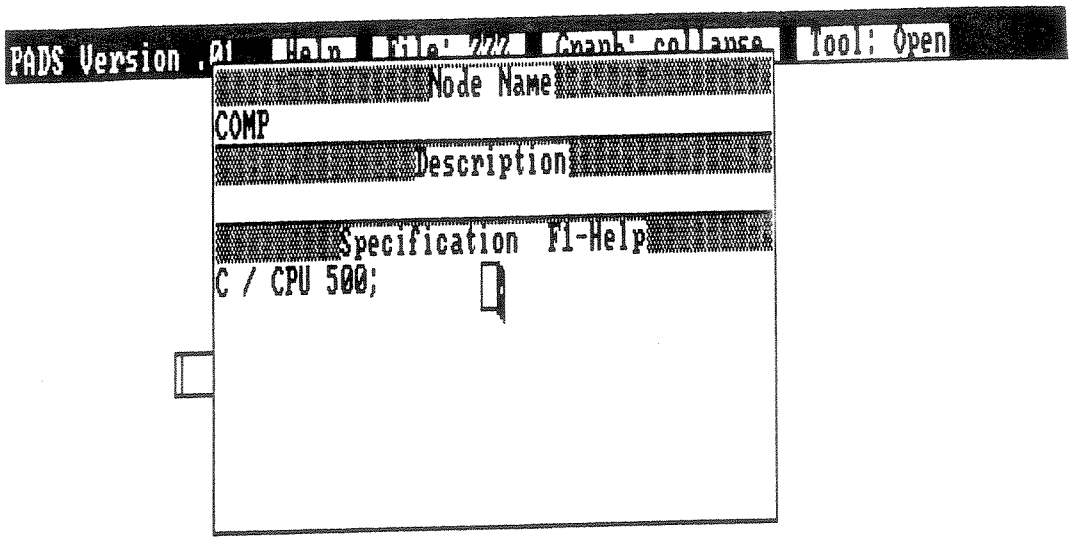


Figure 3-20: The Definition of Node "Comp"

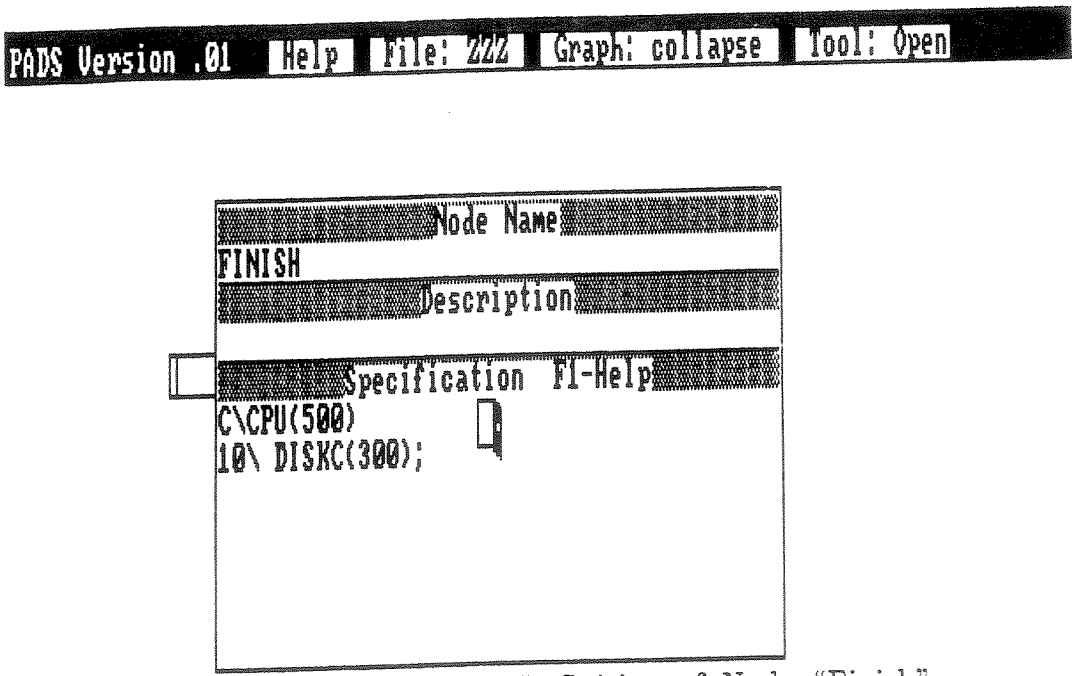


Figure 3-21: The Definition of Node "Finish"

```
? S COLLAPSE
FOR          110 TIMES
RAND
 0.409090900 : CPU          (GEXPO,        5.556, 0);
 0.227272700 : DISKA        (GEXPO,       20.000, 0);
 0.136363600 : DISKB        (GEXPO,       66.667, 0);
 0.036363640 : CPU          (GEXPO,      500.000, 0);
 0.100000000 : CPU          (GEXPO,       45.455, 0);
!( 0.090909090)
 OTHERS       : DISKC       (GEXPO,       30.000, 0)
ENDRAND
ENDFOR
? X COLLAPSE
```

Figure 3-22: The Actual Translation of the Collapse Graph into the PADS Language

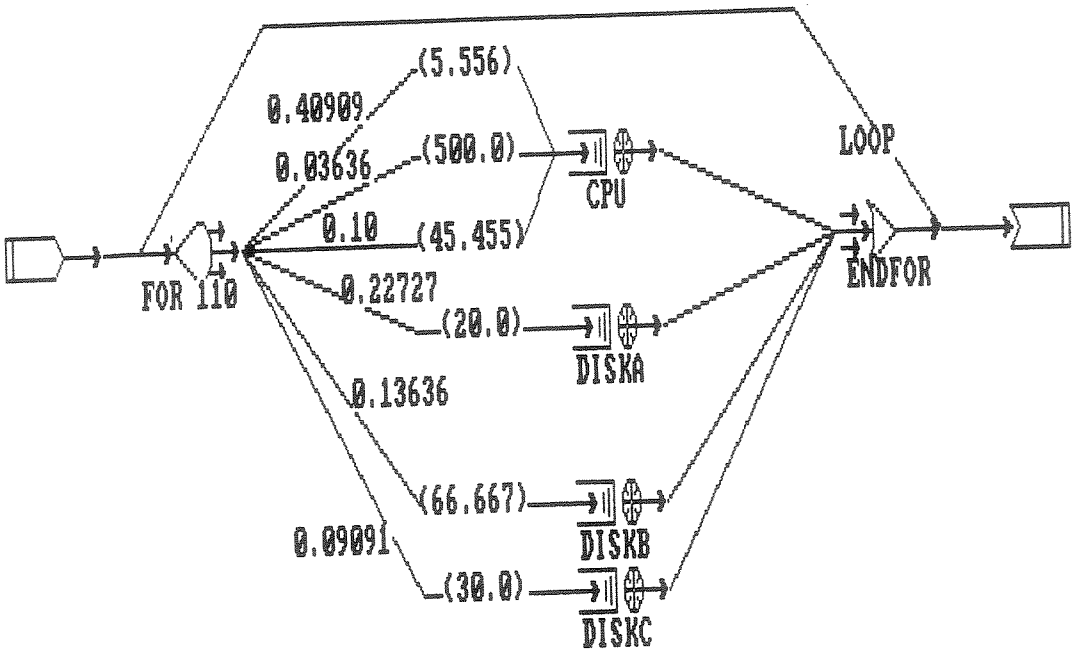


Figure 3-23: A Conceptual View of the Equivalent Hardware Usage Pattern

Chapter 4

Proposed Changes to PADS

The preceding chapters have described a preliminary version of PADS. This chapter discusses a number of improvements that are being considered for future versions of PADS.

