MULTIPROCESSING USING PROGRAM STRUCTURES

Dissertation for the Degree of Ph. D.

MICHIGAN STATE UNIVERSITY
EDWARD LOUIS LAMIE

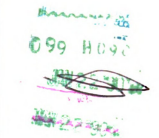
1974

MICHIGAN STATE UNIVERSITY LIBRARIES
3 1293 10641 8761

LIBRARY
Michigan State
University

BINDING BY
HOAG & SONS'
BOOK BINDERY INC.
LIBRARY BINDERS
PORTAGE, MICHIGAN

JAN 22 1997



ABSTRACT

MULTIPROCESSING USING PROGRAM STRUCTURES

By

Edward Louis Lamie

A model which is capable of representing computations on a multiprocessing system is developed. The model takes the appearance of a directed graph where each node contains both a data vector and a mapping vector. The copy of a node concept is introduced and is used extensively throughout the dissertation. Various properties of the model are investigated including equivalence classes of input data and maximally parallel form. Solutions to the problems of repeatability, interference, and deadlock are presented. Properties of models which have restrictions placed on available resources are also studied.

MULTIPROCESSING USING PROGRAM STRUCTURES

By

Edward Louis Lamie

A DISSERTATION

Submitted to

Michigan State University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1974

ACKNOWLEDGEMENTS

I am deeply grateful to Professor Lew Greenberg for helping me to develop many of the ideas in this dissertation and also for serving as my committee chairman. I am also grateful to Professors Harry Hedges, Marshall Hestenes, and Carl Page for serving on my doctoral committee.

I am also indebted to my wife, Mary Ellen, for her patience and understanding throughout my years of study.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF SYMBOLS

Symbol	Meaning	Page
N^i	node N^i	10
D^i	the data vector of N^i	10
M^i	the mapping vector of N^i	10
ϵ	is an element of some set or vector	10
?	undefined element	11
#(B)	number of elements in B	11
$\mathcal{D}(D^i_j)$	predicate to determine whether D^i_j is defined	13
$N^{i,j}$	the jth copy of N^i	16
K	set of known nodes	17
U	set of unknown nodes	17
A	set of active nodes	17
$\exists$	there exist(s)	17
$\ni$	such that	17
B & C	an operation on the elements of B and C	18
$\ominus$	performance of a mapping	19
$\forall$	for all	20
$\subseteq$	is a subset of	20
I	set of input nodes	31
O	set of output nodes	32
$\$N^k$	N^k is a junction	40
N	index set of all nodes	46
C	classification set for all nodes	46

LIST OF SYMBOLS (continued)

Symbol	Meaning	Page
T	set of test values	46
$\wedge$	logical and	48
V^i	value vector of N^i	59
FPS	Finite Program Structures	78
Size(N^i)	amount of memory required by N^i	78
Mem	total amount of memory available	78
t_j, T^i	sets used to describe A and K	81
D	set of deadlock free execution sequences	82
CFPS	Copy Free Program Structures	85

CHAPTER I

INTRODUCTION

1.1 Background

Since the first electronic digital computer was introduced in the 1940's, there have been numerous increases in device speed and reliability. Most of these advances can be directly attributed to hardware technology such as miniaturization. However, technology appears to be approaching a physical limitation as it attempts to greatly increase device speed. Consider the fact that electricity can travel approximately one foot in one nanosecond and many of today's computers operate with speeds in the order of tens of nanoseconds. Future advances in technology such as large scale integration will undoubtedly increase device speed. However, major improvements in device speed and reliability will probably occur as a result of increased parallelism.

