CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

SPECIFICATION AND DESIGN OF MULTIPLE MICROPROCESSOR SYSTEMS USING PROCESS ACTIVITY DIAGRAMS

by Robert Joannis

A thesis presented to the
School of Graduate Studies and Research
of the University of Ottawa
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in Electrical Engineering

OTTAWA, Ontario, August 1986

ISBN 0-315-36514-5

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The University of Ottawa requires the signatures of all persons using or photocopying this thesis. Please sign, below, and give address and data.

ABSTRACT

The developments in the area of microprocessors have permitted the introduction of digital processing capabilities to virtually any system. This has forced many system designers, who are not computer specialists, to design microprocessor-based systems into their product. The tools available for the design of microprocessor-based systems have evolved from classical computer design methodologies and are not well suited for non-computer specialists. In this thesis, two design methodologies targeted for product designers are proposed. The first design methodology is based on finite state machines and is applicable to simple sequential event response systems. For more complex systems, that have more stringent real-time requirements, a new graphical tool, Process Activity Diagrams (PAD's) is proposed. For both of these methodologies, programs are developed which simulate the behavior of the system, and help the designer verify if his system can meet real-time constraints before it is actually implemented.

ACKNOWLEDGMENTS

The author wishes to thank his thesis advisor, Dr. M. Krieger, whose guidance and encouragement throughout this research work is greatly appreciated. The numerous discussions that were held with him have sparked many of the ideas that are presented in this thesis.

The author is also very grateful for financial assistance provided by the National Sciences and Engineering Research Council of Canada and the University of Ottawa.

Also acknowledged, is the help of research colleagues, in particular, Mr. Jose M. Duran, Mr. Charles-Antoine Gauthier and Mr. Vojislav Radonjic, whose constructive criticisms have contributed to the soundness of this presentation.

Finally, the author wishes to thank his wife, Lisa, for her constant encouragement and understanding during his studies. Her help in preparing the figures in this thesis is most appreciated.

TABLE OF CONTENTS

page

LIST OF FIGURES

page

CHAPTER 2

CHAPTER 3

page

CHAPTER 4

CHAPTER 5

page

CHAPTER 6

LIST OF TABLES

page

CHAPTER 2

CHAPTER 6

CHAPTER 1

INTRODUCTION

With the development of microprocessors it has been possible to add digital processing capabilities to virtually any system. One major application of these inexpensive devices is adding automatic control to many systems which were previously controlled manually or by analog devices. For instance, in manufacturing plants, programmable controllers are used to replace the inflexible relay circuits to control various assembly lines, chemical processes, etc... Other more complex applications include various special purpose simulators, telephony equipment such as Private Branch Exchanges (PBX's), etc... All these special purpose systems have quite different requirements than classical computer systems and are often designed by specialists in the application area of the system rather than computer specialists.

These developments in technology have not been paralleled by significant developments in the area of simple design methodologies for microprocessor-based systems, leaving the system designer with ad hoc design methodologies. This lack of simple design methodologies has lead to high design costs, reliability problems, and often prevented projects from being completed. In this thesis, current design methodologies for microprocessor-based systems are surveyed and new tools are

proposed which can simplify the task of the system designer.

To present these ideas, this thesis is divided into five major chapters (numbered from 2 to 6) whose outline is as follows:

Chapter 2 covers general design aspects of microprocessor-based systems by first introducing the hardware and software components of these systems. Microprocessor-based systems are then classified, based on their application area, to properly define the types of systems which are considered in this thesis. These are event driven systems which include real-time special purpose systems. Next, a design methodology based on finite state machines is proposed for simple sequential event response systems. These are single microprocessor systems, or even single chip microcomputers or microcontrollers, which respond, one at a time, to events occurring in their environment. A simulation program was written for these systems to verify their behavior and some results of the simulation are given to show their limitations in terms of real-time response. When real-time constraints cannot be met with such systems, concurrent systems must be used. For this reason, some elements of real-time multitasking operating systems are also introduced in this chapter. Finally, the overall system design process and the need for graphical design tools for concurrent systems is presented in the last section of this chapter.

In chapter 3, some of the current graphical design tools for concurrent systems are briefly presented. These tools are

analysed in terms of their usefulness for the specification and design of event driven systems by non-computer specialists. A system can be viewed at different levels of abstraction, but not all tools allow such a representation; many can only describe the system at a particular level. To show this, the tools discussed in this chapter are divided into system level tools, software structure diagrams and internal task structure diagrams. System level tools can describe systems at various levels of abstractions and do not deal explicitly with lower level multitasking concepts. Two such tools are presented: Petri nets and data flow graphs. The emphasis of the chapter is on Petri nets because of their wide acceptance in the description of concurrent systems. The other tools discussed in this chapter are closely tied to multitasking systems as they either describe the software structure of the system by showing the interrelations between the tasks in a multitasking system, or, at an even lower level of abstraction, they directly represent the internal structure of the tasks.

In chapter 4, a new tool for the specification and design of concurrent systems, Process Activity Diagrams (PAD's), is presented. After defining some basic terminology, the two components of PAD's, Activity Sequence Nets (ASN's) and Activity Timing Charts (ATC's), are discussed separately. Under this design methodology, a system would be partitioned in a top down format, starting from the various system processes until it is broken down into activities. Conceptually, these activities can be considered as a piece of code that simply receives some input

data, manipulates it through various computations and returns a result, just like a subroutine. The ASN's are then used to express the sequencing relationships between these activities while the ATC will show their timing requirements. To conclude this chapter, PAD's are compared to some of the tools presented in the previous chapter, particularly with Petri nets because the various nodes of the ASN's can be directly mapped onto Petri nets.

Chapter 5 discusses the use of PAD's in the design of event driven systems. In the first part, the use of PAD's as a system specification tool is presented. To show exactly how this is done, an actual system, an automatic sorting conveyer, is specified using PAD's. When the system specification is completed, PAD's can be used to carry this specification into an implementation. However, this cannot be done directly if the system will be implemented in a multitasking environment because the building block in PAD's is the activity, which is somewhat different from a task. It is thus shown how PAD's can help the system designer map the various activities of the system onto tasks which will be run under a multitasking operating system. If the system has already been specified with PAD's, the design process could be greatly simplified if this translation of activities into tasks could be eliminated. The last section of this chapter discusses this prospect by introducing the idea of a multi-activity executive which would be able to directly coordinate the execution of the activities as specified by the ASN's.

In chapter 6, an implementation scheme for the multi-activity executive introduced in the previous chapter is proposed. First, some general characteristics of the hardware architecture which will support the executive are given. Then the implementation itself is discussed and finally, a simulation of the executive is described. This simulation was actually implemented on a personal computer and the results it produces are presented in the last section of this chapter.

To conclude this introductory chapter, the major objectives of this thesis will be clearly stated:

- To show the need for system design tools for microprocessor-based systems which can be used by non-computer specialists.

- To introduce two sets of tools which can help the non-computer specialist specify and design microprocessor-based systems:

  - The first set of tools is based on finite state machines and is used for the specification and design of simple sequential event response systems.

  - The second set of tools is based on Process Activity Diagrams, a new graphical notation, and is used for the specification and design of concurrent systems.

- As part of each of these sets of tools, a simulation program is included which can:

- Show the behavior of the system, in order that the system designer may verify the reaction of the system under specific conditions; in other words, so that he can check the system specifications.

- Produce statistics on the simulations which can be used to measure the performance of the system, mainly in terms of response time. This enables the system designer to verify if real-time constraints can be met before the system is implemented.

CHAPTER 2

DESIGN ASPECTS OF MICROPROCESSOR-BASED SYSTEMS

The design of a microprocessor-based system, like any other system, requires taking a number of decisions after carefully considering all relevant design issues. This chapter will investigate some aspects of microprocessor-based systems which must be taken into account during the design process. The main purpose of this discussion is to provide an introduction to the terminology used in microprocessor-based system design and to place in proper perspective the main contribution of this thesis. To accomplish this, the chapter is divided into five sections. The first section introduces the hardware and software components of microprocessor-based systems. The second section classifies microprocessor-based systems according to their general application area and shows how system requirements vary depending on the type of system. In section three, a systematic methodology for designing the software of a simple system is introduced. Section four discusses the elements of real-time multitasking operating systems which are often required in complex systems. Finally, section five will put everything in perspective by describing the various design steps and design tools required to facilitate the design process.

2.1 INTRODUCTION TO MICROPROCESSOR-BASED SYSTEMS

In this section, the hardware and software component of microprocessor-based system will be very briefly presented with a particular emphasis on the options available to the system designer.

2.1.1 Hardware Component

A microprocessor-based system consists of a single-chip implementation of a central processing unit (CPU) called a microprocessor, some memory for program and data, and some input/output (I/O) ports to communicate with the outside world.

Depending on the application, the complexity of a microprocessor-based system can vary greatly. The simpler systems are built around a microcontroller or a single-chip microcomputer which are LSI or VLSI implementation of a complete processing system (i.e. CPU, ROM, RAM and I/O ports) on a single chip. These very inexpensive devices are mainly used for simple control type applications. The next level of complexity can be considered to be the 8 bit microprocessor-based systems which require external ROM, RAM and I/O chips in addition to the microprocessor. Then comes the more advanced microprocessors: the 16 bits, 32 bits and bit-sliced devices. More complex microprocessor-based systems are designed as multi-microprocessor systems.

An important problem for the system designer is to choose the proper hardware for his application in order to meet processing

speed, memory size and reliability requirements. When faster operation is required, the designer may choose to use a faster microprocessor, or a microprocessor with an instruction set better tailored to the application, or even design the system using a number of microprocessors. When a larger memory is needed, the designer may use a microprocessor with larger word size and/or larger address space, or use virtual memory (i.e. adding secondary memory with memory management). To increase the reliability of a system, a multi-microprocessor design may be the only solution, as redundant processors may be added to take over in case of failure.

Another important hardware design issue is whether to use off-the-shelf or custom designed units. This decision must be made at the chip, board and system levels. At the chip level, this would mean, for instance, choosing between using an available analog to digital (A/D) port, designing a port custom made to specification using a number of chips or doing a custom VLSI design. At the board level, the choice would be between designing a processing system using a number of chips (microprocessor, ROM, RAM and I/O chips) or using a single board computer available off-the-shelf. The decision depends on a number of factors including the production run size, the availability of off-the shelf components that can meet specifications, etc... When the production run is large, the custom design choice may be better as the design cost can be spread over a large number of units. For a one of a kind system, or a very small production run, then the objective is usually to

minimize the design cost. In such cases, this can be done by using as much as possible off-the-shelf components.

2.1.2 Software Component

The hardware component of a microprocessor-based system may be the only thing visible when looking at the finished product, but the software accounts for more than 80 per cent of the total system cost [JENS 79]. One of the main characteristics of software, from an economic engineering point of view, is that its initial design cost is very high, but it can be duplicated for virtually nothing. For this reason, it is important to introduce design methodologies which will decrease software design cost.

The software component of a microprocessor-based system can be partitioned into a number of different classes. The most basic classification divides the software into the operating system and application programs. The operating system usually consists of all the routines that interface directly with the hardware, and in that sense provides them as a service to the application programs which are responsible for carrying out the system's actions. In more complex systems, the operating system may also be responsible for coordinating a number of application programs which are running concurrently. This will be discussed in more detail in section 2.4.

The more complex the operating system, the harder it is for the application programmer to write efficient code as he must first learn all the features of the operating system and then use

them properly. It is thus important to select an operating system which is well tailored to the system being designed; i.e. one that contains all the required features and no unnecessary ones. As in the case of hardware however, there is always the choice between designing an operating system or to use one which is readily available. The choice depends again on the production run size and a number of other economic and engineering factors. An additional problem with most types of operating system is that they are very hard to design and test. This is why off-the-shelf operating systems are usually used even though they might not be the best choice for the particular application. Present day operating systems have too many features for most applications since they are designed to be adaptable to a large range of systems. This increases the overhead associated with the operating system in terms of response time and memory requirements.

2.2 TYPES OF MICROPROCESSOR-BASED SYSTEMS

Microprocessor-based systems can be classified under a number of different criteria. In this section however, only their application area will be of concern as this will enable us to obtain general system specifications. A first classification, often used, divides the microprocessor-based systems in general and special purpose systems.

General purpose systems provide one or more users with their own interactive programming environment designed to help write

and/or run programs for various applications: complex computations, word processing,... Microprocessor-based systems, however, find more applications in the area of special purpose systems as turnkey and embedded systems. These could be applications where the low cost of the microprocessor makes it possible to add digital processing capability to the system or highly complex digital systems designed as multi-microprocessor systems. The design of general purpose systems may use techniques developed for classical computer systems, however, special purpose systems have very different requirements and may thus benefit more from innovative design techniques. This is why the focus of this thesis is on the design of special purpose systems.

Another possible classification is according to response time requirements. Microprocessor-based systems can be broadly divided into two classes: man-oriented systems with large data handling requirements and man-machine interfaces, and systems that interact mainly with their environment under real time constraints. Of course there is no real solid line between these two categories, as some systems exhibit properties of both classes of system. Figure 2.1 shows some examples of microprocessor-based systems and their classification. The figure also illustrates their relative complexity by indicating their typical processing requirements in terms of microprocessor word size. At the lower end of the scale, the simpler systems could be designed using programmable switching circuits, while at the other end, the more complex systems may require mini or mainframe computers.

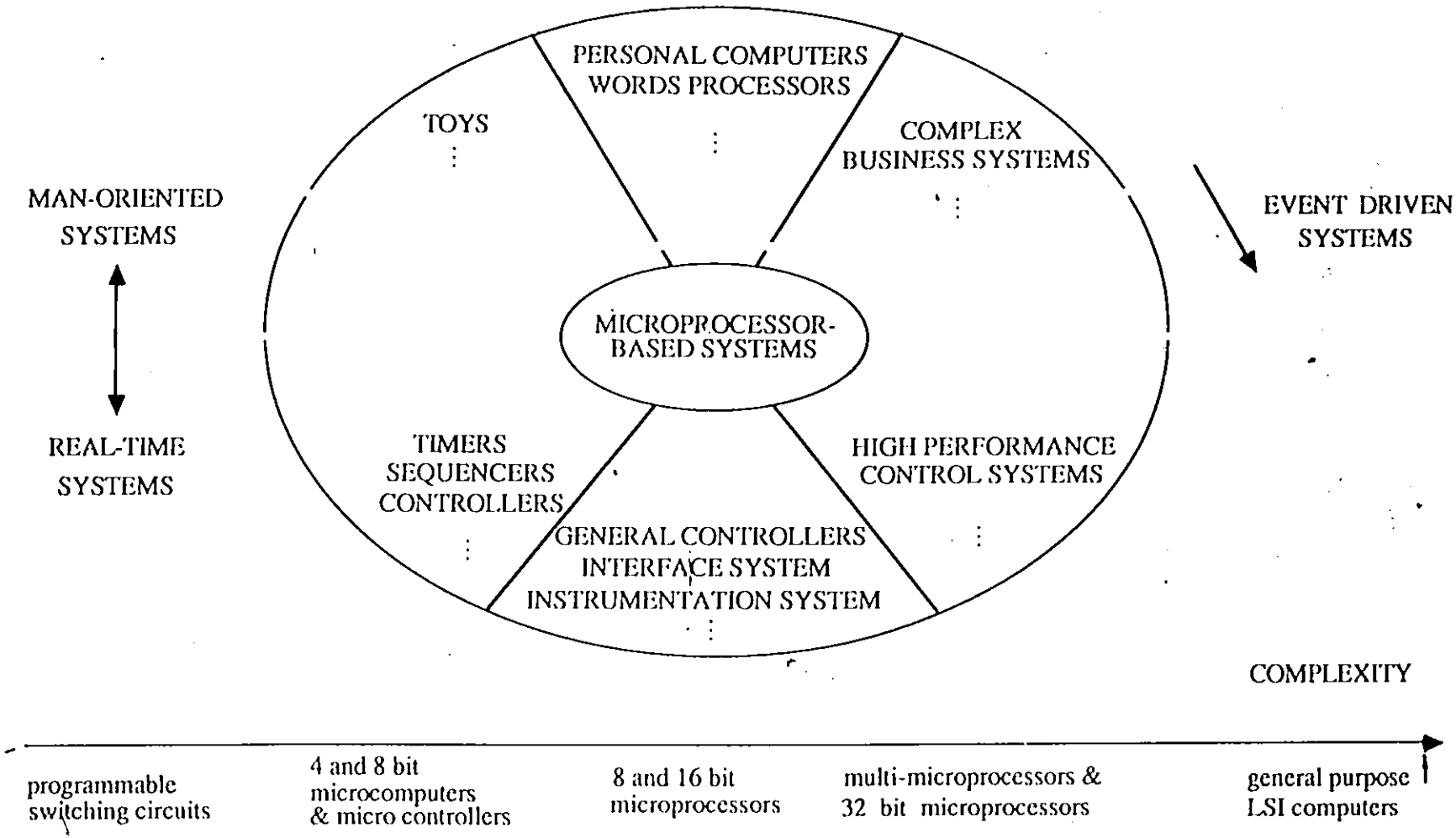


FIGURE 2-1 TYPES OF MICROPROCESSOR-BASED SYSTEMS

2.2.1 Man-Oriented Systems

Man-oriented systems have two main characteristics. The first one is that they have to respond to a user in human time scale, which is very slow compared to the speed of operation of the microprocessor-based system components. Their second characteristic is that they usually have large data handling requirements (otherwise the human user would be better off to use some other means...).

Because of these requirements, most man-oriented systems will need secondary storage devices and, consequently, many of them will also require memory management systems to handle information transfers between secondary and main memory. Since these systems interact with a human user, they will also require complex display systems to simplify this interaction. Thus complex peripheral controller chips must be part of these systems. In addition, arithmetic co-processors may also have to be included for complex computations, as well as other special purpose co-processors. To facilitate handling of more complex data structures, such as arrays and files, 16 or 32 bit high performance microprocessors might have to be used. Finally, it should be noted that these systems require user friendly interfaces and must be designed in such a way that they can be easily upgraded.

As can be seen from the above requirements, man-oriented systems could turn out to be quite costly. Most general purpose microprocessor-based systems would fall in this class of system,

including personal computers. However, there are also special purpose man-oriented systems, such as dedicated word processor machines, video games,... There are also a number of systems which are harder to classify as either general or special purpose man-oriented systems, such as engineering work stations, business and office systems, etc... These systems are initially semi general purpose but they can be dedicated to some special purpose, such as, an engineering workstation used to monitor and control a manufacturing process.

2.2.2 Real-Time Systems

The great majority of microprocessor-based systems, including the various industrial controllers and communication systems, have to interact directly with their environment. These systems require the ability to respond to events that occur synchronously and asynchronously in the environment, and to initiate various control actions in real time; hence the name real time systems.

The actual interaction with the environment is done through a number of sensors that can measure or detect the various system conditions, and through different activators that can carry out the systems response. Depending on the system, the required sensors and/or activators can be very simple (such as switches or relays), or quite complex subsystems that include microprocessors themselves.

In the design of real-time systems, two major points must be considered: SYSTEM RESPONSIVENESS and COMPLETENESS OF DEFINITION

[ALLW 80]. Responsiveness means that the system must respond to changes in the environment within a specified time. This might be because the system has to accept data within a given time interval as in the case of volatile data, or that it has to transmit a command within a predetermined time after receiving an input as in the case of most control applications, or a combination of the two. In general, the designer can distinguish between hard and soft real-time constraints. In the first case, the system's failure to respond within a certain time would cause a critical system error (unacceptable performance), while in the second case, as the response time increases, the performance of the system decreases gradually. This can be illustrated by a performance function which indicates the system performance in relation with the response time. Figure 2.2 shows such a function in qualitative term for hard and soft real time constraints.

There are other points that must be considered when discussing system responsiveness. The various possible concurrencies must be taken into account because more than one event may happen at the same time. Another characteristic of many real-time systems that may affect their responsiveness is the need for periodical activities which must be performed at regular rates.

The other aspect of real-time system that must be considered is completeness of definition. In systems that interact with the environment, all possible situations must be considered, that is, the response of the system under all combinations of events and system conditions must meet system specification. In practice, it

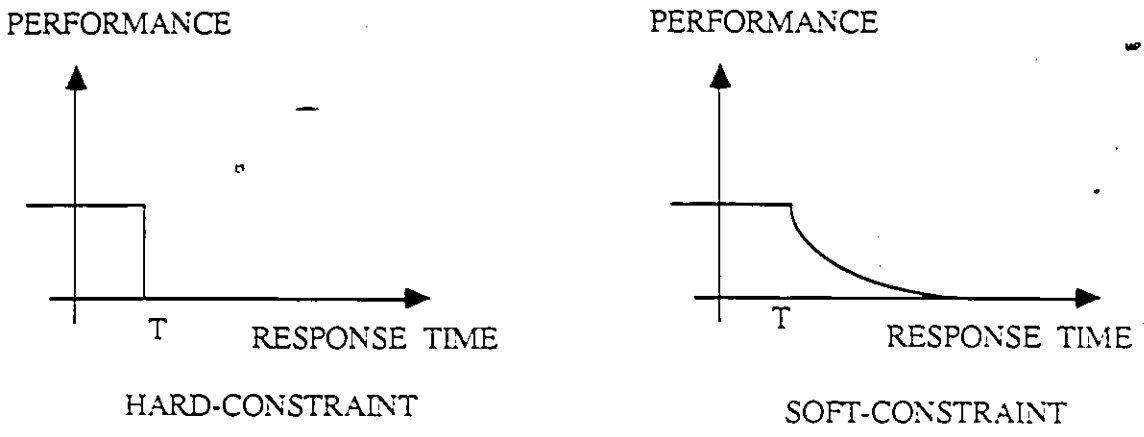


FIGURE 2-2 PERFORMANCE FUNCTION FOR HARD AND SOFT REAL-TIME CONSTRAINTS

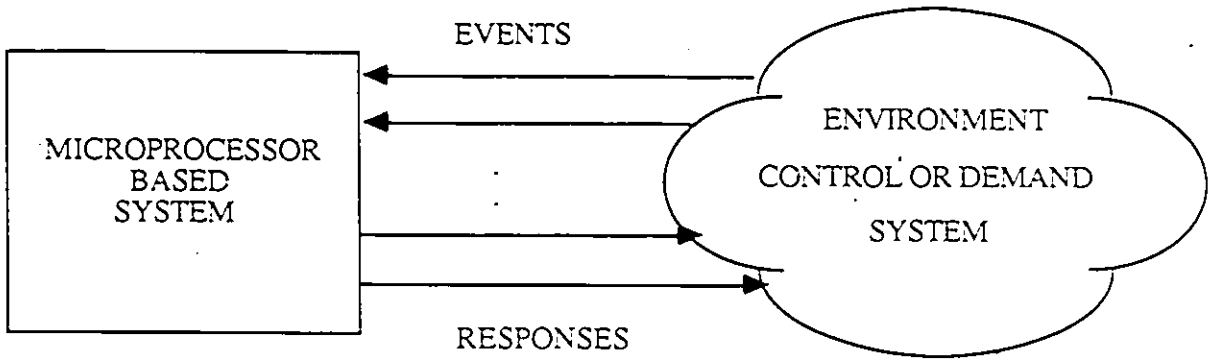


FIGURE 2-3 PROCESSING SYSTEM AND ITS ENVIRONMENT

is very hard to observe the response of the system under all conditions before implementation; this is why field trials (analysis of the final product in its environment) are held.

Not all microprocessor-based systems can be categorized as being purely man-oriented or purely real-time. There are some man-oriented systems that also have to meet some responsiveness requirements; but in such cases, response timing is usually more a question of convenience and is not critical in terms of system operation. Also many real-time systems require relatively complex man-machine interfaces to set up the system, and some may also need relatively large databases. As a rule, the design of these compound system can become quite exacting, requiring the use of complex real-time multitasking operating systems. An alternate implementation scheme is to design each of the system components separately and use a specially designed coordinator to integrate their operation.

2.2.3 Event Driven Systems

From the above discussion it can be seen that all special purpose systems are really event driven systems as they must execute some action in response to an event in the system's environment. These systems are thus driven by the events occurring in their environment. A schematic representation of such a system is shown in figure 2.3.

In this thesis, design methodologies for event driven systems will be developed, with a particular emphasis on real-time

systems because the measure of system performance that will be of most concern is response time.

Like all processing systems, event driven systems can be implemented in two levels. The first level involves carrying out the various responses or activities, and the second is concerned with their coordination and proper timing. This latter level can be done either by using a real-time multitasking operating system, or by designing a special purpose system executive which ensures that all response timing and concurrency requirements are met, as will be discussed in later sections.

2.3 DESIGN OF SIMPLE SEQUENTIAL EVENT RESPONSE SYSTEM

In this section, the design of simple event driven systems will be studied. The scope of this discussion will be limited to sequential event response systems, where the event responses are executed one at a time in some sequence.

The logical behavior of a sequential event response system can be described by the following four phases:

I System Initialization

II Event Detection

III Response Selection

IV Response Execution

The system initialization phase is highly specific to a particular system. Typically, it involves such activities as setting all variables to their initial state, initializing all

sensors and activators, doing self-diagnostic tests, etc.. The next phase is concerned with the detection of events. Once an event is detected, the proper response must be selected and executed. The execution of the response or activity, in practical terms, simply means jumping to a certain portion of the application code and executing it. This may require more or less processing time depending on the complexity of the action that has to be done. Note that the initialization is performed only once, and afterwards, the system must loop in phases II, III and IV.

The response selection and event detection phases will now be discussed in more detail in order to suggest a systematic design methodology which is system independent.

2.3.1 Response Selection

In the simplest case, a response can be associated to each event of the system in a look-up table format. Whenever an event is detected, the system executive simply checks in the table which activity it should execute. Note that it is possible to execute the same activity in response to different events. An example of such a simple event / response table is shown in table 2.1.

In some systems, it might be required that the selection of the response depends not only on the event that occurred, but also on some global system state [LAND 79]. To describe the event / response relationship of such a system, a Mealy type state

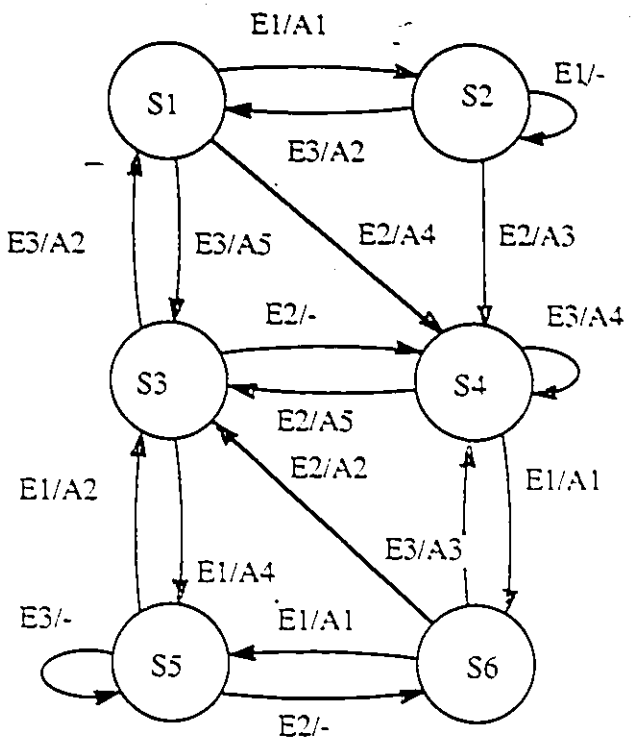
EVENT	RESPONSE
E1	Activity 2
E2	Activity 3
E3	Activity 1

TABLE 2.1 SIMPLE EVENT-RESPONSE TABLE.

diagram as shown in figure 2.4 can be used. The same information is contained in the table shown with the figure. By looking at the current state and the event that just occurred in the table, the system can find what will be the next state and the activity that has to be executed.

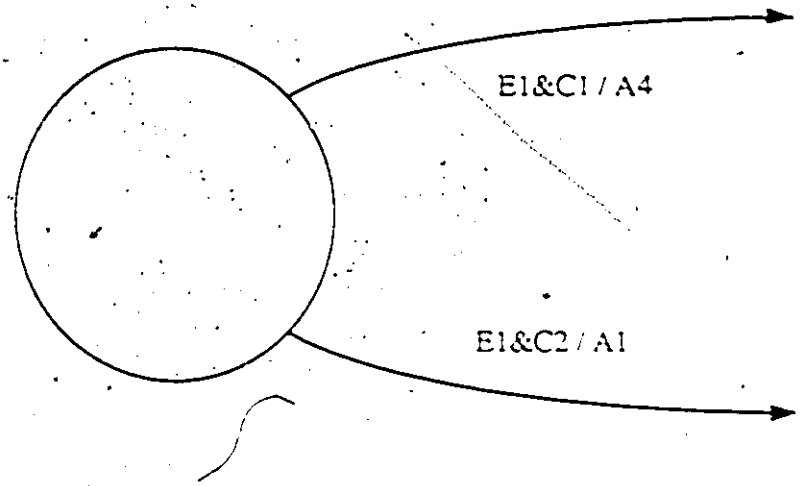
These systems are also called finite state machines (FSM's) because they are specified by a finite state diagram. A major problem associated with this type of system is that the number of system states increases very rapidly with the system complexity. This results in state diagrams which are very difficult to understand. To reduce the number of states, conditions could be added and associated with transitions, as shown in figure 2.5. In this case, the proper response and next state are selected by a combination of the event that occurred and which condition is present as expressed by some condition variable.

Another method of reducing the complexity of the state diagram is to have a number of modes of operation, with a separate state diagram for each mode. The system could be allowed to change modes from every state, or only from prespecified states. This method could be used in a system with a normal mode of operation and a survival or emergency mode. Whenever a major



S6	S5/A1	S3/A2	S4/A3
S5	S3/A2	S6/-	S5/-
S4	S6/A1	S3/A5	S4/A4
S3	S5/A4	S4/-	S1/A2
S2	S2/-	S4/A3	S1/A2
S1	S2/A1	S4/A4	S3/A5
	E1	E2	E3

FIGURE 2-4 STATE EVENT-RESPONSE DIAGRAM AND TABLE



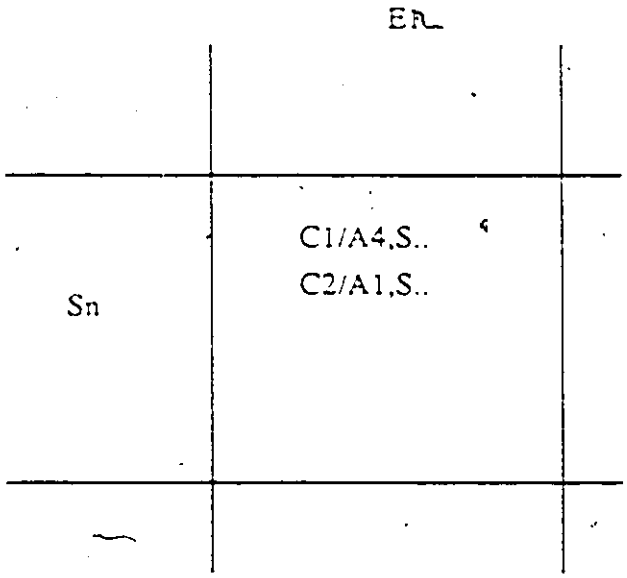


FIGURE 2-5 STATE TRANSITION WITH A CONDITION

component of the system fails, the system would enter the survival mode where the responses are different from the normal mode. Alternatively, the system may have a transient mode for power up or reset, and a steady state mode for normal operation.

2.3.2 Event Detection

Event detection involves two aspects: how to translate changes in the environment into events understandable by the microprocessor based system and when to check for the occurrence of these events. The mechanisms by which events are detected will first be looked at.

In general, event detection must be done using a combination of hardware and software mechanisms. A hardware scheme is necessary to translate the information in the environment into a digital form. In terms of software, event detection can be as simple as reading an input port and finding out if a particular bit is set. In a slightly more complex situation, the microprocessor might have to read in a value and compare it with some other value stored in memory, and only if there is a change will it acknowledge this as an event. This mechanism would be used, for instance, to control temperature. Alternatively, the comparison could be done in hardware, and the microprocessor would then only be disturbed when a change occurred. The detection of more complex events might require reading a number of input ports, doing a number of complex computations, table look-ups, comparisons, etc...

The next point to consider is when to detect events. Here, a distinction can be made between two methods: event checking and interrupt driven. In the first case, when the microprocessor is not executing activities in response to events, it will check for occurrence of events. In the second case, an event would generate an interrupt forcing the microprocessor to respond immediately to this event or to queue it.

The flowchart of an event checking is shown in figure 2.6. Note that the four phases discussed in the beginning of this section appear clearly in the flowchart. In event checking, it is important to consider the relative priority of the different events. Obviously, the occurrence of events can only be verified one at a time. Therefore, the order in which they are checked is important. One method is to start checking from the highest priority event to the lowest one, and as soon as an event is detected the proper response is executed. Afterwards, checking can resume either at the point it left off, or starting again from the highest priority event. If the first possibility is chosen, then some high priority events might not be serviced soon enough to meet real-time constraints, while in the second case, low priority events might never get a response. The choice of the algorithm thus depends on the particular real-time constraints of the system being designed.

Another possible algorithm for event checking would be to check all events during the detection phase, and to queue those that occurred by priority. Then the response for all the events in the queue would be carried out one by one before the detection

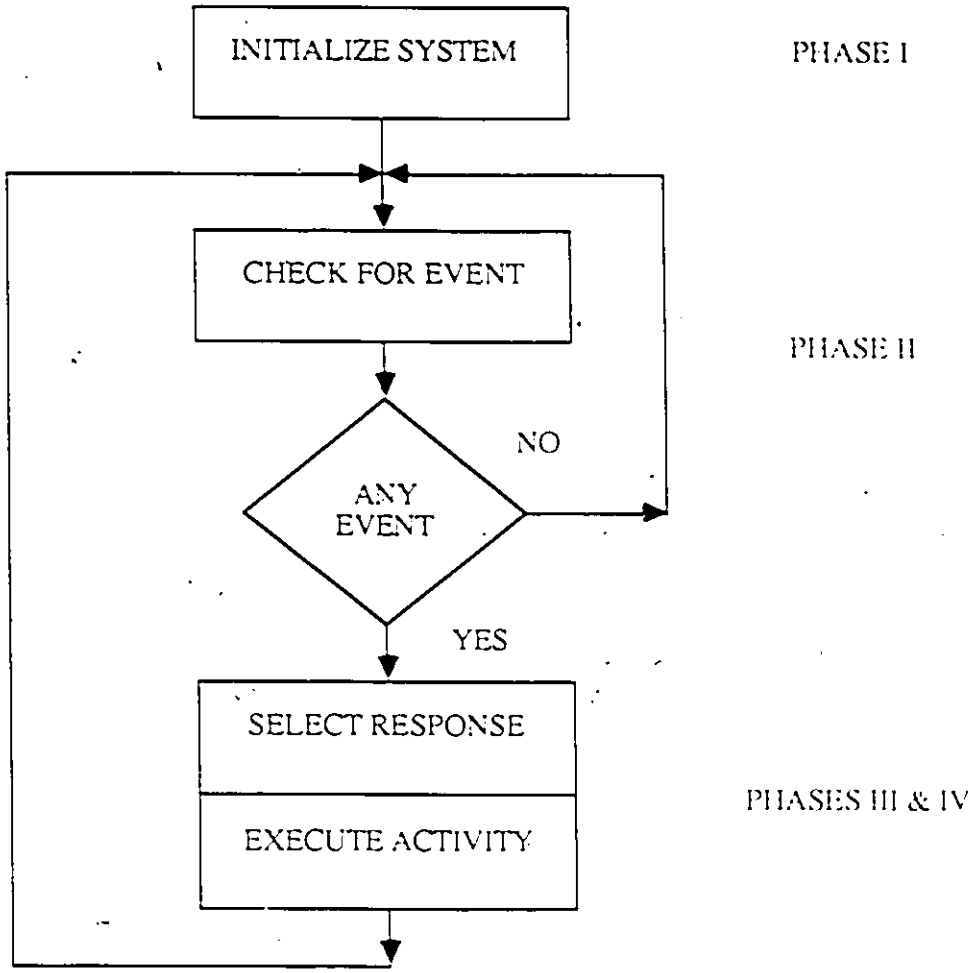


FIGURE 2-6 FLOWCHART OF AN EVENT CHECKING SYSTEM

phase would start again. In the cases where the response of the system depends on a global system state, it is important that the events are queued in the proper order, because each event may trigger a state change which in turn may affect the response of the next event in the queue. This algorithm has one major drawback: it results in lengthy time intervals between detection phase if many event responses have to be executed. This may render it undesirable for some real time applications.

Even with sophisticated event checking algorithms, it might not be possible to meet the real-time requirements of the system. This may be because the event detection phase requires too much processing time, slowing down the overall system. Another problem is that some events might be missed (when reading volatile data for example) if they occur during the response execution phase and are not remembered by the system. In these cases, interrupt driven systems must be used, where the microprocessor is interrupted whenever an event is detected. The interrupt routine would simply queue the events as they occur, where they will be processed by the main program. This scheme will probably require some algorithm for queue management to make sure that the events are serviced in the proper priority order which may not be the order in which they occurred. External clocks may also be required to interrupt the processor at regular intervals to perform periodical actions. To speed up response, certain high priority events could also be responded to directly in the interrupt routine. Note that this should only be done for event responses that are independent of the system state.

It is possible to have the advantages of event checking without the processing time overhead by using a two microprocessor system. One microprocessor would be interfaced to the system's input ports and would be responsible for event detection using an event checking scheme. It could also take care of queue management and response selection. The second microprocessor would only be responsible for carrying out the event response by taking the proper actions through the systems output ports.

In some systems, it is possible to use a combination of the two methods, where the high priority events would interrupt the microprocessor to ensure a speedy response and where event checking would be used for less important events.

The main concern when designing real-time systems using the methodology presented in this section is to find out if the system will respond to external events within an allowable time. It is desirable to verify this before implementing the system to cut down design cost. Two possible methods for checking system responsiveness will now be presented. The first one uses an analytical approach while the second one is based on computer simulation.

2.3.3 Analysis of Sequential Event Response Systems

The most often used analytical method is based on queuing theory [KLEI 75], in which a number of simplifying assumptions have to be made. These relate to the arrival pattern, queue

service discipline and service time pattern.

The event arrival pattern can be deterministic or random. Certain events, such as the clock event, are deterministic, but the majority of events in a real-time system are random. To use the queuing model, the arrival process must be further characterized. An assumption often used, since it simplifies the model, is that each occurrence of an event is statistically independent of any other occurrence of the same event. Furthermore, the event arrival rate is assumed to be constant. In this case, the event arrival pattern follows an exponential distribution. If A(t) is the probability distribution function of the arrival of one event, that is, the probability that the event will occur on or before time t, then:

$$A(t) = 1 - e^{-at} \quad \text{where a is the constant arrival rate.} \qquad (2.1)$$

The probability of having n events in a time interval t can also be computed, resulting in the Poisson Distribution:

$$P_n(t) = \frac{(at)^n e^{-at}}{n!} \qquad (2.2)$$

Note that so far, only the arrival of one event has been considered. The problem gets more complicated when a number of different events (with different arrival rates) may occur. If all the different events follow a Poisson distribution, and if they are all independent, then their arrival rate may be simply summed to obtain the arrival rate of "an" event without specifying which one.

There are a number of queue service disciplines. The most common ones are: FIFO (first in first out) queue, LIFO (last in first out) and priority queues. Queuing theory deals mainly with FIFO queues of finite or infinite size. This assumption simplifies the problem, as there is no worry about events being missed or never serviced because of higher priority events in the queue.

The last part of the model that must be defined is the service pattern which really means the execution time of the event responses (the activities). This execution time could either be fixed or random. Realistically, it can be expected that the execution time is both data and activity dependent. For example, if an algorithm contains some iterations, then the execution time depends on how fast it converges. Also, the system must execute a number of different responses which would normally take different times. However, the queuing model is much simpler if the activity time is fixed and the same for all activities, or if it is random with exponential distribution and constant service rate. The second assumption will be used because it is usually more realistic.

The simplest analytical model that can be applied to this system is the M/M/1/K queuing model which has a Markovian arrival and service rate, a single server and a finite queue length. This model can be analysed using a state-transition-rate diagram [KLEI 75]. The probability of the system being in state k where k is the total number of events in the system (the events in the queue and the one being processed) is then given by:

$$p_k = \frac{(1 - a/b)\,(a/b)^k}{1 - (a/b)^{K+1}} \quad \text{only if } a/b < 1 \qquad (2.3)$$

where a is the constant event arrival rate

and b is the constant service (activity execution) rate.

This probability is zero for k > K, since the system cannot remember more than K events.

In many cases, the assumptions that had to be made to use the M/M/1/K model renders it useless as it does not represent the actual system adequately. Another possibility is to use a G/G/1/K model where the arrival and service process are not Markovian. In this case however, the problem is more complicated as the arrival and service process must be well characterized and a solution must be computed using transforms (Laplace or Z transform) and complex-variables techniques. The assumption that the events are independent must remain even in a G/G/1/K system and this may not be true for the particular system studied.

2.3.4 Simulation of Sequential Event Response Systems

To approximate better the real-time system, a computer simulation model is used. A simulation program for sequential event response systems was written in Pascal and currently runs on a personal computer. The source code of this program is included in appendix A. It can simulate event checking systems with zero queue and interrupt driven systems with a finite queue.

In this program, the following assumptions are made:

- The simulation operates only in discrete time increments, the relative duration of which is determined by the user.

- The event arrival process is Markovian. This assumption could be changed and other arrival processes simulated including some with dependent events.

- Each event is assigned a probability of occurrence in relation with the time increment. For example, if the average arrival rate is one event per hour and the time increment is 1 second, then the probability of occurrence is 1/3600.

- Software checking to detect occurrence of events or the interrupt routine to queue the event takes an insignificant amount of processor time (zero) compared to the activity execution time. (This assumption could be easily modified).

- The activity to be executed in response to an event is determined by an FSM.

- The duration of a particular activity is fixed but different activities may have different execution times.

- In the case of event checking, after an event is detected and a response has been executed, checking may resume at the next event in the list (in a round robin fashion) or back at the highest priority event as selected by the user. In either case, if an event only causes a state change and no activity is to be executed, then checking resumes immediately where it left out

without missing the next event.

After running for a predetermined time, the simulation produces the following results:

- For each different event:

  - The number of occurrences of the event, from which the actual probability of occurrence is derived.

  - The number and percentage of events that are detected or queued (depending on the type of system simulated).

  - In the case of a system with queue, the maximum and average time the event spent in the queue and in the system.

- For each different activity:

  - The number of time the activity was executed.

  - The total time and percentage of time spent by the processor to execute the activity.

Examples of simulation results for the FSM described in figure 2.4 are shown in table 2.2 and 2.3. The first table shows the results of an event checking system (with event priorities), while the second shows the results of the interrupt driven system. The input data for the simulation is as follows:

Simulation duration: 100 000 time units
Probability of event 1: 0.006 / time unit
Probability of event 2: 0.008 / time unit
Probability of event 3: 0.010 / time unit

EVENT STATISTICS

EVENT NUMBER	OCCURRED	DETECTED	PROB OCCURRENCE	% DETECTED
1	610	445	0.00610	72.951
2	793	568	0.00793	71.627
3	1005	738	0.01005	73.433

ACTIVITY STATISTICS

ACTIVITY NUMBER	ACTIVITY DURATION	NUMBER OF TIMES	TOTAL TIME	% OF PROCESSOR TIME
0	0	358	0	0.000
1	16	247	3952	3.952
2	18	334	6012	6.012
3	20	127	2540	2.540
4	22	417	9174	9.174
5	24	268	6432	6.432
TOTAL			28110	28.110

TABLE 2.2 EVENT CHECKING SYSTEM SIMULATION RESULTS.

By comparing the results of these two simulations, the advantage of an interrupt driven system over an event checking scheme can be clearly seen in this particular example. In the event checking scheme only 72 percent of the events are detected and the processor is only executing activities (i.e. not polling) 28 percent of the time, while no events are missed in the interrupt driven system and the processor is doing useful work 38 percent of the time.

A comparison will now be made between the simulation results and the results obtained using queuing theory. In a system with event checking which has no queue, the probability of an event being detected should be equal to the probability of the

EVENT STATISTICS

EVENT NUMBER	OCCURRED	QUEUED	PROB OCCURRENCE	% QUEUED
1	587	587	0.00587	100.000
2	806	806	0.00806	100.000
3	1011	1011	0.01011	100.000

ACTIVITY STATISTICS

ACTIVITY NUMBER	ACTIVITY DURATION	NUMBER OF TIMES	TOTAL TIME	% OF PROCESSOR TIME
0	0	517	0	0.000
1	16	347	5552	5.552
2	18	439	7902	7.902
3	20	173	3460	3.460
4	22	559	12280	12.280
5	24	369	8856	8.856
TOTAL			38050	38.050

QUEUE STATISTICS

MAXIMUM QUEUE SIZE: 4 AVERAGE QUEUE SIZE: 0.140

EVENT RESPONSE STATISTICS

EVENT NUMBER	AVG TIME IN QUEUE	MAX TIME IN QUEUE	AVG TIME IN SYSTEM	MAX TIME IN SYSTEM
1	6.525	71	23.271	87
2	5.474	65	18.720	87
3	5.703	55	23.074	74

TABLE 2.3 INTERRUPT DRIVEN SYSTEM SIMULATION RESULTS.

processor being idle. This is obvious since events can only be detected when the processor is not executing an activity. This can be verified by noting that, from the statistics, the

processor is busy 28.1 % of the time, hence it is idle, or able to detect events 71.9 % of the time. This should be compared to the average detection ratio for all three events which is 72.7 %.

This result can also be verified with equation 2.3 which will give the probability of the system being idle if k is equal 0. To use this equation it must be assumed that the event service process is Markovian with a fixed service rate. This is not true in our system but this assumption will give an idea of theoretical results. It is assumed that activities 0 through 6 are equally likely to be executed (even though this is really dependent on the sequence of events as depicted by the FSM), and use 16.67, the average execution time of these activities, as the mean service time. Hence b, the service rate, is 1/16.67 or 0.06. The event arrival rate a will be the sum of the arrival rates of events 1, 2 and 3, as they are all independent and Markovian. Hence a is 0.024. Equation 2.3 can then be used since a/b is 0.4 which is less than 1. Note that since there is a single server in this system, K is equal to 1. Replacing all these values in equation 2.3 yields 71.4 % which is close to the results obtained in the simulation considering all the assumptions made.

In the case of the interrupt driven system, some of the results obtained in the simulation can also be computed using queuing theory. Since the queue does not overflow in this example, the model with infinite queue may be used to simplify the calculations. Using this model, the average number of events in the system is given by [KLEI 75]:

$$N = \frac{a/b}{1 - a/b} \qquad (2.4)$$

The average time spend in the system is found by applying Little's formula:

$$T = N / a \qquad (2.5)$$

Noting that the average time in the system is composed of the average time in the queue plus the average service time, the average time in the queue is thus given by:

$$T_q = T - 1/b \qquad (2.6)$$

Finally, using Little's result once more, the average length of the queue is:

$$N_q = a * T_q \qquad (2.7)$$

Using the same values for a and b as in the first example and the above equations yields:

$N = 0.667$ $T = 27.78$ $T_q = 11.11$ $N_q = 0.267$

These results are to be compared to those in table 2.2, where the average time in the system is 21.7 (average of the three events), the average time in the queue is 5.9 and the average queue length is 0.14. The average service rate from the simulation would be 21.7 (average time in the system) minus 5.9 (average time in the queue) which is 15.8. In our calculations, 16.57 was used for b. Even if these values were recomputed using 15.8 for b, the simulation results and the analytic results would still be quite different simply because the service rate is not

Markovian.

It should be noted that the simulation produces statistically variable results, while the analytical method will always produce the same result provided the same assumptions are made. However this does not mean that the simulation results are not accurate, since the analytical model may not be as precise a representation of the actual system as the simulation, which can be made more detailed and realistic [MACN 85] as was shown in the previous example.

By changing the event arrival rate and the time requirements of the activities in the simulations, some general conclusion about system performance may be drawn. In the following simulations, two different activity time requirements (table 2.4) and three different event arrival rates (table 2.5) will be used.

Activity	Activity Time 1	Activity Time 2
1	16	4
2	18	6
3	20	8
4	22	40
5	24	42

TABLE 2.4 ACTIVITY TIME REQUIREMENTS.

Event	Arrival Rate 1	Arrival Rate 2	Arrival Rate 3
1	0.006	0.012	0.02
2	0.008	0.016	0.02
3	0.01	0.02	0.02

TABLE 2.5 EVENT ARRIVAL RATES.

Note that the average activity time is the same in both cases, hence the service rate b did not change. What was changed in activity time 2 is that certain activities now require much more time than others. Arrival rate 1 corresponds to the previous example, thus a/b is approximately 0.4. Arrival rate 2 is double arrival rate 1, hence a/b is approximately 0.8. In the third case all events have the same arrival rate of 0.02 per time unit giving a total average arrival rate of 0.06 which is equal to the average service rate, hence a/b is approximately 1. The relevant results of these simulations are shown in tables 2.6, 2.7 and 2.8.

Looking at the results of the event checking system, it can be seen that the different activity time requirements (1 and 2) do not affect the processor utilization and event detection rate. This is to be expected since the average activity execution time is the same in both cases. However, as expected, the processor utilization increases and the event detection rate decreases as the event arrival rate increases. When a/b approaches 1 (table 2.8) the processor utilization and the event detection rate both approach 50 percent as predicted by queuing theory (equation 2.3).

In the interrupt driven system however, there are a number of differences that appear between the two activity time scenarios. When some activities take much more time than others (scenario 2), it can be seen that the maximum and average size of the queue increases, as well as the maximum and average time spent in the system. This is to be expected since the queue builds up when one

	Activity Time 1	Activity Time 2
EVENT CHECKING		
Processor Utilization:	28 %	30 %
Detection Ratio:	73 %	72 %
INTERRUPT DRIVEN		
Processor Utilization:	38 %	42 %
Maximum Queue Size:	4	8
Average Queue Size:	0.14	0.34
Maximum Time in System:	83	158
Average Time in System:	22	32

TABLE 2.6 SIMULATION RESULTS FOR ARRIVAL RATE 1.

	Activity Time 1	Activity Time 2
EVENT CHECKING		
Processor Utilization:	44 %	46 %
Detection Ratio:	58 %	55 %
INTERRUPT DRIVEN		
Processor Utilization:	76 %	83 %
Maximum Queue Size:	19	33
Average Queue Size:	1.5	4.3
Maximum Time in System:	325	698
Average Time in System:	47	108

TABLE 2.7 SIMULATION RESULTS FOR ARRIVAL RATE 2.

	Activity Time 1	Activity Time 2
EVENT CHECKING		
Processor Utilization:	49 %	49 %
Detection Ratio:	53 %	53 %
INTERRUPT DRIVEN		
Processor Utilization:	90 %	91 %
Maximum Queue Size:	30	46
Average Queue Size:	5.1	10
Maximum Time in System:	520	880
Average Time in System:	100	200

TABLE 2.8 SIMULATION RESULTS FOR ARRIVAL RATE 3.

of these long activities is executed. As the queue increases, so those the time spent in the queue and hence the time spent in the system. These increases in response time may be detrimental to system performance as certain real time constraints can no longer be met. To reduce the average size of the queue and the response time, time consuming activities must be divided into smaller activities which must be executed in some sequence to provide the complete event response. These type of systems will be the subject of the next section.