4.1 Translation Methodology

Several improvements can be made to the automatic translation processes used in PADS. Choosing the "Translate" option (when the "Draw" cursor is clicked on the "File" portion of the banner) causes two files to be created, as described in Section 7.4.2 of the *PADS User's Manual*. (That manual is found in Appendix A of this thesis.) Both the hardware and software file are created using a rather "ad hoc" method; the PADS translator has relatively little information about the correct syntax and semantics of the graphical objects or their specifications. Furthermore, the software file must undergo a three-pass translation into the "UNIC" file *after* PADS has relinquished control to the operating system. For both these reasons, relatively few errors in a model can be detected by PADS at translation time.

A possible solution to the above problem would be to create

the "UNIC" file (or at least the "pre-UNIC" file) directly from the software graphs by traversing and parsing the actual graphs, instead of by parsing a file created from them. The hardware graphs could be parsed in the same manner. This method would make it possible to show the source of a syntactic or semantic error to the user while translation is taking place. Notice that this parsing capability would be equivalent to the unimplemented "Critique" function described earlier.

4.2 Impact on Hardware Models

The current PADS system is subject to several constraints with regard to the types of hardware systems that can be modeled. Many, if not all, of these constraints were imposed to simplify the implementation of the first version of PADS, and could be eliminated in future versions of PADS. This is especially true if the method for translating PADS graphs into PAWS is simplified, or if the PADS graphical interface is modified. For example, FORK and JOIN nodes are not currently permitted in PADS hardware models, partly because parallel execution within software was deemed too complex to be worth implementing in a preliminary version of PADS, and partly because PADS currently uses the GPSM version of the icon palette for all its graphs, and uses the FORK and JOIN icons to represent FOR- and ENDFOR- nodes. A similar statement can be made for SPLIT nodes. It is hoped that future implementations of the PADS graphical interface will use a different icon palette for each type of graph, thus freeing the FORK, JOIN, and SPLIT nodes for use (with an appropriate software language interface) in describing parallelism.

Currently, the hardware graphs consist of disjointed PAWS

resource nodes with no arcs connecting them (recall that the routing of transactions between resource nodes is described in the software graphs). One possible extension to the current PADS hardware graphs is the addition of CALL nodes, which are similar to the submodels currently represented by subgraphs in GPSM. These would allow commonly-used patterns of hardware usage (such as a disk controller system) to be specified directly as physical connections rather than indirectly as logical connections through the central USER node.

However, this extension would probably require the implementation of ALIAS nodes in PAWS (nodes which share a queue and which represent the same resource), and would require a more sophisticated method of parameterization than is currently available in PAWS or PADS. These changes are currently scheduled to be incorporated into a later version of PAWS, and are outlined in Section 4.4. Precise statements about the feasibility of their implementation must, however, be postponed until the target language has been precisely defined.

4.3 Transaction Categories

Currently, categories are used in PADS to partially determine the behavior of a transaction within a software or a hardware node; this is similar to their use in PAWS. Another use of categories in PADS is to facilitate the grouping of statistics, since PAWS groups a model's performance statistics by category, not by phase. As in PAWS, the category of a transaction does not change throughout its lifetime.

The chief advantage to this approach is that it corresponds to

the way categories are currently treated in PAWS. Therefore, it is simple to implement and to understand. Its main disadvantage is that statistical results in PAWS are separated by category, which makes it difficult to determine which portions of the *software* require a given amount of resources.

A second possible approach would be to equate the category of a transaction with the software graph it is currently executing, i.e., to change the category of a transaction each time it calls a new software graph. This approach resolves the difficulties posed by the current approach; it also frees the user from having to know about PAWS categories. However, it introduces some additional problems. A PAWS transaction cannot actually change its category. In order to achieve the same effect, the transaction must first create a copy of itself at a SPLIT node (giving that copy the new category), pass all its passive resources (such as tokens and memory) to the copy, and then leave the system at a SINK node. This procedure is more difficult to implement and may create more overhead during the simulation process. Furthermore, the user still does not have full control over the grouping of statistics, although the correspondence between categories and PADS software graphs may be sufficient for grouping purposes.

If the current treatment of categories is to be altered in PADS, the best approach would be the following: the category of a transaction is determined by the user when the transaction is created, but is changed whenever the transaction encounters the statement "NEWCAT *new-category-name*" in the software graphs. This allows the user full control over the grouping of statistics, and ensures that any resultant overhead occurs only when the user specifically requests a category change.

4.4 Future Versions of PAWS

The current version of PADS produces models that run under version 2.0 of PAWS, which is basically a subset of version 3.0. At the same time that PADS was being developed, a new version of PAWS, called ES/PAWS, was being designed. The "ES" stands for "Electronic Systems"; ES/PAWS is an enhancement to PAWS designed for high-level simulation of electronic systems, and was first described in [10]. Most of the enhancements that would affect PADS are listed in [11]. PADS, when appropriately modified, would provide an ideal interface for ES/PAWS.

4.4.1 Topology Enhancements in ES/PAWS

The current proposal for ES/PAWS extends the syntax for arcs (edges between nodes) to include specifications of the form "FOR *integer*" and "IF *boolean*" in addition to the "*probability*" currently allowed in PAWS [11, p. 36]. However, PADS currently uses FOR-nodes instead of FOR-arcs. Changing the PADS methodology of representing FOR-loops to conform to that used in ES/PAWS would make the resulting PADS model easier to translate, and would prevent the user from having to learn two different graphical interfaces for loops in PADS and in ES/PAWS.

Although WHILE-loops are currently not directly implemented in PADS, these same icons could be used to denote the beginning and end of WHILE-loops, the difference between the two being determined by the text in the definition fields of the LOOP node. However, since the proposed version of ES/PAWS does not make any provision for

WHILE-loops, it is not clear that providing them in PADS would offer much advantage over conforming to ES/PAWS usage by not providing them.

4.4.2 ALIAS Nodes

Version 3.0 of PAWS permits the use of submodels, which may be called by a transaction in the PAWS model. Currently, a given node can appear in only one PAWS submodel; the same resource cannot therefore be represented directly in different submodels. This limitation severely inhibits the development of independent PAWS and GPSM submodels, as well as any future ability to integrate GPSM submodels into a PADS model, since there is no way to allow two hardware resources named in separate places to be serviced by the same queue (i.e., to act as aliases for each other).

This problem may be solved by including the concept of a "node alias" in future versions of PAWS. For example, if a SERVICE node named "CPU" is defined (without a REQUEST section) as

```
QUANTITY 1
QD FCFS
```

then this resource may be referred to by a different name and with a different REQUEST section in each of several submodels, as follows:

```
SUBMODEL A
...
PROCESSOR
   ALIAS CPU
   REQUEST <ALL,ALL> EXPO(10.0);
...
END ! of SUBMODEL A
```

```
SUBMODEL B
...
CPU2
   ALIAS CPU
   REQUEST <ALL,ALL> UNIFORM(5.0, 20.0);
...
END ! of SUBMODEL B
```

Thus, transactions arriving at the two nodes PROCESSOR and CPU2 will enter the same queue and compete for the same resource, CPU. If future versions of PADS will have the capability to "USE" GPSM submodels as well as individual hardware resources, then node aliasing would allow a hardware resource declared in a PADS hardware graph to be referred to in a GPSM submodel.