Parallel operation is not a new concept. It has long been present in many hardware devices. For example, one technique to increase the speed of adders and multipliers is to perform many of the bit operations in parallel.† Another way to improve system performance is to have the input/output devices and their channels operate in parallel with the central processing unit. Still another method of increasing computational capability is to develop a system which has two or more processors which can operate in parallel. The ILLIAC IV is an example of such a system (see Figure 1.1). It consists of four quadrants wherein each quadrant contains sixty-four processing (or arithmetic) elements and one control unit. The

---

† See "Design of parallel binary adders", Hellerman (24)

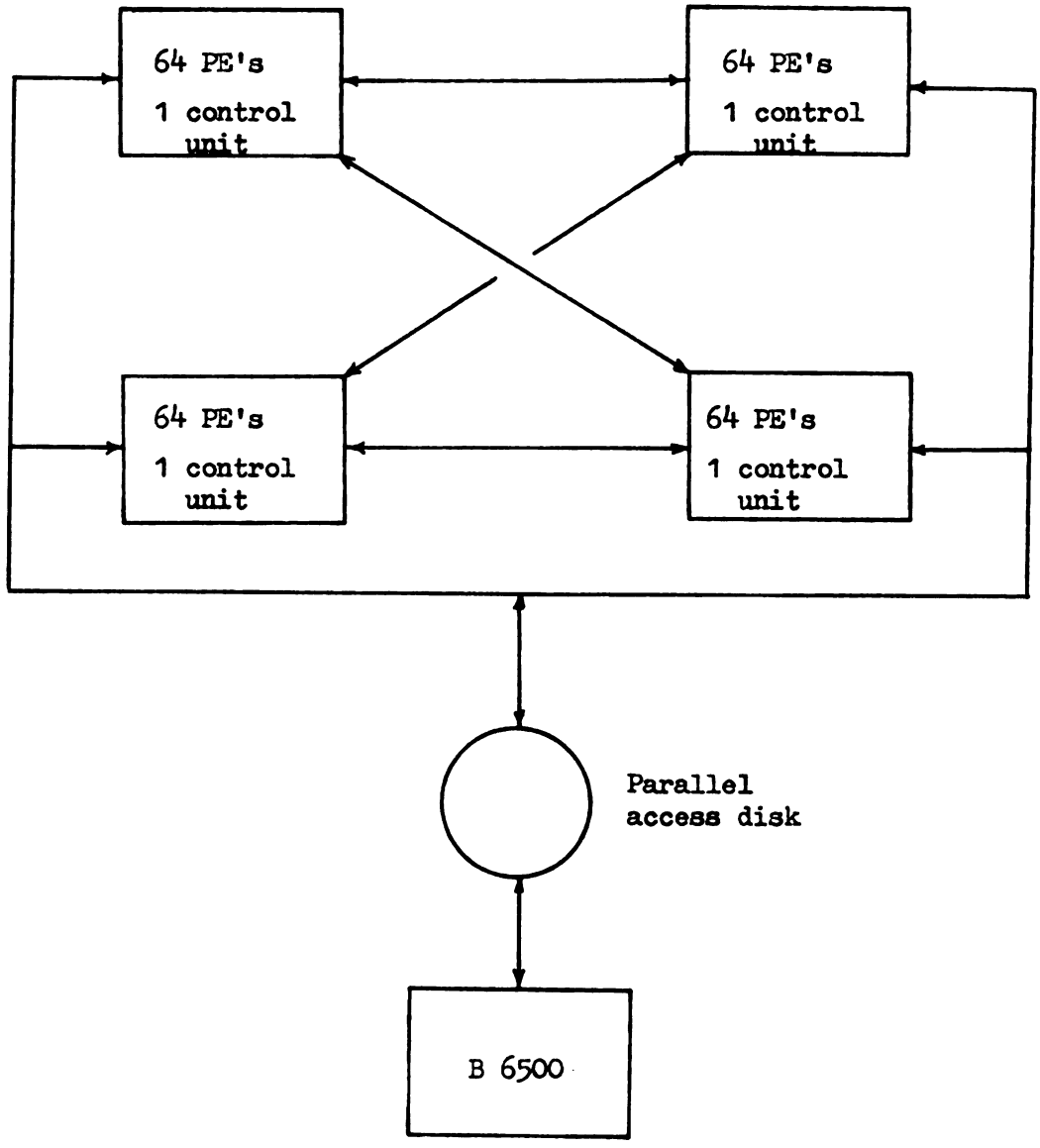


Figure 1.1 Organization of the ILLIAC IV

four quadrants share a uniprocessor (a Burroughs 6500) and a large disc memory. Each of the four quadrants can work independently or they can be joined together in a single array to process large problems. A detailed discussion of the ILLIAC IV can be found in (4, 35).