The simulation was also run for a/b greater than 1. In the case of an event checking system, the processor utilization exceeds 50 percent while the detection rate drops below 50 percent. In the interrupt driven system, the queue increases continuously as the system cannot respond fast enough to the events. Hence all the results are irrelevant as no steady state is reached. Since the event arrival rate is not a parameter that the designer can play with, the only solution is to increase the service rate so that a/b is less than one. This can be done by decreasing the activity execution time (using a faster processor) or by using a multiple server system, i.e. a multiple processor system.

2.4 ELEMENTS OF REAL-TIME MULTITASKING OPERATING SYSTEMS

As seen in the previous section, in certain systems the event response may be better represented by a number of simple

activities executed in some sequence instead of a single large activity. By interleaving the execution of the various activities associated with different events, responses to more than one event can be carried out at a time, while still using a uniprocessor system. This is the main difference between sequential event response system and multitasking systems.

2.4.1 Concept of Task and Concurrency

When using a multitasking operating system, the different activities in the system are grouped into tasks. Thus a complete event response involves one or more tasks. A task has its own thread of control as it can be run concurrently with other tasks in the system.

This concurrency between tasks can be real or apparent. Real concurrency means that tasks are being executed simultaneously, this implies a multiple processor system. Apparent concurrency, however, can be achieved on a single processor using a time sharing scheme. In this case a number of tasks are being executed in the same time interval in an interleaved fashion. It is the job of the operating system to manage these concurrent tasks. In the case of real concurrency, a multiprocessing operating system is required, while in the case of apparent concurrency, a (real-time) multitasking operating system must be used. In very complex systems, it is possible to mix the two modes of operation using a multiple processor system where each processor is running a multitasking multiprocessing operating system.

It should be clear that tasks, especially tasks involved in the same event response, need to communicate between themselves to exchange information. Figure 2.7 shows a generic state diagram representation of a task from an operating system point of view. The exchange of information between tasks is taken care of by the operating system and is done while in the wait state.

The task is initially dormant (or non-existent) until it is awakened (or created) and becomes ready to run. The operating system may then schedule the task to run (the executing state). From this state it may choose to wait for some information and enter the wait state, or it may be suspended by the operating system in order than a higher priority task may be scheduled. A wait is a voluntary action by the task while a suspend is not under the control of the task. If the task receives the information it was waiting for, it will be made ready to run by the operating system. When the condition that justified the suspension of a task is no longer true, the operating system will let the task become ready to run once more. From any state, the task may be aborted (or killed) and put back in the dormant state. Note that this is only a generic state diagram; more states and/or transitions may be present depending on the particular multitasking operating system used.

2.4.2 Problems in Concurrent Systems

When a number of tasks are running concurrently in a system and need to share resources as well as communicate, a number of problems may occur. The two main problems associated with

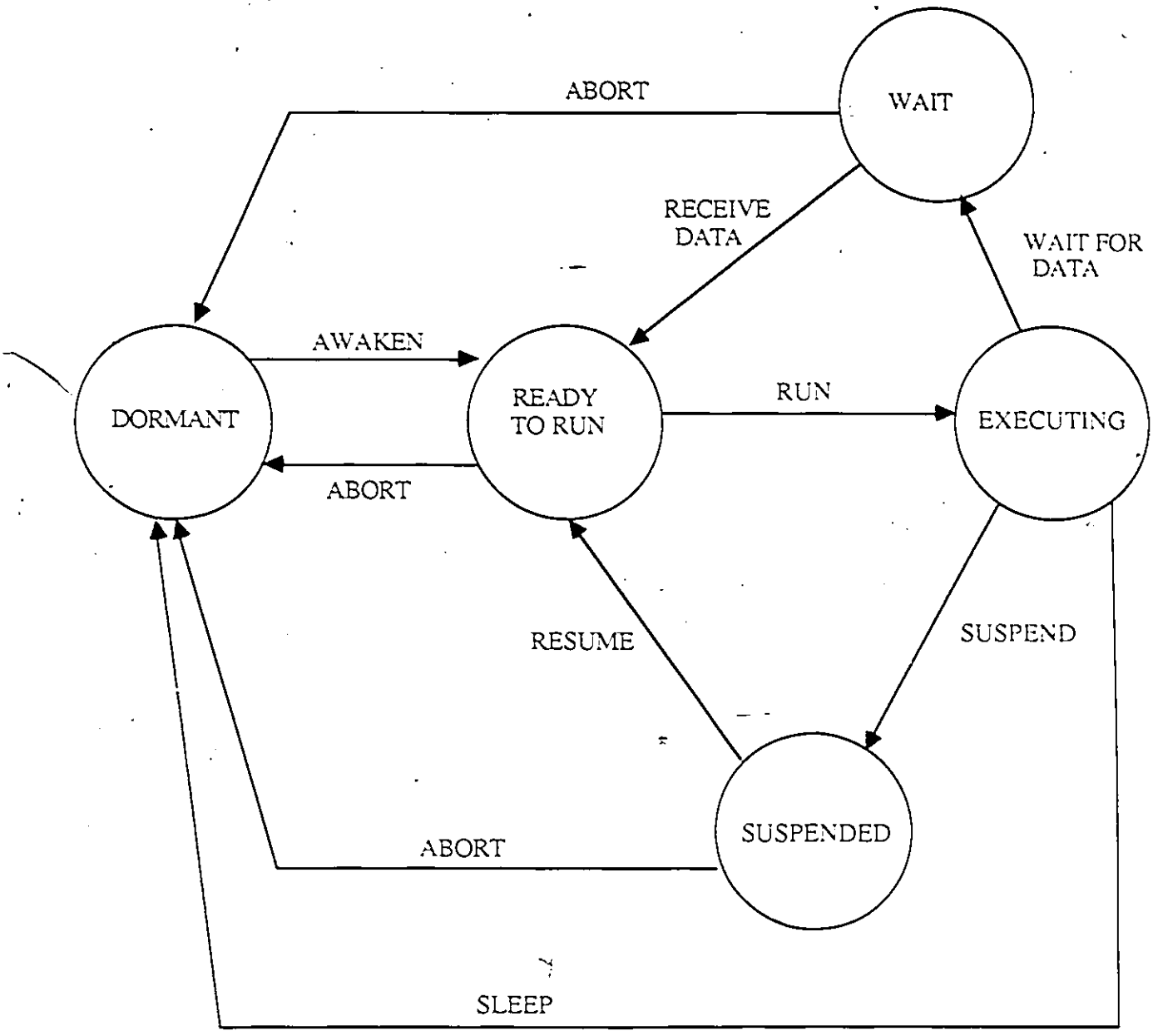


FIGURE 2-7 GENERIC TASK STATE DIAGRAM

concurrent execution are mutual exclusion and deadlock.

The mutual exclusion problem arises when two tasks require access to some common resource. For example, if two tasks need to update a common variable, then it must be made sure that only one task has access to the variable at any one time otherwise the variable may be incorrectly updated. The problem is even clearer when considering a physical resource like a printer. Obviously, two tasks should not output to a printer at the same time. The portion of the software code that access this common resource is called a critical region which must be protected so that other tasks may not access the common resource when the critical region is being executed.

Deadlock occurs when two or more tasks are blocked on some condition which will never become true because the task(s) that could change this condition are also blocked on a similar condition. A typical example of deadlock occurs in resource allocation when two tasks (A and B) both require two resources (1 and 2). If task A request resource 1 first and then 2, and if task B requests 2 first, then there is the possibility of deadlock. This will happen if task A is granted resource 1 and task B is granted resource 2; in this case, task A will wait forever for resource 2 which task B will never release because it is waiting for resource 1. Deadlock can be avoided by careful programming. In this resource allocation problem, one solution is to require all tasks to request all their resources in the same order.

2.4.3 Inter-Task Communication

To avoid the problem of mutual exclusion in inter-task communication, a number of mechanisms have been developed. They can be broadly divided into procedure based and message based mechanisms [CASH 80]. The procedure based mechanisms, which include semaphores, monitors, distributed process, communicating sequential processes and the Ada rendez-vous [WEGN 83] will be considered first.

Semaphores [BENA 82] where introduced by Dijkstra as a mechanism for synchronization between concurrent tasks. A semaphore is a simple data item which can only take non-negative value. It can be manipulated only by the following procedure:

- initialize semaphore
- wait on semaphore
- signal semaphore

Initialize semaphore sets the semaphore to its initial value. The wait operation checks if the semaphore is zero. If it is non-zero then it is decremented by one and the task may continue; if the semaphore is zero, then the task is suspended until it becomes non-zero. The signal operation increments the semaphore count by one, which may unblock a task waiting on this semaphore. The initial count of the semaphore thus determines the number of tasks which may use the resource protected by the semaphore. In most applications, only one task may use the resource, hence the initial value of the semaphore is set to one and this is referred to as a binary or Boolean semaphore. To protect the critical

region, a wait on a semaphore is done just before the critical region and a signal operation is performed right after. For this scheme to work, the test and set operations on the semaphore must be indivisible operations.

Semaphores can provide mutual exclusion, but only if used correctly. The portion of the code which accesses the shared resource, the critical region, is part of the application task. Therefore, it is the responsibility of the application task programmer to use the pair of semaphore calls properly. To guard against programming errors of this type, a higher level of synchronization primitive, such as monitors [HOAR 74], is needed.

Basically a monitor is a procedure that contains all the critical regions needed to access a shared resource encapsulated between semaphores. When a task requires access to write to a shared variable, then it will call the appropriate monitor with the proper parameters. Only one task may be executing inside a monitor at any one time because this portion of the code is protected by a semaphore or a similar construct. Since only one task can be in the monitor at any one time, mutual exclusion is guaranteed. This is an improvement over the semaphore because the shared resource is only declared local to the monitor forcing the application programmer to call the proper monitor to manipulate the shared resource.

The other procedural mechanisms mentioned, distributed processes [HANS 78], communicating sequential processes [HOAR 78] and the Ada rendez-vous [SAIB 85], define higher level

communication mechanisms which have similarities with the message passing schemes. Of these three, Ada has enjoyed the most widespread use because it was designed as a military standard for the U.S. department of defense.

The other group of inter task communication mechanisms includes all message passing schemes. These schemes are based on two primitives send and wait, and sometimes on a third one, the reply primitive.

In order for two tasks to communicate together, one task must use the send primitive while the other should call a wait for message primitive. There are a number of variations in the scheme at this level. For example, the send primitive could be blocking or non-blocking. In the first case, the sender waits for a reply which is sent using the reply primitive before going on with his business; this is a form of rendez-vous. Note that here, the difference between send and reply is that send is blocking while reply isn't. With the non-blocking send, the sender simply carries on with his affairs after sending his message, just like sending a letter in the postal system. The blocking send is often preferred because it assures the sender that the message is received. However, the non-blocking send favors concurrency by allowing the sender to continue; this may be desirable in a multiple processor system.

Other variations are possible with the receive primitive. For instance, the receive may be selective. The selection may be based on which task sent the message or on some other message

identification. This feature enables tasks to respond to higher priority messages first which is important in real-time systems.

The above discussion only focused on the aspects of the message passing scheme that were of concern to the application programmer. The actual implementation of these mechanism can be done using message queues, shared buffers, mailboxes, etc...

2.4.4 System Design in a Multitasking Environment

All the problems associated with concurrent execution and all the various inter-task communication mechanisms must be carefully considered when designing a special purpose system based on a multitasking operating system. The grouping of the various activities that have to be executed in response to events into tasks is not a simple process. This is why it is important, in such a complex environment, to proceed with the design in a structured fashion as will be discussed in the next section.

2.5 NEED FOR DESIGN TOOLS

The development of a microprocessor-based product from an idea to its realization involves a number of steps which can be partitioned as follow:

- Obtain system requirements.

- From the requirements derive the functional specifications.

- Partition the functions into hardware and software.

- Design the hardware and software components.

- Implement and test a prototype system.

- Test the system in the field, i.e. under normal operating conditions.

At any point it is possible to go back to a previous phase to correct design or specification error. Afterwards there is the product support and maintenance phase which consists of upgrading the system to meet changing system requirements and solve some problems which were undetected so far.

The requirements of the system are often found as a result of a marketing study, in the case of a commercial product, or they can be dictated by a specialized user who needs a one of a kind type system. The work of the system designer thus really starts at the specification level where the requirements are mapped into a form understandable by designers. Design tools become useful at this stage to describe system specifications.

These specifications must then be mapped onto hardware and software components. This involves decision of the type: how many processors to use, what hardware architecture, which operating system, etc... At this level, tools would be needed to manipulate the specification in order to find the best mapping possible. This would involve such activities as system simulation to find out if system specification can be met under that particular hardware/software mapping.

There are a number of computer aided design (CAD) tools available on the market today to simplify hardware design. These tools enable the hardware designer to enter his circuit graphically on the computer and test it out before implementation using various circuit simulation software packages. This thesis, however, will focus on tools at the system design level, rather than at the circuit design level. Parts of these tools can also be extended and used at the software design level. This was shown in section 2.3 where finite state machines were used to specify the system and to provide a framework for the design of an executive for sequential event response systems.

These design tools will be based on graphics as an image can convey much more information than text given the same space on a sheet of paper or a video display terminal. A number of graphical programming tools have been or are in the process of being developed [RAED 85], but not all of them can be used for the design of concurrent systems. In the next chapter, some of the design tools currently available for the specification and design of concurrent systems will be presented.

2.6 CONCLUDING REMARKS

In this chapter, a methodology for the design of simple event response systems was discussed. It was shown that the implementation and design scheme to be used depends on the factor a/b; the event arrival rate over the service rate. For a small a/b, an event checking system can be used provided that it is

permissible to miss a few events. If events cannot be missed, then an interrupt driven system must be used. When a/b is large, i.e. when the event arrival rate is close to or greater than the service rate, the service rate must be increased by using a faster processor or a multiple processor system.

In cases where certain activities require much more execution time than others, partitioning the system into smaller activities may improve real-time system performance by executing the activities associated to different event response in an interleaved fashion. This method, however, would require different tools than those discussed in this section. These tools will be the subject of the following chapters.

CHAPTER 3

CURRENT GRAPHICAL DESIGN TOOLS FOR CONCURRENT SYSTEMS

Graphical representations of the hardware and software components of digital processing systems have always been important design tools. Block diagrams are used to represent the hardware architecture while flowcharts where introduced to help assembly language programmers. The purpose of these tools is to design the hardware and software of these systems at a higher level of abstraction before going down to the integrated circuit or assembly language levels. Flowcharts however, cannot express the concept of concurrency as they were designed to indicate only a single flow of control. Even with the addition of fork and join nodes which can express parallel paths, flowcharts are not powerful enough to design concurrent systems.

This chapter will describe and discuss some of the current graphical design tools used to specify and/or design concurrent systems. The first section will present Petri nets and data flow graphs, the most accepted graphical models for asynchronous concurrent systems. Section two surveys tools used to describe the software structure of real-time systems. These tools will illustrate the communication requirements and interrelations between the various tasks in a multitasking system. Finally, section three will discuss graphical notations which specify the internals of the tasks as well as their interrelationships.

3.1 SYSTEM LEVEL TOOLS

A system can be viewed at different levels of abstraction. Most current graphical design tools describe systems at relatively low levels. Tools like Petri nets and data flow charts were originally designed for such low level description. However, they can also be used for higher level specification and design because they can describe the required behavior of the system by showing the flow of information throughout the various functions that the system must provide without specifying how these functions are to be implemented. This can be done because they do not deal explicitly with lower level concepts such as communication mechanisms, tasks and other operating system features.

3.1.1 The Petri Net Model

Petri nets [PETE 77] [BRAM 83] were developed from the original work of Carl Adam Petri who introduced, in his thesis in 1965, a model for control and information flow in asynchronous concurrent systems. Petri nets can be used for software [AGER 79] and hardware [HOZJ 79] specification, as well as a tool to verify correctness of protocols [BERT 82].

A Petri Net consists of a set of nodes connected by directed arcs. There are two types of nodes: places which are represented by circles and transitions represented by bars. Figure 3.1 shows a graph representation of a Petri net. As can be seen in this figure, the arcs are directed from places to transitions or from

transitions to places. Places from which arcs are incident on a transition are called the input places of the transition, while places on which arcs from the transition terminate are called output places. In figure 3.1 for example, place p1 is an input place to transition t1, while places p2 and p3 are output places of transition t1. Formally, a Petri net can be defined by the fourtuple:

C = (P, T, I, O)

where P is a set of places;

T is a set of transitions;

I and O are input and output functions which define the relationship between the places and the transitions.

Using this formalism, the Petri net of figure 3.1 would be defined as:

C = (P, T, I, O)

P = (p1, p2, p3, p4, p5, p6)

T = (t1, t2, t3, t4)

I(t1) = (p1)	O(t1) = (p2, p3)
I(t2) = (p2)	O(t2) = (p4)
I(t3) = (p3)	O(t3) = (p5)
I(t4) = (p4, p5)	O(t4) = (p6)

A Petri net is a dynamic representation of a system. Graphically this is indicated by the position and movement of tokens in the places of the net. Tokens are graphically represented by dots in the places. The distribution of the tokens in one point in time is called the marking of the Petri net and

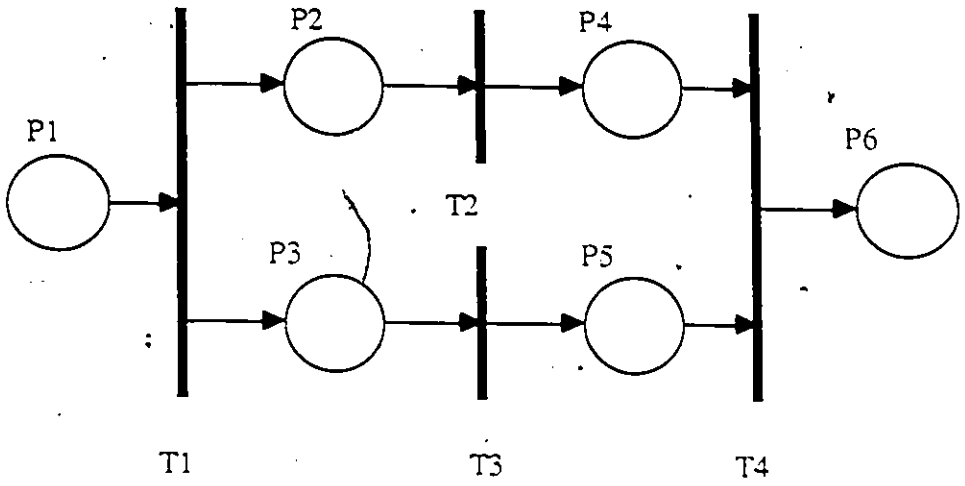


FIGURE 3-1 A GRAPH REPRESENTATION OF A SIMPLE PETRI NET

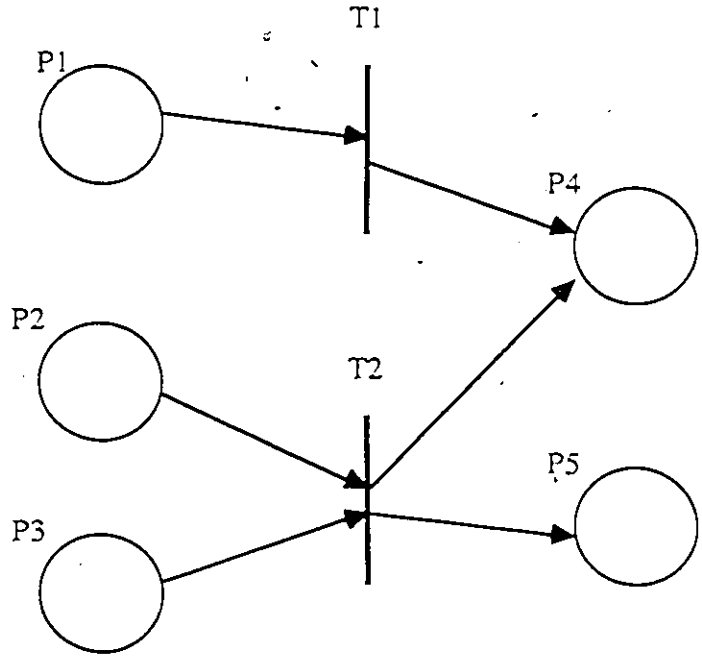


FIGURE 3-2 CONCURRENCY IN PETRI NETS

it really shows the state of the net at that particular time. The movements of the tokens are governed by the following rules:

- A transition is enabled when there is at least one token in each of its input places.

- Any enabled transition may be fired.

- When a transition fires, one token is taken out of each input place of the transition, and one token is put in each output place.

The presence of a token in a place indicates that a certain condition is satisfied, and a transition is only enabled when all its associated conditions are satisfied. Tokens can thus represent availability of data, the data itself, or some other condition. The concept of concurrency in Petri nets is illustrated implicitly by the fact that more than one transition may be enabled in the same net. Such an example is shown in figure 3.2, where both transitions, t1 and t2, could fire at any time.

Note however that transitions may not fire simultaneously in a net since this may cause a conflict situation. Figure 3.3 shows such an example where both transitions t1 and t2 are enabled. Note that the firing of either one of these transition disables the other as there is only one token in place p2. By defining events to occur non-simultaneously, we prevent the possibility of both transitions firing. Conflict situations are required to model such constructs as semaphores where only one process may

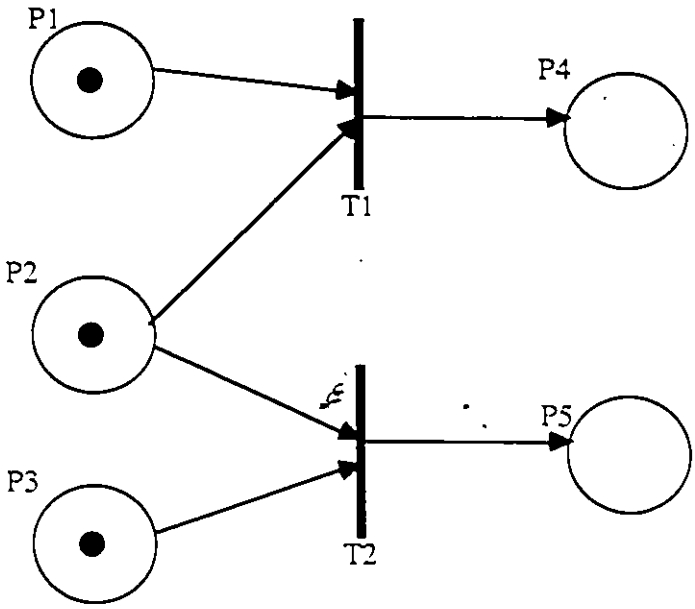


FIGURE 3-3 CONFLICT SITUATION IN PETRI NETS

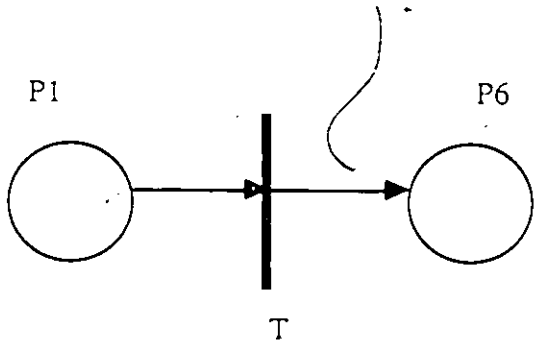


FIGURE 3-4 A SIMPLIFIED PETRI NET

gain access to some resource at any given time.

Petri nets are a nested construct. It is possible to expand a transition into a net consisting of a number of places and transitions. In the same manner, it is possible to simplify a portion of a Petri net between two places and replace it by a single transition. Figure 3.4 shows a much simpler view of the Petri net of figure 3.1, where the portion of the net between transitions t1 and t4 has been replaced with a single transition T. This feature is useful to look at Petri net models of a system at different level of abstraction.

3.1.2 Properties of Petri Nets

In order to analyse the behavior of systems modelled with Petri nets, a number of basic properties related to boundness, reachability and liveliness must be defined. A Petri net is said to be k-bounded if there can be no more than k token(s) in any given place at the same time. If a net is not bounded, then tokens may accumulate infinitely in a given place. In certain systems, the presence or absence of a token indicates if a condition is satisfied or not, hence it is superfluous to have more than one token per place in these systems. For this reason, a Petri net that is 1-bounded is said to be safe.

A marking "N" is said to be reachable from an initial marking "I" if there exists a sequence of transition firing from "I" which will result in the marking "N". It is often required to find the reachability set of a net, that is, the set of all

markings which are reachable from an initial marking. Note that if the net is k-bounded, then the net has a finite number of possible states since each place can only contain between 0 and k tokens. Thus the reachability set can be found using a similar reachability tree as the ones used in finite state machine analysis.

Another important property of Petri nets relates to the firing of transitions. A transition is potentially firable if there is a sequence of transition firing that can enable it, otherwise it is dead. If a transition is potentially firable in all reachable markings then it is live. The liveliness property is useful when modelling operating systems with Petri nets as a dead transition may indicate a possible deadlock situation.

3.1.3 Extensions of Petri Nets

Petri nets were originally developed to provide more modelling power than finite state machines (FSM's). Like FSM's they are used to model many different types of systems. Because of this generality, some extensions have been added to Petri net to model particular types of systems, in the same way that extensions have been added to FSM's. One of the first extensions was to allow multiple arcs between places and transitions. Figure 3.5 shows an example of such a net which is referred to as generalized Petri nets in the literature. In this example, transition t1 will only be enabled if there are three tokens in p1 and two tokens in p2. After t1 fires, two tokens will be

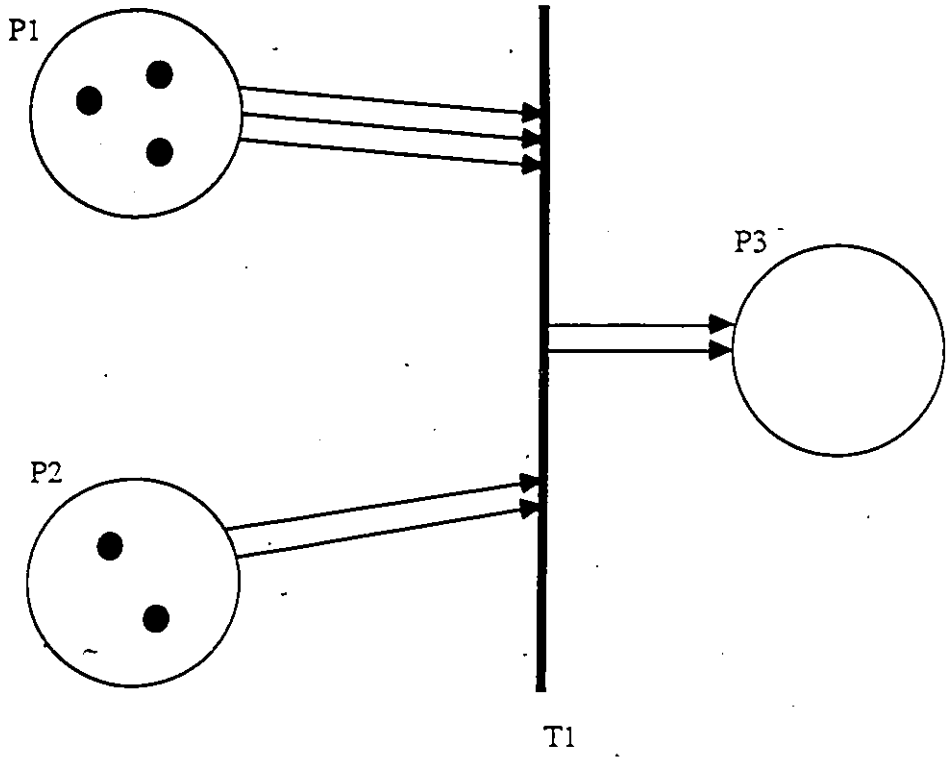


FIGURE 3-5 GENERALIZED PETRI NET

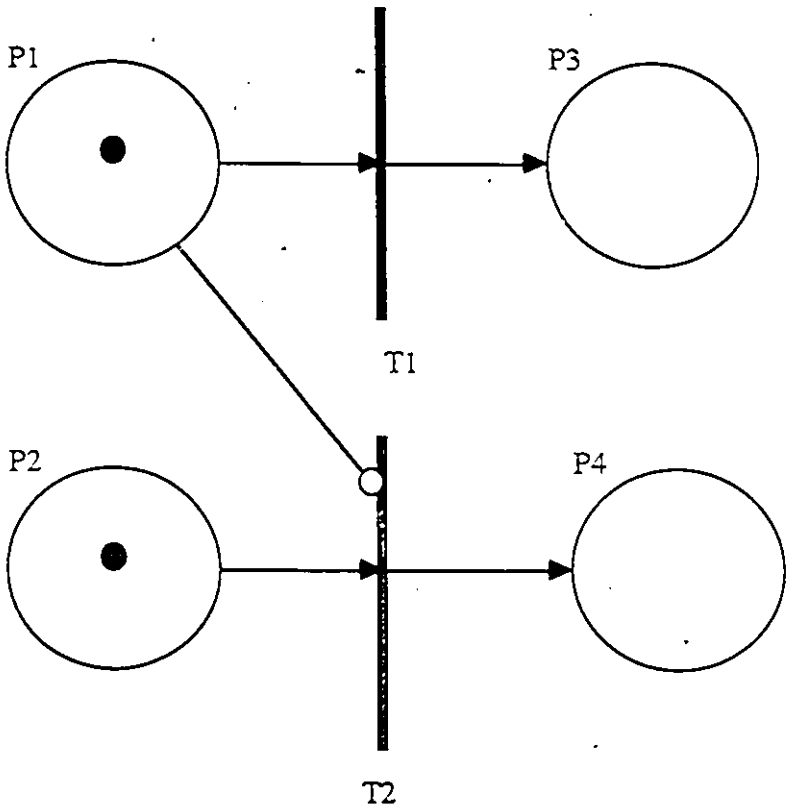


FIGURE 3-6 EXTENDED PETRI NET WITH AN INHIBITOR ARC

placed in p3. If tokens represent events in the system environment, then this extension can be used to count events before taking some action. For example, a transition representing the activation of an alarm will only fire after three unsuccessful attempts have been made to open a combination safe.

To introduce priorities in the firing of transitions, inhibitor arcs can be added to Petri nets. These arcs can only be directed from a place to a transition. Their purpose is to disable the transition if there is one or more token in the place it originates from. Since the transition may only fire when there is no token in the place, no tokens are taken out from this place when the transition fires. Figure 3.6 shows an example of a Petri net with an inhibitor arc. Note that the inhibitor arc terminates with a small circle instead of an arrow. In this example, transition t1 has priority over t2 since if both transition could be enabled by the presence of a token in p1 and p2, the inhibitor arc would prevent t2 from being enabled because of the token in p1.

There are a number of other possible extensions that add decision capabilities to Petri net. For instance, the tokens can be marked by coloring them, or they can carry along a value. The different colored tokens indicate different types of conditions which can be present in the same place while the token with values can be used as counters. The colors and the values are altered at the transitions, which are responsible for putting the tokens in its output places with the proper color or value. Conditions could also be added to the transitions based on the

color or the value of the token in its input place(s). An example of such a marked Petri net with conditions is represented in figure 3.7. In this example, one token is marked with the variable name "a". Every time the transition t1 fires, "a" is incremented by one modulo n. Conditions have been added to transitions t2 and t3 in such a way that if "a" is greater than the value of "b" then t2 will fire otherwise t3 will fire.

When using Petri nets for performance modelling, it is useful to add delayed transitions which can only fire after some fixed or variable delay. They can be used to represent a process which takes some time to execute. Graphically, delayed transitions are illustrated by a narrow rectangle instead of a bar.

A number of major extensions and restrictions have been added to Petri nets to form Evaluation nets (E-nets) and Macro E-nets [NOE 73]. One major difference between E-Nets and Petri nets is that with E-nets, a transition may only fire when all its output places are empty. This property guarantees safe nets (1-bounded nets). Macro E-nets are E-nets complemented by macro constructs which can express priority queues, resource handlers,... They were developed specifically for the description of concurrent systems.

Although the various extensions that where added to the basic Petri nets enhance their modelling power, they complicate their analysis. For instance, the reachability problem is more complex with conditional transitions or inhibitor arcs. There is a trade-off to be made between high level constructs which simplify the

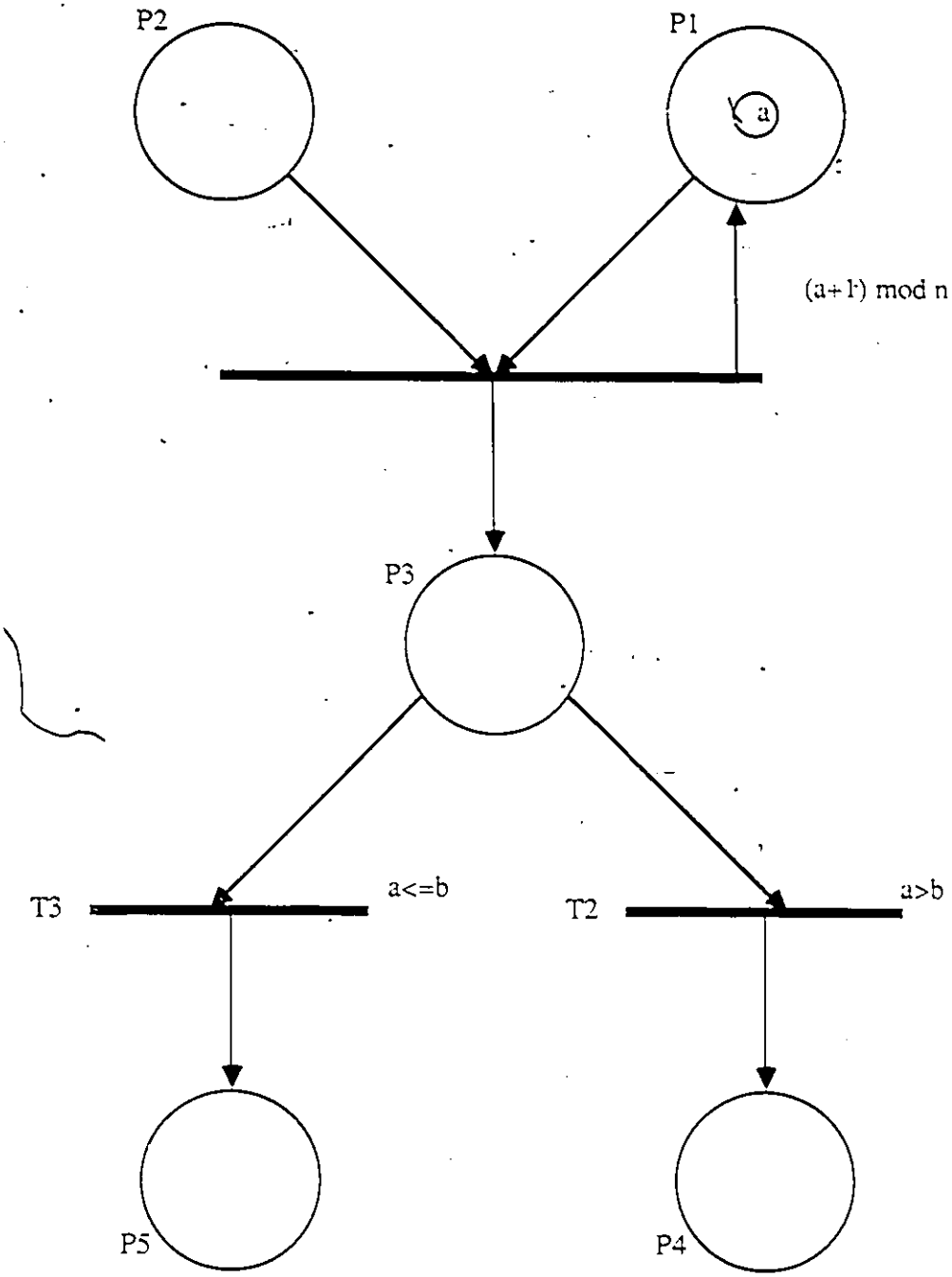


FIGURE 3-7 MARKED PETRI NETS WITH CONDITIONS

description of complex systems and simpler constructs which permit better and easier analysis of the system's behavior.

3.1.4 Use of Petri Nets in Microprocessor-Based System Design

As indicated in section 2.5, the design of a microprocessor-based system involves three major steps after the requirements have been obtained:

- system specification
- hardware and software design
- testing

The applicability of Petri nets in these design steps will now be examined. Petri nets were developed as a modelling tool. This means that once a tentative design is available, Petri nets can be used to model the behavior of the system. By analysing the model, it can be determined if its behavior is correct, and certain conditions like deadlocks and races. can also be detected. Petri nets can thus be used to test the hardware and software design of a microprocessor-based system before implementation. But can Petri nets be used in the specification of the system? Can they also be used to actually design the hardware and software?

Using Petri nets for the specification of event driven microprocessor-based systems would require describing the system at a higher level of abstraction than is usually done with Petri nets as the underlying hardware/software implementation is not yet defined. To express higher level concepts and keep the Petri

nets at a manageable size, extended Petri nets with such features as conditional transitions, marked tokens and inhibitor arcs will be needed. It must also be decided what tokens should represent at this level; either external events and/or data flow and/or control flow. All this information however, would tend to hide the essentials of the specification. Basically, Petri nets include too much graphical information to provide a natural representation of the system specification, but they could be used by experienced system designers.

At the design level, Petri nets can be used to model the proposed implementation and verify it before hand. The Petri net used at this level will be a refinement of the net defined at the specification level, since implementation details can be added. Some system measures, like average response time, can be computed using queuing theory by mapping the different markings of the net into states and analysing the resulting state machine in a similar manner as was done in section 2.3 [RAZO 84]. This requires considering the transitions as servers and finding the probability distribution of their firing time from the moment the transition is enabled. These analytical results can then be used to check if system specifications have been met.

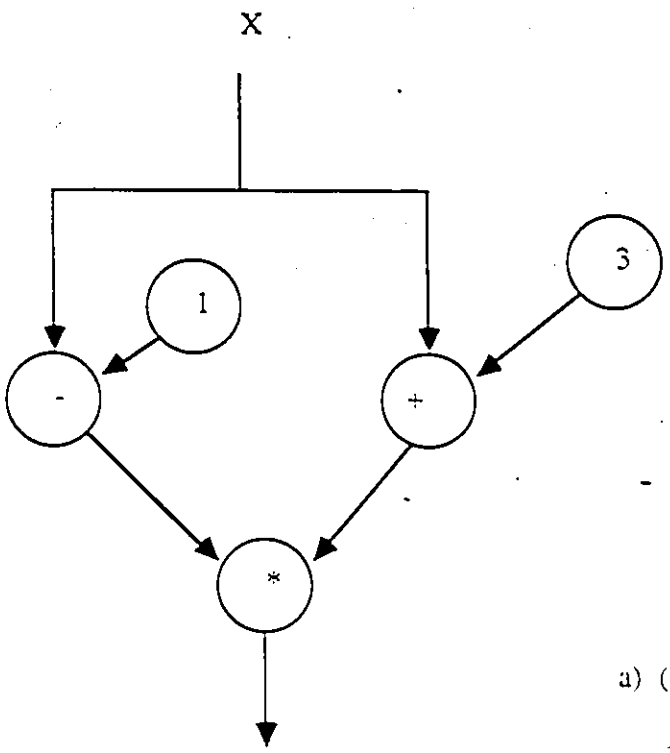
3.1.5 Data Flow Graphs

Since Petri nets are too complex to be used for system specification by non-computer experts, other graphical tools must be investigated. Data flow graphs may be a good candidate as they

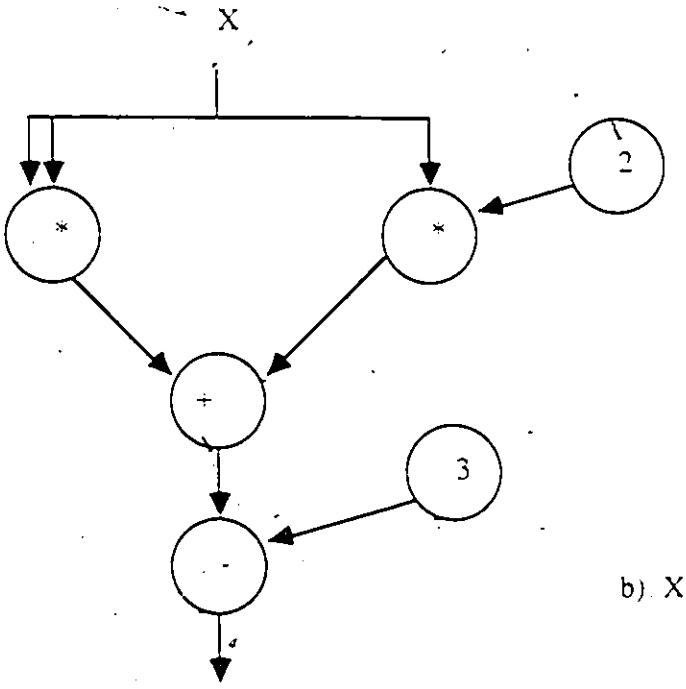
offer a much simpler syntax than Petri nets. Data flow graph is a generic name for various diagrams that describe algorithms by showing the flow of information from one function to the next. These diagrams [DAVI 82] [KWAN 84] represent data dependencies between the functions of the system as a function may only be executed once its input data is present.

Like Petri nets, data flow graphs are composed of nodes interconnected by arcs. The basic node is the operator node which defines some function that has input and output data associated with it. The arcs represent data flow from one function to another. In relation to Petri nets, the operator nodes are transitions while the arcs correspond to places. In the same manner as Petri net tokens flow from one place to another, tokens can be used to represent availability of data on the arcs of data flow graphs. Other nodes have been added to allow for decisions in the flow of data but their syntax varies from one author to the next.

Figure 3.8 shows two traditional examples of a data flow graph representing the equation (X - 1)*(X + 3) and its equivalent form X**2 + 2X - 3. The operator nodes represent simple binary operators. In the first example, the multiply operation can only be performed once the subtraction and addition operations are completed. Note that the subtraction and addition operations can be done in parallel since their data is available simultaneously. This is how data flow graphs implicitly represent concurrency.



a) (X - 1) * (X + 3)



b) $X^2 - 2X - 3$

FIGURE 3-8 EXAMPLE OF A DATA FLOW GRAPH

Data flow graphs have been closely associated with data flow machines and data flow languages [ACKE 82]. However if the description of the system is done at a higher level of abstraction than binary operations like addition and multiplication, data flow graphs can be used to represent a system which will be executed on a von Neumann machine [BOWE 85]. This is possible since the operator node can represent any function which has one or more input parameters and produce one or more results.

Data flow graphs can thus be used to specify event driven systems, providing the system is described at a much higher level of abstraction than simple operations such as addition and multiplication. However, for lower level software design, data flow graphs are more useful if the implementation is on a data flow machine.

3.2 SOFTWARE STRUCTURE DIAGRAMS

In this section, some accepted tools for the description of the software structure of real-time multitasking systems will be presented. These tools are built around the concept of tasks which was introduced in section 2.4. They show the inter-relationships between the various tasks of a system via their communication requirements.

3.2.1 Activity Channel Pool Diagrams

Activity Channel Pool (ACP) Diagrams [ALLW 80] were introduced by Ken Jackson to describe systems designed under MASCOT [JACK 83]. MASCOT is an acronym for "A Modular Approach to Software Construction, Operation and Test". It includes a formalism to express the software structure of real time systems (ACP diagrams); a methodology for design, implementation, testing and documentation of this software; and a kernel (operating system) to provide run-time support for this formalism.

As their name indicates, ACP diagrams are composed of three major constructs: activity, channels and pools. In this context, an activity is really a task (or a process) which can execute concurrently with other processes. In the diagrams, they are represented by circles. These tasks need to communicate between themselves; this is done through channels and pools. A channel is used to show directed communication from a producer to a consumer. The producer writes information in the channel which can then be read, in the same order, by the consumer. Channels are illustrated by a capital letter "I". Pools, on the other hand, are used to hold non-transient data that needs to be referred from more than one task, and are represented by a squared-off letter 'U'. An example of an ACP diagram is shown in Figure 3.9.

The ACP diagrams provide very little information about the system's design; they only show the tasks (or processes) of the system and their communication requirements. In addition, ACP

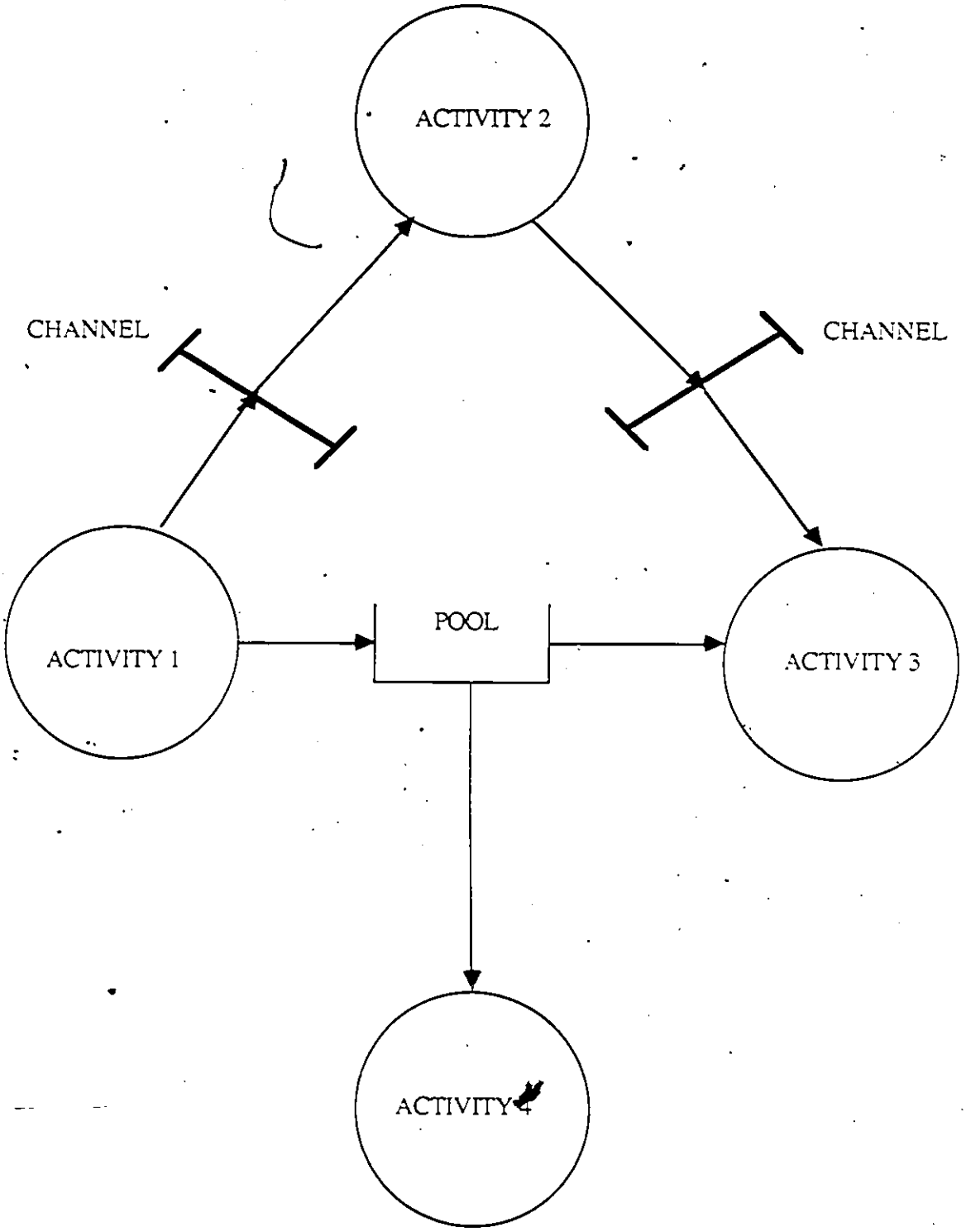


FIGURE 3-9 EXAMPLE OF AN ACTIVITY CHANNEL POOL DIAGRAM

diagrams can only be used effectively when designing with a concurrent language or operating system that supports such constructs as channels and pools. This is why an operating system is part of the MASCOT environment.

3.2.2 Structure Diagrams for Ada

A graphical notation which describes the software structure of an Ada based system has been developed by R.J.A. Buhr [BUHR 84]. Although this notation uses a number of concepts particular to Ada such as rendez-vous and packages, it can be adapted to other multitasking languages or operating systems.

In these diagrams, tasks are represented by parallelograms (since they can be executed in parallel) while rectangles represent packages. Smaller parallelograms and boxes are used to represent entry "sockets" in tasks and procedural "sockets" in packages respectively. A piece of data is indicated by a rectangle with rounded sides. Access connections between these modules are shown by directed arcs and represent procedure calls, entry calls or data accesses. Data flow can also be indicated on these diagrams by means of small circles with an arrow.

An example of such a structure graph is shown in figure 3.10. In this example for instance, task B calls entry C1 in task C and will receive some piece of data as a result of the rendez-vous. Note that the direction of the data flow is not necessarily the same as the access connection as a task can call an entry in another task to "send" and/or "receive" information.

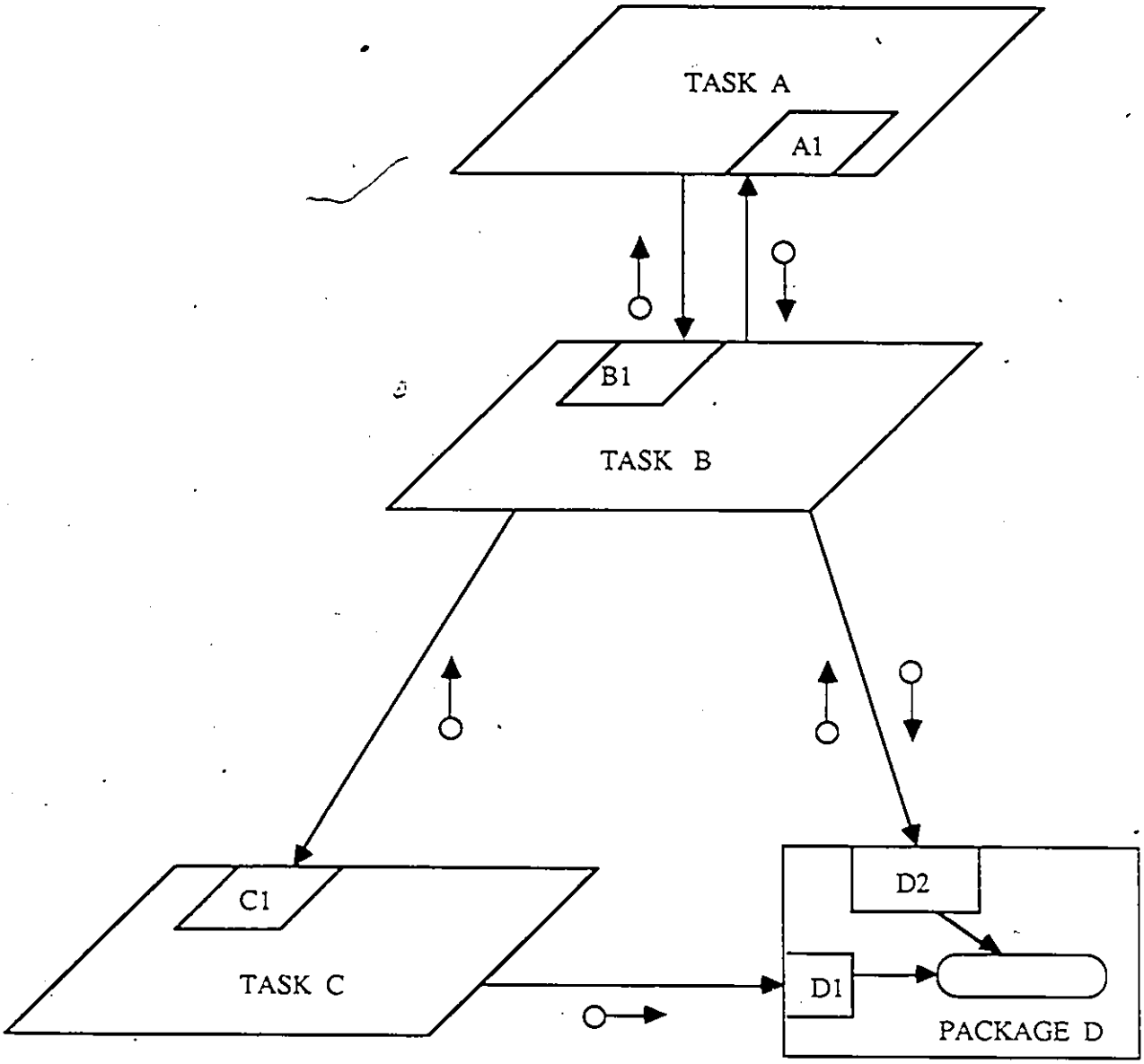


FIGURE 3-10 STRUCTURE GRAPHS

Structure graph are not only useful to show the inter-relationships between the various software modules of the system; they can also be used to check for design errors. Figure 3.10 shows a potential deadlock situation (if the communication mechanism is a rendez-vous type like Ada) where tasks A and B each call an entry in the other. If both tasks simultaneously call these entries, deadlock will result as both tasks will be blocked on their entry call.

