1976

Multicentered computer architecture for real-time data acquisition and display

Larry Jay Ellis
Iowa State University

INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again – beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR

Multicentered computer architecture for real-time data acquisition and display

by

Larry Jay Ellis

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Electrical Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1976

TABLE OF CONTENTS

ACKNOWLEDGEMENT

This work was supported by the United States Energy Research and Development Administration (ERDA). I gratefully acknowledge their financial support without which my graduate program would not have been possible.

I would like to extend my appreciation to Professor Roy J. Zingg for his patience and the many hours spent in consultation on the material contained in this dissertation.

I am indebted to Sheila Hilts for her excellent help in the typing of this manuscript.

Finally, and most of all, I extend my most heartfelt appreciation to my wife Molly and children Scott and Matthew for their patience and understanding during the years of graduate study at Iowa State University.

CHAPTER 1

INTRODUCTION

System Overview

With the advent of low cost computer equipment, in particular the minicomputer, there has been a significant growth in the design and development of Real-time On-line Data Acquisition and Display (RTDAD) facilities. Generally, these facilities are computer-based digital systems with specially designed front-end data handling preprocessors and a standard complement of computer peripherals(49-51,55,57).

Since the terms real-time and on-line have ambiguious connotations, let us define them for the RTDAD context as follows(66):

1. An on-line system accepts input directly from the input environment in which it is created. Additionally, the outputs or results of computation are directed to the user environment where they are required.
2. A real-time system monitors an input environment by receiving data, processing it, and providing results to a user environment within a predefined maximum time limit. If the timing constraints are not met, a significant degradation (if not complete collapse) in system integrity is realized.

In RTDAD systems, the timing bounds vary with the type of data being received. The range of variation, typically, is from a few milliseconds to a few minutes. The time critical nature of the RTDAD problem is the central, recurring theme dictating much of the system design philosophy.

The fundamental tenet is that the entire processing package (hardware and software) is effecting its environment in a real-time on-line manner and must observe rigid-timing requirement even under well-defined worst case conditions.

In general, RTDAD systems maintain a passive interaction with the input environment. That is, the system monitors the activities/events in the input environment with only a minimal amount of control feedback.

An overview of a typical RTDAD facility is shown in Figure 1.1. The input environment contains one or more input transducers which, in some manner, measure an event in the external environment and generate a message corresponding to that event. The resultant messages are filtered by the preprocessing hardware. The preprocessors collect, compress, and format the messages into manageable packets of information.

The user environment receives the results of RTDAD processing and provides a number of functions including visual monitoring of the processed data and some form of long term data logging (paper, magnetic tape, disc cartridge, etc.). In many systems, there is some form of manual input device that allows operating personnel to enter requests to the system. These requests provide for changes to processing subsystem parameters, interrogation of the system state, and display recall of portions of the system data base.

The RTDAD processing environment is clearly a data driven facility. The system responds to input data packets and user command packets as they are created. Furthermore, the general content of all input packets is precisely specified as part of the problem statement. The variation in

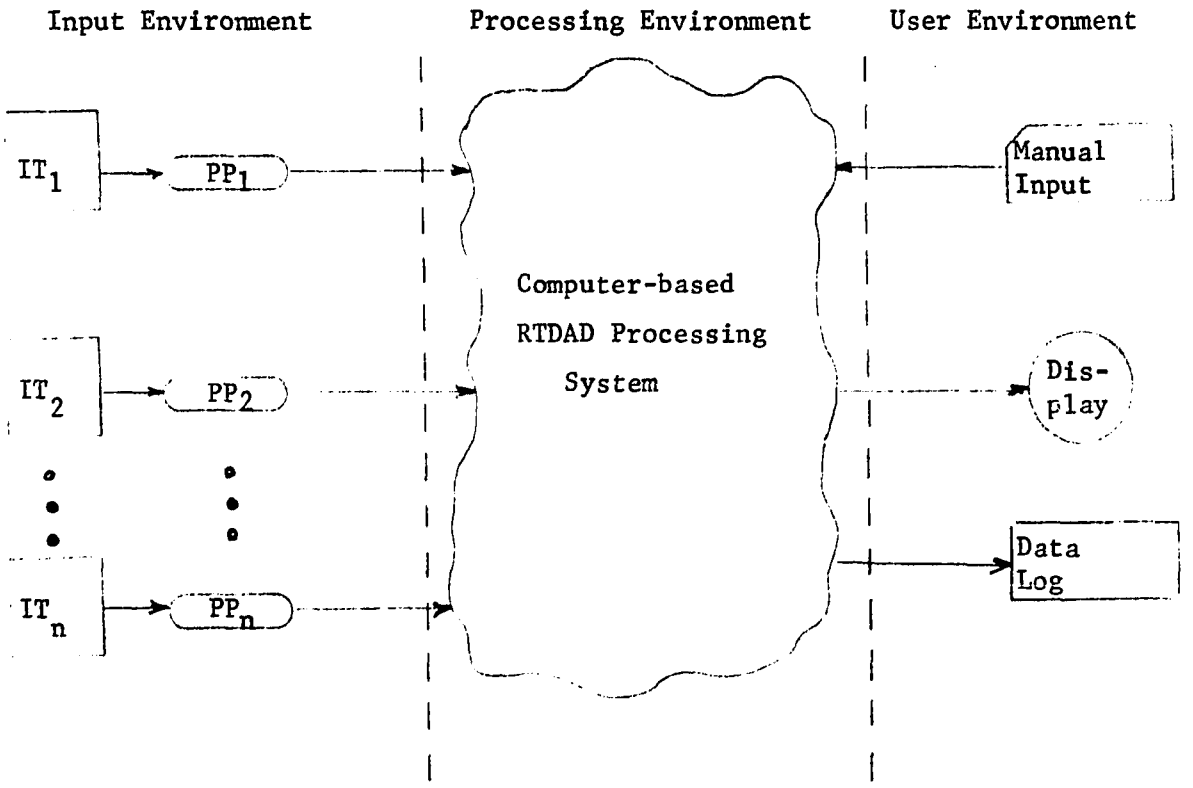


Figure 1.1. RTDAD system overview.

the RTDAD system state is, therefore, a function only of the current state of the system and the content of the input packets.

Since the input data format is known a priori, the functions or processes that operate on the incoming data are also known. Thus, the RTDAD facility can be viewed as a closed system of tasks. That is, all the required processes forming the operational structure of the system are completely defined. The system structure is, therefore, static in the sense that no processes with unknown resource demands are created (or destroyed) during operation of the facility.

Packets from the input transducers can be categorized according to the various types of events that are measured. For each packet type, a number of cooperating processes are required to move, reduce, distribute, catalog, and/or display the necessary results. Packets of different types are generally handled by disjoint processes which do not interact except for access to a common data base, data base storage device, or display facilities.

Conventional Implementation

An hardware implementation of an existing RTDAD facility that is typical of RTDAD systems is shown in Figure 1.2. The material in this section presents a short synopsis of the general composition of that configuration.

The preprocessing front-end provides for:

1. serial to parallel conversion,
2. error detection/correction,

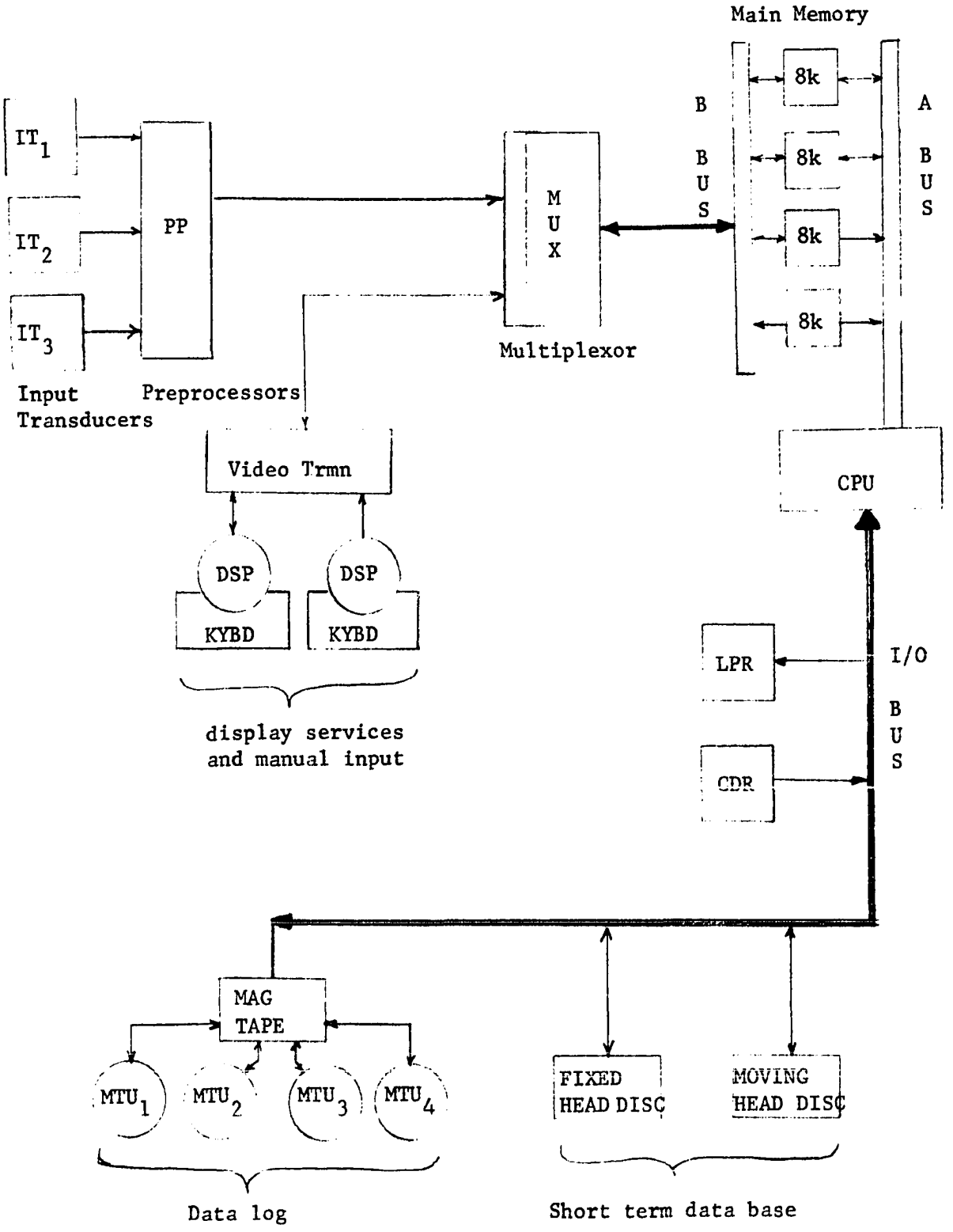


Figure 1.2. A conventional RTDAD system configuration.

3. timing and input data synchronization, and

4. data compression and routing.

The output from the preprocessor is fed directly to the computer main memory via a multiple channel direct memory access (DMA) multiplexor (MUX). The DMA capability provides a data transfer path directly into memory buffers without the use of additional processors other than the DMA. The DMA, therefore, appears as a memory user in much the same manner as the central processing unit (CPU). The multiplexor automatically buffers each data channel with priority interrupts generated at the completion of each block transfer.

Data is subsequently processed and filed in the common data base supported by the combination of the fixed head and moving head discs. Eventually, the data files are displayed and/or logged on the appropriate output devices.

The keyboards associated with the display devices provide for operator input of command and control information. Operator requests are used to control the overall operational state of the facility.

The operating system or real-time monitor is a typical general purpose software package similar to a number of systems now available from various computer vendors[1]. The operating system supports the virtual

---

[1]Example of vendor supplied operating systems include:
a. Digital Equipment Corporation (PDP-11) RSX-11A,
b. Interdata (70,80) RTOS,
c. Data General (Nova) RTOS, RDOS,
d. Hewlett-Packard (2100) RTE, and
e. Varian Data Machines (V73) VORTEX.

real-time machine functions needed to provide:

1. priority task dispatching,
2. time dependent task scheduling,
3. input/output services, and
4. dynamic memory management.

The processing environment is interrupt driven through the common CPU interrupt handling logic. That is, CPU processing activations (or reactivations) are dependent upon:

1. multiplexor block completion interrupts,
2. operator input requests,
3. real-time clock "ticks", and
4. I/O completion interrupts.

Conventional Design Sequence

A very interesting insight into the nature of RTDAD facilities can be gleaned from an examination of the generalized system design sequence. The design steps shown in Figure 1.3 are typified by the following scenario.

First, the problem statement is explicitly described; the input and user environment are precisely and accurately detailed. Next, armed with a functional overview of the processing requirements, the senior project personnel evaluate available general purpose real-time computer systems (hardware plus operating system software).

Based on their evaluation, one vendor is selected to supply the basic computer equipment which forms the core of the RTDAD processing

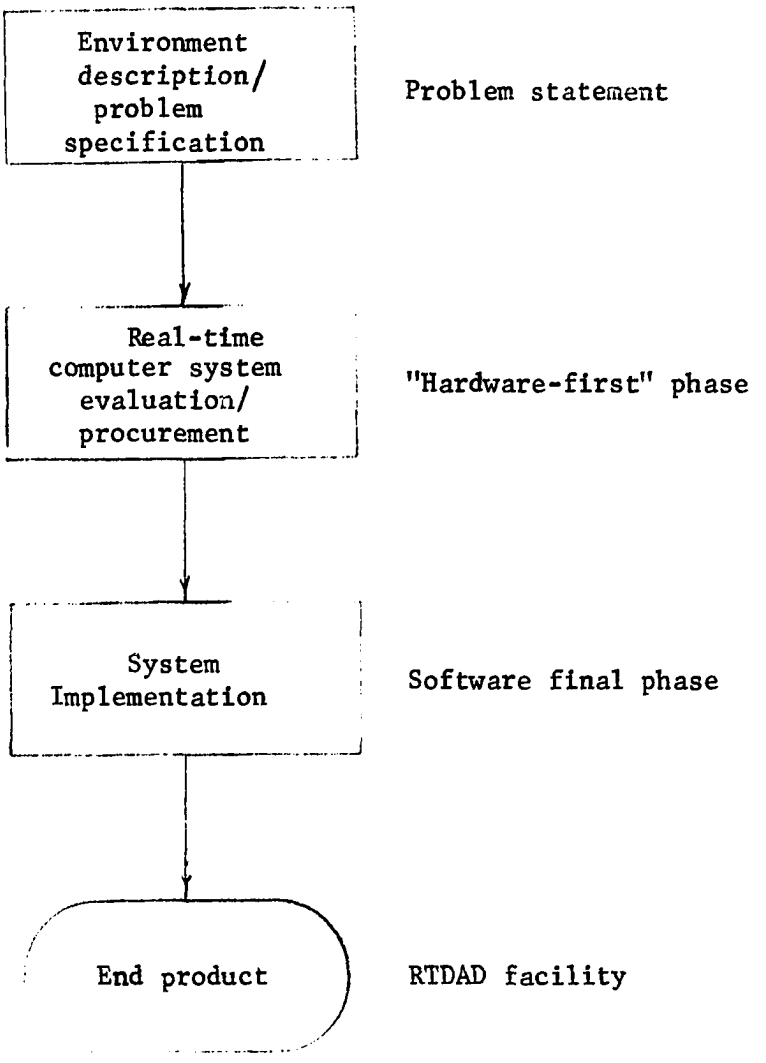


Figure 1.3. Conventional hardware-first design cycle.

system. Finally, the problem is consigned to the software group who is responsible for the modification of the vendor supplied operating system and development of applications software routines needed to complete the task.

The final step places the bulk of the design effort squarely in the hands of the software group. It is their task to make the configuration play as stipulated in the problem statement. Having been given a general purpose hardware base or bare machine, their responsibility is to provide a virtual machine that supports the functional requirements of the external environments.

The end result, therefore, is that instead of letting the problem form the computer solution, the computer solution generally deforms the problem. To substantiate this claim, the following critique of conventional implementations lists a number of system weaknesses resulting from the aforementioned design sequence.

1. concurrency - The RTDAD problem statement embodies potential parallelism in the number of disjoint processes which define system functions. This concurrency is not exploited by conventional architectures except for an overlap in some I/O operations (e.g. DMA) and CPU operations. The illusion of concurrency is created by the multiprogramming aspects of the real-time operating system which interleaves the execution of tasks.

2. scarce resources - The implementation shown in Figure 1.2 is a common general configuration for current RTDAD facilities. The uniprocessor structure, by definition, treats the central processing

unit as the primary scarce resource. The singular CPU is involved in every facet of system operation including task dispatching, memory allocation, I/O resource allocation, and application task execution. The CPU, as shown in Figure 1.2, is the central focal point for all operations in the entire system. The result is a complex interacting set of software modules required to create the virtual machine structure necessary for interfacing to the external environments.

3. protection - Noncorruption of the RTDAD system is dependent upon the "well-behaved" nature of both application tasks and system tasks. In some RTDAD implementations there is no protection provided between software modules. However, even under "lock and key" protection schemes, the additional protection is achieved by the inclusion of add-on hardware that is not an integral part of the problem solution.

4. operational discontinuities - The interrupt system is the prime mover in the RTDAD facility. All interrupts, regardless of origin and priority, are funnelled through the CPU hardware and associated software response modules. Each "tick" of the system clock, I/O device completion, etc. introduces a discontinuity in the processing environment of the entire system. A significant percentage of these interrupts is lost as processing overhead with the task dispatcher returning to the job which was interrupted.

5. functional overhead - The virtual machine functions of task dispatching (job control), I/O control, dynamic memory allocation, etc. do not contribute directly to system throughput, i.e., these routines are pure overhead. However, they do take a significant portion of the

processing capability of the uniprocessor.

6. side effects - Because functions executed by the CPU exhibit complex temporal and spatial interactions, changes in most modules can have numerous subtle side effects that are anything but immediately obvious. As an example, consider the case of one RTDAD installation in which the main memory space was dichotomized into a resident program area and a nonresident program area. A change in the problem statement required the addition of a number of resident program modules. As a result of the additional resident routines, the amount of nonresident memory space decreased. These subsequent reductions manifested themselves in the form of reduced response time for a number of nonresident tasks including display functions and operator communication tasks. The tradeoffs between resident memory size and nonresident task execution speed is complex and extremely difficult to quantify.

7. extendability - A major concern in the uniprocessor implementation is the ability to adapt to changing requirements. The typical system specification is in a constant state of change from the time the first internal specification is released. With the amount and complex nature of interactions among software modules, the actual processing capability (both potential and consumed) is difficult to measure. In particular, there are no current methods which provide for measurement of system usage or unused capacity. If the processing requirements exceed the capacity of the uniprocessor design, catastrophic results are attained. A change in configuration

(e.g., adding processors, memory, etc.) generally requires a significant system reorganization (if not a complete loss of the original system).

In summary, the conventional design process contorts the problem statement and computer based solution until they map into the same design space. The design process never questions the amenability of the conventional general purpose architecture to the original problem definition.

Trends of Change

Conventional systems work. We are not arguing that they produce illogical or incorrect solutions. We do contend, however, that there are significant changes afoot which can provide more cost-effective alternatives (both short-term and long-term). In particular, three important trends that will have substantial impact on RTDAD installation are:

1. decreasing hardware cost in the area of LSI and microprocessors,
2. the exorbitant cost (both short-term and long-term) of typical software solutions, and
3. the introduction of structured design techniques.

Hardware impact

The current literature abounds with articles emphasizing the plummeting costs of hardware and the dramatically increasing functional capability available in small, reliable integrated circuit components (7, 14, 25, 54). For example, Hodges (25) predicts the existence of a complete minicomputer system chip, including a 16-bit CPU, 32 kbits

of read-only and/or read/write memory, and simple input/output (I/O) interfaces by the early 1980s. Even more dramatic is his estimate of manufacturing costs of the chips at ten dollars or less.

Semiconductor main memory costs have declined rapidly over the past decade, primarily because of the increasing density of LSI memory chips. Further large increases in memory density are expected with 64 kbit chips feasible before 1980.

The primary motivation for increasing LSI circuit complexity continues to be lower cost per function and improved reliability. Chip-count continues to be the best first-order measure for electronic hardware cost and reliability. Any LSI circuit which can be manufactured in volume will cost less than ten dollars at the factory. Mean time between failures of about ten million hours per packaged chip is easily obtained by simple means (25).

The point is that there is an availability of large quantities of inexpensive, reliable off-the-shelf components or primitive modules (functional units, central processors, I/O interfaces, and memory elements) that can be used in solving the functional requirements of any RTDAD problem statement[1].

Software incurred costs

An interesting fact which has become painfully apparent in the last few years is that software costs are dominating computing and future

---

[1] We will continue to restrict our attention to components which are now available or will be available in the near future. This implies a more conventional sequential execution of functions defining the problem solutions. The impact of this assumption will become apparent shortly.

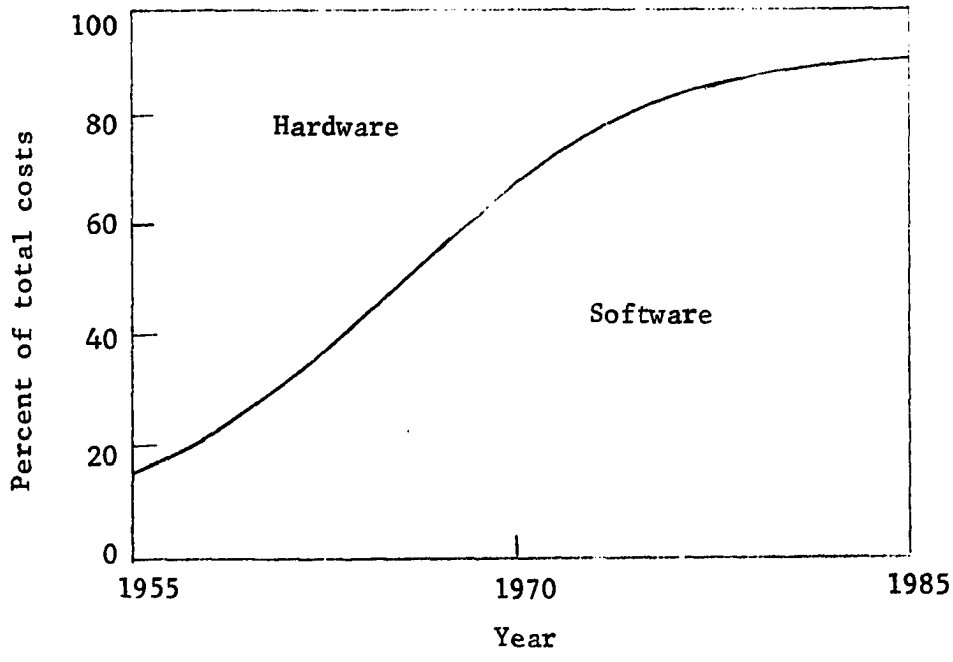
prospects are for much of the same (34). Based on a Rand Corporation study for the United States Air Force, Boehm (2) alleges that software will be the major source of difficult problems and operational performance penalties.

Two particularly salient points made in the study are:

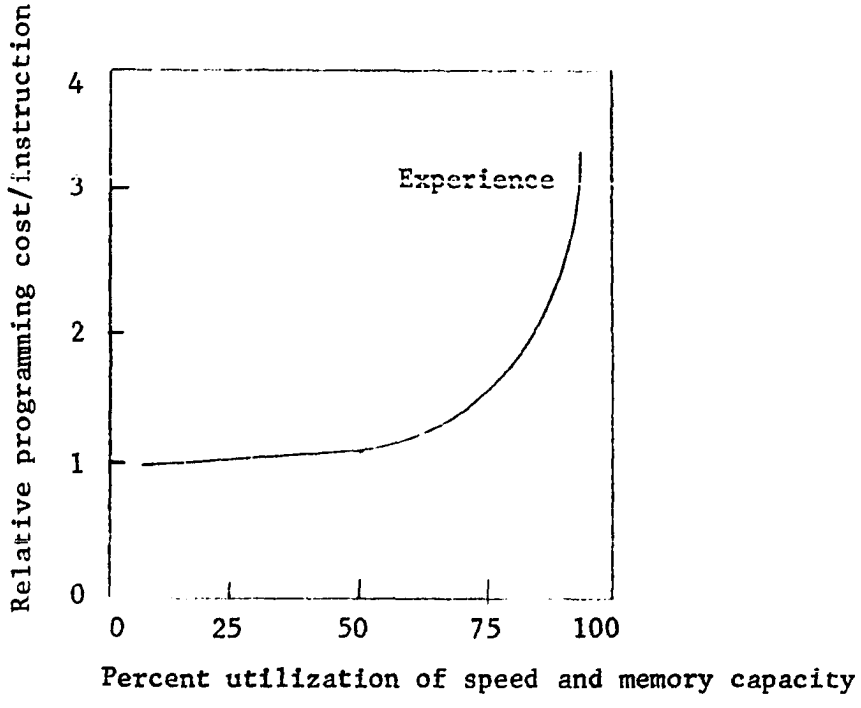
1. In 1970, the software-hardware cost ratio realized by the Air Force was 65/35. If the current trends in computing prevail, the cost ratio will swell to 90/10 by 1985 (see Figure 1.4a).
2. Hardware constraints affect software costs in an exponential manner as complete utilization of processing speed and memory capacity is achieved. The variation (software cost versus hardware utilization is shown in Figure 1.4b). Boehm emphasizes that in the conventional approach one should buy more hardware capacity by 50% to 100% than what is needed. In particular, it is far easier to err by procuring a computer that is too small than one that is too large.

Structured programming

There is vast amount of literature available describing the nature of structured programming (8, 9, 15-18, 30, 31, 35, 38, 45-47, 63). Although not all people concur on the absolute definition of structured design, there is overwhelming agreement that such an approach yields significant coherency to the design and maintenance of the resultant systems. More specifically, the trend in software development is placing efficiency considerations subordinate to clear logical structuring (34).



a. Hardware/software cost trends.



b. Hardware constraints on software production.

Figure 1.4. Software cost trends.

We will not propose another definition of structured design; nor will we attempt to distill one definition from the available literature. Instead, let us suggest, in agreement with Patil (47), that the results of structured design, independent of the approach, should embody three general characteristics.

1. Composition - The system should be composed of well-defined parts which play well-defined (and preferably intuitively appealing) roles in the operation of the system. The parts may be arranged in many levels of hierarchy; at each level, parts may be interconnected with well-defined links to form a network of parts, and a (composite) part at one level may be further expanded (defined) at lower levels of hierarchy.
2. Communication - The logical interaction among parts takes place only through the defined links and follows a communication discipline (a communication mechanism) set forth for the system; the parts should be logically independent of each other except for the explicit interaction set forth in the description of the system.
3. Comprehension - The complexity of the system at each level of description should be within the limits of comprehension of the agent trying to understand the operation of the system or verify its correctness.

The Impact of Changing Computer Trends

The high cost of software and plummeting hardware costs suggest that the conventional approach to system design be reevaluated. The hardware-

first approach which incorporates a final complex software development phase is no longer producing the viable cost-effective RTDAD solutions that were once possible.

With an increasing number of hardware building blocks becoming available, a new insight into the design process needs to be established. Additionally, the nature of the resulting system architectures needs to be examined. The typical dichotomy of a hardware base and a superstructure of software modules may no longer be the correct representation of the end product that results from an application of the new design approach.

Furthermore, we contend that the concepts inherent in structured programming should be extended to encompass all facets of system design. No longer is pure efficiency the primal consideration. Additional features such as simplicity, maintainability, and reliability must be taken into account at all stages of the design process.

System Models

There are a number of articles concerning computer architectures in the current literature suggesting systems that take advantage of the decreasing costs in hardware (10, 20, 27, 62, 64). The proposals typically amount to organizations of hardware modules in some rather complex structure based on the premise that if one processing element is good, then a conglomeration of similar processing elements must be significantly better. However, these architectures still fall under the hardware-first design category. The C.mmp (64) is a notable example of this

form of thinking. The complexities inherent in the development of the associated software system (65) still remain a formidable task.

All this leads us to believe that we need some form of system model or models that allow us to take advantage of all the current trends in computing. A number of models have been suggested for portraying system behavior. Representatives among these are:

1. the architectural design language (ADL) of Chu (9),
2. computer-aided design models such as LOGOS (53) or Evaluation nets (40-44), and
3. models of parallel computation (1, 36, 37) including models suggested by Martin and Estrin (32, 33), Karp and Miller (28), and Dennis (11, 12, 48).

Essentially, Chu, in his ADL, advocates a developmental language that allows for the deferment of implementation binding (hardware, firmware, or software) by providing an abstract language for the definition of system functions. However, he does not provide a model of system behavior in terms of overall system structure. As is true of all the models, ADL is not capable of depicting the impact of resources (hardware and software) on the resultant system structure; nor are any of the models (excluding Evaluation nets) capable of directly representing time as an integral part of their representation.

Additionally, the computer aided design models require extensive and detailed computer simulation runs. We are interested in developing an analytical method of deriving system performance that admits to direct numerical determination of system performance bounds without recourse

to time consuming and expensive computer runs. This is particularly true since the early design stages require a great deal of cut-and-try effort that depends upon a model that is easily modifiable and readily produces numerical results.

Since we have restricted the primitive modules to sequential components, the study of micro-parallelism achievable by models of parallel computation are not directly applicable. We are essentially interested in the exploitation of macro-parallelism that is achieved by functionally allocating physical resources to the problem solution.

However, an adaptation of one of the models of parallel computation, the Petri net, is extremely appealing. The concept of the Petri nets, described by Holt and Commoner (26), has been extended by Ramchandi (52) to the Timed Petri net. Timed Petri nets are particularly attractive as they:

1. directly depict concurrency (macro-parallelism),
2. indicate the impact of the use of hardware and software resources by an analytical determination of system performance,
3. explicitly include timing constraints, and
4. admit to structured design techniques (they are particularly well suited to a multilevel hierarchical description of system structure).

Statement of Work

In summary, the purpose of this dissertation is two fold:

1. Propose an alternate RTDAD system architecture which results from the coherent, integrated application of all available

technologies (hardware, firmware and software). This architecture will be referred to as a multicentered structure (MCS) architecture.

2. Suggest two possible graph-theoretic tools which can be used in the development of MCS systems.

The resultant design sequence which is a product of the proposed alternate approach is shown in Figure 1.5. The design process emphasizes two phases of development. First, the System State Model supports an implementation-independent, functional specification of the problem to be solved. Finally, based on that functional definition, an implementation involving hardware and software resources is derived.

The subsequent chapters in this presentation serve as an expansion of the process depicted by Figure 1.5. Chapter 2 describes the general nature of the MCS structures. Chapter 3 deals with the System State Model and Chapter 4 discusses Timed Petri nets. Chapter 5 brings all the material together with a discussion of the applicability of the proposed models to the RTDAD problem. Additionally, Chapter 5 includes several examples indicating the power of this approach.

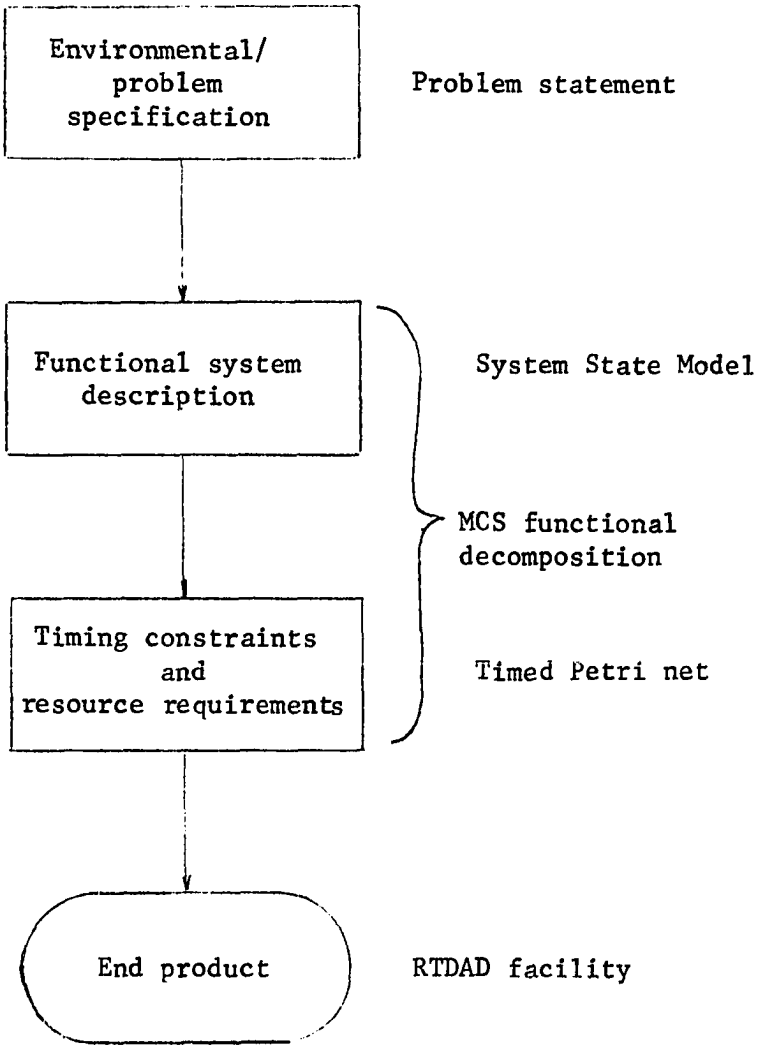


Figure 1.5. MCS system design sequence.

CHAPTER 2

MULTICENTERED STRUCTURES

Introduction

The solution to a given RTDAD problem is a logical engine or machine M whose functional characteristics precisely match those of the problem statement. In particular, the problem statement consists of a specification of:

1. the information input by the external environments,
2. the operations to be performed on that data, and
3. the resultant output formats to be generated.

The input information received by M from the external environment can be perceived as a set of data and control packets. M decodes the input packets and performs sequences of operations as a function of the content of those packets and the current state of the system.

Each input packet, therefore, represents an instruction for machine M. The set of instructions which are input to M constitute the language L of that machine. The design process amounts to the construction of a machine M whose language is L.

If a given hardware configuration embodies all those characteristics defined by L, then that hardware machine is one possible solution. In practice, however, such a solution rarely exists.[1] Instead, the design

---

[1] In general, pure hardware solutions are also impractical due to the rigidity of their design. That is, they lack adaptability, extendability and maintainability which are desired system properties.

process involves a sequence of decisions or steps which transform the language L of machine M into a primitive language L' of machine M' where M' can be realized by a set of primitive hardware modules.

The following material in Chapter 2 presents a rudimentary definition of languages and the machines they represent. Additionally, an abstract framework is suggested for the global description of system characteristics. Of particular interest in this description is:

1. the decomposition of the total solution into a hierarchical structure of machine-language levels, and
2. the repeated application of available technologies (hardware, firmware, and software) to the problem solution at all levels of the decomposition.

Language

In a very broad sense, a language is a vehicle for the expression or communication of meaningful thoughts or ideas between two or more entities[1]. Thus a language may be viewed as the logical medium by which a flow of information is conveyed between a source and a destination. A general language model is shown in Figure 2.1.

The existence of a language between a source and a destination is a mandatory requirement if the transmission of information between the two is to be successful. Additionally, the language must be precisely defined and mutually understood. An ambigiuty in the semantics of elements

---

[1] Meaningful in the sense that the communication results in the transmission of a quantity of information. Whether the information so realized is meaningful at the receiver is a different problem.

in the language will result in the erroneous interpretation of the information received.

In the context of computer systems, a language, L_j, may be defined as a finite set of instructions, Ins_i, which can be interpreted (i.e., recognized and executed) by a given logic engine or machine. That is,

$$L_j = (Ins_1, Ins_2, \ldots, Ins_n)$$

The language L_j consists of two types of instructions:

1. imperative - Instruction Ins_i is imperative if it invokes an action or causes the execution of an activity (e.g., the addition of an element to a LIFO stack represented by the instruction PUSH (Element, Stack-name).).
2. declarative - Instruction Ins_i is declarative if it makes an assertion which can be used by an imperative instruction sequence (e.g., the structure of a data object is defined by a declarative instruction).

The language model for digital machines is given in Figure 2.2. In contrast to the general language model, the roles of the source and destination are fixed in the digital model. That is, the same logical entity does not alternate between being a source and a destination. Thus, the communication rules established between the source and the destination are unilaterally defined by the destination.

As an example, consider the case where the source is a user and the destination is a machine capable of directly executing Fortran programs written by the user (i.e. the machine is a Fortran machine).

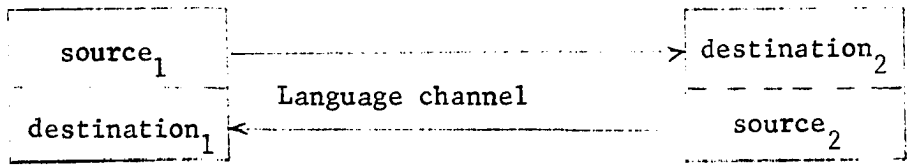


Figure 2.1. General language model.

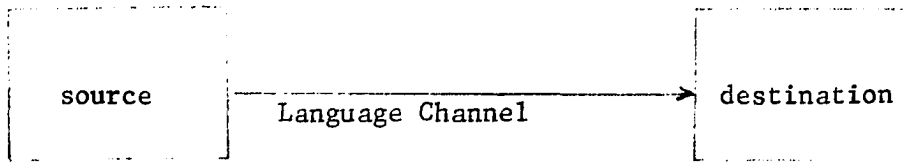


Figure 2.2. Digital system language model.

The communication channel between the user and the destination is clearly the Fortran language as defined unilaterally by the Fortran machine.

Implementation Sets

There is an important relation between a language and a machine. Each machine, M_i, communicates with its external environment via its communication channel or language L_i. Conversely, L_i defines the interface rules by which the external environment communicates with M_i.

The machine M_i can be defined to be a pair

$$M_i = (L_i, H_i)$$

where

L_i is the language of the machine M_i, and

H_i is a particular realization or implementation of M_i which supports L_i.

There is nothing unique about the pair (L_i, H_i). That is, M_i could also be represented by a different implementation H_i', i.e.,

$$M_i = (L_i, H_i')$$

Therefore, there exists a one-to-many mapping between the language L_i and the machine M_i which executes that language.

For example, there are numerous machines which can execute programs written in ALGOL[1]. Thus the terminology an "ALGOL-machine" represents

---

[1]Assuming that all such machines are true to the definition of ALGOL in its totality - an assumption which does not generally hold.

a class or implementation set of such machines. Given a machine in the implementation set and a program written in ALGOL, one could not perceive a difference in machines without specific reference to performance measures (time of execution, etc.).

Thus, a machine uniquely defines a language. But, given a language, there is a class of implementations (implementation set) which can be used to realize the associated machine. The following section discusses machines and the language they represent without explicit reference to any implementation. In fact, it is the deferment of the binding of a language to an implementation which is emphasized throughout the subsequent material.

Virtual Machines

A language has been specified as the communication link between a source and a destination. Furthermore, the utility of such a language was predicated on the mutually understood properties of the language by the source and the destination. In many cases, however, a given source S and destination D have a language interface specification in which the requirements of S and the language defined by D are not compatible. To bridge that gap, one or more intermediate machines are required.

The concept of a multiplicity of machines is best demonstrated by an example. Suppose in the model of Figure 2.2, the destination is a conventional, commercially available computer M_1. Additionally, suppose that the source is a user whose application requires the numerical addition of two integer arrays (refer to Figure 2.3).

User

$M_1 = (L_1, H_1)$

User requirements

input integer array

add integer array

output integer array

Destination Language $L_1 = (Ins_1, Ins_2, \ldots, Ins_m)$

Instruction	Format	Operation
Ins_1	101 | A | M	load the register named A with the contents of memory location M
Ins_2	102 | A | M	add the contents of memory location M to the register named A
Ins_3	103 | A | M	store the register named A into memory location M
⋮	⋮	⋮

Figure 2.3. Simple array problem.

In the model of Figure 2.3, there is a compatibility gap between the user language requirement and the machine supported language L_1. To compensate for this difference, an intermediate language or languages must be developed which will provide the appropriate language interface(s). In particular, consider the revision to this model shown in Figure 2.4. The language L_2 includes instructions which support the integer array functions input, output, and add. The model depicts a sequence of two machines (M_1 and M_2) which provide a solution for the original problem defined by the user environment.

Machine M_2 may be generated in one of two different but equivalent manners. First, L_2 can be defined by a process known as translation. Under translation, the user specifies his requirements in terms of the instructions of a third language L_T (e.g. PL1, ALGOL, BCPL, etc.). These instructions are converted by a translator into an equivalent sequence of instructions from language L_1. That is, each instruction of L_T in the user-created program is expanded into an equivalent set of instructions from L_1. Finally, the translated program is executed directly on machine M_1. The resultant functions, which are procedures consisting of instructions from L_1, define the language interface for machine M_2.

Note that L_T and L_2 could be identical. In general, however, they will be significantly different. Furthermore, the same programming language L_T could be used to generate a sequence of machines which define n-1 language interfaces (L_2, L_3, ..., L_n).

Alternatively, the second process for the definition of the intermediate language is known as interpretation. In interpretation, M_2

is constructed from procedures written directly in language L_1. These procedures decode and execute (i.e., interpret) the instruction strings or source requests written in L_2.

Regardless of the process employed, translation or interpretation, two consistent features of the resultant machines are:

1. both machines consist of functions which are defined in terms of language L_1, and
2. at execution time both machines support the same language L_2.

In order to avoid the need for distinction between translation and interpretation, let machine M_2 be defined as a virtual machine VM_2 (59), i.e.,

$$VM_2 = M_2 = (L_2, H_2).$$

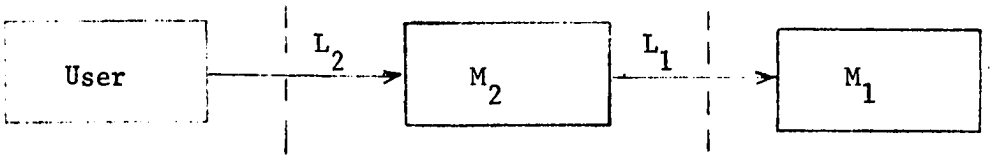
One may perceive of the virtual machine VM_2 as a machine that embodies the essence of a real machine (the ability to interpret instructions of a language), but whose functions are based upon a more primitive execution entity, the real machine, RM. In contrast, the real machine RM can directly execute the instructions of the language which it defines.

A final picture of the simple array example is given in Figure 2.5.

Machine Levels

As shown in the preceding example, a given system realization may consist of a multiplicity of machines. Each machine defines a machine-language level or level of abstraction. The model of Figure 2.5 has two levels, namely M_1 and M_2.

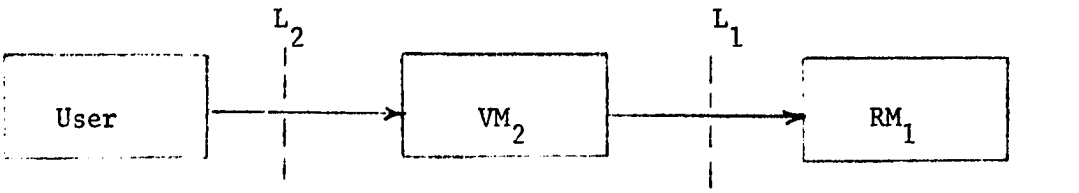
The merit of the use of levels of abstraction have been emphasized by Dijkstra (17), Liskov (29,31) and Parnas (45,46). Conceptually, the



$L_2 = (Ins_1, Ins_2, \ldots, Ins_m)$ $M_2 = (L_2, H_2)$

Instruction	Format	Operation
Ins_1	input-array(A)	Input the elements of array A.
Ins_2	add-array(A, B, C)	Place the result of the element by element sum of array A and array B into array C.
Ins_3	output-array(A)	Output the elements of array A.

Figure 2.4. Multiple machines for simple array example.