4.4.3 PAWS Submodel Parameters

In order for a PADS model to "USE" GPSM submodels, they must have an appropriate method of parameterization. The enhancements proposed in [11] include both compile-time and call-time parameters for submodels. If a similar method of parameterization were offered at the **node** level, the need to reserve local real variables for node parameters (as mentioned in the next section) would be eliminated in PADS.

4.4.4 Avoiding PADS Conflicts with Users

Currently, PADS reserves the first four local integer variables of each transaction for its own purposes (instruction pointer, stack pointers, etc.), as well as reserving function codes 1 and 2 in the USER node. The PADS user should not attempt to reuse any of these objects for his own purposes, or unpredictable errors will result.

To avoid these conflicts and still allow the user full access to the items mentioned above, two features can be added to future versions of PAWS. First, a new node type, called XUSER, would behave exactly like a USER node except that it would call FORTRAN subroutine XUSS instead of subroutine USS. The XUSER node type would be available to PADS but not to the person writing a PADS model. All PADS-related code in USS would be moved to XUSS, freeing the entire USER node for the modeler to use. Second, a new set of local variables for each transaction, called XI and XR instead of LI and LR, could be implemented in the same way, freeing all the current PAWS local variables for the modeler's use. (Currently, PADS does not reserve the use of local boolean variables, and the local real variables are reserved only when passing parameter values to hardware resources from a USE statement in the software graphs, as explained in Appendix A.)

Chapter 5

Conclusions

The graphical interface described in this thesis enables a designer of software systems to specify his workload and hardware configuration separately. A set of hardware devices and a set of software workloads for these devices are combined in a PADS model to determine how effectively the hardware devices can perform under the given workload. Software specifications and hardware configurations can be changed separately and conveniently to allow the evaluation of alternate system designs.

There is a very close integration of the software representation with the hardware representation. This integration allows the designer to think about hardware and software in compatible terms.

The software load is described in terms of virtual devices (i.e., software graphs) which are mapped onto physical devices. These virtual devices may themselves be hierarchically defined, so that specifications for workloads can be structured hierarchically to match the resolution of the hardware specification. Therefore, the hardware devices can be represented down to any level of detail desired.

The output generated by PAWS for a PADS software systems model may then be used to evaluate alternate designs for hardware configurations under a variety of workloads.

Appendix A.

PADS User's Manual

This user's manual assumes that the reader is familiar with PAWS and GPSM, and has read the user's manuals for both [7, 12]. Its purpose is to highlight the differences between PADS and GPSM, and to describe the functions of PADS at the same level of detail as found in the GPSM User's Manual [7].

Where used, section numbers refer to the same sections in [7]. **An empty or missing section** indicates that the text for that section in this manual would be identical to its corresponding section in [7], with the exception that the word "GPSM" would be replaced by the word "PADS" throughout the text.

A note on punctuation: contrary to current English usage, most punctuation marks that are not actually part of the item being quoted are left outside the quotation marks instead of being placed inside them. This is done to avoid confusion as to whether or not the punctuation mark should be included along with the item quoted when it is entered into the computer, program, etc. If the item quoted is not actually entered or stored in the computer, then the punctuation is included in the quotation marks, as required by the rules of English.

Preface

[An introduction to the purpose and background of PADS is found in Chapter 1 of this thesis.]

1.1. PADS graphs vs. GPSM IPGs

In GPSM, each graph is considered to be an Information Processing Graph (IPG), which consists of information processing nodes connected by arcs. In PADS, there are three distinct types of graphs: hardware graphs, software graphs, and collapse graphs.

Hardware graphs provide the definitions of the hardware resources available to the entire model. Software graphs represent the software instructions that control the usage of the hardware resources. Collapse graphs provide an elementary and experimental method of model simplification.

The hardware graphs are collections of resource nodes with no connecting arcs (similar to GPSM IPGs with the arcs removed). They may contain any nodes found in a GPSM IPG **except** CALL, CHANGE, ENTER, FORK, JOIN, RETURN, or SPLIT. All hardware graphs must have names beginning with the letters "HARD". Their node definitions are very similar to those of GPSM IPG nodes, except that any phase numbers appearing in the definitions are replaced by "mode" numbers, which are preceded by dollar signs.

The software graphs are similar in structure to GPSM IPGs, since they consist of nodes connected by arcs. However, they may contain only the following nodes: ENTER, RETURN, CODE (which looks like a CALL node), FOR (which looks like a FORK node), and

ENDFOR (which looks like a JOIN node). Software graphs may have names beginning with any letters except "HARD" and "COLL". Their node definitions consist of special statements in the PADS software language describing the usage patterns of the resources in the hardware graphs, as well as operations on user-controlled variables internal to the model. Their arc definitions describe the conditions under which a transaction would begin following the instructions in one node after finishing the instructions in a previous node.

The collapse graphs are similar to software graphs, but contain only ENTER, RETURN, and CODE nodes. Collapse graphs must have names beginning with the letters "COLL". They provide a very limited description of the usage patterns of SERVICE nodes, and the definitions of their nodes and arcs are similarly constrained.

Further details on the usage of these types of graphs may be found starting at Section 17 of this user's guide.

1.2. Files, Graphs, Nodes, and Arcs

[This section is basically the same as in [7], with the following exceptions:]

The term "graphs" is extended to include hardware, software, and collapse graphs, although only the last two types of graphs have arcs connecting their nodes.

Only software graphs may call or be called by other graphs. This is accomplished by using the CALL statement within a CODE node or other node of a software graph. This statement is defined with

the other constructs of the PADS software language in Section 17.2 of this user's manual. Recursive calls are permitted.

There is no such thing as a "main" graph. Transactions may enter the model at any point in any software graph where the SOURCE statement is encountered. The rate at which transactions enter, their categories, and the SOURCE statements at which they enter are determined by the definitions of the SOURCE *nodes* in the hardware graphs. Transactions leave the model when they encounter an instruction routing them to a SINK node.

2.3.2. Routing Nodes

FORK, JOIN, and SPLIT are not permitted in PADS hardware graphs, as this version of PADS does not support modeling of parallel execution of programs.

2.3.3. Arithmetic Nodes

Most of the functions of the COMPUTE node are available in the PADS software language constructs. However, the few that are not available (such as PRINT, NTOKEN, QL, etc.) may be accessed by creating a COMPUTE node in the hardware graphs and using it in the software code just like any other hardware resource. The global, local, and user-declared variables of PAWS are accessible through both the PADS software language and the COMPUTE node.

CHANGE nodes are not used in PADS hardware graphs. The reason is that PADS uses a transaction's phase to determine which hardware resource it will be sent to next, and thus PADS constantly changes the transaction's phase internally. Therefore, any phase change initiated at a hardware node would have no lasting effect, since the

phase would be automatically reset by PADS before the transaction could reach the next hardware node.

The phase of a transaction may be changed within a COMPUTE node by use of the statement "LETEQ TPHASE *whatever*", just as any other variable may be changed. The COMPUTE node may then be used within a software node in the same manner as any other hardware resource node. However, such changes last only until the next hardware resource is called, and so should be used only to implement branching conditions internal to the PADS statements in the software nodes.

2.3.4. INTERRUPT Nodes

The new phase assigned to a transaction by an INTERRUPT node lasts only until the transaction uses the next hardware node, as explained in 2.3.3., above.