These structure graphs provide more information about the system's design than ACP diagrams since the various relationships between the tasks, packages and data can be shown more precisely. For instance, a piece of data that can be accessed only through a package is indicated by drawing the data item inside the package. In the same manner, a task can be included in a package. Since the entry points of a task are labelled and the data flow is shown in these diagrams, it is possible to derive the task and package specifications (in Ada terms) directly from the diagrams.

3.2.3 Characteristics of Software Structure Diagrams

All the diagrams discussed in this section show the software structure of multitasking systems as they indicate the interrelationships between all the tasks of the system. This means that they can only be used as a design tool in a multitasking environment. Furthermore, some of these tools are closely associated with a particular operating system or concurrent language. This forces the system designer to learn a different tool for each operating system or concurrent language

that he uses.

The structure diagrams are tools closer to the implementation level. They cannot be used for system specification as they deal with such entities as tasks and communication mechanism, which should not be present at this level in the design.

Other types of software structure diagrams have been presented in the literature. A network specification and design environment [ARTH 86] was proposed as a complete toolset for the design of software in a multitasking environment. This particular tool defines tasks as a sequence of "high level operations" and can express concurrency. It has similar constructs and syntax as data flow diagrams but it cannot express any decision making.

3.3 INTERNAL TASK STRUCTURE DIAGRAMS

In this section, we will look at graphical tools which can represent the internals of a task. These tools represent the software of the system at a lower level of abstraction than the previous set of tools. They are more design tools than specification tools and are close to automatic software generation tools.

Two different graphical tools will be considered in this section: Communicating Finite State Machines (CFSM) and Specification and Description Language (SDL).

3.3.1 Communicating Finite State Machines

Since tasks communicate between themselves, we can use communicating finite state machines to show not only the internals of tasks but also their relationships with other tasks. A similar concept has been applied successfully to the description of communication protocols [BOCH 78].

Three types of states can be distinguished in these diagrams: wait states (which are represented by a double circle), decision states and transient states. The wait states indicate that the task is not running but is waiting instead for some information in the form of a message, a rendez-vous, etc... depending on the operating system or concurrent language used. When the information is received, the task may leave the wait state through a "double" arc. This "double" arc represents a transition which occurs as the result of an action in another finite state machine. The decision states are used to choose between different execution paths depending on available information. The transient states can be used to represent internal task activities. The graphical syntax just described is not a standard form of communicating finite state machines, but it can be used efficiently for task description.

A simple example of communicating finite state machines for the description of tasks is shown in figure 3.11. In this example, task A leaves state A.3 after receiving a message sent by task B in the transition from state B.2 to B.1 in B's state machine. This is what is meant by communicating finite state

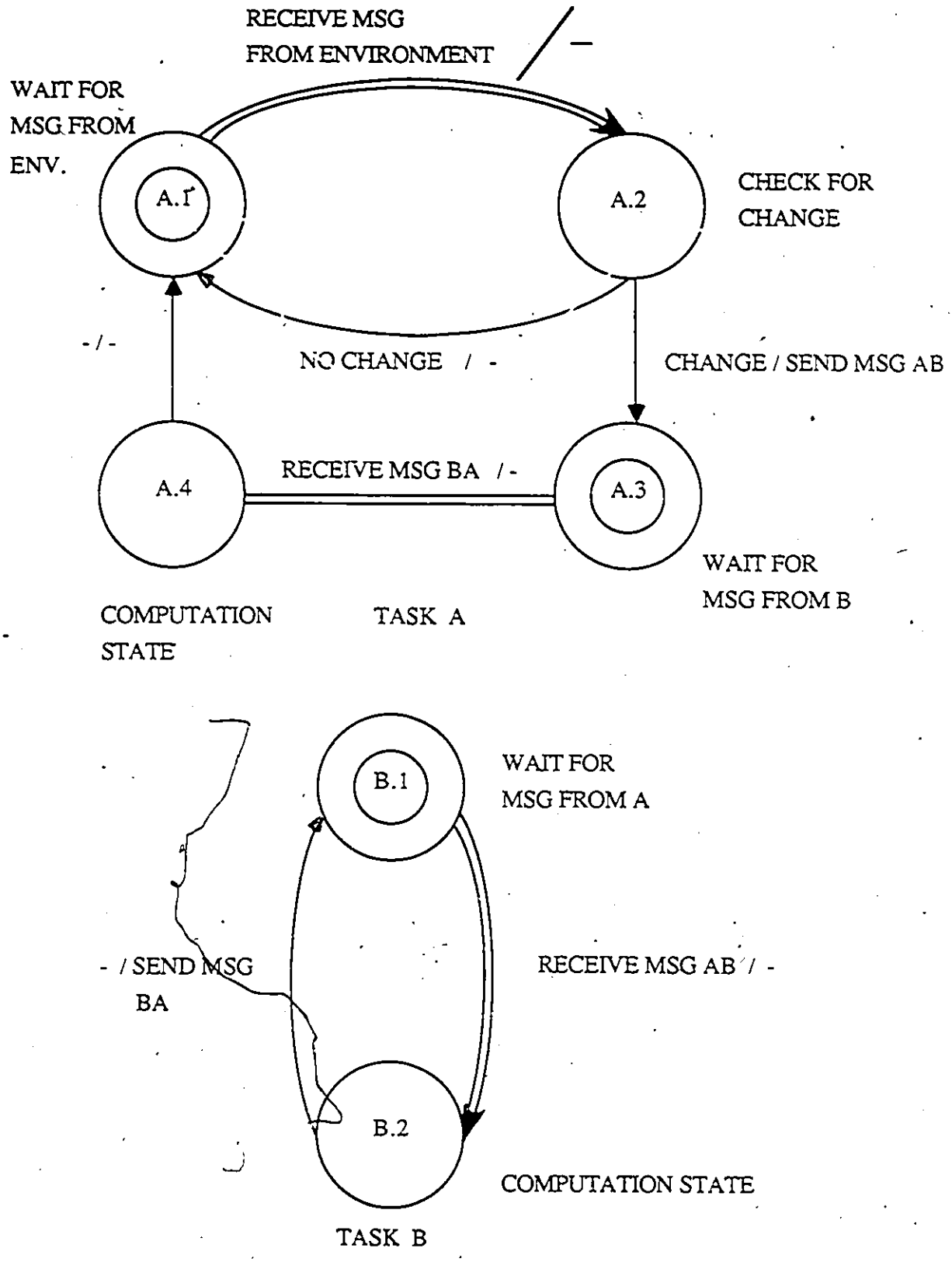


FIGURE 3-11 DESCRIPTION OF TASKS USING COMMUNICATING FINITE STATE MACHINES

machines. A decision state is used in task A's FSM to show a decision based on the message the task received from the system environment (likely through an interrupt routine).

Communicating finite state machines are a useful design tool. By analysing the reachability tree of the joint state machines of all the tasks in the system, certain error conditions such as deadlock can be detected. These finite state machines can also be used to generate skeleton code. The state machine would then be implemented directly in software and the programmer would simply have to code the actions.

However, as with the tools discussed in the previous section, this tool is not designed for non-expert users, as real-time software expertise is required to efficiently map the system requirements into tasks described by communicating finite state machines.

3.3.2 SDL

SDL was introduced as a Specification and Description Language for telecommunication systems. It is being recommended to the International Telegraph and Telephone Consultative Committee (CCITT) as a standard [CCIT 83]. In this context, specification is defined as the description of the required system behavior while description relates to the actual system behavior.

SDL has a rigidly defined syntax and include a number of different diagrams. Block diagrams are used to describe systems.

Each block can then be modelled by one or more processes which can be represented by process graphs. Process graph look like flowcharts with a number of extensions to express such concepts as communication between processes and state transitions within a process. SDL can thus describe the system at various levels of abstraction. It was considered in this section on low level design tools because they can be used all the way to the lowest level.

SDL is directed towards the design of complex telecommunication systems by expert designers. It is not a general purpose tool for all types of systems. Its importance lies in becoming a standard specification and description tool by the CCITT.

3.4 CONCLUDING REMARKS

In this chapter, a number of different graphical tool were described. In general, all these tools have a number of shortcomings which does not permit their use for both the specification and design of event driven microprocessor systems. Petri nets may be useful to model the system at the design level, but contain too much information to be suitable for the specification of these systems. Data flow diagrams can be used for system specification but would require certain modifications to be used for lower level design on a von Neumann architecture. ACP diagrams and structure diagrams assume a particular

multitasking environment and some experience by the user to be able to think in terms of this environment. Communicating finite state machines describe the system at a low level of abstraction which cannot be reached directly from the system requirements without a lot of design experience. Finally, SDL is a complex and complete toolset valid for certain classes of systems but which again requires expert users.

The main conclusion of this chapter is that there exists no graphical tools which permit the non-computer specialist to specify an event driven system. Such a tool would be needed to interface the system requirements with lower level design tools such as communicating finite state machines and other tools familiar to computer specialists. Presently, as no such tool is available, the various system actions are usually partitioned into tasks (in a multitasking system) or assigned to processors (in a multiple processor system) in an ad hoc manner. This does not necessarily result in an acceptable system performance as real-time constraints might not be met. In such cases, the solution is either to speed-up the processor(s), add more processors (in the case of a multiprocessor system) or change the partitioning and priorities of certain tasks. Thus, tools that would ease the partitioning of systems actions into tasks or the assignment of actions to processors by verifying system performance before implementation would be very welcomed by system designers.

CHAPTER 4

PROCESS ACTIVITY DIAGRAMS

As stressed in the previous chapter, there is a need for a graphical specification tool for event driven systems which can be used by non-computer specialists. It would also be useful if this tool could help the system designer in a multitasking environment visualize the various actions that must be executed by the processing system in order that he may be able to better partition these actions into the various tasks of the system. This chapter introduces Process Activity Diagrams (PAD's), a graphical tool developed for such purposes.

The first section of this chapter will discuss processes and activities, the building blocks of PAD's. PAD's have two graphical components, Activity Sequence Networks (ASN's) which will be introduced in section two and Activity Timing Charts (ATC's) which will be described in section three. Finally, in section four, PAD's will be compared to some of the other graphical tools discussed in the previous chapter.

4.1 PROCESSES AND ACTIVITIES

Event driven systems can be specified at a number of levels of abstraction. At the highest level of abstraction, a system

will be considered as being made up of a number of processes. In this context, a process is defined as a series of actions which must be executed in response to some external event occurring in the system's environment. These processes can be further partitioned into sub-processes, and so on until the lowest level of interest to the system designer is attained. The entities at this level will be called activities. An activity is defined as a meaningful sequence of instructions which is executed from beginning to end without waiting for additional data from other activities. This is, in essence, the same meaning as in section 2.3.

A process can thus be decomposed into a number of activities which must be executed in a given sequence and within a certain time (if real-time constraints are present) in response to an external event. Process activity diagrams [KRIE 86] have two graphical components, Activity Sequence Networks (ASN's) and Activity Timing Charts (ATC's), to represent these concepts. The Activity Sequence Networks (ASN's) are used to show the sequencing relationships between the activities of a process. They are a modification of the data flow graphs which where presented in the last chapter. The Activity Timing Charts (ATC's) are used to illustrate the timing requirements of the various activities within a process. They are related to bar charts and Gantt charts [CHAS 81].

4.2 ACTIVITY SEQUENCE NETS

Like data flow graphs, the ASN's are composed of a number of nodes connected by directed arcs. Seven different types of nodes are defined to represent all activity sequences and their associated control flow. These various nodes enable the designer to express such schemes as concurrent execution of activities, choosing between different activity sequences depending on some condition, time-outs, etc...

The directed arcs are called triggers. The triggers are said to be activated when the condition they represent has been satisfied. This activation is only momentary as it is the responsibility of the node to remember or disregard the trigger. There are two types of triggers: internal and external. Internal triggers represent conditions inside the network such as availability of data, completion of activities, etc... They are illustrated as arcs which originate and terminate at nodes in the same network. External triggers, on the other hand, are usually generated as a result of conditions in the "outside world"; an example would be an interrupt generated by some I/O device. Graphically, an external event is represented by an arc with no node at its origin, just a label identifying the external event.

The various nodes of ASN's will now be presented, starting with the activity initiator node.

4.2.1 The Activity Initiator Node

The activity initiator node is the most basic node of the ASN's as it is the only one that deals directly with the execution of activities. An activity initiator node is shown in figure 4.1. The activity initiator node may have any number of input triggers (arcs incident to the node) and any number of output triggers (arcs directed out of the node). The input triggers represent conditions that must be satisfied before the activity associated with the node may be started. These conditions may be related to availability of data (an activity may only start when all its input parameters are available) or they may represent control information (for example, precedence relations between activities not based on actual data). When an input trigger is activated, the activity initiator node remembers it until all the input triggers of the node have been activated. At this point, the node forgets past input triggers and the activity associated with the node may be executed. Once the activity is terminated, all the output triggers are activated to inform the other nodes connected to them that the activity is completed.

The activity initiator node concept is similar to the operator node in data flow graph. As the name activity implies, the activity initiator node deals with higher level functions than the operator node in data flow graphs. In addition, the arcs in data flow graphs may only indicate data dependencies (data flow), while the triggers in ASN's can also represent control flow.

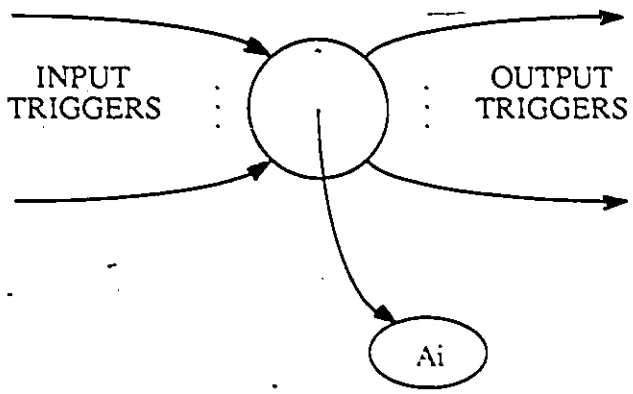


FIGURE 4-1 ACTIVITY INITIATOR NODE

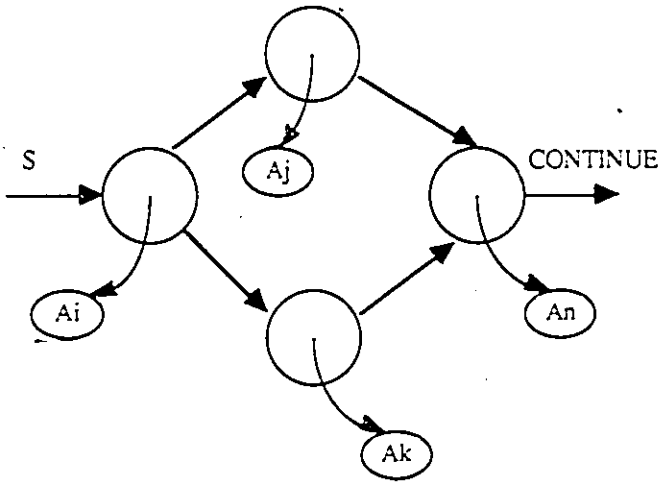


FIGURE 4-2 SIMPLE ASN

An example of a partial ASN using only activity initiator nodes is shown in figure 4.2. From the operation of the activity initiator node, the behavior of this ASN can be described as follows. Activity Ai can only be started once the external event S has occurred. When it is completed, activities Aj and Ak may be started immediately, depending on the availability of processors. Activity An can only start once both activities Aj and Ak are terminated, possibly because it needs data from both of these activities.

As can be seen in this simple example, concurrency between the activities is shown explicitly in ASN's. This concurrency will only be real if the system is implemented on a multiple processor system and activities Aj and Ak are assigned to different processors in the same period of time.

4.2.2 Additional ASN's Nodes

An ASN describes a process, that is the sequence of activities which is executed in response to an external event. For this reason, every ASN is started by an external event, simply denoted as the starting event of the ASN. In figure 4.2, event S is the starting event. To indicate that a response to an event (a process) is terminated, an END NODE is needed. The END node is simply illustrated by a circle with a capital letter "E" inside, as shown in figure 4.3. The operation of this node is quite simple, the node will remember all activations of its input triggers and when all of them have been activated, the ASN is terminated. This node is the only one which does not have any

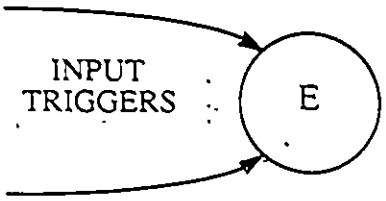


FIGURE 4-3 END NODE

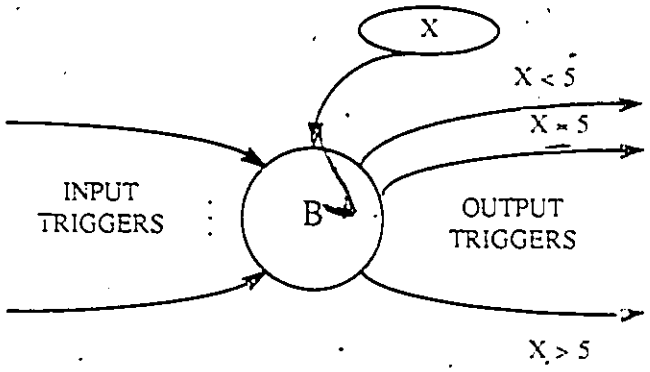


FIGURE 4-4 BRANCH NODE

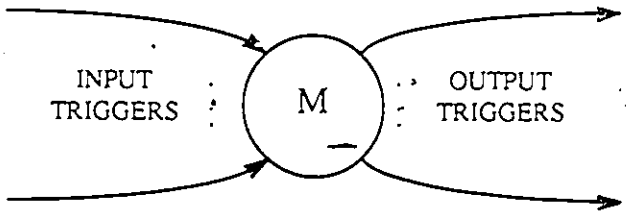


FIGURE 4-5 MERGE NODE

output triggers.

To represent more complex sequences of activities, additional nodes need to be defined. These are the branch, merge, wait, gate and timer/counter nodes which will now be presented..

An important addition to the capabilities of the simple ASN described in figure 4.2 would be the ability to choose between various sequences of activities depending on some condition. The BRANCH NODE (figure 4.4) provides such a feature. It operates as follows: it remembers all occurrences of its input triggers and when all of them have been activated, a condition variable is examined, and only the output trigger(s) corresponding to the proper condition is(are) activated. In the ASN diagram, the BRANCH node is represented by a circle with a capital "B" inside. The condition variable is indicated inside an ellipse while the conditions are labelled on the output triggers.

The BRANCH node will create different paths (activity sequence) in the ASN's which will not be executed concurrently because triggers will only be activated in the path selected by the BRANCH node. At some point, these paths must merge back together and since all the other ASN nodes are of the "AND" type (i.e. ALL their input triggers must have been activated for some action to occur) an "OR" type node is needed. This is the function of the MERGE NODE (figure 4.5) which will activate all its output triggers when ANY ONE of its input triggers is activated. Note that, unlike the other nodes, the MERGE node does not need to remember the activation of its input triggers as it

immediately takes action when any one of its input triggers is activated.

The emphasis in PAD's is to represent concurrency not branching. This is why any node, except the end node, may start a concurrent activity sequence by having more than one output trigger. Also all the nodes seen so far, except the merge node, can join back these parallel paths. However, all these nodes have a function associated with them. In some cases it is necessary to join concurrent paths without executing an activity or checking a condition. This is the purpose of the WAIT NODE (figure 4.6) which operates as a simple "AND" gate. It remembers its input triggers and, when all of them have been activated, it simply activates all its output triggers. It can thus be considered as an activity initiator node without an activity.

To provide a better understanding of the operation and interaction of the nodes introduced so far, figure 4.7 shows an example of an ASN which includes all these nodes. When the starting event occurs, the condition variable "a" is tested. If it is true, activities Ai and Aj may be started concurrently, otherwise activity Ak will be executed. The wait node is necessary in this example to make sure that both activities Ai and Aj are completed, before this path of the branch node ("a" true) is merged with the other path ("a" false). The end node will be reached when either path is completed; as only one was activated, this is correct.

In certain cases it may be required to block the propagation

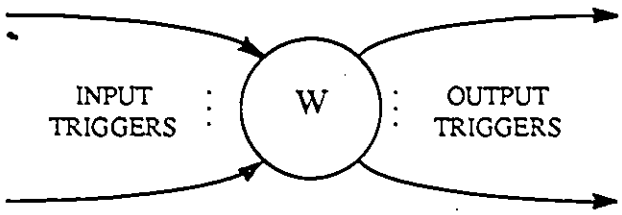


FIGURE 4-6 WAIT NODE

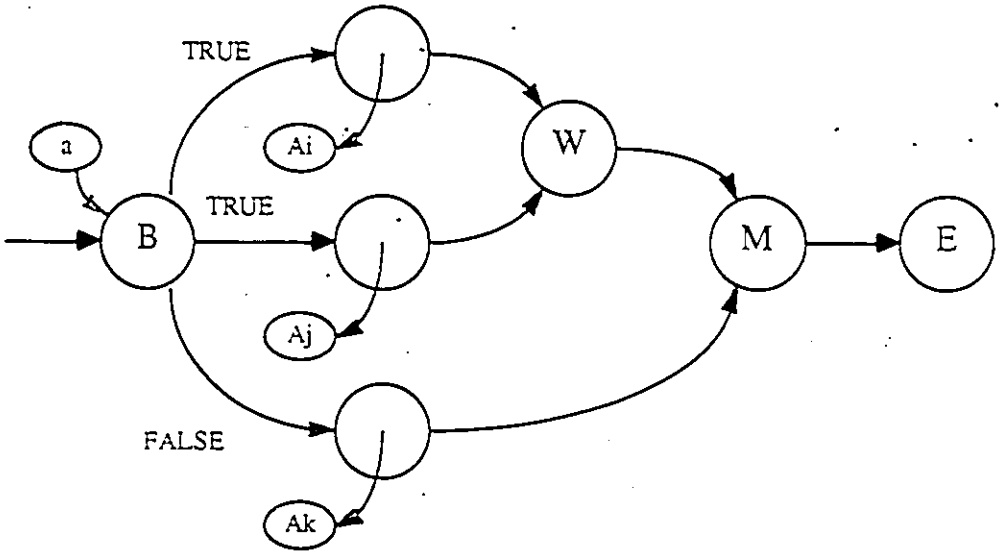


FIGURE 4-7 ASN WITH BRANCH NODE

of triggers if certain conditions are satisfied. If these conditions can be represented by triggers, then a GATE NODE (figure 4.8) can be used. The operation of this node is slightly more complex than the other nodes as it has two additional inputs. These are the enable input represented by a small light square on the edge of the circle and the disable input represented by a dark square. Only one trigger may be connected to the enable and disable input of the gate. If more than one trigger is required then a wait or merge node may be used prior to the enable or disable input. The gate node may also have any number of input and output triggers. It operates in the same manner as the wait node with one major difference: the gate only remembers triggers when it is enabled. Initially, the gate is disabled; it must be enabled by the activation of a trigger on its enable input. Once enabled, the gate may remember its input triggers and if all of them have been activated it will activate all its output triggers and automatically disable itself. However, if its disable input is activated, the gate loses its memory by forgetting all past input triggers and ignores all future input triggers unless it is enabled again.

The TIMER/COUNTER NODE (figure 4.9) is the most complex node. Like the gate node it has an enable and a disable input, but it may only have a single input trigger. Initially it is disabled and not operational. When it is enabled (by the occurrence of a trigger on its enable input), an initial value (represented by a counter variable in the ellipse beside the node) is entered in its "counter register". Once enabled, it will count the

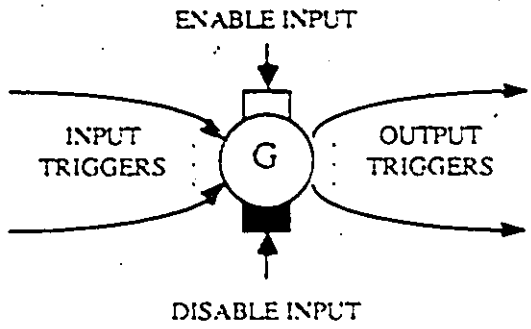


FIGURE 4-8 GATE NODE

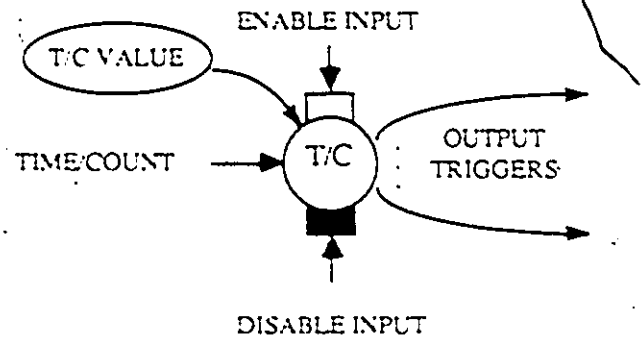


FIGURE 4-9 TIMER COUNTER NODE

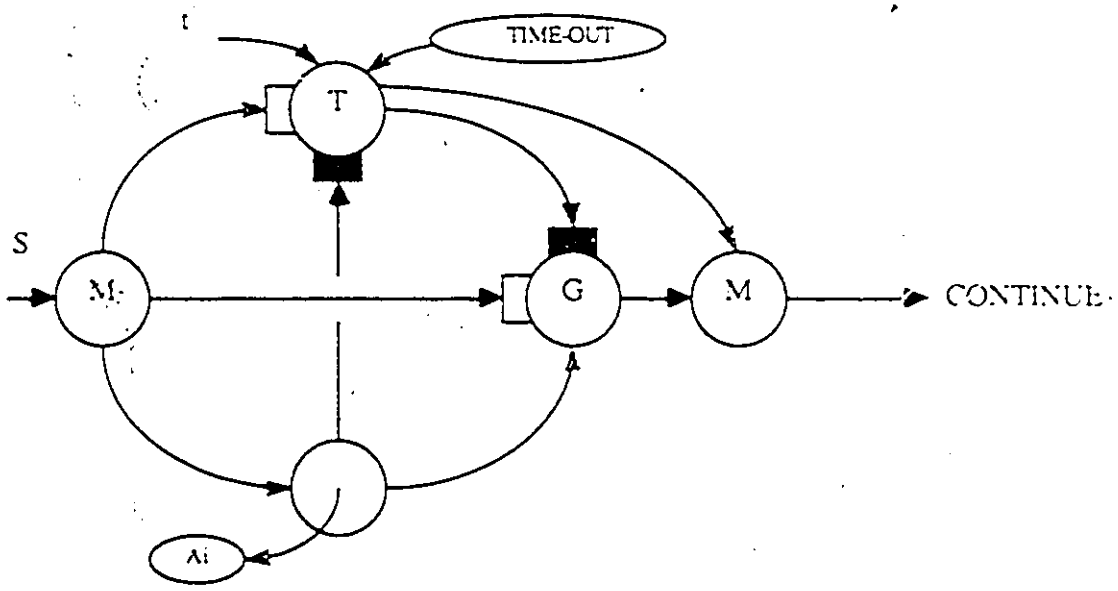


FIGURE 4-10 ACTIVITY WITH TIME-OUT

activation of its single input trigger, by decrementing the "counter register" by one each time the input trigger is activated. If the count reaches zero it will activate all its output triggers and automatically disable itself. However, if at any time its disable input is activated it will stop counting and must be enabled again to operate. Note that when it is re-enabled the initial count is loaded once more in the "counter register" (the count does not resume where it left off). If the input trigger is the system clock (usually defined as an external event), then this node operates as a TIMER node, otherwise it is a COUNTER node.

To understand the usefulness of the gate and timer node, an example of an activity with time-out (figure 4.10) will be discussed. In this example, the starting event will enable the gate and timer nodes as well as initiate activity Ai. The timer will then count clock pulses. If the counter reaches zero (the time allowed for the execution of activity Ai has expired), the gate node will be disabled blocking the propagation of the output trigger of the activity initiator node in case activity Ai terminates. However, if activity Ai is completed before the timer expires, then the timer will be disabled and the activity initiator node output trigger will be allowed to propagate through the gate.

4.2.3 Remarks on ASN

Now that all the ASN nodes have been introduced, certain

remarks and rules will be made concerning their use and interconnections:

- Each ASN describes one process, hence several ASN's are required to specify a system.

- Every ASN must have one and only one starting event. It may have more than one external event (external trigger) but only one of them may start the ASN.

- Initially an ASN is said to be "dormant"; it is "awakened" when the starting event occurs and goes back to "sleep" when the end node is reached.

- When an ASN is awakened, all its gate and timer/counter nodes are initialized in their disabled state.

- ASN are a nested construct: a high level activity can be expanded into an ASN when refining the system specification. In the same manner, a portion of an ASN can be considered as a single activity when specifying the system at a higher level of abstraction.

- An ASN may contain a feedback path, that is, an output trigger of a node can be brought back in the ASN (via a merge node) to a point previous to its own input triggers. It is called a feedback path since the triggers in the path may be activated once more.

- Note that the feedback trigger must go to a merge node. Otherwise deadlock will occur, since a node would be waiting forever for a condition which could only occur if the node would

fire its output triggers.

- All the activities started in the feedback path must be terminated before the feedback trigger can be activated. It is up to the designer to make sure that the proper wait nodes are inserted in the ASN to insure this.

- Each ASN must terminate with one END node unless it has an unconditional feedback path which means that once started, the process will go on forever.

- All nodes, except the merge and the timer/counter, perform a logical AND function of their input triggers before taking some action. Since it has been defined that the triggers are only activated for a short period of time (like pulses), it is the responsibility of the node to remember their occurrence. It is important to note that the node only remembers one occurrence of a trigger activation, it does not count them. These multiple activations of an input trigger should not occur in a properly constructed ASN.

- In the case of the branch node, the different conditions associated with the condition variable must cover all possible values of the condition variable without any overlap. In other words, it is essential that there is always one and only one of the different conditions that corresponds to the condition variable. This rule is necessary because if there would be a situation where no condition represents the state of the condition variable, the branch node will not activate any of its output triggers, blocking the ASN at this point. Also, if more

than one condition is satisfied, then an error will occur when the different paths are merged back together because multiple triggering will result.

- If needed, data flow between the activities of a process can be indicated along the triggers in the same manner as done in data flow graphs or in Buhr's structure diagrams.

4.3 ACTIVITY TIMING CHARTS

The purpose of Activity Timing Charts (ATC's), the other component of PAD's, is to illustrate the execution time of a process by combining the various execution time of the activities which make up an ASN. Each activity associated with an activity initiator node is represented by a timing bar as shown in figure 4.11. The length of this bar is proportional to the execution time of the activity. Since an activity usually needs input data and produces some result(s), this execution time can be decomposed into three distinct time intervals: the input data transfer time, the execution time of the activity itself and the output data transfer time. During early design stages, these times are undefined, hence estimates or bounds (maximum or minimum times) may be used. Note that even once the activity is implemented (coded), its execution time may not be fixed but data dependent. The data transfer times are also highly variable as they depend on the system's architecture and the communication mechanism used. Hence, average or worst case (maximum) execution times may be used in the early design stages.

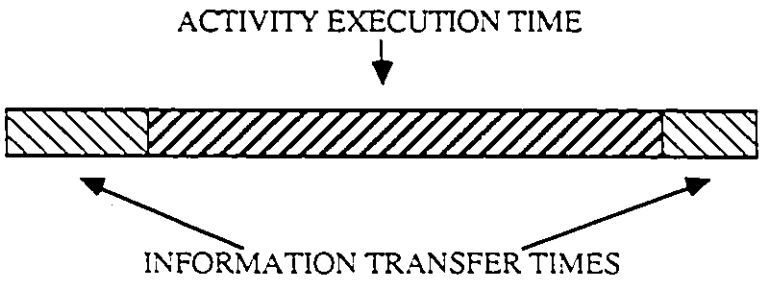


FIGURE 4-11 ACTIVITY TIMING BAR

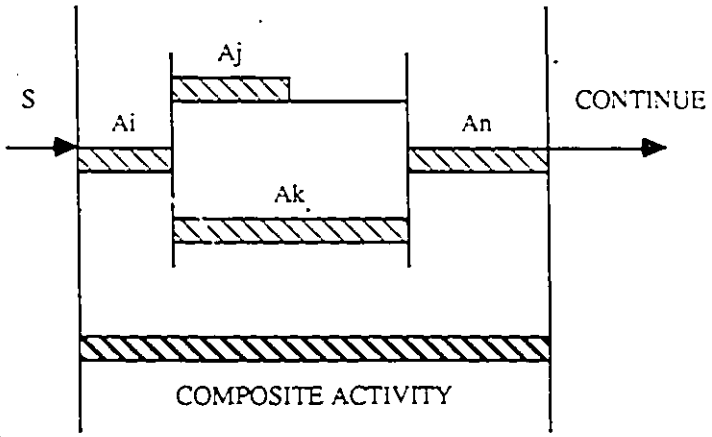


FIGURE 4-12 EXAMPLE OF AN ATC

An example of an ATC which corresponds to the ASN of figure 4.2 is shown in figure 4.12. ATC's are useful to get a rough estimate of the system's response time to external events. The execution time of an ASN would be equal to the length of the longest activity execution path. Note that these diagrams assume that activities are executed as soon as they are ready to run and that there is always a processor free which may execute the activity, hence providing for maximal concurrency. Note that the ATC can only show a single scenario for each ASN. If there is a branch or gate node, a different activity sequence will be executed depending on the path chosen. Thus, a different ATC must be derived for each different path through the ASN (or scenario).

By using techniques such as CPM (Critical Path Method) [RIGG 77], it is possible to find which activities could benefit more from a reduction in execution time (by using a faster processor or a better algorithm). When probabilities are associated to activity execution times, PERT (Program Evaluation and Review Technique) [CHAS. 81] can be used to obtain expected process completion times. By looking at one ATC or a group of ATC's and checking which activities may be executed in parallel, it is also possible to obtain an activity to processor allocation which maximizes concurrency. This activity to processor allocation defines the number of processors in the system and which activities each processor may execute.

Figure 4.12 also illustrates the nested property of PAD's as the entire ATC can be considered as a single composite activity at a higher level of abstraction. This feature of PAD's makes it

possible for the designer to completely specify a system by doing successive refinements starting from a coarse specification of the system. This will be discussed in more details in the next chapter.

4.4 PAD'S VERSUS OTHER GRAPHICAL TOOLS

As presented in the previous chapter, a number of graphical tools presently exist for the design of concurrent systems. In this section, PAD's will be compared to some of these tools, starting with data flow graphs.

4.4.1 PAD's and Data Flow Graphs

As mentioned previously in this chapter, ASN's can be considered as modifications or extensions of data flow graphs. In this context, the basic data flow node, the operator node, would be equivalent to the activity initiator node in ASN's. The other ASN's node have similarities with some of the data flow graphs node (like the distributor and selector node) but they have no exact equivalence.

The major difference between these two representations is that ASN's are based on control flow, while data flow graphs are based on data flow. In ASN's, data flow can be added to the representation (via data tokens along the triggers as mentioned in the previous section), while in data flow graphs, control variables can be used to simulate control flow. Hence a number of

systems can be described using either representation. However, if the system is heavily data oriented then data flow graphs would be preferred; on the other hand, in a highly control oriented system ASN's would be easier to use.

4.4.2 PAD's and Petri Nets

Petri nets are closely related to data flow graphs, but tokens in Petri nets can represent more than data flow; this is why they have gained wide acceptance as a tool to describe the behavior of concurrent systems. For these reasons, it would be more relevant to see how PAD's compare with Petri nets.

To understand the relationship between Petri nets and PAD's, an equivalent Petri net will be derived for each ASN node. The first observation that should be made is that in general, triggers in ASN's correspond to places in Petri nets. This should be obvious from the definition of these entities: in Petri nets, the presence of a token in a place indicates that some condition is satisfied, while in ASN, this is represented by the activation of a trigger.

The equivalence of the activity initiator node (figure 4.13) follows directly from the above observation. The input triggers become input places and when a token is present in each of these places, the transition may fire, i.e. all conditions are satisfied for the execution of the activity. The activity is represented by a delayed transition because tokens are inserted in the output places only after the activity is completed. Hence

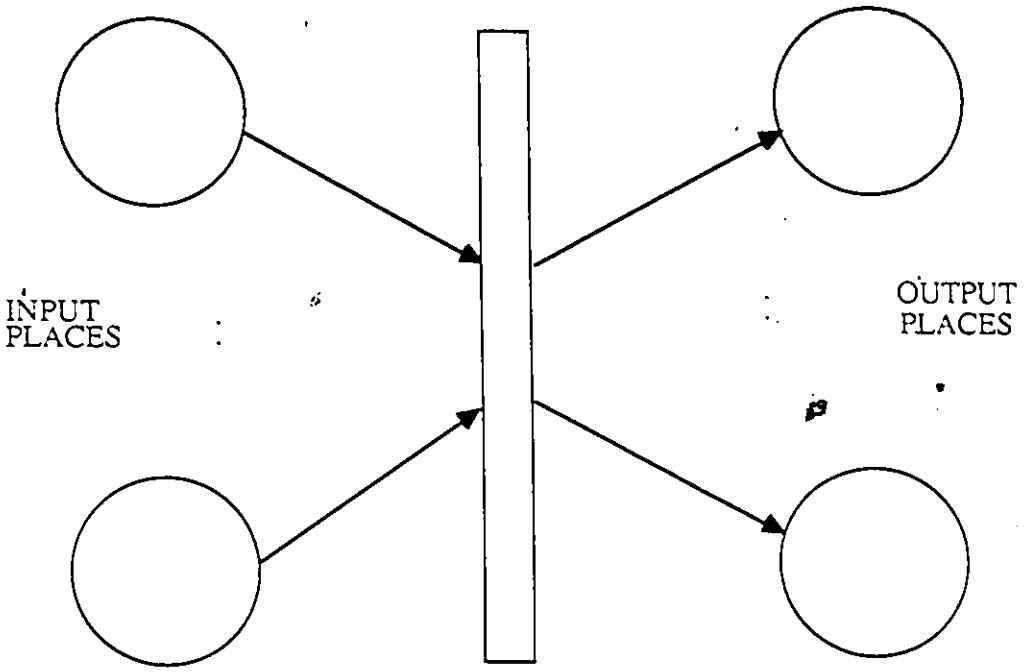


FIGURE 4-13 PETRI NET EQUIVALENCE OF
AN ACTIVITY INITIATOR NODE

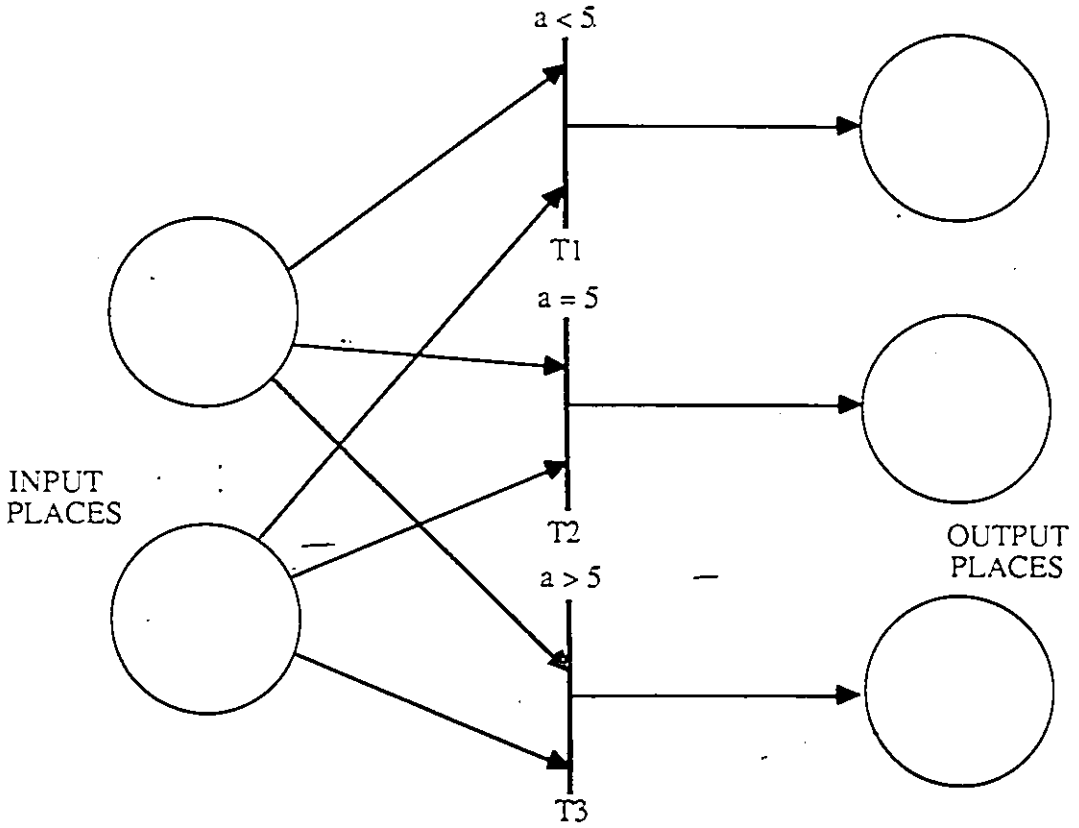


FIGURE 4-14 PETRI NET EQUIVALENCE OF A BRANCH NODE

the delay associated with the transition is related to the activity execution time.

Since the wait node can be considered as an activity initiator node without an activity, it can be represented in Petri nets in the same way as the activity initiator node. The only difference would be that a normal transition is used instead of the delayed transition as there are no delays associated with the "firing" of a wait node.

The branch node is more complex to represent in Petri nets. The input triggers become places which must enable a number of transitions. Conditions must then be added to these transitions to indicate which one will fire based on the condition variable. Figure 4.14 shows an example of this equivalence. Depending on the value of variable "a", transition T1, T2 or T3 will fire, thus putting a token in a different place.

The merge node is the simplest to represent in Petri nets as it does not really require any additional place or transition. The input triggers of a merge node are output triggers of other nodes (except for external events). In Petri nets, this can be simply indicated by having output arcs of different transitions directed to the same place. Figure 4.15 illustrates the equivalence of a merge node where a token from transition T1 or T2 will enable transition T3.

The gate node is more complex to model in Petri nets because of its enable and disable inputs. The gate can be considered as

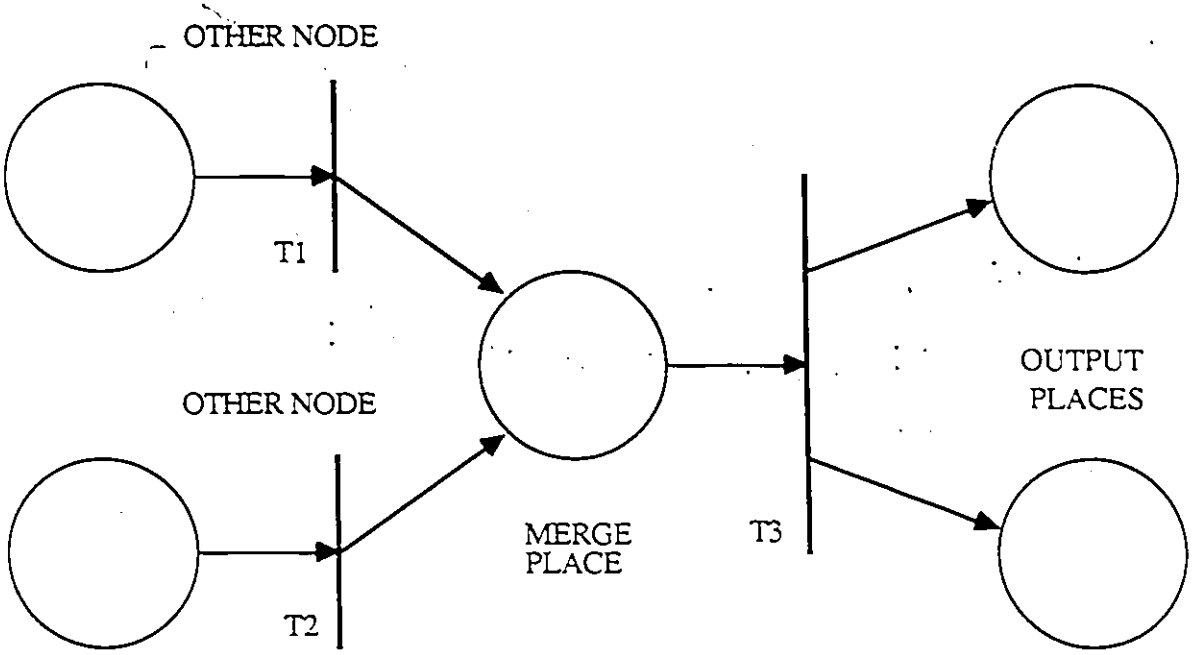


FIGURE 4-15 PETRI NET EQUIVALENCE OF A MERGE NODE

being in one of two states: enabled or disabled. To remember its state, a flip-flop like representation is needed since a transition behaves as an AND gate. This is illustrated in figure 4.16. There will always be a token in either place D' or E', but never in both. Initially the gate is disabled, hence a token must be present in D'. If the gate is enabled by putting a token in the "enable" input, T2 will fire placing a token in E' to effectively enable the gate. Whenever the gate is enabled, the "gate" transition will fire when there is a token present in all of its "input trigger" places. Note that if the "gate" transition fires, a token is inserted in D' as the gate must return back to its initial disable state after activating its output triggers. The gate can also be disabled by putting a token in the "disable" place and when T1 fires, a token is inserted in D'. Whenever the gate is disabled, tokens in the "input trigger" places are absorbed by the transitions Ti to Tn since input triggers must not be remembered when the gate is disabled.

The most complex node of the ASN, the timer/counter node is also the most complex to model in Petri nets. To model it properly, the timer or counter variable must be shown explicitly in the Petri net. The easiest way of doing this is to use a token marked with the value of the counter variable. A possible model of the timer/counter node is shown in figure 4.17. This representation is similar to the gate node because the timer/counter also has an enable and disable input. Note that the figure shows the node in the enable state.