$VM_2 = M_2 = (L_2, H_2)$ $RM_1 = M_1 = (L_1, H_1)$

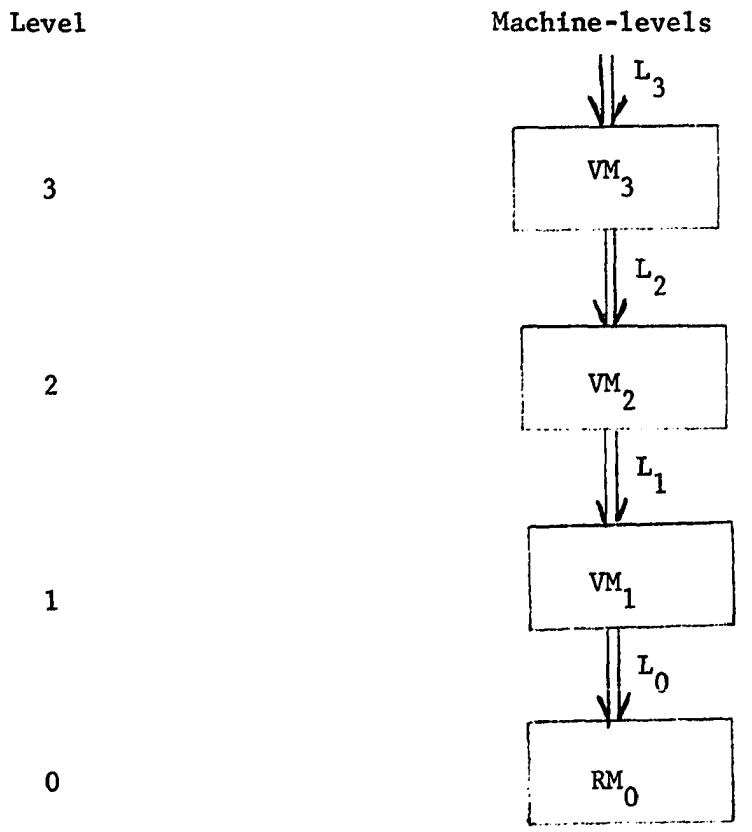
Figure 2.5. Virtual machine for simple array example.

levels of abstraction deal with the removal or suppression of irrelevant details as succeeding (higher) levels of system description are developed. By isolating the various functions on a level basis and establishing clean and well defined channels of communication (languages) between levels, the complexity of design and debugging is considerably reduced. The result is a hierarchical abstract machine decomposition which provides a systematic and manageable procedure for the design of computer systems (56).

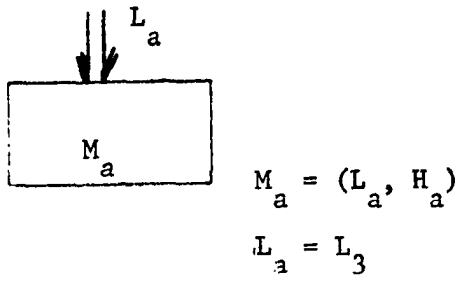
For example, in Figure 2.5, machine M_2 supports the operation on integer arrays. For all practical purposes, the sequence of instructions in language L_1 required to implement M_2 is irrelevant to the user. His major concern is that the language L_2 provides those instructions which are required to meet his problem specification.

A multilevel structure will be represented diagramatically by a graphical tool referred to as a Block Level Diagram (BLD). This notation, which has been casually used in the preceding example, consists of a sequence of machine-blocks interconnected by directed arrows. The blocks represent either virtual machines (VM_i) or real machines (RM_i). The directed arrows are used to depict the language interfaces defined by the machine-blocks upon which the arrow-heads are incident.

An example of a BLD is shown in Figure 2.6a. Figure 2.6b gives the equivalent collapsed description of the machine M_a represented by all the levels contained in Figure 2.6a. The BLD depicts a functional dependency of one machine level upon the preceding levels. This dependency defines an hierarchical access structure which describes an accessibility relation (the "uses" relation of Parnas (45)).



a. Typical block level diagram (BLD).



b. Collapsed machine representation.

Figure 2.6. The block level diagram (BLD).

In terms of the relation, a given level may be classified as dependent or independent. An independent level represents an execution entity, i.e., a real machine RM. The machine provided by such a level can directly execute the instructions defining its language without further recourse to supporting levels of the machine structure.

In contrast, a dependent level is functionally dependent upon the next lower level for support. It is important to note that not all real machine levels are necessarily independent. For example, Figure 2.7 depicts a machine that has two separate hardware levels of which RM_1 is functionally dependent upon RM_0.

Additionally, there is no reason to believe that all machines have virtual machine levels. That is, the required machine could be realized by a total hardware implementation.

Thus far, for the sake of simplicity, the machine descriptions have been limited to only a few levels of abstraction. In general, the solution for a given problem statement involves a number of such levels. The number of levels depends upon the functional decomposition of the system. To some extent the grouping of functions into levels is somewhat arbitrary. Probably one of the most difficult phases of the design process is the specification of the appropriate sequence of such levels.

Level Transformations

A Block Level Diagram graphically displays the transformation of an independent real machine, $M_0 = RM_0$, via a number of machine levels to a final machine M_{FT}. Symbolically, the transformation can be represented:

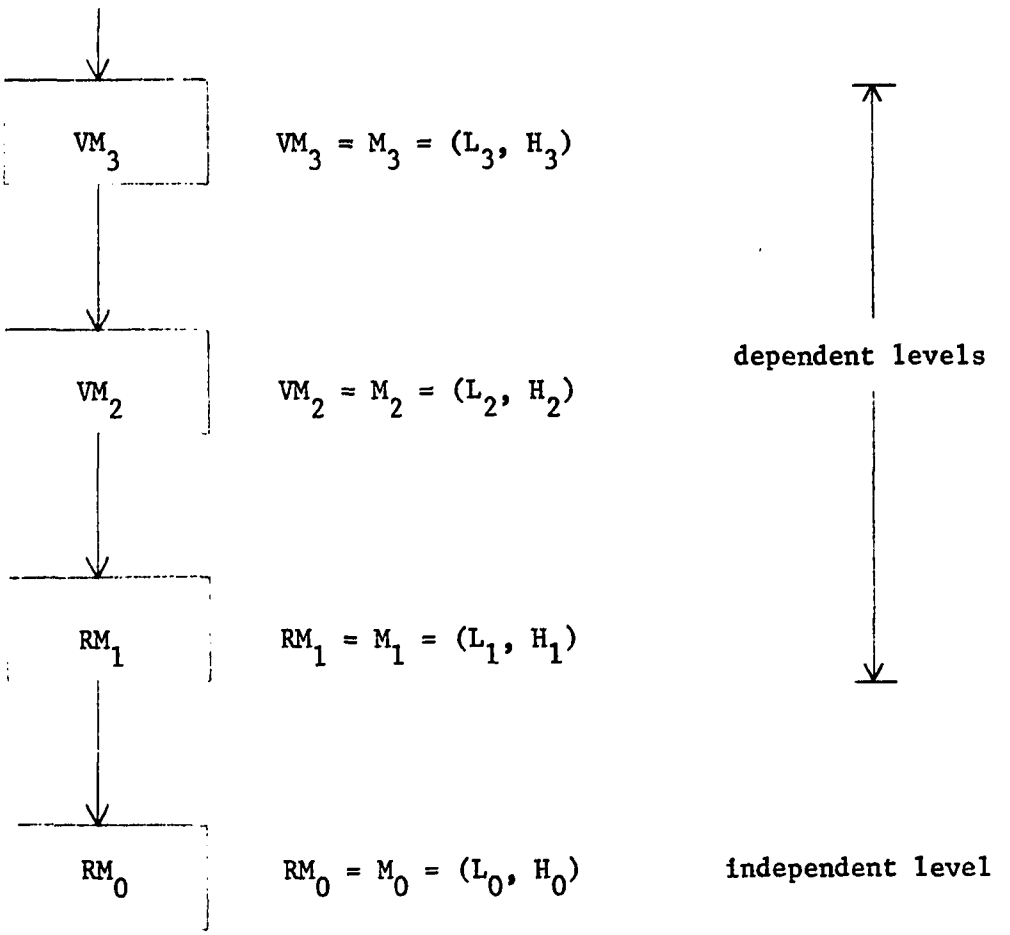


Figure 2.7. Multiple hardware levels.

$$M_{FT} \xleftarrow{Z} M_0$$

where

$$Z = \{Z_i \mid 0 \leq i \leq FT\}$$

$$M_i \xleftarrow{Z_i} M_{i-1} \qquad i = 1,2, \ldots, FT$$

and

$$M_0 \xleftarrow{Z_0} \text{(primitive modules)}$$

At this point a short digression to describe the nature of the transformation set Z is in order. The element Z_0 describes the collection of (an interconnection between) hardware components which are used to implement $M_0 = RM_0$. These components are referred to as primitive modules. Primitive modules include, but are not restricted to, memory elements, processing elements, and bussing structures.

The transformation Z_j ($j > 0$) is a set of transform elements denoted:

$$Z_j = \{Z_{j1}, Z_{j2}, \ldots Z_{jn}\}$$

The Z_{jk} may be characterized by the nature of the transmission of capabilities from one machine level to the next. Let $\bar{K}^i$ be a subset of instructions in L_i. Then Z_{jk} may be classified as:

1. functional transform element - $\bar{K}^{i-1} \xrightarrow{Z_{jk}} \bar{K}^i$ $(\bar{K}^i \neq \bar{K}^{i-1})$.
   Z_{jk} transforms a subset of instructions at level i-1 into a new set of instructions at level i.
2. identity transform element - $\bar{K}^{i-1} \xrightarrow{I} \bar{K}^i$ $(\bar{K}^i = \bar{K}^{i-1})$.
   Z_{jk} is the identity transform, I, which maps all the instructions

of $\bar{K}^{i-1}$ into $\bar{K}^{i}$ of the next higher level.

3. null transform element - $\bar{K}^{i-1} \xrightarrow{\Lambda} \emptyset$. Z_{jk} is the null transform, Λ, which maps the instruction of $\bar{K}^{i-1}$ into the empty class $\emptyset$. That is, instructions in $\bar{K}^{i-1}$ available at level (i-1) are not available at level i.

Based on the capability transmission criterion, the instructions at level i are partitioned into three disjoint classes as a function of the nature of their transformation at the i^{th} level. Specifically, these three classes are given in Table 2.1.

An example of a transform set used to define a machine M_{FT} is given in Figure 2.8. Each machine level is shown functionally decomposed into its constituent modules (software and firmware procedures for the virtual machines and primitive modules for the real machine). For example, the transformation for level 1 is given by:

$$M_1 \xleftarrow{Z_1} M_0$$

$$Z_1 = \{Z_{11}, Z_{12}, Z_{13}, I, \Lambda\}$$

where

$$Z_{11} = P_{11}\ ,\ Z_{12} = P_{12},\ \text{and}\ Z_{13} = P_{13}.$$

Each directed arc references or represents an instruction. For example, the directed arc A1 emanates from procedure P_{12}. Similarly, the procedure P_{13} references the subset of instructions corresponding to the arcs emanating from that procedure.

The concept of the null transform is explicitly shown by arrows from the input instructions at a given level to the null element (Λ) at that

Table 2.1. Instruction transformation classes.

Instruction	Transmission Capability		
Class	Functional	Identity	Null
I	X	X	
II	X		X
III		X	

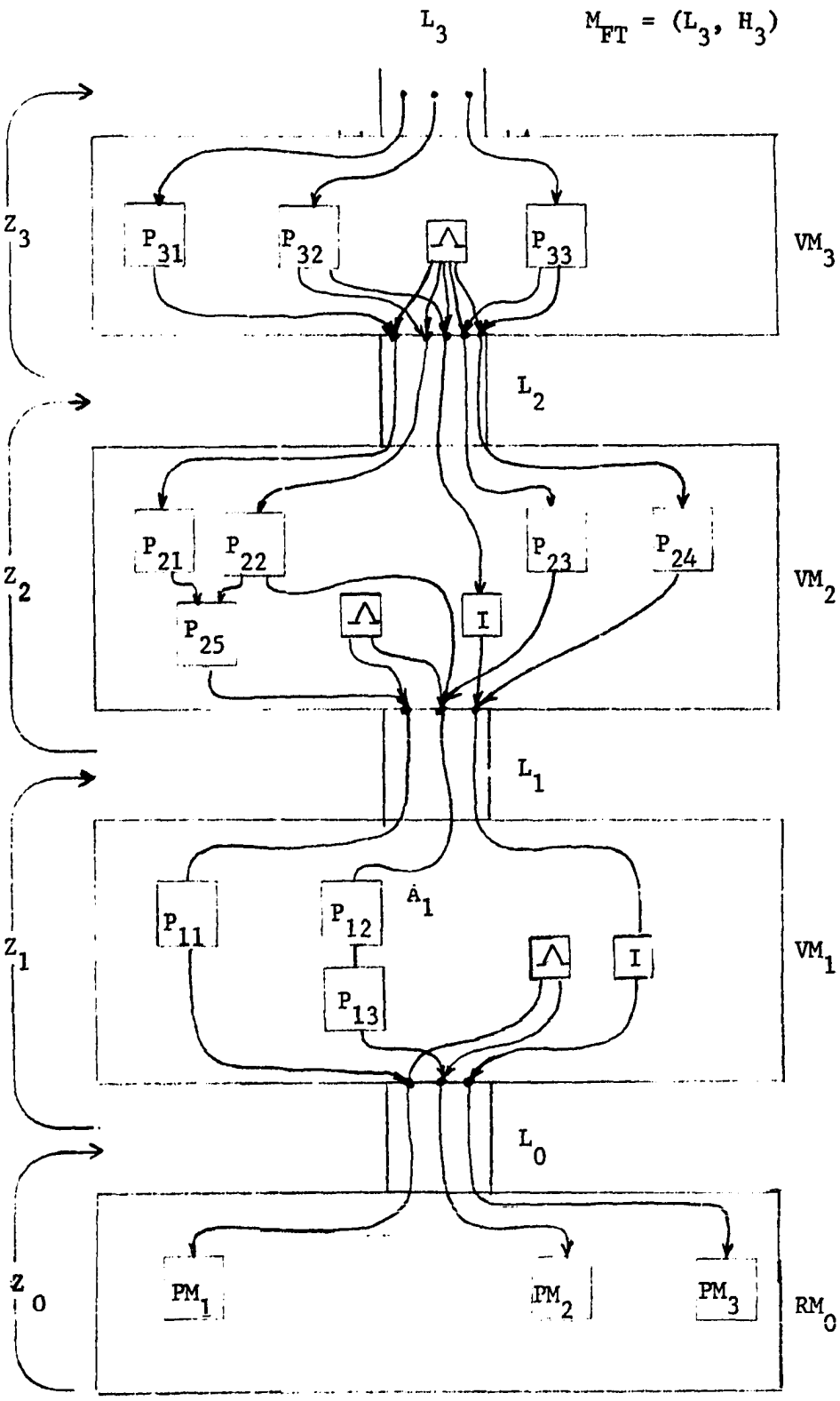


Figure 2.8. Transform set $Z = (Z_0, Z_1, Z_2, Z_3)$.

level. In general, the null transforms are self-evident from the structure of the BLD and will be omitted for clarity.

Decomposition Criteria

The utility of a given Block Level Diagram is dependent upon whether or not the diagram is a realization of a meaningful decomposition. The criteria for the definition of such a meaningful decomposition is threefold (23):

1. level simplification - Higher order levels (numerically larger level indicies) must provide machine levels which are easier to understand and use. Particularly, the higher level machine must support a language which more directly relates to the application environment. The ease of use measure may be quantified at a given level in terms of the number of instructions or length of the source programs which is required to implement the desired functions at that level.
2. hierarchical description - The level diagram must accurately depict a hierarchical access structure. Such a structure defines an accessibility relation in which:
   a. M_i $(i > 0)$ is defined in terms of M_{i-1}, and
   b. M_i cannot use facilities which are not explicitly provided by M_{i-1}. Thus, M_i does not have access to M_j where $j > i$.
3. consistency - Each level must be consistent in its structure with respect to:

a. availability - If $\overline{K}^m$ is accessible to M_n then $\overline{K}^m$ is also accessible to all M_i where $m < i \leq n$.

b. concealment - If $\overline{K}^m$ is not accessible to M_n for $n > m$ then $\overline{K}^m$ is also not accessible to M_i where $i \geq n$.

The concealment property (3b) is extremely important. The major thrust of concealment is inherent in the very nature of the definition of abstraction. Once a set of functions has been virtualized, the only interface of interest to higher levels is the language required to use that set of functions. With respect to the higher level, the nature of the particular functional composition at lower levels is irrelevant. It is the resultant language elements available to the higher level machine which are relevant to those levels.

As a consequence, the particular implementation of a given level may change as long as the language interface to the accessing machine remains unaltered. Thus, if new implementations are devised to "improve" performance, the dependent higher levels are left essentially unchanged.

Machine Contours

The BLD introduced in the previous section is a tool for representing the hierarchical structure of a computer system. In particular, the BLD portrays a system as a sequence of machine layers or levels of abstraction.

An additional aid for displaying the structure of systems is the contour diagram. A contour is a continuous closed curve which represents a machine. Furthermore, the contour encloses or contains one or more machines whose language or languages define access points on the contour.

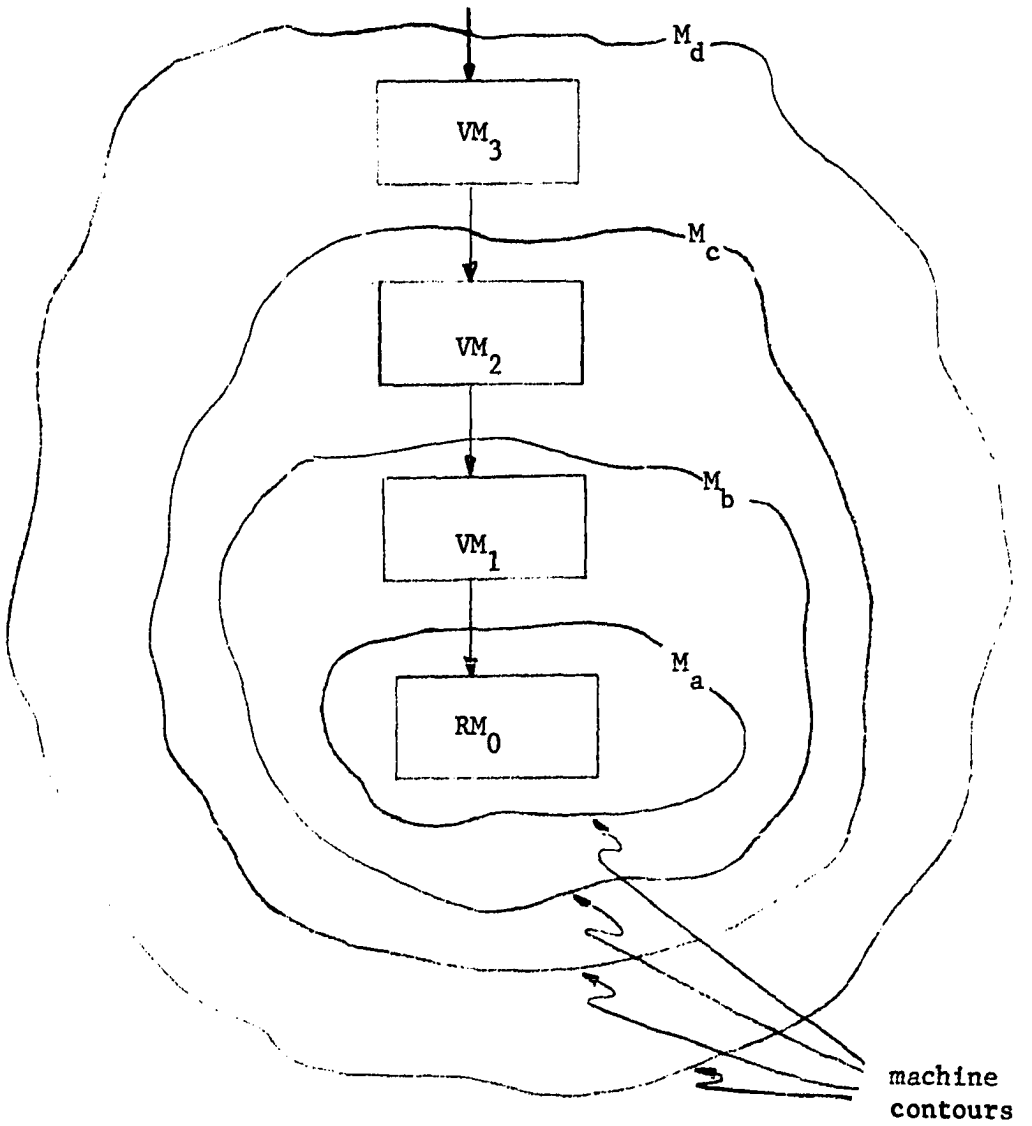
The concept of a contour comes from contour lines which are imaginary lines on a map that connect all points of the same elevation. In the context of computer systems, the contour is an imaginary closed curve connecting instructions (points) at the same machine level (elevation).

However, a significant difference is that the establishment of machine contours is somewhat arbitrary as compared to contour lines. Machines contours or more simply contours delineate logical machine layers. Each layer is chosen on the basis of a logical grouping of the functions defined by the machines enclosed within the contour.

An example of a contour diagram applied to the machine of Figure 2.8 is given in Figure 2.9. In order to distinguish machines defined by contours from machine-blocks of the BLD, the subscript for machine contours are lower case letters. Note, for example, that machine M_d is equivalent in Figure 2.9 to machine VM_3. As will be seen shortly, an equivalence between machine contours and machine-blocks does not generally exist.

The concept of machine dependency carries over from the BLD to the contour diagrams. In particular, a contour which directly contains an independent real machine and no dependent machines is termed an <u>independent contour</u>. All other contours containing a combination of dependent and independent machines are referred to as <u>dependent contours</u>.

The employment of machine layers to represent levels of abstraction has been used by Gagliardi (21) among others. In his work the machine layers appear as contours describing a sequence of machines in much the same manner as the example of Figure 2.9. Specifically, his layer diagram



$M_d = VM_3 = (L_3, H_3)$
$M_a = RM_0 = (L_0, H_0)$

Figure 2.9. Machine contour diagram.

depicts one independent contour and a sequence of dependent contours. In subsequent sections, we will extend the idea of contour diagrams to include more general logical machine structures.

It is important to note that, although contour diagrams can be used to represent logical system structures, the contour diagrams are only an artifice. They depend upon the definition of a BLD for their substance. That is, the contour diagrams are a descriptive aid which are derivable from a BLD. The essential element missing from the contour diagram is an explicit representation of the accessibility relation defined by the BLD.

Level Diagram Example

A concrete example of a level diagram is appropriate at this point in the discussion. Figure 2.10 depicts a contour diagram for an existing RTDAD system. A complete description of the attributes of each machine-level would be exhausting. However, a terse synopsis of each level is given in Table 2.2.

The gross general structure of the system of Figure 2.10 is typical of a large number of RTDAD systems. The salient characteristics of this class of systems as seen in the contour diagram are:

1. Singular independent contour - The design effort for the typical RTDAD system embodies the notion of a "target machine", RM_0. Regardless of the nature of the design philosophy (top-down, bottom-up, most-solid-first, etc.) the resultant configuration is composed of a sequence of virtual machines which depend upon one general purpose real machine.

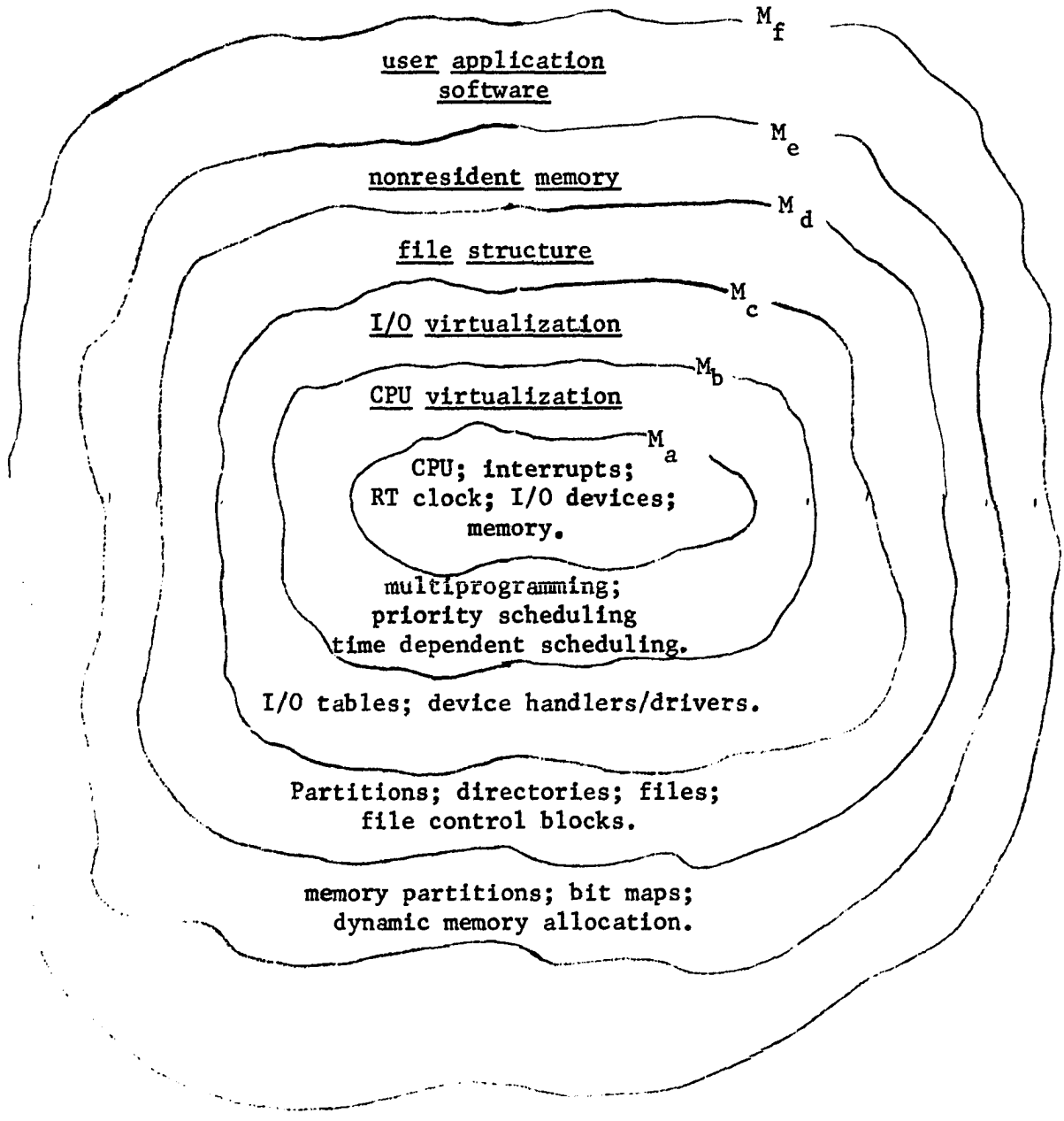


Figure 2.10. Contour diagram for conventional SCS RTDAD system.

Table 2.2. RTDAD level diagram synopsis.

Level	Identification	Description
0	Hardware	Base machine hardware organization.
1	CPU virtualization	This level supports the job scheduler, real-time clock scheduling, synchronization primitives, interrupt response routines, and associated data structures. This level provides for the definition of a multiplicity of virtual processors.
2	I/O virtualization	Level 2 specifies a device independent I/O structure which can be referenced by a standard set of I/O macros. Note that hardware I/O instructions are no longer available at any level above 2.
3	file system	Level 3 defines those operations associated with the manipulation of data on the rotating memory devices (RMD). The level defines a hierarchical file structure with directories and associated contents stored on the RMDs. All RMD references at higher levels must be made through the file system algorithms.
4	nonresident memory virtualization	At level 4 a portion of the main store is defined as the nonresident memory area into which programs which are stored in the file system may be selectively loaded and executed. Note that by definition all levels below 3 are totally implemented in resident memory areas. This level is the interface between the operating system proper and the user application software.

Table 2.2. Continued.

Level	Identification	Description
5	application area	Level 5 is the application software level. This level in turn is represented by a series of levels depicting the decomposition of the application programming effort into a series of virtual machines. As viewed from the external environment, machine M_f accepts inputs in the form of data and control packets from the input transducers and operator communication devices. The input packets are equivalent to instructions in L_f which are decoded and executed by the machine M_f.

We will refer to such a system a single centered structure (SCS) as all the levels form shells about the "center" of the single independent real machine ($M_a = RM_0$).

2. Multilevel SCS interpreter - The highest level virtual machine M_f, when viewed from the external environment appears as a single interpreter which decodes the instruction packets that are received via the external input ports. The level diagram shows a breakdown of that singular interpreter to one and only one independent contour. Thus, the overall composite structure of M_f is an SCS with a multiplicity of virtual machine levels.
3. Functional dependency - The removal of any required layer without an equivalent replacement results in a distribution of the burden of supporting the capabilities of that level across all higher levels which use instructions provided by that level. For example, suppose that the file system virtualization was not developed as level 3 in Figure 2.10. If this level is completely removed, the only available language for manipulating the rotating memory devices (RMD) for VM_j ($j > 3$) is supported by VM_2, the I/O virtualization. Thus, all users of the RMDs would have to agree on an a priori static distribution of the storage space on the mass storage devices. Each user, in turn, would be responsible for the creation and maintenance of file directories, file storage addresses, etc. via machine $M_2 = (L_2, H_2)$.
4. Uniform transformation phases - In general, the transformation set Z used to derive M_f from M_a can be partitioned by technology into

three distinct phases:

$$Z = \underbrace{Z_n \cdots Z_{k+1}}_{\text{software}} \;\Big|\; \underbrace{Z_k \cdots Z_{i+1}}_{\text{firmware}} \;\Big|\; \underbrace{Z_i \cdots Z_1}_{\text{hardware}}$$

The partition boundaries (the number of elements in each phase) can vary as a function of design philosophy. Some designs use little or no firmware (microprogramming) or are wholly unaware of its presence. That is, the firmware is used to support a basic set of macroinstructions which are typical of more conventional random logic machines[1]. On the otherhand, the Venus Operating System (29) uses firmware transformation to achieve a desired set of primitives (P, V, scheduling algorithm, etc.) as a basis for the development of its operating system functions. The SYMBOL 2R computer employs a total hardware transformation set. The system has an operational mode in which all system functions are realized in primitive modules (58).

Typically, however, the transformation set appears as shown with some combination of two or three distinct transformation phases.

MCS Structures

The example of Figure 2.10 is typical of the systems currently being developed for the RTDAD class of problems. A salient characteristic of these systems is the embodiment of the solution as a multilevel software interpreter. The general design philosophy, independent of design

---

[1]For example, the Varian Data Machines V73 minicomputer uses microprogramming to emulate the earlier Varian 620 series machine instructions.

methodology, considers the real machine as the baseline to which or from which the design envolves. This "target hardware machine" is construed as the given commodity from which the final system is created by the methodical addition of a sequence of software/firmware virtual machines.

Such SCS systems are a direct result of the long-standing dichotomy between hardware and software oriented design groups. The inertia inherent in the separation of hard and soft regimes has rejected the integrated coherent application of all available technologies directly to the problem environment.

An alternative to the SCS approach involves the repeated use of all technologies throughout the design process. This approach produces system structures which are characterized by a multiplicity of independent contours or independent real machines; such structures are referred to as multicentered structures (MCS).

An example of the distinction between an MCS and an SCS is shown in Figure 2.11. The concept of a center in an SCS was described previously as the center of the innermost machine contour. A center, by definition, is a real machine which is enclosed by the independent contour. An MCS is an extension of the SCS in which there are a number of independent contours (RM_0^A, RM_0^B, ...) with an equivalent number of centers.

More specifically, the set of real machines represents a number of independent execution entities whose processing activities are supported by independent processing elements. As before, each independent real machine may be augmented by a sequence of virtual machines VM_j^{CI} where

CI is the real machine center index, and

j is the machine level index.

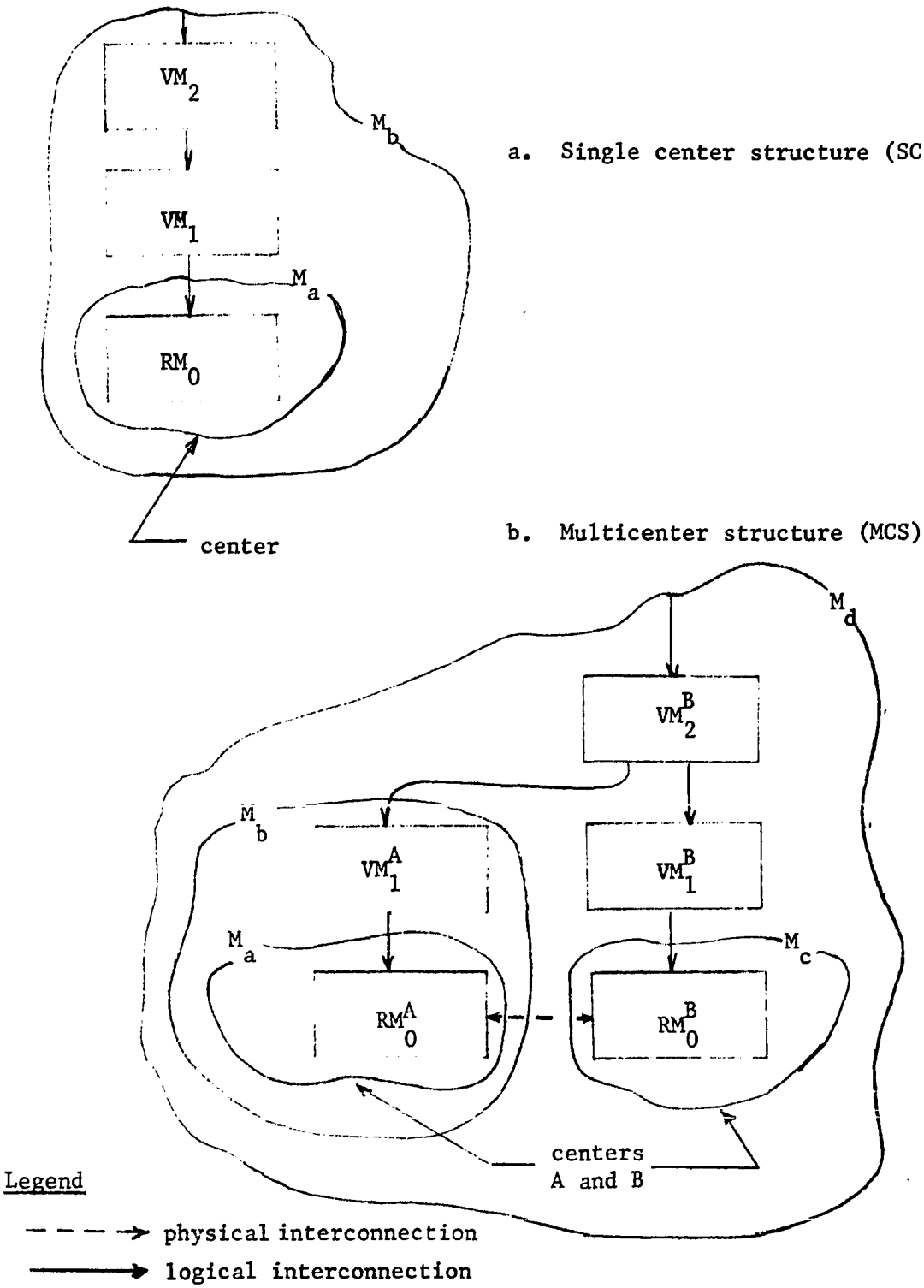


Figure 2.11. MCS and SCS representations.

The notation for the transform set Z is likewise extended to include the MCS:

1. $M_{FT}^{CI} \xleftarrow{Z^{CI}} M_{0}^{CI}$ (FT is the maximum level index for center CI).

2. $M_{i}^{CI} \xleftarrow{Z_i^{CI}} M_{i-1}^{CI} \cup \left\{ M_j^{CI'} \mid L_j^{CI'} \text{ is available to } Z_i^{CI} \right\}$

3. $M_{0}^{LI} \xleftarrow{Z_0^{CI}}$ (primitive modules)

4. $Z_i^{CI} = \left\{ Z_{i1}^{CI}, Z_{i2}^{CI}, \ldots, Z_{im}^{CI} \right\} \qquad 0 \leq i \leq FT$

5. $M_i^{CI} = (L_i^{CI}, H_i^{CI})$

Item 2 above indicates that a machine at the i^{th} level may be transformed from a machine at a lower level (i-1) and a machine based on a different center. For example, the language available at the input of machine VM_2^B in Figure 2.11 is the union of languages L_1^A and L_1^B (i.e. $L_1^A \cup L_1^B$). The union is explicitly indicated by the connection of the logical communication channels from M_1^A and M_1^B to the base of machine M_2^B.

The remainder of the notation is essentially the same as that used for an SCS with the addition of the superscript representing the associated center index CI.

The MCS can be viewed as a distributed function architecture in which the machine M_i^{CI} $(i > 0)$ is supported by the independent real machine M_0^{CI}. The significance of multiple independent machines can be seen by a closer examination of the accessibility of the component machines. In Figure 2.11b machine M_0^A is accessible only to machine M_1^A. L_0^A is, therefore, physically and logically inaccessible to any of the machines M_i^B (i=0,1,2).

This characteristic of a MCS is the extremely important concealment property defined earlier.

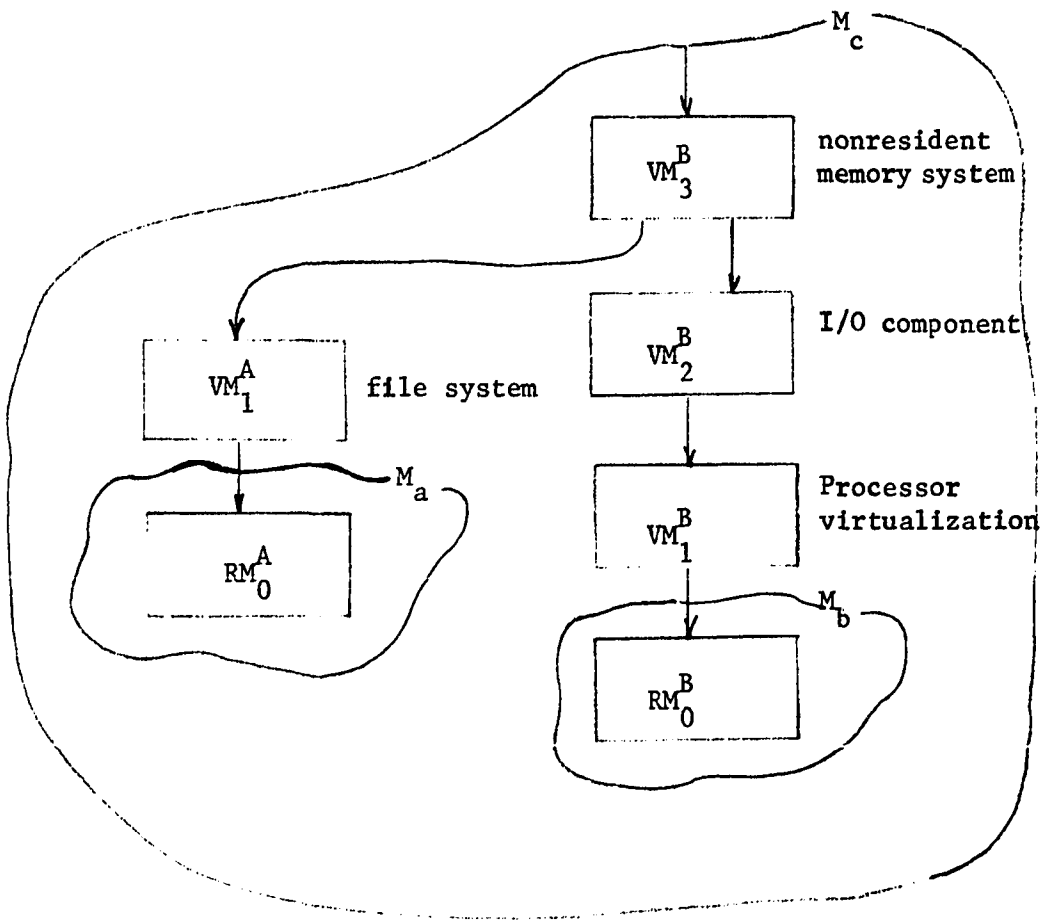
Additionally, L_1^A is the sole external interface supported by a virtualization of real machine M_0^A. This interface, as shown in Figure 2.11b, is made available to machine M_2^B. As a function of the definition of the transformation set, L_1^A may be made available to higher levels of M_i^B $(i > 2)$ or confined by the concealment property to some consecutive subset of levels.

Furthermore, since L_1^A is not available to M_1^B and conversely (L_1^B is not available to M_1^A), there are no functions or procedures in one which can be used, or are needed, to support the other. The machines M_1^A and M_1^B coexist independently of each other.

The physical interconnection of the hardware associated with M_0^A and M_0^B is shown by the dotted line. The physical realization of the hardware channel is implementation dependent (e.g., it might represent a data and control bus). Regardless of the physical implementation, however, the language supported at the hardware interface is that of M_1^A. The physical interconnection merely supports the definition of the instructions of that language.

An amplification of these comments can be made with a more concrete example. Suppose that Figure 2.10 is altered to include a second machine center which supports a file system. The new configuration is shown in Figure 2.12.

The functional definition of the levels (CPU virtualization, I/O virtualization, file system virtualization and nonresident memory



$VM_1^A = M_1^A = (L_1^A, H_1^A)$

L_1^A = (open-file, read-file, write-file, close-file)

Figure 2.12. MCS RTDAD operating system.

virtualization) are logically equivalent to the identical levels identified in Table 2.2. Thus, as viewed from the external environment (outside the defining contours), machines M_e of Figure 2.10 and M_c of Figure 2.12 are functionally identical. That is, given two system "black boxes" representing these two implementations, an input sequence applied to either would produce the same output sequence. Based on the equivalence of the output sequences, the two systems are functionally indistinguishable.

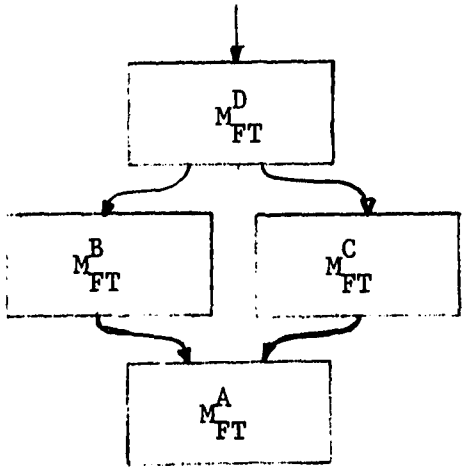
The instructions or language element of machine M_1^A are shown in Figure 2.12[1]. The file system supporting this language L_1^A is completely contained in the machine M_1^A which is, in turn, dependent only upon the real machine M_0^A. These functions are independent of any component of the CPU virtualization or I/O virtualization of the machine sequence with center index B.

As a more complex example, consider the configuration of Figure 2.13. The collapsed BLD showing the total machines M_{FT}^{CI} (CI = A,B,C,D) depicts the global hierarchical access structure. Note that M_{FT}^B and M_{FT}^C are accessible to M_{FT}^D. Similarly, M_{FT}^A is accessible to both M_{FT}^B and M_{FT}^C while it is inaccessible to M_{FT}^D.

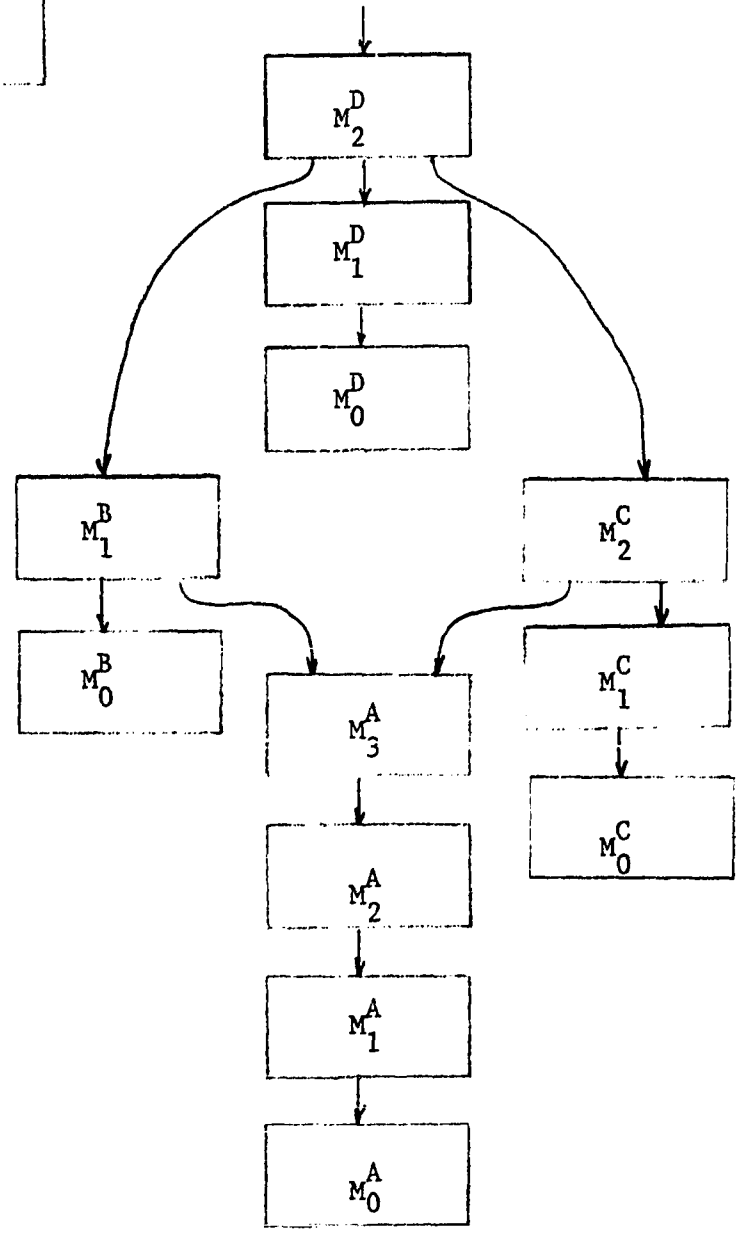
The same properties with the appropriate logical language connectives are shown in Figure 2.13b. This figure depicts a more detailed machine accessibility structure. The virtual machine levels in M_{FT}^B and M_{FT}^C at

---

[1]The language supported by the file system is assumed to be basically the same as that provided by the SCS file system modules. However, there might be more desirable instruction sets which would take advantage of the logical and physical separation of the file system components from the remainder of the system.