Besides speed and reliability, there is another important reason to investigate parallel processing (the term multiprocessing is used synonomously). That reason is economics. As computer systems become more complex, effective utilization of the system's resources is necessary in order for the system to be cost effective. Time sharing and real time processing present problems where all of these considerations must be taken into account.

That multiprocessing is an important topic is evidenced by the fact that discussion of this problem appears in most recent texts on operating systems (6,14,45). Even though the justification for using multiprocessing may be apparent, the techniques for establishing such systems are not well developed. Organizing computations so that a number of processors can be simultaneously working on them is a complex task. Organizing the computations so that all n processors of an n processor system are effectively utilized is an even more complex task.

To illustrate one type of problem which can arise, consider the following data transformation D1 and D2 where x, y, and z are storage locations and f, g, and h are operators.

D1 $f(x) \rightarrow x$
$g(x) \rightarrow y$

D2 $h(y) \rightarrow z$

The data transformations D1 and D2 must be done sequentially, but if the operations in D1 are performed concurrently, then the result of D2 will

vary depending on the speed with which the operations in D1 are performed. For example, if operation f is performed quickly, then it is possible to change the value of x before operation g is begun. Another problem which can arise is that f will attempt to change the value of x at the same instant that g is attempting to access the value of x.

1.2 Previous Work

Petri nets were developed by C. A. Petri (45) and are a means of representing concurrent operations. A Petri net is a labelled directed graph that has only two node types call "places" and "transitions". Performing an operation is analogous to "firing" a transition in Petri terminology. If conditions are met as specified by the places, the transitions can be fired asynchronously. Petri nets have been used to study conditions leading to deadlock (a situation in which it becomes impossible to fire any transitions). Special cases of Petri nets have been analyzed and solutions to the deadlock problem have been developed. However, for many problems including the mutual exclusion relationship between two transitions, only the general Petri net can be used. Many of the concepts established by Petri can be found in other models including Luconi (34), Karp and Miller (29), Rodriguez (41), and Slutz (43). A major drawback of Petri nets is that they are not sufficiently general to handle parallel computations. A good example of this is the fact that the "not" operation (a transition fires if and only if a place if empty) cannot be implemented.

Karp and Miller (29) have introduced a mathematical model for parallel computation which is called "Parallel program schemata". The model is an asynchronous system which consists of a set of operations

which operate on a set of memory locations. Each schemata can be defined by two directed graphs, a data flow graph and a control graph. The data flow graph specifies the domain and range (in terms of memory locations) of each operation. Thus there are two types of nodes in this graph, one type to represent operations and the other to represent memory locations. Similarly, there are two types of nodes in a control graph, one to represent operations and the other to represent control states. The control graph is used to specify the order in which the operations will be initiated. More specifically, the control states determine when operations can be initiated. The control graph is similar to a Petri net. The model developed by Slutz (43) is a generalization of the Karp and Miller model. Among other results obtained by Karp and Miller, decision procedures were established for such properties as equivalence, determinacy, and boundedness.

Dijkstra (19) has proposed a method of communication between two or more processors that share memory. All of the processors have access to special memory locations called semaphores. The semaphores contain information that is used to block a processor from entering its "critical state" if any other processor is in its critical state. Thus, the processors can communicate with each other, but they are prevented from interfering with each other, i.e., having two processors in their critical states at the same time. Dijkstra also considers the problem of "deadly embrace", i.e., two processors need to go into their critical states but each is waiting for the other to go first. Neither processor ever gets to its critical state because each is saying, in effect: After you, after you, after you,..... Dijkstra is not concerned with the writing of parallel programs, but rather specification of a system which operates in parallel.