When the timer/counter is enabled, transition T2 fires,

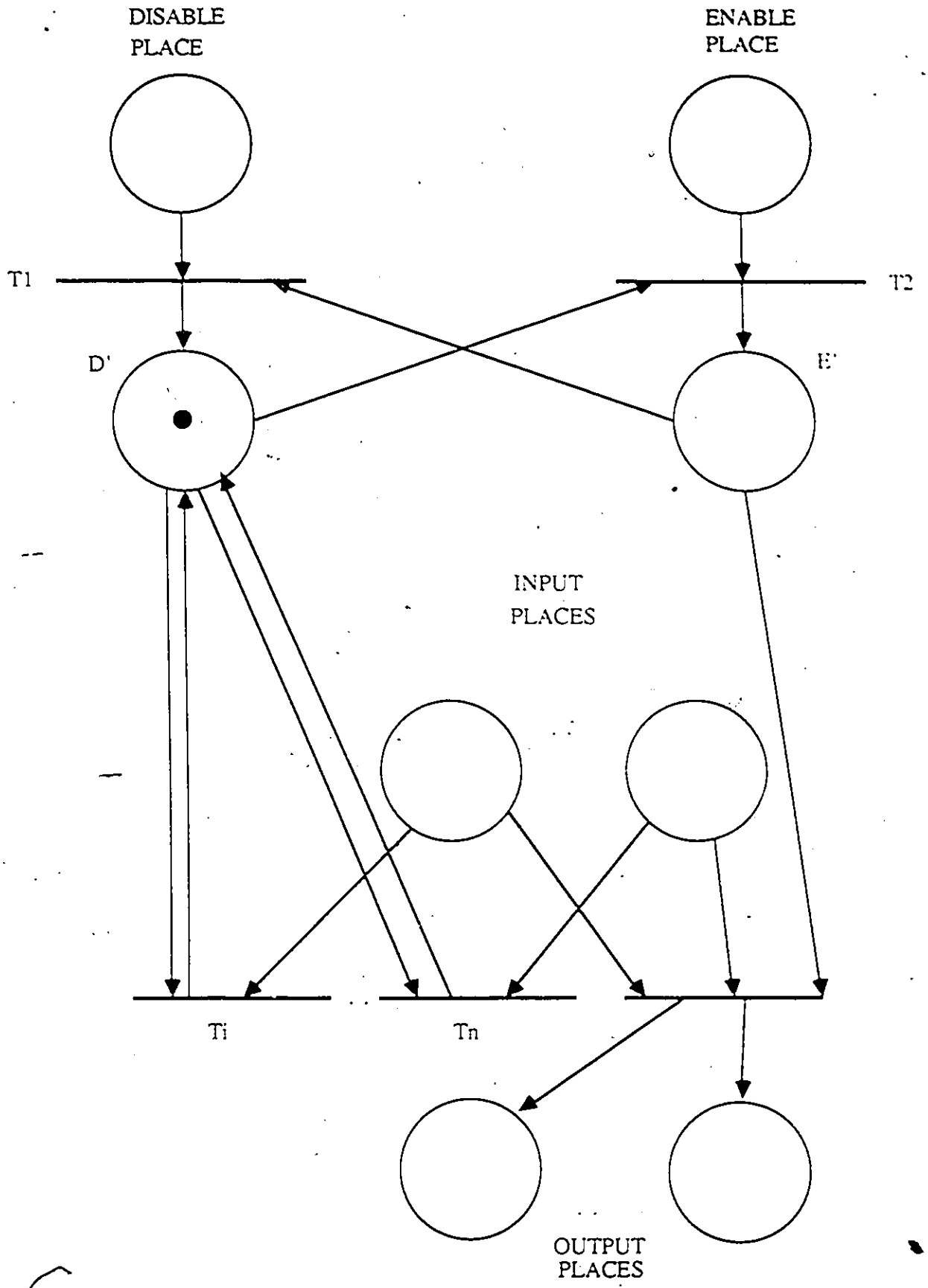


FIGURE 4-16 PETRI NET EQUIVALENCE OF A GATE NODE

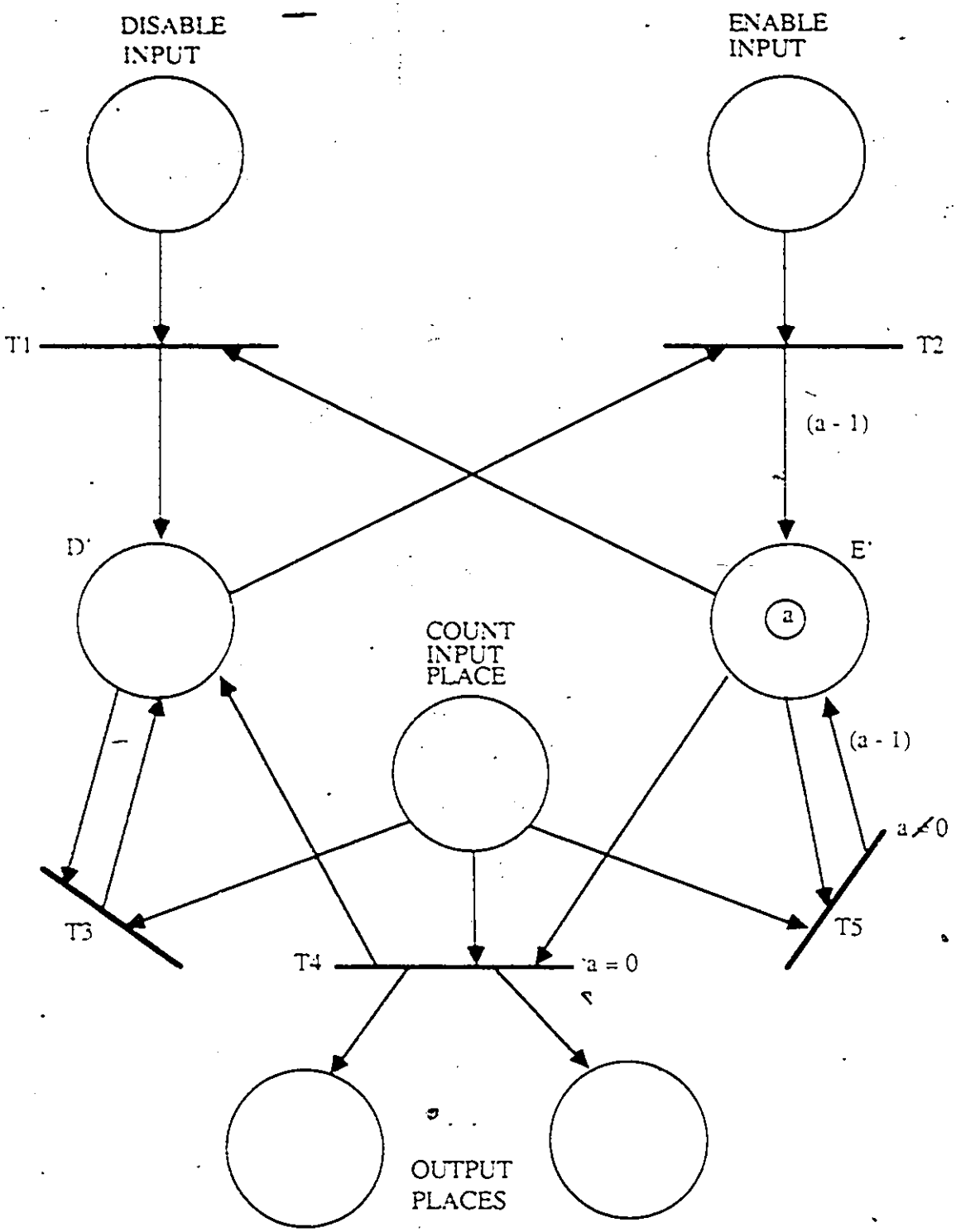


FIGURE 4-17 PETRI NET EQUIVALENCE OF A TIMER/COUNTER NODE

placing a marked token in E' with the value of the counter variable. This indicates that the timer/counter is now enabled. Afterwards, each time a token is present in the "count" place (i.e. the input trigger occur), the marked token is taken out of E' by transition T5, decremented by one, and put back in its place (provided that the counter has not been disabled). However, if the counter reaches zero, transition T4 will fire instead of T5 and the timer/counter will be automatically disabled by putting a token in D'. The timer/counter can also be disabled by inserting a token in the "disable" place enabling transition T1. When T1 fires, the marked "counter" token in E' will disappear disabling the counting process. Note that any token inserted in the "count" place while the node is disabled will be absorbed by transition T3.

From the above discussion it can be seen that in most cases, an ASN node requires quite a number of places and transitions when translated into a Petri net. To show the resulting translation of a complete ASN into a Petri net, an example will be given. Figure 4.18 illustrates an ASN which contains all the different nodes except the wait node. Its operation is quite simple: when the starting event occurs, activity 1 is started with a time-out as shown in figure 4.10. If activity 1 finishes before the timer expires, activities 2 and 3 are started concurrently. When they are both completed a condition is checked. If it is true the process is terminated, otherwise a feedback trigger is initiated. The corresponding Petri net representation of this ASN is shown in figure 4.19. Note that in

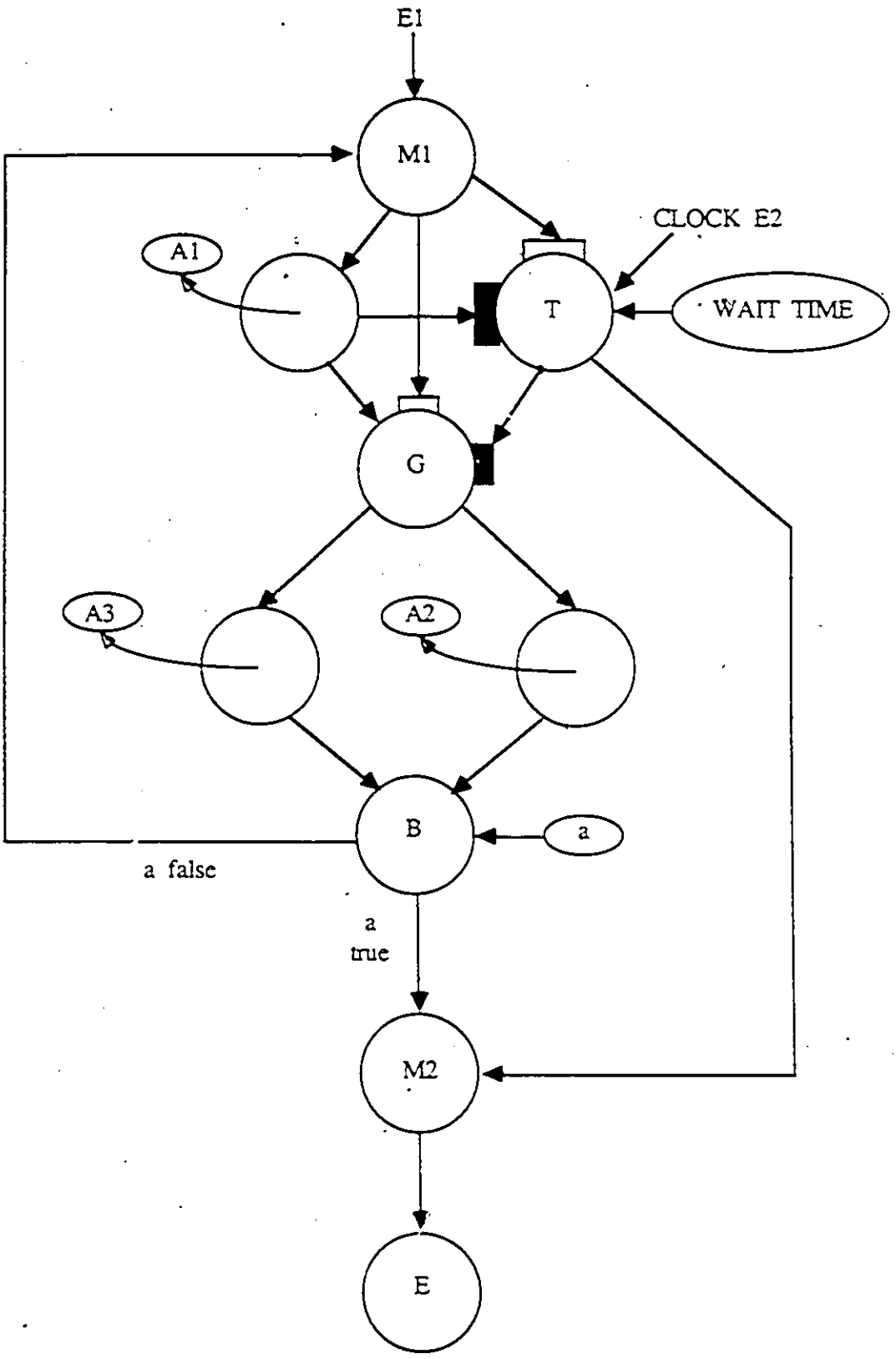


FIGURE 4-18 ASN WITH ALL BASIC NODES

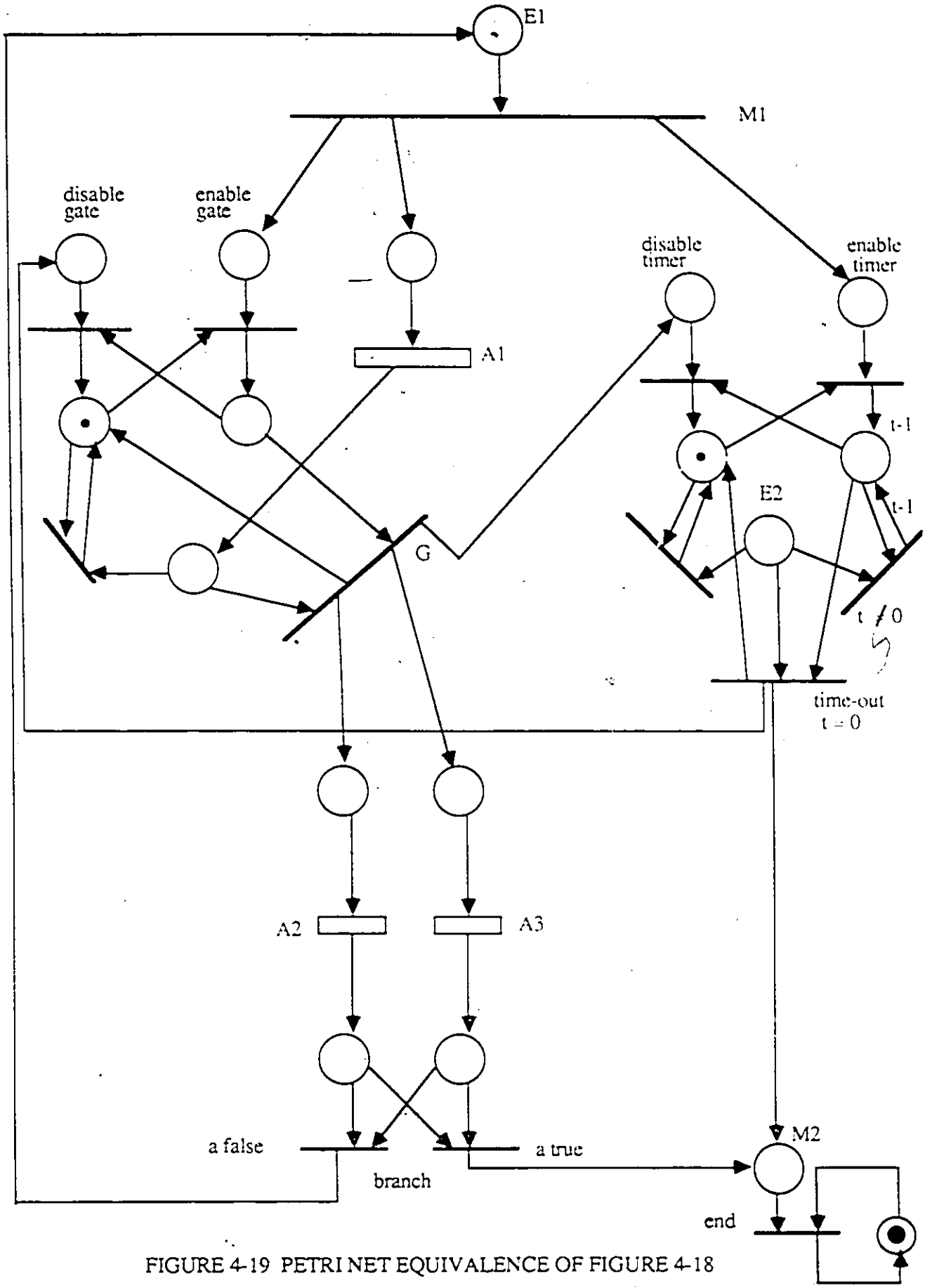


FIGURE 4-19 PETRI NET EQUIVALENCE OF FIGURE 4-18

this figure, the end node is represented by a token sink.

Comparing figures 4.18 and 4.19, it can be seen why PAD's can be considered as higher level constructs than Petri nets. The ASN represents this process in a more concise and simpler way than Petri nets can. Petri nets can better illustrate the dynamic representation of the system by showing the displacement of tokens from input to output places. A similar feature could be done in ASN's by showing which triggers have been activated by placing a token along the arc.

In this section, it was shown that every ASN node can be translated into an equivalent Petri net. By combining these equivalent representations, it is possible to map an entire ASN into a Petri net as was shown in figures 4.18 and 4.19. The behavior of an ASN can thus be modeled by Petri nets, except for one particular point. It was mentioned in section 4.2.3 that any ASN node only remembers one activation of a particular input trigger. However, with Petri nets, more than one token may be fired in one place. In a properly constructed ASN these multiple activation of an input trigger should not occur. Therefore, only a properly constructed ASN can be modeled correctly with Petri nets (using the equivalences given in this section) and the resulting Petri net must be safe (or 1-bounded). This property is important as tools developed for Petri net analysis can be used on translated ASN.

4.4.3 PAD's and Other Tools

The other tools discussed in chapter 3 are lower level tools than Petri net and data flow graphs. They deal explicitly with the concept of task while ASN's represent activities. The mapping from ASN to one of these tools is not as straightforward as in the case of Petri nets. Since a task corresponds to one or more activities, the mapping will depend on the partitioning of activities into tasks. More details on this subject will be given when the use of PAD's in the design of multitasking systems is discussed in the next chapter.

4.5 CONCLUDING REMARKS

PAD's were introduced in this chapter by describing their notation and syntax, and comparing them to some of the tools surveyed in the previous chapter. It was shown that they can be considered as extensions of data flow graphs that can also express control flow explicitly. They can also be considered as higher level Petri nets which are mainly restricted to the description of concurrent systems. This is because any ASN can be translated in an equivalent Petri net, but not all Petri nets can be represented by ASN's.

Now that PAD's have been introduced, it will be shown in the next chapter how PAD's can be used in the specification and design of a multiple microprocessor system. This will be done in the case of a multitasking environment and in the case of a

centrally controlled system which can execute the ASN's in a more direct fashion, i.e. without mapping the activities onto tasks.

-- CHAPTER 5

DESIGNING WITH PAD'S

The design of any microprocessor-based system can be broken down into a number of design steps as discussed in the second chapter. In this chapter, it will be shown how Process Activity Diagrams can be used in the various stages of the design process. The first section will describe how PAD's can be used to map the system's requirements into a functional specification of the system which can be easily understood and verified by the system designer. In section two, a specific system will be specified using PAD's to illustrate the concepts discussed in the first section. Once the system specifications have been derived, they are used to define and design the hardware and software component of the system. This will be discussed in section three with a particular emphasis on the design of the software component.

5.1 PAD'S AS A SPECIFICATION TOOL

The requirements for a microprocessor-based system may come from different sources such as marketing research or, in the case of one-of-a-kind special purpose systems, they often come directly from the end user. Some of these requirements relate to general system characteristics, such as the cost of the system, its physical size, its source of power and power consumption,

etc... Other requirement deal directly with the actions that the system must perform when interacting with its environment. These types of requirements can be mapped onto PAD's to provide a graphical specification of the system. In this case, the ASN's are used to show the required behavior of the system while the AIC's can show the timing requirements. In this section, certain aspects of PAD's which must be kept in mind during system specification will be presented.

As mentioned in the previous chapter, a system must typically respond to a number of different external events; hence, a number of ASN's, each representing one process, are required to specify the system. These processes may be totally independent, in which case, the system would always execute the same response regardless of conditions that may be present in other processes. Such systems would be fairly easy to design. Unfortunately, in most systems, processes are somehow dependent on each other. Dependencies between processes cannot be shown explicitly with ASN's unless their ASN's are combined into one. Since triggers are use to explicitly express dependencies, the nets would be connected together thus forming a single ASN. It is possible, however, to express these dependencies indirectly through external events or condition variables.

Process dependencies can be expressed indirectly using external events in the following way. An activity "i" in process "A", for instance, may consist of checking some condition in the system environment which, if satisfied, will result in an

external event which will start off process "B". Note that in this case, the starting event of process "B" would be dependent on the starting event of process "A", since it can only occur while activity "i" is executing.

Condition variables associated with branch nodes can also be used to express process dependencies indirectly. As seen in the previous chapter, condition variables affect the sequencing of the activities in a process. A process can thus affect the behavior of another by setting a condition variable appropriately. When used in this fashion, the condition variables can be considered as representing some global system status.

Another point to consider when specifying a system with PAD's is multiple occurrences of the same external event. More precisely, how should the system react if an external (starting) event occurs while the response to a previous occurrence of the same event is still being executed. In some cases, this should not happen and the second occurrence may simply be ignored. In other cases, each external event may have different data associated with it. An instance of the process must then be created and executed for each occurrence of the external event. Note that this problem must be taken into account in the design phase when choosing or designing the operating system.

To conclude this section, some features of PAD's which ease the task of specifying a system will be presented:

- PAD's are a nested construct. In a top down approach, the system designer can start with a high level view of the system,

and then refine the PAD's by expanding activities into a number of lower level activities. These low level activities must be complex enough to justify the overhead associated with scheduling the activity, yet simple enough to maximize concurrency.

- The time of the lowest level activities can be estimated and using ATC's, the minimal execution time of a particular ASN can be determined. The feasibility of meeting real-time constraints can thus be assessed at this level.

- The two graphical component of PAD's, ASN's and ATC's can be implemented as a Computer Aided Design (CAD) tool. A graphical editor can be used to enter the specification of the system using the ASN notation. The syntax of the diagrams could then be verified automatically on the computer. Errors such as missing triggers, multiple triggering (caused by the improper use of a merger node) and deadlock (when an input trigger to an activity will never be activated) could be detected.

5.2 SYSTEM SPECIFICATION EXAMPLE

To illustrate the ideas presented in the previous section, a specification example will be discussed. First, a brief description of the system will be given, then the requirements will be pointed out and finally, from these requirements, the system specifications will be derived.

5.2.1 System Description

The system that will be specified in this section is an industrial control system for a conveyer which is capable of automatically sorting boxes to a number of destination conveyers (figure 5.1). This example is loosely based on an actual system briefly outlined in [PAWL 84]. Boxes would be manually loaded on a number of feeder conveyers which are connected to an accumulator conveyer. The accumulator conveyer has rollers which are free wheeling allowing the boxes to roll on the conveyer close to each other, but the rollers may also be engaged forcing the boxes to move. This mechanism is used to regulate to flow of boxes to a distribution conveyer which is connected to the end of the accumulator conveyer. From the distribution conveyer, the boxes would be automatically routed to their proper destination conveyer. All the destination conveyers are connected to the shipping conveyer which is used to send the sorted boxes to the loading docks. To simplify the example, only the control of the accumulator, distribution and destination conveyers will be considered.

5.2.2 System Requirements

The system requirements (which are adapted from [PAWL 84]) are as follows:

- Boxes fed to the distribution conveyer should be spaced far apart to prevent jamming and errors in the sorting.

- A bar code reader must be able to read the product code and

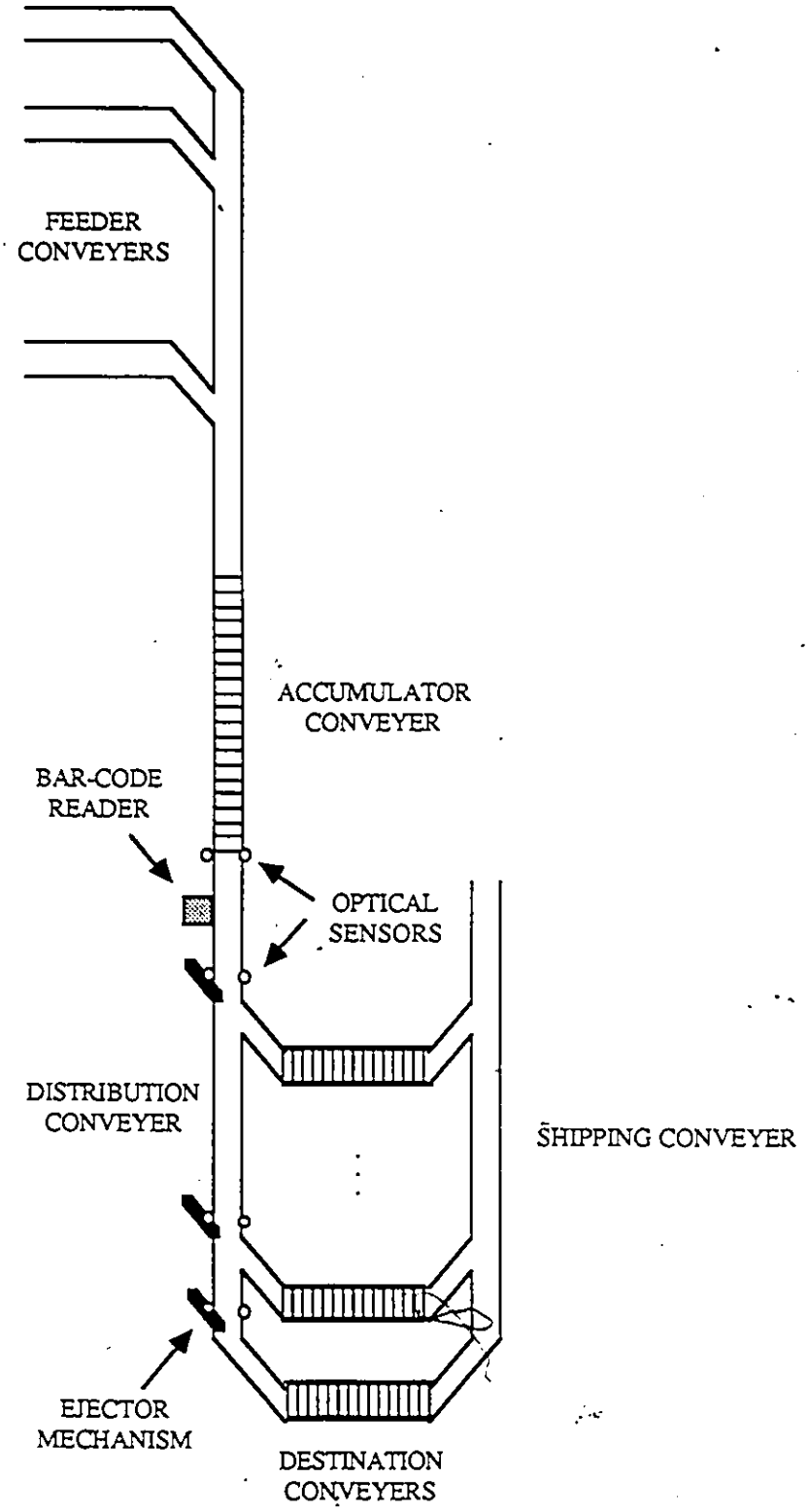


FIGURE 5-1 AUTOMATIC SORTING CONVEYER

destination of the box.

- The boxes must be sorted correctly onto the destination conveyers.

- Simple optical sensors are used to check for the presence of a box at the various connecting points between the distribution and destination conveyers. There is also a mechanism to eject the box onto the destination conveyers at these points.

- If a box is jammed in the system or a sorting error occurs, the conveyer must be stopped and an alarm activated to request the assistance of a human operator.

- A minor alarm must be reported when a box is received in damaged condition at the destination conveyer. This condition would be detected by some special sensor located on each destination conveyer.

- All alarms are to be recorded on a printer.

- Statistics must be kept on: products taken out of the warehouse, number of boxes routed correctly in the system, number of damaged boxes, down time of the conveyer and number of boxes in the system at any time.

5.2.3 System Specification

The first step in the specification of a system is to identify the major processes which make up the system. By examining our system's requirements, the following processes

would be needed:

- "Box Spacer" Process: this process would be responsible for making sure there is enough space between boxes on the distribution conveyer.

- "Product & Destination Determination" Process: this process would take care of reading the product and destination bar label on the box, updating the proper databases and setting up some kind of routing table so that the box is sent to the proper destination conveyer.

- "Box Ejection" Process: when a box arrives at a connecting point between the distribution conveyer and a destination conveyer, this process would be required to decide (based on the routing table) what should be done with the box; to let it go by or eject it on the destination conveyer.

- "Distribution Conveyer Monitor" Process: this process would be responsible for monitoring the distribution conveyer to make sure that no boxes are jammed or sorted incorrectly. There is no sensor in the system that can precisely detect if a box is jammed or sorted incorrectly. Since the speed of the distribution conveyer is known to the control system, it can be estimated fairly accurately when a box should arrive at each destination conveyer (i.e. at each optical sensor). Hence a box will be assumed jammed or wrongfully sorted if it does not reach the optical sensor in time.

- "Damaged Box Report" Process: this process would report a minor

alarm when a box is received in a damaged condition at a destination conveyer.

This set of processes represents only one of the possible partitioning of the various functions of the system into processes. These specific processes were chosen because they follow naturally from the requirements of the system. The next step is to specify these processes using ASN's. This requires defining the starting event of each process and the sequence of activities which must be executed in each case. As with any design tool, a number of iterations are required to properly specify a system with ASN's. These iterations will not be discussed here, as they would only complicate the presentation; only the final specification will be described, one process at a time.

The first process considered is the "box spacer" process which is really responsible for controlling the accumulator conveyer at the point where it meets the distribution conveyer. It must be continuously running whenever the distribution conveyer is operating to maintain spacing between the boxes. It is thus started when the distribution conveyer is started and must be "killed" when the conveyer is stopped. The ASN corresponding to this process is shown in figure 5.2.

The control of the spacing between the boxes is basically done by engaging or not the rollers on the accumulator conveyer. As can be seen in the ASN, when the system is started, the accumulator conveyer is engaged. When a box arrives at the end of

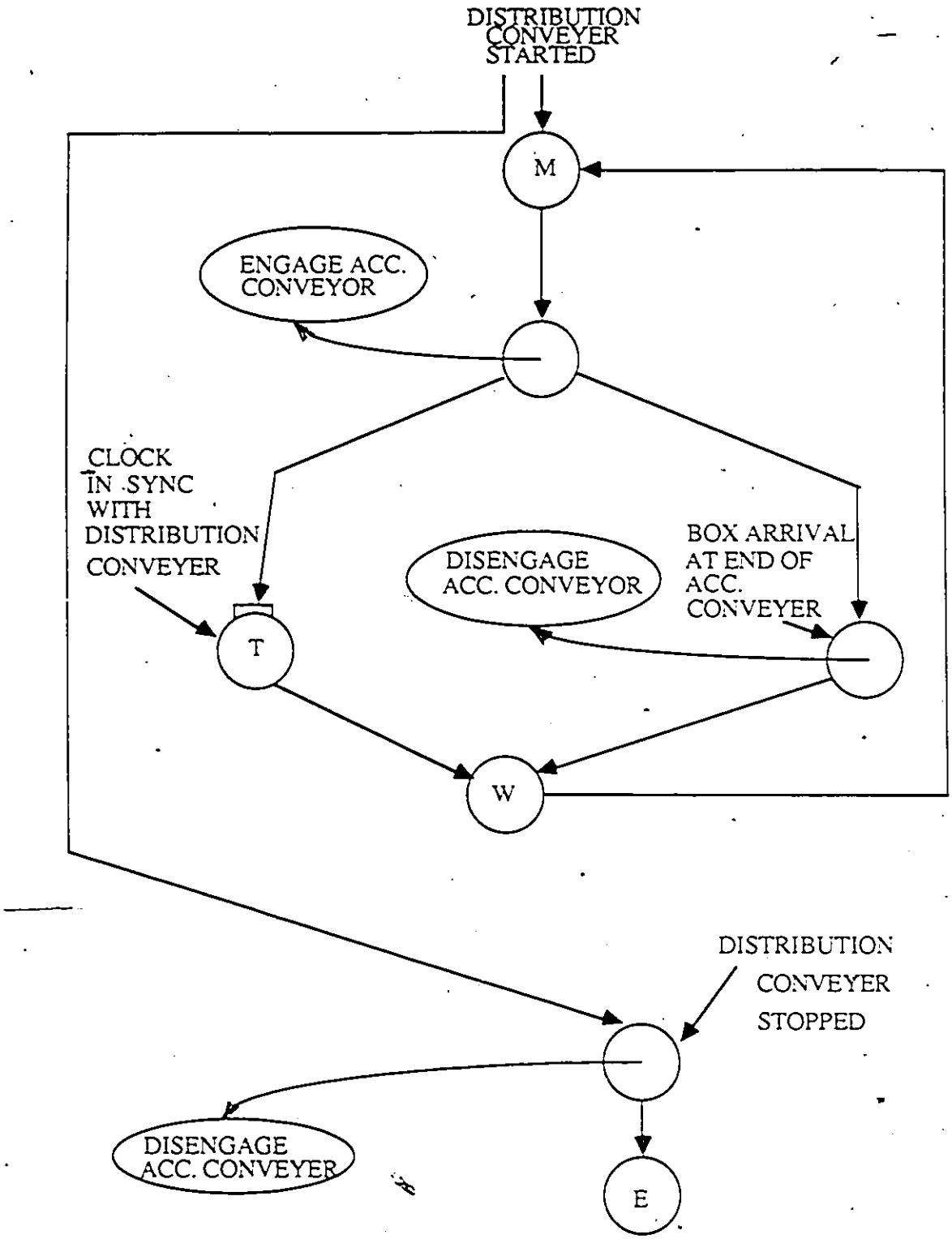


FIGURE 5-2 BOX SPACER PROCESS

the accumulator conveyer, it is detected by the optical sensor, triggering an external event which disengages the accumulator conveyer. A timer which was started after the conveyer was engaged is used to make sure that the box is not sent to the distribution conveyer (i.e. the accumulator conveyer is not turned back on) before the proper spacing is observed between boxes. This is the reason for the wait node before the feedback path. Note that the external clock used by the timer must be synchronized with the speed of the distribution conveyer to ensure the proper translation of space between boxes into time between boxes. This process is ended when the distribution conveyer is stopped. This is illustrated by an external event which will make sure that the accumulator conveyer is disengaged before the process is terminated. Remember that when a process is terminated (i.e. the end node is reached), all the activities being executed for this process are aborted.

The next process, the "product & destination determination" process only needs to be carried out when a box passes by the bar code reader. An optical sensor can be used to detect when the box reaches the bar code reader or it can be assumed that the reader can detect this by itself. In either case, this would cause the external event that starts this process. The first activity (figure 5.3) which must then be executed is to read the bar code to find out which product is being taken out of the warehouse and its destination. Once this is established, three activities may be started concurrently to update the inventory database and the statistics of the conveyer process, and to setup information

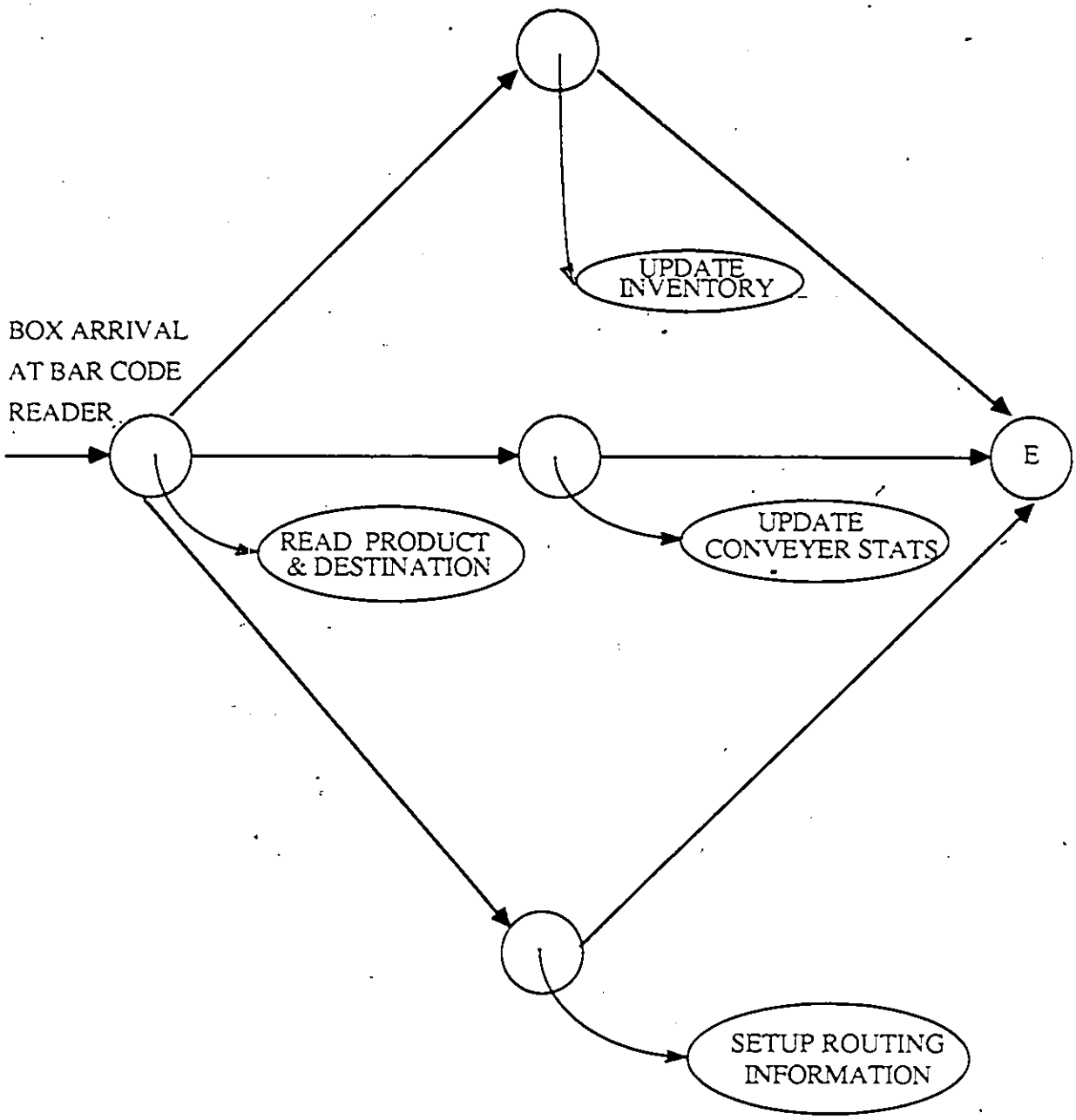


FIGURE 5-3 PRODUCT AND DESTINATION DETERMINATION PROCESS

which will be used to route the box to its proper destination. Note that a serialization can be forced upon these three processes (i.e. executing them in sequence rather than in parallel) but if only data dependencies are considered, there are no reasons why they cannot be run in parallel. These activities may or may not be executed in parallel in the final system; this will depend on implementation details, but at the specification level, all potential concurrencies should be shown in the ASN's to provide for the best possible response time in the design phase.

Once a box has passed the bar code reader, the distribution conveyer will carry it along until it is ejected on one of the destination conveyers. Just before an intersection of a destination and distribution conveyer, an optical sensor is used to alert the system that a box has arrived. This will start off the "box ejection" process (figure 5.4).

At this point, a simple decision must be made: should the box be ejected from the distribution conveyer onto the destination conveyer or should it be left alone because this is not its final destination. The decision cannot be made by reading the bar code once more since there is no bar code reader at this location. One solution is to use "routing" queues which must be updated in the "product & destination determination" process. This scheme would operate as follows: each destination conveyer would have a queue associated with it. Each element in the queue would correspond to a box that should go by this destination conveyer and would

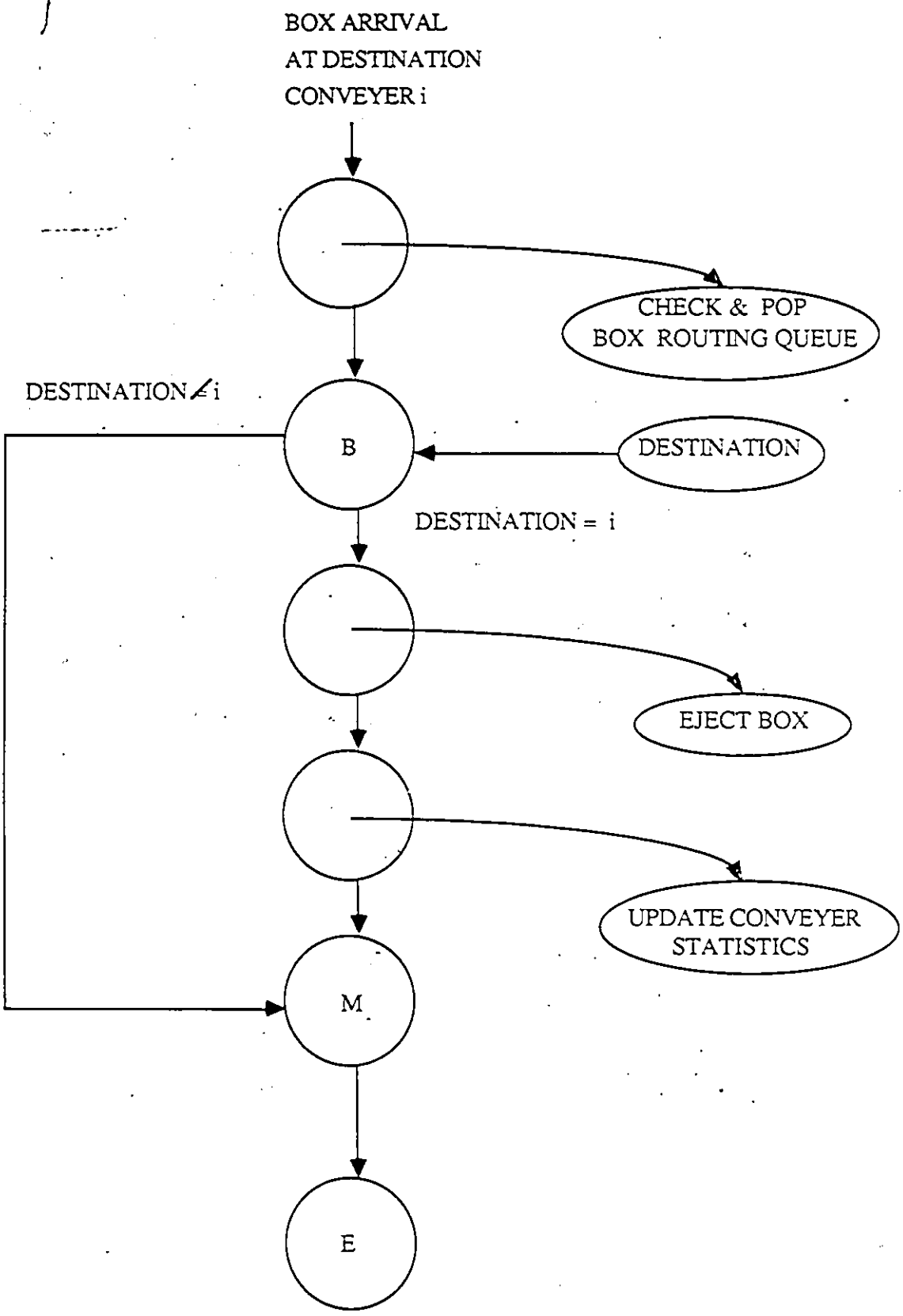


FIGURE 5-4 BOX EJECTION PROCESS

contain two fields: one indicating if the box must be ejected or not and the other indicating the time of arrival. Hence the first activity in the "box ejection process" is to verify the first element in the queue to find out what to do with the box. If the box needs to be ejected, an activity to do so will be executed and the conveyer statistics will be updated (one more box arrived at its destination).

The activity "setup route information" in the "product & destination determination" process is responsible for adding the proper elements to the "routing" queues of the various destination conveyer. If the destination of the box is the conveyer i, then an entry will be added to the queues of conveyers 1 to i-1 indicating the time the box will go by that particular conveyer and that the box must be left alone. An entry will also be added to conveyer's i queue indicating the arrival time of the box and that the box must be ejected at this point. All the other queues will not be modified since the box will not go by these conveyers.

Note that a "box ejection" process must be started each time a box reaches any of the destination conveyer. Since it can be assumed that only on box may be at one of these location at a time and that the process can be completed before the next box comes along (otherwise it would not be ejected in time), a specific copy of this process can be made for each destination conveyer. In this case, each optical sensor would generate a different external event which would start the process associated with that particular destination conveyer. Another solution is to

create an instance of this process each time a box goes by any one of the destination conveyer. This would be the case of multiple starting event with different input data; the input data would simply be the ID of the destination conveyer. The decision of which scheme to select depends on the features of the operating system used. Hence, this will only be done in the design phase.

Another requirement which must be met in our system is to detect if a box is jammed or incorrectly sorted. Since there are no sensors that can actually identify this situation, a monitoring process must be defined to constantly check for these conditions. The "distribution conveyer monitor" process (figure 5.5) will serve this purpose and will also be responsible for the operation of the distribution conveyer. The starting event for this process is thus the start-up of the system. Its first activity will be to start the distribution conveyer. It will then be responsible for generating the external clock in synchronization with the conveyer speed. Note that this is the case of an activity generating an external event which will be used in another process.

The monitoring portion of this process is done in between the update to the clock. At this time, the process will check all "routing" queues to make sure that no boxes are late or early. If a box is late, it will be assumed that it is jammed or incorrectly sorted (ejected into a previous destination conveyer). If a box is early, then it can be assumed that this

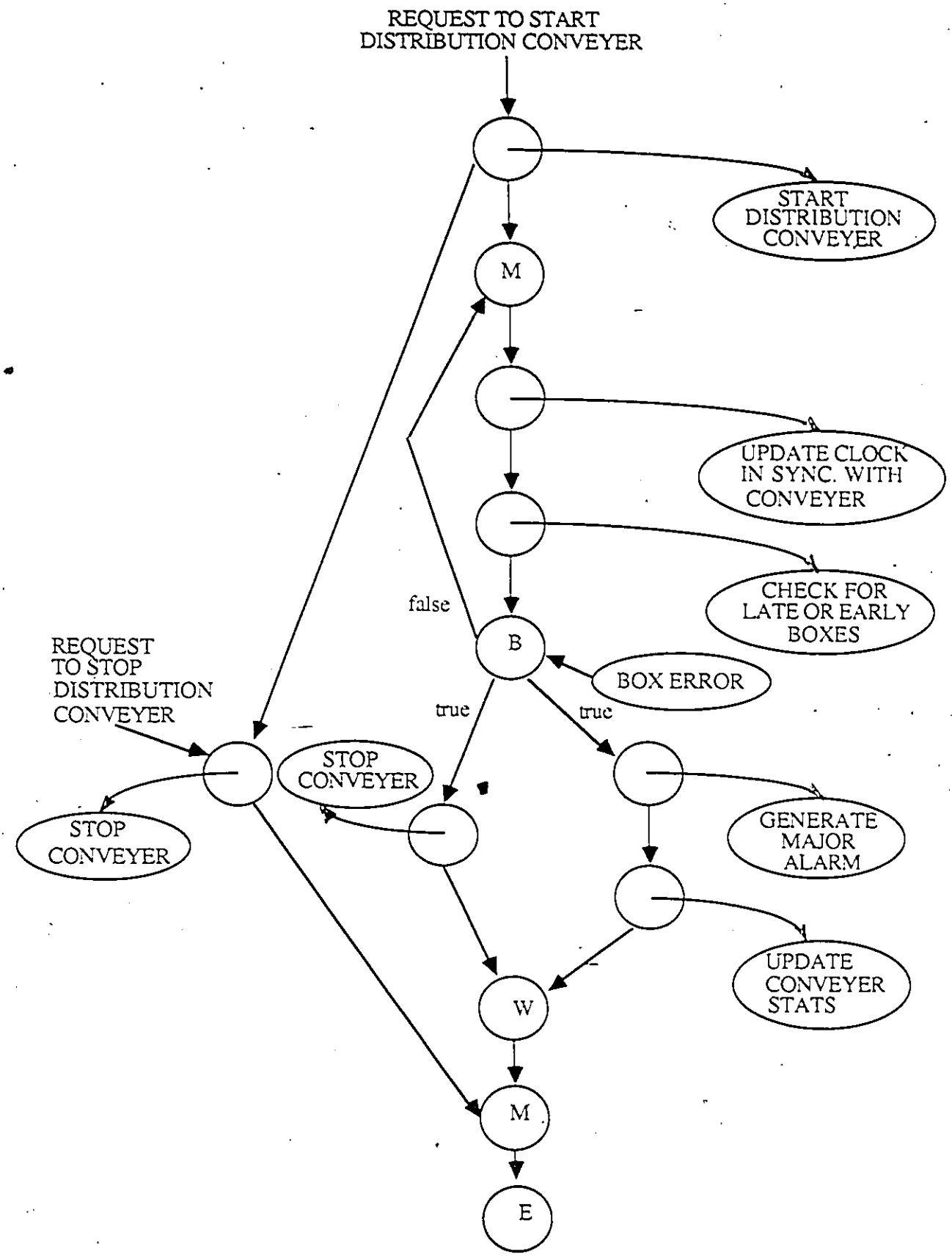


FIGURE 5-5 MONITORING PROCESS

box should not have arrived at this conveyer, hence the previous conveyer did not eject it and it will be incorrectly sorted. In theses cases, a major alarm will be reported and the conveyer will be stopped, ending the process. The alarm should be audible since human intervention is required before the conveyer system may be started again.

The last process in this system, the "damaged box report" process is the simplest one (figure 5.6). Whenever a box is detected to be damaged by some advanced sensor, a minor alarm is reported on a printer (and perhaps on some light indicator panel) and the statistics for damaged boxes are updated.

The entire automatic routing conveyer system would require more processes than the ones just described. For instance, man-machine interface processes are required to check the system status, printout statistics, etc... But this simple set of processes was shown to illustrate how Activity Sequence Nets can be used to specify an actual system.

5.3 PAD'S AS A DESIGN TOOL

Once a system has been specified using PAD's, it is possible to use this notation to carry out the specification into an implementation. This involves partitioning the system into hardware and software components and designing these components. In a multiple processor system, it must also be decided how many and what type of processors to use, how they should be

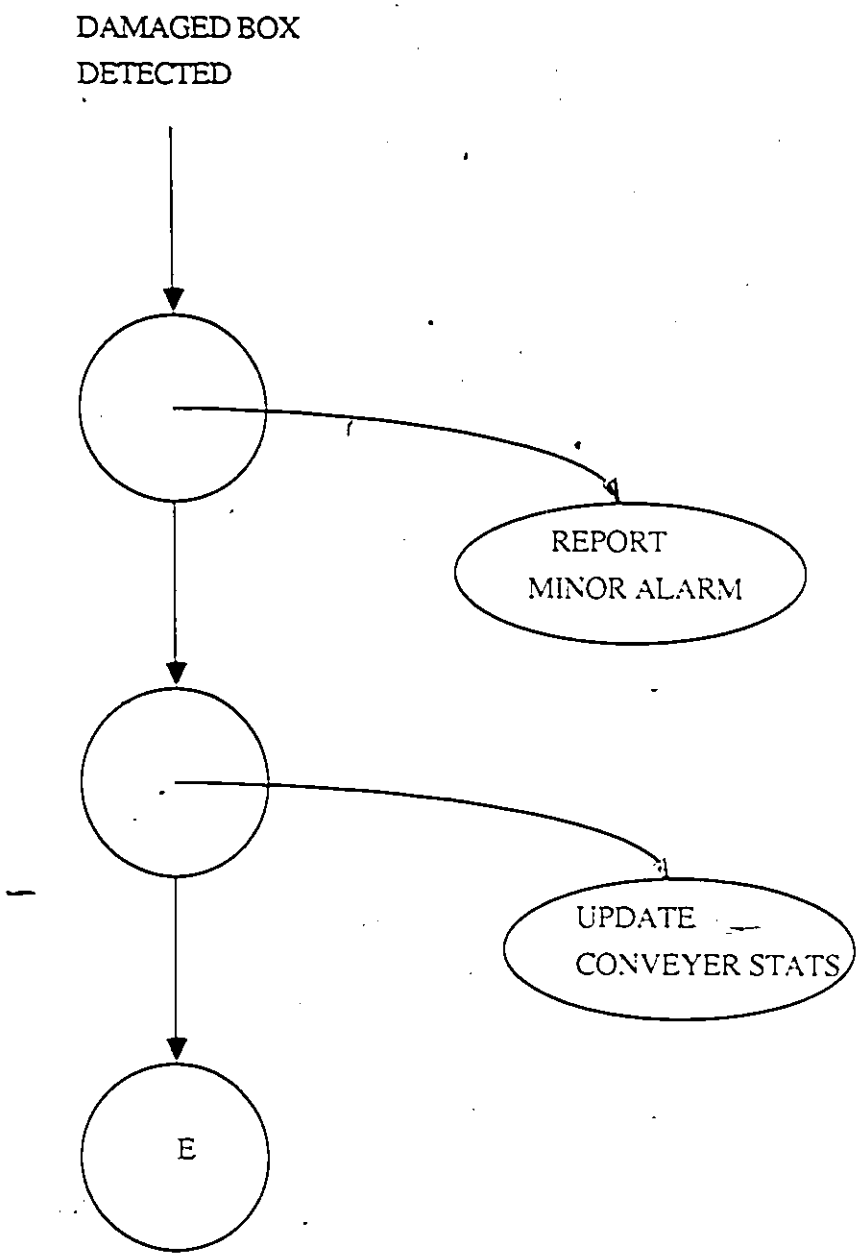


FIGURE 5-6. DAMAGED BOX REPORT PROCESS

interconnected and how the software component is to be partitioned between the various processors in the system. The answer to these questions depend on system requirements such as cost, physical size, real-time constraints, etc... PAD's can help make these decisions, but the method to use depends on the type of environment that the system will be implemented in.

In this section, two different design approaches will be presented. In the first one, the "classic" approach to real-time system design will be taken by using a real-time multitasking operating system. In this case, the schedulable entity in the system is the task. Since PAD's do not deal explicitly with the concept of tasks, it will be shown how they can help the system designer define the tasks in such a way as to minimize certain overhead associated with real-time multitasking operating systems. The second approach that will be presented is more adapted to PAD's, since instead of working around tasks, a system executive that can directly schedule activities will be used. These design environments will be referred to as multitasking and multi-activity respectively.