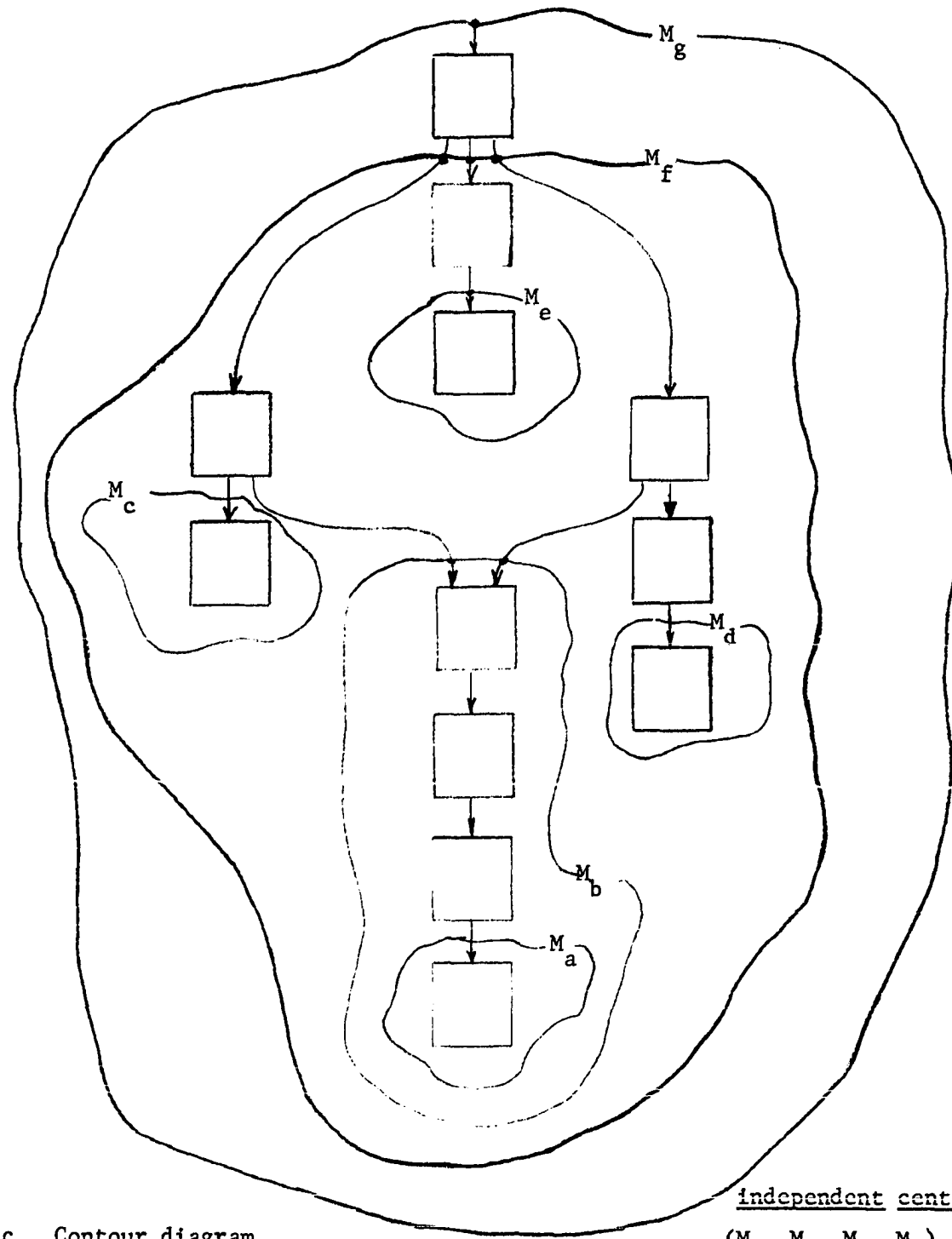


a. Collapsed BLD.



b. Detailed BLD.

Figure 2.13. MCS block level diagrams.



c. Contour diagram.

independent centers
(M_a, M_c, M_d, M_e)

Figure 2.13. Continued.

which M_{FT}^{A} is referenced depend upon the structure of these machines. More to the point, there is essentially no restriction as to the levels at which the logical connections are made. They depend solely on the needs of the levels of the accessing machines.

Finally, the contour diagram and machine associations of the contour diagram and BLD are shown in Figure 2.13c. The contour diagram emphasizes the existence of the multicentered nature of the system being represented.

A detailed example of a BLD representing an MCS will be presented in Chapter 5. However, it suffices to note here the following general characteristics of an MCS system in contrast to an SCS system:

1. An SCS system is a multilevel interpreter whose levels are derived from the sequential application of hardware, firmware, and software technologies. The resultant system has a singular independent contour supplying the fundamental execution units upon which a complex software/firmware superstructure is designed.
2. An MCS system consists of a multiplicity of multilevel interpreters. The number of distinct interpreters is equal to the number of independent contours or centers displayed in the associated contour diagram. An MCS is derived by the repeated application of hardware, firmware, and software technologies.
3. In an MCS, the implementation of each machine sequence, M_{FT}^{CI} (CI = A, B, C, ...), is physically isolated from the other machine sequences. Given that the language interface and performance requirements can be satisfied, the actual implementation of each component machine sequence is completely hidden from the

other machine sequences. The importance of this concealment property will be revisited again in Chapter 5.

CHAPTER 3

SYSTEM STATE MODEL

Introduction

Chapter 2 introduced the concept of a multicentered structure (MCS). An MCS defines a system architecture consisting of a multiplicity of independent interpreters. Each center or interpreter supports a number of levels of abstraction which are used to generate a virtual machine language compatible with the functional specification of the desired system.

As shown in Figure 1.5, an MCS decomposition is achieved by the application of two graph theoretic tools:

1. the System State Model, and
2. the Timed Petri net.

Fundamentally, the System State Model depicts "snapshots" of the results of system activity while the Timed Petri net incorporates the appropriate timing constraints and resource requirements needed to support that activity. The discussion of Timed Petri nets will be deferred until Chapter 4. The following material deals with the definition of the abstract system state concept.

The motivation for employing the System State Model is threefold. First, the System State Model places a strong emphasis on a top-down design of the problem solution. The central thrust of the approach is in the functional representation of the system requirements

as specified in the problem statement. The System State Model describes:

1. the information retained in a system, and
2. the instructions used in manipulating that information.

The model is particularly well-suited for the description of concurrency, sharing, and protection.

Secondly, the System State Model is implementation independent. That is, the descriptive powers of the model do not depend upon architectural configurations or structures. As will be shown in Chapter 5, the resolution of architectual issues is attained by a combination of the system state definition and the Timed Petri nets. It should be noted that the System State Model does not depict resource requirements nor does it represent any real-time constraints as described in Chapter 1. However, the model does provide a complete description of resource allocation algorithms that result from the combined application of the model of chapters 3 and 4.

The System State Model is not an exercise in mathematical or graphical elegance. The existence of a translation from the System State Model to an executable machine environment is a prerequisite of the design philosophy. That is, the model is sufficiently powerful to describe all the functions and structures required to solve the RTDAD problem. In particular, the model does not exclude any feature which might be deemed either necessary or convenient in the problem solution.

Finally, the System State Model provides a semantic definition of the contour languages described in Chapter 2. The model precisely and

unambiguously defines the functional characteristics of a system which allows for the decomposition of the solution into the appropriate machine levels. The semantics of these levels are established by the function and instructions of the model.

In passing, it may be noted that there are implications of correctness issues which, due to the scope of this research, are not covered. Given a semantic definition of the language of the contours, the correctness of a functional description can be verified. Once the abstract model is shown to be correct, the associated implementation will be correct if it can be proven to be an accurate representation of the abstract model. The concepts of a priori correctness and the impact of designing "error free" software have been discussed by Dijkstra (16), McGowan and Kelly (34), Parnas (46), and others. Although correctness is not emphasized in this work, we feel that it is an important and natural result of the tools presented.

The technique used for the description of the System State Model is based on the Operational Method of defining the semantics of programming languages (61, 39). These concepts are extended to allow for the semantic definition of the contour languages of an MCS architecutre. In particular, the model defines:

1. the state of an abstract interpreter whose components are a set of abstract programs and a representation of the system environment, and
2. the abstract instruction set which defines transformations on the state of the interpreter.

The abstract state represents the totality of programs, data, and control information present in the computer system. The abstract programs consist of sequences of abstract instructions of an Abstract Base Language (ABL). The abstract instructions are precisely defined in terms of their effect on the system state.

The formal semantics of the contour languages take the form of two sets of rules (13):

1. translator - Elements of the contour languages are translated into equivalent sequences of abstract instructions.
2. interpreter - The interpreter expresses the meaning of programs in the abstract language by giving directions for carrying out the computations of any well-formed abstract program as a countable set of primitive steps.

Thus, the translator defines the semantics of the contour languages while the interpreter specifies the state transitions that are realized by the execution of those language elements.

The next two sections describe the abstract instructions and the structure of the state of the abstract interpreter.

Abstract Base Language

The Abstract Base Language (ABL) has two constituent elements:

1. abstract objects which represent information, and
2. abstract instructions which define a set of operations to be performed.

The remainder of this section describes these two items. The bulk of this material is based on work from Wegner (61), Neuhold (39), Dennis (13), and Hawryskiewitz (24).

Abstract objects

An abstract object can be either an elementary object or a compound object. Elementary objects are members of the class E, where

$$E = Z \cup R \cup B \cup W$$

and

Z = set of integers

R = set of real numbers

B = boolean (T = true, F = false)

W = set of strings on some alphabet.

Since the abstract instructions are elements of W, they are also members of the class E.

A compound object is the combination of elementary objects and/or compound objects into a structure. The definition of compound objects is recursive providing the possibility of compound objects containing other compound objects.

Abstract objects are represented by directed graphs. These graphs contain edges called branches and vertices which are called nodes. A node can be a root node of a compound object, a leaf node representing an elementary object (a leaf node is a node which has no emanating branches), or a general junction node (neither a leaf node nor a root node).

A branch is identified by an attached label called a selector. A selector is a member of the class S, where

$$S = Z \cup W$$

and

Z = set of integers

W = set of strings or some alphabet.

An example of an abstract object is given in Figure 3.1. This abstract object has the following components:

1. selectors = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'}
2. elementary objects = {1, 2.5, T, SCOTT, STRING}
3. compound object nodes = $\{n_1, n_2, n_3, n_4, n_5\}$

(the node identifiers n_i, $1 \leq i \leq 5$, have no significance other than their use in specifying nodes for descriptive purposes.)

Object references

Objects can be referenced by the use of variables of the type pointer. A pointer value uniquely identifies a node of an abstract object. A pointer value can be assigned to a pointer variable. If the identifier of a pointer variable is P, then P is said to refer to the node represented by its value.

To select a particular branch emanating from a compound node, a selector expression can be used to qualify the pointer identifier. A selector expression is either a selector value or the identifier of a selector variable.

A pointer expression can be one of three forms:

1. a pointer value,
2. the identifier of a pointer variable, or

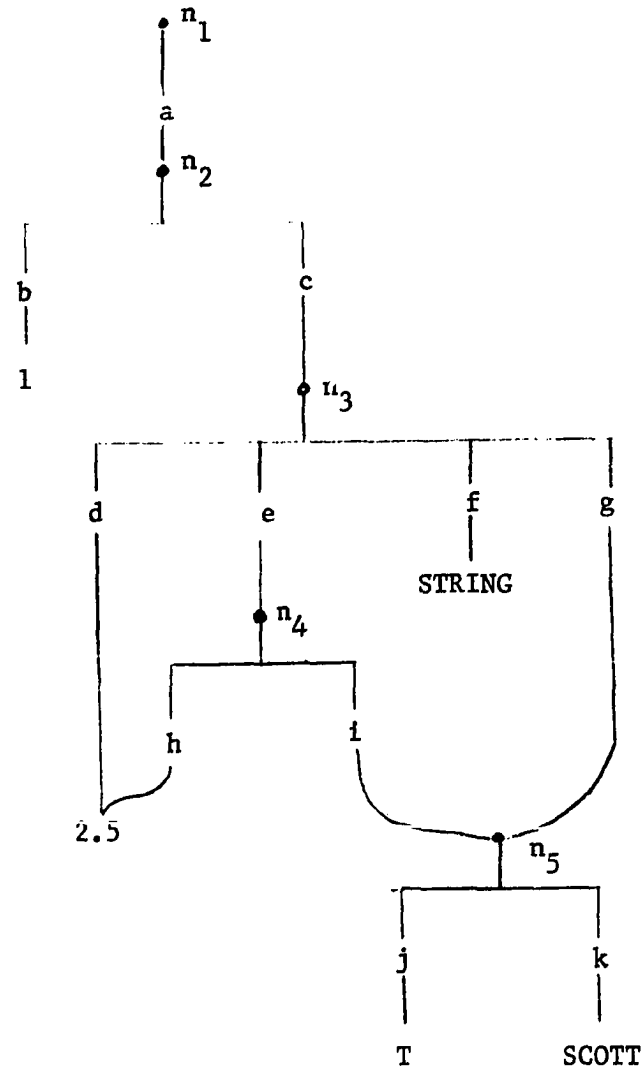


Figure 3.1. An abstract object.

3. a pointer qualified by a selector expression denoted by the pointer followed by a period (.) and the selector expression.

Qualification permits the access of any component of a compound object. Since the definition of compound objects is recursive, the qualification can also be used in a recursive manner to thread through a chain of branches from any given compound node.

For syntactic convenience the following conventions will be employed:

1. pointer variables commence with an upper case letter,
2. selector variables commence with a lower case letter, and
3. a selector value appears in a selector expression enclosed in single quotes (').

Referring again to Figure 3.1, the following examples demonstrate the use of pointers and selectors in referencing the components of an abstract object. Let P, Q, R, and S be pointer variables and sel_1, sel_2, and sel_3 be selector variables.

Example 1. If P has the value n_1, then

Q ⟵ P.'a' makes Q refer to node n_2.

Example 2. Similarly,

sel_1 ⟵ 'a'

Q ⟵ P.sel_1 makes Q refer to node n_2.

Example 3. A reference to the elementary object STRING is represented by the pointer variable R where

R ⟵ P.'a'.'c'.'f'

or

sel_1 ⟵ 'a'

sel_2 ⟵ 'c'

sel_3 ⟵ 'f'

R ⟵ P. sel_1. sel_2. sel_3.

Compound objects can be shared by other compound objects. In Figure 3.1, the elementary object 2.5 and the compound object rooted at node n_5 are shared by the compound structures rooted at n_3 and n_4.

Two additional properties of nodes need to be introduced. A node n_i is reachable from the node n_j if there exists a chain of branches from n_j to n_i. Node n_i is a direct successor of node n_j if the chain is of unity length.

A root node of an abstract object has only emanating branches. A given object may have more than one root node. A closed object is an abstract object which has exactly one root node.

Again with respect to Figure 3.1, n_1 is the only root node for this object. Therefore, it is a closed object. The reachability set of nodes for node n_3 is the set

$$N = \{2.5,\ n_4,\ n_5,\ \text{STRING},\ \text{SCOTT},\ T\}.$$

Additionally, the direct successor of n_4 is the set

$$M = \{2.5,\ n_5\}.$$

The power of the directed graph approach can be seen in the range of abstract objects expressible in this format. In particular, all objects which are useful in the RTDAD environment can be represented. The range

of such data objects include, but are not restricted to:

1. single variables, constants, or pointer variables,
2. tables, arrays,
3. sparse arrays,
4. linked lists (singly, doubly, circular),
5. queues, deques,
6. stacks, and
7. trees.

As an example of a complex data structure consider Figure 3.2. The pointer variable LIST points to the head of the list. LIST. 'front' branches to the first element in the list or is null if the list is empty. Note that the list elements in this particular example are of variable length. Each element has a length specified by the branch labelled 'size'.

Note also that the list structure is independent of any particular implementation. The directed graph description of abstract objects expresses the structure of objects without regard to implementation issues such as the nature of the link fields or location of data and pointers in memory. By suppressing these issues, the functional motivation for employing the list structure becomes more evident in the design process.

Additionally, the architectural "binding time" of the list structure is deferred until the latest possible moment. At that time, the space-time tradeoffs can be assessed and a particular technology for implementation can be selected.

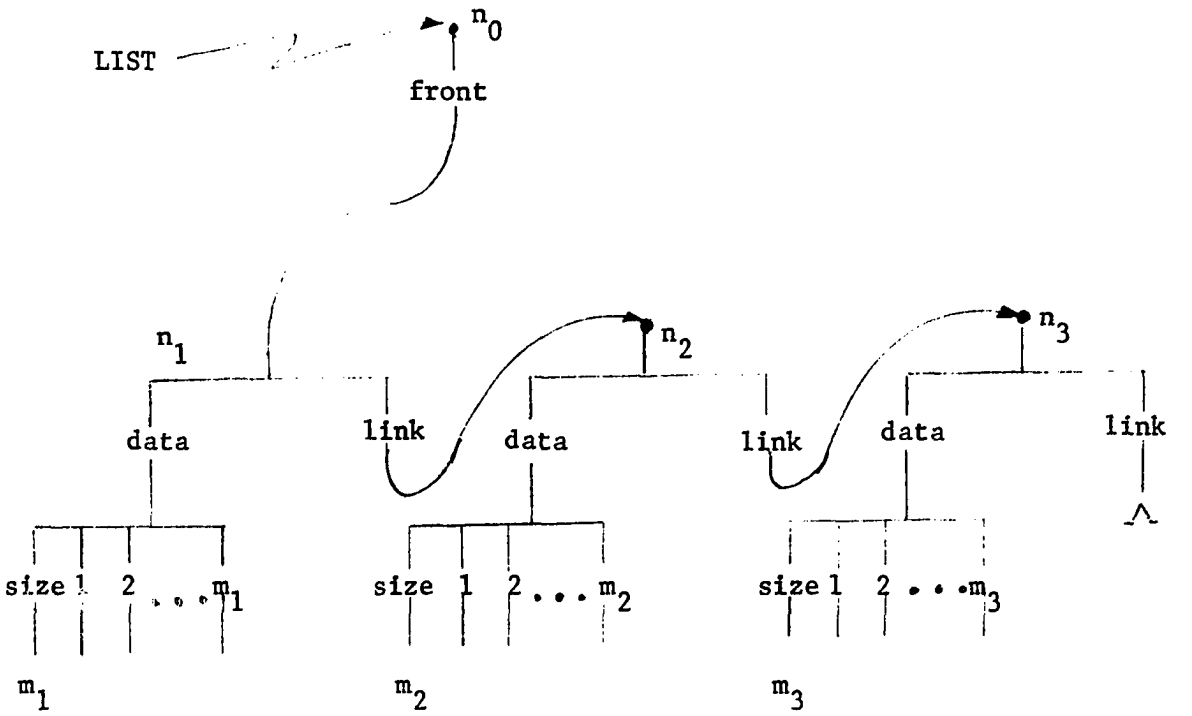


Figure 3.2. Abstract object representing a singly linked list.

Abstract instructions

The abstract instructions provide the primitive operations which are used to manipulate the information stored in abstract objects and to provide the desired sequence of control. The abstract instructions are partitioned into five categories:

1. object reference,
2. synchronization,
3. control constructs,
4. procedure reference, and
5. process identification.

A list of instructions by category is specified in Table 3.1. We do not present an exhaustive technical description of the syntax and semantics of the language elements given in Table 3.1. Instead, the remaining material in the section will serve to clarify the information presented by a sequence of short examples. The general composition of the abstract programs is in the spirit of a number of current programming languages [Algol 68, PL1, BCPL].

Suppose that the abstract object referenced by the pointer variable A has the structure shown in Figure 3.3. The effect of the selt and empty instructions is shown. These two functions are predicates used to test for the presence or absence of branches at a given node.

The val and assign instructions are used to reference elementary values. The assign function associates an elementary value from E with a leaf node, while the val function returns the value associated with a leaf node. For notational simplicity equivalent operations are denoted

Table 3.1. Abstract instruction set.

Let P and Q be any pointer expressions and x any selector expression.

Category I - Object Reference

1.	selt(P,x)	= T, if there is a branch labelled with the value x, and emanating from the node to which P refers. = F, otherwise.
2.	empty(P)	= T, if there are no branches emanating from the node to which P refers. = F, otherwise.
3.	val(P) or *P	= the value of the elementary object to which P refers.
4.	assign(P) ⟵ w or *P ⟵ w	w is assigned to the elementary object to which P refers.
5.	delete(P,x)	The branch labelled with the value of x and emanating from the node to which P refers is deleted.
6.	create(P)	A new node is created and assigned to P. The lock variable is set to "F". The value of the new node is the null value.
7.	append(P,x)	A branch labelled with the value of x is attached to the node to which P refers. A new node is created as part of this instruction and P•x refers to the node. The value of the new node is the null value.
8.	link(P,Q,x)	A branch labelled with the value of x is created. It emanates from the node to which P refers and terminates on the node to which Q refers.

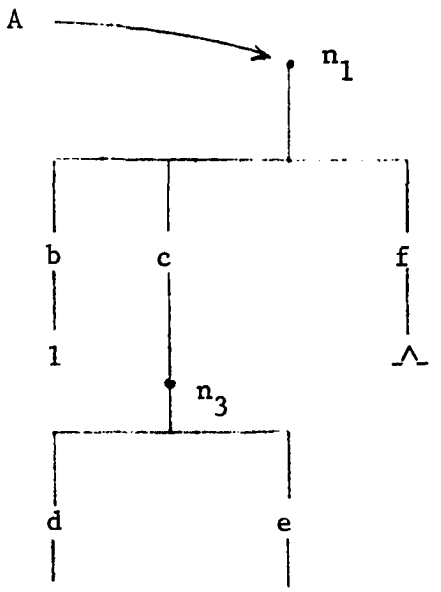
Table 3.1. Continued.

	Category II - Synchronization	
1.	lock(P)	If the lock variable associated with P is "T", the process waits until it becomes "F". It is then set to "T" and the process continues. If it is "F", the process sets it to "T" and continues.
2.	unlock(P)	The lock variable associated with the node to which P refers is set to "F".
	Category III - Control Constructs	
1.	if B then S_1 else S_2 fi	
2.	while B do S od	
3.	repeat S until B	
	where B is a boolean expression and S, S_1, and S_2 are simple statements or sequences of statements.	
	Category IV - Procedure Reference	
1.	f(a,b,c) or f( )	The function f is invoked by the calling procedure that contains this instruction. The procedure f can be either an internal or external procedure.[a]
2.	return	The return instruction initiates the transfer of control from the called procedure to the point immediately following the function call in the calling procedure.

[a]An internal procedure is a procedure that is an element of the interpreter in which it is invoked; otherwise, the procedure is termed an external procedure (see the description of the abstract interpreter state).

Table 3.1. Continued.

	Category V - Process Identification	
1.	v = principal	The unique identification of the process which invoked this instruction is assigned to v.
2.	state(Q)	A pointer to the state word of a process is assigned to Q. Following the instruction state(Q) the pointer variable Q will be assigned the pointer value of the state word associated with the process which executed the instruction.
3.	abort	If the instruction abort is executed by a process, its state word becomes unavailable.
4.	place(Q)	In addition to the state word that identifies the instruction following place(Q) a new state word is made available. The pointer variable Q refers to that state word.



15.2 STRING

	Operation	Result
1.	selt(A.'c','d')	T
2.	selt(A.'a','d')	F
3.	selt(A,'z')	F
4.	empty(A.'f')	T
5.	empty(A)	F

Figure 3.3. Abstract predicates selt and empty.

by the use of an asterisk (*) with a pointer expression. The designation of either the val or assign instructions depends upon the context of the use of the * P description. An example of the usage of val, assign, and the asterisk notation is given in Figure 3.4.

The create, delete, append, and link instructions are used to alter the structural characteristics of an abstract object. The transformation defined by these instructions is depicted in Figure 3.5. Each example represents the state of the abstract object immediately following the execution of the associated instruction.

The instructions create, append, and link are self-evident. However, the result of the delete instruction needs some clarification. The resultant object in Figure 3.5f indicates that the branch labelled 'b' has been removed but node n_2 and branch 'c' remain intact. If there are no pointer expressions which refer to n_2, that node is unrecoverable. Additionally, any node in the reachability set for n_2 which is not in this intersection of a reachability set of node n_j, $j \neq 2$, is also unrecoverable. It is assumed that such dangling nodes and associated branches occur only as errors in the logic of an algorithm. That is, algorithms which employ the delete operator are responsible for the detection and deletion of nodes and branches which might become dangling elements.

For notational convenience an alternate method of appending branches can be invoked by means of the assign operation. An assign (P) instruction creates new branches (and associated nodes) so that the pointer expression P will be valid. An example of this technique is shown in Figure 3.6.

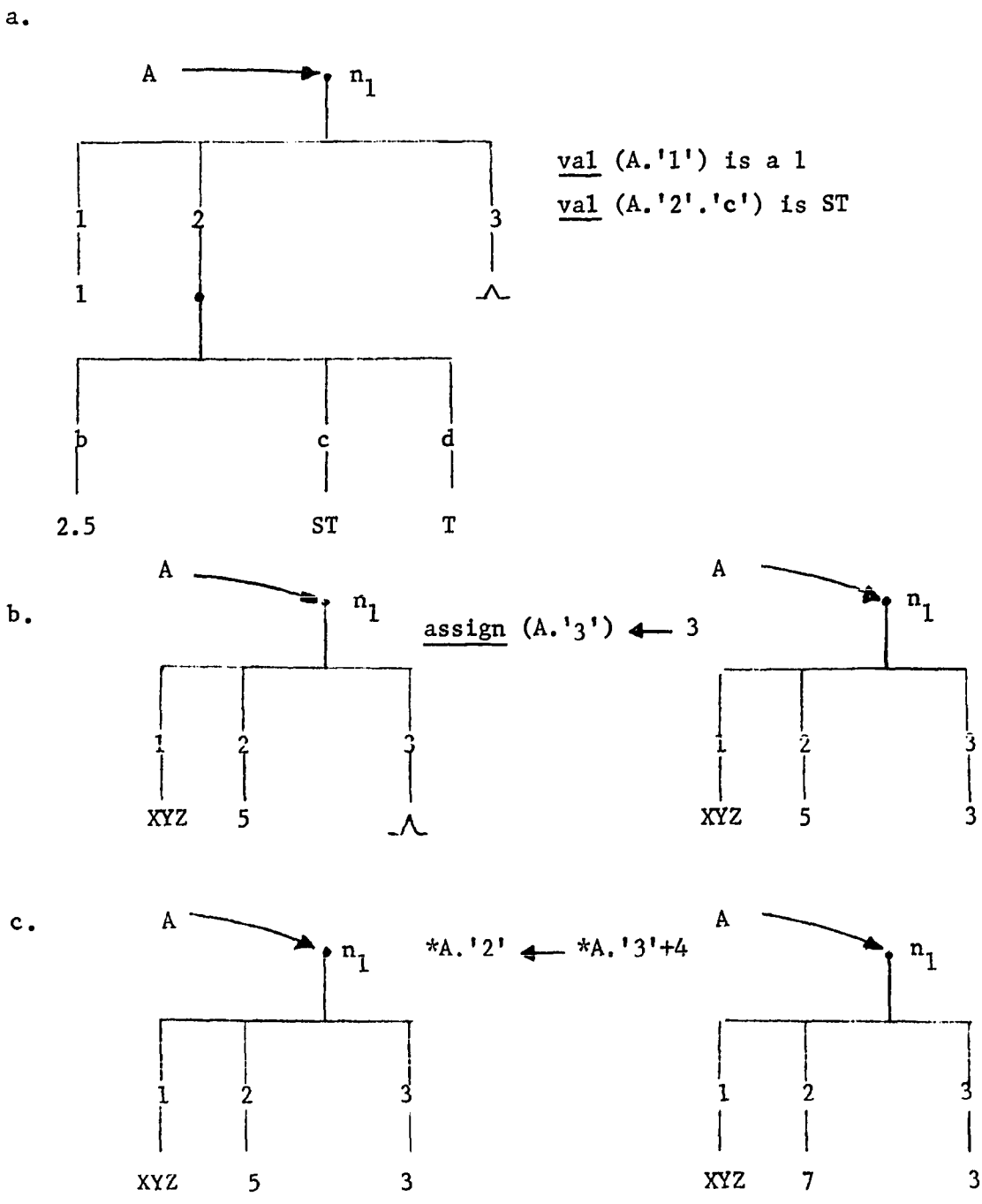
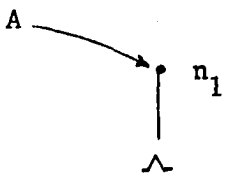
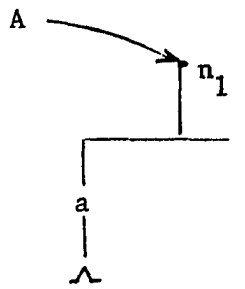


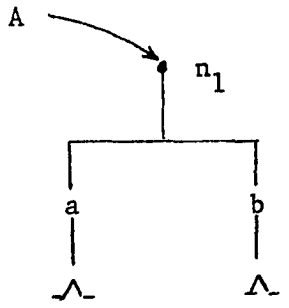
Figure 3.4. Val and assign instructions.



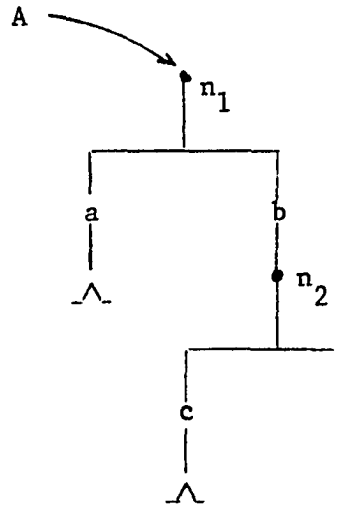
a. Create(A)



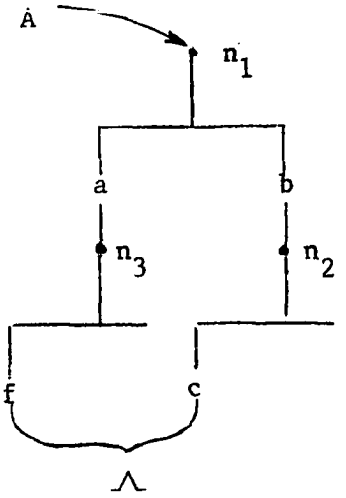
b. append (A, 'a')



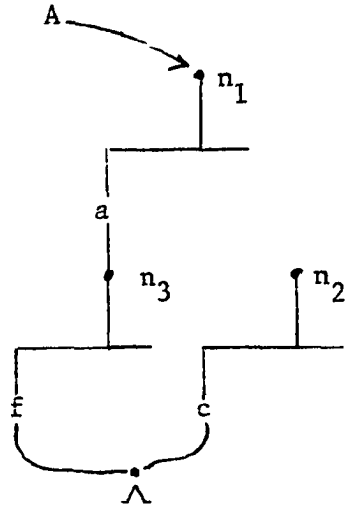
c. append (A, 'b')



d. append (A.'b', 'c')



e. link (A.'a', A.'b'.'c', 'f')



f. delete (A, 'b')

Figure 3.5. Create, append, link and delete instructions.

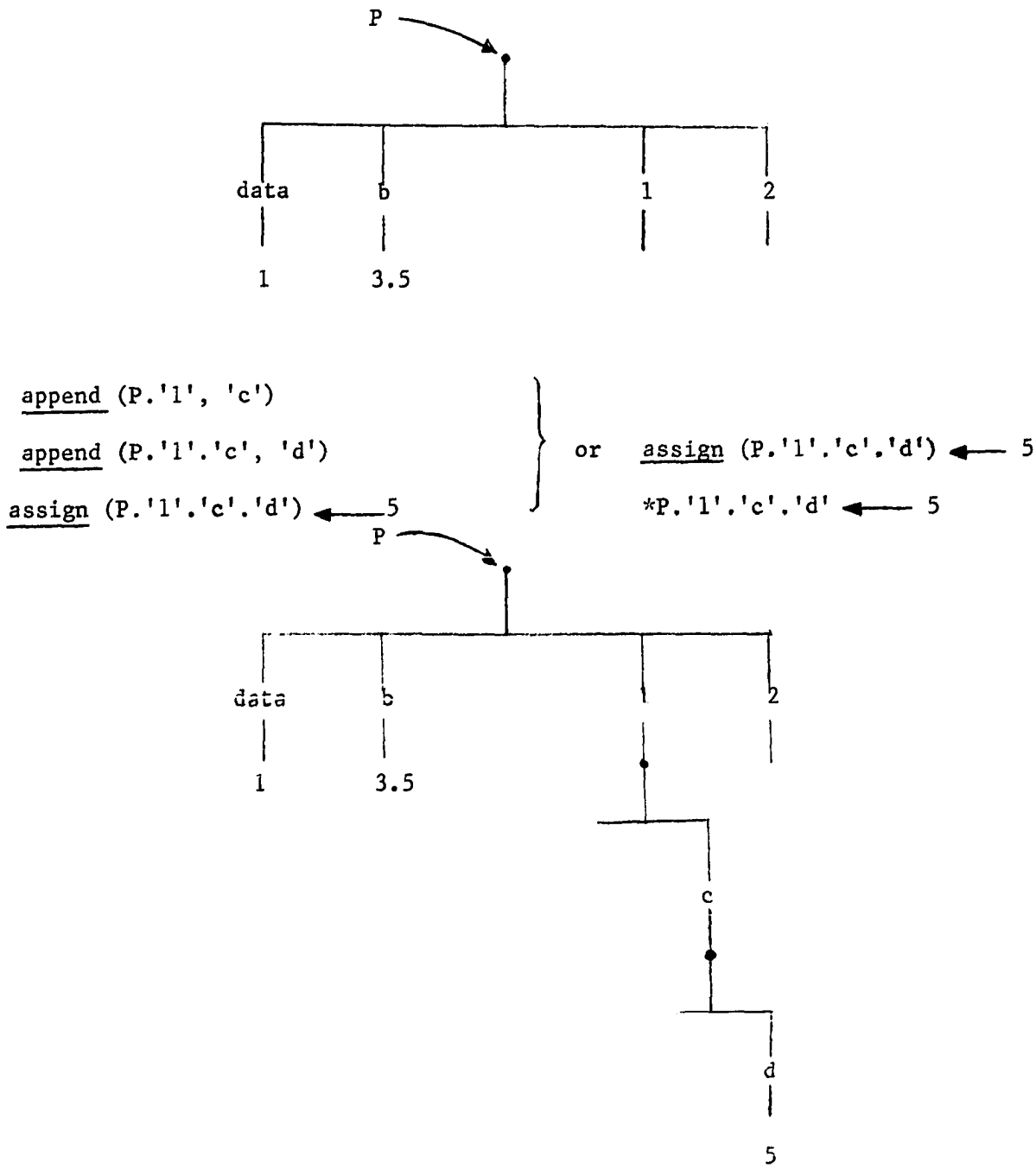


Figure 3.6. Extended use of assign in place of append.

Note that new branches are added until the pointer expression P.'1'.'c'.'d' defines a valid structure.

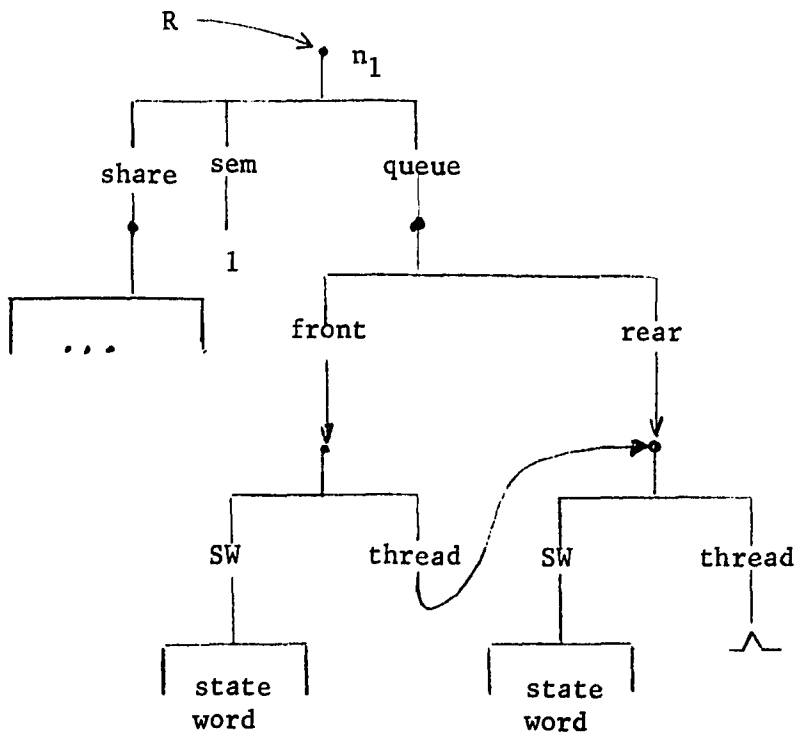
The lock and unlock instructions are synchronization primitives which apply only to nodes which are root nodes of closed objects. A root node is initially defined to be in the "F" state when it is created by either an append or create primitive. The lock and unlock operations then define the alteration of the node from the "F" state to the "T" state and back to the "F" state. The lock operator employs a busy-form-of-waiting if the referenced node is in the "T" state. Lock and unlock are the primitives required for the implementation of all higher synchronization functions.

As an example of the use of the synchronization primitives and the control constructs, the semaphore functions signal and wait as defined by Dijkstra (17) are presented in Figure 3.7. The particular abstract object depicted in Figure 3.7 uses the concept of a semaphore to provide access rules to the shared resources defined by the branch labelled 'share'.

If the value of R.'sem' is initially one, the semaphore defines mutual exclusion (binary semaphore). In contrast, if the value of R.'sem' is initialized to an integer value j ($j \geq 2$) then j processes may share the resources concurrently.

A description of the primitives in categories IV and V will be presented in conjunction with the description of the state of the abstract interpreter.

a. semaphore - closed object



b. signal

```
Procedure     signal (R)
    begin
      state (Q)
      lock  (R)

      if empty (R.'queue') then *R.'sem' ← *R.'sem' + 1

         else     remove(N,R.'queue')     $N is the element removed
                  P ← N.'SW'              from the nonempty queue.
                  place (P)
         fi
      unlock(R)
    end
```

Figure 3.7. Representation of semaphores in the System State Model.

c. wait

```
procedure    wait (R)
    begin
      state(Q); lock(R)
      if     *R.'sem' > 0 then * R.'sem' ←—— *R.'sem' - 1
                 unlock(R)
          else     create(N)
                   *N.'SW' ←—— Q
                   *N.'thread' ←—— ⌒
                   insert (N, R.'queue')
                   abort
    end
```

Notes: 1. The functions insert (N.R.'queue') and remove (N, R.'queue') are unspecified routines which deposit and withdraw entries from the semaphore waiting queue.

2. The branches labelled SW point to the state word of the associated processes. State words are defined in the next section.

Figure 3.7. Continued.

Abstract Interpreter State

The general model for the state of an abstract interpreter is the directed graph shown in Figure 3.8. The execution of an abstract instruction results in the transformation of the given state to a new state. Furthermore, the execution of a sequence of instructions of an abstract procedure produces a sequence of abstract states. Each state can be considered as a "snapshot" of the system activity incurred up to that point in time. Thus, the interpreter defines a state machine that steps through a sequence of states as a function of a sequence of inputs represented by a set of abstract instructions.

The state of the interpreter has four component objects:

1. the universe ('univ'),
2. the local environment ('local'),
3. the control objects ('cntrl'), and
4. the external interface ('ext_int').

The following four subsections describe the composition of these four component objects.

Universe

The universe is an object which represents all the information present when the machine is idle, i.e., no computation is in progress. The universe has two constituent objects - the global data structure ('global') and the procedure structure ('proc').

The global data structure contains abstract objects that are classified, according to their accessibility, into two categories:

1. file objects ('file') - File objects are shared objects which are accessible to all procedures. As will be seen shortly, file objects are not explicitly declared by the procedure objects. Accessibility to these items is specified by an access matrix (22) associated with the procedures which maintain the global objects[1].
2. external objects ('ext') - External objects are accessible to those procedures in which they are explicitly declared. External objects exhibit restrictive sharing as defined by the structure of the procedure objects.

The essential distinction between 'file' and'ext' objects is the presence or absence of the object in a procedure's virtual address space. An external object is explicitly declared to be part of the virtual address space while file objects are not. Thus the access rights to an external object is a capability either explicitly stated or not possessed by a given procedure.

The procedure structure ('proc') is a compound object whose components are the set of all procedure objects that are executable by the interpreter. A procedure object has the general format shown in Figure 3.9. The constituent elements of a procedure object are the access capabilities ('access') and the procedure text ('text').

---

[1]The access matrix may be empty. That is, all the file objects are equally accessible to all procedures.

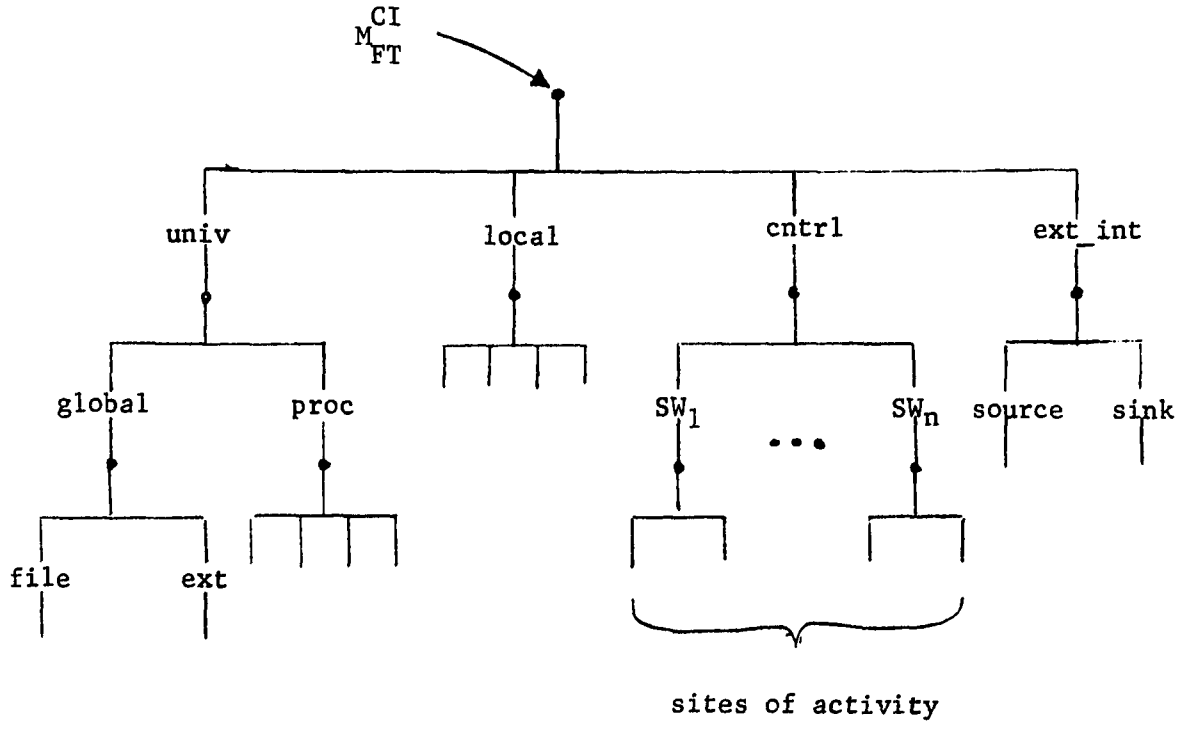


Figure 3.8. State of the abstract interpreter for machine M_{FT}^{CI}.