2.3.5. USER Nodes

PADS runs a version of PAWS with a rather large USER node already implemented. This USER node, referred to in the resulting PAWS model as "QQUSER", is the central routing node that executes the PADS software code, and determines to which hardware nodes transactions are to be routed during execution of a model. The FORTRAN code for the main routine of this USER node is found in subroutine USS of the PAWS FORTRAN code, which is where all USER node code must reside. PADS reserves "user-function-codes" 1 and 2 for its own use. Therefore, if the user wishes to implement his own USER node functions in PADS, he must be careful not to overwrite the PADS USER-node code and to use user-function-codes other than 1 or 2.

2.3.6. Subgraph Nodes

Subgraph nodes (ENTER, RETURN, and CALL nodes) are not used in PADS hardware graphs. Each software and collapse graph in PADS must have exactly one ENTER node and one RETURN node. The CODE node has the same shape as the CALL node, and is used only in software and collapse graphs.

3. Introduction to PADS

[The word "IPG" is replaced throughout by the word "graph".]

In addition to the PAWS program file with the extension ".DAT", a second file is produced with extension ".SSS". This file is the ASCII representation of the software graphs, and is processed by the post-translator (outside the graphical interface) to produce a final file with extension ".UNI", which is the actual "User-Node-Interpretable Code", or UNIC, that the PADS USER node interprets at simulation time to determine the routing of transactions.

4.1 Command Line Format and Signon

PADS is invoked by typing

PADS *graphfilename*

in response to the operating system's prompt, where *graphfilename* has no extension.

7.4.2. Translating to a PAWS model

The PADS translator produces two output text files. One contains a PAWS program. Its name is the same as that of the graph file being edited but its extension (file type) is ".DAT". The other file that is created in the graph file directory is the software file, and it has the same name as the previous file but with the extension ".SSS".

During translation, error or warning messages may appear on the screen to indicate that the PADS translator found an error or irregularity in the set of graphs and where the problem was found. These messages disappear when translation is completed, but may be reviewed by running the translation again (without exiting PADS) or by examining the .DAT and .SSS files created by the translator (after exiting PADS). Not all errors can be detected by the current translator; some are detected by the post-translator (described below), some by PAWS at compile time, and some only while the PAWS model is running.

After PADS is exited, the software file must be retranslated into the code that the PAWS model can interpret during model simulation. This is accomplished by typing

mu *graphfilename*

in response to the operating system's prompt, where *graphfilename* has no extension. Mu is an abbreviation for "make UNIC," and is the post-translator for PADS software files. It calls a sequence of programs to produce three intermediate files, all of which have the extension ".PAD", and each of which requires the previous file for its own creation. If an error occurs at any point in the post-translation, mu halts execution, displays an error message, and tells the user in which file the source of the error may be found. Since the PADS software statements (as written by the user in the CODE nodes) are echoed in each file produced by mu just before their corresponding translations or error messages, the error is usually easily retraced to its source. If no errors occur, the two files with extension .DAT and .UNI may be uploaded or transferred to some other computer that runs PAWS. (Currently, the version of PAWS that runs PADS expects the UNIC file to have the name "unic". This

renaming is easily accomplished with a batch file command to copy the ".UNI" file to file "unic" before PAWS is run.)

12. Using the HELP Tool

Currently, the messages produced in PADS with the HELP tool are the same as those produced in GPSM. Therefore, the HELP message produced for some of the items may not be much help. Future versions of PADS will have the correct HELP messages.

13. Using the CRITIQUE Tool

The CRITIQUE tool has not been implemented in PADS (nor in GPSM).

16. On-Line Help

Using the function keys (F1, etc.) for **syntax help** produces syntax descriptions identical to those issued for GPSM. In most cases, these are almost the same as those needed for PADS. Future versions of PADS will have the correct syntax descriptions.

17. Node Specifications

17.1 Hardware Node Specifications

Each hardware node in a PADS model must have a unique name, which is entered by the user in the node's name field. Since the nodes generated internally by PADS have names beginning with the letters "QQ", the user should avoid names that begin with "QQ".

As in GPSM, the description field for a hardware node consists of a one-line comment and does not affect the node's meaning.

The specification fields for hardware nodes in PADS have a syntax that is very similar to that of their corresponding nodes in

GPSM. As in GPSM, the node name and type are omitted. Comment lines in hardware nodes must begin with an exclamation point (!), **not** a percent sign (%). The reason is that a percent sign in PADS indicates a user-defined constant (see Section 20 of this manual for more details). The phase number is replaced by the word "ALL" or by a "mode number," which is a dollar sign followed by a single digit. The mode number is used by the instructions in the PADS software graphs to determine which section of the hardware node specification will be followed; in this sense it acts like a PAWS phase. Categories are treated the same as in PAWS and GPSM. If a pound sign (#) appears anywhere in the REQUEST field, it is treated as a point of insertion of a parameter value, and is considered equivalent to its corresponding LR[] variable. (See the example that follows.) Semicolons need not appear in hardware nodes and are ignored if specified. Finally, if the word GEXPO appears on a line by itself in the REQUEST field of a SERVICE node, it adds the following lines to the hardware node specification:

```
<ALL, $6> CONSTANT ( LR[1] )
<ALL, $7> ERLANG ( LR[1] LR[2] )
<ALL, $8> EXPO ( LR[1] )
<ALL, $9> HYPER ( LR[1] LR[2] )
```

The purpose of GEXPO (Generalized EXPOnential) is to allow the user to refer to the usage of SERVICE nodes in terms of mean and coefficient of variation instead of mean and standard deviation. It is also used to allow service resources to be included in a collapse graph, since the collapse graph assumes all resources to have CONSTANT usage under GEXPO (with coefficient of variation equal to 0). GEXPO is not permitted for nodes other than SERVICE nodes. (Since the collapse graphs may be eliminated in future versions of PADS, and the

GAMMA distribution in version 3.0 of PAWS provides basically the same function as GEXPO, the GEXPO construct may be eliminated as well in future versions of PADS.)

Notice that in the GEXPO example, if all instances of LR[1] and LR[2] were replaced by pound signs, as in ... CONSTANT (#) ... ERLANG (# #) ... etc., the code would have the same meaning. This illustrates the use of pound signs in providing parameters for hardware node usage. For example, if we define a SERVICE node, named DISK, as follows:

```
! This is a comment.
DIMENSION 3 ! Declares DISK[1], DISK[2], DISK[3].
QUANTITY 1
QD DELAY
REQUEST <BATCH, $1> CONSTANT(#)
        <INTER, $1> ERLANG(# #)
        <ALL, $2> UNIFORM (1.0, #)
        <ALL, ALL> EXPO (5.0)
```

then DISK may be used within the PADS **software** graphs as follows:

USE DISK[1] ($1, 3.0)
: Uses DISK[1] for CONSTANT (3.0), assuming the transaction's category is BATCH. The value 3.0 is passed to the hardware node through the transaction's local real variable LR[1].

USE DISK[1] ($1, 3.0, 0.5)
: Uses DISK[1] for ERLANG (3.0, 0.5), assuming the transaction's category is INTER. The value 3.0 is passed through LR[1], and 0.5 is passed through LR[2].

USE DISK[3] ($2, 8.5)
: Uses DISK[3] for UNIFORM (1.0, 8.5) regardless of the transaction's category. Again, 8.5 is passed through LR[1].

USE DISK[2] ($3) *or* USE DISK[2] ($4) , *etc.*
Uses DISK[2] for EXPO(5.0), regardless of the transaction's category. No values are passed to any LR[] variables. As DISK is defined above, $3 or $4 could be replaced by any mode number other than $1 or $2, and the statement would have the same effect.