Rodriguez (41) has introduced a parallel program model which he calls "Program graphs". In this model, the computation elements are represented by nodes of a directed graph. Storage and transmission of information are represented by the links between the nodes. The activation of a node depends entirely on information residing in links that point to that node. At any point in time, there may be many nodes that are active. Looping is achieved with the help of special node types and connection rules are specified so that determinism is assured. Program graphs are shown to be deterministic in general, i.e., for any computation the final state is unique if started from the same initial state.

Luconi (34) has proposed a model for representing communicating processes which he calls "Asynchronous Computational Structures". These structures have the ability of sharing memory and allowing the processes to proceed concurrently. Depicted graphically, operators reside in named nodes while information resides in links which are also given a node-like representation. There are no explicit timing constraints placed on the operators, so they can act asynchronously with respect to each other. Complete functionality (a form of determinism) is proven based on several conditions. The work of Rodriguez is given as an example of a computational structure.

Slutz (43) has developed a model for parallel algorithms which he calls "Flow Graph Schemata". The model is depicted as two directed graphs, one for data flow and the other for control. In the data flow graph, memory cells are represented by circular nodes. As in other models, the functions may act asynchronously with respect to each other. It is also possible for two or more applications of the same function to be in progress concurrently. This behavior, called pipelining, uses

a FIFO queue to store invocations of any function. Conditions for determinism are established and equivalence of certain classes of Flow Graph Schemata are investigated.

Some attempts to model parallel computations have taken the approach of adding parallel instructions to an existing programming language such as ALGOL or PL/I. Anderson (2) has proposed use of the fork/join statements as one method of explicitly declaring that two or more procedures may be executed in parallel. Similar approaches have been taken by Dijkstra (19), Conway (16), and Dennis and Van Horn (18). This type of approach is satisfactory as long as the parallel procedures do not operate on common variables. If the procedures do operate on common variables, then precautions must be taken to avoid certain timing problems such as the possibility of simultaneously changing the value of a common variable. Anderson (2) has stated that his terminate, obtain, and release statements solve this problem for his approach.

Although previous papers have proposed various solutions to problems posed by multiprocessing, none of them are widely used. This is due principally to a lack of generality or the cumbersomeness of the models. It is difficult to imagine any large scale problem being programmed using any of the previously discussed models. The model described in chapter II is intended to be useful in the creation of programs that are easily written and can be efficiently executed on multiprocessor systems. As such it is a programming scheme rather than a model for machine design.

1.3 Statement of Problem

This research is centered around the development of a model for parallel computation which is called "Program Structures". The model

is more general, more powerful, and conceptually simpler than the previous models discussed. Some problems that plague other models are not relevant here. For example, it is shown that the problem of two processors being in their critical states simultaneously is not really a problem at all. Consequently, other problems such as the deadly embrace also disappear in this model.

This dissertation is directed toward the creation of algorithms or programs which can be executed on a multiprocessing system. The model that achieves this situation takes on the appearance of a directed graph where the nodes contain both information and transformations or mappings of that imformation. The directed arcs are used to indicate the transmission of the transformed information to other nodes. There is no explicit control mechanism established to direct the execution of the program. Control is considered to be taken care of by the problem description itself and thus is an integral part of the parallel program.

Shared memory is used in this model. However, many traditional problems are avoided by the use of a new feature call the "copy" of a node. In previous models, there could only be one access to a given piece of information in memory allowed at any one time. Using the concept of copied nodes, there is no limit to the number of simultaneous accesses to a given piece of information in memory. Besides eliminating many previous problems, use of this concept has the effect of speeding up the execution of a program.

A formal description of the model is presented, along with several examples. Capabilities of the model are discussed as well as a proof that the model is capable of implementing any flowchart schema. It is shown that for some Program Structures the set of input data can be

placed in equivalence classes. An analysis formalism is developed to aid in the discussion of various properties of Program Structures. Conditions for a form of determinism (called repeatability) are established and maximal parallelism is investigated. Solutions to problems of interference and deadlock are presented. Finally, properties of Finite Program Strutures and Copy Free Program Structures are investigated and conditions for equivalence are presented.