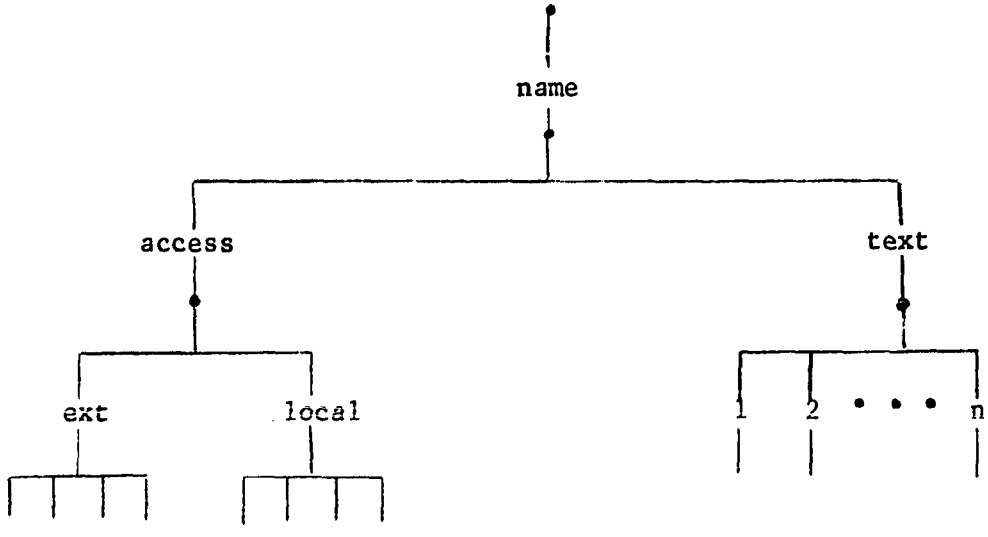


Figure 3.9. Structure of a procedure object.

The branch labelled'text' specifies the abstract instructions which define the procedure. When the procedure is invoked, the instructions are sequentially executed in the numerical order of their selectors.

The 'access' component defines the capabilities possessed by procedure 'name'. This component specifies all external and local variables that belong to the procedure.

It is important to note that procedure objects are a special case of the more general class of abstract objects. As such, a procedure object may be an element of the global data structure (file component) specified earlier. In that case, global procedure objects may be manipulated as data objects by other procedure objects. In conventional computer systems such procedure objects may include nonresident tasks (pages, overlays, segments, etc.) which are loaded from the file system and executed upon demand.

Finally, the hierarchial structure of the Block Level Diagram (Chapter 2) is explicitly defined by 'proc' component. A more appealing notion of the hierarchical nature of the procedure structure is shown in Figure 3.10. Note the comparison of this figure with Figure 2.8. Figure 2.8 includes the specification of all languages and machines including the independent real machine M_0^{CI}. The procedure structure 'proc' specifies all levels down to the point that all the language elements of the lowest level are assumed to be given. The definition of that level is somewhat arbitrary and is based on the general nature of the primitive modules (PM) used to define machine M_0^{CI}.

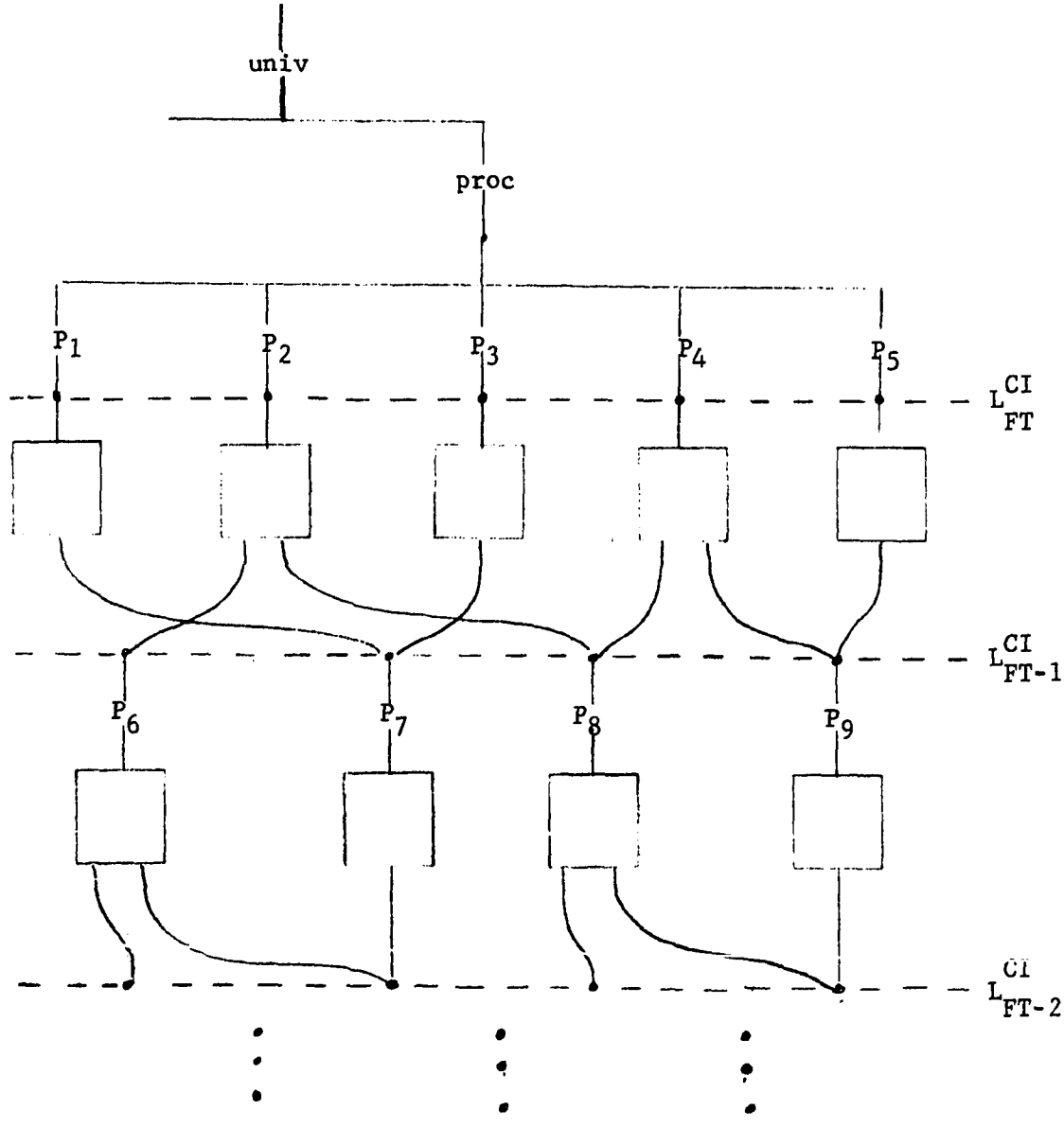


Figure 3.10. Hierarchical procedure structure.

Local objects

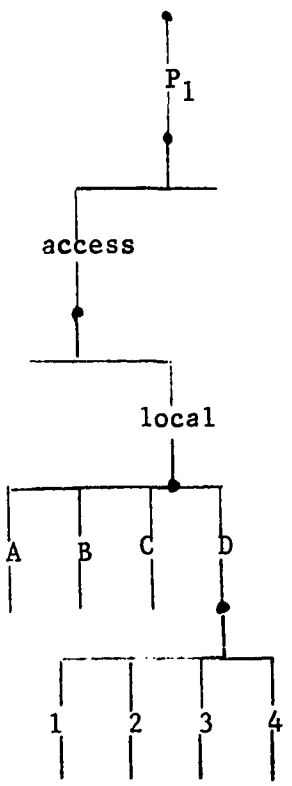
The 'local' component of the abstract interpreter state contains a local object for each current activation of an internal procedure. A local object has a component object for each variable defined by the local portion (branch labelled 'local') of the associated procedure. Additionally, the elementary values which are the variable identifiers in the procedure objects are used as the selectors for the corresponding local components. An example of the invocation of a procedure and the resultant local object is depicted in Figure 3.11.

The formal arguments of a procedure are considered as local to that procedure. When a procedure is called, the local branches (and selectors) are created and the appropriate actual parameters are linked to the formal parameters (refer to Figure 3.12). A more detailed description of the invocation of procedures is given by Dennis (13).

In addition to the local variables and formal parameters, the local structure of an invoked procedure contains a 'return' component. The branch labelled 'return' is used when the invoked procedure executes the return instruction. When this instruction is executed, the elementary value specified in the 'return' component indicates the next instruction to be executed in the calling procedure.

The execution of the return instruction results in a deallocation of the local object of the called procedure and any links established for the formal parameters. Thus, Figure 3.12b represents the state of the local component for P1 both before the invocation of P2 and after the return. However, some subset of the actual arguments may possibly have been altered by the execution of P_2.

a. Invoked procedure object (no actual parameters)



b. Local structure after invocation of P_1.

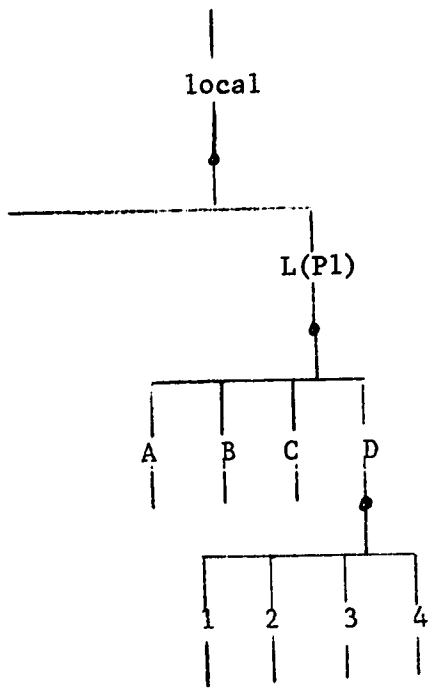
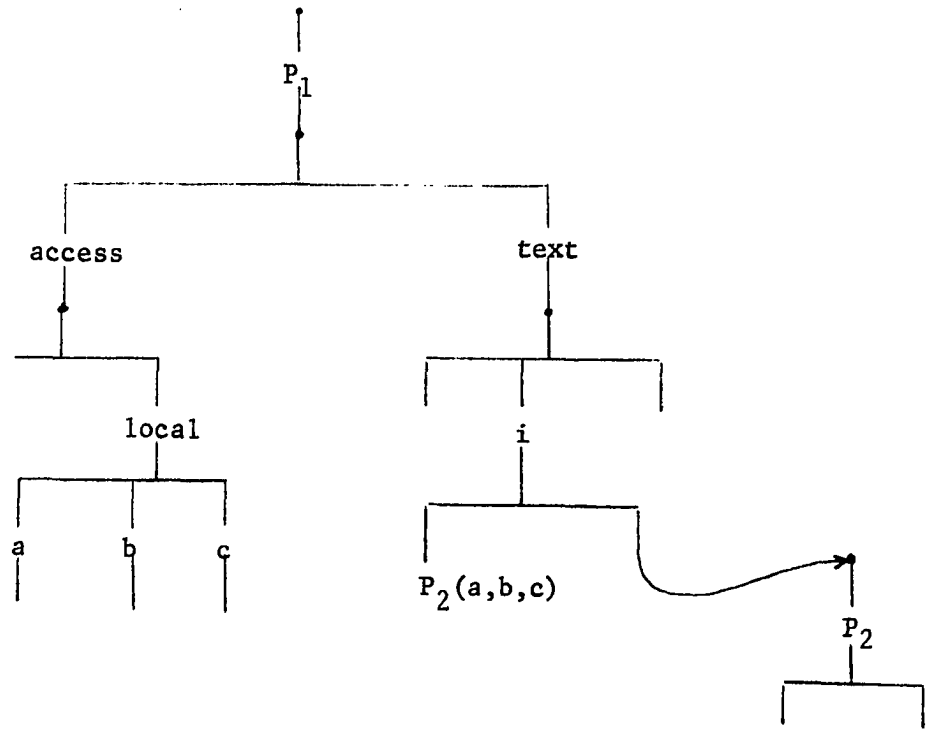
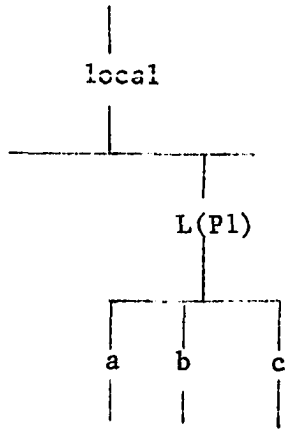


Figure 3.11. Local environment creation by procedure invocation.

a. Procedure P_1.



b. Before invocation



c. After invocation

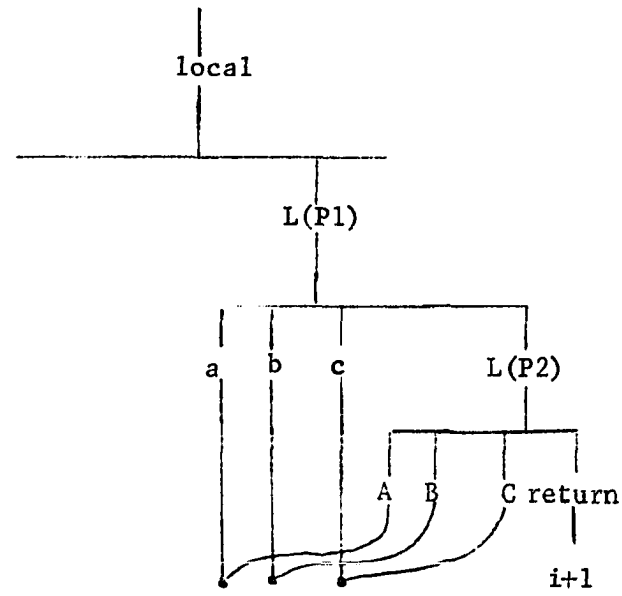


Figure 3.12. Procedure invocation with parameters.

Control objects

The control component of the abstract interpreter state contains an unordered set of sites of activity which specify the operational state of the interpreter. A site of activity is defined by a state word which contains four components (refer to Figure 3.13):

1. a pointer value that identifies a local environment ('loc'),
2. a pointer value that identifies a procedure object ('proc'),
3. the selector of the next instruction to be executed ('instr'), and
4. the unique identification of the process defined by the state word ('prin').[1]

A state transition of the interpreter results from the execution of an instruction for some procedure activation at a site of activity selected from the control of the current state. Following the execution of an instruction by some processing unit, the associated state word is replaced by a new state word that defines the next instruction in the procedure to be executed. Replacement with two sites of activity designating two successor instructions would occur in interpretation of an instruction that initiates additional concurrent action (place (Q)); deletion of the site of activity without replacement would occur in execution of an instruction that ends activity (abort).

---

[1]Each state word defines a unique process whose name is given by the branch labelled 'prin'. The abstract instruction principal retrieves the elementary value associated with this branch.

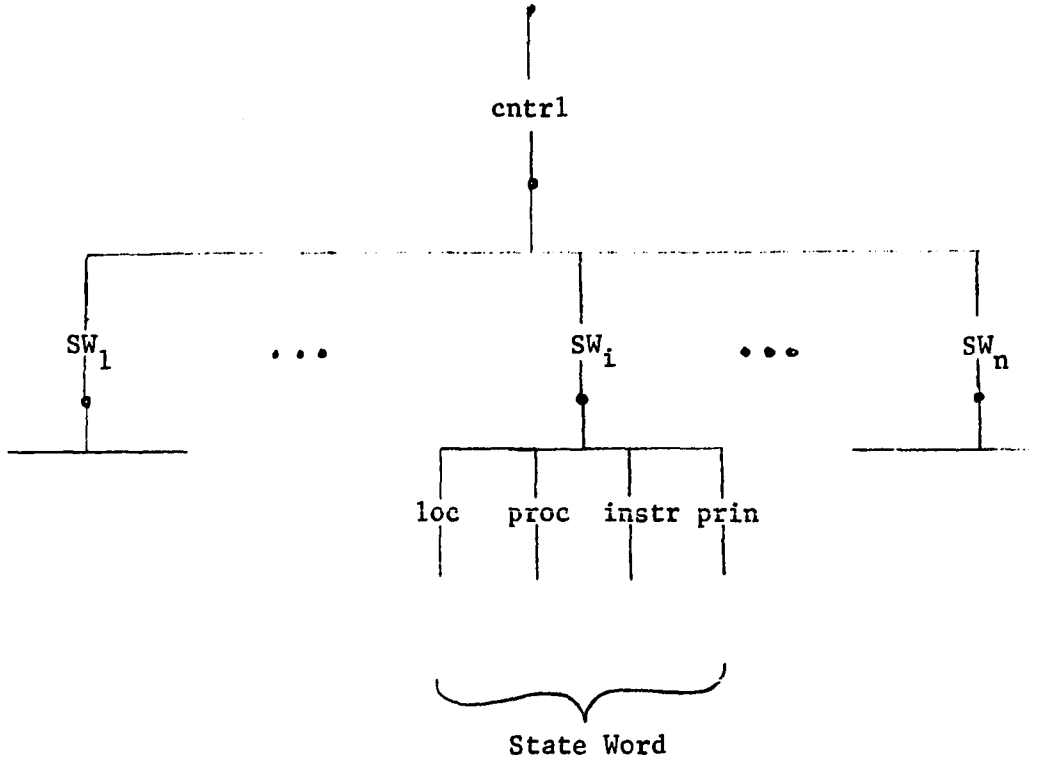


Figure 3.13. Control component.

An instruction of a procedure is said to be partially enabled if:

1. all the data required for its operation is present, and
2. an existing state word points to that instruction.

Once an instruction is partially enabled, it will start initiation whenever a processing unit can be assigned to it. The instruction will be active until the operation defined by the instruction is terminated.

External interface

The external interface of the abstract interpreter state is denoted by the branch labelled 'ext_int'. This component defines the interface to the external environment. In particular, the external interface is involved in the movement of abstract objects:

1. to the interpreter from an input transducers, IT_i,
2. from the interpreter to an output device in the user environment, $\emptyset T_i$, and
3. to the interpreter from external interpreters.

The 'source' and 'sink' branches specify the relative direction of the movement of the objects with respect to the external environment. Sink objects originate in the interpreter and are consumed by the external environment. Conversely, source objects originate on the external environment and are attached to the appropriate source nodes.

Source and sink objects are data and control packets. The components of packets typically include:

1. source identification (packet originator),
2. destination identification,

3. operation requested, and

4. data description.

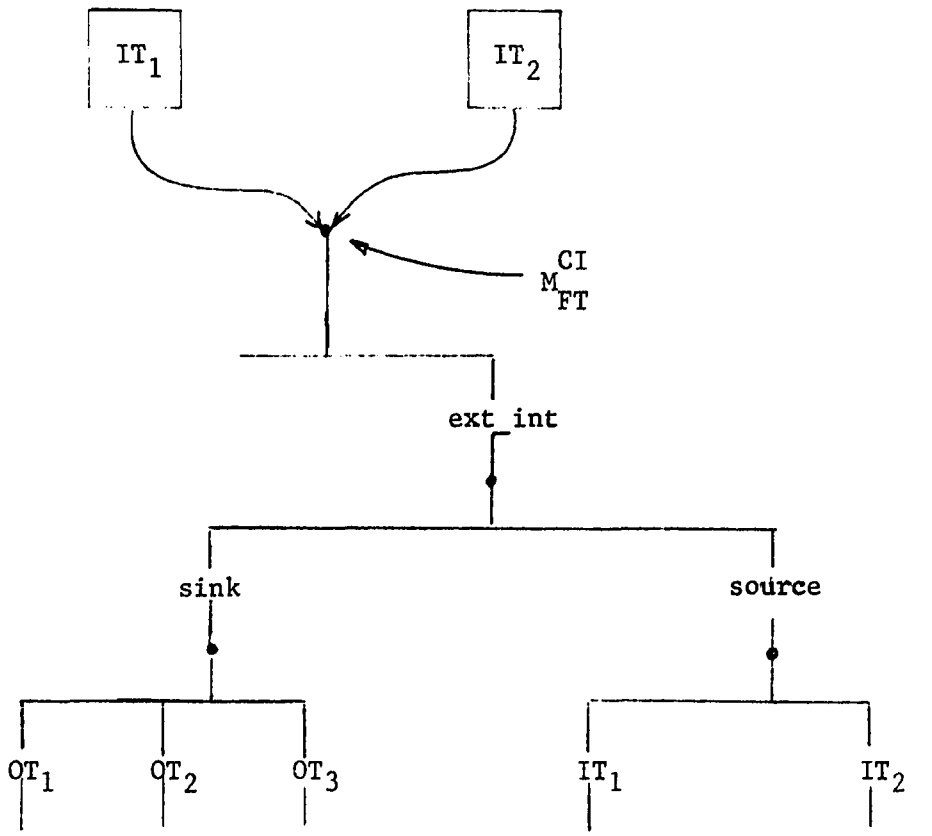
Figure 3.14 gives an example of a possible 'ext_int' object. In particular, this figure includes only nodes for input and output to or from external environments. The operation of a typical input transducer can be specified as:

```
$ Input transducer IT_i
 Q <---- M_FT^CI . 'ext_int' . 'source' . 'IT_i'
repeat
    sample_external_environment
    generate_data_packet (data)
    *Q <---- Data
    generate_state_word(SW)
    place (SW)
until     IT_shutdown.
```

The input transducer periodically generates a data packet and attaches it to the 'source' branch labelled 'IT_i'. Once a packet has been attached, a site of activity is created. The state word for the site of activity contains a pointer to the procedure responsible for handling the input packet.

Similarly, the 'sink' nodes are used by the internal procedures for the movement of data to the user environment. In the example of Figure 3.14, an internal procedure generates a data packet to be output and appends it to the appropriate '<u>sink</u>' node (OT_i); new output objects

a. 'ext_int' component



b. input packet

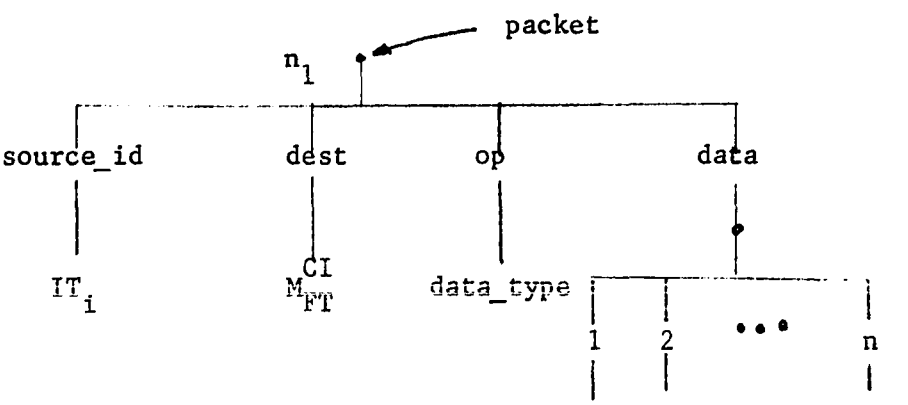


Figure 3.14. Packet communication with external environments.

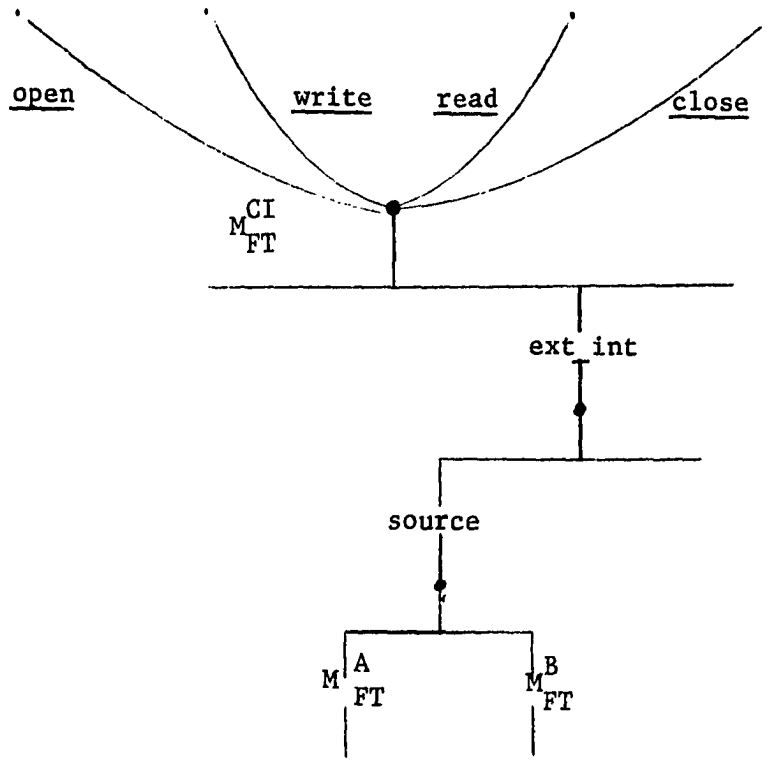
overlay older objects which were previously attached to the nodes.

An example of the state of an interpreter which receives inputs from an external interpreter is shown in Figure 3.15. The machine M_{FT}^{CI} supports a language which has four elements:

$$L_{FT}^{CI} = (\underline{read}, \underline{write}, \underline{open}, \underline{close}).$$

These language elements, or some subsets thereof, are accessed by the machines M_{FT}^{A} and M_{FT}^{B} (The instructions of L_{FT}^{CI} are external procedures to these machines). Whenever M_{FT}^{A} executes a call to the external procedure open, it generates a data packet which is appended to the branch labelled M_{FT}^{A}. Additionally, a site of activity is created whose 'proc' value points to the internal procedure (internal to M_{FT}^{CI}) open.

a. 'ext_int" component



b. Input packet

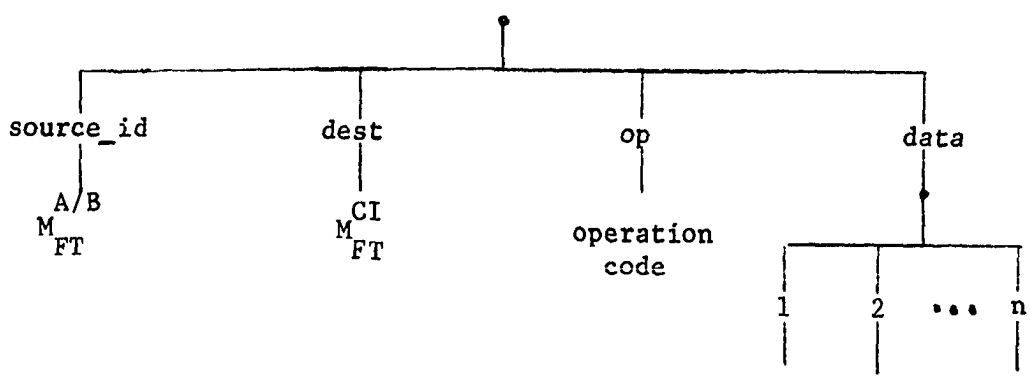


Figure 3.15. Packet communication with external interpreters.

CHAPTER 4

TIMED PETRI NETS

Introduction

Chapter 3 presented the notion of the System State Model. The model graphically depicts the complete state of an interpreter at any given instant in time. Additionally, a set of abstract instructions are used to transform the interpreter state. The System State Model specifies an abstract framework in which the functional definition of the problem solution can be made. The model graphically represents program and data objects, local and global environments, and system control objects.

The concept of a system state and a set of instructions, or inputs, for operating on the state are the essential ingredients for the representation of a finite state machine model of conventional automata theory. The material of this chapter introduces an additional graphical model, the Timed Petri net, which is used to specify the operation of the finite state machines. This model includes the representation of structural composition (parallelism, concurrency, and distribution of active resources) and temporal relationships (timing of state transitions).

The timing information available in the Timed Petri nets allows for an analytical determination of a bound on the computation rate of the associated state machine. In contrast to simulation techniques which are extremely time consumptive and expensive, the Timed Petri nets allows for a direct determination of the computation rate of an activity from

the structure and marking of the associated Petri net.

Furthermore, the Timed Petri net admits readily to a top-down design process. A particular operation in a Timed Petri net may be refined in a number of steps to a more detailed level of definition. At each level of definition the analytical power of determining the computation rate of the operations modelled is available to the designer.

The discussion in this chapter begins with a definition of Petri nets. The Petri net model is then extended to include timing information, thereby yielding the Timed Petri net. Next a technique for decomposing the nets into component state machines is discussed. Finally, a bound on the computation rate of the state machines represented by the Timed Petri net is determined from the state machine components.

The bulk of the material relative to Petri nets has been taken from the work of Ramchandi (52), and Holt and Commoner (26). That material has been tailored and extended to meet the needs of the MCS design technique. Several theorems representing the major results of these studies are presented. Unless otherwise specified, the proofs of the theorems are omitted. For a rigorous derivation of the results obtained herein, the reader should refer to Ramchandi (52).

Petri Nets

A Petri net is a bipartite acyclic directed graph. The net is formally defined as:

Definition 4.1 - A Petri net B is a three-tuple <P, T, A>

where P is a nonempty set of distinctly labelled places

$$P = (p_1, p_2, \ldots, p_n),$$

T is a nonempty set of distinctly labelled transitions

$$T = (t_1, t_2, \ldots, t_m), \qquad \text{and}$$

A is a relation which corresponds to a set of arcs. Each arc is either from a place to a transition or from a transition to a place.

$$A \subseteq (P \times T) \cup (T \times P).$$

An example of a Petri net is given in Figure 4.1. The nodes of the graph are either circles representing places or bars representing transitions. The arcs are directed branches connecting places to transitions and transitions to places.

Places represent the holding of a condition or system state. A transition defines the execution of a function (one or more instructions of a language) which transforms the system state.

The small darkened circle in place p_1 is a marker which traverses the net in the direction specified by the arcs of the net. The marker, called a token, is the active element of the Petri net. It is the movement of the tokens through the net which results in the transformation of the system state of the associated System State Model. For our purposes, a token may be viewed as an active processing resource or functional unit that is capable of executing the transformation specified by the transitions.

Definition 4.2 - A marking M is a function such that

$$M: \; P \longrightarrow I$$

where I is the set of nonnegative integers.

The nonnegative integers associated with a place represents the token load of that place, or the number of tokens on it.

A Petri net with a marking is referred to as a marked Petri net. In general, a distinction will only be made between a marked and an unmarked net if it is significant in the context in which it is presented.

A convenient notation called the dot notation can be used for the predecessor or successor nodes of any node in a net. Lex x and y represent any two nodes in a Petri net. Then

1. $<x,y> \in A$ is written $x \cdot y$,
2. $\{y \mid x \cdot y\}$ is written $x^{\bullet}$, and
3. $\{y \mid y \cdot x\}$ is written $^{\bullet}x$.

As an example of the use of the dot notation the following sets can be found in Figure 4.1:

$^{\bullet}t_2 = (p_2)$

$^{\bullet}p_4 = (t_2, t_3)$

$^{\bullet}t_2 \cup {}^{\bullet}t_3 = {}^{\bullet}(t_2, t_3) = (p_2, p_3)$

where $\cup$ is the set union operation.

A transition t in a Petri net is said to be enabled iff every input place ($p_i \in {}^{\bullet}t$) has at least one token on it. An enabled transition can be fired. When a transition fires, a token is removed from each input place and added to each output place ($p_j \in t^{\bullet}$). The firing of a transition corresponds to the execution of the function defined by that transition.

Live, bounded, and, persistent markings

A marking of a Petri net is live if all the transitions are usable throughout the course of the operation of a system. That is, there are no operations which are executed once and never again fired. It can be

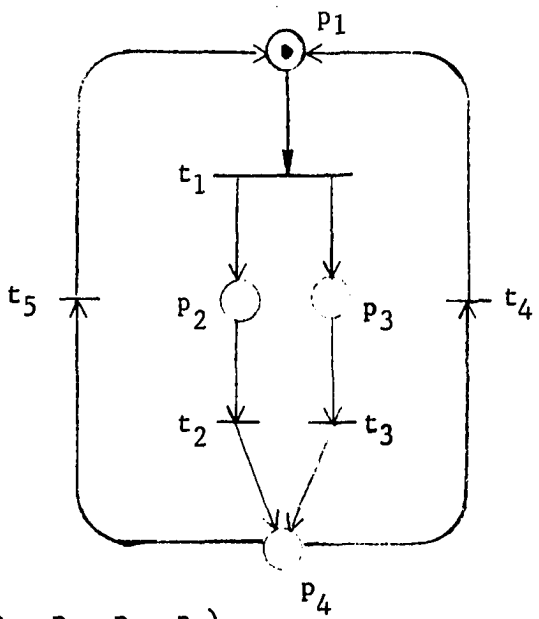
shown that a necessary requirement for a live marking is a Petri net which is strongly-connected, i.e., from every node there exists a directed path to every other node.

A marking is bounded for the operation of a net if there exists an integer N such that at all times $M(p_i) \leq N$ for all $p_i \in P$. A bounded net has a maximum of N tokens on any place at one instant in time. If the integer N is unity, the marking is said to safe.

Finally, a marking is persistent for a Petri net B if any transition $t_i \in T$, once enabled, cannot cease to be enabled by the firing of a transition $t_j \in T$ $(i \neq j)$. A net which is nonpersistent represents a system in which there is a distinct choice between alternate activities. An example of both a persistent and a nonpersistent net is shown in Figure 4.2. In Figure 4.2a there is a distinct choice which must be exercised between transition t_1 and t_2 when a token is in place p_1. However, in Figure 4.2b, transitions t_1 and t_2 will fire alternately as a function of the token content of places p_2 and p_3.

For notational convenience a net with a live bounded or live safe marking is termed an LB or LS net, respectively. If an LB net or LS net is also persistent, then the net is referred to as LBP or LSP. The net in Figure 4.2a is an LS net while the net in Figure 4.2b is an LSP net.

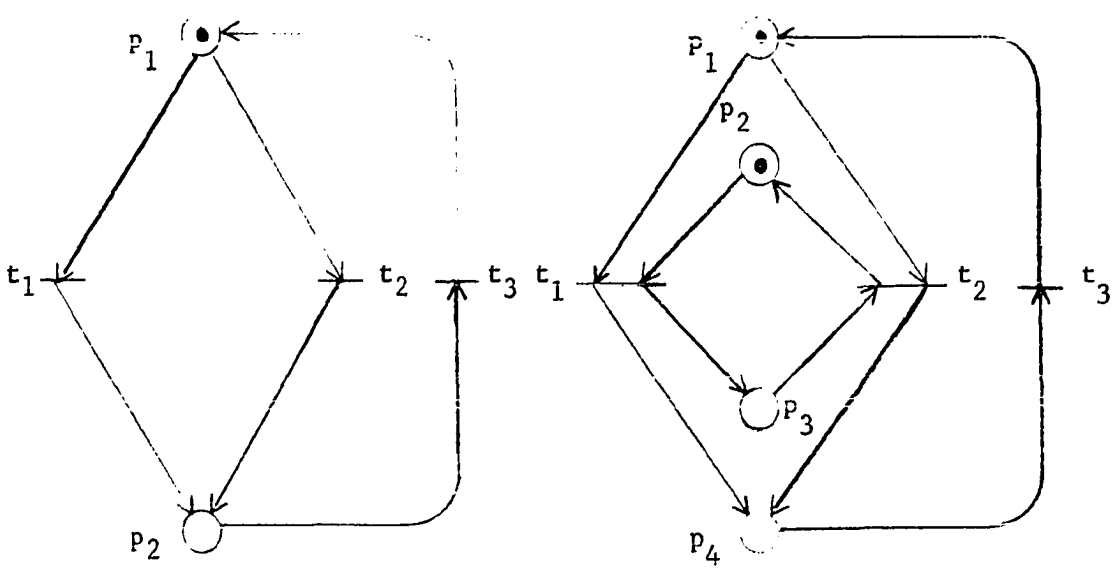
A more formal treatment of the properties of markings (liveliness, boundedness, and persistence) can be found in Holt and Commoner (26), and Ramchandi (52). In particular, their work proves that, given a marking for a Petri net B, it is decidable if the marking for the net is:



$P = (p_1, p_2, p_3, p_4)$

$T = (t_1, t_2, t_3, t_4, t_5)$

Figure 4.1. Petri net model.



a. Nonpersistent net (LS net) b. Persistent net (LSP net)

Figure 4.2. Persistent and nonpersistent Petri nets.

1. live
2. bounded (or safe), and
3. persistent.

The practical systems of interest in this research are live and bounded. In general, they are also nonpersistent.

Consistency

Consistency is a property of the structure of a Petri net independent of the marking of that net. In particular, consistency deals with the number of firings or executions of the transitions of a net. Consistency draws upon the concept of current flow from electrical circuit theory (Kirchoff's current law).

Definition 4.3 - A current assignment for a Petri net

$$B = < P, T, A >$$

is a function $\underline{\Phi}$ which assigns to each transition $t_i \in T$ a positive integer c_i called it's current. A current assignment for a Petri net must satisfy the following two constraints:

1. Every arc carries a current equal to that associated with the transition to which it is incident.
2. At every place, p_i, the sum of the currents on the input arcs must equal the sum of the currents on the output arcs.

That is for place p_i:

$$\sum_k c_{t_k \cdot p_i} = \sum_j c_{p_i \cdot t_j}$$

Definition 4.4 - A Petri net is consistent if and only if it has a current assignment $\underline{\Phi}$ with

$$c_i > 0 \qquad \text{for all } t_i \in T.$$

Consider the Petri net B of Figure 4.3. Let each transition $t_i \in T$ be assigned a current c_i. Each place $p_j \in P$ has an associated current equation which specifies the constraints on the input and output currents. These equations can only be valid if k_1 is identically zero. Therefore, the net is not consistent (inconsistent).

A slight modification of this net which leads to a consistent net is shown in Figure 4.4. The equations are consistent and have a solution which is a constant k . However, there is no unique solution for these equations. The value of k can be taken to be any positive integer.

Theorem 4.1 - A Petri net B which has a live, bounded marking is consistent.

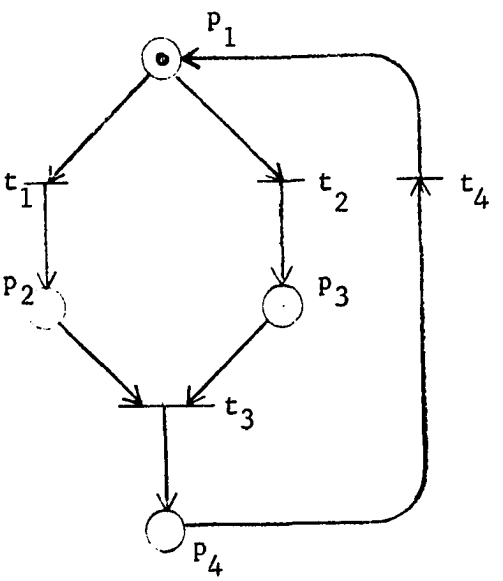
Since the nets we are interested in are LB, or LS, they are always consistent. Thus, the nets considered herein always have a current assignment $\underline{\Phi}$ for which $c_i \neq 0$ for all $t_i \in T$.

Timing Constraints and Petri Nets

In the discussion so far, the execution of transitions (instructions) have been considered independent of any timing considerations. In practical systems, operations require a nonzero amount of time for completion. In particular, each activity has a time duration different from zero, and all activities complete in a finite amount of time.

The concept of a Petri net can be extended to include timing information as follows:

a. Petri net B



b. Current equations

p_1: $c_4 = c_1 + c_2$

p_2: $c_1 = c_3$

p_3: $c_2 = c_3$

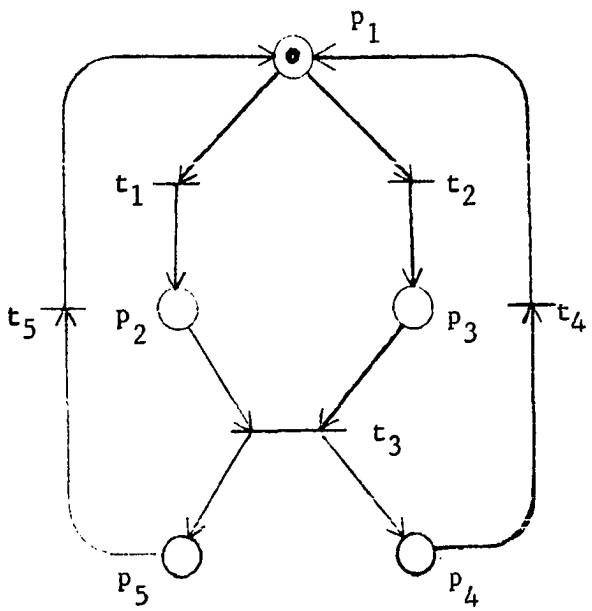
p_4: $c_3 = c_4$

$c_1 = c_2 = c_3 = c_4 = k$

$c_4 = c_1 + c_2 = 2k$

Figure 4.3. Inconsistent Petri net.

a. Petri net B



b. Current equations

p_1: $c_4 + c_5 = c_1 + c_2$

p_2: $c_1 = c_3$

p_3: $c_2 = c_3$

p_4: $c_3 = c_4$

p_5: $c_3 = c_5$

$c_1 = c_2 = c_3 = c_4 = c_5 = k$ (constant)

Figure 4.4. Consistent Petri net.

Definition 4.5 - A Timed Petri Net X is a pair

$$X = < B, W >$$

where B is a Petri Net, $B = < P, T, A >$, and W is a function that assigns a real nonnegative number τ_i to each transition $t_i \in T$.

$$W: \; T \longrightarrow \text{Reals}; \quad \text{Reals} = (\text{set of nonnegative reals}).$$

The nonnegative $\tau_i = W(t_i)$ is termed the firing time of a transition t_i. A transition t_i is enabled when there exists at least one token on each place of the input set $^{\bullet}t_i$. When t_i is enabled, a firing can be initiated. Upon initiation, a token is removed from each input place of $^{\bullet}t_i$ and transition t_i is said to be active. The active or execution phase lasts for τ_i seconds, where τ_i is the firing time of transition t_i. At the end of this time duration, the firing of transition t_i terminates, and a token is placed on each output place $p_j \in t_i$. This completes the firing of transition t_i.

The notation relative to the firing of a transition is in complete agreement with that previously discussed relative to the activity of the sites of activity of the System State Model. Once an active resource is assigned to a partially enabled transition, that transition become enabled and the initiation of the associated activity begins.

The active time of a firing τ_i has been defined to be a nonnegative real number. Thus, it has been assumed that the duration of the activity is fixed. However, in practical systems the value of τ_i may depend upon the nature of the data it is called upon to handle. For example, a transition might represent a function that searches a list for an element

matching a given key. The search time for the list is a function of the key and the number of elements containing that key. For such a transition, the active time may be assumed to be a random variable. The associated value of τ_i may then be chosen to be a mean time or possibly a worst case time depending upon the type of net analysis being considered.

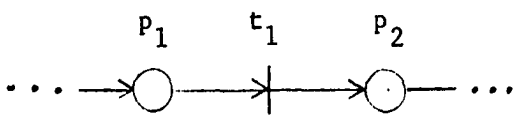
Note, however, that if a more accurate or finer resolution of the transition time is required, the function defined by the transition may be refined to a more representative model for which the associated times are more readily computed. An example of such a refinement process is given in Figure 4.5. Transition t_1 with current c_1 and firing time τ_1, is expanded into a more refined set of instructions (t_{11}, t_{12}, t_{13}, t_{14}). Given the current values c_{1i} and firing times τ_{1i} i=1, 2, 3, 4, the associated value of τ_1 can be obtained.

State Machine Decomposable Petri Nets

LB Petri nets constitute a very large class of possible nets. A more restricted class of Petri nets which can be used to model RTDAD systems (as will be seen in Chapter 5) is a class of nets known as state machine decomposable (SMD) Petri nets. SMD Petri nets will be shown to have the desirable property that a bound on the computation rate of the entire net can be obtained by analytically determining the computation rates of the associated state machine components. Given such a bound, a number of interesting characteristics of the system can be investigated.

The following material discusses the properties of a net which permit a state machine decomposition. In particular, the notion of state machines

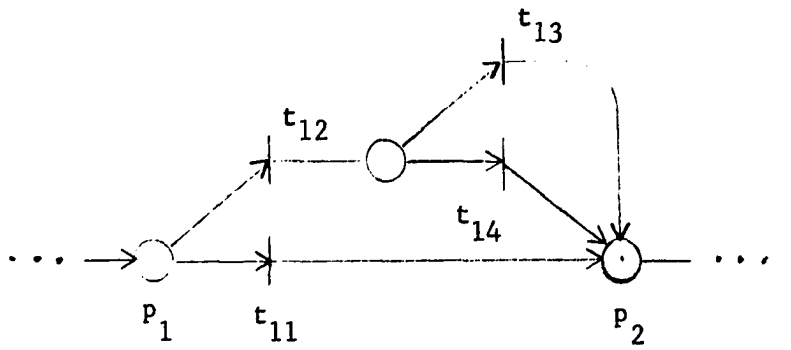
a.



$\tau_1 = W(t_1)$

$c_1 = \Phi(t_1)$

b.



transition	current	firing time
t_{11}	c_1-X	τ_{11}
t_{12}	X	τ_{12}
t_{13}	X-X_2	τ_{13}
t_{14}	X_2	τ_{14}

c.

$$\tau_1 = \left(\frac{c_1-X}{c_1}\right)\tau_{11} + \frac{X}{c_1}\tau_{12} + \left(\frac{X-X_2}{c_1}\right)\tau_{13} + \frac{X_2}{c_1}\tau_{14}$$

Figure 4.5. Refinement of transition firing time.

and SMD Petri nets are formally defined.