5.3.1 Design in a Multitasking Environment

The major difference between a task and an activity is that a task may decide to stop and wait for additional data before going on with its execution. This is illustrated by the wait state in figure 2.7. An activity, on the other hand, cannot wait for additional data once it is started. An activity can thus be considered as a simple task which waits for its input data, does

its computations, send back its results and goes back to waiting for its input data, ready for the next time it is required. However, to reduce the inter-task communication requirements, activities can be grouped together into tasks (activities that are grouped together in the same task can share data directly without the overhead of inter-task communication).

PAD's can help the system designer choose which activities to group together. Using the simple example of figure 4.2, for instance, we could group activities Ai, Aj and An together to form task A and activity Ak could form task B (as shown in figure 5.7). Using an operating system with message passing and a non-blocking send the pseudo-code for tasks A and B would look something like this:

```
TASK A

DO FOREVER
   BEGIN
      WAIT for E1
      execute activity Ai
      SEND data to TASK B
      execute activity Aj
      WAIT for result from TASK B
      execute activity An
   END
```

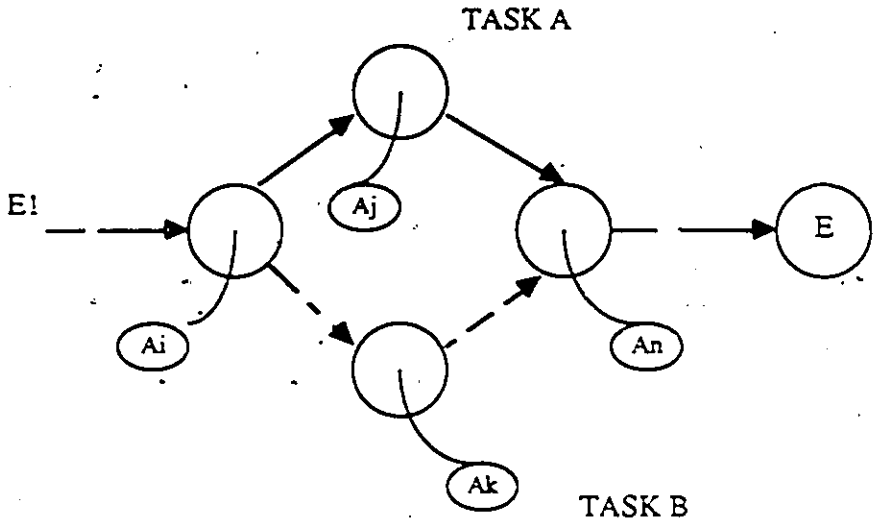


FIGURE 5-7 GROUPING ACTIVITIES INTO TASKS

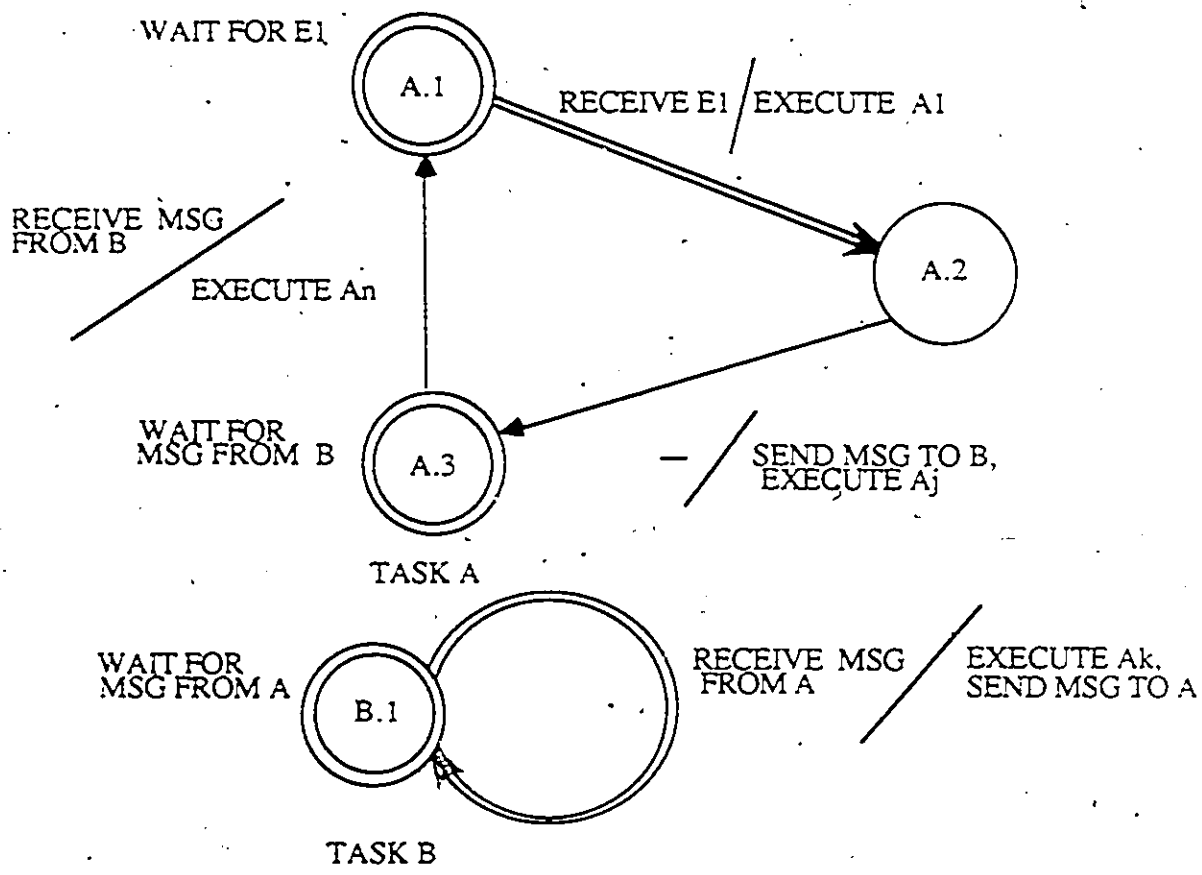


FIGURE 5-8 COMMUNICATING FINITE STATE MACHINE OF TASKS A AND B

```
TASK B
DO FOREVER
   BEGIN
     WAIT for data from TASK A
       execute activity Ak
       SEND result back to TASK A
   END
```

The internal of tasks A and B can also be represented using the communicating finite state machine notation discussed in chapter three. The resulting representation is shown in figure 5.8.

The grouping of activities into tasks in this example was not totally arbitrary. For instance tasks Aj and Ak were deliberately assigned to different tasks to maximize concurrency. Activities Ai, Aj and Al were probably grouped together because they needed to share more information between themselves than with activity Ak. Hence, when grouping activities into tasks, the following points must be considered:

- By mapping concurrent activities to different tasks, the response time of the system can be increased. This is especially true in the case of a multiple processor system if the concurrent tasks are assigned to different processors.

- When an arc in the ASN is cut to group activities together, some form of inter-task communication is required to maintain synchronization between these activities and possibly to transfer data. Thus by cutting arcs with minimal communication

requirements, the communication overhead can be reduced.

Once the activities have been grouped into tasks, the tools available for the design of multitasking systems can be used to verify if the tasks chosen do not present any problems such as deadlock. For instance, reachability trees can be used in the case of communicating finite state machine. If need be, the partitioning of activities into tasks can be reiterated based on this new information and the ASN's.

5.3.2 Design in a Multi-Activity Environment

If the system as been specified using PAD's, the design process could be greatly simplified if the system could directly "execute" the ASN's. This would require a system executive which is capable of coordinating the execution of the various activities that make up a system by following the sequencing relations expressed by the ASN's. In this case, the entity that is scheduled to run by the system executive is the activity itself, not a task. To illustrate this, a state diagram of the activity from the system executive point of view is shown in figure 5.9. Note that the only major difference with the task state diagram (figure 2.7) is the absence of a wait state.

There is a major difference between a multitasking and a multi-activity executive, apart from the fact that one schedules tasks and the other activities. A multitasking operating system has no knowledge of the application, it simply schedules tasks according to the rules of the inter-task communication and

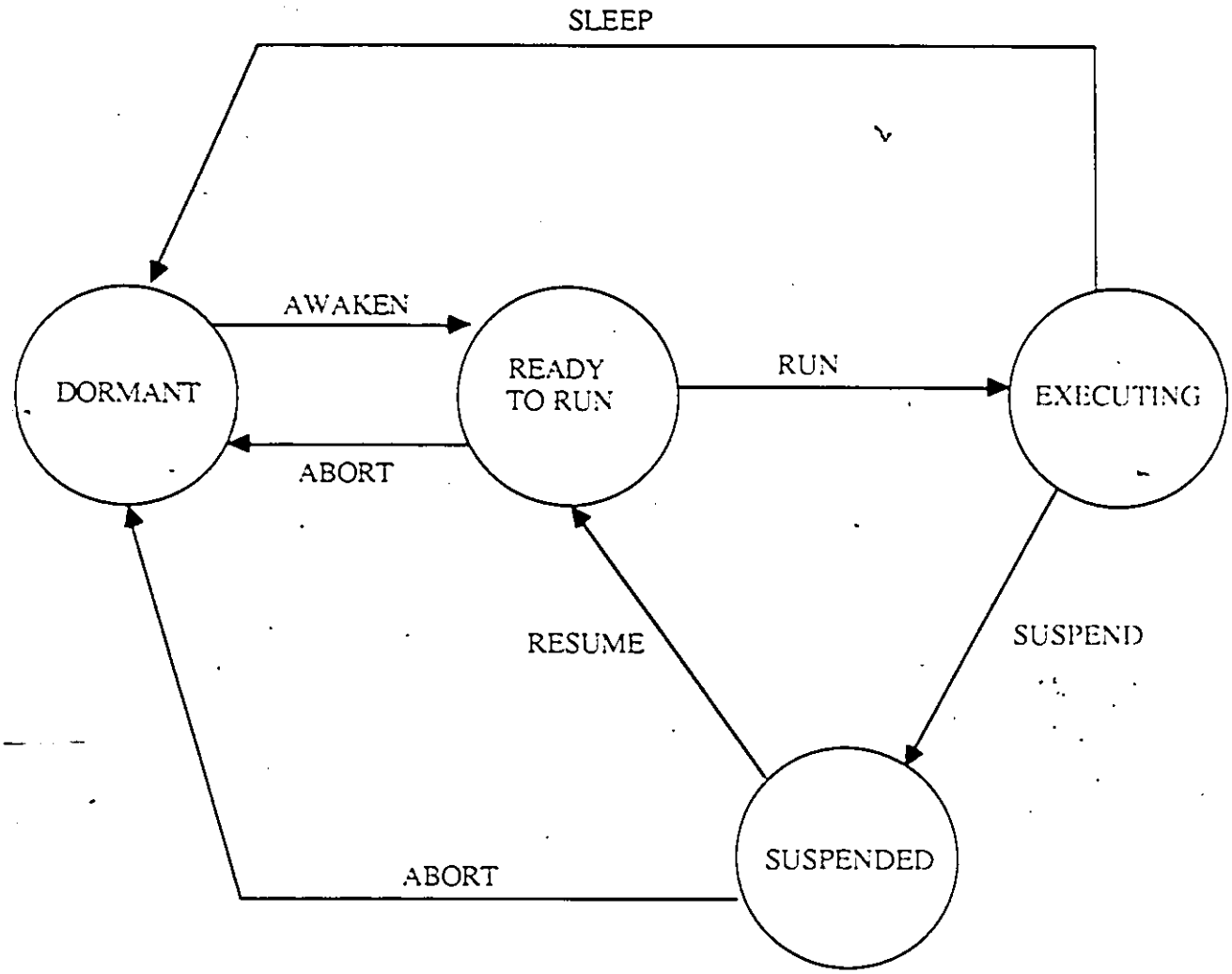


FIGURE 5-9 GENERIC ACTIVITY DIAGRAM

synchronization mechanisms. The information specified by the ASN's, i.e. which activity runs before or concurrently with which other is completely hidden in the inter-task communication scheme. In the case of a multi-activity executive, the executive must have knowledge of these ASN's to be able to schedule the activities properly. Thus the multi-activity executive is dependent on the application as opposed to the multitasking operating system. However, it will be shown in the next chapter, how this dependency can be reduced by using a standard matrix format to represent the ASN's and having the executive simply use these matrices as input data, saving the designer the trouble of rewriting an executive for each different application.

There are some advantages when the executive has knowledge of the sequencing relationships between the activities. For instance, since it knows which activities will need to be scheduled next, it can assign the activities to processors in such a way as to meet real-time constraint in a dynamic fashion. If a response to an external event (an ASN) is running slow and might not meet its deadline, the activities in the process could be dynamically given a higher priority to ensure that the response will be carried out in the required time. The executive could also make sure that all activities are executed within a certain time so that if a processor fails, the activity can be reassigned. Another important feature of such an executive, in a multiple processor design, is that the system could be easily simulated before implementation to make sure that the activity to processor allocation chosen can meet real-time constraints. The

simulation of such an executive will be discussed in more detail in the next chapter.

5.4 CONCLUDING REMARKS

In this chapter, the use of PAD's in the design process of a microprocessor-based system was presented. It must be emphasized once more that PAD's were designed as a tool for non-computer specialists. Such systems as the automatic routing conveyer and other industrial control processes are often designed by mechanical and industrial engineers [MACQ 84]. Other typical systems that could be designed with PAD's are chemical process controllers (distillation columns in refineries, etc...) which would be typically designed by chemical engineers. All these professionals are not computer specialists, they cannot be expected to design systems operating in a multitasking environment. Presently to design these systems, they use tools such as primitive relay ladders (to program the PLC's, the programmable logic controllers which they use to replace relay systems) and flowcharts, and program mostly in BASIC using personal computers [BELL 84]. To design more complex real-time systems, they need simple tools which are highly automated (CAD type tools). This is not to say that PAD's cannot be used by computer specialists in a multitasking environment, as it was shown that they can be helpful in partitioning activities into tasks. Note that a number of ideas which apply to task allocation on a distributed processing system [CHU 80] can be used to map

activities onto tasks, since the two problems have similar features.

In order to carry out the entire design process using PAD's, from specification to implementation, the following aids are needed: to specify the system, a graphical editor is needed to enter the ASN's and translate them in a form which can be easily manipulated by the computer. Then using these ASN's with their corresponding ATC's, a "feasibility check" program could be used to verify if it is possible to meet real-time constraints (assuming infinite number of processors), check for bottlenecks, and get an initial activity to processor allocation. To verify the design before implementation, it would be very useful to simulate the system under development. The simulation is dependent on the executive chosen. In the next chapter, a multi-activity executive for a centrally controlled system will be discussed along with a simulation program based on this executive.

CHAPTER 6

SYSTEM EXECUTIVE AND SIMULATOR

In this chapter, an implementation scheme for a multi-activity executive for a multiple microprocessor-based system will be proposed. The design of the executive is based on a centrally controlled system which will be presented in the first section. A description of the multi-activity executive will follow in section two. As mentioned in the previous chapter, it would be useful to simulate the system before implementation. Section three will discuss a simulation program which can be used to verify the performance of a centrally controlled system designed with PAD's. Finally, section four will discuss some of the performance measures computed by the simulator.

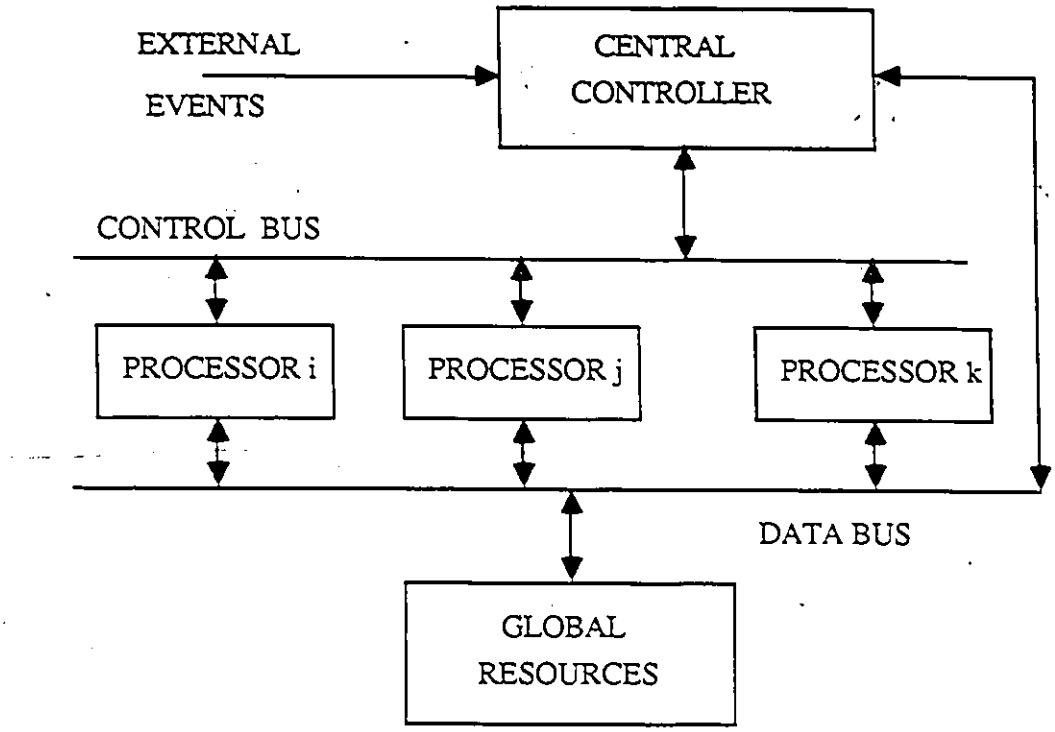
6.1 CENTRALLY CONTROLLED SYSTEM ARCHITECTURE

Many different multiple microprocessor architectures could be chosen as the underlying hardware for the implementation of the multi-activity executive, but to make a meaningful selection, the characteristics of the system must first be examined. One of the first points that should be noted, is that the system can be considered as being made up of two logical levels. The first one is the system executive which is concerned with the coordination of the various activities in response to external events, while

the second level is concerned with the execution of the activities themselves. Another characteristic of this system, as pointed out in the previous chapter, is that the executive must know the sequencing of activities (i.e. the ASN's).

From these characteristics, the following architecture can be suggested. Since there is a logical separation between the executive and the execution of the activities, why not make this separation physical. This can be done by assigning one or more processors the responsibility of coordinating the activities and assigning to one or more processors the execution of these activities. Since the executive must have global knowledge of the ASN's, a centrally controlled system will be chosen for simplicity [KRIE 81] [FATH 83]. The resulting architecture is depicted in figure 6.1.

In this scheme, either a single processor or a multiprocessor system is used as the central controller which is responsible for the coordination of the various ASN's. A number of subordinate processor modules are responsible for the execution of the activities. Each module consists of a microprocessor, ROM to store the code of the activities, RAM for local data storage and I/O ports to communicate with the environment. Note that, in certain cases, these modules could simply consist of a 4 or 8 bit single-chip microcomputer or microcontroller. The central controller would communicate with the subordinate processors via a control bus. To request the execution of an activity, the central controller would simply send the activity identifier to



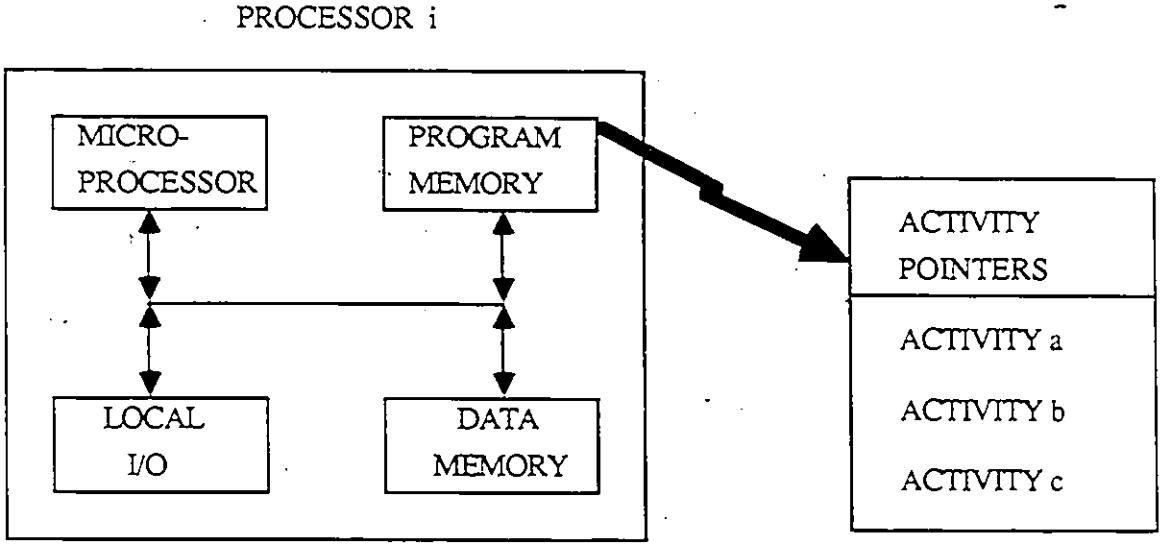


FIGURE 6-1 CENTRALLY CONTROLLED ARCHITECTURE

the appropriate subordinate processor on the control bus. The subordinate controller would then find the starting address of the activity in a table and jump to this location. There is no need to download the code of the activity to the subordinate processor because it is stored in its local ROM.

Whenever a centrally controlled architecture is mentioned, the system designer is faced with two potential problems: reliability and communication bottleneck between the central processor and its subordinates. The reliability problem arises because the entire system rests on the central controller; if it fails the system fails. To solve this problem, in a system where reliability is important, the central processor could have a hot standby ready to take over in case of failure. Alternatively, one of the subordinate processor modules could be a cold standby to save on hardware cost. On the other hand, the communication bottleneck problem is dependent on the data traffic between the central controller and the subordinate processors. To avoid this problem, the communication between these units should be kept to a minimum. This is the approach that will be taken in this design. If the system requires much inter-processor communications, a separate data bus connected to a shared memory (shown in figure 6.1) could be used to share information between processors.

6.2 MULTI-ACTIVITY EXECUTIVE

Now that the hardware architecture of the system has been

outlined, the multi-activity executive can be designed. In the same way as a system is partitioned into processes which are broken down into sub-processes and so on, the executive will be broken down into modules to facilitate the design process. The three main modules that were chosen are the system manager, the ASN managers and the activity scheduler [KRIE 86]. Each one of these modules have well defined responsibilities and they work together to provide all the services of the multi-activity executive. Note that in this conceptual view of the executive, it is assumed that these three modules are being executed concurrently. Their interrelations are shown graphically in figure 6.2.

The system manager has general knowledge of the state of the system: it is informed of occurrences of external events, it is responsible for initiating ASN's (processes) in response to these external events, and it also knows which ASN's are "awakened" (executing) and which are "dormant". The system manager, however, is not responsible for coordinating the activities of the ASN; this is the role of the ASN managers. For each awakened process in the system, there is an ASN manager responsible for finding out which activities are ready to run. The scheduling of these activities is not the responsibility of the individual ASN managers since it requires knowledge about the state of all the various processors in the system. A module that can centralize this information is easier and safer to implement than it is to devise a mechanism to share this information among all the ASN managers. This is the function of the activity scheduler which

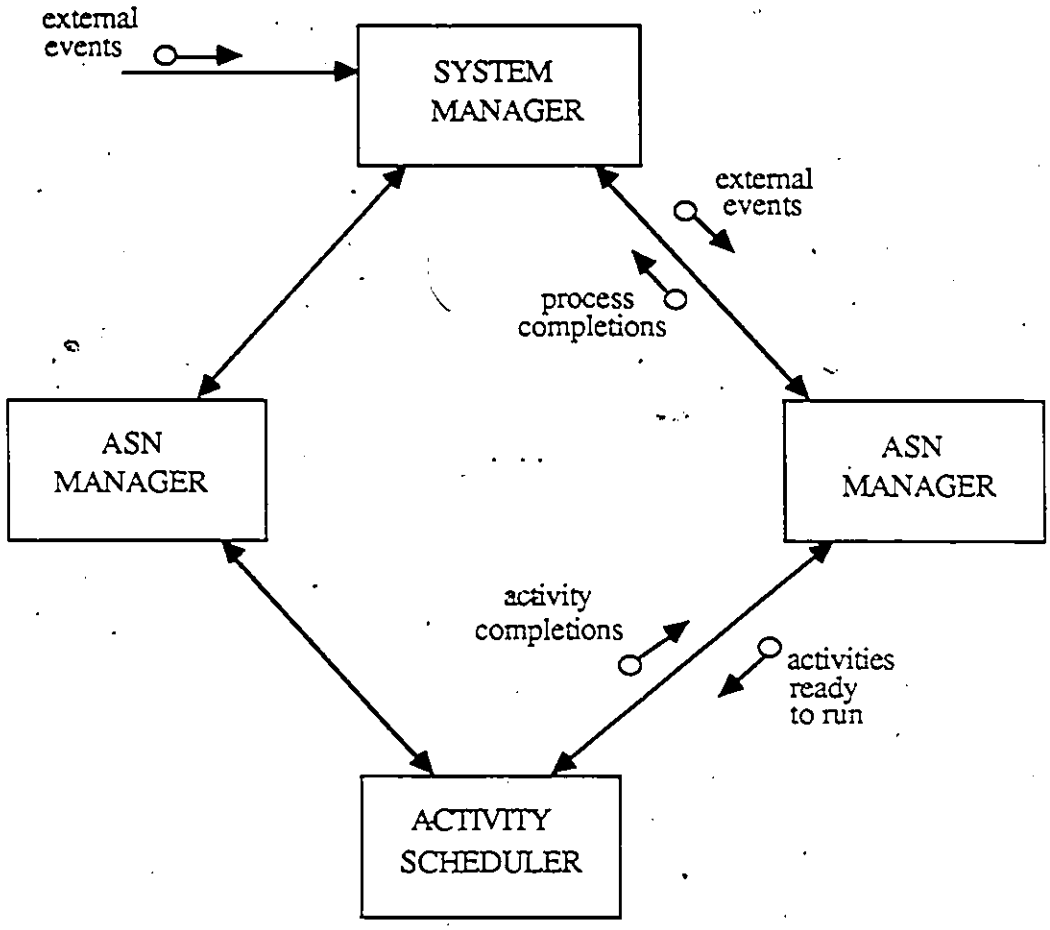


FIGURE 6-2 MULTI-ACTIVITY EXECUTIVE MODULES

will assign (when possible) a processor to execute the various activities in the ready to run activity queue. Each of these modules will now be presented in more detail.

6.2.1 The System Manager

The system manager module receives two different types of information: external events from activities or interrupt routines, and process completions from the various ASN managers. Using this information, the system manager will keep track of the state of all the processes in the system. It has a table indicating which external event is the starting event for all the ASN's and when such an event occurs; it will "create" an ASN manager to take care of this process. It will also inform any awakened ASN manager of the occurrence of any external events that are relevant to the process it monitors. Whenever a process completion is received from an ASN manager, it is the responsibility of the system manager to "kill" this ASN manager. Note that in the case where a starting event for a process occurs while the ASN for a previous occurrence is still awakened (situation discussed in the previous chapter), it is up to the system manager to decide if the event is to be ignored or if a new instance of an ASN manager must be created to respond to this event.

6.2.2 The ASN Managers

Conceptually, there is an ASN manager for each awakened ASN,

which is dynamically "created" and "killed" by the system manager because it is only needed when the ASN needs to be executed. In the actual implementation, as will be seen in the next section, there may be a single ASN manager taking care of all the awakened ASN's.

An ASN manager would receive as input external events from the system manager and activity completions from the activity scheduler. Using this information, the ASN manager can inform the activity scheduler of which activities must be executed as governed by the sequencing relations in the ASN. To avoid writing a separate ASN manager program for each different ASN, the sequencing relations expressed by the ASN should not be explicitly programmed in the ASN manager. Instead, a general ASN manager program which would take the ASN as input data will be implemented. In this case, the information contained in the ASN must be translated in a form easily manipulated by the ASN manager program. The representation chosen is a matrix which is illustrated in table 6.1. This example shows the ASN matrix corresponding to the ASN in figure 6.3. It was chosen because it has all the ASN nodes except for the wait node.

The columns of this matrix represent the various nodes of the network, one column for each node. The rows represent the events associated with this network, both internal and external events. There are four possible entries in the ASN matrix. If event i is an input trigger to node j, then the entry in position (i,j) would be "I" for Input trigger. If event i is an enable input to node j (a gate or timer/counter node) then the entry in position

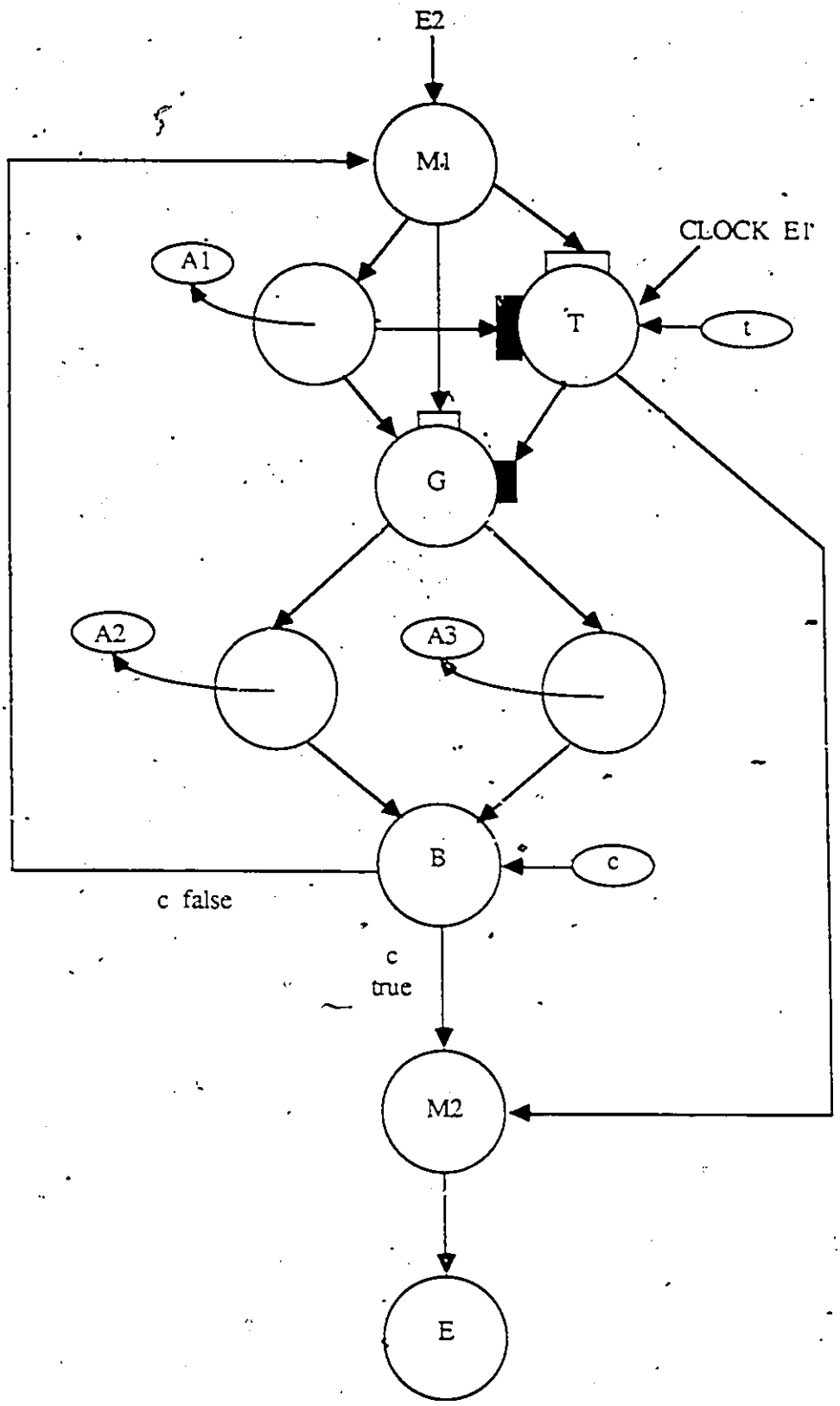


FIGURE 6-3 ASN WITH ALL BASIC NODES

ASN MATRIX

	M1	A1	T	G	A2	A3	B	M2	E
M1	O	I	E	E	O	O	O	O	O
A1	O	O	D	I	O	O	O	O	O
T	O	O	O	D	O	O	O	I	O
G	O	O	O	O	I	I	O	O	O
A2	O	O	O	O	O	O	I	O	O
A3	O	O	O	O	O	O	I	O	O
B (C T)	O	O	O	O	O	O	O	I	O
B (C F)	I	O	O	O	O	O	O	O	O
M2	O	O	O	O	O	O	O	O	I
E1	I	O	O	O	O	O	O	O	O
E2	O	O	I	O	O	O	O	O	O

Note: B (C T) means branch with condition C true.
B (C F) means branch with condition C false.

TABLE 6.1 ASN MATRIX CORRESPONDING TO ASN OF FIGURE 6.3.

(i,j) would be "E". In the same way if it is a disable input, then the entry would be "D". If event i and node j are not connected, then a "O" would be entered in position (i,j).

The algorithm used by the ASN managers to properly schedule the activities according to the ASN matrix can be briefly described as follow: whenever an internal or external event occurs, an event vector is generated for that ASN. The event vector has the same format as a column of the ASN matrix. It will have a "1" in position i if event i occurred, a "O" otherwise. Each active ASN manager will have a state matrix which will have exactly the same format as the ASN matrix, but it will be initialized to "O" in all entries. The purpose of this state matrix is to remember the state of the ASN, i.e. the past triggers of all nodes, except the merge node which is memoryless.

The state matrix is updated every time the event vector changes. This is simply accomplished by performing the following operation on each element of the state matrix S:

$$S_{ij} = (E_i \text{ AND } A_{ij}) \text{ OR } S_{ij}$$

where E is the event vector and A is the ASN matrix. Note that all non-zero entries in the ASN matrix are considered to be one's.

The state matrix is then verified, column by column (i.e. node by node) to see what action should be taken. The procedure to check the column is different depending on the type of node:

- In the case of the activity initiator node, the activity associated with the node will be placed in the ready-to-run queue if all its input triggers have been activated.

- In the case of the end node, if all its input triggers have been activated, the system manager will be advised so that it can "kill" this ASN manager.

- In the case of the wait node, a new event vector indicating that the output trigger(s) of the wait node are activated will be generated if all its input triggers have been activated.

- In the case of the branch node, when all the input triggers have been activated, a condition variable (its location being specified by a pointer) is checked and only the internal event corresponding to the condition variable is generated. As seen in table 6.1, there is a different internal event for each condition

of the branch node. To simplify the implementation, only simple Boolean conditions may be allowed.

- In the case of the merge node, a new event vector indicating that the output trigger(s) of the merge node are activated will be generated if any one of its input triggers has been activated.

- In the case of the gate node, the state of the gate must be remembered. This can be done simply in the state matrix at the position corresponding to the enable input of the gate. When the enable input occurs, it is simply kept in the state matrix. Input triggers are only remembered in the state matrix if the gate is enabled. If the gate is enabled and all of its input triggers have been activated, a new event vector will be generated to indicate that the gate output trigger(s) are now active. If a disable input occurs, the column corresponding to the gate node in the state matrix would simply be cleared to disable the gate and forget all past triggers.

- The procedure to check timer/counter node is similar to the check gate procedure. When the timer is first enabled, a count location is initialized with the counter's initial value. After this time, any occurrence of the input trigger will cause the count to be decremented. If it reaches zero, the proper internal event will be generated. When the counter is disabled, the counter column is simply cleared as in the case of the gate node.

Note that this algorithm is iterative, because a new event vector may be generated when the state matrix is checked, requiring another pass through the state matrix. Also note that

all nodes are reset to their initial condition after they fire their output triggers or, in the case of the activity initiator node, when the activity is placed in the ready-to-run queue. This is accomplished by clearing the node's column in the state matrix.

In summary, the ASN managers are responsible for putting activities that need to be executed in the "ready-to-run" queue and for informing the system manager when the process is terminated.

6.2.3 The Activity Scheduler

The responsibility of the activity scheduler is to find idle processors to execute the activities in the "ready-to-run" queue. To be able to do so, it must have knowledge of the state of all the processors in the system, i.e. if a processor is idle or busy and if it is busy, it must know which activity it is executing. The activity scheduler may also require additional information depending on the scheduling algorithm used.

There are a number of possible scheduling algorithms that can be used [BAKE 74]. In a system consisting of homogeneous processors (all the processors can execute all the activities in the system) and activities with equal priorities, the best scheduling algorithm is a simple First Come First Served (FCFS) algorithm. In this algorithm, a single FIFO queue of ready-to-run activities is used and the first activity in the queue is simply assigned to the first free processor. More complex algorithms are

possible by adding priorities to activities. This would require multiple ready-to-run activity queues, one for each priority level. These priorities may be fixed during the design phase or they may be dynamically assigned based on the real-time constraints of the process.

Scheduling algorithms will not be discussed further in this presentation as they are not the main concern of this thesis. The important point to note is that a multi-activity executive based on ASN's can provide scheduling algorithms that are able to meet real-time constraints imposed on processes. This is because the executive can look at the ASN's and find out ahead of time which activities will have to be scheduled before the process is terminated. It can then associate higher priorities to these activities if the process is "running late". This is much harder to do in a multitasking system because the sequencing relationships between the activities of a process are hidden in the tasks.

6.2.4 Inter-Activity Communication

So far in this section, only the problem of control flow between the activities has been addressed. But data flow is also present between the activities. When an activity is started, it needs its input parameters, and when it terminates, it must know what to do with its results. As mentioned in previous chapters, the data flow in ASN's can be closely associated with the control flow by showing the data along the triggers in the ASN (figure

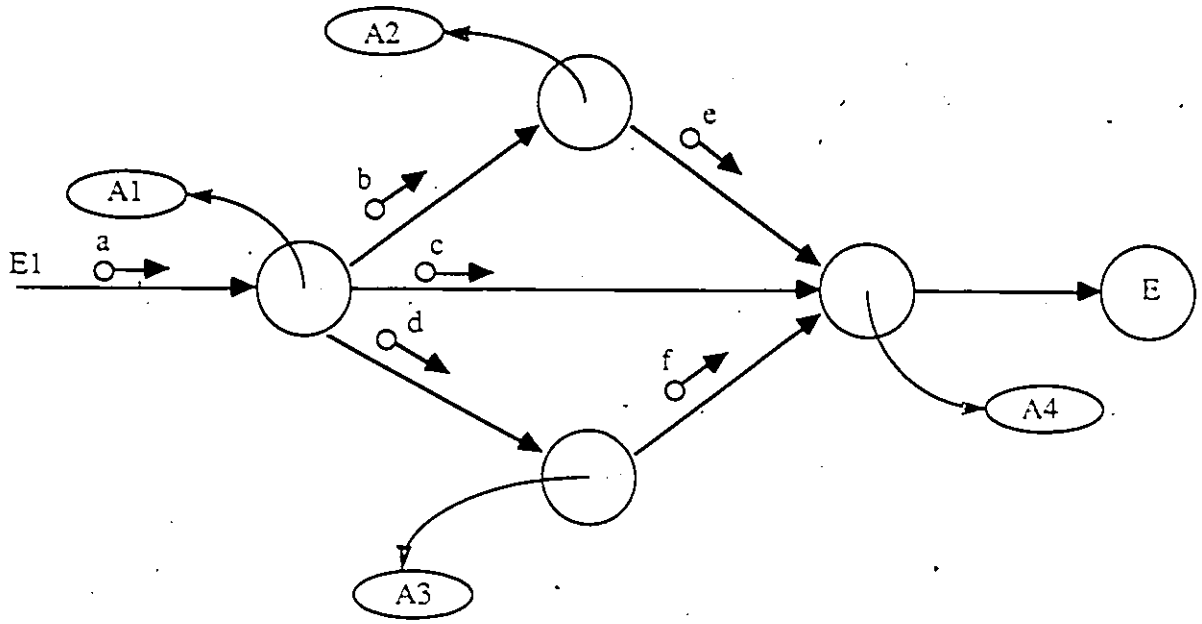


FIGURE 6-4 SIMPLE ASN WITH DATA FLOW INDICATED

6.4). In a similar manner, the multi-activity executive can add data flow information in the ASN matrix. However, since data only flows between activity initiator nodes (on the diagram, the data may pass through a number of nodes, but data is only "produced" and "consumed" by activity initiator nodes), a simple matrix indicating the flow of data between activities can be used by the executive to correctly route the data between the activities. For example, the data flow in the ASN of figure 6.4. is shown in table 6.2.

		Input Sink			
		A1	A2	A3	A4
	E1	a			
Data	A1		b	d	c
Source	A2				e
	A3				f
	A4				

TABLE 6.2 DATA FLOW MATRIX FOR EXAMPLE IN FIGURE 6.4.

This table would inform the executive, for instance, that activity A4 requires data "c", "e" and "f", from activities A1, A2 and A3 respectively. Note that input data for activities may also come from external events (e.g. a reading from an D/A port). In figure 6.4, a trigger is explicitly shown to carry the data item "c" from activity A1 to A4. This trigger is not really needed because it is evident from the sequencing rules in ASN's that A1 will be completed when A2 and A3 are completed.

The data flow matrix indicates the source and destination(s) of a piece of data, but the executive designer must still decide how to send this information to the processor which will execute the activity. Note that the responsibility of transmitting input data to activities will have to fall on the activity scheduler, because it is the only module of the executive that knows on which processor an activity will be executed. A number of schemes can be used to transmit this data between activities without congesting the control bus between the activity processors and the central controller:

- If the data to be transmitted is relatively small (a few bytes), the data can be sent directly on the control bus between the central processor and the subordinate processor. In this case, when the subordinate processor completes the activity, it would send the results (output data of the activity) to the central controller along with its activity completion message. In the same way, when the activity scheduler requests a subordinate processor to start an activity, it would send the data directly along with the activity identifier. It would be the responsibility of the central control to store these results until they would be sent to other activities.

- If the data to be transmitted is relatively large, it should not be transmitted on the control bus as this will affect the performance of the entire system. One solution to this problem is to use a shared memory with an arbitration unit. In this case, only a data identifier or an address will be transmitted on the control bus instead of the data itself. It will then be the

responsibility of the subordinate processor to request the information from the shared memory arbitration unit. Note that this will reduce the traffic on the control bus, but the problem might simply be transferred to the data bus.

- Another solution to large data transmission problem is simply not to transmit the data at all. The output data of the activity can simply be left in the local RAM of the subordinate processor and the activity scheduler can make sure that the activities that need this information as input data are executed on the same subordinate processor.

- A combination of the above scheme may also be used, where the data is left in the local RAM of the subordinate processor and it will only be transmitted when the next activity that needs it is assigned a processor. In this manner, if the next activity is assigned to the same processor, the data will not be transmitted for nothing.

These are not the only solutions to the inter-activity communication problem in a multiple processor environment, but a thorough discussion of this problem is beyond the scope of this thesis. It should be pointed out that the problems associated with inter-processor communication are not unique to the multi-activity system but also exist in multitasking multiple processor systems. Hence, solutions used in these cases can also be applied to this problem.

6.3 MULTI-ACTIVITY EXECUTIVE SIMULATOR

As discussed in the first chapter, the design effort can be greatly reduced if a system can be simulated before it is implemented. In this section, a simulator program based on the multi-activity executive described in the previous section will be discussed. The purpose of this simulation is really to find out if the system's real-time constraints can be met under a particular activity to processor allocation. The simulation was implemented on a personal computer using a version of PASCAL. The program listings of the simulation are included in the appendix B of this thesis.

Three main programs make-up the simulation package: the System Specification Program, the Activity to Processor Allocation Program and the ASN Simulator itself.

6.3.1 The System Specification Program

Before using the simulator program, the designer must specify the system using ASN's. These ASN's must then be translated into ASN matrices and entered in the computer. The System Specification Program enables the designer to enter the system specification simply by answering a series of questions on the connectivity of his ASN's. It does not provide any editing capabilities, hence if a modification needs to be made, the entire system must be re-entered. The output of this program is a file containing the ASN matrices. This file will be read as input data by the simulator.

Presently, a program is being written which will let the designer enter the specification directly in the ASN matrix format and edit it at will. If a commercial implementation of the simulation system were to be done, these programs would have to be replaced by a graphical editor which would permit the user to enter the various ASN nodes directly on the screen. A syntax checker would also be added to such a system to verify if all the rules concerning the connectivity of nodes in ASN's have been obeyed.

6.3.2 The Activity to Processor Allocation Program

Certain implementation details must also be entered before the simulation program can be run. These details relate to the number and type of processors in the system, and are referred to as the activity to processor allocation of the system. The activity to processor allocation consists of: the number of processors in the system, the activities that each processor may execute and the time it would take to execute each activity on that particular processor. A number of different activity to processor allocations can be entered for the same system specification. This enables the designer to try out various allocation schemes and compare their simulation results. The resulting allocation is saved in a file which will also be read by the ASN simulator.

6.3.3 The ASN Simulator Program

The reasons for implementing the ASN simulator were two fold:

one, to verify if a multi-activity executive for a centrally controlled multiple processor system could be implemented, and two, to examine the feasibility of a design tool based on this executive which can check if a system's real-time constraints can be met before actually implementing the system. Since this is really a feasibility study, a number of assumptions were made to simplify the problem and a number of features where added to show the behavior of the executive:

- Like the FSM simulator, the ASN simulator operates in discrete time units. This means that all systems events (external event, activity completion, etc...) can only occur at one of these discrete time slices.

- All the activities, external events, and processors are simply identified by consecutive numbers, not names (e.g activities 1 through 11). Only condition and counter variables may be labelled using a single letter.

- The simulator has two modes of operation: interactive and continuous.

- In the interactive mode, the status of the system is displayed at each discrete time unit and the user is responsible for "simulating" the system's environment by entering the external events. Presently, only one external event can be entered at each time unit. The purpose of the interactive mode is really to show the behavior of the system under certain conditions. It would not be used to gather statistics on the system's response.

- In the continuous mode, all simulation parameters must be initialized in the set-up phase and the simulation runs continuously until some prespecified time. Hence, the program must simulate the occurrence of external events. The arrival rate of these events is then assumed to be Poisson; and the user must simply enter the arrival rate of the different external events in the set-up phase.

- The branch nodes are restricted to have only Boolean conditions. In the interactive mode, the status of the condition variable (true or false) must be entered by the user when the node is reached in the ASN. In the continuous mode, each condition variable is assigned a probability that it is true. When the branch node is reached, a uniform random number is generated which is used to decide on the state of the condition variable.

- The timer/counter's initial count can be entered when the simulation is initially set-up, or in the interactive mode, the user may choose to enter the initial count each time a timer/counter node is reached.

- The inter-activity communication requirements are totally ignored. Data transfer times should thus be included in the execution times.

- There is no congestion on the control bus between the central processor and the subordinate processors, hence when the activity scheduler requests the execution of an activity, it is started immediately.

- The occurrence of a starting event for a process will be ignored if it occurs while the process is still running due to a previous occurrence of that starting event. Hence multiple instances of the same ASN are not allowed.

- Statistics are constantly being updated during the simulation and are only displayed at the end of the simulation run. The types of statistics gathered will be discussed in the next section.

The structure of the ASN simulator is shown in figure 6.5. The first step in the simulation is to get the system name and allocation identifier to be able to read the specification and allocation files. Then a number of initialization parameters are requested from the user in the "set-up simulation" procedure (mode of operation, event and branch probabilities, initial timer values, etc...). The simulation can then be properly started by continuously running the "system manager", the "ASN managers" and the "activity scheduler" procedures in sequence until the simulation is complete. The "display system status" procedure may also be called in this loop to show the status of the system at each time unit. If an error occurs during the simulation, such as a ready to run queue overflow or too many active timers, the simulation is stopped and the procedure "display error message" is executed. If the simulation ends properly, then the "display statistics" procedure is executed, letting the user see statistical results of his simulation run. Each of these procedure will be briefly described in the next sub-sections.

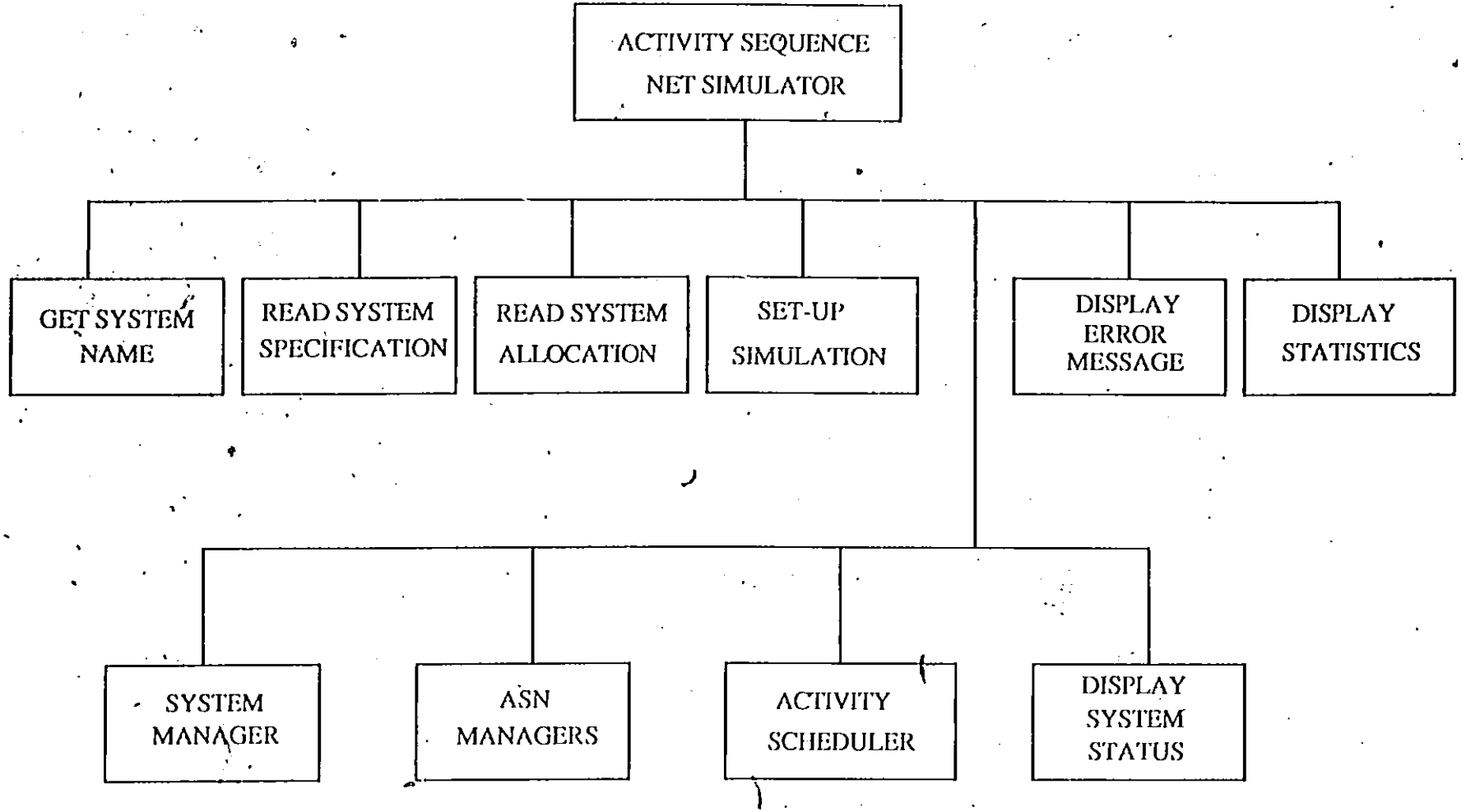


FIGURE 6-5 HIGH LEVEL BLOCK OF THE ASN SIMULATOR

6.3.4 The System Manager Procedure

The "system manager" procedure, as its name implies, simulates the system manager module of the multi-activity executive. The block diagram of this procedure is shown in figure 6.6. The system manager must be informed of occurrences of external events; this is done by calling the procedure "get external event from user" when the simulation is run in the interactive mode or "get random external event" when in continuous mode. Note that these procedures could be modified if the external event arrival process is not Markovian. The ASN manager will awaken ASN's if their respective starting event occurs, provided the ASN was "dormant". If an external event needs to be passed on to an ASN manager, the "set-up external event vector" procedure is called to update the event vector of the ASN concerned.