CHAPTER II

FORMAL DEFINITION

2.1 Model Specification

The purpose of this section is to describe Program Structures as well as some of their components and attributes. In order to do this, the simplest element, the node, is defined first. After that, the properties of the node are defined and then the definition of Program Structure is presented.

As has been stated before, this model is depicted as a directed graph wherein the nodes contain both data values and mappings between nodes. The directed arcs are used to indicate which nodes are to have information mapped into them.

Definition 2.1

A node N^i is a set defined as follows:
$N^i = \{D^i, M^i\}$ where D^i is an n-tuple which contains information associated with N^i, where $n > 0$, and M^i is an m-tuple whose components are mappings of D^i into various nodes, where $m \geq 0$. If $f \in M^i$, then f is a rule which uses some or all of the data in D^i to determine some or all of the data values of some D^j. D^i is referred to as the data vector and M^i is referred to as the mapping vector.†

For example, Figure 2.1 shows a pictorial representation of a node. In this example, node N^i might be used to represent a point in space that

---

† The set notation $f \in M^i$ is also used to denote some element in the m-tuple M^i.

passes information on to two other nodes. The mappings f and g would be previously defined.

In order to facilitate the discussion of properties of nodes, several notations and definitions are now introduced.

Definition 2.2

An element $x \in D^i$ is said to be defined if it has a value assigned to it. Otherwise, that element is said to be undefined. For convenience, undefined elements will be denoted by a question mark "?".

Definition 2.3

Given a set or vector B, let #(B) be defined to be n, where n is the number of elements in B.

For example, given $D^i \in N^i$, then $\#(D^i) = j$ where D^i is a j-tuple.

Frequently there may be many elements in D^i or M^i and it is often necessary to refer to a specific element. The next definition provides the notation to accomplish this.

Definition 2.4

Given D^i, the j^{th} element of D^i is denoted by D^i_j. Similarly, the k^{th} element of M^i is denoted by M^i_k or f^i_k.

It has been established that any element in D^i is either defined or is undefined. This property is very important when the type of a node is being determined. The next definition provides notation that can be used in classifying a node.

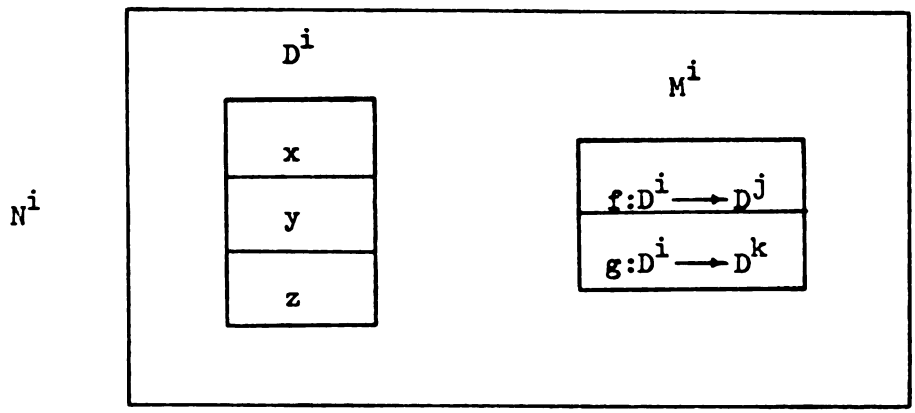


Figure 2.1 A pictorial representation of node N^i

Definition 2.5

Given $D_j^i \in D^i$, let $@(D_j^i)$ be defined as follows:

$$@(D_j^i) = \begin{cases} 1 \text{ if } D_j^i \text{ is defined} \\ 0 \text{ otherwise} \end{cases}$$

Now it is possible to determine the type classification of any node. This classification is not only fundamental, but essential to the asynchronous operation of Program Strutures.

Definition 2.6

If $D^i \in N^i$ and $\sum_{j=1}^{\#(D^i)} @(D_j^i) = \#(D^i)$, then node N^i is classified as a known node. Furthermore, if a node is not a known node, then it is classified as an unknown node.