Definition 4.6 - A closed subnet of a Petri net B is a strongly-connected Petri net

$$B^1 = \langle P^1, T^1, A^1 \rangle$$

where

$P^1 \subseteq P$ is a set of places

$T^1 \subseteq T$ is a set of transitions

$A^1 \subseteq A$ is a set of arcs such that

$${}^\bullet p^1 = P^1{}^\bullet = T^1 \text{ and } A^1 = [(P^1 \times T^1) \cup T^1 \times P^1] \cap A.$$

The Petri net N in Figure 4.6 has five closed subnets (N_1, N_2, N_3, N_4 and N). Clearly, every strongly-connected Petri net is a closed subnet of itself, because the relation ${}^\bullet P = P^\bullet = T$ is trivially satisfied.

Definition 4.7 - A closed subnet is a minimal closed subnet if and only if no closed subnet can be obtained by deleting any portion of it.

In Figure 4.6, N is not a minimal closed subnet since the four closed subnets are obtainable from N by deleting appropriate places and/or transitions. However, N_1, N_2, N_3 and N_4 are easily verified to be closed subnets.

As an additional example consider the Petri net B of Figure 4.7. B has three minimal closed subnets (S_1, S_2, S_3). The subnet S_4 is obviously not closed since it can readily be decomposed into subnets S_1 and S_2.

Definition 4.8 - A Petri net B is a state machine if and only if every transition has exactly one input place and exactly one output place.

An example of a state machine is given in Figure 4.8. The structural restriction of one incident arc and one emanating arc at any transition

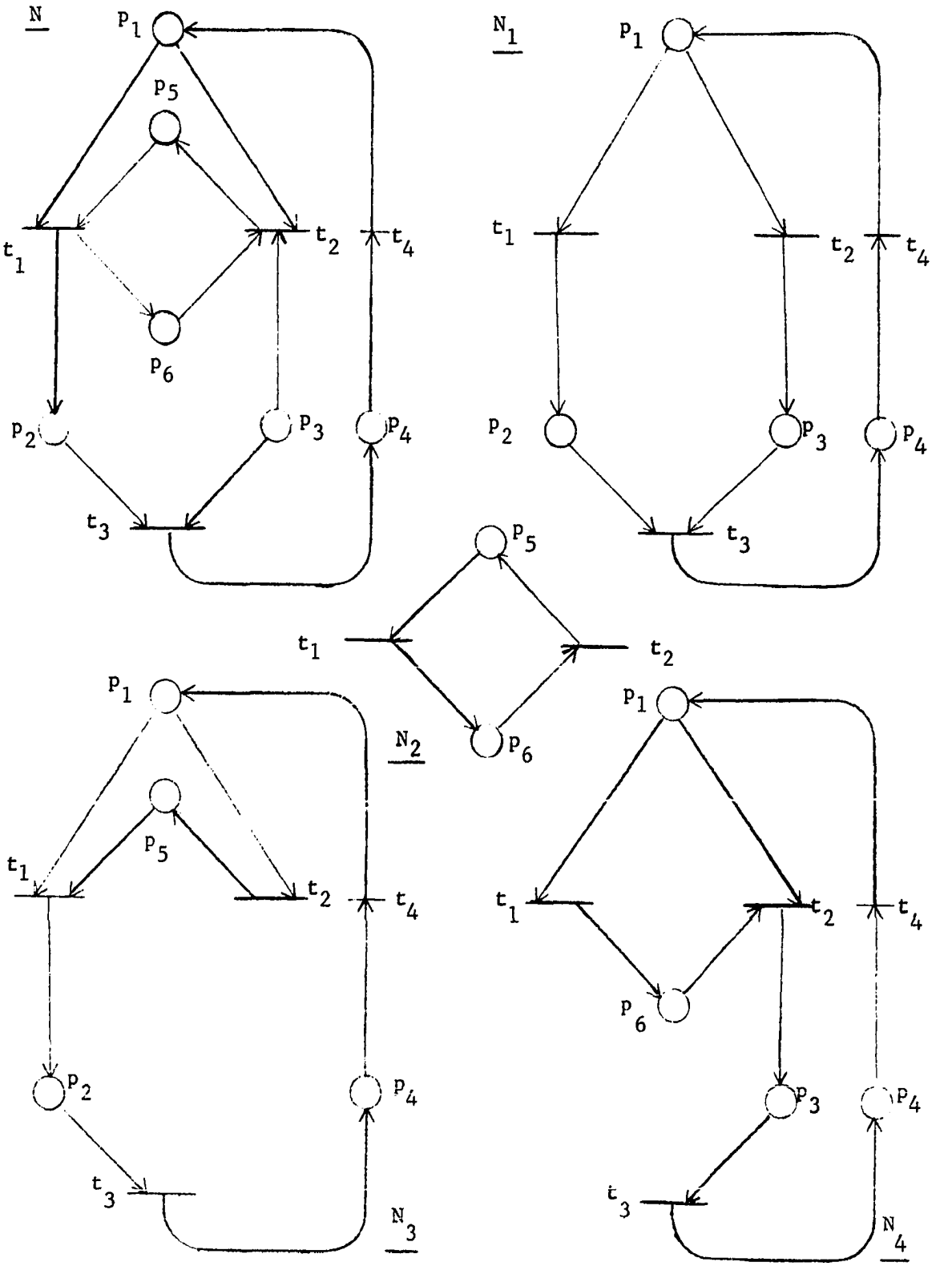


Figure 4.6. Closed subnets of Petri net N.

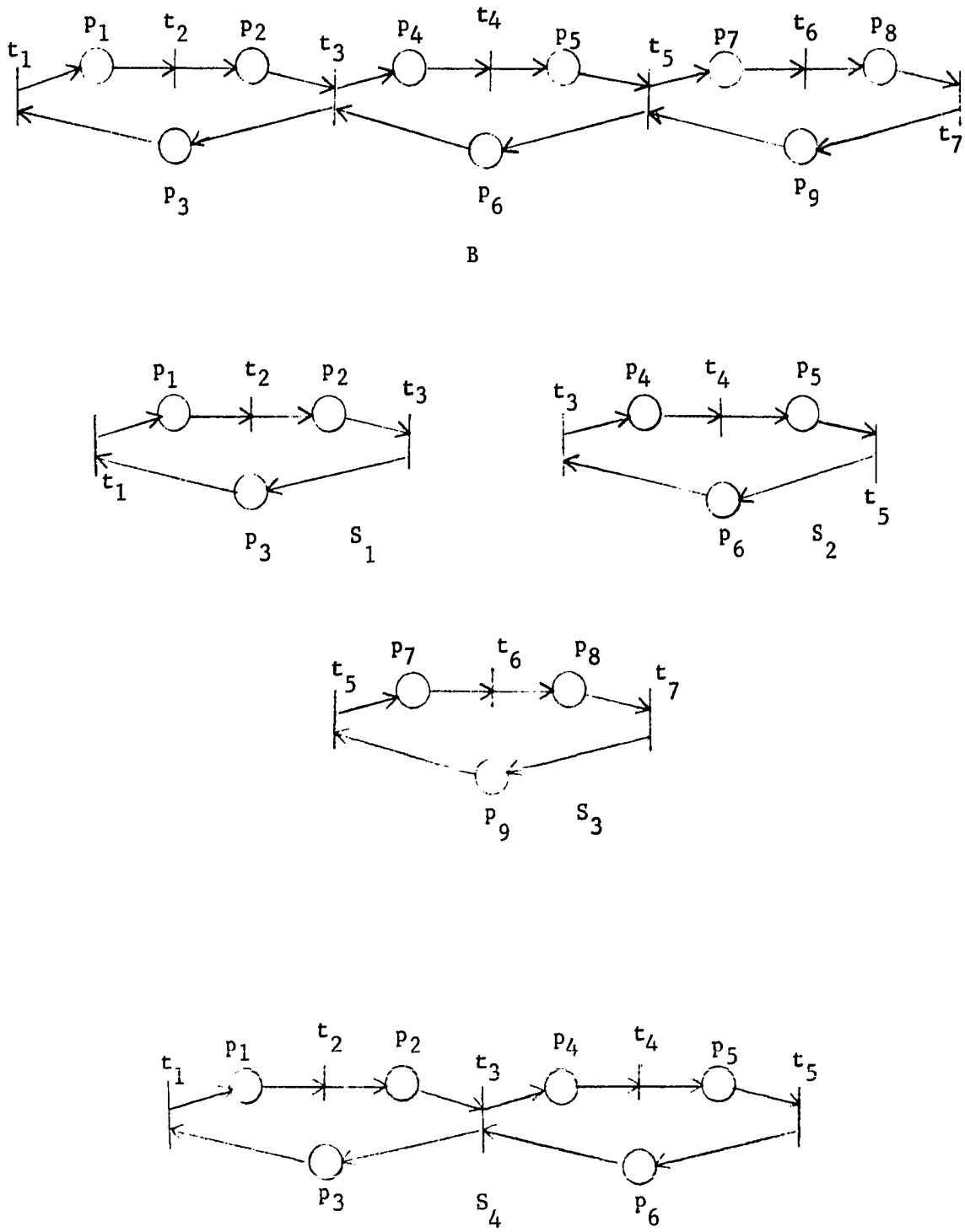


Figure 4.7. Subnets of Petri net B.

is compatible with the familiar concept of the state diagram representation of finite state machines. Specifically, the restriction implies that given any state represented by a holding of a condition, one and only one new state will be generated as the result of the firing of a transition.

Definition 4.9 - A Petri net B is said to be covered by a set of closed subnets $N_1, N_2, \ldots, N_k$ if and only if

$$P = \langle \underset{i}{UP_i}, \underset{i}{UT_i}, \underset{i}{UA_i} \rangle$$

where

$$N_i = \langle P_i, T_i, A_i \rangle \quad 1 \leq i \leq k.$$

The Petri net B in Figure 4.7 is covered by the set of closed subnets S_1, S_2, S_3 .

Definition 4.10 - A Petri net B is state machine decomposable (SMD) if and only if every minimal closed subnet S_i is a state machine and there exists a set of state machines $(S_1, S_2, \ldots, S_k)$ which covers the net.

The Petri net B in Figure 4.7 is SMD. However, the net N in Figure 4.6 is not SMD because the minimal closed subnets N_1, N_2 and N_4 are not state machines.

One of the interesting and important properties of an SMD Petri net is that all markings for it are bounded. That is, the number of tokens in any one place in the net will always be less than some integer N.

Dynamic Behavior of SMD Petri Nets

Thus far, the general characteristics of Timed Petri nets have been described. The discussion has focused on a subset of nets which are live,

bounded, and state machine decomposable (LB SMD). With this material as background, the dynamic behavior of an SMD net operating in real time can be considered.

First, the dynamic nature of a Timed Petri net representing a simple circuit is analyzed. A computation rate for the net is computed. The example is then extended to include two circuits which share a common transition, i.e., a net that can be decomposed into two state machines which are each simple circuits. In particular, the impact of the common transition on the computation rate is investigated.

Finally, the construction of an equivalent model for an LB SMD Petri net is presented. The equivalent model, termed the occurrence graph, is generated in such a manner as to result in an LBP net. A technique for computing the bound on the computation rate for the original LB SMD net can be obtained by an analysis of the equivalent LBP occurrence graph.

Computation rates of simple circuits

Consider the circuit of Figure 4.9. Each transition of X is assigned a time τ_i by the function W, i.e., $\tau_i = W(t_i)$. Assume that the marking of the net places one token in place p_i. The operation of the net consists of the transitions firing in sequence as the token traverses the circuit. Let $\Pi = \tau_1 + \tau_2 + \ldots + \tau_m$. Then, the token fires every transition in the circuit in turn and reappears on p_i every Π seconds. Under this assumption, every transition initiates at intervals of Π seconds, and Π is the period for the simple circuit corresponding to its maximum computation rate. Thus the maximum computation rate denoted by R is

$$R = \frac{1}{\Pi} = 1/\sum_{i=1}^{n} \tau_i$$

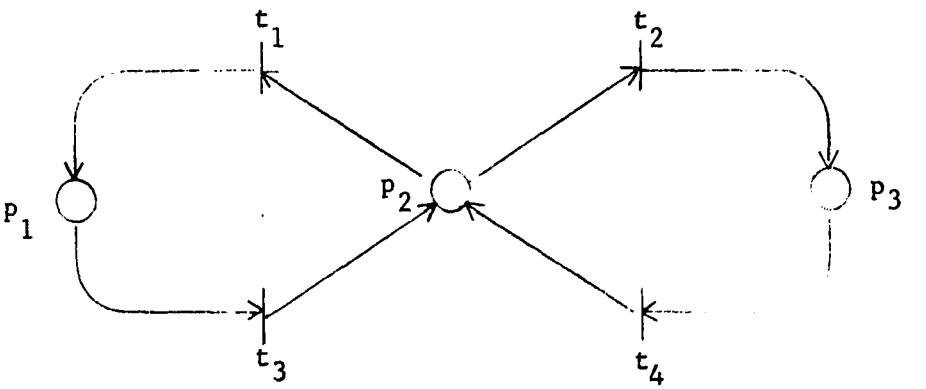


Figure 4.8. State machine net.

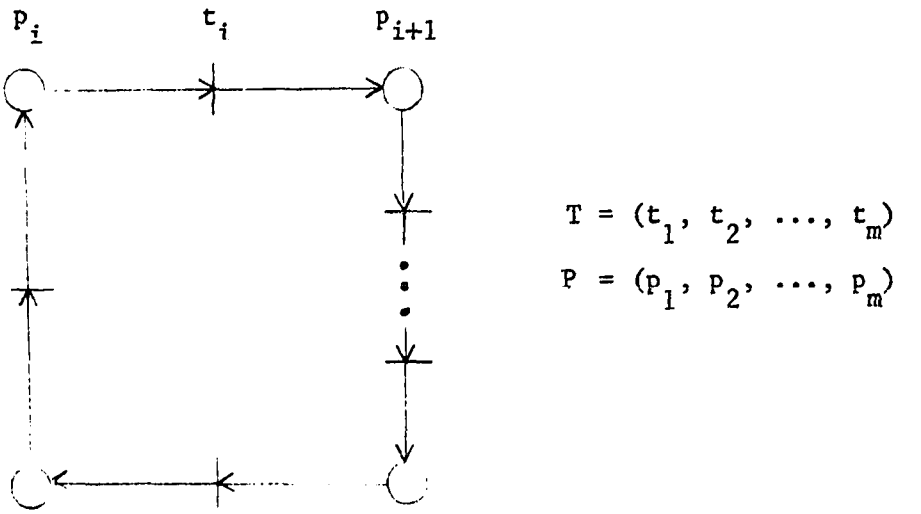


Figure 4.9. Simple circuit $X = \langle B, W \rangle$.

Now assume that instead of one token, n tokens are distributed across the net. For the combined action of the n tokens, the firing rate becomes

$$R = \frac{n}{\pi} = \frac{n}{\sum_{i=1}^{m} \tau_i} .$$

Every transition in the circuit has a maximum computation rate given by this expression. The computation rate R is termed the natural computation rate of the circuit.

Consider next an LSP SMD net which consists of two circuits which share a common transition. An example of such a net is given in Figure 4.10a. Additionally, assume that the marking places one token on each of the two circuits. The state machine decomposition of the net B is shown in Figure 4.10. Let π_1 and π_2 be the periods for SM_1 and SM_2, respectively. Then the natural computation rates of the two state machines are:

$$R_1 = \frac{1}{\pi_1} = \frac{1}{\sum_{SM_1} \tau_i}$$

$$R_2 = \frac{1}{\pi_2} = \frac{1}{\sum_{SM_2} \tau_j}$$

These are the computation rates of the state machine subnets if they were completely isolated from each other. However, in a strongly-connected net, the circuits are interconnected and, intuitively, it is clear that they effect each other's natural computation rate.

Without loss of generality, let $R_1 < R_2$. Obviously, all transitions in SM_1 have a computation rate which is given by R_1. In particular,

transition t_2 fires at a maximum computation rate given by R_1. Since t_2 is also a node in SM_2, the computation rate of the transitions on a directed path from t_2 in SM_2 cannot have a computation rate greater than R_1. If they were to operate at a rate $R_2 > R_1$, then transition t_2 would be executing at a computation rate larger then that of the slower circuit. However, this contradicts the original assumption that t_2 fires at rate R_1.

The argument can be easily extended to LBP SMD Petri nets which consist of a set of simple circuits $(C_1, C_2, \ldots, C_k)$[1]. The resultant computation rate for the entire net defines the fundamental computation rate given by:

$$R = \min\left(\frac{n_1}{\pi_1}, \frac{n_2}{\pi_2}, \ldots, \frac{n_k}{\pi_k}\right)$$

where n_i is the token content of C_i and

π_i is the sum of the firing times of the transitions in circuit C_i.

C - equivalent nets

This section introduces the concept of occurrence graphs and consistency-equivalent nets for the general class of LB SMD Petri nets. The technique for generating the occurrence graph and deriving the associated consistency-equivalent net is demonstrated by example. Once derived, the consistency-equivalent net defines a set of circuits whose computation

---

[1] The result is rigorously derived by Ramchandi (52). It is sufficient for our purposes to have a firm grasp on the intuitive interaction of the computation rate for a multiplicity of circuits.

rates can be analytically determined in the manner prescribed in the last section. The fundamental computation rate for the system is then established by an examination of the natural computation rates of the constituent circuits.

Consider the SMD Petri net B of Figure 4.11. The starting point for the generation of the occurrence graph is to draw and label the set of marked places in B. Let this set be P_1. For Figure 4.11, $P_1 = (p_2, p_5)$. Let T_1^1 be the set of enabled transitions corresponding to P_1, and note that P_1 corresponds to the initial marking of B.

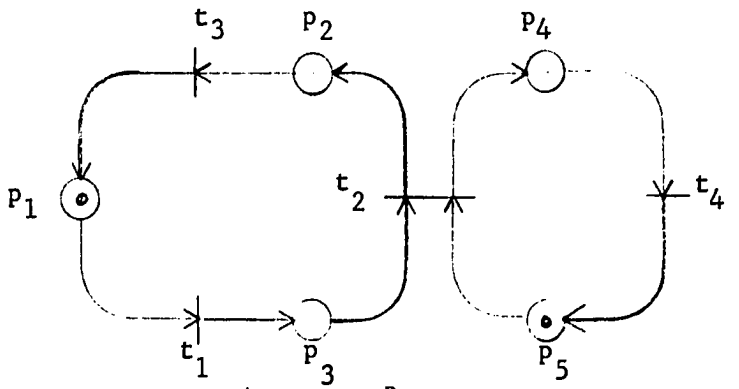
Since a place $p_i \in P$ may have more than one output transition, several enabled transitions may be connected to the same place. Since only one of these transitions can be fired, a choice is made between them and only that one fired. Let $T_1 \subseteq T_1^1$ be the set of transitions which are enabled and fired. Let P_2^1 be the set of marked places that result when the transitions in T_1 have completed firing.

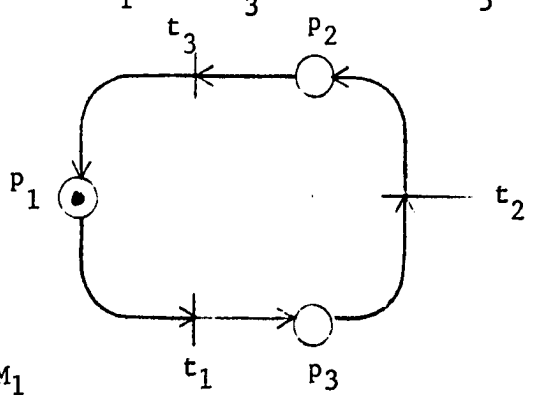
Next draw all arcs $P_1 \times T_1$ that are contained in A. Draw the places $P_2^1 = T_1^{\bullet}$. Draw all the arcs $T_1 \times P_2^1$ which are contained in $T \times P$. Finally, define $P_2 = (P_1 - {}^{\bullet}T_1) \cup T_1^{\bullet}$.

The process of constructing the marked places P_{k+1} that result when all transitions in T_k are fired is called <u>extending</u> the occurrence graph from P_k to P_{k+1}. Since B has a live marking, the occurrence graph can be extended indefinitely.

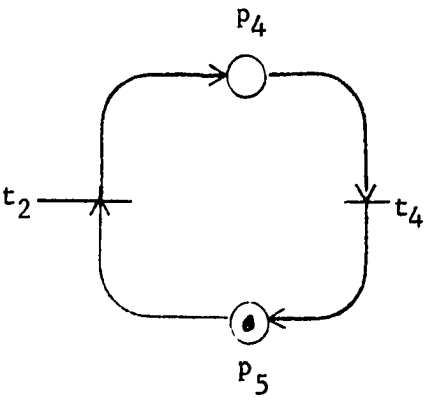
A <u>slice</u> of an occurrence graph is a set of places that forms a cut-set (6) of the graph. The construction technique amounts to extending the slice from one slice to the next.

a. Petri net B





b. SM_1



c. SM_2

Figure 4.10. LSP SMD net.

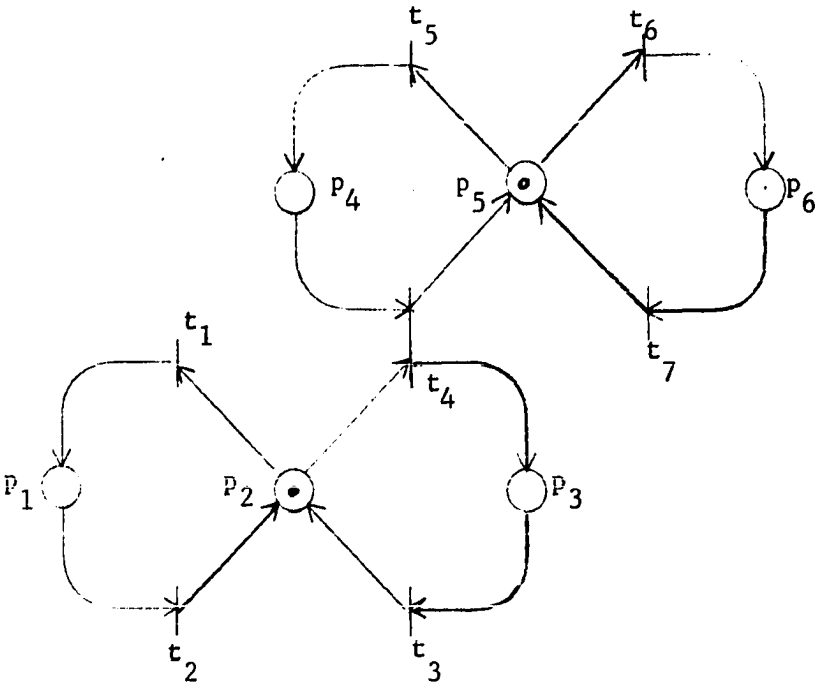


Figure 4.11. LS SMD Petri net.

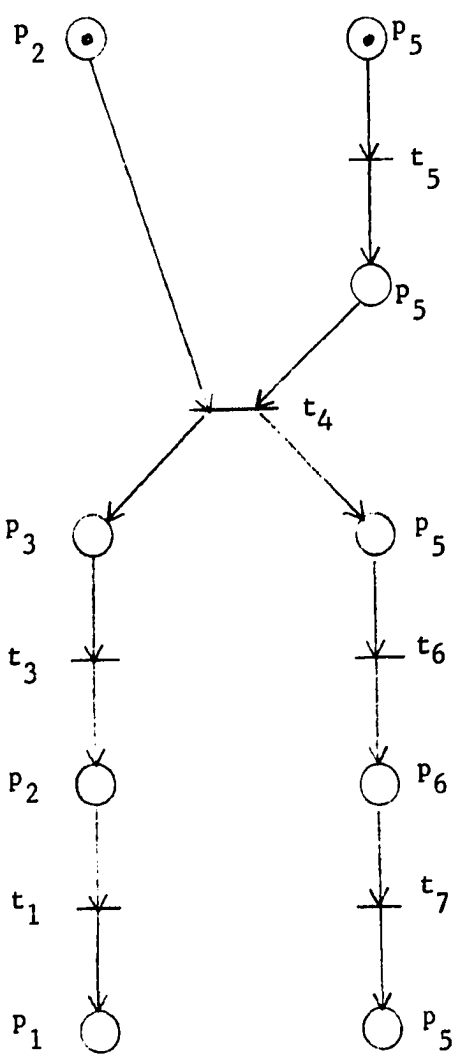
In the construction of an occurrence graph for a LP SMD Petri net, there never occurs a slice for which a choice has to be made between output transitions. Such an occurrence graph is called a behavior graph. Thus, there is only one behavior graph for an LP SMD Petri net and this graph is unique.

On the other hand, several occurrence graphs may be possible for an SMD Petri net. In Figure 4.12 two possible occurrence graphs for the net of Figure 4.11 are shown. A cursory examination reveals the infinite number of occurrence graphs that are possible for this net, or, for that matter, in any LB SMD net with a nonpersistent marking.

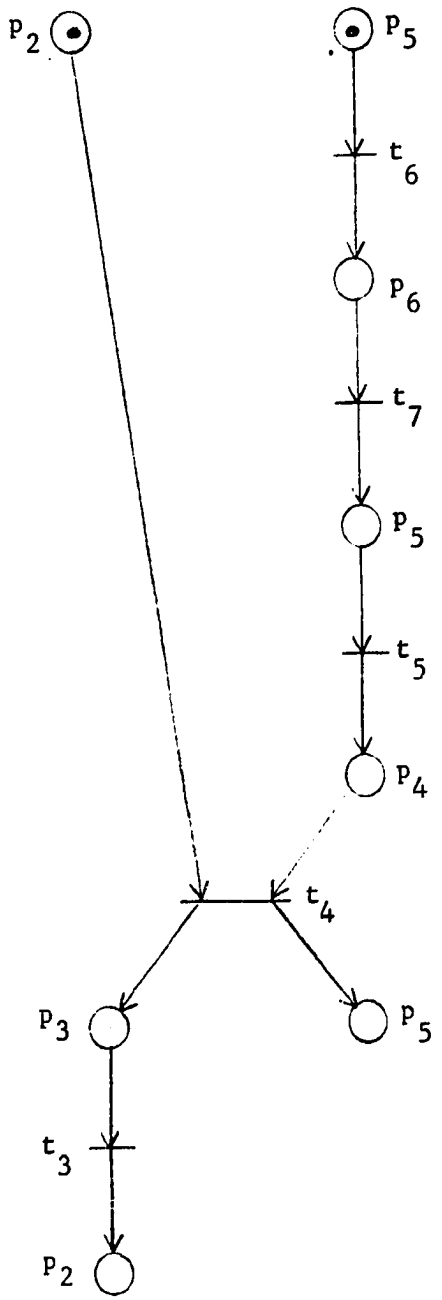
It can be rigorously shown that there exists a slice in the occurrence graph of an SMD net that occurs repeatedly. The portion of an occurrence graph between two consecutive occurrences of a slice is termed a cyclic frustrum. Additionally, it can be proved that for any consistent current assignment for an LB SMD Petri net, there exists a cyclic frustrum in the occurrence graph of the net. Furthermore, the number of occurrences of any transition in the cyclic frustrum equals its current in a consistent current assignment.

Consider again the SMD Petri net of Figure 4.11. In Figure 4.13, a consistent current assignment is given and one possible cyclic frustrum is drawn.

The two repeated slices of the cyclic frustrum are coalesced as shown. The resulting strongly-connected net is termed a consistency-equivalent net for the SMD Petri net, abbreviated to c-equivalent net.



a. Occurrence graph 1.



b. Occurrence graph 2.

Figure 4.12. Occurrence graphs.

a. Consistent current assignment

p_1: $c_1 = c_2$ $\qquad c_1 = c_2 = K_1 = 1$

p_2: $c_2 + c_3 = c_1 + c_4$ $\qquad c_3 = c_4 = c_5 = K_2 = 2$

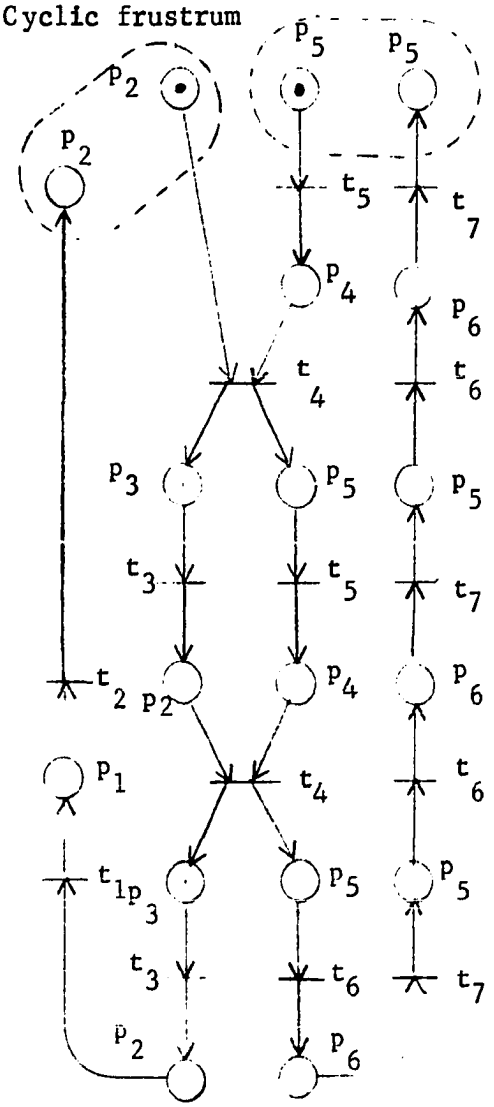
p_3: $c_4 = c_3$ $\qquad c_6 = c_7 = K_5 = 3$

p_4: $c_5 = c_4$

p_5: $c_4 + c_7 = c_5 + c_6$

p_6: $c_6 = c_7$

b. Cyclic frustrum



c. Consistency-equivalent net

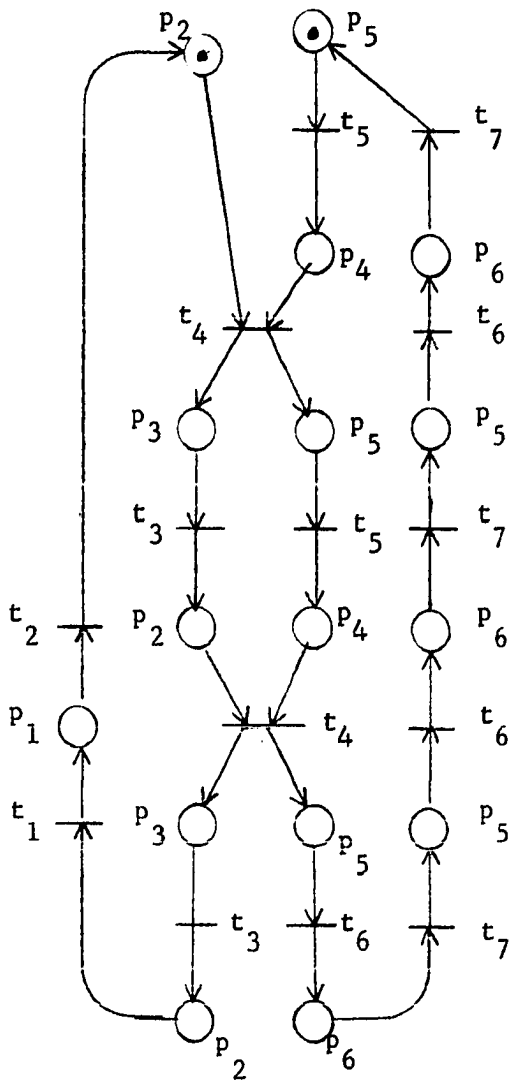


Figure 4.13. Derivation of a consistency-equivalent net.

Note that the c-equivalent net for an SMD Petri net is not unique. By an alternate choice of transition firings (at places p_2 and p_5) a different c-equivalent net can be realized. Of extreme importance, however, is the following result:

Theorem 4.2 - Let B be an LS SMD Petri net with a consistent current assignment $\underline{\Phi}$, and CN be a c-equivalent net for B. Then, every state machine component in B corresponds to a single circuit in CN.

Referring to the c-equivalent net of Figure 4.13, there are four circuits:

C_1: $p_2t_4p_3t_3p_2t_4p_3t_3p_2t_1p_1t_2$

C_2: $p_5t_5p_4t_4p_5t_5p_4t_4p_5t_6p_6t_7p_5t_6p_6t_7p_5t_6p_6t_7$

C_3: $p_2t_np_5t_5p_4t_4p_3t_3p_2t_1p_1t_2$

C_4: $p_5t_5p_4t_4p_3t_3p_2t_4p_5t_6p_6t_7p_5t_6p_6t_7p_5t_6p_6t_7$.

where C_1 and C_2 correspond to the two state machines which result from a state machine decomposition of the net B. (The state machines are derived by splitting the net into two machines at transition t_4). Note also that there are two circuits, C_3 and C_4, which do not correspond to any state machine in the SMD Petri net.

Computation rate for Timed SMD Petri nets

Consider an SMD Petri net B for which one possible c-equivalent net is denoted by CN_i. The set of all possible circuits in CN_i is the set of circuits $C_1, C_2, \ldots, C_k$. In a manner analogous to the discussion of the computation rate in the previous section, it can be shown that the maximum computation rate for CN_i is given by:

$$R_{CN_i} = \min \frac{n_j}{\sum_{t_i \in C_j} \tau_i} \qquad j = 1, 2, \ldots, k$$

where n_j is the number of tokens on circuit C_j; and

τ_i is the firing times of the transitions in C_j.

That is, the maximum fundamental computation rate for a c-equivalent net CN_i is found by determining the computation rate R_j for each circuit C_j in CN_i and selecting the smallest R_j. Thus, $R = \min (R_1, R_2, \ldots, R_k)$ defines the computation rate of CN_i. Furthermore, the maximum computation rate for a transition in the c-equivalent net is given by $c_i R$ where c_i is the number of times transition t_i fires during one period (one complete cycle though the cyclic frustrum).

If the SMD Petri net B is persistent, the c-equivalent net is unique and R defines the maximum computation rate for the net B. However, for an SMD Petri net which is nonpersistent there are a number of possible c-equivalent nets. In general, the maximum fundamental computation rates of two c-equivalent nets are different. In order to find the maximum fundamental computation rate of B, the timed c-equivalent net which has the largest computation rate must be found.

The maximum fundamental computation rate of a c-equivalent net for the Timed Petri net B represents the maximum fundamental computation rate of transitions in the net B for the behavior specified by that c-equivalent net. This notion leads to the following definitions:

Definition 4.11 - The maximum fundamental computation rate of transitions in a timed SMD net $X = < B, W >$ for a consistent current assignment $\overline{\Phi}$

is given by the fundamental computation rate of the c-eqivalent net which has the largest fundamental computation rate.

Definition 4.12 - The maximum computation rate of a transition t_i belonging to a timed SMD net $X = \langle B, W \rangle$ with a consistent current assignment $\underline{\Phi}$ is given by

$$r_i = c_i R$$

where $c_i = \underline{\Phi}(t_i)$ and R = maximum fundamental computation rate for X.

In order to obtain an exact value for the maximum fundamental computation rate R of X, all the c-equivalent nets for X must be found. The c-equivalent net with the largest fundamental computation rate defines the value of R.

This is clearly a very tedious process. It is desirable to find a simple method which gives a bound on the fundamental computation rate of X. In particular, a bound can be determined from the state machine decomposition of X as specified by the following theorem.

Theorem 4.3 - In a Timed SMD Petri net $X = \langle B,W \rangle$ with a consistent current assignment $\underline{\Phi}$, the maximum fundamental computation rate is bounded by

$$R = \min \left[R_1, R_2, \ldots, R_m \right]$$

$R_1, R_2, \ldots, R_m$ are the fundamental computation rates of the state machine components of X. The fundamental computation rate R_k of state machine S_k is given by

$$R_k = \frac{n_k}{\sum_{j=1}^{r} c_{kj} \tau_{kj}}$$

where n_k = number of tokens on state machine S_k

$t_{k1}, \ldots, t_{kr}$ are the transitions of S_k

C_{kj}, τ_{kJ} are the current and firing time, respectively, of transition t_{kj}.

Proof: Consider the c-equivalent net CN of the SMD Petri net B for the consistent current assignment $\underline{\Phi}$. For every state machine component S_k in B, there exists a corresponding circuit C_k in CN with the following property:

Every transition t_i in C_k has a multiplicity equal to the current c_i assigned to transition t_i in B by $\underline{\Phi}$.

Now let $C_1, \ldots, C_m, C_{m+1}, \ldots, C_r$ be the simple circuits in CN, where $C_1, \ldots, C_m$ correspond to state machines of B and $C_{m+1}, \ldots, C_r$ are simple circuits in CN that do not correspond to state machines of B. Let $R_1, \ldots, R_m, R_{m+1}, \ldots, R_r$ be their respective fundamental computation rates. For any circuit $C_i \in \{C_1, \ldots, C_M\}$

$$R_i = \frac{n_i}{\sum_{C_i} c_{ij} \tau_{ij}}$$

Now the fundamental computation rate R' of the timed net X is bounded by

$$R' = \min [R_1, \ldots, R_r]$$

$$R' = \min [R_1, \ldots, R_m, R_{m+1}, \ldots R_r]$$

$$\text{or } R' \leq R = \min [R_1, \ldots, R_m].$$

Note that there may exist $R_S \in [R_{m+1}, \ldots, R_r]$ such that $R_S < \min [R_1, \ldots, R_m]$. Thus, while R' is certainly a bound, this bound may not be achievable. The computation rate r_i of any transition t_i is bounded by $r_i = R' * c_i$.

Theorem 4.3 establishes a bound on the computation rate of any transition in a Timed SMD Petri net by finding for each state machine component S_i the corresponding fundamental computation rate R_i. This is certainly simpler than finding the maximum fundamental computation rate of all the c-equivalent nets for the Timed SMD Petri net.

Since Theorem 4.3 plays a major role in the remainder of the thesis, it is appropriate that a numerical example should be considered. Suppose the SMD net of Figure 4.11 has the current assignment given in Figure 4.13. Additionally, assume that the firing time assignment is as specified in Figure 4.14a. Two possible c-equivalent nets and their maximum fundamental computation rates are shown in Figure 4.14b and Figure 4.14c. The maximum computation rate from Theorem 4.3 can be seen to be given by:

$$R = \min \left[R_1, R_2\right]$$

$$R_1 = \frac{1}{\sum_{i=1}^{4} C_i \tau_i} = \frac{1}{15}; \quad R_2 = \frac{1}{\sum_{i=4}^{n} C_i \tau_i} = \frac{1}{29}$$

$$R = \frac{1}{29}.$$

The maximum computation rate for the net from two c-equivalent nets is

$$R = \max\left[\frac{1}{30}, \frac{1}{29}\right] = \frac{1}{29}.$$

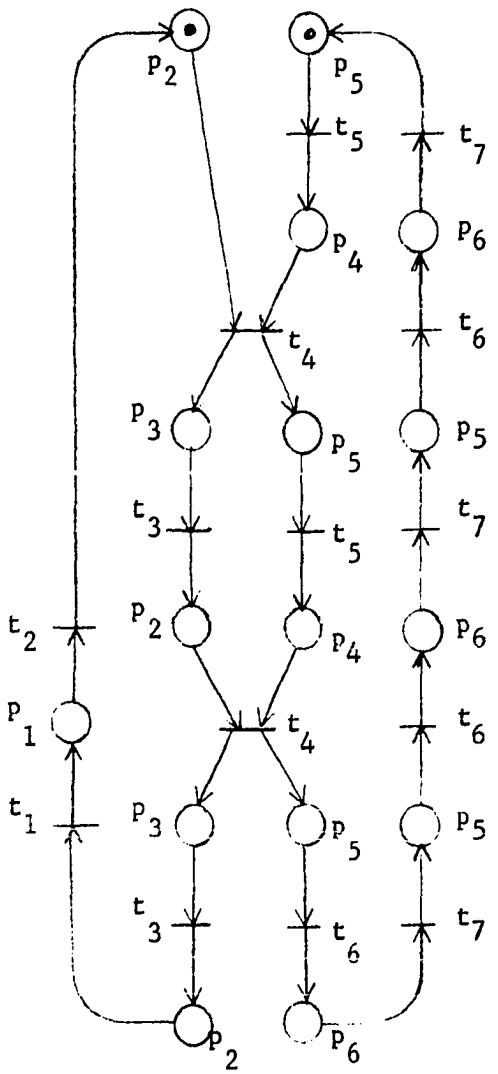
In this particular example, the bound determined by Theorem 4.3 is the same as would be determined by the more exhaustive technique of generating all possible c-equivalent nets.

The maximum fundamental computation rate for any transition t_i in the SMD Petri net can now be found from the equation $r_i = \bar{R}c_i$.

a. firing time assignment W

Transition	Time	Transition	Time	Transition	Time
1	2	4	1	6	5
2	3	5	3	7	2
3	4				

b. c-equivalent net 1



c. c-equivalent net 2

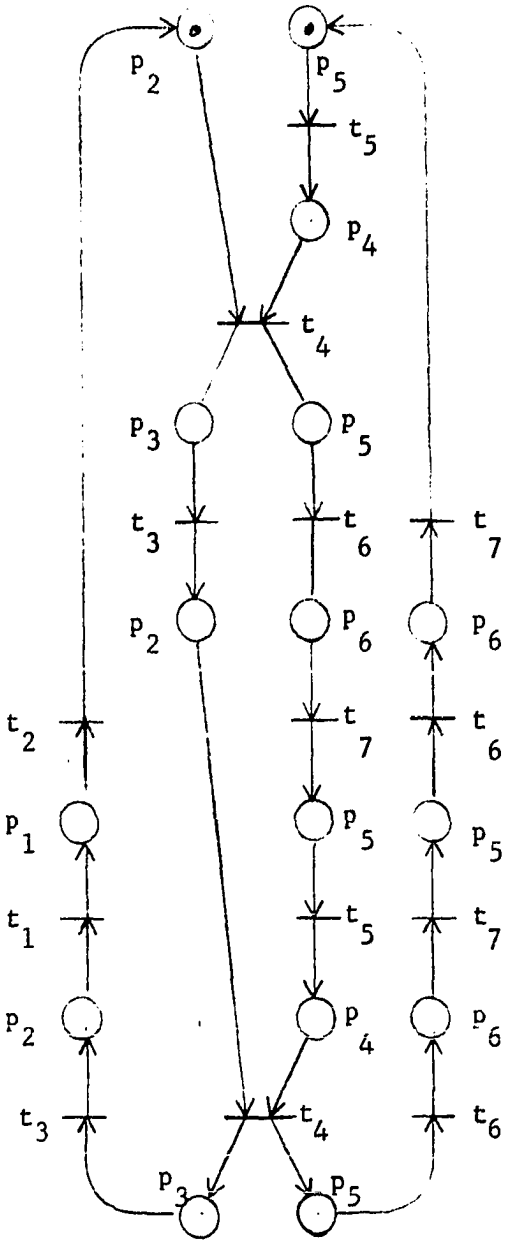


Figure 4.14. c-equivalent nets for the Petri net of Figure 4.11.

CHAPTER 5

APPLICATION OF THE MCS DESIGN APPROACH

Introduction

Thus far, the concept of a finite state machine representation of an RTDAD system has been supported by the introduction of two graph-theoretic models.

1. The System State Model specifies the state of the machine. through the abstract representation of programs and data, and it describes the semantics of the functional operation upon that state.
2. Timed Petri Nets define a technique for:
   a. modelling the flow of control in a system,
   b. representing the concurrency achieved at any point in time, and
   c. quantifying the computation rate of components of a system and, thereby, establishing a bound on the maximum fundamental computation rate of the system as a whole.

The material of Chapter 5 substantiates the appropriateness of these models and provides several applicable examples. Because of the extensive nature of the details necessary to describe a typical system, many of the concepts that need to be emphasized might be lost in an exhaustive presentation. Instead, a few representative examples expressing the highlights of the foregoing material at a manageable level of presentation are discussed.

Timed Petri Nets in the RTDAD Environment

Before presenting specific examples, a few words of introduction are necessary. Principally, a few comments about the applicability of the Timed Petri net analysis to the RTDAD environment are in order.

Flow of control in Timed Petri nets

Chapter 2 introduced the notion of a multiplicity of machine-language levels in the representation of an RTDAD problem solution. Furthermore, the concept of an MCS design approach was shown to employ the repeated use of hardware, firmware, and software technologies for the implementation of algorithms necessary for that problem solution.