6.3.5 The ASN Managers Procedure

The "ASN managers" procedure is used to simulate the execution of ALL the ASN managers. It simply calls the "ASN manager" procedure for each ASN that is awakened in the system and that has a new event vector that needs to be processed. The block diagram of this procedure is shown in figure 6.7. Note that the "ASN manager" procedure has an input parameter identifying which ASN is being considered. After updating the process state matrix of this ASN, it simply checks the matrix, column by column, by calling the appropriate procedure depending on the type of node represented in the column. The "all input trigger

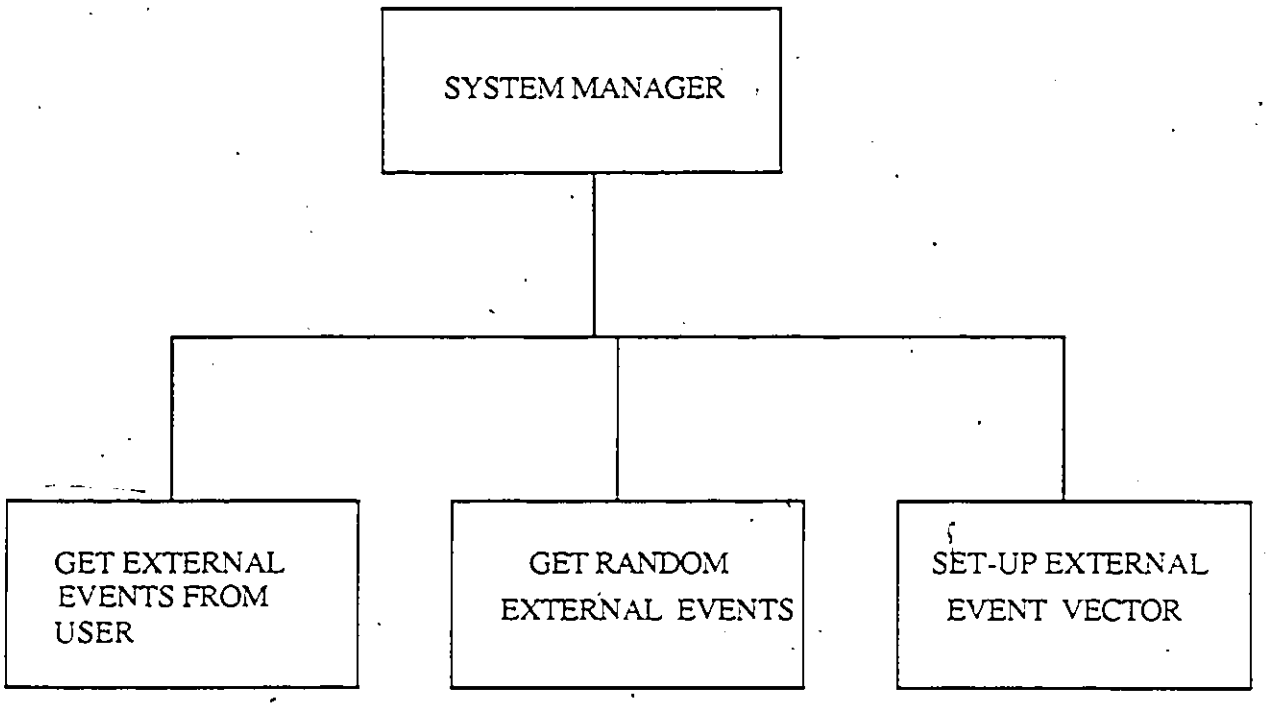


FIGURE 6-6 BLOCK DIAGRAM OF SYSTEM MANAGER

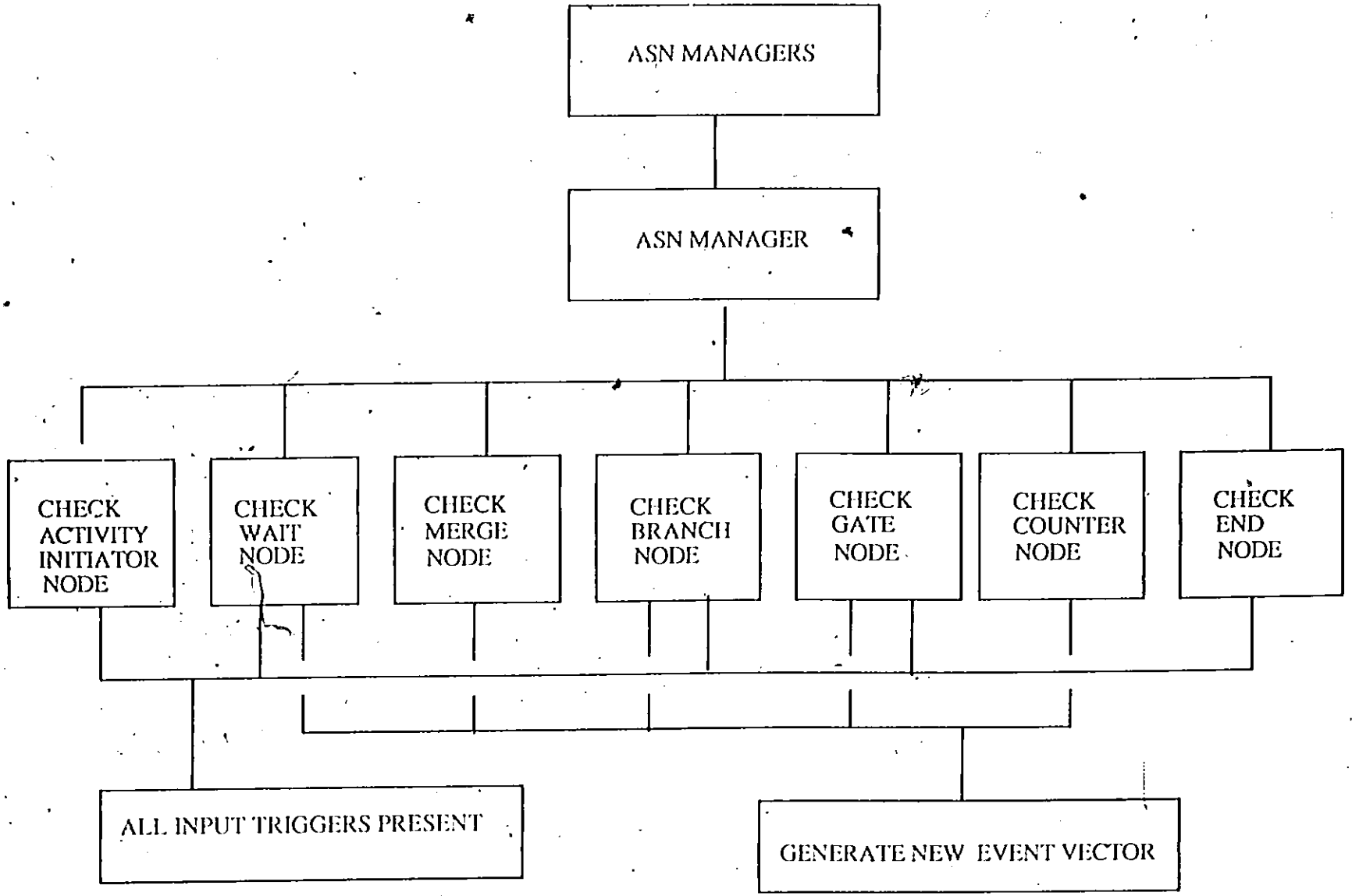


FIGURE 6-7 BLOCK DIAGRAM OF ASN MANAGERS

present" function is used by all the "check nodes" procedure that perform an "and" function of their input triggers to verify if all their input triggers have occurred. All the "check nodes" procedures, except those for the activity initiator and end node, may also call "generate new event vector" if a node needs to activate its output triggers (i.e. an internal event has occurred). Note that to simplify the implementation of the executive in the simulation, the "check end node" procedure takes care of terminating the ASN if all the input triggers of the end node are present, instead of the system manager. This was necessary because the three modules of the executive, the system manager, the ASN managers and the activity scheduler are run in sequence, not concurrently.

6.3.6 The Activity Scheduler Procedure

The "activity scheduler" procedure in the simulation (figure 6.8) implements more than the activity scheduler module in the multi-activity executive. It is also responsible for simulating the activities on the subordinate processors by simply down counting their execution time. This is done in the "simulate activity execution" procedure. When an activity is completed, (actually just before, because of the sequential nature of this implementation), the procedure "generate new event vector" is called to notify the ASN manager of this internal event. The scheduling algorithm is part of the "try to assign activity to processor" procedure. Different scheduling algorithms could be tried by changing this procedure accordingly. At this time, only

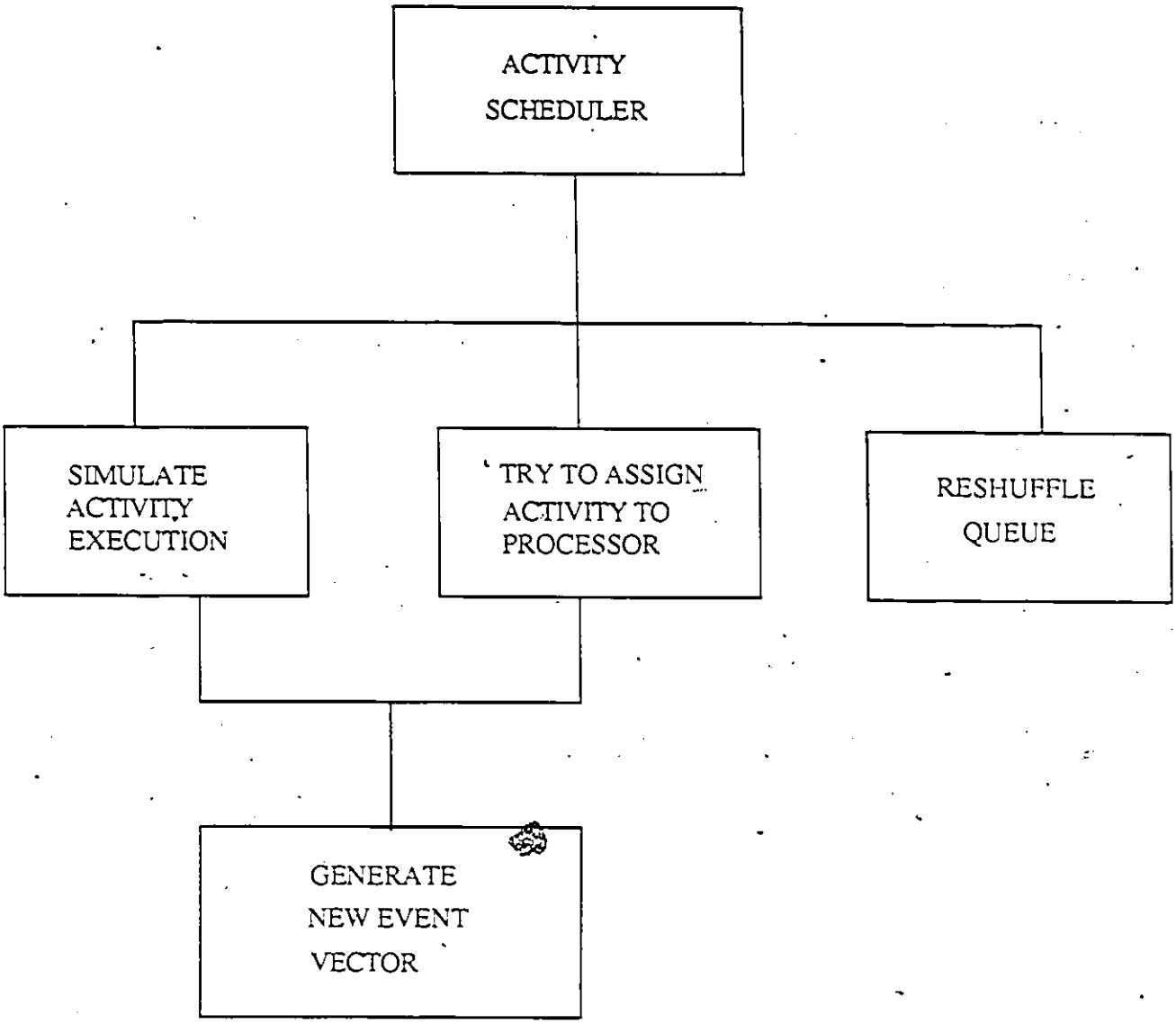


FIGURE 6-8 BLOCK DIAGRAM OF ACTIVITY SCHEDULER

the simple FCFS algorithm discussed in the previous section was implemented.

6.4 SIMULATOR RESULTS

The purpose of simulating a system is to gather statistical result on its behavior to verify, before hand, if it can be implemented successfully. In this section, the statistical results that are produced by the ASN simulator program will be presented. To discuss these results, an example system will be simulated. It will consist of the process depicted by the ASN in figure 6.3 which will be labelled ASN 1, and three other processes illustrated in figure 6.9.

Two different activity to processor allocations will be used. In the fist one (table 6.3), there will be only one subordinate processor while in the second (table 6.4), there will be four. In this example, the execution time of a particular activity is always the same regardless of which processor it is running on. Allocation 2 was chosen in such a way that concurrent activities are assigned to different processors. For example, activities A2 and A3 will always be executed concurrently in ASN 1, for this reason they were assigned to different processors; A2 to processors 1 and 2, and A3 to processors 3 and 4. These two activities were also assigned to more than one processor because both of them are present in other ASN's; A2 is part of ASN 1 and 3, while A3 is part of ASN 1 and 2.

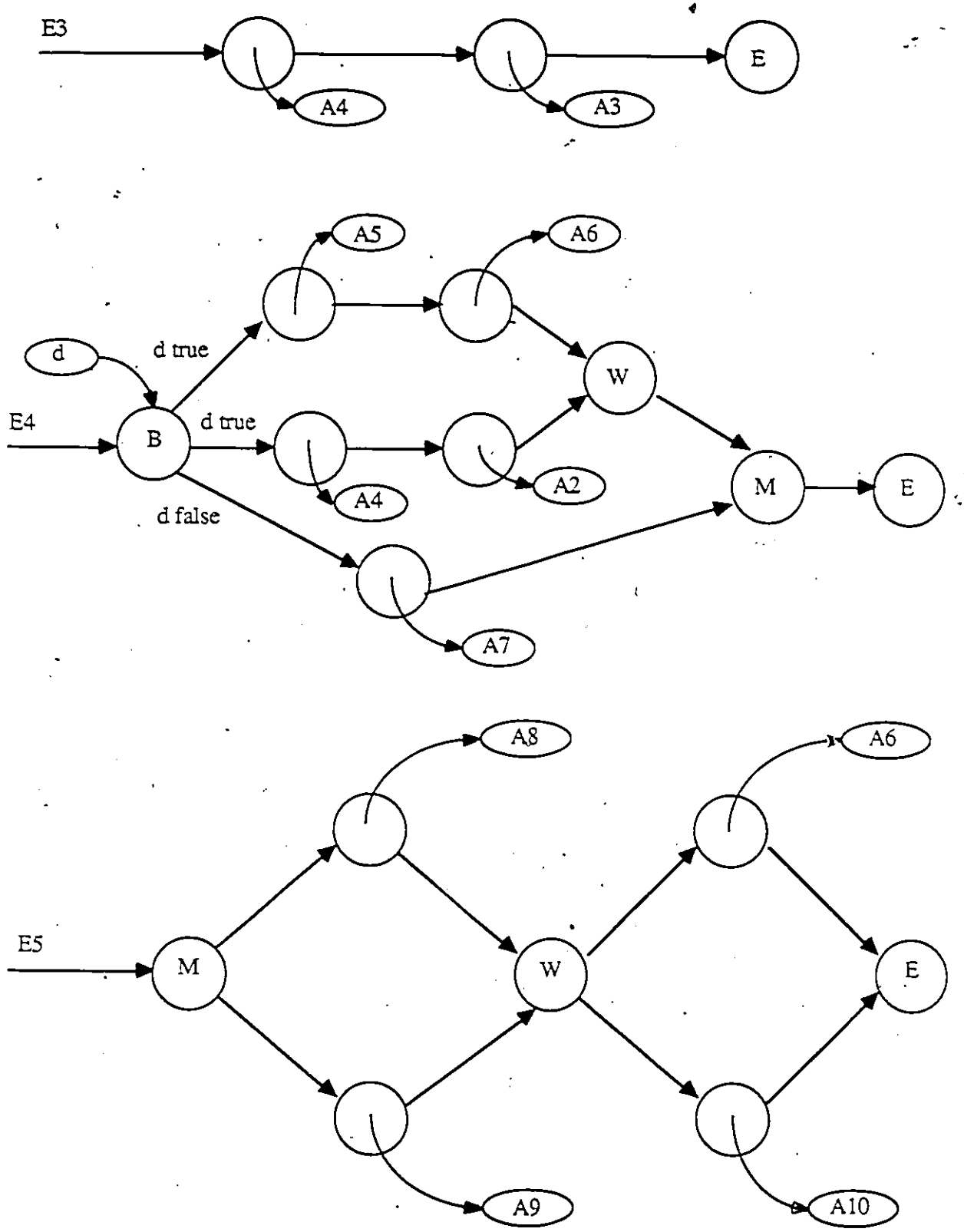


FIGURE 6-9 ASN'S 2, 3 AND 4 OF EXAMPLE SYSTEM

Processor	Activity ID	Execution time
1	1	4
1	2	3
1	3	5
1	4	2
1	5	4
1	6	6
1	7	4
1	8	3
1	9	5
1	10	1

TABLE 6.3 ACTIVITY TO PROCESSOR ALLOCATION 1.

Processor	Activity ID	Execution time
1	1	4
1	2	3
1	6	6
1	8	3
2	2	3
2	4	2
2	5	4
2	9	5
3	3	5
3	4	2
3	6	6
4	1	4
4	3	5
4	7	4
4	10	1

TABLE 6.4 ACTIVITY TO PROCESSOR ALLOCATION 2.

In these simulations, the following values will be used for the set-up of the simulation:

- The event arrival rate will be 0.02 per time unit for all the events, except E1 which is the clock event and will thus occur at each discrete time unit (probability of occurrence 1).

- The condition variable "c" of the branch node in ASN 1 will

have a probability of 0.9 of being true, hence the feedback path would only be executed one out of every ten times.

- The branch variable "d" in ASN 3 will have equal probability of being true or false (0.5).

- The counter variable "t" in ASN 1 will always start at 10.

- The duration of the simulation will be 30 000 time units.

Using the first activity to processor allocation scheme (table 6.3), the simulation produces the results shown in table 6.5.

The first group of statistics in the table shows the processor utilization, i.e. what was each processor doing during the simulation period. By looking at these statistics, the activities that consume most of the processor time can be found and if necessary, their execution time may be speeded-up by using faster processors. The percentage of idle time may also be a useful figure to find out if any processors are under-utilized.

The next set of statistics are related to the external events. It gives the actual number of occurrences of the events during the simulation. Statistics are also kept on external events that are "rejected". This would happen if no ASN is waiting for them (in our example, if the clock event occurs while ASN 1 is dormant), or if they are starting events for an ASN which is already awakened.

The most important output of the simulator is probably the

STATISTICS OF PROCESSOR 1:

ACTIVITY	NUMBER OF TIMES EXECUTED	PERCENTAGE OF TIME
IDLE		38.30 %
1	399.5	5.33 %
2	612.0	6.12 %
3	844.0	14.07 %
4	692.0	4.61 %
5	230.0	3.07 %
6	647.0	12.94 %
7	229.0	3.05 %
8	417.0	4.17 %
9	417.0	6.95 %
10	417.0	1.39 %

EVENT STATISTICS

EVENT NUMBER	NUMBER OF OCCURRENCES	NUMBER OF REJECTIONS	PERCENTAGE OF REJECTIONS
1	30000	23373	77.91 %
2	557	142	25.49 %
3	578	116	20.07 %
4	605	146	24.13 %
5	574	157	27.35 %

PROCESS STATISTICS

ASN ID	NUMBER OF TIMES AWAKENED	COMPLETION TIME MAXIMUM	COMPLETION TIME AVERAGE
1	415	64	16.9
2	462	41	14.1
3	459	49	14.9
4	417	46	21.0

TIMER/COUNTER STATISTICS

TIMER "t": STARTED 446 times
EXPIRED 63 times or 14.13 % of the time.

QUEUE STATISTICS

	BEFORE SCHEDULING	AFTER SCHEDULING
AVERAGE SIZE	0.862	0.698
MAXIMUM SIZE	7	6

TABLE 6.5 SIMULATION RESULTS FOR ALLOCATION 1.

process statistics. These give, for each ASN, the number of times it was awakened, and the average and maximum execution time of the entire process, that is, the time elapsed between the occurrence of the external starting event and the time the end node of the ASN is reached. These statistics are used to verify if real-time constraints on response time can be met.

The simulator also produces statistics on timer variable and ready-to-run activity queue length (average and maximum). In this example, the timer statistics shows how many times activity 1 in ASN 1 timed-out. The queue statistics are divided into before and after the activity scheduler is executed to show how many activities are scheduled in one run of the scheduler.

To see the effect of adding more processors to the system, the system was simulated again with the same input data, except for the activity to processor allocation. Four processors are now used according to the allocation in table 6.4. The simulation results are shown in tables 6.6 and 6.7.

Comparing the results from simulation with allocations 1 and 2, a few remarks can be made:

- The individual processor utilization in allocation 2 is much lower (Higher percentage of time spent doing nothing, idle).

- In allocation 2, the idle processor time generally increases from processor 1 through 4. This is due to the scheduling algorithm used: when an activity needs to be executed, the search for an idle processor that can execute the activity always starts

STATISTICS OF PROCESSOR 1:

ACTIVITY	NUMBER OF TIMES EXECUTED	PERCENTAGE OF TIME
IDLE		72.29 %
1	451.5	6.02 %
2	707.0	7.07 %
6	490.0	9.80 %
8	482.0	4.82 %

STATISTICS OF PROCESSOR 2:

ACTIVITY	NUMBER OF TIMES EXECUTED	PERCENTAGE OF TIME
IDLE		84.09 %
2	90.0	0.90 %
4	479.0	3.19 %
5	284.0	3.79 %
9	482.0	8.03 %

STATISTICS OF PROCESSOR 3:

ACTIVITY	NUMBER OF TIMES EXECUTED	PERCENTAGE OF TIME
IDLE		78.11 %
3	860.0	14.33 %
4	306.0	2.04 %
6	276.0	5.52 %

STATISTICS OF PROCESSOR 4:

ACTIVITY	NUMBER OF TIMES EXECUTED	PERCENTAGE OF TIME
IDLE		91.45 %
1	62.0	0.83 %
3	154.0	2.57 %
7	266.0	3.55 %
10	482.0	1.61 %

TABLE 6.6 PROCESSOR UTILIZATION FOR ALLOCATION 2.

EVENT STATISTICS

EVENT NUMBER	NUMBER OF OCCURRENCES	NUMBER OF REJECTIONS	PERCENTAGE OF REJECTIONS
1	30000	25353	84.51 %
2	523	71	13.58 %
3	568	67	11.80 %
4	633	83	13.11 %
5	589	107	18.17 %

PROCESS STATISTICS

ASN ID	NUMBER OF TIMES AWAKENED	COMPLETION TIME MAXIMUM	COMPLETION TIME AVERAGE
1	452	28	11.3
2	501	11	8.0
3	550	17	8.3
4	482	19	12.4

TIMER/COUNTER STATISTICS

TIMER "t": STARTED 514 times
EXPIRED 0 times or 0.00 % of the time.

QUEUE STATISTICS

	BEFORE SCHEDULING	AFTER SCHEDULING
AVERAGE SIZE	0.225	0.029
MAXIMUM SIZE	6	3

TABLE 6.7 OTHER SIMULATION RESULTS FOR ALLOCATION 2.

at processor 1. Hence if processor 4 is assigned an activity it means that processors 1, 2 and 3 were not able to do it.

- Since more processors are available, the maximum and average process completion time is much lower with allocation 2.

- The event rejection rate is lower with allocation 2, because the average completion time of the processes is lower reducing

the chances of a starting event occurring while the response to the previous event is still being carried out.

- Because there can be more idle processors to execute the activities, the average and maximum length of the ready-to-run activity queue are much lower with allocation 2.

- The timer "t" did not expire in the second case because it took less time on the average to complete activity 1, since it did not need to spend so much time in the queue.

By using these statistics properly, the system designer can find an activity to processor allocation that can meet his real-time constraints while keeping the hardware cost of his system down.

6.5 CONCLUDING REMARKS

If an actual system were to be implemented under the multi-activity executive described in this chapter, the executive would have to be implemented on the central controller. As in the case of many popular operating systems, it would be possible to code the executive for various hardware configurations and have it available as an off-the-shelf component. The system designer would then simply have to download the various tables required by the executive (tables which are the outputs of the various specification programs) and write the code for the activities themselves.

The simulator program discussed in this section was implemented only to verify the feasibility of such a program. If a commercial product were to be made out of it, a number of important modifications would have to be made:

- The external event generator procedure would have to be modified to be able to generate events that do not follow a Markovian process. For instance, in many real-time systems external events are dependent on one another.

- Multiple instances of ASN's may be needed (i.e. starting event that occur while the ASN is still awakened due to a previous occurrence of the event should not always be rejected, instead a new instance of the ASN may be started).

- The inter-activity communication process may need to be accurately modelled in the simulation.

- A number of activity schedulers with different scheduling algorithm should be implemented. This would let the system designer choose which algorithm is better suited to his particular system.

CHAPTER 7

CONCLUSION

This concluding chapter is divided into two sections. In the first section, a brief summary of the thesis is presented, emphasizing what was achieved in this research. In the second section, some of the parallel work being done in this area is discussed and some suggestions for future work and research directions are made.

7.1 THESIS SUMMARY

This thesis has addressed the problem of designing event driven microprocessor-based systems. To design simple sequential event response systems, a tool based on finite state machines was presented. This tool can provide a complement to the traditional flowcharts when a system has to respond to external events by executing different activities whose selection depends on the state of the system. In these cases, FSM's are used to represent the response selection while flowcharts can be used to design the activities. A table look-up executive is then used to coordinate the execution of the activities in response to external events. One advantage of this type of executive is that it adds very little overhead to the system's operation. A simulation program

was written to analyze the behavior of such systems and verify their performance in terms of missing events and response time.

When it is not possible to meet real-time requirements with sequential event response systems, concurrent event response systems must be used. It was found in chapter 3 that there are no design tools suitable for system designers who are not computer specialists. These tools were found too complex for such users for two main reasons: either they are specifically targeted for use in a multitasking environment, and hence require experience in designing with multitasking operating system; or, they are too general and do not provide a proper framework for inexperienced users. To help these designers, a new graphical tool for the design of concurrent systems, Process Activity Diagrams, was introduced in chapter 4. This tool, through Activity Sequence Nets, expresses the system's response to external events (the system's processes) in terms of the sequencing relationships between activities. As in the case of the FSM tool for simple systems, an executive is required to coordinate the execution of the various activities of the system. This multi-activity executive was chosen to be based on a centrally controlled multiple microprocessor architecture for the following reasons:

- Only a multiple processor system may provide true concurrent execution of activities and hence meet stringent real-time constraints.

- A centrally controlled system made the implementation easier because the executive requires knowledge of the sequencing

relationships between the activities (the ASN's) and the current state of each of the processes. Since this information needs to be continuously updated, it is simpler to have it available only in a central location rather than distributing it over all the processors as this would increase the inter-processor communication requirements.

- As was mentioned in chapter 5, a centrally controlled system can be a good design choice, provided that the proper steps are taken to solve the problems related to reliability and communication bottlenecks between the central controller and the subordinate processors.

- A centrally controlled system may also offer some advantages over a distributed system in the following areas:

  - They are simpler to simulate making it possible to verify system performance before implementation.

  - When knowledge on the state of the processors is available at a central site, it is possible to implement better scheduling algorithms which can provide guaranteed real-time response, load balancing, etc...

To show the feasibility of this multi-activity executive, the ASN simulator described in chapter 6 was implemented. The performance measures obtained from the simulation can be used by the system designer to find an activity to processor allocation which meets his needs.

In certain cases, a designer who has experience with

multitasking operating systems may choose to implement his system in this environment. As was shown in chapter 5, PAD's could be used in the design process as they can help the designer partition activities into tasks in such a way as to minimize inter-task communication and/or maximize concurrency. It is thus possible to let the product designer, which is most familiar with the system to be designed, first specify it using PAD's and then let the computer specialist map these requirements onto tasks. The computer specialist may then use tools developed for multitasking systems, with which he is familiar, to assist him in the implementation of the system.

7.2 PARALLEL AND FUTURE WORK

The purpose of this thesis was to introduce PAD's and show their applicability as a design tool for concurrent systems. At this level, only a feasibility study was required. The next step would be to design the proper toolset which would consist of the following:

- A program to enter the system specifications in the computer
- A program to verify the syntax of the ASN in the specifications
- A program to obtain an initial activity to processor allocation
- A simulator program to verify the performance of the system

The "System Specification" program discussed in chapter 6 is much too rudimentary for any practical use because it does not provide any editing facility to the user. A program with such a

feature is currently being written. It permits the entry of the ASN in matrix notation and lets the user edit the specification at will. To make a product out of this toolset, a graphical editor would be essential as it would allow the user to see the diagrams directly on the screen.

A program is also being written to check the syntax of the ASN which was entered in the computer by one of the previous programs. It can detect such errors as multiple triggering and deadlock situations, and issue warnings in other cases such as an enable input missing on a gate or timer node.

To implement the system using the multi-activity executive, an activity to processor allocation must be defined. A first allocation could be found using the ATC's and it could be fine-tuned using the simulator program. A feasibility study is currently under way to determine if a program can be written which will automatically find an initial allocation which can meet certain requirements imposed by the user. The possibility of using an expert system to implement this program is also being investigated.

The ASN simulator program would also require some enhancements to make it more useful for the simulation of real systems:

- The random event generator could be modified to allow for dependent and non-Markovian events.

- Multiple instances of the same ASN could be permitted.

In this case, the occurrence of a starting event could start a new instance of an ASN if the response to a previous occurrence of this event is still being carried on.

- The execution times of certain activities could be random instead of fixed to model data dependent execution times.

- The inter-processor communication requirements could be modelled in the simulation to show their effect on system performance.

- Different scheduling algorithms could be implemented to let the user try them out and find out which algorithm results in the best performance for his particular system. This is currently being done to verify certain scheduling algorithm used in job shop scheduling problems.

If systems are to be implemented under the multi-activity executive, the executive will have to be implemented on an actual multiple microprocessor system. Note that, in this case, part of the simulator program could be used directly to code the executive because the simulator was divided into modules corresponding to those of the executive.

If the system is to be implemented under a multitasking operating system, tools could be developed to automate the

process of mapping activities to tasks. The feasibility of this program is currently being considered and again, expert systems may provide an interesting solution to this problem.

Current research is also being carried out in application areas of PAD's as they are being considered for the design of flexible manufacturing systems (FMS) [PICH 86].

To conclude this thesis, two final remarks should be made about ASN's:

- The set of nodes that were defined does not represent a closed set. It was chosen in this fashion to maintain a relatively small set of nodes which would still provide some interesting features for the specification of many control type systems. Many systems could be specified using only activity initiator, branch, merge, wait and end nodes. On the other hand, the description of other systems may be greatly facilitated by defining more complex nodes than the ones presented here. It is conceivable that different sets of nodes may be available to the user depending on the type of system that will be designed.

- PAD's may not be suitable for the design of all event driven systems. For instance, in cases where processes are very interrelated and require much communication between themselves, PAD's may be inadequate because they do not explicitly show these interrelations. More research needs to be done to find the limitations of PAD's and possibly add some features to them to overcome these limitations.

REFERENCES

ACKE 82 William B. Ackerman, "Data Flow Languages", in IEEE Computer, Vol. 15, No. 2, February 1982, pp. 15-25.

AGER 79 Tilak Agerwala, "Putting Petri Nets to Work", in IEEE Computer, Vol. 12, No. 12, December 1979, pp. 85-94.

ALLW 80 S.T. Allworth, "Introduction to Real-time Software Design", Springer-Verlag New York Inc., 1980, 137 pages.

ARTH 86 James D. Arthur, Roger W. Ehrich, Kelly C. Mulheren, "A Network Specification and Execution Environment", Proceedings of the ISMM International Symposium on Software and Hardware Applications of Microcomputers, Beverly Hills, California, 1986, pp. 132-136..

BAKE 74 Kenneth R. Baker, "Introduction to Sequencing and Scheduling", John Wiley & Sons Inc., New York, 1974, 301 pages.

BELL 84 Herb Bellinger, "Breaking the Constraints of Controller Programming", in Industrial Control, a supplement of Oilweek, Maclean Hunter Publication, Calgary, Alberta, Vol. 35, No. 34, September 24, 1984, pp. S25-S29.

BENA 82 M. Ben-Ari, "Principles of Concurrent Programming", Prentice/Hall International, Englewood Cliffs, New Jersey, 1982, 172 pages.

BERT 82 Gerard Berthelot and Richard Terrat, "Petri Nets for the Correctness of Protocols", IEEE Transactions on Communications, Vol. COM-30, No. 12, December 1982, pp. 2497-2505.

BOCH 78 Gregor V. Bochmann, "Finite State Description of Communication Protocols", in "Computer Networks 2", North-Holland Publishing Co., 1978, pp. 361-372.

BOWE 85 B.A. Bowen and W.R. Brown, "Systems Design Volume II of VLSI Systems design for Digital Signal Processing", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985, 413 pages.

BRAM 83 G.W. Brams, "Reseaux de Petri: Theorie et Pratique, Tome I Theorie et Analyse", E.S.I., Paris, 1983, 183 pages.

BUHR 84 R.J.A. Buhr, "System Design with Ada", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984, 256 pages.

CASH 80 P. Cashin, "Inter-Process Communication", BNR Report, Ottawa, Ontario, 1980, 46 pages.

CCIT 83 CCITT SDL Newsletter Annex 1 to 5, Recommendations Z.100 to Z.104. May 1983.

Z.100 "Introduction to SDL"

Z.101 "Basic SDL"

Z.102 "Structural Concepts in SDL"

Z.103 "Functional Extensions to SDL"

Z.104 "Data in SDL"

CHAS 81 Richard B. Chase and Nicholas J. Aquilano, "Production and Operations Management", Richard D. Irwin Inc., Homewood, Illinois, 1981, 741 pages.

CHU 80 Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan, and Kemal Efe, "Task Allocation in Distributed Data Processing", in IEEE Computer, Vol. 13, No. 11, November 1980, pp. 57-69.

DAVI 82 Alan L. Davis and Robert M. Keller, "Data Flow Program Graphs", in IEEE Computer, Vol. 15, No. 2, February 1982, pp. 26-41.

FATH 83 E.T. Fathi and M. Krieger, "Executive for Task-Driven Multi-Microcomputer System", in IEEE Micro, October 1983.

HANS 78 Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol. 21, No. 11, November 1978, pp. 934-941.

HOAR 78 C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

HOAR 74 C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

HOZJ 79 J. Hozjajenok, "The Use of Petri Nets Models and FPLAs in Programmable Hardwired System Design", in

"Microprocessors and their Applications", Nort-Holland Publishing Co., Euromicro, 1979, pp. 259-269.

HWAN 84 Kai Hwang and Faye A. Briggs, "Computer Architecture and Parallel Processing", McGraw-Hill Book Company, 1984, 846 pages.

JACK 83 Ken Jackson, "MASCOT and Multiprocessor Systems", in "Distributed Computing Systems", Academic Press London, pp. 225-248.

JENS 79 Randall W. Jensen and Charles C. Tonies, "Software Engineering", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979, 580 pages.

KLEI 75 Leonard Kleinrock, "Queueing Systems Volume 1: Theory", John Wiley & Sons, Inc., New York, 1975, 417 pages.

KRIE 81 M. Krieger and E.T. Fathi, "Design Aspects of a Simple Distributed Microprocessor System", Proceedings of the International Conference on Communication, Circuits and Systems, Calcutta, December 1981.

KRIE 86 Moshe Krieger and Robert Joannis, "Process Activity Diagrams: A Tool for the Specification and Design of Multiple Microprocessor Systems", Proceedings of the ISMM International Symposium on Software and Hardware Applications of Microcomputers, Beverly Hills, California, 1986, pp. 157-161.

LAND 79 Jack V. Landau, "State Description Techniques Applied to

Industrial Machine Control", IEEE Computer, Vol. 12, No. 2, February 1979, pp. 32-40.

MACN 85 Edward A. MacNair, Charles H. Sauer, "Elements of Practical Performance Modeling", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985, 269 pages.

MACQ 84 J.J. MacQuarrie, "Programmable Controllers: the Quiet Industrial Revolution", in Industrial Control, a supplement of Oilweek, Maclean Hunter Publication, Calgary, Alberta, Vol. 35, No. 34, September 24, 1984, pp. S12-S21.

NOE 73 Jerre D. Noe and Gary J. Nutt, "Macro E-Nets for Representation of Parallel Systems", in IEEE Transactions on Computers, Vol. C-22, No. 8, August 1973, pp. 718-727.

PAWL 84 Frank Pawlitza, "The Wave of the Future", in Industrial Control, a supplement of Oilweek, Maclean Hunter Publication, Calgary, Alberta, Vol. 35, No. 34, September 24, 1984, pp. S8-S10.

PETE 77 James L. Peterson, "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, September 1977, pp. 223-252.

PHIC 86 Philippe R. Piche, Moshe Krieger, Charles Gauthier and Emile Petriu, "PAD: A New Tool for FMS Design", to be presented at the IECON'86 Twelfth Annual IEEE Industrial Electronics Society Conference in Milwaukee, Wisconsin, September 1986.

RAED 85 Georg Raeder, "A Survey of Current Graphical Programming Techniques", IEEE Computer, Vol. 18, No. 8, August 1985, pp. 11-25.

RAZO 84 Rami R. Razouk, Charles V. Phelps, "Performance Analysis Using Timed Petri Nets", Protocol Workshop, Skytop, Pennsylvania, 1984, 19 pages.

RIGG 77 James L. Riggs, "Engineering Economics", McGraw-Hill Book Company, New York, 1977, 617 pages.

SAIB 85 Sabina Saib, "Ada: An Introduction", Holt, Rinehart and Winston, New York, 1985, 358 pages.

WEGN 83 Peter Wegner and Scott A. Smolka, "Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives", IEEE Transaction on Software Engineering, Vol. SE-9, No. 4, July 1983, pp. 446-462.

APPENDIX A

FINITE STATE MACHINE SIMULATOR

This appendix contains the program listing for the finite state machine simulator written in TURBO PASCAL (c) for the IBM Personal Computer (c).

TURBO PASCAL (c) is a trade mark of Borland International Inc. IBM Personal Computer (c) is a trade mark of International Business Machines Corp.