If the last line in the definition of DISK were replaced by the word "GEXPO", the following uses of DISK could also be made:

USE DISK[2] (GEXPO, 5.0, 0.0)
Uses DISK[2] for CONSTANT(5.0), since the c.v. (coefficient of variation) is equal to 0.

USE DISK[2] (GEXPO, 5.0, 1.0)
Uses DISK[2] for EXPO(5.0), since the c.v. is equal to 1.0 (c.v = standard deviation (s.d.) divided by mean).

USE DISK[2] (GEXPO, 5.0, 0.5)
Uses DISK[2] for ERLANG (5.0, 2.5), since c.v. (0.5) is less than 1.0, and s.d. (2.5) equals mean (5.0) times c.v. (0.5).

USE DISK[2] (GEXPO, 5.0, 1.5)
Uses DISK[2] for HYPER (5.0, 7.5), since c.v. (1.5) is greater than 1.0, and s.d. (7.5) equals mean (5.0) times c.v. (1.5).

When using GEXPO as part of a hardware SERVICE node definition, care must be taken not to define other parts of the node that will interfere with the automatically defined GEXPO sections mentioned above. In particular, a section defined as <ALL,ALL> will produce a PAWS compilation error when combined with the GEXPO sections, since those sections always appear **after** any other defined sections. See the PAWS user's manual [12] for more details on the meaning and proper placement of the keyword ALL.

While GEXPO cannot be specified in the definition of non-SERVICE hardware nodes, the "mode" numbers are used in the same way. For example, if the ALLOCATE node named "GETTOKEN" is defined as:

```
QUANTITY 10 MSGBUF
QD FCFS
REQUEST <ALL, $1> CONSTANT(3.0)
        <ALL, $2> CONSTANT(#)
```

then the PADS software statement "USE GETTOKEN ($1)" would cause three tokens of type MSGBUF to be allocated to the current transaction, while "USE GETTOKEN ($2,4)" would cause four of those tokens to be allocated.

The word USE is always optional within a PADS software statement, so that "GETTOKEN($1)" means the same as "USE GETTOKEN($1)", and "USE DISK[2] (GEXPO, 5.0, 1.0)" means the same thing as "DISK[2] (GEXPO, 5.0, 1.0)". Further details on PADS software syntax may be found in Section 17.2 of this user's manual.

The SOURCE node is the only hardware node that has no PADS mode number specified in place of a GPSM phase number. Its syntax is identical to that for the SOURCE node in GPSM; however, the "phase" number is treated as an "entry point" number. The entry point number corresponds to the identical number following a SOURCE statement in the PADS software graph. For example, if the SOURCE node named SRC is defined as:

```
REQUEST <BATCH, 3> EXPO(1.0)
        <BATCH, 4> CONSTANT(6.0)
        <BATCH, 7> HYPER(1.0, 10.0)
        <INTER, 4> EXPO(2.0)
```

then the PADS software statement "SOURCE 3" will cause BATCH transactions to enter the software model at an exponentially distributed rate with a mean value of 1.0. These transactions will begin executing the software statements following the "SOURCE 3" statement. Similarly, the statement "SOURCE 4" will cause transactions of type BATCH and INTER to enter the software model at their respective rates of arrival, all of which will begin executing the software statements following the "SOURCE 4" statement. While entry point numbers need not be unique within or among SOURCE *nodes*, they must be unique within the set of all PADS software SOURCE *statements* within a model (e.g., the statement "SOURCE 4" may not appear more than once throughout the model). The entry point number must be a literal positive integer; currently it may have a value between 1 and 50 inclusive, the upper limit being determined by the value of the FORTRAN parameter "mxinjb" in subroutine "uss".

Notice that the name of the SOURCE node is not referenced in the SOURCE statement. It would have made no difference if the four <category, entry-point> clauses had been placed in separate SOURCE nodes. In fact, the use of multiple SOURCE nodes (which must, of course, have distinct names) can act as an aid in the gathering and grouping of statistics, as can the use of named SINK nodes and BRANCH nodes in the PADS hardware graphs.

17.2 Software Node Specifications

Software nodes need not have unique names, or any names at all. Any names or descriptions specified are for the user's own documentation purposes only.

The specification fields for most software nodes in PADS contain statements in the PADS software language. The ENTER, EXIT and CODE nodes contain PADS software statements which may optionally be grouped into category sections. Category sections are explained later in this section of the manual.

The FOR nodes must contain exactly one FOR statement that is not paired with an ENDFOR statement; the ENDFOR nodes may contain at most one unpaired ENDFOR statement (if none is found, an ENDFOR is inserted at the end of the node during translation). FOR and ENDFOR nodes may contain other statements as well, but it is customary for them to contain only the statements just described. If a FOR node contains additional statements, only the ones *after* the unpaired FOR statement may be grouped into category sections. If an ENDFOR node contains statements that are grouped into category sections, they must all appear *after* an unpaired ENDFOR statement, which must then be explicitly written.

The following list describes the statements available in the PADS software language. Portions in braces {} are optional; the braces themselves are **not** included in the statements. PADS keywords are in UPPERCASE and should be typed as shown (although PADS will accept either upper- or lowercase); words in *lowercase italics*, possibly containing hyphens, are defined following each statement and symbolize other constructs to be entered at that point in the statement. If a set of symbols appears between the pairs of characters "[+" and "+]", as in [+ *construct* +], it means that the *construct* may be repeated one or more times; the "[+" and "+]" themselves are **not** included in the statements. Quotation marks are never included in PADS software

statements. Other symbols (parentheses, pound signs, etc.) are to be entered as shown.

IF *condition* {THEN}
statement-list
{ELSE
statement-list }
ENDIF

A *statement-list* is a series of one or more of the PADS software language statements, and *condition* may have one of the following forms:

value arithcomp value
{NOT} *boolean*
{NOT} *boolean boolcomp* {NOT} *boolean*

The entire *condition* may be surrounded by parentheses, but parentheses may **not** be used to separate the parts of a condition, including the word "NOT", from each other. Therefore, the parentheses may be omitted without changing *condition*'s meaning.

A *value* is a *numeric* or a *distribution*.

A *numeric* may be any one of the following: LI[*sub*], LR[*sub*], GI[*sub*], GR[*sub*], RANDOM, TIME, TID, TPHASE, *unsigned-integer*, *unsigned-real*, *identifier*, or #*identifier*. The last construct in this list, #*identifier*, corresponds to the parameter of the same name in the PARMS statement (described later) for the current software graph. The construct before it, *identifier*, refers to a user-declared integer or real variable, which must be declared in the specification of the current PADS file (see Section 20 of this user's manual). The *unsigned-integer* and *unsigned-real* are strings of digits (and other characters, in the *unsigned-real*) of up to fourteen characters representing legal integer and real values, respectively. The *unsigned-real* may be written in

FORTRAN "E-format" as well. A user-defined constant (percent sign followed by an identifier) of type integer or real may always be substituted for an *unsigned-integer* and *unsigned-real*, respectively. The other *numeric* constructs refer to the same variables as their PAWS counterparts, and are explained in [12]. However, the local integer variables LI[1] through LI[4] have uses internal to PADS, and thus should not have their values altered by any software statements. A *sub* may be an *unsigned-integer* or an *identifier* representing a user-declared integer or real (if real, its value is automatically truncated). Finally, a *numeric* may always be preceded by a *prefix*. A *prefix* is an optional plus or minus sign, optionally followed by "FIX" or "FLOAT". Parentheses are **not** used after the *prefix*. The word "FLOAT" has no effect, since type conversion from integer to real is performed automatically in PADS. The word "FIX" is used to truncate a real value, but it must **not** be used within a *sub*.