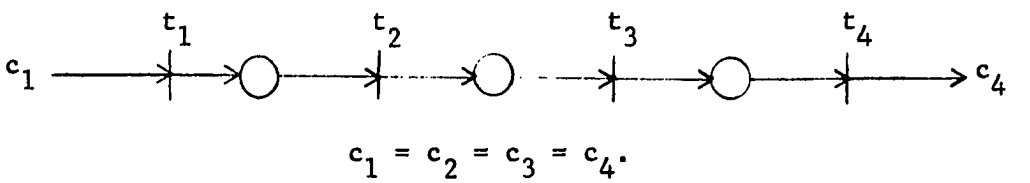
These algorithms are functionally specified by the abstract instructions of the System State Model.

The abstract instructions support the semantic definition of the operations required to provide the necessary computations. Timed Petri nets, in turn, support the representation of the flow of control or sequencing necessary to implement these algorithms.

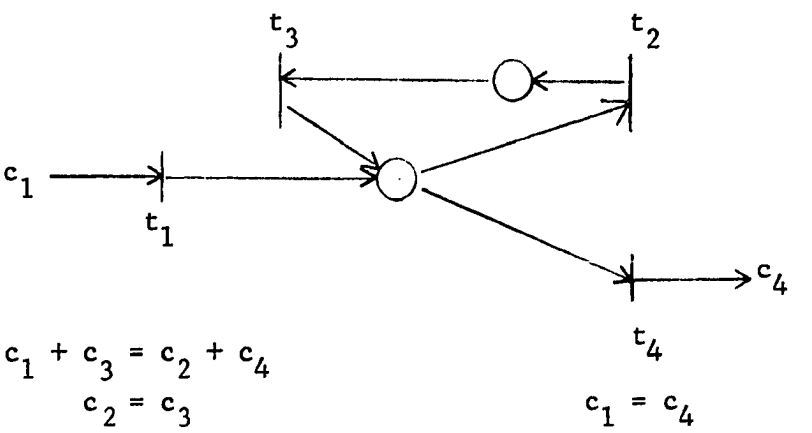
Bohm and Jacopini (3) first showed that a sufficient set of control structures for expressing any flow-chartable algorithm can be realized through the use of three basic constructs, namely:

1. sequencing or concatenation,
2. IF - THEN - ELSE conditional branching, and
3. WHILE - DO conditional iteration.

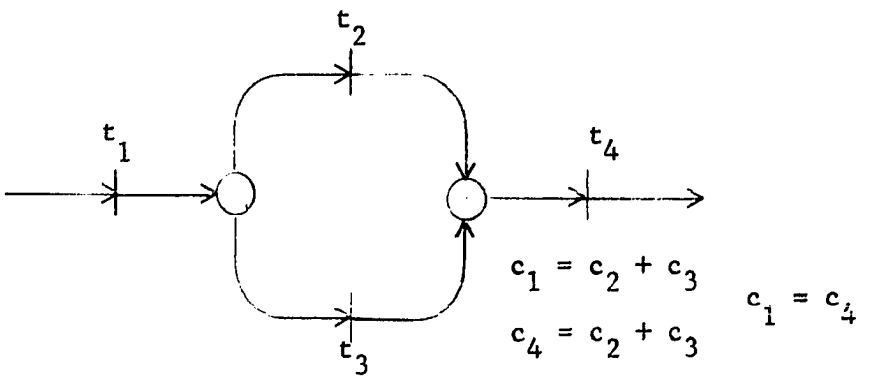
Figure 5.1 gives the Petri net representation for each of these types of control structures. Since Petri nets can be used to represent the



a. concatenation



b. WHILE_DO



c. IF_THEN_ELSE

Figure 5.1. Complete set of Petri net constructs.

minimum set of constructs, they form an appropriate vehicle for the representation of algorithms expressible in the abstract instruction of the System State Model.

Note, however, that the algorithms expressed in this manner do not presume a technology in which the operations are to be specified. It is inherent in the entire design philosophy that there is a complete freedom of choice[1] of implementation based on quantitative measures of system performance.

Explicit and implicit resource description

One of the salient characteristics of the use of Petri nets, in contrast to more conventional forms of flowcharts, is the inclusion of system resources in the model description. In particular, resources can appear explicitly or implicitly.

Tokens, or markings, on the Petri nets represent active processing elements (primitive modules) which are capable of executing the functions defined by the semantics of the transitions. Thus, the token content of a net explicitly depicts the processing resources of the associated system. Active processing elements include central processing units, functional units, and controllers.

Implicit resources are those resources whose presence (or absence) affects the definition of a net. That is, the availability of implicit

---

[1]Naturally not all transitions can be specified independently of one another. The freedom of choice is of a general nature reflecting the overall design objectives.

resources affects the structure of a net or one of the two net functions W or $\overline{\Phi}$.

As an example of an implicit resource, consider a system which has a hardware stack feature. The operations PUSH and POP for a hardware stack typically execute faster than the same operations in a more conventional software implementation of a stack[1]. The inclusion of such a feature manifests itself in a reduction of the firing times for those transitions employing the hardware stack operations.

As a second example, the Petri net structure for a system which has just enough main memory resource to include all software modules is significantly different from the same system with a lesser amount of main memory. The latter system must provide those processes necessary to support some form of dynamic memory management.

Applicability of Timed Petri nets

In order for the results of the previous section to be applicable to the RTDAD environment, the following conditions need to be satisfied:

1. The Timed Petri net for an RTDAD solution must be state machine decomposable.
2. The functions W and $\overline{\Phi}$ must be defined.

By an earlier assumption, our scope of interest has been restricted to conventional primitive modules capable of strictly sequential execution

---

[1] Not only is a reduced firing time achieved with the hardware stack, but also a reduction is possible in the amount of memory required to support the invocation and implementation of the stack operations.

of instructions. That is, each active resource or token of a Petri net executes the transitions it encounters in a sequential manner.

The set of transitions of a Petri net $B = \langle P, T, A \rangle$ can be dichotomized into internal and external transitions with

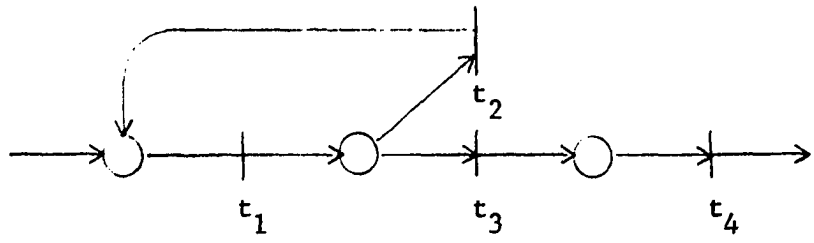
$$T = T_{int} \cup T_{ext} \qquad (T_{int} \cap T_{ext} = \emptyset).$$

External transitions are those transitions which occur at the interface between communicating subnets of B. On the other hand, internal transitions portray operations relegated to only one subnet of B.

Because of the sequential nature of primitive modules, internal transitions are executed one at a time in the order specified by the net structure. Therefore, for internal transition $t_i \in T_{int}$ both $\cdot t_i$ and $t_i \cdot$ consist of a singular place (refer to Figure 5.2a).

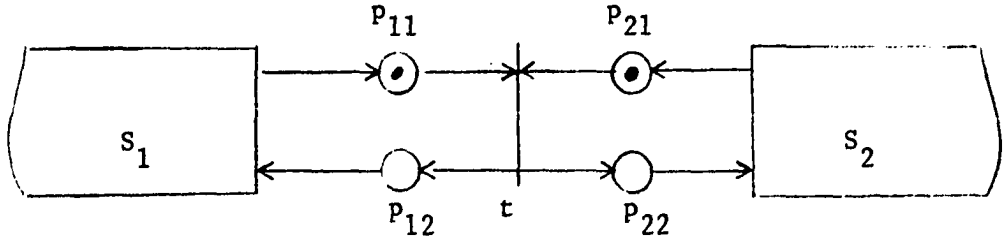
External transitions are common to two subnets of B. An enabled interface transition $t \in T_{ext}$ is shown in Figure 5.2b. The resultant marking after the firing of t has terminated is given in Figure 5.2c. By definition of tokens as unique active resources, the tokens originally in places p_{11} and p_{21} appear in places p_{12} and p_{22}, respectively. Since there must be a conservation of active resources, the external transitions always return a token to the subnet from which it originated. Thus, all external transitions are of the form shown in Figure 5.2b. The subnets S_1 and S_2 may, therefore, be "split" at the external transition t forming two components with one incident arc and one emanating arc from each reproduction of transition t.

Since all internal transitions and all external transitions resulting from such a decomposition have one incident arc and one emanating arc,

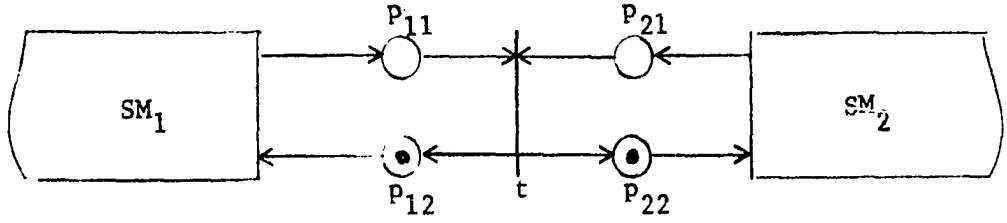


$T_{int} = (\ldots, t_1, t_2, t_3, t_4, \ldots)$

a. Internal transitions



b. External transition (enabled)



c. External transition (post-execution)

Figure 5.2. Internal and external transitions.

the Petri net B is state machine decomposable (SMD).

The timing function W is dependent upon both the explicit and implicit resources. Once a set of primitive modules has been determined, the definition of all real and virtual machines can be specified. A knowledge of the operating characteristics of the primitive modules allows for the resolution of the firing times for the transitions of the nets.

The current function $\underline{\Phi}$ depends upon the statistical nature of the external environment. In particular, the external environment needs to be sufficiently well-defined such that the statistical distribution of current at each decision place can be enumerated. Specifically, that requires the input from the external environment be

1. "periodic" in nature, and
2. statistically defined over the period of interest.

These constraints imply the existence of a finite period of time T in which the number and type of input packets can be completely specified. The RTDAD environment satisfies both restrictions. The input transducers typically generate data packets at constant or predictable rates. Furthermore, the sampling of the external environment is performed in a precisely known manner. That is, the number and type of input data packets can be predicted a priori. When the number cannot be precisely defined, a mean and/or worst case set of packets can be enumerated over a known period of time.

Given a known distribution of input packets, the discrete probability distribution function for alternate routes (arcs) at each decision place can be found. The current assignment can then be determined from the

probability density function. For example, Figure 5.3 depicts the use of a probability density function at place p_i to compute the associated output currents. Thus, the current assignment ($\underline{\Phi}$) for the transition of the net can be determined from the composition of the input stream and the required operations specified in the problem statement.

Therefore, for the RTDAD environment, the function $\underline{\Phi}$ and W are definable. Additionally, the Timed Petri net representing the RTDAD system of interest are SMD. Since both conditions (a decomposable net and definable net functions) have been satisfied, the results of the previous chapter are applicable to these systems.

Boundary conditions

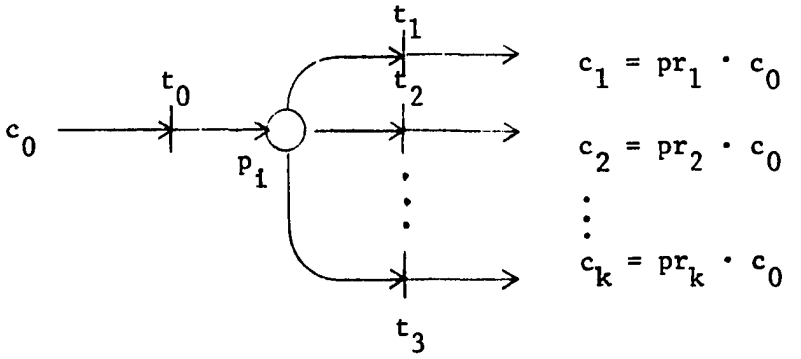
The general RTDAD system configuration of Figure 1.1 is reproduced in Figure 5.4a. The natural computation rates for each of the three constituent parts are shown in the figure. The input transducer may be represented by the Petri nets of the form given in Figure 5.4b. The natural computation rate of an input transducer is

$$R_{IT} = \frac{1}{\sum_{IT} c_i \tau_i} = \frac{1}{c_{IT} \sum_{IT} \tau_j} = \frac{f}{c_{IT}} \quad \text{where } f = \frac{1}{\sum_{IT} \tau_i} \quad \text{is}$$

the sample rate of the transducer.

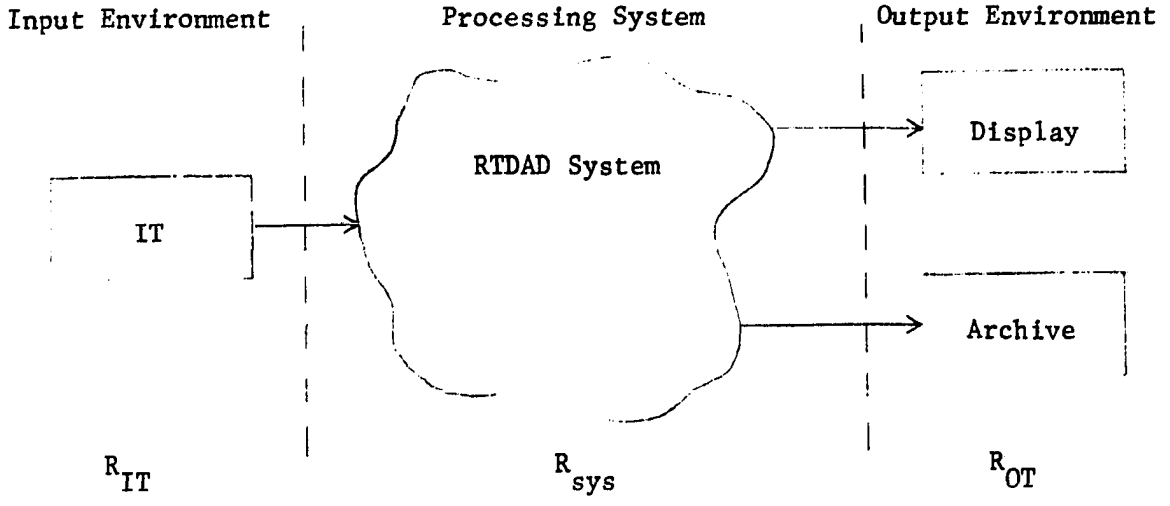
The computation rate R_{IT} establishes a boundary value for the operation of the system. That is, the maximum computation rate of a system with one input transducer must be

$$R_{IT} \leq \min \left[R_{sys}, R_{OT}\right] = \min \left[R_{s1}, R_{s2}, \ldots, R_{sk}, R_{OT}\right]$$

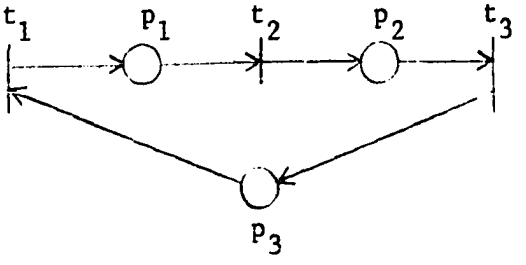


Probability density function $pr = (pr_1, pr_2, \ldots, pr_k)$

Figure 5.3. Current assignment at a decision place.



a. General RTDAD system configuration



b. Net structure for input transducers.

$$c_{IT} = c_1 = c_2 = c_3$$

$$R_{IT} = \frac{1}{\sum c_i \tau_i} = \frac{1}{c_{IT} \sum \tau_i} = \frac{f}{c_{IT}}$$

$$f = \frac{1}{\sum_{IT} \tau_i}$$

Figure 5.4. RTDAD system configuration.

where R_{Si} is the computation rate for state machine component

$$S_i \ (i = 1, 2, \ldots, k).$$

Since it has been assumed that the typical input transducers are operating in a continous mode, the addition of multiple input transducers does not alter the input computation rate bound. In fact, the computation rates for k input tranducers are given by

$$R_{IT} = \frac{f_1}{c_{IT1}} = \frac{f_2}{c_{IT2}} = \ldots = \frac{f_k}{c_{IT_k}}$$

where $$f_j = \frac{1}{\sum_{IT_j} \tau_i}.$$

The ratio of frequency to current remains constant because a linear increase in frequency involves an offsetting linear increase in current.

However, the inclusion of additional input tranducers has a significant impact on the current assignment $\underline{\Phi}$ for the system net. The augmented input load increases the current values for existing transitions and/or requires the inclusion of new transitions. The change in $\underline{\Phi}$ induces a corresponding reduction in the computation rates of part or all of the system components.

The boundary value R_{IT} is a function of the nature of the input stream and the operations to be performed on that stream. In particular, for each unit of input, composed of a well-defined mix of packets, the boundary values R_{IT} can be determined. The unit of input is termed the data set.

Definition 5.1 - A data set, DS , is a well-defined and repeatable set of input packets. The amount of time required to receive DS is the period of the data set, T_{DS} .

The value of T_{DS} includes two operational characteristics associated with the given system, namely:

1. the amount of time allowable for the input of the data set, and
2. the total system response time for the operation resulting from the input of the data set.

An important property of the data set is its repeatability. A given system must be able to respond to a repeated application of a data set or a combination of data sets. Additionally, a worst case data set may be considered as a benchmark for system performance. The ability of the system to meet the worst case data set defines the feasibility of the proposed system design.

Example I - Structural Identification of Macro-parallelism

As mentioned in the Chapter 1, a primary motivating force behind this investigation is the exposure of possible concurrency among functions (macro-parallelism). Such parallel activity can be seen by an examination of the structure of Timed Petri nets. Specifically, three forms of parallelism of interest will be identified in this example. It will be assumed that a predetermined set of primitive modules is to be applied to given Petri net structures. The Petri nets considered are assumed to represent the functional activity necessary to satisfy a given problem statement.

The three forms of parallelism to be investigated are

1. sequential concurrency (pipelining),
2. disjoint concurrency (process overlap), and
3. n^{th} order parallelism (multi-processing).

Sequential concurrency

Consider the Timed Petri net $X = <B, W>$ ($B = <P,T,A>$) of Figure 5.5a. There is one active resource (single token) defining the operation of the net. The computation rate for X is given by

$$R_1 = \frac{1}{\sum_{i=1}^{5} c_i \tau_i} .$$

Suppose that the processing time of the concatenated sequence of operations is uniformly distributed across the transitions, i.e., $t_i \doteq$ constant for all i. An improvement in the maximum computation rate for B can be realized by the net $X^1 = <B^1, W>$ of Figure 5.5b. X^1 includes two active resources with half the net apportioned to each resource. X^1 is SMD and can be decomposed into the two state machines S_1 and S_2.

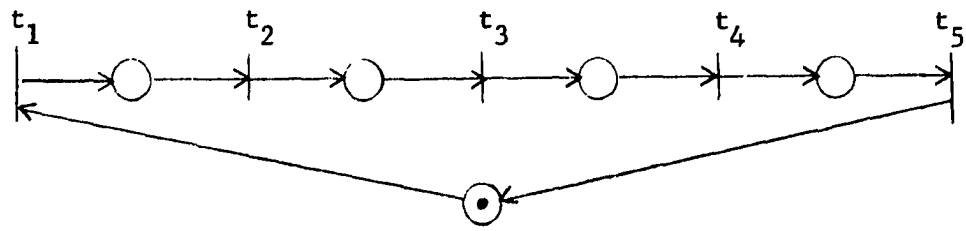
The maximum computation rate for X^1 can now be seen to be

$$R_2 = \min \left[R_{S1}, R_{S2} \right]$$

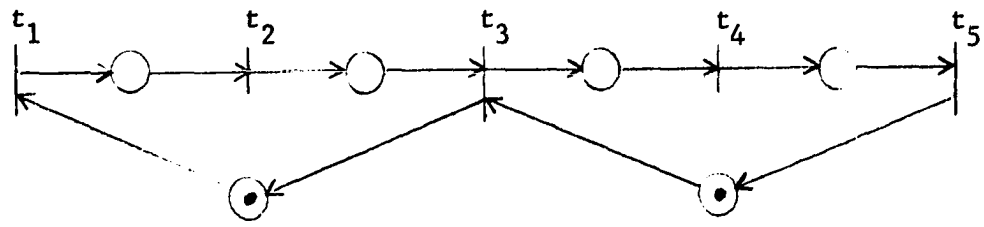
where

$$R_{S1} = \frac{1}{\sum_{i=1}^{3} c_i \tau_i} \quad \text{and} \quad R_{S2} = \frac{1}{\sum_{i=3}^{5} c_i \tau_i} .$$

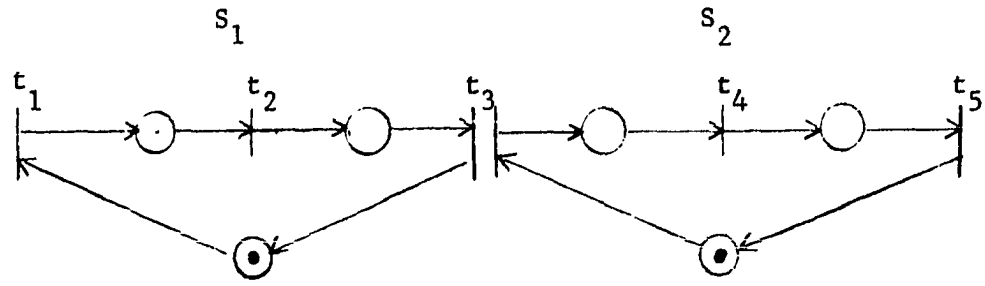
Since the transition times were assumed to be approximately equal, we have $R_{S1} \doteq R_{S2}$, and $R_2 \doteq 2R_1$. The concurrency so derived is referred to as sequential concurrency or pipelining.



a. Sequential execution



b. Pipelined execution



c. SMD equivalent subnets

Figure 5.5. Example of strict concurrency.

An important concept in pipelining is system balance. Basically, the set of pipelined sections can operate no faster than the slowest pipelined section. Therefore, it is desirable if all sections of a pipeline are designed to operate at nearly the same computation rate.

Disjoint concurrency

Consider next the Timed Petri net $X = <B, W>$ $(B = <P, T, A>)$ given in Figure 5.6. Assume that X is part of a larger net which has been isolated for the purpose of this example[1]. Further assume that the four Petri net sections represent four processes which operate on disjoint sets of variables V_1, V_2 and V_3 as follows:

1. process 1 operates on a variable set V_1,
2. process 2 operates on a variable set V_2, and
3. process 3 and 4 operate on a variable set V_3.

Such processes are termed disjoint or noninteracting processes (4,5).

If the box labelled active resource contains a singular processing element (one token) that is to be shared among the four processes, then the maximum computation rate is given by

$$R_1 = \frac{1}{\Sigma c_i \tau_i} .$$

Fundamentally, the structure embodies the concept of a multiprogrammed uniprocessor system. That is, the activity of the various system processes are interleaved in time.

---

[1] It is assumed that the processes communicate with the remainder of the net by means of data structures such as FIFO queues.

Alternatively, the four processes of Figure 5.6 could be assigned active resources as in Figure 5.7. The parallelism achieved by such a structure is termed disjoint concurrency. The maximum computation rate for the net of Figure 5.7 is

$$R_2 = \min\left[R_{S1}, R_{S2}, R_{S3}\right]$$

where

$$R_{S1} = \frac{1}{\sum_{1}^{3} c_i \tau_i}$$

$$R_{S2} = \frac{1}{\sum_{4}^{6} c_i \tau_i}$$

$$R_{S3} = \frac{1}{\sum_{7}^{12} c_i \tau_i}$$

clearly $R_2 > R_1$ since

$$R_{S1}, R_{S2}, R_{S3} > R_1.$$

Note that if one of the four processes has a large current-time ratio ($\Sigma c_i \tau_i$) as compared to the other processes, the application of additional resources shows only a slight improvement in the maximum computation rate. Thus, the concept of system balance also applies to disjoint concurrency.

n^{th} order parallelism

Finally, consider the Timed Petri net $X = < B, W >$ ($B = <P, T, A>$) of Figure 5.8. The net consists of a number of processes represented by sections of the net B. The portion of the net contained inside the box, the Task Controller, is that part of the system responsible for assigning

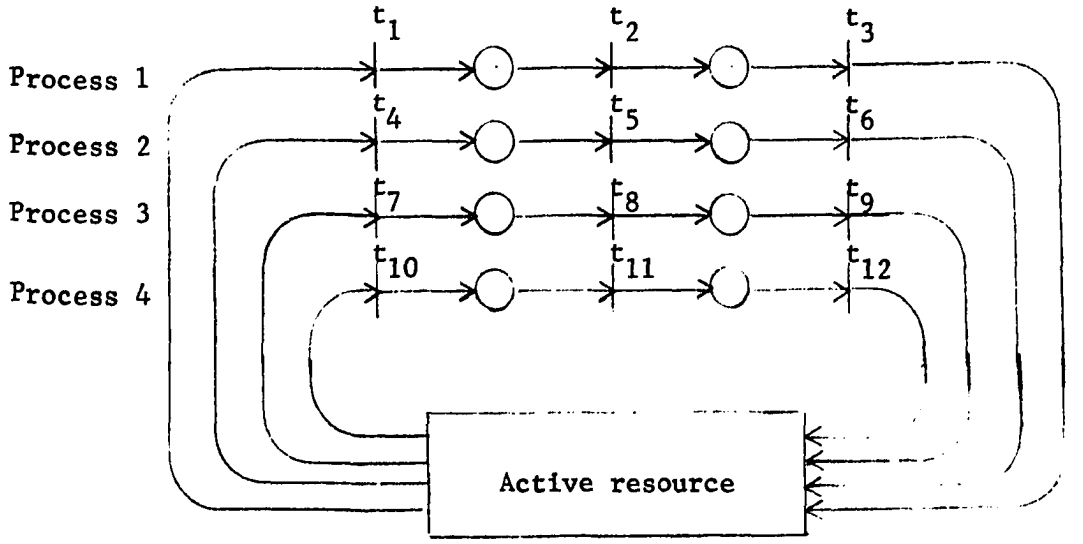


Figure 5.6. Disjoint processes-general model.

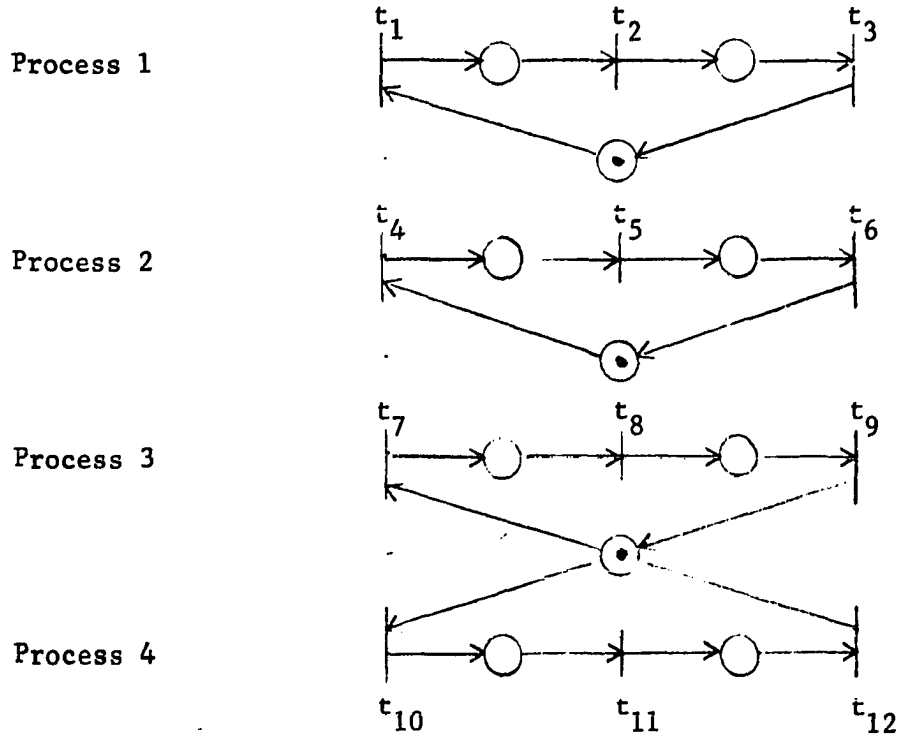


Figure 5.7. Disjoint processes - disjoint concurrency.

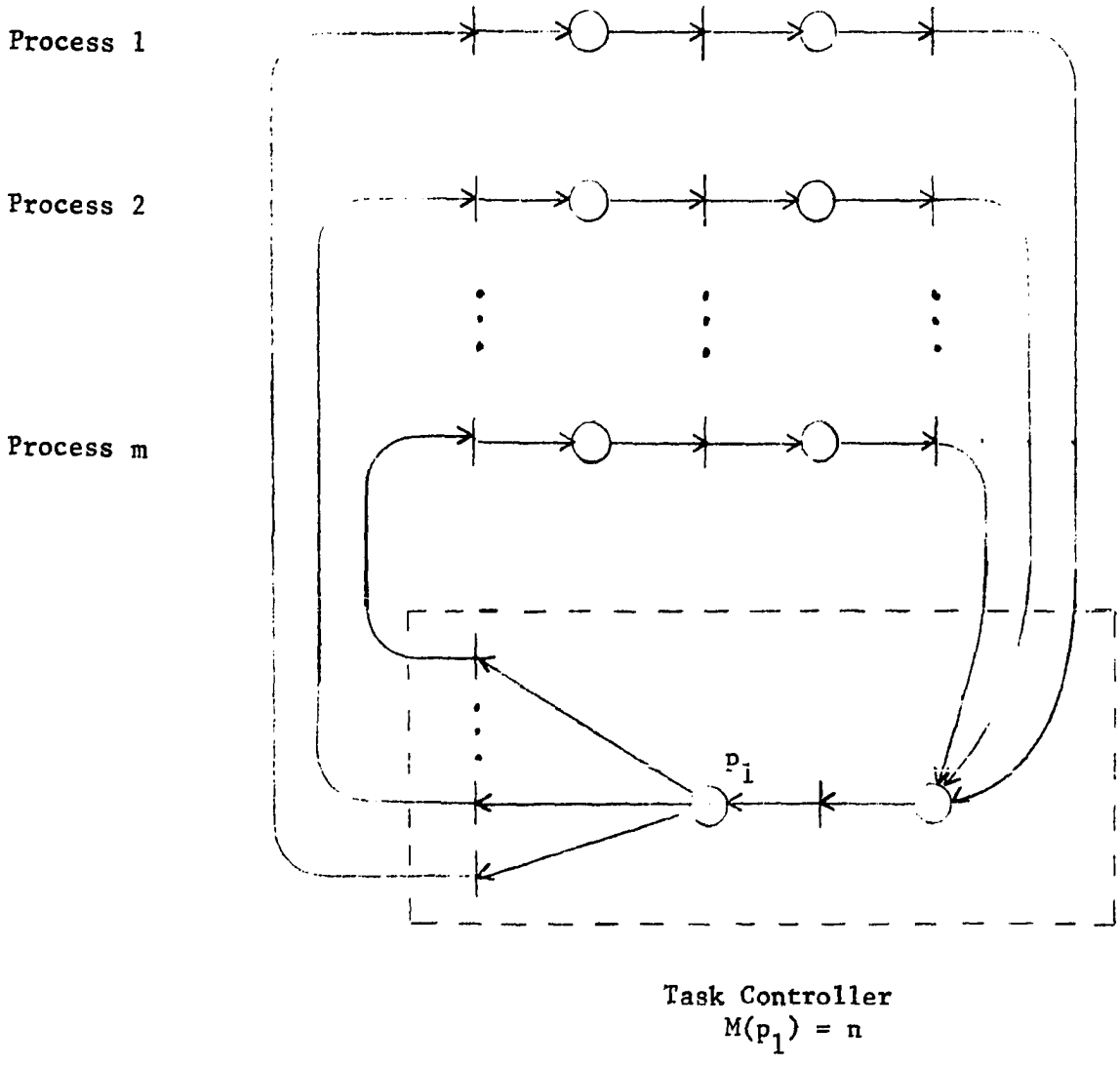


Figure 5.8. n^{th} order parallelism.

the available resources to the requesting processes.

If the initial marking has only a single token, X trivially corresponds to a 1^{st} order parallel net (the multiprogrammed uniprocessor system referred to earlier). However, if n tokens are included in the initial marking of the Task Controller, the net is said to have n^{th} order parallelism (or equivalently, parallelism of degree n). A comparison of the maximum computation rates gives

$$R_{1st\ order} = \frac{1}{\Sigma c_i \tau_i}$$

$$R_{n^{th}\ order} = \frac{n}{\Sigma c_i \tau_i} .$$

It is extremely important to realize that the assignment of n tokens ($n \geq 2$) to X has a far reaching impact. First, the Task Controller function is now complicated by the requirement of assigning a multiplicity of resources to ready-to-run processes. Secondly, the transitions in the net need to change as the processes must now be concerned with resource sharing and synchronization problems which did not exist in the first order parallel net.

The result of the additional processing requirements manifests itself in the form of longer firing times in the net and/or additional transitions. Thus, it can be seen that

$$\underset{\substack{n^{th}\ order \\ net}}{\Sigma c_i \tau_i} \geq \underset{\substack{1^{st}\ order \\ net}}{\Sigma c_i \tau_i} \quad \text{and, therefore,}$$

$$R_{n^{th}\ order} < n(R_{1st\ order}).$$

That is, the maximum fundamental computation rate for n^{th} order parallel system is a nonlinear function of the number of active resources. This concept is found to be true in practice. Enslow (19) supports the nonlinear nature of processing power vs. processors in his treatment of multiprocessor systems.

Example II - Bussing Networks

The functional decomposition of a problem statement into a multiplicity of centers of an MCS requires an interconnecting network of busses. Bus structures have a variety of characteristics. Thurber et al. (60) has categorized busses by the following general properties:

1.	generic type	dedicated nondedicated
2.	control type	centralized decentralized
3.	transfer philosophy	single word block (fixed or variable length) combination of single word and block.

Regardless of the combination of characteristics chosen for a given interconnection of system components, the resulting bus structure can be described as a finite state machine with an appropriate Timed Petri net representation. An example of a typical interconnection is shown in Figure 5.9. For the configuration given, the bus structure services one-way communication from two sources to one destination. Because of the sharing of the bus structure on the input side, it is clearly nondedicated. However, the method of operation for the bus (control type and transfer philosophy) may vary.

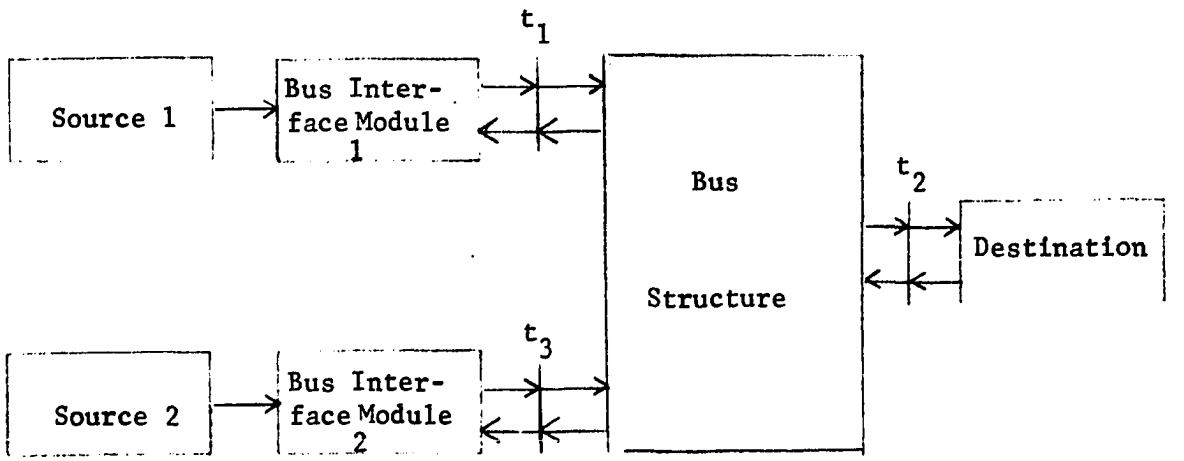


Figure 5.9. One-way nondedicated bus structure.

As a specific implementation, consider the Petri net B =<P, T, A> which represents a bus structure that transmits complete data packets (fixed length block transfer) using a centralized control technique (refer to Figure 5.10). The associated functional definition of the transition is given in Table 5.1.

Basically, transitions t_{11} and t_{12} assign the bus resource to a requesting bus interface module. Once the bus is assigned, the self-loops in the bus interface modules transfer a block of information across the bus one value at a time. At the completion of the transfer, the bus interface module executes a bus release and the interface module and bus structure return to their idle states pending the next bus activity.

In a decentralized control version of this example, the general structure is the same as that shown for the centralized control. However, in this case there is more logic involved in each bus interface module. The additional logic allows each interface module to determine the availability of the bus. This is in contrast to the logic in the bus proper making the same decisions for the centralized version of Figure 5.10.

The salient property of the bus structure is the decomposability of its Petri net representation into state machine components. The three state machine subnets, S_1, S_2 and S_3, of B (Figure 5.10) are illustrated in Figure 5.11. For S_1, S_2 and S_3 the maximum natural computation rates are

$$R_1 = \frac{1}{\sum_{S_1} c_i \tau_i}$$

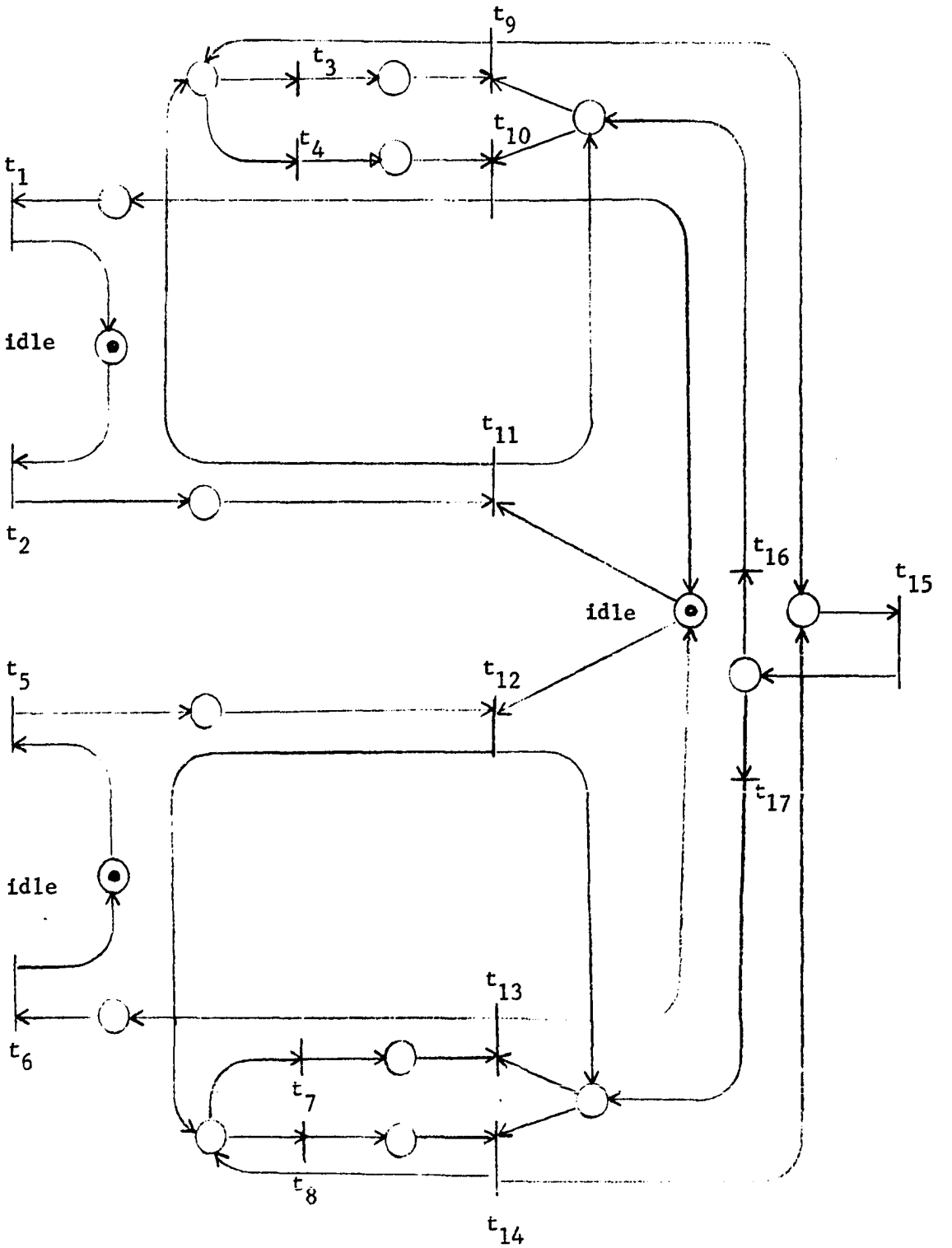
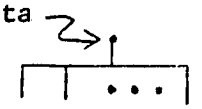


Figure 5.10. Petri net representation of centralized nondedicated bus structure.

Table 5.1. Bus structure functional definition.

Transition	Function	Comment
t_2, t_5	source_transfer_request(Data)	Data 
t_1, t_6	source_transfer_complete	
t_3, t_4; t_7, t_8	block_data_transfer_loop	i ← 1 while ¬ empty (Data) do value ← *Data . i delete (Data, i) i ← i + 1 od
t_9, t_{14}	in_bus_xfer(value)	$data value onto bus
t_{15}	out_bus_xfer(value)	$data value received
t_{10}, t_{13}	bus_release	
t_{11}, t_{12}	bus_request	
t_{16}, t_{17}	bus_transfer_continue	$continue transfer on selected source

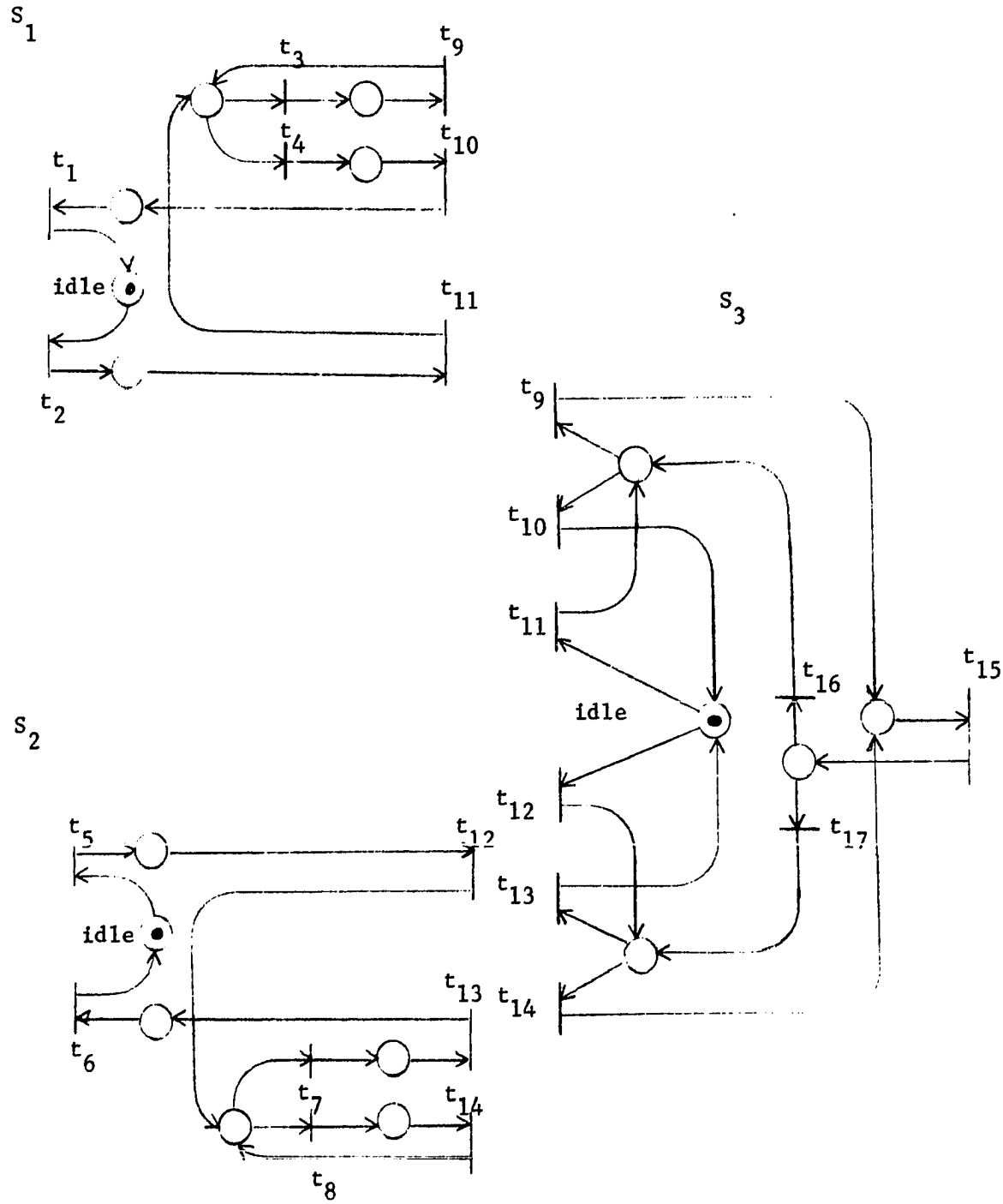


Figure 5.11. State machine decomposition of bus structure.

$$R_2 = \frac{1}{\sum_{S_2} c_i \tau_i}$$

$$R_3 = \frac{1}{\sum_{S_3} c_i \tau_i}$$

Intuitively, $W_{(t)}$ defines the bus "bandwidth" while $\underline{\Phi}(t)$ reflects the "loading" on the bus. In general, the firing times defined by W are small (large bandwidth). However, the currents defined by a particular $\underline{\Phi}$ may be large (heavy loading factor). In the examination of the feasibility of an MCS, the maximum computation rates of the proposed bus structure components must be taken into account. It may turn out that the "weakest link" or critical element in the system is an overloaded bus that is unable to handle the load imposed on it.