This definition simply states that if every element in D^i is defined, then node N^i is a known node.

During the execution of a Program Structure, nodes will have values mapped into their data vectors. Whenever a node has values specified for every element in its data vector, then that node becomes a known node. Until that happens, the node is classified as unknown. The reason for making this distinction is that when the execution of a Program Struture is defined, only known nodes can be processed.

The mapping vector of a node is used to transform or manipulate information in the data vector and then place it in other nodes. The mapping vector can be the empty set or it can contain one or more mappings. In any case, the contents of the mapping vector remains

unchanged throughout the execution of a Program Structure.

Definition 2.7

The domain of a node N^i will be denoted by Domain (N^i). The range of a node N^i will be denoted by Range (N^i). For any node N^i,

$$\text{Domain } (N^i) = D^i \text{ where } D^i \in N^i$$

$$\text{Range } (N^i) = \left\{ D^j \;\middle|\; D^j \in N^j \text{ where } N^j \text{ is an unknown node and } f^j : D^i \rightarrow D^j \right\}$$

The primary consequence of this definition is that if f is a mapping from N^i into N^j, then N^j must be an unknown node. This further eliminates the possibility of mapping a known node into another known node and changing the result of a previous computation.

As an example, consider the trigonometric problem in Figure 2.2 where (x_1, y_1, d_1) and (x_2, y_2, d_2) are known and (x_3, y_3) is unknown. The quantities d_1 and d_2 are direction vectors and (x_i, y_i) is a two dimensional coordinate.

The problem can be solved by using two known nodes N^1 and N^2, and two unknown nodes N^3 and N^4. The mappings f, g, and h are used and are defined as follows:

$f(D^1) = (D^1_1, D^1_2, D^1_3, ?, ?, ?)$ where ? indicates an undefined value

$g(D^2) = (?, ?, ?, D^2_1, D^2_2, D^2_3)$

$h(D^3) = (x_3, y_3, ?)$ where h calculates x_3 and y_3

From the preceding example it is apparent that for any $M^i \in N^i$, M^i consists of zero or more mappings from $D^i \in M^i$ into the range of N^i.

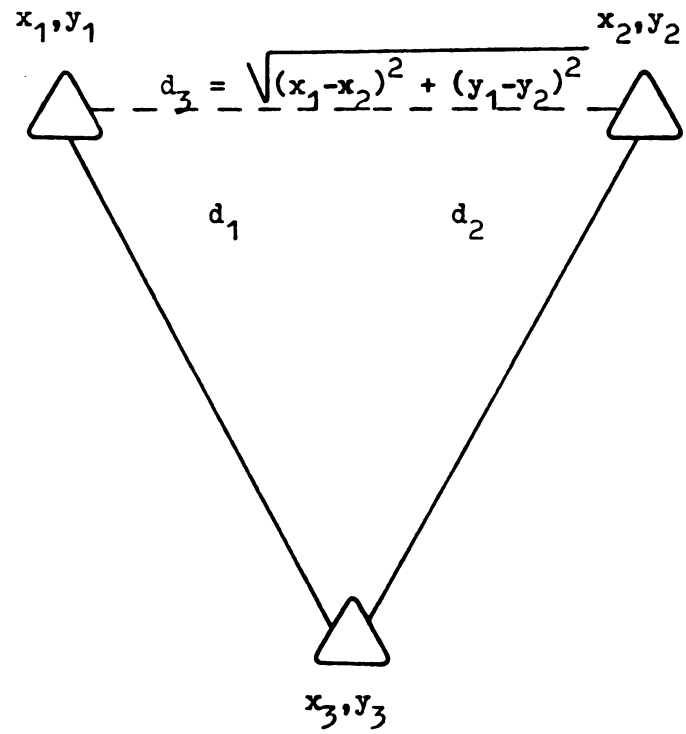


Figure 2.2 Trigonometric problem using two dimensional coordinates

In most models of parallel processing, much attention is given to the control of execution of the model. In some cases, such as the models proposed by Rodriguez and Slutz, control is maintained by the use of control nodes or even a control graph. In the model being proposed here, control is not a major concern because the data being processed will serve implicitly as the control. The next concept which is defined will help achieve this effect.