A *distribution* may be any of the following:

UNIFORM (*numeric* , *numeric*)
HYPER (*numeric* , *numeric*)
ERLANG (*numeric* , *numeric*)
EXPO (*numeric*)
CONSTANT (*numeric*)

preceded optionally by a *prefix*, as explained above. The commas are optional; the parentheses are mandatory. They have the same meaning as in PAWS. Other PAWS distributions (such as EMPIRICAL) may have their values computed by using a COMPUTE node in the hardware graphs (see the USE statement, below).

An *arithcomp* is one of the following six symbols: = (is equal to), > (is greater than), < (is less than), <> (is not equal to), <= (is

not greater than), >= (is not less than). The last three symbols may not contain any embedded spaces.

A *boolean* may be any of the following: TRUE, FALSE, GB[*sub*], GI[*sub*], or a user-defined constant of type boolean. These symbols have the same meaning as in PAWS.

A *boolcomp* is one of the following four symbols: AND, OR, = (is equivalent to), <> (is not equivalent to). These symbols may not contain any embedded spaces.

The IF statement has the usual semantic effect, similar to that of the structured IF in FORTRAN-77. IF statements may be nested.

LET *boolvar* = *condition*
LET *numvar* = *value* {*op value*}

A *boolvar* is either GB[*sub*] or GI[*sub*]. A *numvar* is one of the following: LI[*sub*], LR[*sub*], GI[*sub*], GR[*sub*], or an *identifier* representing a user-declared integer or real variable. An *op* is one of the following four symbols: + (addition), - (subtraction), * (multiplication), or / (division). The keyword "LET" may **not** be omitted. The "=" symbol may be replaced by ":=" (as in Pascal) if desired.

The LET statement assigns the value of the expression on the right side of the "=" (or ":=") symbol to the variable on the left side.

USE *hw-resource* { [*numeric*] } { (*parameter-list*) }

The keyword "USE" is optional and may be omitted. The *hw-resource* is the name of a hardware resource node that has been defined by the

user in the hardware graph. If specified, [*numeric*] refers to the subscript of a hardware resource node with DIMENSION greater than one. The *parameter-list*, if specified, is of one of the two forms

({$*mode*} {*numeric-list*})

or

(GEXPO {,} *numeric* {,} *unsigned-real*)

The second form (GEXPO...) was explained in Section 17.1 of this user's manual. In the first form, $*mode* is a dollar sign ($) followed by a single digit, and *numeric-list* is a list of *numerics*, separated by commas or blanks. Either $*mode* or *numeric-list* may be omitted in the first form; if both are omitted, then so is the entire *parameter-list* (i.e., no empty parentheses pairs () are allowed). The $*mode* corresponds to the same number in the hardware resource definition and indicates which portion of the definition is to be used by the transaction. If no modes are specified in the hardware node, or if the only mode specified is "ALL", then $*mode* should be omitted from the *parameter-list*. The items in *numeric-list* refer, in order, to the values to be substituted for the "pound signs" in the corresponding hardware resource node definition, as explained in Section 17.1 of this user's manual.

FOR (*count*) {TIMES}
statement-list
ENDFOR

The *count* is a *numeric* or a *distribution* as defined above, and *statement-list* is as defined above. FOR statements may be nested. The meaning of this statement is similar to that of "FOR I := 1 TO *count* DO *statement-list*" in Pascal. The value of *count* is computed and truncated to an integer, and if the result is less than one, the construct has no effect and is skipped. Otherwise, the instructions in *statement-list* are executed *count* times.

Since the FOR statement contains no explicitly defined index variable, the user may access the number of iterations yet to be executed in the current FOR statement with the identifier "LOOPCOUNT". "LOOPCOUNT" may be used, within the *statement-list* of a FOR statement, anywhere that an *identifier* representing a user-defined integer variable may be used, except that its value may not be altered (e.g., LOOPCOUNT may not be used on the **left** side of the equals sign in a LET statement.) LOOPCOUNT refers to the number of iterations (not including the current one) yet to be performed in the innermost FOR statement at the instruction where LOOPCOUNT is found. If the FOR statements are nested, and the user needs to access this same value for an outer FOR statement while within an inner one, he will need to use an expression similar to "LET LI[5] = LOOPCOUNT" while in the outer FOR statement to save its value for access in the inner one. (FOR statements may be replaced with IF statements and GOTOs, if explicit index variables are required.)

GOTO *number*

LABEL *number*

Each *number* is an unsigned integer between 0 and 32767, which must be unique for each LABEL statement within a given software node. The GOTO statement passes control to the LABEL statement with the corresponding *number* in the **same** software node; a GOTO may **not** pass control to a statement in another software node.

```
   RAND
[+ probability : statement-list ; +]
 { OTHERS : statement-list {;} }
   ENDRAND
```

Each *probability* is a *numeric* or a *distribution* as defined above, (but

representing a real numerical value between 0.0 and 1.0), and *statement-list* is defined as above (except that RAND constructs may not be nested). The colons are optional. The semicolons are required, except that the semicolon before the word ENDRAND may be omitted. The semicolons indicate the end of each statement-list, and signal to the PADS compiler that a *probability* (or the word "OTHERS") is the next thing that will be written; they are **not** use to separate the statements within a *statement-list*. If the optional OTHERS clause is included, the probabilities must sum to not more than 1.0; otherwise, they must sum to exactly 1.0. One of the *statement-lists* is chosen and executed at random; the probability that a *statement-list* is chosen is equal to the *probability* value preceding it, the value of OTHERS being considered equal to 1.0 minus the sum of the other probabilities in the RAND construct.

PARMS ([+ *#identifier* +])

The *#identifiers* are separated from each other by commas or by blanks. Each *#identifier* in the list refers to a parameter of the current software graph. They are given values when the software graph is called by the CALL statement (described below) in another software graph; they may not be assigned values directly by any other statement within a software graph. (If different values are to be returned to a calling graph, the unused local variables of a transaction may be used for this purpose.) The PARMS statement need not be the first statement encountered by a transaction in the called software graph, but it must be encountered by the transaction before any other statement containing one of the *#identifiers* is encountered. If the called software graph has no parameters, the entire PARMS statement should be omitted.

CALL *sw-module-name* { (*value-list*) }

The *sw-module-name* is the name of any other declared software graph. If specified, (*value-list*) is a list of one or more *values* separated by commas or blanks, and enclosed in parentheses. A *value* is a *numeric* or a *distribution.* This statement causes control to pass to the first line of the ENTER node of the called graph. The *values* in *value-list*, if it is specified, are assigned in order to the parameters specified in the PARMS statement of the called software graph; if the called graph has no PARMS statement, then *value-list* and its surrounding parentheses should be omitted as well. Control returns to the line after the CALL when complete.

EXIT

causes a transaction to leave the current software graph with no further processing. If the current graph was CALLed from another graph, control passes to the statement following that CALL statement. If no graph called the current one, the EXIT statement produces an "empty stack" run-time error in PADS.

SINK

causes a transaction to be routed to the default hardware SINK node. Therefore, this statement is equivalent to "{USE} QQSINK", since QQSINK is the name of the default hardware SINK node in PADS. Routing a transaction to any SINK node causes it to clear all its run-time and parameter stacks and immediately leave the entire model with no further processing.