For the remainder of the chapter, it will be assumed that the computation rates of the bus components of a network are much larger than those associated with the functional system modules. That is,

$$R_{bus} \gg R_{sys} = \max \left[R_1, R_2, \ldots, R_m\right]$$

This restriction simplifies the analysis process. In particular, it allows for a concentration on the strictly functional aspects of the problem being considered.

However, it is important to realize that in a practical design such an assumption can not be usually made. Specifically, a good deal of effort should be invested in the design of the interconnecting network (60).

Example III - Condensed RTDAD System

A unification of the material introduced in the preceding chapter is in order. Let us consider a condensed RTDAD specification typifying the general nature of the RTDAD problem. The restricted version which has been taken from an existing system will be referred to as the condensed RTDAD system or CRS.

Specifically, the restricted problem statement has been chosen in such a manner as to highlight the general techniques involved while attempting to suppress irrelevant and time consuming details. A number of assumptions have been made which are explicitly stated as part of the CRS problem statement.

Generally, the algorithms given here depend upon high level functional definitions. These procedural definitions would normally be expanded in an actual design to a level of detail defining explicitly the semantics of the operations involved. However, for our purposes, it is sufficient to confine our attention to specifications at a more intuitively appealing level of presentation.

The purpose of the analysis of the CRS is fourfold:

1. demonstrate the structural and functional representation of the CRS using the System State Model and Timed Petri nets,
2. examine the feasibility of the models in the light of the results derived from Timed Petri nets,
3. show the existence of one possible MCS decomposition of the CRS, and

4. summarize the advantages inherent in the MCS decomposition.

CRS problem statement

A pictorial overview of the input and output environment is given in Figure 5.12a. Input packets originating at the input transducer are processed by the CRS and the resultant output is displayed on one of two identical display devices.

Input packets Data packets are input to the CRS at a constant rate of one frame per 50 msec[1]. The general packet format is given in Figure 5.12b. The elementary values associated with the selectors shown are:

1. DT (data type) - identification of the type of data contained in the packet,
2. Time (GMT) - origination time of the data packet,
3. SN (sequence number) - integer count denoting the sequencing of consecutive frames, and
4. Data (data content) - actual information content as sampled by the input transducer.

The input transducer produces three categories of data:

1. multiframe event data (MF),
2. analog data (ANLG), and
3. configuration data (CONFG).

---

[1] Input is via synchronous devices which are continually clocked regardless of the presence or absence of bona fide data. Empty packets contain a null data identifier.

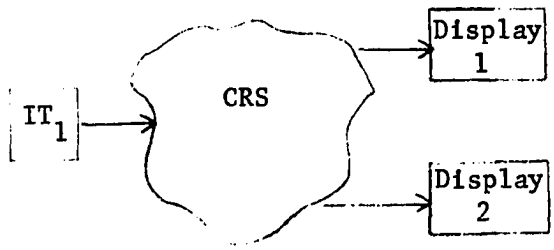
The exact frame formats and repetition rates for these data types is shown in Table 5.2. Each MF event requires the transmission of five unique frames.

The frames are assumed to be received in order with sequence numbers one to five (SN = 1, 2, 3, 4, 5). An additional piece of information, the integer event number (EN), distinguishes unique events for each data type. That is, there is a distinct counter for each MF data type that is incremented by one for each successive event transmission.

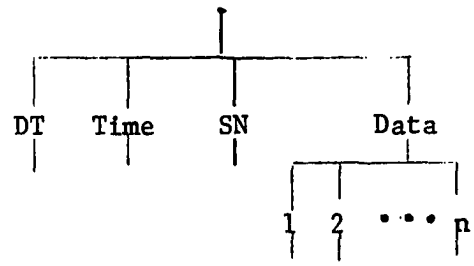
Analog frames contain data points that are digitized analog monitors representing values such as voltages, temperatures, pressures, etc. Every analog input frame contains a complete collection of forty monitor values as sampled by the input transducer. The sequence number, SN, is an integer counter that distinguishes consecutive frames in the range $1 \leq SN \leq 3600$.

Configuration frames carry strictly digital data. The data points represent discrete system conditions such as power on/off, equipment redundancy configurations, switch closures, etc. Each configuration frame contains thirty sample values. The sequence number, SN, is an interger counter that distinguishes consecutive frames in the range $1 \leq SN \leq 3600$.

Processing/display requirements - The processing requirements are different for each type of data. One similarity, however, among all three types is the temporary storage of data in a common data base. It is assumed that the MF, CONFG, and ANLG data to be displayed are first appended to the appropriate file structure component. Although it does



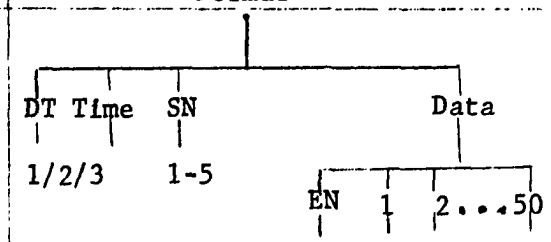
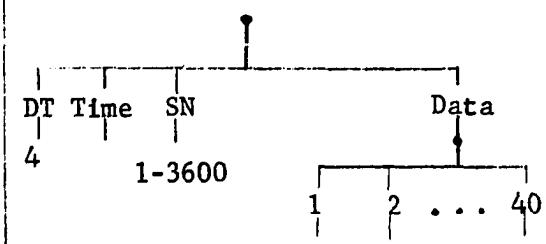
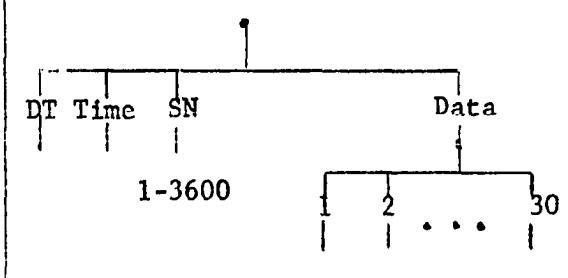
a. CRS overview.



b.. Input packet format

Figure 5.12. CRS overview.

Table 5.2. CRS data frames.

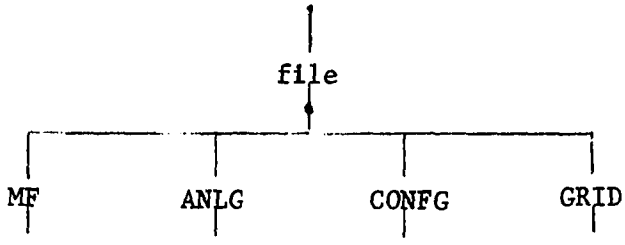
Data type	Repition rate	Frames/trans	Format
1. MF DT_1,DT_2, DT_3	Random	5	
2. ANLG DT_4	1 frame/250 ms.	1	
3. CONFG DT_5	1 frame/250 ms.	1	

not appear explicitly in the CRS, the motivation for storing the data is for the purpose of data continuity. If the system fails, the operating state prior to the failure can be partially recreated (assuming that the file structure remains intact). Additionally, there is a requirement that any data which is displayed automatically (the only mode available in the CRS) can also be redisplayed on request from a system keyboard (available in the original system).

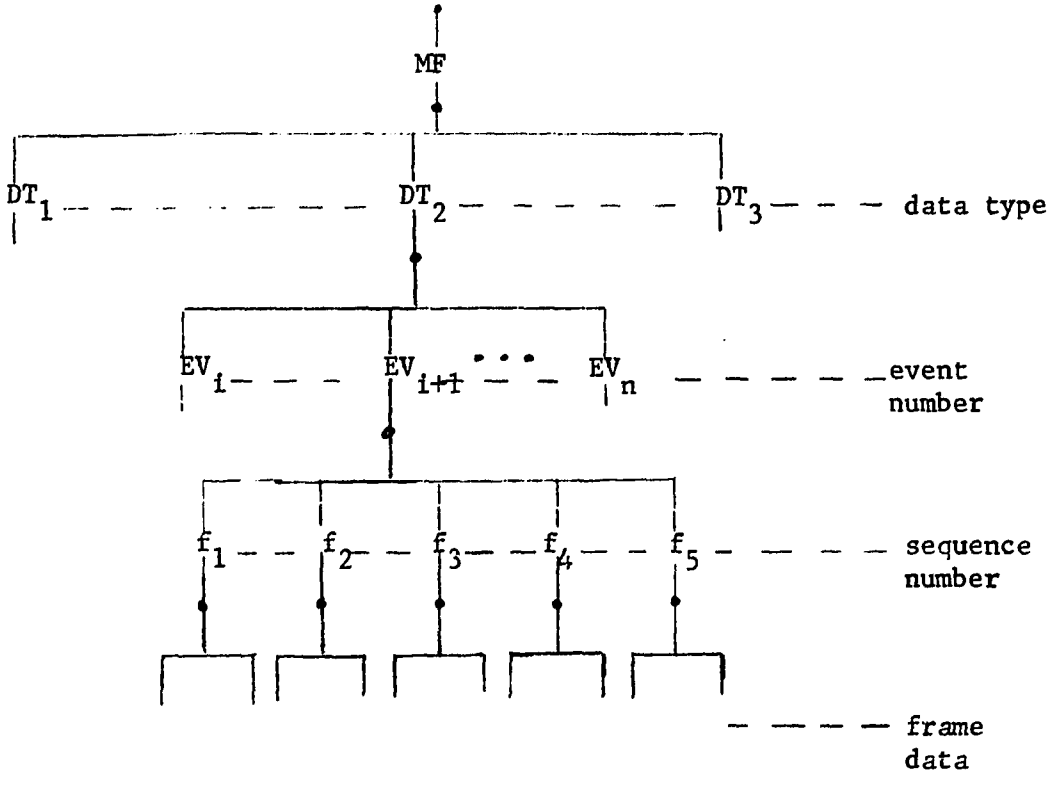
The file structure is shown in Figure 5.13. There are four main components to the file system; three components correspond to data taken from the three data types in the input stream. The fourth component contains static display information. Each of the five unique displays (3MF, 1 CONFG, 1 ANLG) require a moderate amount of annotation that does not change from one display to the next. This static information is called the grid data. There are grid data objects for each of the three major data types.

The requirement for the processing of the dynamic file information is as follows. All MF data packets are retained according to their associated event numbers. Each MF event has five input packets which are collected in a common area on the file component. Once all five packets have been received, a display of the composite event is generated. MF events occur randomly, i.e., there is no a priori knowledge as to the time of occurrence of MF events. However, a maximum number of MF events per time period can be specified.

Analog data is received every 250 msec. Each monitor point in every frame is compared with a predefined upper and lower critical limit (see



a. major file structure component



b. MF component

Figure 5.13. File structure.

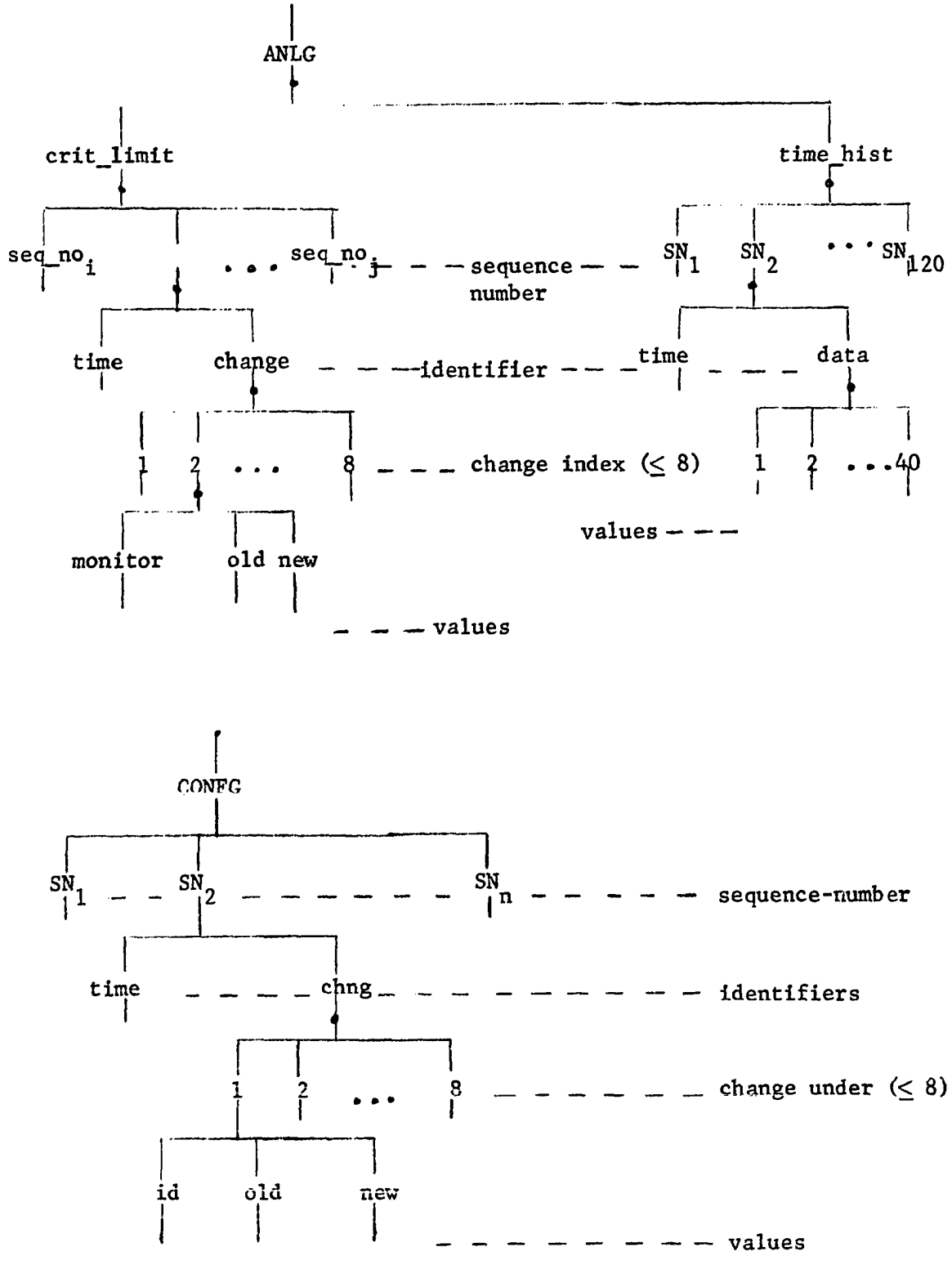


Figure 5.13. Continued.

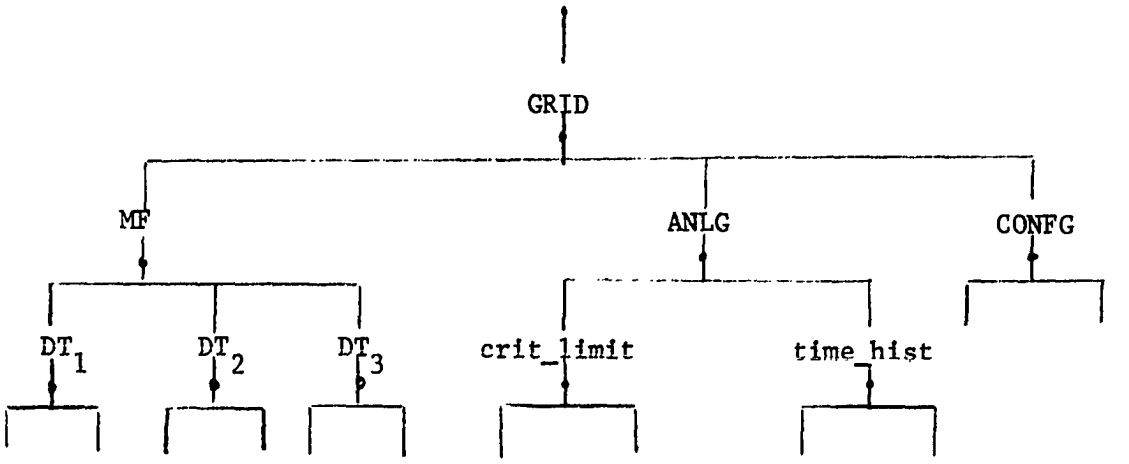


Figure 5.13. Continued.

Figure 5.14a). If the monitor value lies outside the range bounded by these limits an out-of-limits analog display is generated.

Additionally, one entire set of forty monitor values (one input packet) is retained from each set of thirty consecutive packets. In particular, the packets whose sequence numbers are integral multiples of thirty are saved. Every fifteen minutes (SN = 3600) a display of all forty monitors is output. It is assumed that the file component for analog time history data (file.'ANLG'.'time-hist')is capable of holding only 120 unique frames[1]. The problem of new frames overwriting old frames prior to a display being output is ignored[2].

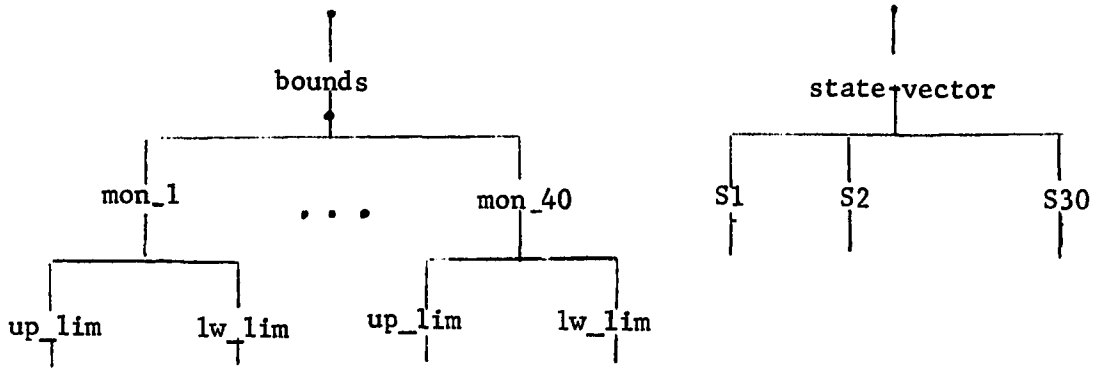
Finally, configuration data is handled in a manner somewhat similar to analog limit checks. The thirty digital sample points are compared bit by bit against the most recently received 30 samples (see Figure 5.14b). Whenever a change occurs, the digital sample identifier (id), the old value, and the new value are to be displayed.

It is assumed for both analog limit checks and configuration comparisons that a maximum of eight samples can change in any one frame. If more than eight changes are found, the comparison algorithms assumes a fault in the input data and disregard the entire frame.

---

[1] 120 samples = (15 min·60,000 msec/min)/(250 msec/frame·30 frames/samples).

[2] This problem can be solved in a number of ways. For example, two identical tim-hist objects can be incorporated into the file; one may be used for display while the second is recording new data.



a. Critical limit bounds

b. Configuration comparison base

Figure 5.14. Comparison base for analog and configuration frames.

Table 5.3. CRS data set.

Data Type	Number of Frames in T_{DS}	Processing Required	Number of disp in T_{DS}
MF	50	10 events	10
ANLG	3600	5 out-of-limit 1 time history	5 1
CONFG	3600	10 bit changes	10
Totals	7250 input frames		26 displays
	T_{DS} = 15 minutes		

The display devices have a special hard copy feature. If enabled, every completed display is automatically printed on the hard copy device. The copying device requires ten seconds for one cycle of operation. Therefore, at the completion of each display (one grid plus one set of data values) a ten second delay/copy command is issued to the display device.

Data set The data set has been defined earlier as a benchmark for measuring system performance. For the CRS example, only one data set is considered. That data set, shown in Table 5.3, precisely defines the current function $\underline{\Phi}$ for the associated system Timed Petri net. The boundary value constraints stipulate that the processing rates of the system components must allow all 7250 input packets and 26 displays to be handled in any given 15 minute period (T_{DS} = 15 minutes). Furthermore, if an arbitrary number of such periods should occur consecutively, the system must perform identically[1] within each such period.

Additional data sets are definable for the CRS example. Each such data set produces a different current assignment $\underline{\Phi}$ and, therefore, different sets of natural computation rates for the state machine components of the CRS Petri net. In all cases, the boundary value constraint equation discussed previously must be satisfied. For our purposes only one data set is treated. The singular data set chosen is sufficient for the demonstration of the basic concepts that need to be explored.

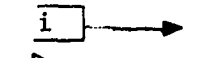
---

[1] The performance is identical except for the randomness inherent in the input data set.

CRS structure

Figure 5.15 and Table 5.4 present a complete description of the Timed Petri net representation of the CRS. Based on the data set of Table 5.3, a current assignment $\underline{\Phi}$ is included in Table 5.4. The following list of comments should sufficiently clarify the Timed Petri net structure so that a general understanding can be gleaned from this material.

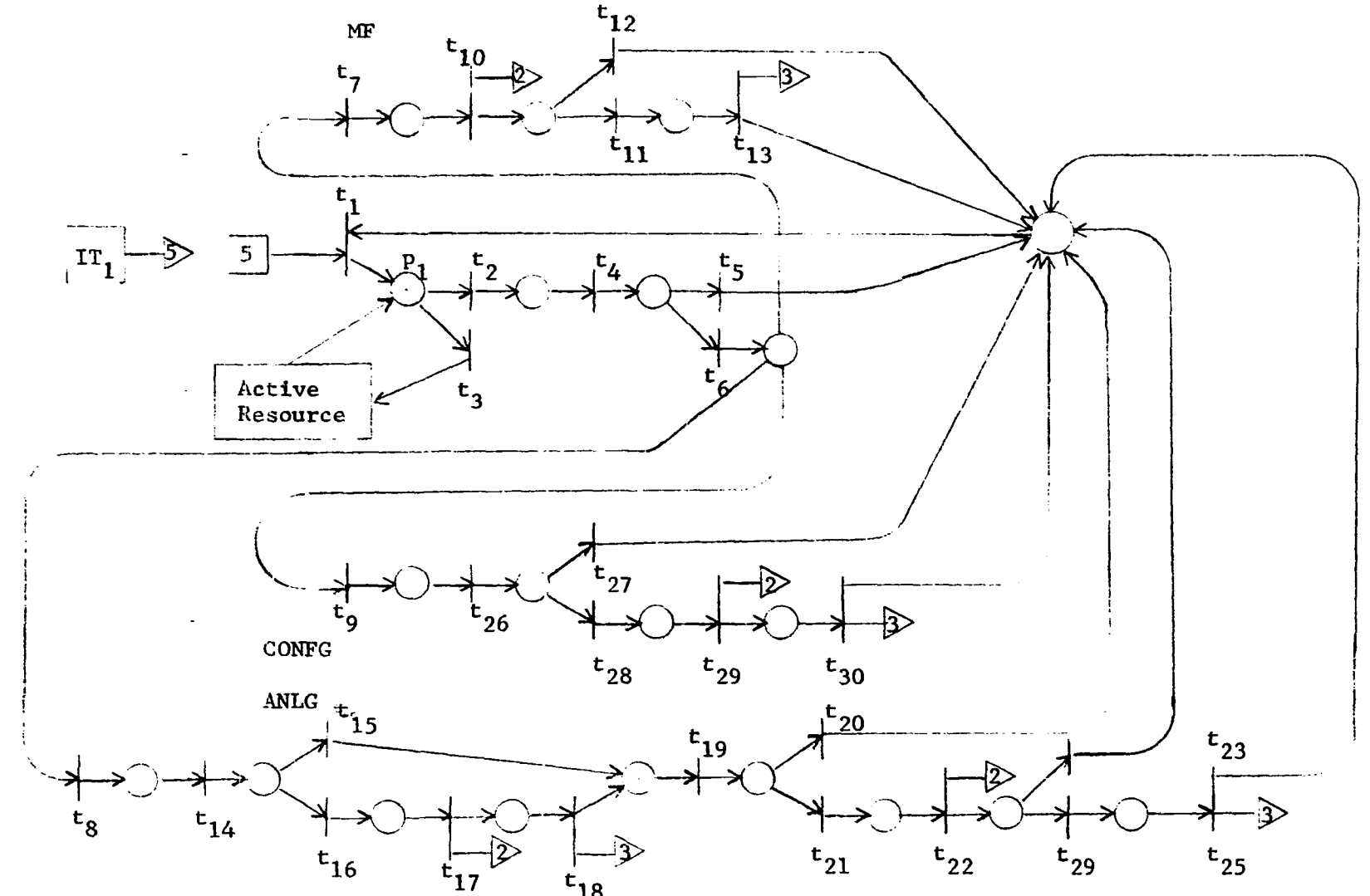
1. Data connectivity between processes does not appear explicitly in the Petri net representation. For completeness, special graphical devices are included to indicate the presence of queues and to mark the functions which insert entries into these queues. The notation is as follows:

   a. queue  (i^{th} queue)

   b. queue insertion  (insert entry into the i^{th} queue)

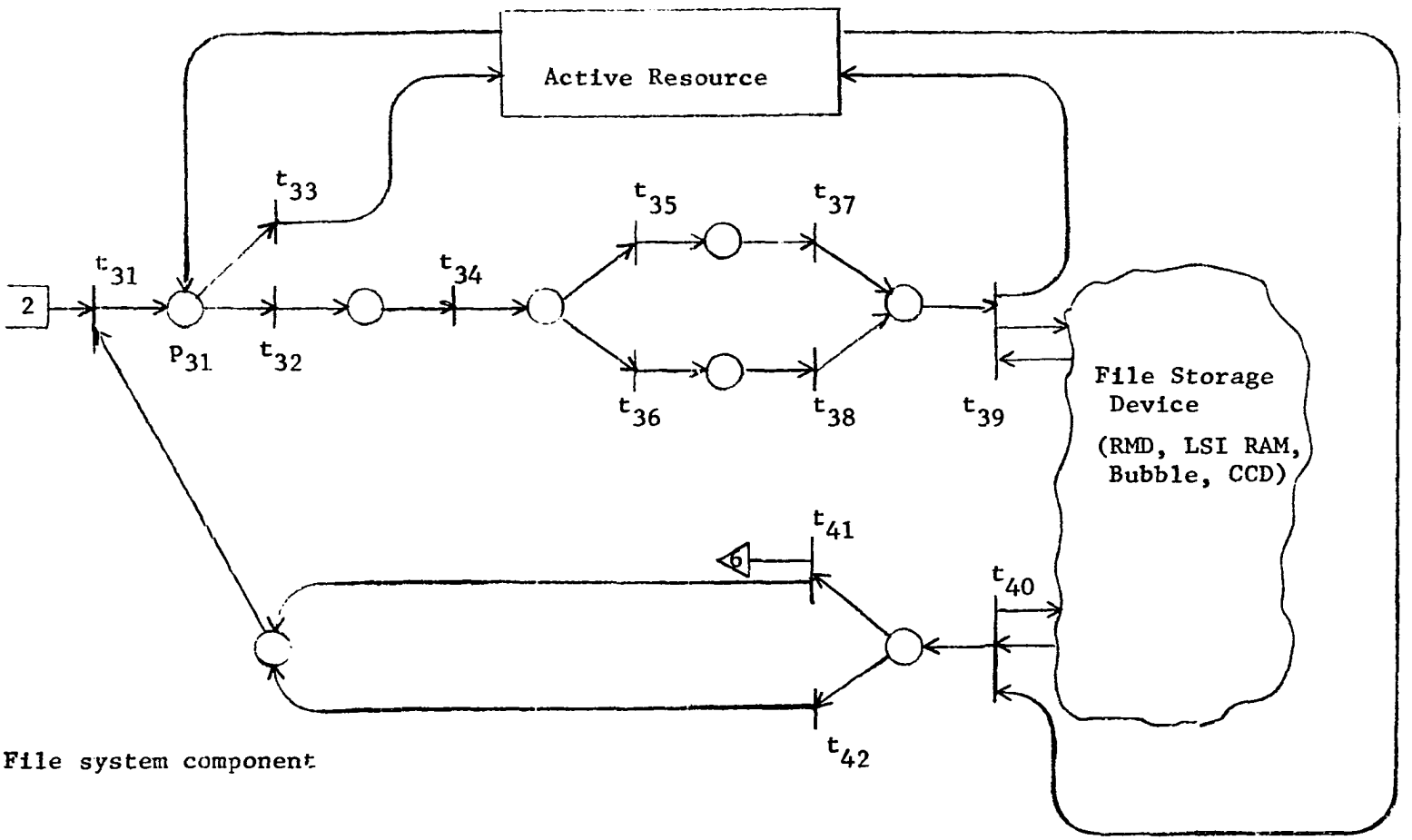
   These symbols convey no importance in the structure of the Timed Petri net. They are used only as an aid in the clarification of the overall operation of the net.

2. The input and output currents at the wait operations on input queues (t_1, t_2, t_{44}, t_{60}) have not been assigned numerical values. These currents depend upon the probability that the queue is found not empty at the time the present operation on an entry from the queue is completed. In general, the analysis necessary to determine the queue lengths depends upon the timing of the remainder of the net. A worst case assumption for the assignment of these currents is that the queues are empty at the time of the check. As a result the currents are identical for all four



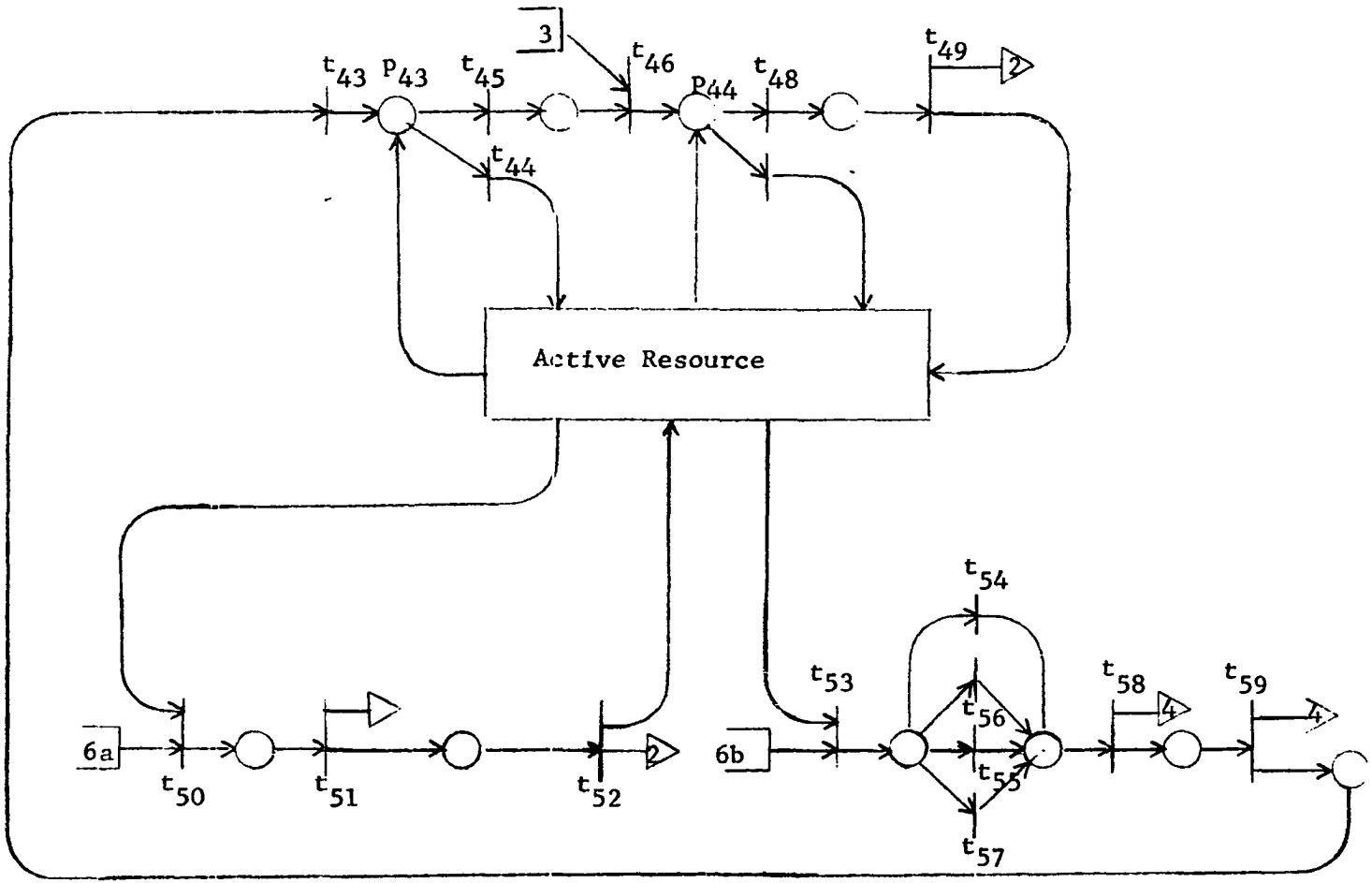
a. primary data handling

Figure 5.15. Timed Petri net for CRS.



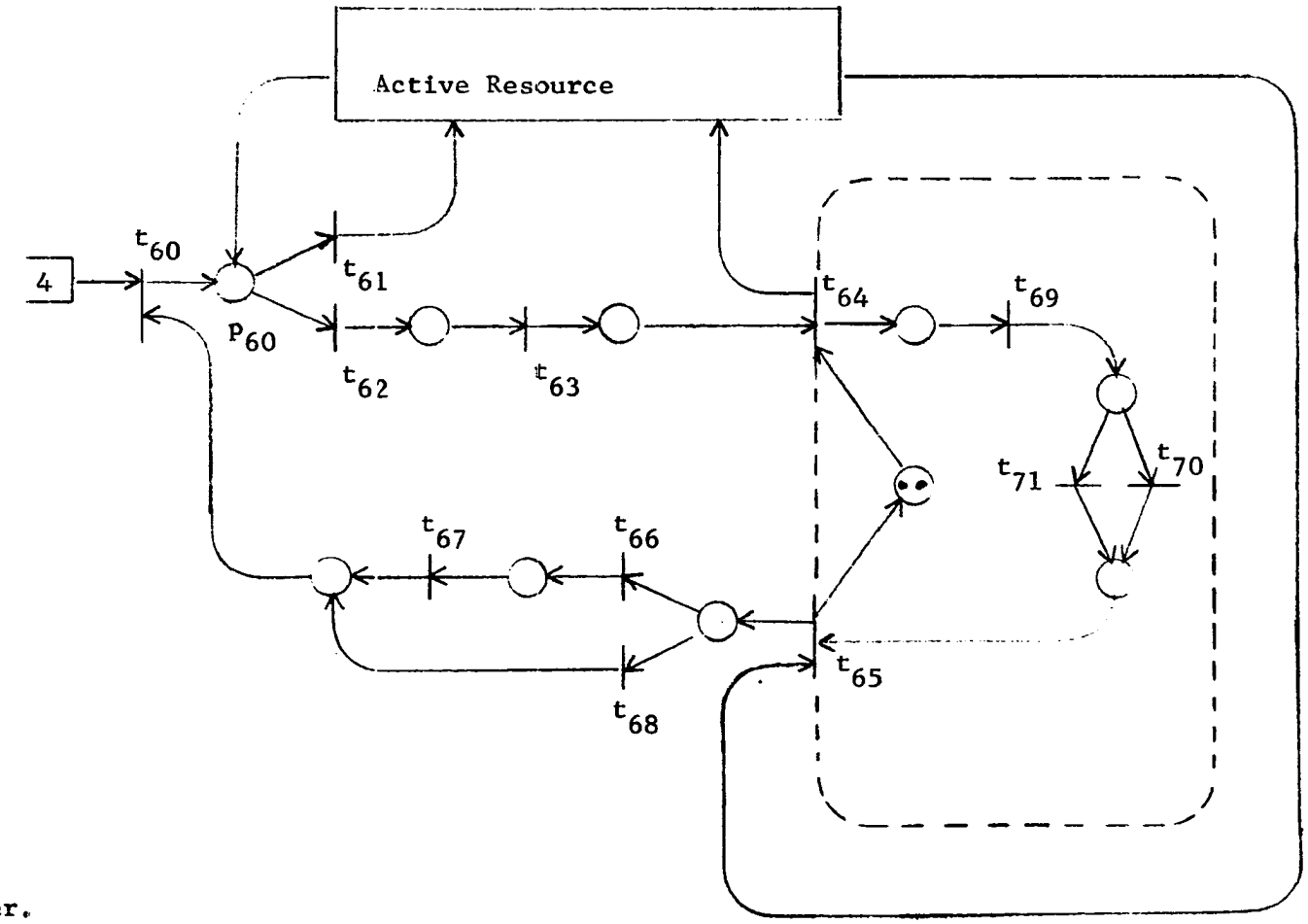
b. File system component

Figure 5.15. Continued.



c. Display handler.

Figure 5.15. Continued.



d. Display driver.

Figure 5.15. Continued.

Table 5.4. CRS functional description.

Transition	Current	Function	Comment
		INPUT	
t_1	18000	wait (IT_input)	
t_2	18000	input_queue(IT) ¬= empty remove_queue (P)	See Table 5.2
t_3	X_3	input_queue(IT) = empty	
t_4	18000	data_type ← *P.'DT'	
t_5	10750	data_type = null	
t_6	7250	data_type = MF | ANLG | CONFG	
t_7	50	data_type = MF	
t_8	3600	data_type = ANLG	
t_9	3600	data_type = CONFG	
		MF	
t_{10}	50	seq_no ← *P.SN write_file(F1) event_num ← *P.'Data'. 'EN'	See Figure 5.16a
t_{11}	10	SN = 5	
t_{12}	40	SN ¬ = 5	
t_{13}	10	display (D1)	See Figure 5.16b
		ANLG	
t_{14}	3600	sn ← P.'SN' compare(P.'data', crit_lim, out_of_limit, lim_chng)	See Figure 5.14a and Figure 5.16c
t_{15}	3595	out_of_limit = false	

Table 5.4 Continued.

Transition	Current	Function	Comment
t_{16}	5	out_of_limit = true	
t_{17}	5	write_file(F_2)	See Figure 5.16d
t_{18}	5	display (D2)	See Figure 5.16e
t_{19}	3600	num ⟵ sn mod 30	
t_{20}	3580	num ¬ = 0	
t_{21}	120	num = 0	
t_{22}	120	write_file (F3)	See Figure 5.16f
t_{23}	119	sn ¬ = 3600	
t_{24}	1	sn = 3600	
t_{25}	1	display (D3)	See Figure 5.16g
		CONFG	
t_{26}	3600	sn ⟵ P.'SN' compare(P.'data', state_vector,change, config_chng)	 See Figure 5.14b See Figure 5.16h
t_{27}	3590	change = false	
t_{28}	10	change = true	
t_{29}	10	write_file (F4)	See Figure 5.16i
t_{30}	10	display (D4)	See Figure 5.16j
		File system component	
t_{31}	237	wait(file_request)	
t_{32}	237	input_queue(file)¬= empty	
t_{33}	X33	input_queue(file)= empty	

Table 5.4. Continued.

Transition	Current	Function	Comment
t_{34}	237	remove_queue(F) op_code = *F.'op'	
t_{35}	52	op_code = read	
t_{36}	185	op_code = write	
t_{37}	52	pointer ← * F. 'file_loc" cmnd ← build_command (read, pointer)	
t_{38}	185	pointer ← *F. 'file_loc' cmnd ← build_command (write, pointer)	
t_{39}	237	command_file_device(cmnd)	
t_{40}	237	wait (device_complete)	
t_{41}	5	op_code = read return_data(F.'source', Rtn)	See Figure 5.16k
t_{42}	185	op_code = write	
		Display handler	
t_{43}	26	wait (avail_display_ device)	
t_{44}	X_{44}	display_device_queue (avail_device) = empty	
t_{45}	26	display_device_queue (avail_device) ¬ = empty device_num ←avail_device	
t_{46}	26	wait (input_request)	
t_{47}	X_{47}	input_queue(display_ request) = empty	

Table 5.4. Continued.

Transition	Current	Function	Comment
t_{48}	26	input_queue(display_ request) ¬ = empty	
t_{49}	26	remove_queue(D) sel ⟵ grid_selector (D.'file_loc') read_file(F5.)	See Figure 5.161
t_{50}	26	wait (input_file_data)	
t_{51}	26	write_display (Grid)	See Figure 5.16m
t_{52}	26	sel ⟵ D.'file_loc' read_file (F6)	See Figure 5.16n
t_{53}	26	wait(input_file_data) type ⟵ *D.'source'	
t_{54}	10	type = MF convert_MF_to_display_ format(Dsp)	
t_{55}	5	type = ANLG.'crit_lim' convert_crit_lim_to_ display_format(Dsp)	
t_{56}	1	type = ANLG.'time_hist' convert_time_hist_to_ display_format(Dsp)	
t_{57}	10	type = CONFG convert_config_to_ display_format(Dsp)	
t_{58}	26	write_display (Dsp)	See Figure 5.160
t_{59}	26	write_display (Delay)	See Figure 5.16p
		Display	
t_{60}	78	wait(display_cmnd)	
t_{61}	60	input_queue(display_ driver) = empty	

Table 5.4. Continued

Transition	Current	Function	Comment
t_{62}	78	input_queue(display_driver) ¬= empty remove_queue(Display	
t_{63}	78	cmnd ← generate_display_command(display)	
t_{64}	78	command_display_device(cmnd)	See Figure 5.16q
t_{65}	78	wait(display_complete) op_code ← *Display.'op'	
t_{66}	26	op_code = delay	
t_{67}	26	insert(display_device_queue,*Display.'dev_num') signal(avail_display_device)	
t_{68}	52	op_code = write	
t_{69}	78	input_data	
t_{70}	52	write_data	
t_{71}	26	delay	

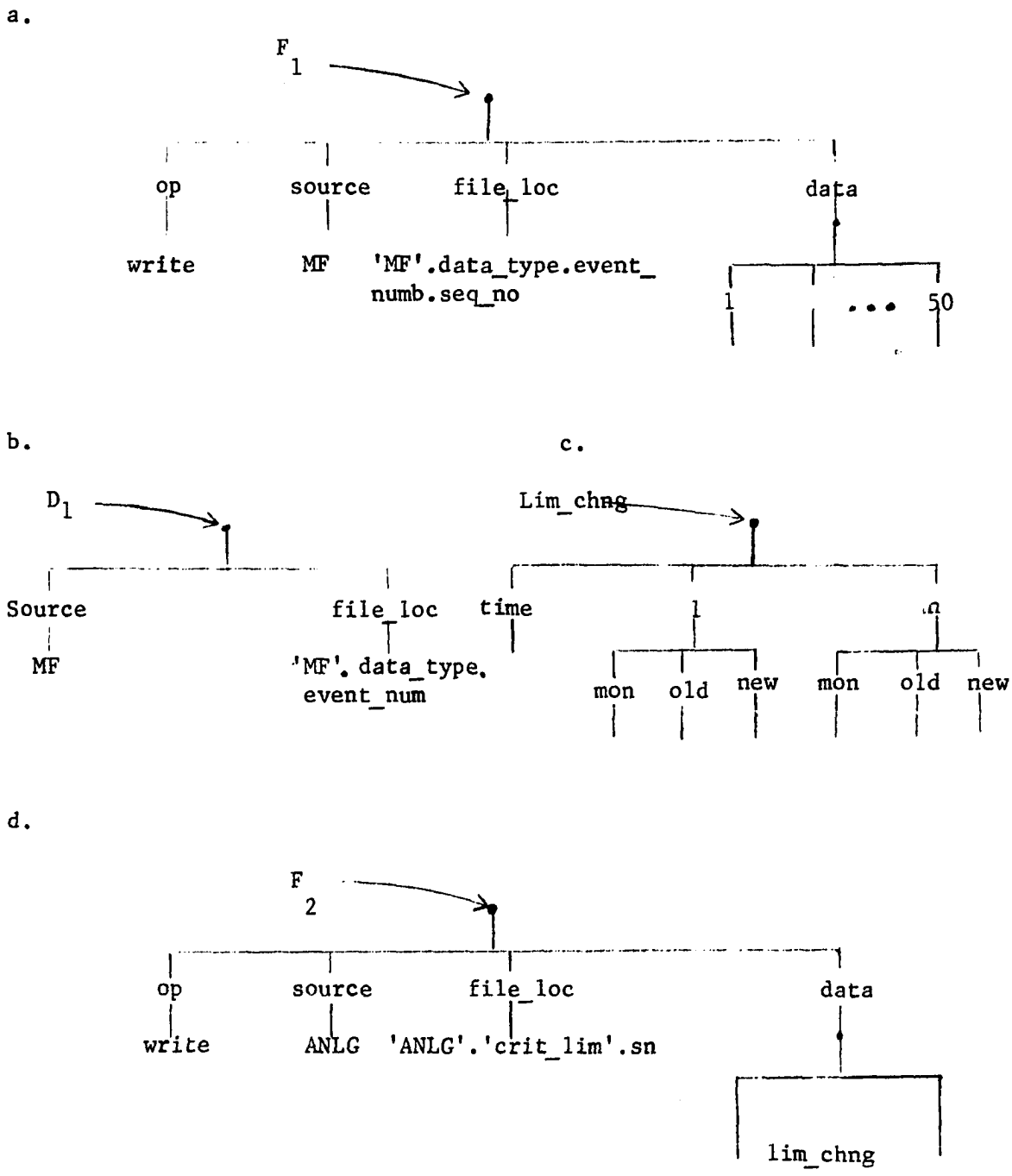
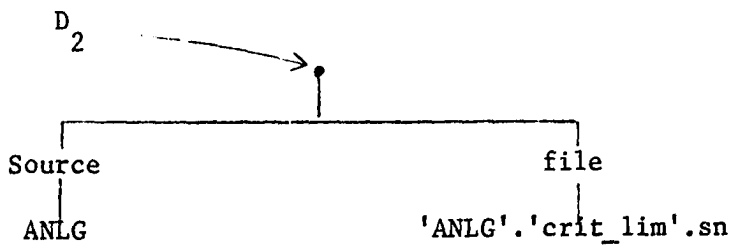
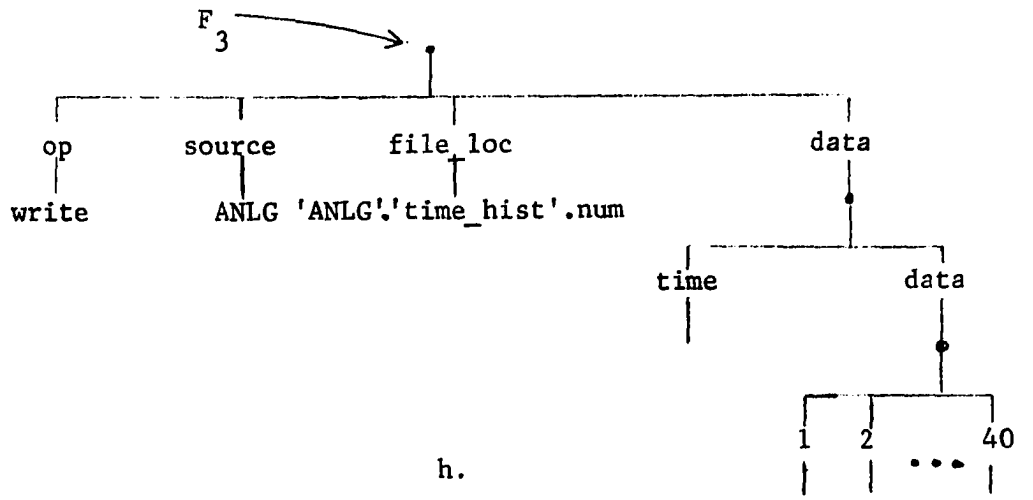


Figure 5.16. Argument data objects.

e.



f.



g.

h.