TABLE OF CONTENTS OF APPENDIX A

```
PROGRAM FSM_SIMULATOR (input, output);

{$R+}

CONST
   max_n_of_states = 9;
   max_n_of_events = 9;
   max_n_of_activities = 9;

{-------------------------------------------------------------------------}

{ Include files }

{$I TABLE.PAS}
{$I FSMSIM1.PAS}
{$I FSMSIM2.PAS}

{-------------------------------------------------------------------------}

VAR
  answer: char;

BEGIN {FSM_simulator}

  {Check for color monitor}
  write(' Do you have a color monitor? [Y/N]: ');
  read(kbd,answer);
  if upcase(answer) = 'Y' then
    begin
      TextColor(yellow);
      TextBackground(blue);
    end;

  repeat

    {Display menu}
    clrscr;
    answer := '0';
    gotoxy(38,7);
    writeln('MENU');
    gotoxy(22,10);
    writeln('1 :  Create or Edit FSM Table');
    gotoxy(22,12);
    writeln('2 :  Simulate Event Checking System');
    gotoxy(22,14);
    writeln('3 :  Simulate Interrupt Driven System');
    gotoxy(22,16);
    writeln('4 :  Quit');
    repeat
      gotoxy(22,19);
      write('Enter Selection [1-4]: ');
      read(kbd,answer);
    until answer in ['1','2','3','4'];
```

```
    case answer of
        '1':  state_table_entry;
        '2':  event_checking_simulator;
        '3':  interrupt_driven_simulator;
    end;

  until answer = '4';
  clrscr;

END.
```

```
{This procedure is used to enter the state table and the activity
 execution time in a file which will be read by the FSM simulator.}

PROCEDURE STATE_TABLE_ENTRY;

  VAR
    ch: char;
    num_of_events: integer;
    num_of_states: integer;
    num_of_activities: integer;
    state_table: array [1..max_n_of_states,1..max_n_of_events,1..2]
                 of char;
    activity_table: array [1..max_n_of_activities] of integer;
    x_table, y_table: integer;
    return: char;
    filename: string[14];
    f: text;

{-------------------------------------------------------------------}

   Procedure display_help;

    begin {display_help}

     writeln;
     writeln;
     writeln(' Upon entering <C>, to create the state table, you will',
             ' be asked');
     writeln(' for the number of states, events, and activities. You',
             ' will then');
     writeln(' be  given a blank table to fill in.  You may only',
             '  enter  valid');
     writeln(' characters  to  describe each cell in the table,  for',
             '  example,');
     writeln(' s2/a1.  If  an  event causes no activity to be',
             '  executed,  then');
     writeln(' label the activity a0.  If you choose <E> then you will',
             ' be able');
     writeln(' to edit an existing state table.  Leave no blanks in',
             ' any of the');
     writeln(' cells.  Use the arrow keys on the keypad to move in the',
             '  table.');
     writeln(' Press ''X'' at any time to exit table creation.  You',
             ' will then be');
     writeln(' asked  for  activity  execution times, note that  the',
             '  default');
     writeln(' value for these times is indicated in brackets.');
     writeln;
     write(' Press any key to continue');
     while not keypressed do {nothing};

    end;

{-------------------------------------------------------------------}
```

```
  Procedure create_table;

   var
     i,j,x,y: integer;
     cell_string: string[8];

   begin {create_table}

      {This procedure simply displays the state table on the screen}
      clrscr;
      y := 4;
      gotoxy(7,y);
      for j := 1 to num_of_events do write('  E',j:2,'    ');
      gotoxy(1,y+1);
      write('     |');
      for j := 1 to num_of_events do write('-------|');
      y_table := 5;

      y := y_table + 1; {to get to input line }
      gotoxy(1,y);
      for i := 1 to num_of_states do
      begin
         write(' S',i:2,' |');
         x_table := 6;
         for j := 1 to num_of_events do {write(' s /a  |');}
            begin
              cell_string := concat(' s',state_table[i,j,1],
                             '/a',state_table[i,j,2],' |');
              write(cell_string);
            end;

         y := y + 1;
         gotoxy(1,y);
         write('     |');
         for j := 1 to num_of_events do write('-------|');
         y := y + 1;
         gotoxy(1,y);
      end;

  end;

{-------------------------------------------------------------------------}

 Procedure fill_table;

 var
    left_lim,right_lim,up_lim,down_lim,x,y: integer;
    event,state,side: integer;

{-------------------------------------------------------------------------}

    Procedure where_in_table;

    {this procedure gives the position of the cursor according to:
```

```
      current state, event and which side of /}

       var
         x_offset,modval: integer;

       begin {where_in_table}

         state := (y-(y_table-1)) div 2;
         x_offset := x - x_table ;
         event := x_offset div 8 + 1;
         modval := x_offset mod 8;

         if modval < 4 then
            side := 1
         else
            side := 2;

       end;

{-------------------------------------------------------------------------}

   Procedure check_for_error;

   {Check for blanks in the state table}

     var
       i,j: integer;
       error: integer;
       n,code: integer;

     begin {check_for_error}

       return := 'X';
       error := 0;
       i := 0;
       while (error = 0) and (i < num_of_states) do
        begin
          i := i + 1;
          j := 0;
          while (error = 0) and (j < num_of_events) do
            begin
             j := j + 1;
             if ((state_table[i,j,1] = ' ') or
                (state_table[i,j,2] = ' '))
                then error := 1
              else
               begin
                 val(state_table[i,j,1],n,code);
                 if (n > num_of_states) or (n < 1) or (code <> 0)
                    then error := 2;
                 val(state_table[i,j,2],n,code);
                 if (n > num_of_activities) or (n < 0) or (code <> 0)
                    then error := 3;
               end;
            end;
```

```
      end;

    gotoxy(1,down_lim + 3);
    if error <> 0 then
      begin
        case error of
          1: write(' Missing entry in');
          2: write(' Invalid state in');
          3: write(' Invalid activity in');
        end;
        write(' state ',i:1,' and event ',j:1,'. ');
        write('<A>bort or <F>ix: ');
        repeat
          read(kbd,return);
        until return in ['a','A','f','F'];
      end;

  end;

{-------------------------------------------------------------------------}

begin {fill_table}

  left_lim  := x_table + 3; { leftmost line}
  right_lim := x_table + num_of_events*8  - 2; { rightmost line}
  up_lim    := y_table + 1;
  down_lim  := y_table + num_of_states*2 - 1;

  repeat
   gotoxy(1,down_lim + 3);
   write(' Press ''X'' to exit                                    ',
         '                                                        ');
   x := left_lim;
   y := y_table +1;
   gotoxy(x,y);
   while not keypressed do {nothing} ;
   read(kbd,ch);

   while upcase(ch) <> 'X' do
   begin
     if ch IN ['0'..'9'] then
        begin
           where_in_table;
           state_table[state,event,side] := ch;
           write(ch);
        end
     else
        if ch = #27 then
          begin
            read(kbd,ch);
            where_in_table;
            CASE ch of
              #72 : if y <>  up_lim then                    { ^ }
                       y := y - 2;
```

```
                #80 : if y <> down_lim then                    {down}
                        y := y + 2;  {down}

                #77 : if x <> right_lim then                   { > }
                        if side = 1 then
                           x := x + 3
                        else
                           x := x + 5;

                #75 : if x <> left_lim then                    { < }
                        if side = 1 then
                          x := x - 5
                        else
                          x := x - 3;

              end;
            end;

       { cursor now validated and updated }

       gotoxy(x,y);
       while not keypressed do {nothing};
       read(kbd,ch);

     end; {while}

     check_for_error;
    until return in ['a','A','x','X'];

  end;
{-------------------------------------------------------------------------}

var i,j: integer;
    n1,n2: integer;

BEGIN  {state_table_entry}

  writeln;

  for i := 1 to max_n_of_states do  {initialize array to all blanks}
    for j:= 1 to max_n_of_events do
      begin
        state_table[i,j,1] := ' ';
        state_table[i,j,2] := ' ';
      end;

  for i := 1 to max_n_of_activities do
     activity_table[i] := 0;

  num_of_states := 0;
  num_of_events := 0;

  repeat
    clrscr;
```

```
    gotoxy(31,2);
    writeln('STATE TABLE ENTRY');
    writeln;
    write(' Enter <C>reate table, <E>dit, <H>elp: ');
    read(kbd,ch);
    if upcase(ch) = 'H' then display_help;
  until upcase(ch) in ['C','E'];
  writeln;

  if upcase(ch) = 'C' then {request all parameters from user}
    begin
      repeat
        writeln;
        write(' How many events (1-->9): ');
        readln(num_of_events);
      until ((num_of_events >= 1) and
             (num_of_events <= max_n_of_events));

      repeat
        writeln;
        write(' How many states (1-->9): ');
        readln(num_of_states);
      until ((num_of_states >= 1) and
             (num_of_states <= max_n_of_states));

      repeat
        writeln;
        write(' How many activities (1-->9) : ');
        readln(num_of_activities);
      until ((num_of_activities >= 1) and
             (num_of_activities <= max_n_of_activities));

      writeln;
      write(' Under what filename will you store this table: ');
      readln(filename);
      assign(f,filename);
    end

  else {edit existing table, hence read state table file}
    begin
      writeln;
      write(' Under what filename is your state table stored: ');
      readln(filename);
      assign(f,filename);
      reset(f);
      readln(f,num_of_states,num_of_events,num_of_activities);
      if ((num_of_states > 9) or (num_of_events > 9)) then
         writeln(' *** state table is too large for screen')
      else
         begin
           for i := 1 to num_of_states do
               for j := 1 to num_of_events do
                 begin
                   read(f,n1,n2);
                   state_table[i,j,1] := chr(48 + n1);
```

```
                    state_table[i,j,2] := chr(48 + n2);
                    readln(f);
                  end;

            for i := 1 to num_of_activities do
              read(f,activity_table[i]);
          end;
      end;

    if ((num_of_states <= 9) or (num_of_events <= 9)) then
      begin

        {draw table on screen}
        create_table;

        {let user edit table}
        fill_table;

        if return = 'X' then
          begin
            if upcase(ch) <> 'C' then
              begin
                close(f);
                assign(f,filename);
              end;
            rewrite(f);

            writeln(f,num_of_states,' ',num_of_events,' ',
                    num_of_activities);
            for i := 1 to num_of_states do
              begin
                for j:= 1 to num_of_events do
                  begin
                    write(f,state_table[i,j,1],' ',state_table[i,j,2]);
                    writeln(f);
                  end;
              end;

            clrscr; {now obtain activity times, do for <E> or <C>}
            writeln;
            for i := 1 to num_of_activities do
              begin
                write(' Enter time for activity ',i,'[',
                      activity_table[i],']',': ');
                readln(activity_table[i]);
                write(f,activity_table[i],' ');
                writeln;
              end;
          end;
      end;

  close(f);

end;
```

```
PROCEDURE EVENT_CHECKING_SIMULATOR;

    VAR
     num_of_states, num_of_events, num_of_activities: integer;
      state_table: array[1..max_n_of_states,1..max_n_of_events,1..2]
                   of integer;
      act_time_table: array[0..max_n_of_activities] of integer;
      simulation_time, time: real;
      event_occ_count, event_det_count: array[1..max_n_of_events]
                                        of real;
      prob: array[1..max_n_of_events] of real;
      event_same_priority: boolean;
      activity_count, activity_time_count: array[0..max_n_of_activities]
                                           of real;
      current_state, next_state, activity, event: integer;
      processor_busy: boolean;
      processor_busy_time: integer;
      display, return: char;
      f: text;
      filename: string[14];

{-------------------------------------------------------------------------}

 Procedure output_display;

 var
  i: integer;
  ch: char;
  choice: char;
  display_stat: boolean;
  percent_det: real;
  busy_time: real;

{-------------------------------------------------------------------------}

  Procedure display_event_stats;

  begin {display_event_stats}

    clrscr;
    gotoxy(32,2);
    write ('EVENT STATISTICS');
    gotoxy(1,4);
    writeln(' TIME: ',time:10:0);
    writeln;
    writeln(' Current State: ',current_state,'   Activity Executing: ',
             activity);
    writeln;
    write(' EVENT NUMBER      OCCURRED       DETECTED ');
    if display_stat then {add headings for final stats}
       write('   PROB OCCURRENCE   % DETECTION');
    writeln;
    writeln;
```

```
    for i := 1 to num_of_events do
       begin
         write(' ',i:6,event_occ_count[i]:18:0,event_det_count[i]:15:0,'
    ');
         if display_stat then {give final percentages}
           begin
             if event_det_count[i] = 0 then
                percent_det := 0
             else
               percent_det := event_det_count[i]/event_occ_count[i]*100

             write(event_occ_count[i]/time:6:5,'          ',
                   percent_det:7:3);
           end;
         writeln;
       end;

   if display_stat then
     begin
       gotoxy(1,20);
       write(' Hit any key to continue ');
       while not keypressed do {nothing};
     end;

  end;

{-------------------------------------------------------------------------}

  Procedure display_activity_stats;

   begin {display_activity_stats}

     clrscr;
     gotoxy(31,2);
     write ('ACTIVITY STATISTICS');
     gotoxy(1,4);
     writeln(' ACTIVITY','ACTIVITY TIME':17,'# OF TIMES':13,
             'TOTAL TIME':15,'% OF PROC TIME':20);
     writeln;
     busy_time := 0;
     for i := 0 to num_of_activities do
      begin
        busy_time := busy_time + activity_time_count[i];
        writeln('    ',i:3,act_time_table[i]:18,
                activity_count[i]:13:0,activity_time_count[i]:15:0,
                ' ':13,(activity_time_count[i]/time)*100:7:3);
      end;
     writeln;
     writeln('  TOTAL ',' ':34,busy_time:10:0,' ':13,
             (busy_time/time)*100:7:3);

     gotoxy(1,20);
     write(' Type any key to continue ');
     while not keypressed do {nothing};

   end;
```

```
{-------------------------------------------------------------------}

begin {display_output}

  if (time = simulation_time) or (return = 'X') then
    begin
      display_stat := true;
      repeat
        clrscr;
        gotoxy(32,7);
        write ('STATISTICS MENU');
        gotoxy(25,10);
        write ('1  Display Event Statistics');
        gotoxy(25,12);
        write ('2  Display Activity Statistics');
        gotoxy(25,14);
        write ('3  Exit');
        gotoxy(25,17);
        write ('Enter Selection [1-3]: ');
        choice := ' ';
        repeat
          read (kbd,choice);
        until choice in ['1','2','3'];

        case choice of
          '1': display_event_stats;
          '2': display_activity_stats;
        end;
      until choice = '3';
    end

  else
    begin
      display_stat := false;
      display_event_stats;
      gotoxy(1,20);
      write(' Type <X> to exit, any other key to continue ');
      while not keypressed do {nothing};
      read (KBD,ch);
      if  upcase(ch) = 'X' then
        return := 'X'  {abort program execution}
      else
        return := 'C';  {continue}
    end;

end;

{-------------------------------------------------------------------}

var i,j: integer;
    ch: char;
    last_event, event_count: integer;

begin  {event_checking_simulator}
```

```
clrscr;
gotoxy(27,2);
writeln('FSM EXECUTIVE SIMULATOR I');
writeln;
write(' Enter State Table filename:  ');
readln(filename);
writeln;
assign(f,filename);
reset(f);

{read state table file}
read(f, num_of_states, num_of_events, num_of_activities);
for i := 1 to num_of_states do
   for j := 1 to num_of_events do
      begin
         read(f,state_table[i,j,1],state_table[i,j,2]);
         readln(f);
      end;
act_time_table[0] := 0;

for i := 1 to num_of_activities do
   read(f,act_time_table[i]);

writeln(' PROBABILITY ASSIGNMENT');
for i := 1 to num_of_events do
 begin
   prob[i] := -1;
   repeat
     write (' Probability of event ',i, ' occurring:  ');
     readln (prob[i]);
   until (prob[i] >= 0) and (prob[i] <= 1);
 end;

writeln;
simulation_time := 0;
repeat
  write(' Enter desired  simulation length:  ');
  readln(simulation_time);
until (simulation_time > 0) and (simulation_time < 1e10);

for i := 1 to num_of_events do
   begin
      event_occ_count[i] := 0;
      event_det_count[i] := 0;
   end;

for i := 0 to num_of_activities do
  begin
    activity_count[i] := 0;
    activity_time_count[i] := 0;
  end;

writeln;
repeat
  write(' Enter current state of system:  ');
```

```
    readln(current_state);
  until (current_state > 0) and (current_state <= num_of_states);

  writeln;
  repeat
    write(' Do you want a timecount by timecount display?  [Y/N]:  ');
    readln(display);
    display:= upcase(display);
  until display in ['Y','N'];

  event_same_priority := false;
  writeln;
  repeat
    write(' Do all events have the same priority? [Y/N]: ');
    readln(ch);
  until upcase(ch) in ['Y','N'];
  if upcase(ch) = 'Y' then event_same_priority := true;

  if display = 'N' then
    begin
      clrscr;
      writeln;
      writeln(' Processing...');
    end;

  time := 0;
  return := ' ';
  activity := 0;
  last_event := 0;
  processor_busy := false;
  processor_busy_time := 0;

  while (time < simulation_time) and (return <> 'X') do
     begin

      if display = 'Y' then
         output_display;

      if return <> 'X' then
       begin

        time := time + 1;

        {Check for occurence of all events}
        event := last_event + 1;
        if event > num_of_events then event := 1;
        event_count := num_of_events;
        while (event_count <> 0) do
          begin
            if prob[event] > Random then
              begin
                event_occ_count[event] := event_occ_count[event] + 1;

                {Process event only if processor is "idle"}
                if processor_busy = false then
```

```
                    begin

                      activity := state_table[current_state,event,2];
                      next_state := state_table[current_state,event,1];
                      event_det_count[event] := event_det_count[event]
                                                + 1;
                      activity_count[activity] := activity_count
                                                  [activity] + 1;

                      if activity <> 0 then
                        begin
                          {Remember last event detected if all events
                           have the same priority, for a round robin
                           type polling.}
                          if event_same_priority = true
                             then last_event := event
                             else last_event := 0;
                          {Assign activity to processor}
                          processor_busy := true;
                          processor_busy_time := act_time_table
                                                 [activity];
                        end
                       else {Only update state if activity 0}
                          current_state := next_state;

                    end;

                end;
              event := event + 1;
              if event > num_of_events then event := 1;
              event_count := event_count - 1;
            end;

          {Simulate activity execution}
          if (processor_busy = true) then
           begin
             activity_time_count[activity] := activity_time_count
                                              [activity] + 1;
             processor_busy_time := processor_busy_time - 1;
             if processor_busy_time = 0 then
               begin
                 current_state := next_state;
                 processor_busy := false;
                 activity := 0;
               end;
           end;

        end;

      end;

    if display <> 'Y' then
      begin
        sound(720);
        delay(150);
```

```
            nosound;
        end;

    output_display;

end;
```

```
PROCEDURE INTERRUPT_DRIVEN_SIMULATOR;

    CONST
      max_queue_size = 1000;

    VAR
      num_of_states, num_of_events, num_of_activities: integer;
      state_table: array[1..max_n_of_states,1..max_n_of_events,1..2]
                   of integer;
      act_time_table: array[0..max_n_of_activities] of integer;
      simulation_time, time: real;
      event_occ_count, event_queued_count: array[1..max_n_of_events]
                                           of real;
      event_queued_time, event_service_time: array[1..max_n_of_events]
                                             of
                                             RECORD
                                               max: real;
                                               total: real;
                                             end;
      prob: array[1..max_n_of_events] of real;
      activity_count, activity_time_count: array[0..max_n_of_activities]
                                           of real;
      current_state, next_state, activity, event: integer;
      queue_size: integer;
      event_queue: RECORD
                     size: integer;
                     in_index:   integer;
                     out_index:  integer;
                     queue: array[1..max_queue_size] of
                            RECORD
                              entry: integer;
                              time: real;
                            end;
                   end;
      queue_stat: RECORD
                    max: integer;
                    size: real;
                  end;
      processor_busy: boolean;
      processor_busy_time: integer;
      display, return: char;
      f: text;
      filename: string[14];

{-------------------------------------------------------------------------}

 Procedure output_display;

 var
  i: integer;
  ch: char;
  choice: char;
  display_stat: boolean;
  percent_queued: real;
```

```
  busy_time: real;

{-------------------------------------------------------------------}

 Procedure display_event_stats;

 begin {display_event_stats}

  clrscr;
  gotoxy(32,2);
  write ('EVENT STATISTICS');
  gotoxy(1,4);
  writeln(' TIME: ',time:10:0);
  writeln;
  writeln(' Current State: ',current_state,'   Activity Executing: ',
          activity,'     Queue Size: ',event_queue.size);
  writeln;
  write(' EVENT NUMBER      OCCURRED        QUEUED  ');
  if display_stat then {add headings for final stats}
     write('   PROB OCCURRENCE     % QUEUED  ');
  writeln;
  writeln;

  for i := 1 to num_of_events do
     begin
       write(' ',i:6,event_occ_count[i]:18:0,event_queued_count[i]:15:0,
                '                ');
       if display_stat then {give final percentages}
         begin
          if event_queued_count[i] = 0 then
             percent_queued := 0
          else
             percent_queued := event_queued_count[i]/event_occ_count[i]
                               * 100;
          write(event_occ_count[i]/time:6:5,
                '          ',percent_queued:7:3);
         end;
       writeln;
     end;

     if display_stat then
       begin
        gotoxy(1,20);
        write(' Hit any key to continue statistics');
        while not keypressed do {nothing};
       end;

 end;

{-------------------------------------------------------------------}

 Procedure display_activity_stats;

   begin {display_activity_stats}
```

```
      clrscr;
      gotoxy(31,2);
      write ('ACTIVITY STATISTICS');
      gotoxy(1,4);
      writeln(' MAXIMUM QUEUE SIZE: ',queue_stat.max:4,' ':8,
              'AVERAGE QUEUE SIZE: ',(queue_stat.size/time):7:3);
      writeln;
      writeln(' ACTIVITY','ACTIVITY TIME':17,'# OF TIMES':13,
              'TOTAL TIME':15,'  % OF PROC TIME':20);
      writeln;
      busy_time := 0;
      for i := 0 to num_of_activities do
       begin
         busy_time := busy_time + activity_time_count[i];
         writeln('    ',i:3,act_time_table[i]:18,
                 activity_count[i]:13:0,activity_time_count[i]:15:0,
                 ' ':13,(activity_time_count[i]/time)*100.0:7:3);
       end;
      writeln;
      writeln('  TOTAL ',' ':34,busy_time:10:0,' ':13,
              (busy_time/time)*100.0:7:3);
      gotoxy(1,21);
      write(' Type any key to exit');
      while not keypressed do {nothing};

    end;

{-------------------------------------------------------------------------}

  Procedure display_response_stats;

  var avg_queued_time: real;
      avg_service_time: real;

  begin {display_response_stats}

   clrscr;
   gotoxy(28,2);
   write ('EVENT RESPONSE STATISTICS');
   gotoxy(1,4);
   writeln(' EVENT NUMBER      AVG TIME      MAX TIME   ',
           '    AVG TIME      MAX TIME ');
   writeln('                   IN QUEUE      IN QUEUE   ',
           '    IN SYSTEM     IN SYSTEM');
   writeln;

   for i := 1 to num_of_events do
      begin
        {Compute average times}
        if event_queued_time[i].total = 0
           then avg_queued_time := 0
           else avg_queued_time := event_queued_time[i].total
                                   /event_queued_count[i];
        if event_service_time[i].total = 0
           then avg_service_time := 0
```

```
        else avg_service_time := event_service_time[i].total
                                 /event_queued_count[i];
      write(' ',i:6,' ':12,
            avg_queued_time:7:3,' ':6,
            event_queued_time[i].max:5:0,' ':10,
            avg_service_time:7:3,' ':7,
            event_service_time[i].max:5:0);
      writeln;
    end;

    if display_stat then
      begin
        gotoxy(1,20);
        write(' Hit any key to continue statistics');
        while not keypressed do {nothing};
      end;

end;

{-------------------------------------------------------------------------}

begin {output_display}

 if (time = simulation_time) or (return = 'X') then
   begin
     display_stat := true;
     repeat
       clrscr;
       gotoxy(32,7);
       write ('STATISTICS MENU');
       gotoxy(25,10);
       write ('1  Display Event Statistics');
       gotoxy(25,12);
       write ('2  Display Activity Statistics');
       gotoxy(25,14);
       write ('3  Display Response Time Statistics');
       gotoxy(25,16);
       write ('4  Exit');
       gotoxy(25,19);
       write ('Enter Selection [1-4]: ');
       choice := ' ';
       repeat
         read (kbd,choice);
       until choice in ['1','2','3','4'];

       case choice of
         '1': display_event_stats;
         '2': display_activity_stats;
         '3': display_response_stats;
       end;
     until choice = '4';
   end
 else
   begin
     display_stat := false;
```

```
      display_event_stats;
      gotoxy(1,20);
      write(' Type <X> to exit, any other key to continue');
      while not keypressed do {nothing};
      read (KBD,ch);
      if  upcase(ch) = 'X' then
        return := 'X'  {abort program execution}
      else
        return := 'C';  {continue}
    end;

end;

{-------------------------------------------------------------------}

var i,j: integer;
    ch: char;
    queued_time: real;
    service_time: real;

begin  {interrupt_driven_simulator}

   clrscr;
   gotoxy(27,2);
   writeln('FSM EXECUTIVE SIMULATOR II');
   writeln;
   write(' Enter State Table filename:  ');
   readln(filename);
   writeln;
   assign(f,filename);
   reset(f);

   {read state table file}
   read(f, num_of_states, num_of_events, num_of_activities);
   for i := 1 to num_of_states do
      for j := 1 to num_of_events do
         begin
            read(f,state_table[i,j,1],state_table[i,j,2]);
            readln(f);
         end;
   act_time_table[0] := 0;

   for i := 1 to num_of_activities do
      read(f,act_time_table[i]);

   writeln(' PROBABILITY ASSIGNMENT');
   for i := 1 to num_of_events do
    begin
      prob[i] := -1;
      repeat
        write (' Probability of event ',i, ' occurring:  ');
        readln (prob[i]);
      until (prob[i] >= 0) and (prob[i] <= 1);
    end;
```

```
writeln;
simulation_time := 0;
repeat
  write(' Enter desired  simulation length:  ');
  readln(simulation_time);
until (simulation_time > 0) and (simulation_time < 1e10);

writeln;
queue_size := -1;
repeat
  write(' Enter desired queue length (max ',max_queue_size,'): ');
  readln(queue_size);
until (queue_size >= 0) and (queue_size <= max_queue_size);

for i := 1 to num_of_events do
   begin
      event_occ_count[i] := 0;
      event_queued_count[i] := 0;
      event_queued_time[i].max := 0;
      event_queued_time[i].total := 0;
      event_service_time[i].max := 0;
      event_service_time[i].total := 0;
   end;

for i := 0 to num_of_activities do
  begin
    activity_count[i] := 0;
    activity_time_count[i] := 0;
  end;

writeln;
repeat
  write(' Enter current state of system:  ');
  readln(current_state);
until (current_state > 0) and (current_state <= num_of_states);

writeln;
repeat
   write(' Do you want a timecount by timecount display?  [Y/N]:  ');
   readln(display);
   display:= upcase(display);
until display in ['Y','N'];

if display = 'N' then
  begin
    clrscr;
    writeln;
    writeln(' Processing...');
  end;

time := 0;
return := ' ';
activity := 0;
event_queue.size := 0;
event_queue.in_index := 1;
```

```
event_queue.out_index := 1;
queue_stat.max := 0;
queue_stat.size := 0;
processor_busy := false;
processor_busy_time := 0;

while (time < simulation_time) and (return <> 'X') do
   begin

    if display = 'Y' then
       output_display;

    if return <> 'X' then
     begin
      time := time + 1;
      event := 0;

      (Check for events and queue them)
      for i := 1 to num_of_events do
        begin
          if prob[i] > Random then
            begin
              event_occ_count[i] := event_occ_count[i] + 1;
              if event_queue.size < queue_size then
                begin
                  event_queued_count[i] := event_queued_count[i]
                                           + 1;
                  event_queue.queue[event_queue.in_index].entry
                                        := i;
                  event_queue.queue[event_queue.in_index].time :=
                                        time;
                  event_queue.in_index := event_queue.in_index + 1;
                  if event_queue.in_index > queue_size
                     then event_queue.in_index := 1;
                  event_queue.size := event_queue.size + 1;
                end;
            end;
        end;

      (Execute an activity in the queue if possible)
      while (processor_busy = false) and (event_queue.size <> 0) do
       begin
         (Unqueue event)
         event := event_queue.queue[event_queue.out_index].entry;

         (Update queue time statistics)
         queued_time := time - event_queue.queue
                               [event_queue.out_index].time;
         if event_queued_time[event].max < queued_time
            then event_queued_time[event].max := queued_time;
         event_queued_time[event].total := event_queued_time[event]
                                           .total + queued_time;

         (Update out pointer)
         event_queue.out_index := event_queue.out_index + 1;
```

```
            if event_queue.out_index > queue_size
               then event_queue.out_index := 1;
            event_queue.size := event_queue.size - 1;

            {Find out which activity to execute}
            activity := state_table[current_state,event,2];
            activity_count[activity] := activity_count[activity] + 1;
            next_state := state_table[current_state,event,1];

            {If activity is 0, simply change state, else assign
             to processor}
            if activity <> 0 then
              begin
                processor_busy := true;
                processor_busy_time := act_time_table[activity];
                service_time := queued_time + processor_busy_time;
              end
             else
              begin
                current_state := next_state;
                service_time := queued_time;
              end;

            {Update service time statistics}
             if event_service_time[event].max < service_time
                then event_service_time[event].max := service_time;
             event_service_time[event].total := event_service_time[event]
                                                .total + service_time;

          end;

        {Simulate activity execution}
        if processor_busy = true then
          begin
            activity_time_count[activity] := activity_time_count
                                             [activity] + 1;
            processor_busy_time := processor_busy_time - 1;
            if processor_busy_time = 0 then
              begin
                current_state := next_state;
                processor_busy := false;
                activity := 0;
              end;
          end;

        {Check queue statistics}
        if event_queue.size > queue_stat.max
           then queue_stat.max := event_queue.size;
        queue_stat.size := queue_stat.size + event_queue.size;

      end;

    end;

if display <> 'Y' then
```

```
      begin
       sound(720);
       delay(150);
       nosound;
      end;

   output_display;

end;
```

APPENDIX B

ACTIVITY SEQUENCE NET SIMULATOR

This appendix contains the program listing for the activity sequence net simulator written in TURBO PASCAL (c) for the IBM Personal Computer (c).

TURBO PASCAL (c) is a trade mark of Borland International Inc. IBM Personal Computer (c) is a trade mark of International Business Machines Corp.