Comments are preceded by an exclamation point and continue to the end of the line.

As mentioned earlier in this section, PADS software statements may be grouped by category sections within each software node. Each category section begins with the words "CAT *cat-id*", although the last category section in a node may begin with the word "ALL" instead. Each *cat-id* is the name of a category that has been declared in the CATEGORIES section of the current PADS file specification, as explained in Section 20 of this user's manual.

Here is an illustration of the use of category sections within a software node:

```
!Beginning of software node
    statement-list-1
    CAT BATCH
    statement-list-2
    CAT INTER
    statement-list-3
    ALL
    statement-list-4
!End of software node
```

In the above illustration, a transaction entering the software node first executes all statements in *statement-list-1*. If the transaction's category is BATCH, it will then execute all statements in *statement-list-2* and leave the software node. However, if the transaction's category is INTER, it will execute all statements in *statement-list-3* and leave the software node. Finally, if the transaction's category is anything but BATCH or INTER, it will execute all statements in *statement-list-4* and leave the software node.

This illustration assumes that flow of control is not interrupted by GOTOs, SINKs, etc. within the *statement-lists*. If *statement-list-2* above is replaced by the statement "GOTO 10" and the first statement

in *statement-list-3* is "LABEL 10", then all transactions of category BATCH or INTER will execute *statement-list-1* and *statement-list-3* and leave the software node. The same would be true if *statement-list-3* above is replaced by the statement "GOTO 10" and the first statement in *statement-list-2* is "LABEL 10". (This would **not** result in an infinite loop, since every "CAT" or "ALL" statement **after** the first one in the node causes a transaction whose category has already been matched to leave the software node.) On the other hand, if *statement-list-2* is deleted in the original illustration, then a transaction of category BATCH will execute only *statement-list-1* before leaving the software node.

The limitations on the use of category sections in FOR and ENDFOR nodes were described at the beginning of this section.

17.2.1 Software Arc Specifications

Software nodes are connected by software arcs. The labels and specifications of software arcs define under what conditions a transaction will begin executing the statements in a software node after leaving another software node. Each arc has exactly one source node (or "from-node") and one destination node (or "to-node").

Several types of arcs are available, representing different types of branching or flow control.

An **unconditional branch** (or GOTO) from a node is represented by a single arc from that node to the destination node. This arc has an empty specification. If software node A is connected to software node B by an unconditional branch, then all transactions finishing the instructions in A will begin executing the instructions in B.

A conditional branch from a node A to a set C of nodes is represented by a set of "IF-arcs," each of which has A as its from-node and a distinct node in C as its to-node. The specification of each of the IF-arcs begins with the word "IF" or "IF*nnn*", where *nnn* is a series of one to three decimal digits; however, exactly one of the IF-arcs may begin with the word "ELSE", which is considered equivalent to "IF999". The purpose of *nnn* is to control the order of evaluation of the IF-arcs. If *nnn* is omitted, it is considered equal to zero. If more than one arc has the same value for *nnn*, e.g., if all arcs begin with "IF", the PADS translator fixes an arbitrary order for their evaluation. However, no more than one IF-arc from a node may have *nnn* equal to 999 (i.e., a set of IF arcs may have no more than one "ELSE" arc). Following the word "IF" or "IF*nnn*" (but not the word "ELSE" or "IF999") is a *condition* as defined in Section 17.2 of this user's manual, defining under what condition a transaction will follow that arc after completing the instructions in node A. At run time (i.e., while the PADS model is being simulated), the conditions are evaluated in the order described above, and the first condition that evaluates to TRUE results in the transaction following the corresponding arc. If no condition is found to be TRUE, the "ELSE" or "IF999" arc is followed if there is one; if there is no such arc, a PADS run-time error ("ERR=NOELSE") occurs, and the model simulation terminates.

A probabilistic branch from a node A to a set C of nodes is also represented by a set of arcs, called "RAND arcs." The specification of each RAND arc in the set is a *probability* (i.e., a *numeric* or a *distribution*) as defined in Section 17.2 of this user's manual; however, at most one arc in the set may have the word "OTHERS" instead of a *probability*. The result of adding the values of each *probability* in a set of

RAND-arcs must be exactly 1.0, unless an arc marked "OTHERS" is included in the set, in which case the sum may be no more than 1.0. The probability that an arc is traversed is equivalent to the value of its attached *probability* at run time. If no "OTHERS" arc is included and the sum of the *probability* values at run time is less than 1.0, a PADS run-time error "ERR=NOPROBAB" may result.

A software graph containing FOR and ENDFOR nodes has two additional types of arcs: LOOP arcs and EXIT arcs. A LOOP arc is a specially labeled arc from an ENDFOR node to its corresponding FOR node. The first four letters in the **label** of a LOOP arc must be "LOOP". As long as the FOR-loop is to be executed by a transaction, that transaction follows the LOOP arc when it leaves the ENDFOR node. After its last pass through the loop, the transaction follows a different arc from the ENDFOR node, depending on the type of branching represented by the other arcs from that ENDFOR node. An EXIT arc is a special case of a RAND-, IF-, or GOTO-arc, one which has "EXIT" as the first four letters of its label. This label is **required** whenever the traversal of an arc results in the **premature exit** of a FOR-loop.

A single physical arc between two nodes may represent several different types of branching, i.e., one type of branching for each of several different transaction categories. For example, if the arc from node A to node B has the following specification:

```
CAT BATCH IF GI[2] > 4
CAT INTER 0.45
ALL IF LB[1]
```

and the arc from node A to node C has this specification:

```
CAT BATCH ELSE
ALL ELSE
```

and the arc from node A to node D has this specification:

```
CAT INTER OTHERS
```

and there are no other arcs leaving node A, then all transactions leaving node A will act as follows: The BATCH transactions will go to node B if GI[2] is greater than four, and to node C otherwise; the INTER transactions will go to node B with probability 0.45, and to node D with probability 0.55; the transactions of all **other** categories will go to node B if LB[1] is true, and to node C otherwise.

The word "ALL" may be specified in place of "CAT *category-name*" to refer only to categories that have not been specified in **any** other arc from the same from-node. In the specification of the arc from A to C, "ALL" cannot refer to category INTER, since this category is specified in at least one other arc from node A. Mixing IF- and RAND-arc specifications from the same node for the same category results in an error at translation time.

A LOOP arc is the only arc that may not have "CAT" or "ALL" (or any other significant words) in its specification.

17.3 Collapse Node and Arc Specifications

A collapse graph has exactly one ENTER node and one RETURN node, both of which have blank specifications. The other nodes are all of type CODE and have specifications whose lines are all of the following form:

freq \ *resource-id* { [*numeric*] } (*total-req*)

Here, *resource-id* is the name of the hardware resource being used

(which must be of type SERVICE and have the word "GEXPO" as part of its specification), [*numeric*] is the subscript (if any) of that resource, *total-req* is an *unsigned-real* representing the total amount of service time that a transaction requests from that resource while (conceptually) within the collapse node, and *freq* is an *unsigned-integer* representing the number of times the transaction visits that resource while within the collapse node. The letter "C" may be used in place of *freq*, in which case the SERVICE resource is treated like a CPU and is requested n+1 times, where n is the sum of the *freq*s (not including "C"s) in that collapse node.

The arcs connecting the nodes in a collapse graph all have specifications consisting of *unsigned-real*s whose values are between 0.0 and 1.0; a blank specification defaults to 1.0.