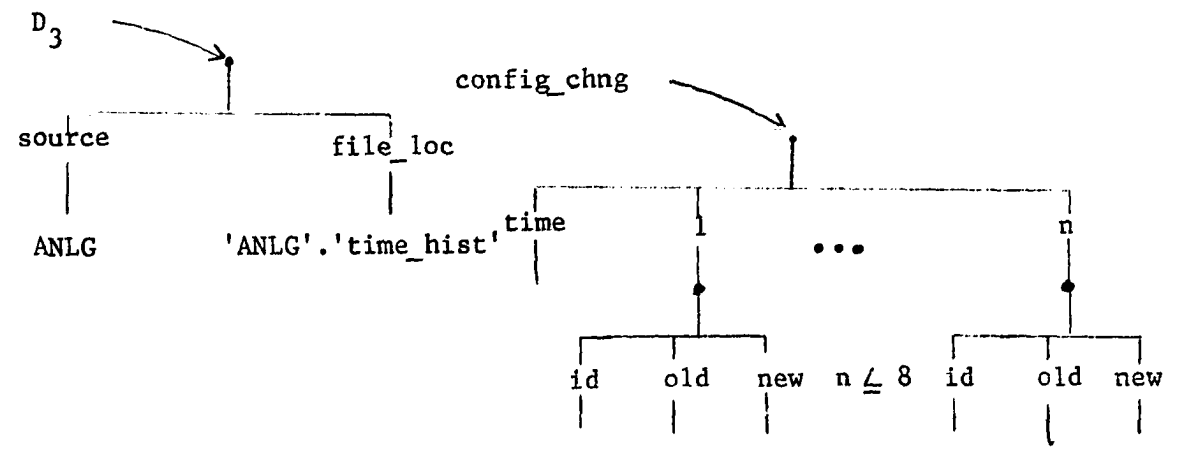


Figure 5.16. Continued.

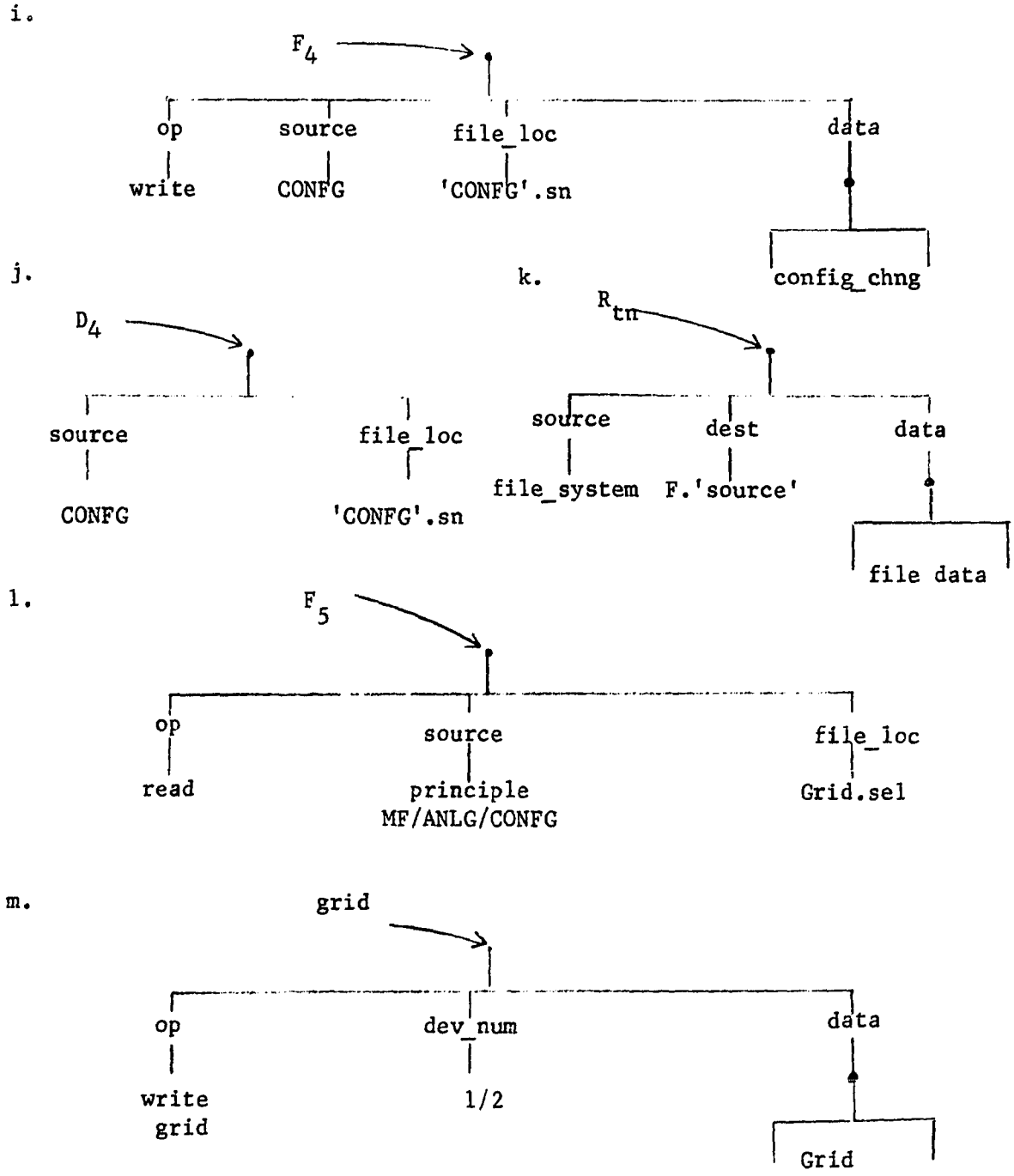
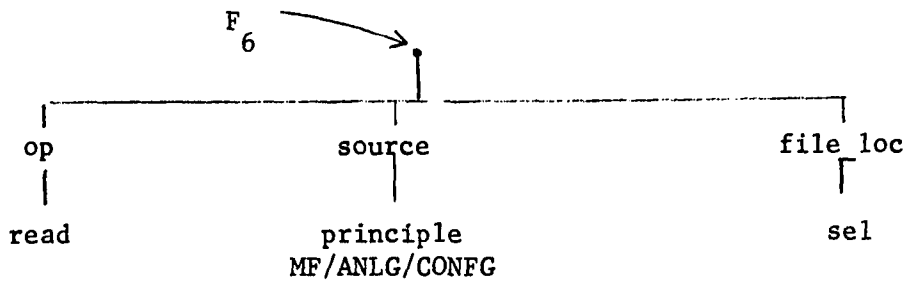
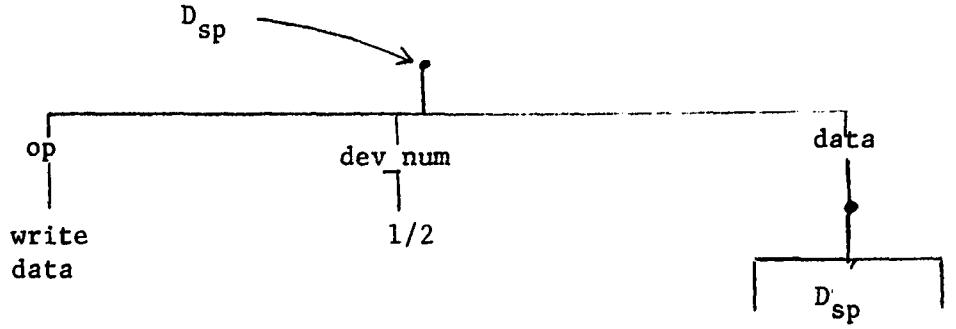


Figure 5.16. Continued.

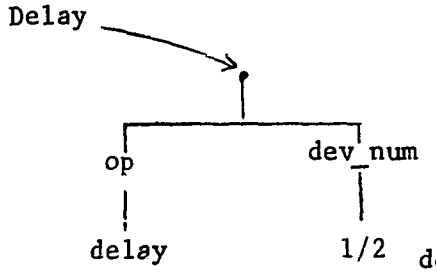
n.



o.



p.



q.

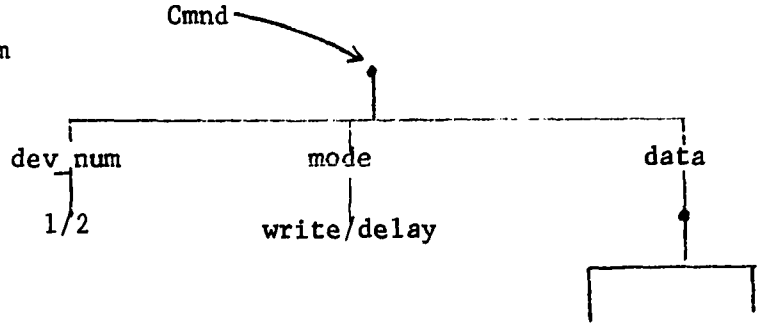


Figure 5.16. Continued.

arcs about the wait instances p_1, p_3, p_{44} and p_{60} (two arcs in and two arcs out).

3. At this point in the example, the nature of the active resource(s) is not specified. If the active resource is a singular processing element for all the net parts, then the system would be an SCS. In that case, the CRS would be of the form of a typical real-time system like the one used to solve the original expanded problem statement. The application of a multiplicity of resources will be treated shortly.

4. The precise implementation of the file system has not been specified. This choice depends upon a number of parameters, namely:
   a. storage space requirements,
   b. access/latency time bounds, and
   c. device cost ceilings.
   It is assumed, however, that there is sufficient random access memory space available to retain the directory information portrayed by the selector structure of Figure 5.13. This allows for the resolution of file storage addresses without recourse to the file storage device.

CRS analysis

From an inspection of the Petri net of Figure 5.15, we can see that the CRS and the I/O devices can be conceptualized as a number of state machine components. There are state machines representing the input transducer, the output display devices, the file storage device, and one

or more state machines for the decomposition of the CRS. The number of components in the state machine decomposition of the CRS is a function of the assignment of the active resources depicted by the resource boxes in Figure 5.15. The following material examines a numerical treatment of the decomposition of the CRS into state machine components.

From the input specification and current assignments the following information can be derived directly:

1. $R_{IT} = \frac{1}{\sum_{IT} c_i \tau_i} \doteq \frac{1}{(18000)(50\text{msec})} = 1/(900 \text{ sec})$

2. $R_{display} = \frac{1}{\sum_{display} c_i \tau_i} \doteq \frac{2}{260 \text{ sec}} = 1/(130 \text{ sec})$

One part of the boundary constraint equation is, therefore, satisfied, i.e.,

$$R_{IT} \leq R_{display}.$$

For notational convenience, let I_i $(1 \leq i \leq 7)$ be the sets of integers given by

$I_1 = (i \mid 1 \leq i \leq 9)$ $\qquad$ $I_5 = (i \mid 31 \leq i \leq 42)$

$I_2 = (i \mid 10 \leq i \leq 13)$ $\qquad$ $I_6 = (i \mid 43 \leq i \leq 54)$

$I_3 = (i \mid 14 \leq i \leq 25)$ $\qquad$ $I_7 = (i \mid 60 \leq i \leq 66)$

$I_4 = (i \mid 26 \leq i \leq 30)$

Additionally, let S_i be the current-time product for integer set I_i. That is,

$$S_i = \sum_{I_i} c_j \tau_j$$

where $c_j = \Phi(t_j)$, and $\tau_j = W(t_j)$.

Next, let PM be a complete set of primitive modules including processing elements, memory elements, and bus and bus interface modules[1]. Let us assume that the values of the current-time products based on the given PM can be computed to be

$S_1 = 25$ sec $\qquad S_5 = 100$ sec

$S_2 = 50$ sec $\qquad S_6 = 200$ sec

$S_3 = 100$ sec $\qquad S_7 = 50$ sec

$S_4 = 75$ sec

$$\sum_{i=1}^{7} S_i = 600 \text{ sec}$$

If the CRS is to be used as an SCS system with a multiprogrammed uniprocessor, the active resource boxes of Figure 5.15 are coalesced into one resource as shown in Figure 5.17. In Figure 5.17, the current-time product S_8 depicts a Job Controller (JC) which is responsible for allocating the single processor resource to the various requesting processes. If t_{JC} is assumed to have a firing time given by

$$\tau_{JC} = W(t_{JC}) = 2 \text{ msec} \qquad \text{(based on PM)}$$

then

$$S_8 = c_{JC}\tau_{JC} = 37 \text{ sec.}$$

---

[1] PM is complete in the sense that all functions of the Timed Petri net can be implemented in that set of primitive modules. For example, one possible set of primitive modules might be a minicomputer central processing unit, main core memory, data channels, and a rotating memory device. An alternate set might include microprocessors, an assortment of LSI RAMs, ROMs, and CAMs, etc.

Therefore, the maximum fundamental computation rate for the CRS can be found to be

$$R_{CRS} = \frac{1}{\sum_{i=1}^{8} S_i} = \frac{1}{600+37} = 1/637 \text{ sec.}$$

The second part of the constraint equation is now satisfied, i.e.,

$$R_{IT} \leq R_{CRS}.$$

However, consider the results of a change in the problem statement. Specifically, let us examine the impact on the multiprogrammed uniprocessor if a second identical input transducer is added to the system. The Petri net representation of the change in the input environment is shown in Figure 5.18.

The computation rate of the input is the same as before. However, the effect on the CRS has a threefold impact:

1. the current assignment for all the transitions essentially doubles,
2. a number of data objects (e.g., file system components) must be expanded to include the additional data base, and
3. additional transitions are required for functions that provide synchronization and distinguish between the data objects for the two unique sources.

Thus, an optimistic measure of performance is obtained if we assume only a doubling of the value in the current assignment. For such an assumption the new maximum computation rate for the CRS is given by

$$R_{CRS} = \frac{1}{2\sum_{i=1}^{8} S_i} = \frac{1}{1274} \text{ sec.}$$

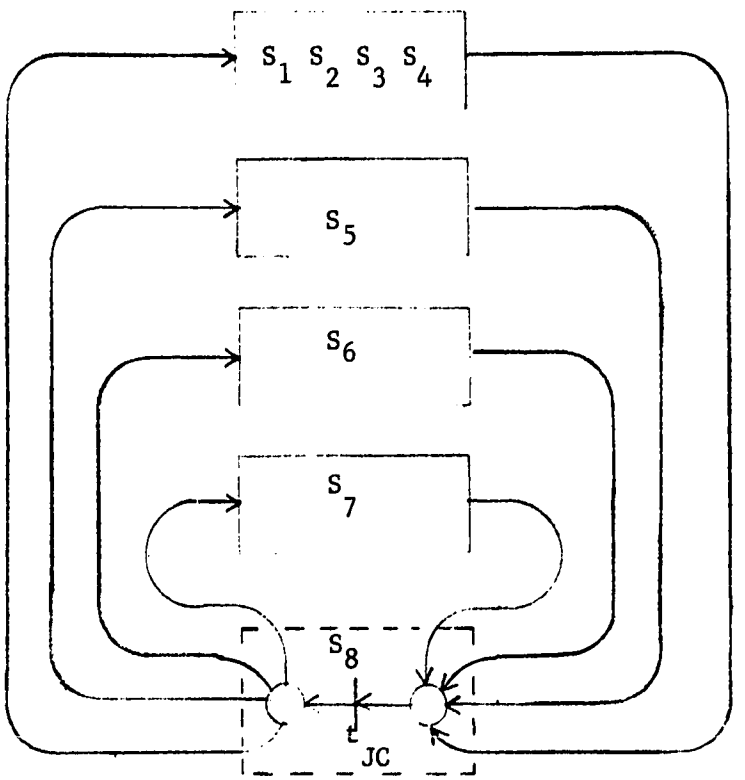
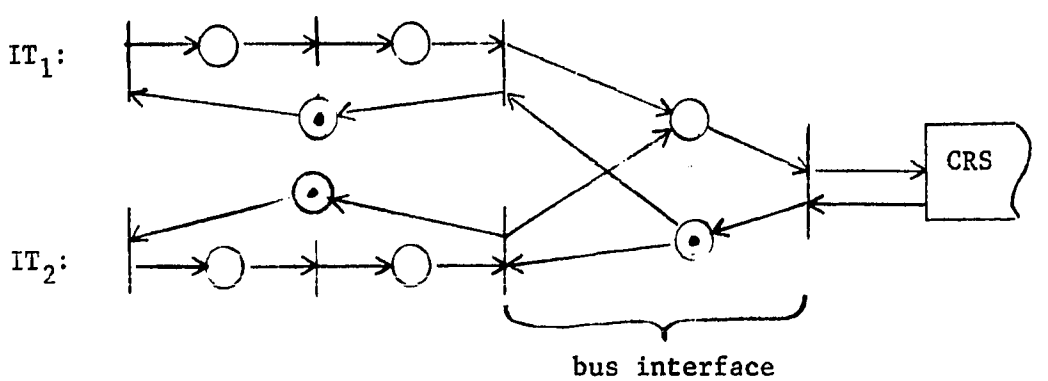


Figure 5.17. Multiprogrammed uniprocessing CRS.



$$R_{IT_1} = R_{IT_2} = f/c_{IT_1} = f/c_{IT_2}$$

Figure 5.18. Updated CRS input environment.

Now the constraint equation is not satisfied, i.e.,

$$R_{IT} = \frac{1}{900} \not\leq \frac{1}{1274} = R_{CRS}.$$

One possible solution to this problem, is to decompose the CRS into component state machines by exploiting the macroparallelism available in the system definition. In order not to belabor the example, consider one possible MCS derived as shown in the block level diagram of Figure 5.19.

The maximum natural computation rates for the components are given by

R_{S1} = 1/(50 sec) R_{S5} = 1/(200 sec)

R_{S2} = 1/(100 sec) R_{S6} = 1/(400 sec)

R_{S3} = 1/(200 sec) R_{S7} = 1/(100 sec)

R_{S4} = 1/(150 sec)

Note that these computation rates are optimistic in that each requires additional time for the allocation of active resources and the provision of the appropriate interfaces between modules. However, the point being made is still basically intact. That is, the constraint equation

$$R_{IT} = \frac{1}{900} \leq \frac{1}{400} = \min \left[R_{Si} \right] \quad i = 1, 2, \ldots, 7.$$

is now satisfied for the altered system specification. One possible hardware block diagram for the implementation of the multicenter CRS of Figure 5.19 is portrayed in Figure 5.20.

Multicentered CRS evaluation

The CRS example considered here is a restricted form of a larger and more complex problem. However, it does contain the essential characteristics a typical RTDAD problem. Furthermore, the foregoing analysis has resulted in an MCS architecture that is typical of the larger

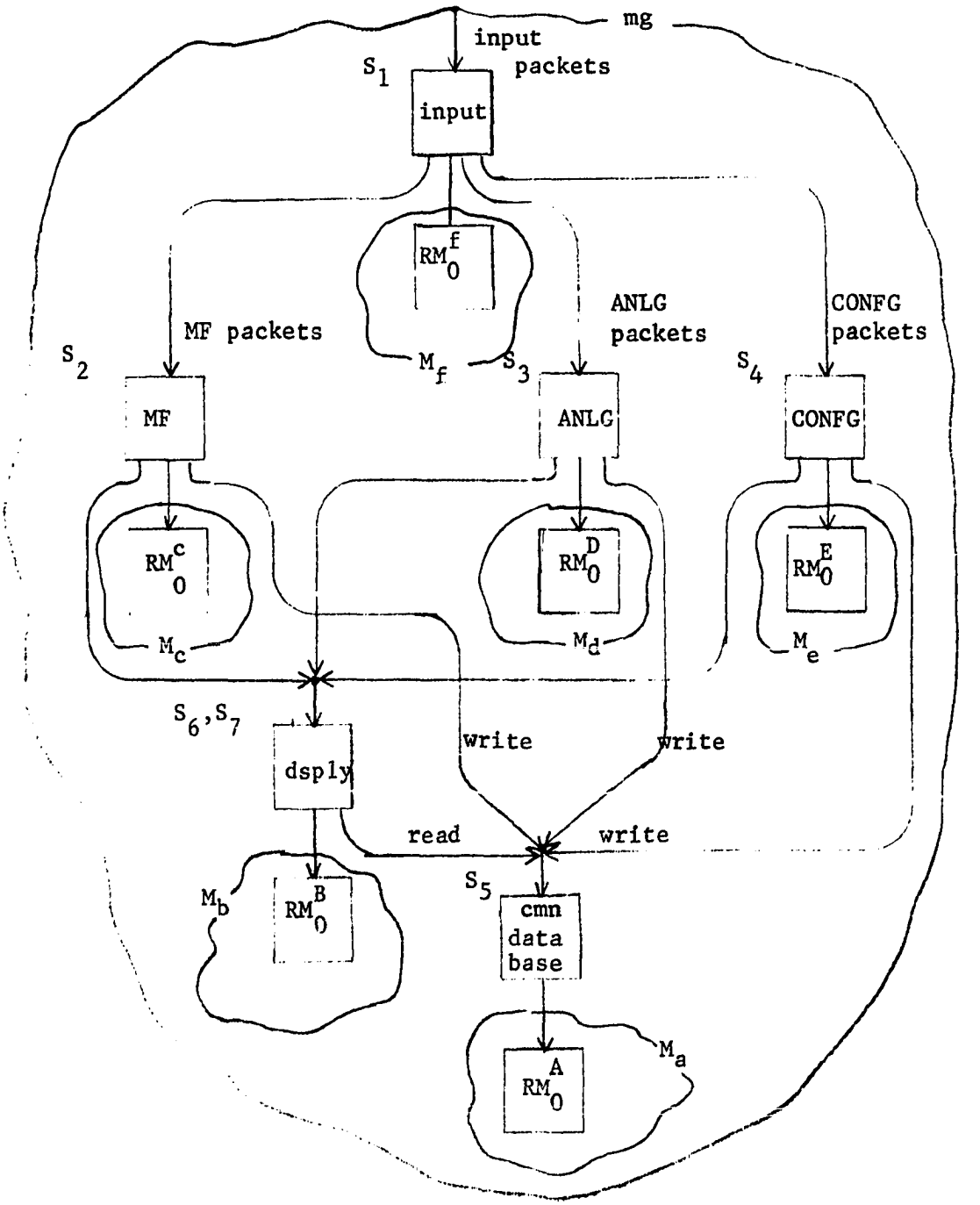


Figure 5.19. Composite BLD/contour diagram for multicentered CRS.

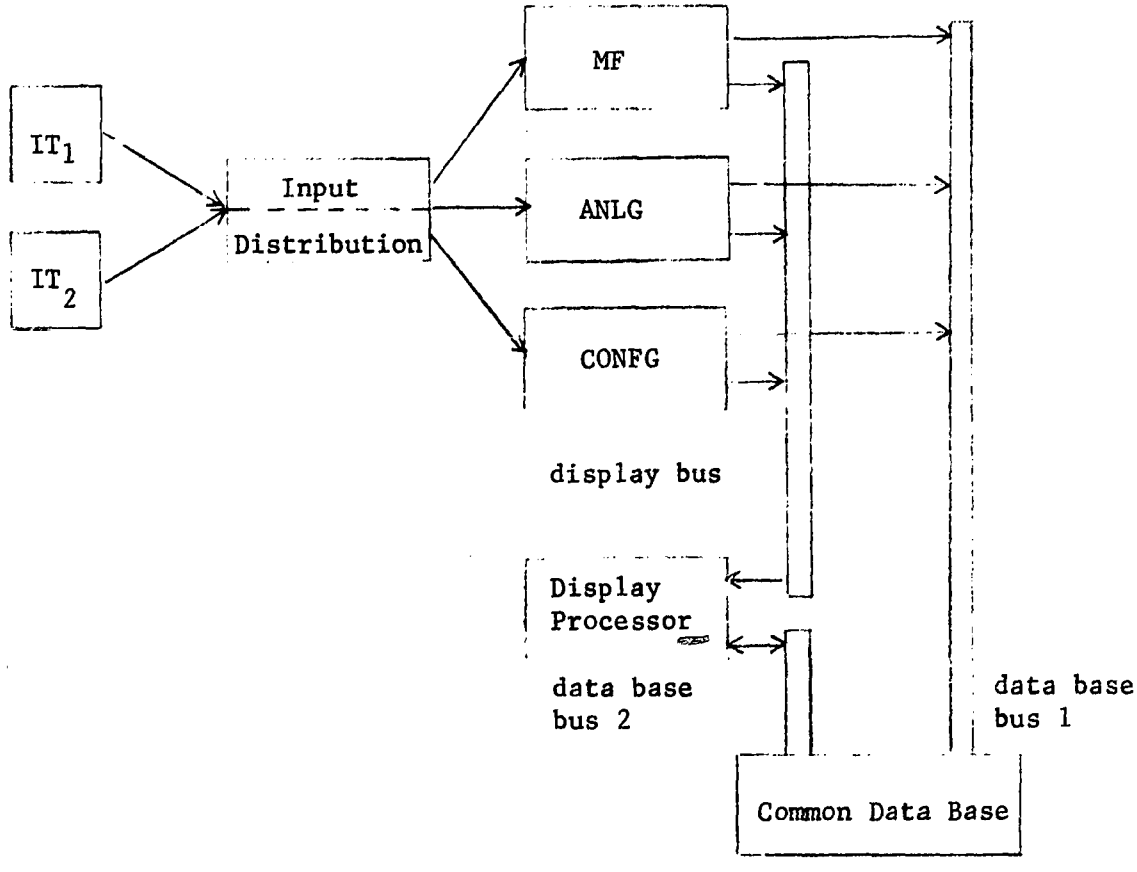


Figure 5.20. CRS hardware block diagram.

problem taken as a whole. A number of salient characteristics of the MCS architectural approach can be exposed by an examination of the CRS example. In particular, the following list summarizes these features which we feel make the multicentered structure and the attendent graph-theoretic design tools a viable approach to RTDAD system design.

1. Feasibility - The motivation behind the MCS architecture is the application of all available technologies to the functional design of a processing system. The Timed Petri nets provide a graphical tool for the portrayal of the overall structure of such a system. An application of available primitive modules (resources) results in a decomposition of the structure into a multiplicity of centers.

   The computation rate analysis provides a quantitative means by which the designer can ascertain if his decomposition will support a viable system. Specifically, he is searching for system components whose computation rates satisfy the boundary condition imposed by the input environment. Additionally, the determination of computation rates for the system components identifies the critical components or the "weakest links" in the system structure.

   As the system development progresses, the designer can verify the performance of the resulting system against the Timed Petri net model. The Petri net model supplies the basis for taking performance measurements. The model should, therefore, be verified or the reason for deviation (model error or implememtation error) should be determined.

2. Quantification of system parameters - It is our contention that one of the major problems in current RTDAD design approaches is the absence of quantitative assessment of system performance during the design stages. Many of the early (and very important) design decisions are based on intuitive or "seat-of-the-pants" judgments. The design tools presented herein equips the designer with a quantitative measure of system performance.

   Arguments for or against the use of specific resources, data structures, or operations on data structures can now be based on a quantitative assessment of the impact of the decision upon system performance. Hopefully, such quantification will remove much of the emotion of the design process with decisions substantiated by numerically determined design parameters.

3. Extendability - It is a generally accepted fact that system specifications are not a static entity. When RTDAD designs are initiated, it is appropriate that a reasonable growth potential be designed into the facility. The foregoing analysis provides a measure of the amount of excess capability that can be observed by additional processing requirements.

   Let FM be defined by

$$FM_i = \frac{\text{maximum computation rate of component } S_i}{\text{input computation boundary value}} .$$

   FM_i defines a figure of merit of the unused computation capacity of component S_i. In an MCS system, the designer is looking for both a computation rate balance among the components and favorable figures of merit.

The FM_i for the CRS system are given in Table 5.5.

An FM of, or near unity, indicates that a component can no longer absorb additional processing without seriously degrading system performance. A larger figure of merit indicates an increased capacity for extending the capabilities of the associated portion of the system.

4. Parallelism - The MCS approach exploits the inherent parallelism of the functional definition of the RTDAD processing requirements. The parallelism achieved is real and not apparent, and is not limited to the overlap of I/O processes and computation. Additionally, the parallelism is not achieved at the expense of complex and voluminous amounts of hardware and software. Individual components can actually be more simply designed by the isolation of functions to distinct components. This is in contrast to more conventional SCS systems employing complex machine-level hierarchies required to support the myriad of necessary functions.

5. Scarce resourses - The design tools and concepts introduced here support the contention that the expensive ingredient in RTDAD systems is manpower costs. These costs are incurred for software design, hardware design, system development (integration and testing) and particularly in systems maintenance. Furthermore, it is our contention that an emphasis be placed on finding the simplest, most reliable solution that provides a feasible answer to the problem statement.

Table 5.5. Figures of merit for CRS system.

a. Uniprocessor (SCS)	$FM_1 = \frac{900}{637} = 1.41$
b. MCS	$FM_1 = \frac{900}{25} = 36$ $FM_4 = \frac{900}{75} = 12$
	$FM_2 = \frac{900}{50} = 18$ $FM_5 = \frac{900}{100} = 9$
	$FM_3 = \frac{900}{100} = 9$ $FM_5 = \frac{900}{250} = 3.6$

With plummenting hardware costs, there is no longer a need to extract the last measure of system performance from each primitive module. Rather, there must be an impetus to designing systems that are cost-effective over the long-term. The concepts used in the design of the CRS are particularly well-suited for such goals. They provide an emphasis on the overall functional system design with all available technologies applied where they best fit.

6. Protection - The nature of the inherent protection embodied by MCS systems appears to be extremely attractive. The interface between components is enforced by the hardware connecting network. Only those language elements explicitly provided by a system module are available to external modules. The impact of this structure is twofold:

   a. If corruption of a system component occurs, the possible set of tasks which may have caused the failure is isolated to those constituent processes of that module. Therefore, the amount of time necessary to determine the cause of the corruption can generally be significantly reduced.

   b. Tasks or processes requesting operations of a given module can be easily identified. First of all, because of the restricted nature of the interconnect network, only a limited set of tasks physically have access to a given module. Secondly, the structure of the system can be defined so that each system component that wishes to use another component

can be uniquely identified by physical mechanisms in the network.

The latter property of unique process identification is a fundamental prerequisite for an adequate protection model (22). Furthermore, system protection can be generated at a functional level. In particular, protection constraints can be made a basic requirement in the problem statement as opposed to being an after-the-fact design consideration.

7. Concealment - The interconnection of modules physically defines the communication capability among the components for MCS systems. The operational requirements for any system component are twofold:
   a. The component must meet the language interface specified by the system design.
   b. The computation rate of the component must meet the desired system specifications.

   Otherwise, the freedom of choice in terms of implementation is unbounded. As long as the above two conditions are met, the external environment is unaware of the composition of a module. The immediate advantages of the concealment property is that components can be extended to increase efficiency, increase computation rates, improve maintenance, or increase cost-effectiveness without effecting components external to itself.

CHAPTER 6

SUMMARY

The material in this dissertation has presented an alternate approach to the design of computer-based RTDAD facilities. This approach resulted in the introduction of the concept of multicentered structures (MCS). MCS systems were shown to be based on an integrated technological design philosophy in which the designer employs hardware, firmware, and software technologies throughout the design process.

MCS system design was supported by the introduction of two graph-theoretic tools for the representation of system behavior. The System State Model depicted the functional characteristics of the problem statement; the Timed Petri nets allowed for the application of resources to the implementation of the functions defined by the System State Model.

The advantages of an MCS system architecture were discussed in Chapter 5 by means of a simplified RTDAD example. However, the full impact of the MCS approach is not yet completely known. The real test of the validity of the technique is to apply it to a real-life situation. Our next step is to do just that. Hopefully, we can refine and improve the notions presented herein based on the experience gained from such a project.

We do feel strongly that the ideas and concepts presented in this dissertation provide a solid basis for the development of such an RTDAD system. It is our hope that we can show that MCS structures will provide both short-term and long-term cost-effective RTDAD facilities.

BIBLIOGRAPHY

1. Baer, J. L.; Bovet, D. P.; and Estrin, G. "Legality and other properties of graph models of computation." Journal of the ACM 17 (July 1970): 543-554.

2. Boehm, B. W. "Software and its impact: A quantitative assessment." Datamation 19 (May 1973): 48-59.

3. Bohm, C. and Jacopini, G. "Flow diagrams, Turing machines and languages with only two formation rules." Communications of the ACM 9 (May 1966): 366-371.

4. Brinch Hansen, P. Operating System Principles. Englewood Cliffs: Prentice-Hall, Inc., 1973.

5. Brinch Hansen, P. "The nucleus of a multiprogramming system." Communications of the ACM 13 (April 1970): 238-250.

6. Busacker, R. G. and Saaty, T. L. Finite Graphs and Networks. New York: McGraw-Hill Book Co., 1965.

7. Butler, M. K. "Prospective capabilities in hardware." AFIPS National Computer Conference Proceedings 45 (1976): 323-336.

8. Caplender, H. D. and Janku, J. A. "Top-down approach to LSI system design." Computer Design 13 (August 1974): 143-148.

9. Chu, Y. "A methodology for software engineering." IEEE Transactions on Software Engineering SE-1 (September 1975): 262-270.

10. Colon, F. C.; Glorioso, R. M.; Walter, H. K.; and Li, D. W. "Coupling small computers for performance enhancement." AFIPS National Computer Conference Proceedings 45 (1976): 755-764.

11. Dennis, J. B. "First version of a data-flow procedure language." Project MAC Computation Structures Group Memo 93-1. Massachusetts Institute of Technology, Cambridge, Mass., 1974.

12. Dennis, J. B. "Modular, asynchronous control structures for a high performance processor." Record of the Project MAC Conference on Concurrent Systems and Parallel Computation. New York: ACM, 1970.

13. Dennis, J. B. "Notes on the design of a common base language." Tutorial Symposium on Semantic Models of Computation, New Mexico State University, Las Cruces, N. M., 1972.

14. Dennis, S. F. and Smith M. G. "LSI-implications for future design and architecture." AFIPS Spring Joint Computer Conference Proceedings 40(1972): 343-351.

15. Dijkstra, E. W. "Go To statement considered harmful." Communications of the ACM 11 (March 1968): 147-148.

16. Dijkstra, E. W. "The humble programmer." Communications of the ACM 15 (October 1972): 859-866.

17. Dijkstra, E. W. "The structure of the 'THE' multiprogramming system." Communications of the ACM 11 (May 1968): 341-346.

18. Donaldson, J. R. "Structured programming." Datamation 19 (December 1973): 52-54.

19. Enslow, P. E., Jr., gen. ed. Multiprocessors and parallel processing. New York: John Wiley and Sons, Inc., 1974.

20. Farber, D. J. "Networks: An introduction." Datamation 18 (April 1972): 36-39.

21. Gagliardi, J. O. "Trends in computer-system architecture." Proceedings of the IEEE 63 (June 1975): 858-862.

22. Graham, G. S. and Denning, P. J. "Protection: Principles and practice." AFIPS Spring Joint Computer Conference Proceedings 40 (1972): 417-429.

23. Habermann, A. N. Introduction to Operating System Design. Chicago: Science Research Associates, Inc., 1976.

24. Hawryskiewitz, I. T. "Semantics of data base systems." Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Mass., 1973.

25. Hodges, D. A. "Trends in computer hardware technology." Computer Design 15 (February 1976): 66-85.

26. Holt, A. W. and Commoner, F. "Events and conditions." Record of the Project MAC Conference on Concurrent Systems and Parallel Computations. New York: ACM, 1970.

27. Joseph, E. C. "Innovations in heterogeneous and homogeneous distributed-function architectures." Computer 7 (March 1974): 17-24.

28. Karp, R. M. and Miller, R. E. "Properties of a model for parallel computation - determinancy, termination, and queueing." SIAM Journal of Applied Mathematics 14 (November 1966): 1390-1411.

29. Liskov, B. H. "The design of the Venus Operating system." Communications of the ACM 15 (March 1972): 144-156.

30. Liskov, B. H. and Zilles, S. "Programming with abstract data types." Project MAC Computation Structures Group Memo 99. Massachusetts Institute of Technology, Cambridge, Mass., 1974.

31. Liskov, B. H. "SPIL: A language for system design and implementation." Project MAC Computation Structures Group Memo 80. Massachusetts Institute of Technology, Cambridge, Mass., 1973.

32. Martin, D. F. and Estrin, G. "Experiments on models of computations and systems." IEEE Transactions on Computers 16 (February 1967): 59-60.

33. Martin, D. F. and Estrin, G. "Models of computation and systems evaluation of vertex probabilities in graph models of computation." Journal of the ACM 14 (April 1967): 281-299.

34. McGowan, C. L. and Kelly, J. R. Top-Down Structured Programming. New York: Mason/Charter Publishers, Inc., 1975.

35. Miller, G. F. and Lindamood, G. E. "Structured programming: Top-down approach." Datamation 19 (December 1973): 55-57.

36. Miller, R. E. "A comparison of some theoretical models of parallel computation." IEEE Transactions on Computers 8 (August 1973): 710-717.

37. Misunas, D. "Petri nets and speed independent designs." Communications of the ACM 16 (August 1973): 474-481.

38. Myers, G. J. Reliable Software Through Composite Design. New York: Mason/Charter Publishers, Inc., 1975.

39. Neuhold, E. J. "Formal description of programming languages." IBM Systems Journal 2 (1971): 86-112.

40. Noe, J. D. "A Petri net model of the CDC 6400." Workshop on System Performance Evaluation. New York: ACM, 1971.

41. Noe, J. D. and Nutt, G. J. "Macro E-nets for representation of parallel systems." IEEE Transactions on Computers C-22 (August 1973): 718-727.

42. Noe, J. D. and Nutt, G. J. "Multiprogramming and multiprocessing system description." Computer Science Group Report TR 73-01-08. University of Washington, Seattle, Wash., 1973.

43. Noe, J. D. and Nutt, G. J. "Some evaluation net macro structures." Computer Science Group Report TR 73-01-07. University of Washington, Seattle, Wash., 1973.

44. Nutt, G. J. "Evaluation nets for computer system performance analysis." AFIPS Fall Joint Computer Conference Proceedings 41 (1972): 279-286.

45. Parnas, D. L. "On the criteria to be used in decomposing systems into modules." Communications of the ACM 15 (December 1972): 1053-1058.

46. Parnas, D. L. "A technique for software module specifications." Communications of the ACM 15 (May 1972): 330-336.

47. Patil, S. S. "On structured digital systems." Proceedings of IEEE Workshop on Computer Hardware Design Languages and Their Application. New York: IEEE, 1975.

48. Patil, S. S. and Dennis, J. B. "The description and realization of digital systems." Digest of Papers -- 1972: Innovative Architecture. COMPCON72. New York: IEEE, 1972.

49. Pike, H. E. "Future trends in software development for real-time industrial automation." AFIPS Spring Joint Computer Conference Proceedings 40 (1972): 915-923.

50. Pike, H. E. "Process control software." Proceedings of IEEE 58 (January 1970): 87-97.

51. Purser, W. F. C. and Jennings, D. M. "The design of a real-time operating system for a minicomputer. Part 1." Software - Practice and Experience 5 (1975): 147-167.

52. Ramchandi, C. "Analysis of asynchronous concurrent systems by Petri nets." Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Mass., 1974.

53. Rose, C. W. "LOGOS and the software engineer." AFIPS Fall Joint Computer Conference Proceedings 41 (1972): 311-323.

54. Rudenberg, H. G. "Approaching the minicomputer on a silicon chip - progress and expectation for LSI circuits." AFIPS Spring Joint Computer Conference Proceedings 40 (1972): 775-781.

55. Schaeffler, J. D. "The development of process control software." AFIPS Spring Joint Computer Conference Proceedings 40 (1972): 907-914.

56. Shaw, A. C. The Logical Design of Operating Systems. Englewood Cliffs: Prentice-Hall, Inc., 1974.

57. Smith, C. L. "Digital control of industrial processes." Computing Surveys 2 (September 1970): 211-241.

58. Smith, W. R.; Rice, R.; Chesley, G. D.; Laliotis, T. A.; Lundstrom, S. F.; Calhoun, M. A.; Gerould, L. D.; and Cook, T. G. "Symbol -- a large experimental system exploring major hardware replacement of software." AFIPS Spring Joint Computer Conference Proceedings 38 (1971): 601-616.

59. Tanenbaum, A. S. Structured Computer Organization. Englewood Cliffs: Prentice-Hall, Inc., 1976.

60. Thurber, K. J.; Jensen, E. D.; Jack, L. A.; Kinney, L. L.; Patton, P. C.; and Anderson L. C. "A systematic approach to the design of digital bussing structures." AFIPS Fall Joint Computer Conference Proceedings 41 (1972): 719-740.

61. Wegner, P. "The Vienna definition language." ACM Computing Surveys 4 (March 1972): 5-63.

62. Weissberger, A. J. "Distributed function microprocessor architecture." Computer Design 13 (November 1974): 77-83.

63. Wirth, N. "Program development by stepwise refinement." Communications of the ACM 14 (April 1971): 221-227.

64. Wulf, W. A. and Bell, C. G. "C.mmp -- a multi-mini-processor." AFIPS Fall Joint Computer Conference Proceedings 41 (1972): 765-778.

65. Wulf, W. A.; Cohan, W. C.; Jones, A.; Levin, R.; Pierce, C.; and Pollack, F. "Hydra: The kernel of a multiprocessor system." Communications of the ACM 17 (June 1974): 337-345.

66. Yourdon, E. Design of On-Line Computer Systems. Englewood Cliffs: Prentice-Hall, Inc., 1972.