TABLE OF CONTENTS OF APPENDIX B

```
{Activity Sequence Nets System declarations}

CONST max_number_of_nets = 10;
      max_number_of_activities = 50;
      max_number_of_ext_events = 20;

      max_number_of_nodes_per_net = 20;
      max_number_of_events_per_net = 30;

      {Valid entries for ASN tables}
      not_connected  = '0';
      input_trigger  = 'I';
      enable         = 'E';
      disable        = 'D';

{*****************************************************************}

TYPE node_type = (activity_initiator, wait, merge, branch, gate,
                  counter, terminator);

     event_type = (int_event, ext_event); {internal event (node trigger)
                                           or external event}

     counter_status = (counter_expired, counter_disabled);

     node_definition = RECORD
                       CASE node: node_type of
                         activity_initiator:
                             (activity_id: 1..max_number_of_activities);
                         wait: ();
                         merge: ();
                         branch: (branch_variable_id: string[1]);
                         gate: ();
                         counter: (counter_variable_id: string[1]);
                         terminator: ();
                       end;

     event_definition = RECORD
                        CASE event: event_type of
                          ext_event: (event_number:
                                      1..max_number_of_ext_events);
                          int_event: (node_number:
                                      1..max_number_of_nodes_per_net;
                                      CASE node: node_type of
                                        activity_initiator: ();
                                        wait: ();
                                        merge: ();
                                        branch: (branch_condition:
                                                 boolean);
                                        gate: ();
                                        counter: ();
                                        terminator: ());
                        end;
```

```
    matrix_definition = RECORD
                        number_of_events:
                              1..max_number_of_events_per_net;
                        number_of_nodes:
                              1..max_number_of_nodes_per_net;
                        matrix: array [1..max_number_of_events_per_net,
                                       1..max_number_of_nodes_per_net]
                                  of string[1];
                        end;

{*************************************************************************}

{ Global variable declarations }

VAR system_name: string[8];

    number_of_nets: 1..max_number_of_nets;
    number_of_activities: 1..max_number_of_activities;
    number_of_ext_events: 1..max_number_of_ext_events;

    asn_starting_event: array [1..max_number_of_nets]
                        of 1..max_number_of_ext_events;

    asn_node_definition: array [1..max_number_of_nets,
                                1..max_number_of_nodes_per_net]
                         of node_definition;

    asn_event_definition: array [1..max_number_of_nets,
                                 1..max_number_of_events_per_net]
                          of event_definition;

    asn_matrix: array [1..max_number_of_nets] of matrix_definition;
```

```
{Activity to Processor Allocation declarations}

CONST max_number_of_processors = 10;
      max_n_activities_per_processor = 20;

{************************************************************************}

TYPE processor_activity_table_def = RECORD
                                      activity_id:
                                           1..max_number_of_activities;
                                      activity_execution_time: integer;
                                      end;

{************************************************************************}

{ Global variable declarations }

VAR number_of_processors: 1..max_number_of_processors;

    allocation_identifier: string[1];

    processor_activity_table_size: array [1..max_number_of_processors]
                                   of 1..max_n_activities_per_processor;

    processor_activity_table: array [1..max_number_of_processors,
                                     1..max_n_activities_per_processor]
                              of processor_activity_table_def;
```

```
{ ASN Simulator declarations }

CONST max_simulation_time = 30000;
      max_ready_to_run_queue_length = 30;
      max_number_of_counters = 10;
      max_number_of_branch_nodes = 20;
      max_num_of_simultaneous_events = 10;

{*****************************************************************}

TYPE error_type = (no_error,
                   ready_queue_overflow,
                   no_enable_input,
                   no_internal_event,
                   too_many_simultaneous_events,
                   too_many_branch_variables,
                   too_many_counter_variables,
                   branch_prob_not_found,
                   counter_variable_not_found);

     network_state = (dormant,awake);
     trigger_state = (off,on);
     processor_state = (idle,busy);

     network_information = RECORD
                           state: network_state;
                           state_table: array
                                [1..max_number_of_events_per_net,
                                 1..max_number_of_nodes_per_net]
                                       of trigger_state;
                           end;

     ready_to_run_info = RECORD
                         activity_id: 1..max_number_of_activities;
                         network_id: 1..max_number_of_nets;
                         node_number: 1..max_number_of_nodes_per_net;
                         end;

{*****************************************************************}

VAR time: 0..max_simulation_time;
    simulation_time: 0..max_simulation_time;
    simulation_state: (started,finished);
    simulation_error: error_type;
    simulation_mode: (interactive,continuous);
    display_flag: boolean;
    display_delay: integer;

    clock_event: 1..max_number_of_ext_events;

    {Probability of event for continuous mode of operation only}
    probability_of_event: array [1..max_number_of_ext_events] of real;
```

```
{Event vector and state tables associated variables}
event_vector_state: array [1..max_number_of_nets]
                    of (no_change,updated);
event_vector: array [1..max_number_of_nets,
                     1..max_number_of_events_per_net]
                     of trigger_state;
network_info: array [1..max_number_of_nets] of
                     network_information;

{Branch node variables for continuous mode of operation}
number_of_branch_nodes: integer;
branch_probability: array [1..max_number_of_branch_nodes] of
                           RECORD
                           network_id: 1..max_number_of_nets;
                           node_number:
                              1..max_number_of_nodes_per_net;
                           prob_true: real;
                           end;

{Counter variables associated variables}
number_of_counters: 0..max_number_of_counters;
counters_initialization: (prompt_user,once_at_begining);
counter_variables: array [1..max_number_of_counters] of
                           RECORD
                           network_id: 1..max_number_of_nets;
                           node_number:
                               1..max_number_of_nodes_per_net;
                           status: (not_initialized,initialized);
                           initial_value: integer;
                           value: integer;
                           end;
counter_stat: array [1..max_number_of_counters] of
                    RECORD
                    n_times_started: integer;
                    n_times_expired: integer;
                    end;

{Ready to run queue variables}
ready_to_run_queue_length: 0..max_ready_to_run_queue_length;
ready_to_run_queue: array [1..max_ready_to_run_queue_length]
                    of ready_to_run_info;

processor_info: array [1..max_number_of_processors] of
                      RECORD
                      state: processor_state;
                      activity_id: 1..max_number_of_activities;
                      network_id: 1..max_number_of_nets;
                      node_number: 1..max_number_of_nodes_per_net;
                      execution_time_left: integer;
                      end;

{ Variables used for statistic gathering purposes }
processor_activity_stat_time: array [1..max_number_of_processors,
                                     1..max_number_of_activities]
```

```
                                    of integer;
event_statistics: array [1..max_number_of_ext_events] of
                  RECORD
                  number_of_occurrences: integer;
                  number_of_rejections: integer;
                  end;
network_statistics: array [1..max_number_of_nets] of
                    RECORD
                    number_of_times: integer;
                    cumulative_completion_time: integer;
                    max_completion_time: integer;
                    completion_time: integer;
                    end;
cumulative_queue_length_before: real;
cumulative_queue_length_after:  real;
max_queue_length_before: integer;
max_queue_length_after:  integer;
```

```
PROGRAM SYSTEM_SPECIFICATION (input,output);

{Compiler Directives}
{$R+}

{*****************************************************************}

{$I DEFASN.DEC}

VAR color_display: boolean;

{*****************************************************************}

PROCEDURE GET_SYSTEM_PARAMETERS;

   VAR answer: string[1];
       number: integer;

   BEGIN { Get_system_parameters }

      color_display := false;
      writeln;
      write (' Do you have a color monitor (Y/N) ? ');
      read (kbd,answer);
      if (answer = 'Y') or (answer = 'y') then
          color_display := true;

      if color_display then
        begin
          TextColor(yellow);
          TextBackground(blue);
        end;
      clrscr;
      writeln;
      writeln (' DEFINE ACTIVITY SEQUENCE NETS PROGRAM');
      writeln ('                                  written by Robert Joannis');
      writeln;
      writeln ('    This program permits entry of the description of',
               ' the');
      writeln (' system specified using activity sequence nets. This');
      writeln (' description will be used as input to the ASN',
               ' executive');
      writeln (' simulator.');
      writeln;
      write (' Enter system name (maximum 8 characters): ');
      readln (system_name);

      number := 0;
      write (' Enter the number of activity sequence networks in',
             ' the system: ');
      readln (number);
      while (number > max_number_of_nets) or
            (number <= 0) do
```

```
        begin
          writeln (' The number of networks must be between 1 and ',
                   max_number_of_nets:2,'.');
          write (' Re-enter number of networks: ');
          readln (number);
        end;
      number_of_nets := number;

      number := 0;
      write (' Enter the number of activities in the system: ');
      readln (number);
      while (number > max_number_of_activities) or
            (number <= 0) do
        begin
          writeln (' The number of activities must be between 1 and ',
                   max_number_of_activities:2,'.');
          write (' Re-enter number of activities: ');
          readln (number);
        end;
      number_of_activities := number;

      number := 0;
      write (' Enter the number of external events in the system: ');
      readln (number);
      while (number > max_number_of_ext_events) or
            (number <= 0) do
        begin
          writeln (' The number of external events must be between 1',
                   ' and ',max_number_of_ext_events:2,'.');
          write (' Re-enter number of external events: ');
          readln (number);
        end;
      number_of_ext_events := number;

   END;

{*************************************************************************}

PROCEDURE GET_ASN_MATRICES;

   VAR net_number: 1..max_number_of_nets;

{-------------------------------------------------------------------------}

   PROCEDURE GET_STARTING_EVENT;

      VAR  number: integer;

      BEGIN { Get_starting_event }

         clrscr;
         writeln;
         number := 0;
         write (' Enter starting event for activity sequence network ',
                net_number:2,': ');
```

```
            readln (number);
            while (number > number_of_ext_events) or
                  (number <= 0) do
              begin
                 writeln (' Invalid event number. Must be between 1 and ',
                          number_of_ext_events:2,' .');
                 write (' Re-enter event number: ');
                 readln (number);
              end;
            asn_starting_event[net_number] := number;

        END;

{-------------------------------------------------------------------------}

    PROCEDURE GET_NODE_DEFINITION;

      VAR node_number: 1..max_number_of_nodes_per_net;
          character: string[1];
          number: integer;
          answer_status: (invalid,valid);

{-------------------------------------------------------------------------}

       PROCEDURE DISPLAY_VALID_NODE_TYPES;

          BEGIN { Display_valid_nodes }

             clrscr;
             writeln;
             writeln (' The valid node types are:');
             writeln;
             writeln (' A: Activity Initiator');
             writeln (' W: Wait');
             writeln (' M: Merge');
             writeln (' B: Branch');
             writeln (' G: Gate');
             writeln (' C: Counter');
             writeln (' E: End');
             writeln;
             window (1,13,80,25);

          END;

{-------------------------------------------------------------------------}

       BEGIN { Get_node_definition }

          number := 0;
          writeln;
          write (' How many nodes in ASN of activity sequence network ',
                 net_number:2,' ? ');
          readln (number);
          while (number > max_number_of_nodes_per_net) or
                (number <= 1) do
```

```
      begin
         writeln (' Invalid number of nodes. Must be between 2 ',
                  'and ',max_number_of_nodes_per_net:2,'.');
         write (' Re-enter number of nodes: ');
         readln (number);
      end;
   asn_matrix[net_number].number_of_nodes := number;

   {Get description of each node in the net}
   display_valid_node_types;
   for node_number := 1 to asn_matrix[net_number].number_of_nodes
    do
      begin

         answer_status := invalid;
         while answer_status = invalid do
           begin

             write (' Enter type of node ',node_number:2,': ');
             readln (character);

             {Assume answer is valid}
             answer_status := valid;

             If (character = 'A') or (character = 'a') then
                begin
                   asn_node_definition[net_number,node_number]
                           .node := activity_initiator;
                   number := 0;
                   write (' Enter activity number: ');
                   readln (number);
                   while (number > number_of_activities) or
                         (number <= 0) do
                      begin
                         writeln (' Invalid activity number.',
                                  ' Must be ','between 1 and ',
                                  number_of_activities:2);
                         write (' Re-enter activity number: ');
                         readln (number);
                      end;
                   asn_node_definition[net_number,node_number]
                           .activity_id := number;
                end

             else if (character = 'W') or (character = 'w') then
                begin
                   asn_node_definition[net_number,node_number]
                           .node := wait;
                end

             else if (character = 'M') or (character = 'm') then
                begin
                   asn_node_definition[net_number,node_number]
                           .node := merge;
                end
```

```
                    else if (character = 'B') or (character = 'b') then
                       begin
                          asn_node_definition[net_number,node_number]
                                   .node := branch;
                          write (' Enter variable name (1 letter): ');
                          readln (asn_node_definition[net_number,
                                       node_number].branch_variable_id);
                       end

                    else if (character = 'G') or (character = 'g') then
                       begin
                          asn_node_definition[net_number,node_number]
                                   .node := gate;
                       end

                    else if (character = 'C') or (character = 'c') then
                       begin
                         asn_node_definition[net_number,node_number]
                                  .node := counter;
                         write (' Enter variable name (1 letter): ');
                         readln (asn_node_definition[net_number,
                                      node_number].counter_variable_id);
                       end

                    else if (character = 'E') or (character = 'e') then
                       begin
                         asn_node_definition[net_number,node_number]
                                  .node := terminator;
                       end

                    else
                       {A valid node type was not entered}
                       begin
                          answer_status := invalid;
                          writeln (' Invalid node type.');
                       end;

                 end;
           end;

        { Reset window }
        window (1,1,80,25);

     END;

{-------------------------------------------------------------------------}

  PROCEDURE GET_EVENT_DEFINITION;

     VAR event_number: 1..max_number_of_events_per_net;
         node_number:  1..max_number_of_nodes_per_net;
         node: node_type;
         space_left: 1..max_number_of_events_per_net;
         ext_event_number: 1..max_number_of_ext_events;
```

```
    n_of_ext_events_for_this_net: 1..max_number_of_ext_events;
    number: integer;

BEGIN { Get_event_definition }

  event_number := 1;
  for node_number := 1 to asn_matrix[net_number].number_of_nodes
   do
    begin

      node := asn_node_definition[net_number,node_number]
                .node;

      {Do not include terminator node in event definition}
      if node <> terminator then
        begin
          asn_event_definition[net_number,event_number]
                .event := int_event;
          asn_event_definition[net_number,event_number]
                .node_number := node_number;
          asn_event_definition[net_number,event_number]
                .node := node;

          if node = branch then
            begin

              {Setup true branch condition}
              asn_event_definition[net_number,event_number]
                    .branch_condition := true;

              {Setup false branch condition}
              if event_number < max_number_of_events_per_net
                 then event_number := event_number + 1
                 else writeln (' WARNING: too many events...');
              asn_event_definition[net_number,event_number]
                    .event := int_event;
              asn_event_definition[net_number,event_number]
                    .node_number := node_number;
              asn_event_definition[net_number,event_number]
                    .node := branch;
              asn_event_definition[net_number,event_number]
                    .branch_condition := false;
            end;

          if event_number < max_number_of_events_per_net
             then event_number := event_number + 1
             else writeln (' WARNING: too many events...');

        end;

    end;

    number := 0;
    writeln;
    write (' Enter number of external events for this network: ');
```

```
readln (number);

{Find out maximum number of events for this network}
space_left := max_number_of_events_per_net - event_number + 1;
{Make sure it does not exceed number of events in system}
if space_left > number_of_ext_events then
                space_left := number_of_ext_events;

while (number > space_left) or
      (number <= 0) do
   begin
      writeln (' The number of events for this network must',
               ' be between 1 and ',space_left:2,'.');
      write (' Re-enter number of external events: ');
      readln (number);
   end;
n_of_ext_events_for_this_net := number;

{Setup event definition of starting event}
asn_event_definition[net_number,event_number]
         .event := ext_event;
asn_event_definition[net_number,event_number]
         .event_number := asn_starting_event[net_number];
event_number := event_number + 1;
writeln (' The starting event has been entered in the',
         ' definition.');

ext_event_number := 2;
while ext_event_number <= n_of_ext_events_for_this_net do
  begin
     number := 0;
     write (' Enter external event number (excluding ',
            'starting event): ');
     readln (number);
     while (number > number_of_ext_events) or
           (number <= 0) do
        begin
           writeln (' Invalid event number. Must be between',
                    ' 1 and ',number_of_ext_events:2,'.');
           write (' Re-enter external event number: ');
           readln (number);
        end;

     {Setup event definition for external event using event
      number}
     asn_event_definition[net_number,event_number]
              .event := ext_event;
     asn_event_definition[net_number,event_number]
              .event_number := number;

     event_number := event_number + 1;
     ext_event_number := ext_event_number + 1;
  end;

asn_matrix[net_number].number_of_events := event_number - 1;
```

```
      END;

{-------------------------------------------------------------------------}

   PROCEDURE GET_MATRIX_ENTRIES;

      VAR row:    0.. max_number_of_events_per_net;
          column: 1..max_number_of_nodes_per_net;

{-------------------------------------------------------------------------}

      PROCEDURE DISPLAY_VALID_TRIGGER_TYPES;

      BEGIN { Display_valid_trigger_types }

         clrscr;
         writeln;
         writeln (' The valid input trigger types are:');
         writeln;
         writeln (' E: external event');
         writeln (' N: node output trigger');
         writeln (' X: no more input triggers');
         writeln;
         window (1,10,80,25);

      END;

{-------------------------------------------------------------------------}

      PROCEDURE GET_NEXT_TRIGGER;

      VAR answer_status: boolean;
          char: string[1];
          event: event_type;

{-------------------------------------------------------------------------}

         PROCEDURE SEARCH_EVENT_DEFINITION;

         VAR search_index: integer;
             search_flag: boolean;
             number: integer;

         BEGIN { Search_event_definition }

             number := 0;
             write (' Enter event or node number: ');
             readln (number);

             search_flag := false;
             search_index := 1;

             while (search_index <= asn_matrix[net_number]
                    .number_of_events) and (search_flag = false) do
```

```
begin

   {Check for proper event type}
   if asn_event_definition[net_number,search_index]
         .event = event then
     begin
        case event of
          ext_event:
            begin
               {Check for proper event number, if a
                match set search flag to true to exit
                the loop}
               if asn_event_definition[net_number,
                     search_index].event_number = number
                     then search_flag := true;
             end;

          int_event:
            begin
               {Check for proper node number, if a match
                set search flag to true to exit the loop}
               if asn_event_definition[net_number,
                     search_index].node_number = number
                     then
                 begin
                    search_flag := true;
                    {If branch node check for true
                     or false condition}
                    if asn_event_definition[net_number,
                          search_index].node = branch
                          then
                      begin
                         write (' Enter "T" for true',
                                ' or "F" for false',
                                ' branch condition: ');
                         readln (char);

                         {If true condition then it must
                          be the next entry in event
                          definition}
                         if (char = 'F') or (char = 'f')
                           then search_index :=
                                search_index + 1;
                      end;
                 end;

             end; {end case of event_type}
        end;
     end;

   search_index := search_index + 1;
end; {end while}

{Check if the input trigger was found in the event
 definition}
```

```
            if search_flag = true then
              {the answer was OK and the row is found}
              begin
                 answer_status := true;
                 row := search_index - 1;
              end

            else
               writeln (' INVALID trigger. Re-enter.');

      END;

   BEGIN { Get_next_trigger }

      answer_status := false;
      while answer_status = false do
        begin
           write (' Enter input trigger type: ');
           readln (char);

           if (char = 'E') or (char = 'e') then
              begin
                 event := ext_event;
                 search_event_definition;
              end

           else if (char = 'N') or (char = 'n') then
              begin
                 event := int_event;
                 search_event_definition;
              end

           else if (char = 'X') or (char = 'x') then
              begin
                 row := 0;
                 answer_status := true;
              end

           else
              begin
                 writeln (' INVALID trigger type');
              end;
        end;
   END;

{-------------------------------------------------------------------}

   PROCEDURE GET_NORMAL_NODE_COLUMN;

      BEGIN { Get_normal_node_column }
         repeat
            get_next_trigger;
            if row <> 0 then
               asn_matrix[net_number].matrix[row,column]
                        := input_trigger;
```

```
        until row = 0;
      END;

{-------------------------------------------------------------------}

    PROCEDURE GET_GATE_NODE_COLUMN;

      BEGIN { Get_gate_node_column }

        writeln (' Enter ENABLE trigger for the gate.');
        get_next_trigger;
        while row = 0 do
          begin
            writeln (' You must have an ENABLE trigger');
            get_next_trigger;
          end;
        asn_matrix[net_number].matrix[row,column] := enable;

        writeln (' Enter DISABLE trigger for the gate.');
        get_next_trigger;
        while row = 0 do
          begin
            writeln (' You must have a DISABLE trigger');
            get_next_trigger;
          end;
        asn_matrix[net_number].matrix[row,column] := disable;

        writeln (' Enter INPUT trigger for the gate.');
        get_next_trigger;
        while row = 0 do
          begin
            writeln (' You must have an INPUT trigger');
            get_next_trigger;
          end;
        asn_matrix[net_number].matrix[row,column] := input_trigger;

      END;

{-------------------------------------------------------------------}

    PROCEDURE GET_COUNTER_NODE_COLUMN;

      BEGIN { Get_counter_node_column }

        writeln (' Enter ENABLE trigger for the counter.');
        get_next_trigger;
        while row = 0 do
          begin
            writeln (' You must have an ENABLE trigger');
            get_next_trigger;
          end;
        asn_matrix[net_number].matrix[row,column] := enable;

        writeln (' Enter DISABLE trigger for the counter.');
        get_next_trigger;
```

```
            while row = 0 do
              begin
                 writeln (' You must have a DISABLE trigger');
                 get_next_trigger;
              end;
            asn_matrix[net_number].matrix[row,column] := disable;

            writeln (' Enter INPUT trigger for the counter.');
            get_next_trigger;
            while row = 0 do
              begin
                 writeln (' You must have an INPUT trigger');
                 get_next_trigger;
              end;
            asn_matrix[net_number].matrix[row,column] := input_trigger;

         END;

{-------------------------------------------------------------------------}


      BEGIN { Get_matrix_entries }

         {Initialize ASN matrix to not connected}
         for row := 1 to asn_matrix[net_number].number_of_events do
           begin
              for column := 1 to asn_matrix[net_number].number_of_nodes
               do
                asn_matrix[net_number].matrix[row,column] :=
                           not_connected;
           end;

         display_valid_trigger_types;

         {Prompt for entry column by column}
         for column := 1 to asn_matrix[net_number].number_of_nodes do
           begin
              writeln;
              writeln (' Enter input triggers for node ',column:2,'.');
              CASE asn_node_definition[net_number,column].node of
                activity_initiator: get_normal_node_column;
                wait: get_normal_node_column;
                merge: get_normal_node_column;
                branch: get_normal_node_column;
                gate: get_gate_node_column;
                counter: get_counter_node_column;
                terminator: get_normal_node_column;
              end;
           end;

         { Reset window }
         window (1,1,80,25);

      END;
```

```
{-------------------------------------------------------------------------}

   BEGIN { Get_asn_matrix }

      for net_number := 1 to number_of_nets do
        begin
           get_starting_event;
           get_node_definition;
           get_event_definition;
           get_matrix_entries;
        end;

   END;

{*************************************************************************}

PROCEDURE SAVE_SYSTEM_DEFINITION;

   VAR char: string[1];
       file_name: string[14];
       f: text;
       net_number: integer;
       i: integer;
       j: integer;

   BEGIN

      clrscr;
      writeln;
      writeln (' Insert data disk in drive B.');
      write (' Do you want to save the system definition on disk',
             ' (Y/N)? ');
      readln (char);

      if (char = 'Y') or (char = 'y') then
        begin

           {Form file name using system name}
           file_name := 'b:' + system_name + '.asn';

           {Open the file}
           ASSIGN (f,file_name);
           rewrite (f);

           {Save the system definition}
           writeln (f,number_of_nets);
           writeln (f,number_of_activities);
           writeln (f,number_of_ext_events);

           for net_number := 1 to number_of_nets do
             begin

               {write starting event}
               writeln (f,asn_starting_event[net_number]);
```

```
{write asn matrix}
writeln (f,asn_matrix[net_number].number_of_nodes);
writeln (f,asn_matrix[net_number].number_of_events);
for i := 1 to asn_matrix[net_number].number_of_events do
  begin
     for j := 1 to asn_matrix[net_number]
                        .number_of_nodes do
           write (f,asn_matrix[net_number].matrix[i,j]);
  end;
writeln (f);

{write asn node definition}
for i := 1 to asn_matrix[net_number].number_of_nodes do
  begin
    writeln (f,integer(asn_node_definition[net_number,i]
                                    .node));
    case asn_node_definition[net_number,i].node of
      activity_initiator:
              writeln (f,asn_node_definition[net_number,
                          i].activity_id);
      branch: writeln (f,asn_node_definition[net_number,
                          i].branch_variable_id);
      counter: writeln (f,asn_node_definition[net_number,
                           i].counter_variable_id);
    end;
  end;

{write asn event definition}
for i := 1 to asn_matrix[net_number].number_of_events do
  begin

    writeln (f,integer(asn_event_definition[net_number,i]
                                .event));
    case asn_event_definition[net_number,i].event of
      ext_event: writeln (f,asn_event_definition
                             [net_number,i].event_number);
      int_event:
          begin
             writeln (f,asn_event_definition[net_number,
                         i].node_number);
             writeln (f,integer(asn_event_definition
                         [net_number,i].node));
             if asn_event_definition[net_number,i]
                         .node = branch then
                  writeln (f,integer(asn_event_definition
                             [net_number,i]
                             .branch_condition));
          end;
    end; {end case}

  end;


end; {end for loop on net_number}
```

```
            CLOSE (F);
            writeln;
            writeln (' The system definition has been saved in the',__
                     ' file: ',file_name);
            writeln;

          end;

    END;

{*************************************************************************}

BEGIN { Main Program }

  get_system_parameters;
  get_asn_matrices;
  save_system_definition;

END.
```

```
PROGRAM ACTIVITY_TO_PROCESSOR_ALLOCATION (input,output);


{Compiler Directives}
{$R+}

{*************************************************************************}

{$I DEFASN.DEC}
{$I DEFALL.DEC}

VAR color_display: boolean;

{*************************************************************************}

PROCEDURE GET_SYSTEM_NAME;

   VAR answer: string[1];

   BEGIN { Get_system_parameters }

      color_display := false;
      writeln;
      write (' Do you have a color monitor (Y/N) ? ');
      read (kbd,answer);
      if (answer = 'Y') or (answer = 'y') then
         color_display := true;

      if color_display then
        begin
          TextColor(yellow);
          TextBackground(blue);
        end;
      clrscr;
      writeln;
      writeln (' ACTIVITY TO PROCESSOR ALLOCATION PROGRAM');
      writeln ('                                written by Robert Joannis');
      writeln;
      writeln ('    This program permits entry of the allocation of',
               ' the');
      writeln (' activities to processors of a system which has',
               ' already ');
      writeln (' been defined using DEFASN.');
      writeln;
      write (' Enter system name (maximum 8 characters): ');
      readln (system_name);
      writeln;
      write (' Enter allocation idendifier (only 1 character): ');
      readln (allocation_identifier);

   END;

{*************************************************************************}
```

```
{ EXTERNAL PROCEDURES }

{$I RDASN.PAS}

{*****************************************************************}

PROCEDURE GET_PROCESSOR_ACTIVITY_TABLES;

   VAR processor_number: 1..max_number_of_processors;
       max_activities_per_processor: 1..max_n_activities_per_processor;
       activity_id: 0..max_number_of_activities;
       number: integer;
       index: integer;

   BEGIN { Get_processor_activity_tables }

      clrscr;

      {Find out number of processors in the system}
      number := 0;
      writeln;
      write (' Enter the number of processors in the system: ');
      readln (number);
      while (number > max_number_of_processors) or
            (number <= 0) do
         begin
           writeln (' The number of processors must be between 1 and ',
                    max_number_of_processors:2,'.');
           write (' Re-enter number of processors: ');
           readln (number);
         end;
      number_of_processors := number;

      {Find out maximum number of activities that can be executed
       by a processor}
      if number_of_activities > max_n_activities_per_processor
         then max_activities_per_processor :=
              max_n_activities_per_processor
         else max_activities_per_processor := number_of_activities;

      processor_number := 1;
      while processor_number <= number_of_processors do
        begin

          {Find out how many activities can be executed by that
           processor}
          number := 0;
          writeln;
          write (' How many activities can processor ',
                 processor_number:2,' execute ? ');
          readln (number);
          while (number > max_activities_per_processor) or
                (number <= 0) do
            begin
               writeln (' A processor must be able to execute',
```

```
                        ' between 1 and ',
                        max_activities_per_processor:2,
                        ' activities.');
                write (' Re-enter number of activities: ');
                readln (number);
              end;
            processor_activity_table_size[processor_number]
                              := number;

            {Find out the activity id (activity number) of the activities
             that can be executed on that processor}
            index := 1;
            while index <= number do
              begin
                activity_id := 0;
                write (' Enter activity id of executable activity: ');
                readln (activity_id);
                while (activity_id > number_of_activities) or
                      (activity_id <= 0) do
                  begin
                    writeln (' Invalid activity id. Must be a number',
                             ' between 1 and ',
                             number_of_activities:2,' .');
                    write (' Re-enter activity id: ');
                    readln (activity_id);
                  end;

                processor_activity_table[processor_number,index]
                         .activity_id := activity_id;

                write (' Enter execution time of activity ',
                       activity_id:2,': ');
                readln (processor_activity_table[processor_number,index]
                                .activity_execution_time);
                index := index + 1;
              end;

            processor_number := processor_number + 1;

        end;

    END;

{*****************************************************************}

PROCEDURE SAVE_ALLOCATION_DEFINITION;

    VAR char: string[1];
        file_name: string[14];
        f: text;
        i: integer;
        j: integer;

    BEGIN
```

```
      clrscr;
      writeln;
      writeln (' Insert data disk in drive B.');
      write (' Do you want to save the allocation definition on disk',
             ' (Y/N)? ');
      readln (char);

      if (char = 'Y') or (char = 'y') then
        begin

          {Form file name using system name}
          file_name := 'b:' + system_name + '.al' +
                       allocation_identifier;

          {Open the file}
          ASSIGN (f,file_name);
          rewrite (f);

          writeln (f,number_of_processors);

            for j := 1 to number_of_processors do
              begin
                {write processor activity tables}
                writeln (f,processor_activity_table_size[j]);
                for i := 1 to processor_activity_table_size[j] do
                begin
                  writeln (f,processor_activity_table[j,i]
                              .activity_id);
                  writeln (f,processor_activity_table[j,i]
                              .activity_execution_time);
                end;
              end;

          CLOSE (f);
          writeln;
          writeln (' The allocation definition has been saved in',
                   ' the file: ',file_name);
          writeln;

        end;

    END;

{*****************************************************************}

BEGIN { Main Program }

  get_system_name;
  read_system_specifications;
  get_processor_activity_tables;
  save_allocation_definition;

END.
```

```
PROGRAM ACTIVITY_SEQUENCE_NETWORK_SIMULATOR (input,output);

{*************************************************************************}

{ Compiler directives }
{$R+}

{*************************************************************************}

{$I DEFASN.DEC }
{$I DEFALL.DEC }
{$I ASNSIM.DEC }

VAR color_display: boolean;
    answer: char;
    first_time: boolean;

{*************************************************************************}

  PROCEDURE GET_SYSTEM_NAME;

    VAR answer: string[1];

    BEGIN { Get_system_name }

     if first_time then
      begin
        color_display := false;
        writeln;
        write (' Do you have a color monitor (Y/N) ? ');
        read (kbd,answer);
        if (answer = 'Y') or (answer = 'y')
           then color_display := true;

        if color_display then
          begin
            TextColor (yellow);
            TextBackground (blue);
          end;

        clrscr;
        writeln;
        writeln (' ASN SIMULATOR PROGRAM');
        writeln ('                                written by Robert',
                 ' Joannis');
        writeln;
        writeln ('    This program will simulate the behavior of a',
                 ' system');
        writeln (' that operates with an event driven executive',
                 ' based on');
        writeln (' Activity Sequence Nets. The definition of the',
                 ' system');
        writeln (' must first be entered using the program: DEFASN',
                 ' and the');
```

```
        writeln (' allocation specified using the program: DEFALL.');
        writeln;
      end;

     write (' Enter system name (maximum 8 characters): ');
     readln (system_name);

   END;

{*************************************************************************}

{ EXTERNAL PROCEDURES }

{$I RDASN.PAS}
{$I RDALL.PAS}
{$I DISSTS.PAS}
{$I DISSTA.PAS}
{$I DISERR.PAS}
{$I GENVEC.PAS}
{$I SYSMGT.PAS}
{$I ASNMGT.PAS}
{$I ACTSCH1.PAS}

{*************************************************************************}

  PROCEDURE SET_UP_SIMULATION;

    VAR answer: char;
        i,j,k: integer;

{-------------------------------------------------------------------------}

    PROCEDURE SET_UP_CONTINUOUS_SIMULATION;

      VAR prob: real;

      BEGIN

        clrscr;
        writeln;
        simulation_time := 0;
        write (' Enter simulation time: ');
        readln (simulation_time);
        while simulation_time > max_simulation_time do
          begin
            write (' Maximum simulation time is ',max_simulation_time,
                   '. Re-enter: ');
            readln (simulation_time);
          end;

        {Find out the probability of each external event}
        writeln;
        for i := 1 to number_of_ext_events do
           begin
             if i <> clock_event then
```

```
        begin
          probability_of_event[i] := -1;
          write (' Enter probability of event ',i:2,': ');
          readln (probability_of_event[i]);
          while (probability_of_event[i] > 1) or
                (probability_of_event[i] < 0) do
            begin
              write (' INVALID. Must be between 0 and 1.',
                     ' Re-enter: ');
              readln (probability_of_event[i]);
            end;
        end;
    end;

{Find out probabilities of all the branches}
clrscr;
writeln;
number_of_branch_nodes := 0;
for i := 1 to number_of_nets do
  begin
    for j := 1 to asn_matrix[i].number_of_nodes do
      begin
        if asn_node_definition[i,j].node = branch then
          begin
            number_of_branch_nodes := number_of_branch_nodes
                                      + 1;
            if number_of_branch_nodes <=
               max_number_of_branch_nodes
             then
              begin
                branch_probability[number_of_branch_nodes]
                                  .network_id := i;
                branch_probability[number_of_branch_nodes]
                                  .node_number := j;
                write (' Enter probability that branch',
                       ' variable "',asn_node_definition[i,j]
                       .branch_variable_id,'" is true: ');
                prob := -1;
                readln (prob);
                while (prob > 1) or (prob < 0) do
                  begin
                    write (' INVALID. Must be between 0 and 1.',
                           ' Re-enter: ');
                    readln (prob);
                  end;
                branch_probability[number_of_branch_nodes]
                      .prob_true := prob;
              end
             else
               simulation_error := too_many_branch_variables;
          end;
      end;
  end;

{Initialize random number generator}
```

```
        randomize;

        {Find out if user still wants the display}
        clrscr;
        writeln;
        write (' Do you want to see a display after each time unit',
              ' (Y/N) ? ');
        readln (answer);
        if (answer = 'Y') or (answer = 'y') then
           begin
             display_flag := true;
             {Find out speed of display}
             writeln;
             writeln (' You may select one of the following display',
                      ' speed:');
             writeln;
             writeln (' S: Slow   (every 7 seconds)');
             writeln (' M: Medium (every 3 seconds)');
             writeln (' F: Fast   (no delay)');
             writeln;
             write (' Enter speed choice: ');
             repeat
               read (kbd,answer);
               answer := upcase(answer);
             until answer in ['S','M','F'];

             if answer = 'S'
                then display_delay := 7000
                else
                   if answer = 'M'
                      then display_delay := 2500
                      else display_delay := 0;
           end

         else
            display_flag := false;


      END;

{-------------------------------------------------------------------}

    PROCEDURE INITIALIZE_COUNTERS;

      VAR value: integer;

      BEGIN

        {Find all counter variables and initialize them}
        clrscr;
        writeln;
        number_of_counters := 0;
        for i := 1 to number_of_nets do
          begin
            for j := 1 to asn_matrix[i].number_of_nodes do
```

```
            begin
              if asn_node_definition[i,j].node = counter then
                begin
                  number_of_counters := number_of_counters + 1;
                  if number_of_counters <= max_number_of_counters
                   then
                    begin
                      counter_variables[number_of_counters]
                                        .network_id := i;
                      counter_variables[number_of_counters]
                                        .node_number := j;
                      counter_variables[number_of_counters]
                                        .status := not_initialized;

                      if counters_initialization = once_at_begining
                       then
                       begin
                        write (' Enter initial value of counter',
                               ' variable "',asn_node_definition[i,j]
                                  .counter_variable_id,'": ');
                        value := 0;
                        readln (value);
                        while value <= 0 do
                          begin
                            write (' INVALID. Re-enter: ');
                            readln (value);
                          end;
                        counter_variables[number_of_counters]
                              .initial_value := value;
                       end;
                    end
                   else
                      simulation_error := too_many_counter_variables;
                end;
            end;
        end;

    END;

{-------------------------------------------------------------------------}

  BEGIN  { Set_up_simulation }

    {Find out if there is a system clock event}
    writeln;
    writeln (' You may have one clock event in the system.',
             ' This event');
    writeln (' will be generated automatically at every time unit.');
    writeln;
    write (' Enter clock event number ("0" for no clock event): ');
    readln (clock_event);
    while clock_event > number_of_ext_events do
      begin
        write (' INVALID CLOCK EVENT. Re-enter: ');
        readln (clock_event);
```

```
  end;

{Find out mode of operation of the simulation}
clrscr;
writeln;
writeln (' The valid simulation modes are:');
writeln;
writeln (' I: Interactive');
writeln (' C: Continuous');
writeln;
write (' Enter simulation mode: ');
repeat
  read (kbd,answer);
  answer := upcase(answer);
until answer in ['I','C'];

if answer = 'I'
   then simulation_mode := interactive
   else simulation_mode := continuous;

if simulation_mode = interactive then
  begin
    display_flag := true;
    clrscr;
    writeln;
    write (' Do you want to initialize the counters immediately',
          ' (Y/N) ? ');
    repeat
      read (kbd,answer);
      answer := upcase(answer);
    until answer in ['Y','N'];

    if answer = 'Y'
       then counters_initialization := once_at_begining
       else counters_initialization := prompt_user;
  end
 else
  begin
    set_up_continuous_simulation;
    counters_initialization := once_at_begining;
  end;
initialize_counters;

if display_flag = true then set_up_display;

{Initialize all nets related information}
for i := 1 to number_of_nets do
   begin
      {Initially all nets are dormant}
      network_info[i].state := dormant;
      {Initialize their state table}
      for j := 1 to asn_matrix[i].number_of_events do
         for k := 1 to asn_matrix[i].number_of_nodes do
            network_info[i].state_table[j,k] := off;
```

```
            {Initialize all event_vectors state to no change and all
             the components of the vector to off}
            event_vector_state[i] := no_change;
            for j := 1 to asn_matrix[i].number_of_events do
                event_vector[i,j] := off;
        end;

    {Initialize ready_to_run_queue to empty}
    ready_to_run_queue_length := 0;

    {Initialize all processors to idle}
    for i := 1 to number_of_processors do
        processor_info[i].state := idle;

    {Initialize all processor statistic variables}
    for i := 1 to number_of_processors do
        for j := 1 to number_of_activities do
            processor_activity_stat_time[i,j] := 0;

    {Initialize all event statistic variables}
    for i := 1 to number_of_ext_events do
        begin
          event_statistics[i].number_of_occurrences := 0;
          event_statistics[i].number_of_rejections := 0;
        end;

    {Initialize all process statistics}
    for i := 1 to number_of_nets do
        begin
          network_statistics[i].number_of_times := 0;
          network_statistics[i].cumulative_completion_time := 0;
          network_statistics[i].max_completion_time := 0;
          network_statistics[i].completion_time := 0;
        end;

    {Initialize all timer statistics}
    for i := 1 to number_of_counters do
        begin
          counter_stat[i].n_times_started := 0;
          counter_stat[i].n_times_expired := 0;
        end;

    {Initialize all queue statistics to zero}
    cumulative_queue_length_before := 0;
    cumulative_queue_length_after  := 0;
    max_queue_length_before  := 0;
    max_queue_length_after  := 0;


END;

{*****************************************************************}

BEGIN { Main Program }
```

```
first_time := true;
repeat

  {Initialize to no simulation error}
  simulation_error := no_error;

  get_system_name;
  read_system_specifications;
  read_system_allocation;
  set_up_simulation;

  time := 0;
  simulation_state := started;
  if display_flag = true
     then display_system_status
     else
       begin
         clrscr;
         writeln;
         writeln (' Simulation started...');
         writeln;
         {writeln (' Time: ');}
       end;

  while (simulation_state <> finished) and (simulation_error = no_error)
   do
    begin

      {The system manager "continuously" monitors the environment for
       external events, and "sends" these events to the proper process
       manager.}
      system_manager;

      {The process managers are responsible for the sequencing of
       activities within a process. For the simulation  purposes,
       there will only be one process manager, which will take care
       of all active nets. The process manager will also be responsible
       for simulating the execution of the activities by the various
       processors of the system.                                      }
      asn_managers;

      {The activity scheduler is responsible for assigning activities
       to processors based on a predetermined priority basis.}
      activity_scheduler;

      {Increment the "system time".}
      time := time + 1;

      if display_flag = true then display_system_status
         else
           begin
            {GotoXY (9,4);
             writeln (time:6);}
           end;
```

```
      if simulation_mode = continuous then
         if time >= simulation_time then
           begin
             {Simulation has ended, inform user by sound}
             simulation_state := finished;
             Sound(720);
             Delay(100);
             NoSound;
           end;

    end;

  if simulation_error <> no_error
     then display_error_message
     else display_statistics;

  answer := ' ';
  writeln;
  write (' Run another simulation [Y/N] ? ');
  repeat
    read (kbd,answer);
  until upcase(answer) in ['Y','N'];
  first_time := false;
  clrscr;
  writeln;

until upcase(answer) = 'N';

END.
```

```
PROCEDURE READ_SYSTEM_SPECIFICATIONS;

   VAR char: string[1];
       file_name: string[14];
       f: text;
       net_number: integer;
       i: integer;
       j: integer;
       number: integer;

   BEGIN

      writeln;
      writeln (' Insert data disk in drive B.');
      write (' Press Return when ready.');
      readln;

      {Form file name using system name}
      file_name := 'b:' + system_name + '.asn';

      {Open the file}
      ASSIGN (f,file_name);
      reset (f);

      {Read the system definition}
      readln (f,number_of_nets);
      readln (f,number_of_activities);
      readln (f,number_of_ext_events);

      for net_number := 1 to number_of_nets do
         begin

            {read starting event}
            readln (f,asn_starting_event[net_number]);

            {read asn matrix}
            readln (f,asn_matrix[net_number].number_of_nodes);
            readln (f,asn_matrix[net_number].number_of_events);
            for i := 1 to asn_matrix[net_number].number_of_events do
              begin
                 for j := 1 to asn_matrix[net_number]
                                  .number_of_nodes do
                       read (f,asn_matrix[net_number].matrix[i,j]);
              end;
            readln (f);

            {read asn node definition}
            for i := 1 to asn_matrix[net_number].number_of_nodes do
              begin
                readln (f,number);
                asn_node_definition[net_number,i].node :=
                                              node_type(number);
                case asn_node_definition[net_number,i].node of
                  activity_initiator:
```

```
                    readln (f,asn_node_definition[net_number,i]
                                        .activity_id);
            branch: readln (f,asn_node_definition[net_number,i]
                                        .branch_variable_id);
            counter: readln (f,asn_node_definition[net_number,i]
                                        .counter_variable_id);
          end;
        end;

      {read asn event definition}
      for i := 1 to asn_matrix[net_number].number_of_events do
        begin

          readln (f,number);
          asn_event_definition[net_number,i]
                              .event := event_type(number);
          case asn_event_definition[net_number,i].event of
            ext_event: readln (f,asn_event_definition[net_number
                                        ,i].event_number);
            int_event:
                begin

                    readln (f,asn_event_definition[net_number,i]
                                .node_number);
                    readln (f,number);
                    asn_event_definition[net_number,i]
                                .node := node_type(number);
                    if asn_event_definition[net_number,i]
                                .node = branch then
                      begin
                         readln (f,number);
                         asn_event_definition[net_number,i]
                            .branch_condition := boolean(number);
                       end;

                end;
          end; {end case}

        end;


    end; {end for loop on net_number}


  CLOSE (f);

END;
```

```
PROCEDURE READ_SYSTEM_ALLOCATION;

   VAR char: string[1];
       file_name: string[14];
       f: text;
       i: integer;
       j: integer;
       number: integer;

   BEGIN

      writeln;
      write (' Enter allocation identifier: ');
      readln (allocation_identifier);

      writeln;
      writeln (' Insert data disk in drive B.');
      write (' Press Return when ready.');
      readln;

      {Form file name using system name}
      file_name := 'b:' + system_name + '.al' + allocation_identifier;

      {Open the file}
      ASSIGN (f,file_name);
      reset (f);

      readln (f,number_of_processors);

      for j := 1 to number_of_processors do
         begin
            readln (f,processor_activity_table_size[j]);
            for i := 1 to processor_activity_table_size[j] do
              begin
                readln (f,processor_activity_table[j,i].activity_id);
                readln (f,processor_activity_table[j,i]
                          .activity_execution_time);
              end;
         end;

      CLOSE (f);


   END;
```

```
PROCEDURE GENERATE_NEW_EVENT_VECTOR (network_number: integer;
                                     node_number: integer);

  VAR answer: char;
      condition: boolean;
      i: integer;
      branch_variable_found: boolean;
      row: integer;
      event_found: boolean;

  BEGIN

    event_found := false;

    {Check if it is a branch node}
    if asn_node_definition[network_number,node_number].node = branch
      then

       begin

          if simulation_mode = interactive then

            begin
              {Ask user for the condition of the branch}
              repeat
                write (' Is the branch variable "',asn_node_definition
                       [network_number,node_number].branch_variable_id,
                       '" true or false (T/F) ? ');
                readln (answer);
              until answer in ['T','t','F','f'];

              if (answer = 'T') or (answer = 't')
                 then condition := true
                 else condition := false;
            end

          else
            begin
              {Find probability of branch variable}
              i := 1;
              branch_variable_found := false;
              while (branch_variable_found = false) and
                    (i <= max_number_of_branch_nodes) do
                begin
                  if (branch_probability[i].network_id = network_number)
                     and (branch_probability[i].node_number =
                          node_number)
                   then
                    begin
                      branch_variable_found := true;
                      if random < branch_probability[i].prob_true
                         then condition := true
                         else condition := false;
                    end
```

```
                else
                   i := i + 1;
             end;

          if branch_variable_found = false
             then simulation_error := branch_prob_not_found;
       end;

       row := 1;
       while (event_found = false) and
             (row <= asn_matrix[network_number].number_of_events) do
         begin
           if (asn_event_definition[network_number,row].event =
              int_event) and (asn_event_definition[network_number
              ,row].node_number = node_number) and
              (asn_event_definition[network_number,row]
              .branch_condition = condition)
              then
                  event_found := true
              else
                  row := row + 1;
         end;

    end

 else {Not a branch node}

    begin
       row := 1;
       while (event_found = false) and
             (row <= asn_matrix[network_number].number_of_events) do
         begin
           if (asn_event_definition[network_number,row].event =
              int_event) and (asn_event_definition[network_number
              ,row].node_number = node_number)
              then
                  event_found := true
              else
                 row := row + 1;
         end;
    end;

 if event_found = true then
    begin
      {Set event vector}
      event_vector_state[network_number] := updated;
      event_vector[network_number,row] := on;
    end

  else
    simulation_error := no_internal_event;

END;
```

```
PROCEDURE SYSTEM_MANAGER;

  VAR number_of_current_ext_events: integer;
      external_event: array [1..max_num_of_simultaneous_events]
                      of integer;
      discard_event_flag: array [1..max_number_of_nets] of boolean;

{*****************************************************************}

  PROCEDURE GET_EXTERNAL_EVENT_FROM_USER;

    VAR answer: string[3];
        answer_status: boolean;
        numerical_value: integer;
        return_code: integer;

    BEGIN  { Get_external_event_from_user }

      { The user can enter only one event at a time, no events can occur
        simultaneously in the interactive mode }
      number_of_current_ext_events := 0;
      answer_status := false;
      write (' Enter external event (if any) or "Q", then press',
             ' return: ');

      while answer_status = false do
        begin

          readln (answer);

          if answer <> '' then
            begin

              if (answer[1] = 'Q') or (answer[1] = 'q') then
                 begin
                   simulation_state := finished;
                   answer_status := true;
                 end

              else
                 begin
                   val(answer,numerical_value,return_code);
                   if return_code = 0 then
                      if numerical_value <= number_of_ext_events then
                         begin
                           number_of_current_ext_events := 1;
                           external_event[1] := numerical_value;
                           answer_status := true;
                         end
                       else
                          write (' INVALID EVENT. Re-enter: ')
                   else
                     write (' INVALID RESPONSE. Re-enter: ');
```

```
                end;

            end

          {A carriage return alone is a valid answer}
          else
            answer_status := true;

        end;

    END;
{*****************************************************************}

  PROCEDURE GET_RANDOM_EXTERNAL_EVENT;

    VAR i: integer;

    BEGIN  { Get_random_external_event }

      number_of_current_ext_events := 0;
      for i := 1 to number_of_ext_events do
         begin
           if i <> clock_event then
             begin
               if random < probability_of_event[i] then
                  begin
                    if number_of_current_ext_events <
                       max_num_of_simultaneous_events
                     then
                       begin
                          number_of_current_ext_events :=
                                    number_of_current_ext_events + 1;
                          external_event[number_of_current_ext_events]
                                    := i;
                       end
                     else
                          simulation_error :=
                            too_many_simultaneous_events;
                  end;
             end;
         end;

    END;
{*****************************************************************}

  PROCEDURE SET_UP_EXTERNAL_EVENT_VECTOR (event: integer);

    VAR k,l: integer;
        event_rejected_flag: boolean;

    BEGIN

      { For statistical purpose, count event occurence and set flag }
```

```
      event_statistics[event].number_of_occurrences :=
            event_statistics[event].number_of_occurrences + 1;
      event_rejected_flag := true;

      { Set-up event vector for all awaken process that are waiting
        for this event }
      for k := 1 to number_of_nets do
        begin
          if (network_info[k].state = awake) and
             (discard_event_flag[k] = false) then
            begin
              { Find out if this external event is part of the event
                definition }
              for l := 1 to asn_matrix[k].number_of_events do
                begin
                  if (asn_event_definition[k,l].event = ext_event) and
                     (asn_event_definition[k,l].event_number = event)
                    then
                          begin
                            event_rejected_flag := false;
                            event_vector_state[k] := updated;
                            event_vector[k,l] := on;
                          end;
                end;
            end;
        end;

      { If event was rejected, i-e not accepted by any ASN, update event
        statistics }
      if event_rejected_flag = true then
         event_statistics[event].number_of_rejections :=
               event_statistics[event].number_of_rejections + 1;

    END;

{*****************************************************************}


VAR i: integer;
    event_index: integer;

BEGIN { system_manager }

  { Set discard_event_flag to false on all nets }
  for i := 1 to number_of_nets do
     discard_event_flag[i] := false;

  { If there is a system clock event, update event vector }
  if clock_event <> 0 then set_up_external_event_vector (clock_event);

  { Check for external events }
  if simulation_mode = interactive
     then get_external_event_from_user
     else get_random_external_event;
```

```
{ Process all simultaneous external events }
event_index := 1;
while number_of_current_ext_events <> 0 do
 begin

   { Awaken any dormant process that will start on this event }
   for i := 1 to number_of_nets do
     begin
       if asn_starting_event[i] = external_event[event_index] then
          { If the process is already awake, the external event will
            be discarded. This restriction could be taken out...     }
          if network_info[i].state = dormant then
             begin
               network_info[i].state := awake;
               network_statistics[i].number_of_times :=
                        network_statistics[i].number_of_times + 1;
               network_statistics[i].completion_time := 0;
             end
           else
             discard_event_flag[i] := true;
     end;

   { Now that all the proper process are awaken, set up the event
     vector }
   set_up_external_event_vector (external_event[event_index]);

   number_of_current_ext_events := number_of_current_ext_events - 1;
   event_index := event_index + 1;

 end;

END;
```

```
PROCEDURE  ASN_MANAGER (network_number:integer);

  VAR i,j: integer;
      node_number: integer;

{-------------------------------------------------------------------------}

  FUNCTION ALL_INPUT_TRIGGERS_PRESENT: boolean;

    VAR  row: integer;
         present_flag: boolean;

    BEGIN { All_input_triggers_present }

      row := 1;
      present_flag := true;

      while (row <= asn_matrix[network_number].number_of_events) and
            (present_flag = true) do
       begin
         if (asn_matrix[network_number].matrix[row,node_number] =
             input_trigger) and
            (network_info[network_number].state_table[row,node_number]
             = off)
           then
               present_flag := false
           else
               row := row + 1;
       end;

       all_input_triggers_present := present_flag;

    END;

{-------------------------------------------------------------------------}

  PROCEDURE CHECK_ACTIVITY_INITIATOR_NODE;

    VAR  row: integer;
         start_activity_flag: boolean;

    BEGIN { Check_activity_initiator_node }

      if all_input_triggers_present then
        begin

          {Clear the column of the process state table (forget past
           triggers)}
          for row := 1 to asn_matrix[network_number].number_of_events do
             network_info[network_number].state_table[row,node_number]
                    := off;

          {Put the activity in the ready to run queue}
          if ready_to_run_queue_length < max_ready_to_run_queue_length
```

```
            then
              begin
                ready_to_run_queue_length := ready_to_run_queue_length
                                             + 1;
                ready_to_run_queue[ready_to_run_queue_length]
                    .activity_id := asn_node_definition[network_number,
                                    node_number].activity_id;
                ready_to_run_queue[ready_to_run_queue_length].network_id
                                  := network_number;
                ready_to_run_queue[ready_to_run_queue_length].node_number
                                  := node_number;
              end
            else
              {Set a simulation error, which will stop the simulation}
              simulation_error := ready_queue_overflow;

        end;

    END;

{-------------------------------------------------------------------}

  PROCEDURE CHECK_WAIT_NODE;

    VAR  row: integer;
         wait_completed_flag: boolean;

    BEGIN { Check_wait_node }

      if all_input_triggers_present then
        begin

          {Clear the column of the process state table (forget past
           triggers)}
          for row := 1 to asn_matrix[network_number].number_of_events do
            network_info[network_number].state_table[row,node_number]
                    := off;

          {Prepare a new event vector}
          generate_new_event_vector (network_number,node_number);

        end;

    END;

{-------------------------------------------------------------------}

  PROCEDURE CHECK_MERGE_NODE;

    VAR  row: integer;
         merge_completed_flag: boolean;

    BEGIN { Check_merge_node }

      row := 1;
```

```
      merge_completed_flag := false;

      {Check if any input trigger is present}
      while (row <= asn_matrix[network_number].number_of_events) and
            (merge_completed_flag = false) do
        begin
          if (asn_matrix[network_number].matrix[row,node_number] =
              input_trigger) and
             (network_info[network_number].state_table[row,node_number]
              = on)
            then
                  merge_completed_flag := true
            else
                  row := row + 1;
        end;

      {If the flag is set then the merge is completed}
      if merge_completed_flag = true then
        begin

          {Clear the column of the process state table (forget past
           triggers)}
          for row := 1 to asn_matrix[network_number].number_of_events do
             network_info[network_number].state_table[row,node_number]
                  := off;

          {Prepare a new event vector}
          generate_new_event_vector (network_number,node_number);

        end;

    END;

{-------------------------------------------------------------------------}

  PROCEDURE CHECK_BRANCH_NODE;

    VAR  row: integer;
         branch_ready_flag: boolean;

    BEGIN { Check_branch_node }

      if all_input_triggers_present then
        begin

          {Clear the column of the process state table (forget past
           triggers)}
          for row := 1 to asn_matrix[network_number].number_of_events do
             network_info[network_number].state_table[row,node_number]
                  := off;

          {Prepare a new event vector}
          generate_new_event_vector (network_number,node_number);

        end;
```

```
    END;

{-------------------------------------------------------------------------}

  PROCEDURE CHECK_GATE_NODE;

    VAR row: integer;
        disable_input_found: boolean;
        enable_input_found: boolean;
        n: integer;

{-------------------------------------------------------------------------}

    BEGIN { Check_gate_node }

      {Check disable input first}
      row := 1;
      disable_input_found := false;
      while (disable_input_found = false) and
            (row <= asn_matrix[network_number].number_of_events) do
        begin
          if asn_matrix[network_number].matrix[row,node_number] =
             disable
            then
              begin
                disable_input_found := true;
                if network_info[network_number].state_table[row,
                   node_number] = on
                  then
                    {Clear the column in the process state table}
                    for n := 1 to asn_matrix[network_number]
                                          .number_of_events do
                        network_info[network_number].state_table
                                    [n,node_number] := off;

              end
            else
                row := row + 1;
        end;

      {Check if gate is enabled}
      row := 1;
      enable_input_found := false;
      while (enable_input_found = false) and
            (row <= asn_matrix[network_number].number_of_events) do
            if asn_matrix[network_number].matrix[row,node_number] = enab
le
               then
                   enable_input_found := true
               else
                   row := row + 1;

      if enable_input_found = false
         then simulation_error := no_enable_input;
```

```
      {If gate is enabled and all input triggers are present generate
       an event vector}
      if network_info[network_number].state_table[row,node_number] = on
         then
           if all_input_triggers_present
             then generate_new_event_vector (network_number,
                                             node_number)
         else
           {Gate is not enabled: clear the column to forget
            input triggers}
           for n := 1 to asn_matrix[network_number].number_of_events do
               network_info[network_number].state_table
                        [n,node_number] := off;

    END;

{-------------------------------------------------------------------------}

  PROCEDURE CHECK_COUNTER_NODE;

    VAR row: integer;

{-------------------------------------------------------------------------}

    PROCEDURE DISABLE_COUNTER_NODE;

      VAR  k: integer;

      BEGIN { Disable_counter_node }

        {Clear column in process state table}
        for k := 1 to asn_matrix[network_number].number_of_events do
          network_info[network_number].state_table[k,node_number]
              := off;

        {Reset counter variable}
        for k := 1 to number_of_counters do
          if (counter_variables[k].network_id = network_number) and
             (counter_variables[k].node_number = node_number)
             then counter_variables[k].status := not_initialized;

    END;

{-------------------------------------------------------------------------}

    PROCEDURE CHECK_ENABLED_COUNTER;

      VAR counter_index: integer;
          counter_found: boolean;
          value: integer;

      BEGIN { Check_enabled_counter }

        {Find the counter variable}
        counter_index := 1;
```

```
counter_found := false;

while (counter_index <= number_of_counters) and
      (counter_found = false) do
  begin
    if (counter_variables[counter_index].network_id =
        network_number) and (counter_variables[counter_index]
        .node_number = node_number)
       then counter_found := true
       else counter_index := counter_index + 1;
  end;

if counter_found = true then
   begin
     if counter_variables[counter_index].status =
        not_initialized then
      begin
        if counters_initialization = prompt_user then
          begin
            {Ask user for initial value of variable}
            write (' Enter initial value of counter ',
                   asn_node_definition[network_number
                   ,node_number].counter_variable_id,
                   ' for process number ',network_number:2,
                   ': ');
            value := 0;
            readln (value);
            while value <= 0 do
              begin
                write (' INVALID. Re-enter: ');
                readln (value);
              end;
            counter_variables[counter_index].initial_value
                   := value;
          end;
        counter_variables[counter_index].value :=
               counter_variables[counter_index].initial_value;
        counter_variables[counter_index].status := initialized;
        counter_stat[counter_index].n_times_started :=
               counter_stat[counter_index].n_times_started + 1;
      end

     else {counter is initialized, check for input triggers}
      begin
        if all_input_triggers_present then
          begin
            counter_variables[counter_index].value :=
                   counter_variables[counter_index].value - 1;
            if counter_variables[counter_index].value = 0 then
              begin
                counter_stat[counter_index].n_times_expired :=
                     counter_stat[counter_index].n_times_expired
                     + 1;
                disable_counter_node;
                {And generate the output trigger}
```

```
                        generate_new_event_vector (network_number,
                                                   node_number);
                    end;
                end;
            end;
        end
        else
            simulation_error := counter_variable_not_found;

    END;

{-------------------------------------------------------------------}

    VAR enable_found: boolean;

    BEGIN { Check_counter_node }

      {Check for disable input}
      for row := 1 to asn_matrix[network_number].number_of_events do
        if (asn_matrix[network_number].matrix[row,node_number] =
            disable) and
           (network_info[network_number].state_table[row,node_number]
            = on)
           then disable_counter_node;

      {Check if counter is enabled}
      enable_found := false;
      row := 1;
      while (enable_found = false) and
            (row <= asn_matrix[network_number].number_of_events) do
       begin
         if asn_matrix[network_number].matrix[row,node_number] = enable
          then
           begin
             enable_found := true;
             if network_info[network_number].state_table[row,
                node_number] = on
              then {counter is enabled}
                   check_enabled_counter;
           end
          else
             row := row + 1;
       end;

      {Clear column except for enable input}
      for row := 1 to asn_matrix[network_number].number_of_events do
        if asn_matrix[network_number].matrix[row,node_number] <> enable
           then network_info[network_number].state_table[row,
                                node_number] := off;

    END;

{-------------------------------------------------------------------}

  PROCEDURE CHECK_TERMINATOR_NODE;
```

```
VAR  row: integer;
     column: integer;
     network_completed_flag: boolean;
     index: integer;

BEGIN { Check_network_terminator_node }

  row := 1;
  network_completed_flag := true;

  {Check if any input trigger is present}
  while (row <= asn_matrix[network_number].number_of_events) and
        (network_completed_flag = true) do
    begin
      if (asn_matrix[network_number].matrix[row,node_number] =
          input_trigger) and
         (network_info[network_number].state_table[row,node_number]
          = off)
        then
              network_completed_flag := false
        else
              row := row + 1;
    end;

  {If the flag is set then the process is completed}
  if network_completed_flag = true then
    begin

      {Set process state back to dormant}
      network_info[network_number].state := dormant;

      {Clear the entire process state table}
      for row := 1 to asn_matrix[network_number].number_of_events do
         for column := 1 to asn_matrix[network_number]
         .number_of_nodes do
            network_info[network_number].state_table[row,column]
            := off;

      {Clear the event vector as a precaution}
      event_vector_state[network_number] := no_change;
      for index := 1 to asn_matrix[network_number].number_of_events
      do
         event_vector[network_number,index] := off;

      {Reset any counter variable in this process}
      for index := 1 to number_of_counters do
           if (counter_variables[index].status = initialized) and
              (counter_variables[index].network_id = network_number)
              then counter_variables[index].status
                        := not_initialized;

      {Keep track of completion time for statistics}
      network_statistics[network_number].cumulative_completion_time
         := network_statistics[network_number]
```

```
                .cumulative_completion_time + network_statistics
                [network_number].completion_time;
            if network_statistics[network_number].completion_time >
               network_statistics[network_number].max_completion_time then
               network_statistics[network_number].max_completion_time :=
               network_statistics[network_number].completion_time;

         end;

   END;

{-------------------------------------------------------------------------}


BEGIN  { asn_manager }

   {Update processor state table using event vector}
   for i := 1 to asn_matrix[network_number].number_of_events do
     begin
       if event_vector[network_number,i] = on then
         begin
           for j := 1 to asn_matrix[network_number].number_of_nodes do
              if asn_matrix[network_number].matrix[i,j] <> not_connected
                 then network_info[network_number].state_table[i,j]
                      := on;
         end;
     end;

   {Event vector is no longer useful}
   event_vector_state[network_number] := no_change;
   for i := 1 to asn_matrix[network_number].number_of_events do
      event_vector[network_number,i] := off;

   {Check every node in the asn table according to the process state
    table}
   for node_number := 1 to asn_matrix[network_number].number_of_nodes do
      begin
        case asn_node_definition[network_number,node_number].node of
          activity_initiator: check_activity_initiator_node;
          wait:               check_wait_node;
          merge:              check_merge_node;
          branch:             check_branch_node;
          gate:               check_gate_node;
          counter:            check_counter_node;
          terminator:         check_terminator_node;
        end;
      end;

END;

{*************************************************************************}

PROCEDURE ASN_MANAGERS;

  VAR network_number: integer;
```

```
BEGIN { asn_managers }

  {Run asn manager for each active process in the system}
  for network_number := 1 to number_of_nets do
    begin

      {Increment process_completion time for statistics}
      if network_info[network_number].state = awake
         then network_statistics[network_number].completion_time :=
              network_statistics[network_number].completion_time + 1;

      {Run process manager for all awaken process until there is no
       more change in the event vector, or it goes to sleep...      }
      while (network_info[network_number].state = awake) and
            (event_vector_state[network_number] = updated) do
            asn_manager(network_number);

    end;

END;
```

```
{ This is the first version of the activity scheduler. It assigned
  activities in the queue in a FIFO fashion to the first processor
  available. If it can not find a free processor that can execute
  the first activity in the queue, it will try the second one in
  the queue and so on...                                          }

PROCEDURE ACTIVITY_SCHEDULER;

  VAR queue_index: integer;
      activity_assigned: boolean;
      processor_number: integer;

{*****************************************************************}

  PROCEDURE SIMULATE_ACTIVITY_EXECUTION;

  VAR i: integer;

  BEGIN  { Simulate_activity_execution }

    {Decrement execution time left for activity on every busy processor}
    for i := 1 to number_of_processors do
        if processor_info[i].state = busy then
           if processor_info[i].execution_time_left <> 0 then
              begin

                processor_info[i].execution_time_left :=
                         processor_info[i].execution_time_left - 1;

                {Increment cummulative execution time of that activity
                 for statistical purposes}
                processor_activity_stat_time[i,processor_info[i]
                         .activity_id] := processor_activity_stat_time
                         [i,processor_info[i].activity_id] + 1;

                {If activity is almost finished prepare output trigger}
                if processor_info[i].execution_time_left = 1
                  then
                       generate_new_event_vector (processor_info[i]
                                .network_id,processor_info[i]
                                .node_number)
                  else
                       {If activity is completed free processor}
                       if processor_info[i].execution_time_left = 0 then
                       processor_info[i].state := idle;

              end;


  END;

{*****************************************************************}

  PROCEDURE TRY_TO_ASSIGN_ACTIVITY_TO_PROCESSOR;
```

```
    VAR activity_index: integer;
        activity_found: boolean;

    BEGIN

      {Find out if this processor can execute the activity in the queue}
      activity_index := 1;
      activity_found := false;
      while (activity_index <= processor_activity_table_size
            [processor_number]) and (activity_found = false) do
        begin
          if processor_activity_table[processor_number,activity_index]
             .activity_id = ready_to_run_queue[queue_index].activity_id
           then
                activity_found := true
           else
                activity_index := activity_index + 1;
        end;

      {If the processor can execute the activity, then assign it to it}
      if activity_found = true then
         begin

           processor_info[processor_number].state := busy;
           processor_info[processor_number].activity_id :=
                     ready_to_run_queue[queue_index].activity_id;
           processor_info[processor_number].network_id :=
                     ready_to_run_queue[queue_index].network_id;
           processor_info[processor_number].node_number :=
                    ready_to_run_queue[queue_index].node_number;
           processor_info[processor_number].execution_time_left :=
                    processor_activity_table[processor_number,
                    activity_index].activity_execution_time;
           activity_assigned := true;

           {If the activity only requires one execution time unit to
            run, generate output trigger, as it is "almost" finished. }
           if processor_info[processor_number].execution_time_left = 1
              then generate_new_event_vector
                     (processor_info[processor_number].network_id,
                      processor_info[processor_number].node_number);

         end;

    END;

{*****************************************************************}

  PROCEDURE RESHUFFLE_QUEUE;

    VAR move_index_1: integer;
        move_index_2: integer;

    BEGIN
```

```
        move_index_1 := queue_index;
        move_index_2 := queue_index + 1;
        while move_index_2 <= ready_to_run_queue_length do
          begin
            ready_to_run_queue[move_index_1].activity_id :=
                      ready_to_run_queue[move_index_2].activity_id;
            ready_to_run_queue[move_index_1].network_id :=
                      ready_to_run_queue[move_index_2].network_id;
            ready_to_run_queue[move_index_1].node_number :=
                      ready_to_run_queue[move_index_2].node_number;
            move_index_1 := move_index_1 + 1;
            move_index_2 := move_index_2 + 1;
          end;

        {Adjust queue size and queue index accordingly}
        ready_to_run_queue_length := ready_to_run_queue_length - 1;
        queue_index := queue_index - 1;

      END;

  {*************************************************************************}

  VAR i: integer;

  BEGIN { activity_scheduler }

    simulate_activity_execution;

    {"abort" any activities running for a dormant process}
    for i := 1 to number_of_processors do
      if (processor_info[i].state = busy) then
         if (network_info[processor_info[i].network_id].state = dormant)
            then processor_info[i].state := idle;

    {Clean up queue: take out activities associated with a dormant
     process}
    queue_index := 1;
    while queue_index <= ready_to_run_queue_length do
      begin
        if network_info[ready_to_run_queue[queue_index].network_id]
                  .state = dormant
           then reshuffle_queue;
        queue_index := queue_index + 1;
      end;

    {For statistical purpose add queue size}
    cumulative_queue_length_before := cumulative_queue_length_before +
                                      ready_to_run_queue_length;
    if ready_to_run_queue_length > max_queue_length_before then
       max_queue_length_before := ready_to_run_queue_length;

    if ready_to_run_queue_length <> 0 then

       begin {Try to assign a processor to all activities in the queue}
```

```
        queue_index := 1;
        while queue_index <= ready_to_run_queue_length do
          begin
            processor_number := 1;
            activity_assigned := false;
            while (processor_number <= number_of_processors) and
                  (activity_assigned = false) do
              begin

                if processor_info[processor_number].state = idle then
                   begin
                     try_to_assign_activity_to_processor;
                     if activity_assigned = true
                        then reshuffle_queue
                        else processor_number := processor_number + 1;
                   end

                 else
                      processor_number := processor_number + 1;

              end;

            queue_index := queue_index + 1;

          end;

      end;

    {For statistical purpose add queue size after assignment of activities
     to processors}
    cumulative_queue_length_after := cumulative_queue_length_after +
                                     ready_to_run_queue_length;
    if ready_to_run_queue_length > max_queue_length_after then
       max_queue_length_after := ready_to_run_queue_length;

END;
```

```
PROCEDURE SET_UP_DISPLAY;

BEGIN { Set_up_display }

  if color_display then
    begin

      window (1,1,80,3);
      TextBackground (magenta);
      clrscr;

      window (1,4,80,15);
      TextBackground (blue);
      clrscr;

      window (1,16,80,23);
      TextBackground (yellow);
      clrscr;

      window (1,24,80,25);
      TextBackground (green);
      clrscr;

      window (1,1,80,25);

    end

  else
    clrscr;

END;

{*************************************************************************}

PROCEDURE DISPLAY_SYSTEM_STATUS;

VAR i: integer;
    display_number: integer;

BEGIN { Display_system_status }

   window (1,1,80,25);
   GotoXY (50,2);
   if color_display then
     begin
       TextBackground (magenta);
       TextColor (white);
     end;
   writeln (' SIMULATION TIME: ',time:6);

   {Display processor status}
   GotoXY (1,4);
   if color_display then
     begin
```

```
    TextBackground (blue);
    TextColor (yellow);
  end;
writeln (' Processor Number    Status   Activity ID      ASN ID ',
         '     Execution Time Left');

writeln;
for i:= 1 to number_of_processors do
   begin
      if processor_info[i].state = idle then
         writeln (i:10,'idle':16,' ':50)
      else
         writeln (i:10,'busy':16,processor_info[i].activity_id:11,
                  processor_info[i].network_id:13,
                  processor_info[i].execution_time_left:23);
   end;

{Display ready to run queue info}
window (1,16,30,23);
if color_display then
  begin
    TextBackground (yellow);
    TextColor (black);
  end;
clrscr;
writeln (' Ready to Run Queue ');
writeln;
writeln (' Queue size: ',ready_to_run_queue_length:2);
writeln;
if ready_to_run_queue_length > 3
   then display_number := 3
   else display_number := ready_to_run_queue_length;
for i := 1 to display_number do
    writeln (' Activity ID: ',ready_to_run_queue[i].activity_id:2,
             '  ASN ID: ',ready_to_run_queue[i].network_id:2);

{Display counter variables, (if any)}
window (31,16,56,23);
clrscr;
writeln ('Counter variables ');
writeln;
display_number := 0;
for i := 1 to number_of_counters do
   if (counter_variables[i].status = initialized) and
      (display_number < 6) then
     begin
         writeln ('ASN ID: ',counter_variables[i].network_id:2,
                  asn_node_definition[counter_variables[i]
                  .network_id, counter_variables[i].node_number]
                  .counter_variable_id:3,' = ',counter_variables[i]
                  .value:3);
         display_number := display_number + 1;
     end;

{Display net status}
```

```
    window (57,16,80,23);
    clrscr;
    writeln (' Status of Networks ');
    window (57,18,68,23);
    GotoXY (1,1);
    {Will display a maximum of 10 net status because of space
     limitations}
    for i := 1 to 10 do
      begin
        if i = 6 then begin
                        window (69,18,80,23);
                        GotoXY (1,1);
                      end;
        if i > number_of_nets
          then writeln (i:2)
          else if network_info[i].state = dormant
                 then writeln (i:2,' dormant')
                 else writeln (i:2,' awaken ');
      end;

    window (1,24,80,25);
    if color_display then
      begin
        TextBackground (green);
        TextColor (black);
      end;
    clrscr;

    {Delay if in continuous mode}
    if (simulation_mode = continuous) and (display_delay <> 0)
       then delay (display_delay);

END;
```

```
PROCEDURE DISPLAY_STATISTICS;

  PROCEDURE WAIT_FOR_USER;

    BEGIN
       window (1,24,80,25);
       if color_display then
         begin
           TextBackground (green);
           TextColor (black);
         end;
       clrscr;
       write (' Press any key to continue. ');
       while not keypressed do;
       clrscr;
    END;

{-------------------------------------------------------------------------}

VAR i: integer;
    processor_number: integer;
    network_number: integer;
    activity_index: integer;
    display_number: integer;
    total_busy_time: integer;
    percentage: real;
    number_of_times: real;
    average: real;

{-------------------------------------------------------------------------}

PROCEDURE DISPLAY_PROC_STATS;

 BEGIN

   {Display statistics for all processors}
   processor_number := 1;
   activity_index := 1;
   while processor_number <= number_of_processors do
     begin

       window (1,4,80,23);
       if color_display then
         begin
           TextBackground (blue);
           TextColor (yellow);
         end;
       clrscr;
       writeln (' STATISTICS OF PROCESSOR ',processor_number:2);
       writeln;
       writeln (' ':20,'Number of times executed',' ':17,
                'Percentage of time');
       writeln;
```

```
      {Find total busy time}
      total_busy_time := 0;
      for i := 1 to number_of_activities do
          total_busy_time := total_busy_time
                           + processor_activity_stat_time
                             [processor_number,i];

      {Find percentage idle time and display it}
      percentage := ((time - total_busy_time)/time)*100;
      writeln (' IDLE',' ':61,percentage:6:2,' %');

      display_number := 1;
      while (activity_index <= processor_activity_table_size
            [processor_number]) and (display_number <= 14) do
        begin

          {Find number of times activity was executed by dividing the
           number of time units the processor was executing that
           activity by the activity execution time}
          number_of_times :=
             processor_activity_stat_time[processor_number,
                      processor_activity_table [processor_number,
                      activity_index].activity_id]
             / processor_activity_table[processor_number,
                      activity_index].activity_execution_time;

          {Find percentage of time}
          percentage := (processor_activity_stat_time[processor_number,
                         processor_activity_table[processor_number,
                         activity_index].activity_id] / time)*100;

          {Display this information}
          writeln (' Activity',processor_activity_table
                   [processor_number,activity_index].activity_id:3,
                   ' ':18,number_of_times:6:1,' ':30,percentage:6:2,
                   ' %');

          {Increment activity index}
          activity_index := activity_index + 1;

        end;

      if activity_index >
         processor_activity_table_size[processor_number] then
         begin
           processor_number := processor_number + 1;
           activity_index := 1;
         end;

      wait_for_user;

    end;

END;
```

```
{-------------------------------------------------------------------------}

PROCEDURE DISPLAY_EVENT_STATS;

 BEGIN

   {Display event statistics}
   window (1,4,80,23);
   if color_display then
     begin
       TextBackground (blue);
       TextColor (yellow);
     end;
   clrscr;
   writeln (' EVENT STATISTICS ');
   writeln;
   writeln (' Event Number',' ':11,'Number of',
            ' ':11,'Number of',' ':12,'Percentage of');
   writeln (' ':23,'Occurrences',' ':10,'Rejections',' ':12,
            'Rejections');
   writeln;
   for i := 1 to number_of_ext_events do
     begin
       {Calculate percentage of rejections}
       if event_statistics[i].number_of_occurrences <> 0
         then
           percentage := (event_statistics[i].number_of_rejections /
                          event_statistics[i].number_of_occurrences)
                         * 100
         else
           percentage := 0.0;

       writeln ('      ',i:2,
                ' ':18,event_statistics[i].number_of_occurrences:6,
                ' ':14,event_statistics[i].number_of_rejections:6,
                ' ':16,percentage:6:2,' %');
     end;

   wait_for_user;

 END;

{-------------------------------------------------------------------------}

PROCEDURE DISPLAY_NET_STATS;

 BEGIN

   {Display statistics for all processes}
   network_number := 1;
   while network_number <= number_of_nets do
     begin

       window (1,4,80,23);
       if color_display then
```

```
        begin
          TextBackground (blue);
          TextColor (yellow);
        end;
      clrscr;
      writeln (' NETWORK STATISTICS');
      writeln;
      writeln (' Network                    Number of          ',
                                        'Completion Time          ');
      writeln ('    ID                   times awakened     ',
                                'Maximum              Average  ');
      writeln;

      display_number := 1;
      while (display_number <= 10) and
            (network_number <= number_of_nets) do
        begin
          if network_statistics[network_number].number_of_times <> 0
           then
            average := network_statistics[network_number]
                       .cumulative_completion_time /
                       network_statistics[network_number]
                       .number_of_times
            else average := 0;

          writeln ('    ',network_number:2,
                   ' ':20,network_statistics[network_number]
                          .number_of_times:6,
                   ' ':19,network_statistics[network_number]
                          .max_completion_time:6,
                   ' ':12,average:8:1);

          network_number := network_number + 1;
          display_number := display_number + 1;
        end;

      wait_for_user;

    end;

  END;

{-------------------------------------------------------------------}

PROCEDURE DISPLAY_TIMER_STATS;

 BEGIN

   {Display timer/counter statistics}
   window (1,4,80,23);
   if color_display then
     begin
       TextBackground (blue);
       TextColor (yellow);
     end;
```

```
    clrscr;
    writeln (' TIMER/COUNTER STATISTICS ');
    writeln;
    writeln (' ASN Number     Node Number      Timer ID      # Times      ',
             '# Times','     Percentage');
    writeln ('                                                Started      ',
             'Expired');
    writeln;
    if number_of_counters <> 0 then
      for i := 1 to number_of_counters do
        begin
          {Calculate percentage of expiration}
          if counter_stat[i].n_times_started <> 0
            then
              percentage := (counter_stat[i].n_times_expired /
                             counter_stat[i].n_times_started) * 100
            else
              percentage := 0.0;

          writeln (counter_variables[i].network_id:8,
                   counter_variables[i].node_number:13,
                   asn_node_definition[counter_variables[i].network_id,
                                       counter_variables[i].node_number]
                                       .counter_variable_id:15,
                   counter_stat[i].n_times_started:15,
                   counter_stat[i].n_times_expired:11,
                   ' ':7,percentage:6:2,' %');
        end;

    wait_for_user;

  END;

{-------------------------------------------------------------------------}

PROCEDURE DISPLAY_QUEUE_STATS;

  BEGIN

    {Display queue statistics}
    window (1,4,80,23);
    if color_display then
      begin
        TextBackground (blue);
        TextColor (yellow);
      end;
    clrscr;
    writeln (' QUEUE STATISTICS ');
    writeln;

    writeln (' BEFORE assignement of activities to processor:');
    writeln (' The maximum Ready to Run queue length is: ',
             max_queue_length_before:2);
    average := cumulative_queue_length_before / time;
    writeln (' The average Ready to Run queue length is: ',average:5:3);
```

```
    writeln;
    writeln (' AFTER assignement of activities to processor:');
    writeln (' The maximum Ready to Run queue length is: ',
             max_queue_length_after:2);
    average := cumulative_queue_length_after / time;
    writeln (' The average Ready to Run queue-length is: ',average:5:3);

    (End of display)
    wait_for_user;

  END;

{-------------------------------------------------------------------------}

VAR choice: char;

BEGIN ( Display_statistics )

    window (1,1,80,25);
    GotoXY (50,2);
    if color_display then
      begin
        TextBackground (magenta);
        TextColor (white);
      end;
    clrscr;
    GotoXY (50,2);
    writeln ('SIMULATION TIME: ',time:6);

    repeat

      window (1,4,80,23);
      if color_display then
        begin
          TextBackground (blue);
          TextColor (yellow);
        end;
      clrscr;
      writeln (' STATISTIC MENU ');
      writeln;
      writeln (' 1  Display Processors Statistics');
      writeln (' 2  Display Event Statistics');
      writeln (' 3  Display Network Statistics');
      writeln (' 4  Display Timer Statistics');
      writeln (' 5  Display Queue Statistics');
      writeln (' 6  Exit');
      writeln;
      write (' Enter choice: ');

      repeat
        read (kbd,choice);
      until choice in ['1','2','3','4','5','6'];

      case choice of
```

```
            '1': display_proc_stats;
            '2': display_event_stats;
            '3': display_net_stats;
            '4': display_timer_stats;
            '5': display_queue_stats;
        end;
    until choice = '6';

    window (1,1,80,25);
    clrscr;

END;
```

```
PROCEDURE DISPLAY_ERROR_MESSAGE;

  BEGIN

    window (1,1,80,25);
    if color_display = true then
      begin
        TextBackground (blue);
        TextColor (yellow);
      end;
    clrscr;
    writeln;
    writeln (' Simulation Error');
    writeln;

    case simulation_error of

      ready_queue_overflow:
           writeln (' Overflow of the ready to run queue.');

      no_enable_input:
           begin
             writeln (' Error in the system definition (TSD).');
             writeln (' No enable input entered for a gate node.');
           end;

      no_internal_event:
           begin
             writeln (' Error in the system definition (TSD).');
             writeln (' Can not find an internal event corresponding ',
                      'an output trigger of a node.');
           end;

      too_many_simultaneous_events:
           writeln (' Too many simultaneous events.');

      too_many_branch_variables:
           writeln (' Too many branch variables for continuous mode.');

      too_many_counter_variables:
           writeln (' Too many counter variables.');

      branch_prob_not_found:
           writeln (' Program error: Branch probability not found.');

      counter_variable_not_found:
           writeln (' Program error: Counter variable not found.');

    end;

  END;
```