[The method of collapsing and an example of a collapsed graph are described in this thesis.]

18. Arc Specifications

[These are explained in Sections 17.2.1 and 17.3 of this user's manual.]

19. Graph Specifications

There may be specifications attached to each graph in PADS. The primary use of graph specifications is to store comment lines explaining the graph or its purpose. The specifications (and description fields) of hardware, software, and collapse graphs are ignored by PADS, and may contain any text whatsoever. The name field of a graph is used to determine the type of a graph: hardware graphs must have names beginning with the letters "HARD", collapse graphs must have

names beginning with the letters "COLL", and software graphs must have names beginning with neither "HARD" nor "COLL". In addition, the name of a software or collapse graph allows it to be called by a software "CALL" statement; the name of a hardware graph has no further significance.

20. File Specifications

The specifications attached to a file in PADS are used to provide information about the model as a whole. In particular, the file specification contains declarations for objects used anywhere within the hardware, software, or collapse graphs of the PADS model. The detailed syntax for each of the declarations listed below (except CONSTANTS) is identical to that used in PAWS and is described in [12]. The overall syntax for the file specification is as follows:

```
CONSTANTS constant-list ;
OPTIONS EFORMAT ;
INTEGERS    [+ int-scalar +] ;
REALS       [+ real-scalar +] ;
CATEGORIES [+ category +] ;
TOKENS      [+ token +] ;
MEMORIES    [+ memory +] ;
INITIAL     pop ;
STATISTICS report ;
RUN         run-stmt ;
```

Any combination of these sections may appear in the file specification and in any order. None of them are required. However, each of the keywords that begin a section (CONSTANTS, OPTIONS, INTEGERS, etc.) must begin the line on which they are placed, and each section must end with a semicolon. Comments in the specification begin with an exclamation point and continue to the end of the line.

The syntax for the *constant-list* is a series of constant definitions, each of which has the form

%identifier = constant-value

where *constant-value* may be an *unsigned-integer*, an *unsigned-real*, the string "TRUE", or the string "FALSE" (the quotation marks are **not** entered as part of the definition). Therefore, the form of the *constant-value* implicitly determines whether the constant is integer, real, or boolean. Constant definitions should not be split across lines, although there may be more than one constant definition per line. The percent sign at the beginning of a constant name is part of the name and is included whenever the constant is used in the model; it signals the PADS translator to replace that *%identifier*, wherever it appears, with the appropriate *constant-value.* Therefore, "%NUMSER", a constant, is not the same as "NUMSER", a user-defined variable, and both may be used within the same model. Constants may not be redefined or have new values assigned to them anywhere in the model.

Bibliography

1. Aho, Alfred V. and Ullman, Jeffrey D. *Principles of Compiler Design.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.

2. Anderson, Gordon E. "The Coordinated Use of Five Performance Evaluation Methodologies". *Communications of the ACM 27*, 2 (February 1984), 119-125.

3. Chandy, K.M., et al. The Use of Performance Models in Systematic Design. Proceedings of the National Computer Conference, AFIPS, Houston, June, 1982, pp. 251-256.

4. Denning, Peter J. and Buzen, Jeffrey P. "The Operational Analysis of Queueing Network Models". *Computing Surveys 10*, 3 (September 1978), 236-237.

5. Franklin, Jeffrey L., et al. Software Analysis Tools: A Method for Developing Performance Model Inputs. Proceedings of CMG 85, Dallas, December, 1985, pp. 400-410.

6. Garbo, Martin J. and Bingham, Paris E. Disk System Modeling in a CYBER Environment. Proceedings of the CMG XV International Conference, Computer Measurement Group, December, 1984, pp. 181-188.

7. *GPSM User's Manual.* Information Research Associates, Austin, Texas, 1987.

8. Information Research Associates. Graphical Programming for Simulation Models of Computer Systems. Final Project Report, NSF SBIR - 1983 Phase I, Award Number DCR-8360779, Information Research Associates, July 31, 1984.

9. Kelly, John Clifford. *The Theory of Repetition Networks with Application to Computer Programs.* Ph.D. Th., Purdue University, December 1974.

10. Neuse, D. M. and Browne, J. C. High Level Simulation of Electronic Systems. Final Report, Contract Number N60921-85-C-A026, Report Number A003, Information Research Associates, July 26, 1985.

11. Neuse, D. M., et al. Definition of Software Module Interface, Task 1 of Navy SBIR Phase II Project: High Level Simulation of Electronic Systems. Technical Interim Report, Contract Number N60921-86-C-0145, Report Number A001, Information Research Associates, October 15, 1986.

12. *PAWS 2.0 User's Manual.* Information Research Associates, Austin, Texas, 1984.

13. Seelinger, Deborah J. Application of PAWS in the Sperry Univac Environment. Proceedings of the CMG XIII International Conference, Computer Measurement Group, December, 1982, pp. 200-219.

14. Shampine, Lawrence F. and Allen, Richard C., Jr. *Numerical Computing: An Introduction.* W. B. Saunders Company, Philadelphia, 1973. pp. 252-255.

15. Smith, Connie U. and Browne, J. C. Performance Specfications and Analysis of Software Designs. Proceedings of the Conference on Simulation Measurement and Modeling of Computer Systems, Boulder, August, 1979.

16. Smith, Connie Umland. *The Prediction and Evaluation of the Performance of Software from Extended Design Specifications.* Ph.D. Th., The University of Texas at Austin, August 1980.

17. Smith, Connie U. Software Performance Engineering. Proceedings of the Computer Measurement Group Conference XII, December, 1981, pp. 5-14.

18. Smith, C. U. and Browne, J. C. Performance Engineering of Software Systems: A Case Study. Proceedings of the National Computer Conference, AFIPS, Houston, June, 1982, pp. 217-224.

19. Smith, Connie U. Experience with Tools for Software Performance Engineering. Proceedings of CMG 85, Dallas, December, 1985, pp. 411-417.

20. Smith, Connie U. The Evolution of Software Performance Engineering: A Survey. Proceedings of the Fall Joint Computer Conference, Dallas, November, 1986, pp. 778-783.

21. Smith, Connie U. "Performance Engineering: A Bibliography". *CMG Transactions 55* (Winter 1987), 115-122.

22. Upchurch, E. T. Modeling Packet Switched Interprocessor Communications. Proceedings of the 15th Annual Modeling and Simulation Conference, April, 1984.

23. Upchurch, E. T. Top-Down Performance Modeling of a Large C^3I Hardware/Software System Using PAWS. Proceedings of the 15th Annual Modeling and Simulation Conference, April, 1984.

VITA

Dana Mark Whiting was born in Cincinnati, Ohio, on June 27, 1959, and is the son of Margaret Puccini Whiting and Dana Cutler Whiting. After completing his work at Anderson High School, Cincinnati, Ohio, in 1977, he entered Wabash College in Crawfordsville, Indiana. He received the degree of Bachelor of Arts, Summa Cum Laude, from Wabash College in May, 1981, and entered the Graduate School of the University of Texas with a University Fellowship in August of that year. While attending school, he worked as a teaching assistant from 1982 to 1986, a Summer Associate for IBM Corporation from 1982 to 1984, and a Senior Analyst for Information Research Associates from 1984 to 1987. He is a member of Phi Beta Kappa, Phi Kappa Phi, and the Mathematical Association of America.

Permanent Address: 502 Longspur Blvd., Apt. 6-203
Austin, Texas 78753

This thesis was typed by the author.