Definition 2.8

A copy of a node N^i is another node which has the identical mapping vector M^i, the same number of elements in the data vector D^i, but possibly the contents of one or more of the data elements differ.

The notion of a copy of a node will be used extensively in the execution of Program Structures. Since it is possible to have a number of copies of a node, some additional notation is necessary to eliminate any possible confusion.

Definition 2.9

The j^{th} copy of node N^i will be denoted by $N^{i,j}$. In particular, the original node N^i will be denoted by $N^{i,0}$ or simply as N^i. The same notation will be used for the mapping and data vectors as well.

Having established these preliminary concepts, it is now possible to present the definition for the Program Structure.

Definition 2.10

A Program Structure is a triple (K, U, A) where

$K = \{N^{i,j} \mid N^{i,j}$ is a known node$\}$ and K is called the set of known nodes

$U = \{N^{i,0} \mid N^{i,0}$ is an unknown node$\}$ and U is called the set of unknown nodes, the set U never decreases in size and the data elements in each node of U are never changed.

$A = \{N^{i,j} \mid N^{i,j}$ is a copy of some node in U, $j > 0$, $N^{i,j}$ is unknown, and $\exists\ x \in D^{i,j} \ni @(x) = 1\}$ and A is called the set of active nodes

The most striking difference between Program Structures and earlier models of parallel computation is the simplicity of the former. This simplicity does have a price, i.e., the amount of storage required by Program Structures would probably be greater than that of most other models if they were implemented on a multiprocessing system. On the other hand, the amount of internal overhead would most likely be drastically reduced and many problems encountered in other models are easily overcome by Program Strutures as demonstrated in Chapter V.

The set of known nodes is analogous to a set of "current states" using Finite State Machine terminology. Since a Finite State Machine is a sequential machine, there is at most one current state at any instant in time. Since a Program Structure is intended to resemble a parallel machine, there can be many current states at any point in time. Similarly, the set of active nodes is analogous to a set of "next states". The set of unknown nodes does not have a direct correlation to Finite State Machines, but it can be regarded as a "master file".

When a node maps information to another node it might be possible that the contents of a previously defined element of a data vector would be altered. To avoid this situation, mappings will be performed in a special manner. In order to define how mappings are carried out, the following operation on sets is required.

Definition 2.11

Given n-tuples B and C where #(B) = #(C), B & C is the operation on the components of B and C defined as follows:

If $\exists\ b_i \in B,\ c_i \in C \ni @(b_i) = @(c_i) = 1$ then B & C = B

else B & C = D where #(D) = #(B) and

$$D = \left\{ d_i \mid d_i = \begin{cases} b_i & \text{if } @(c_i) = 0 \\ c_i & \text{if } @(b_i) = 0 \\ ? & \text{if } @(b_i) = @(c_i) = 0 \end{cases} \right\}$$

$$\text{for } i = 1,2,\ldots.\#(B)$$

For example, if B = $\{1, 2, ?, ?\}$ and C = $\{?, ?, 3, ?\}$ then B & C = $\{1, 2, 3, ?\}$.

Similarly, if B = $\{1, 2, ?, ?\}$ and C = $\{?, 3, 2, 1\}$ then B & C = $\{1, 2, ?, ?\}$.

Definition 2.12

The mapping $f:D^{i,j} \rightarrow D^{k,m}$ is performed as follows:

1. A copy of $N^{k,0}$ is placed in A only if another copy of $N^{k,0}$ is

not in A or if the information portion of those copies in A have values defined in the positions which would be defined by f

2. $f(D^{i,j}) = (d_1, d_2, \ldots\ldots, d_n)$ is evaluated where $n = \#(D^{k,m})$ and $N^{i,j}$ is a known node
3. $D^{k,r} \leftarrow D^{k,r}$ & $f(D^{i,j})$ for all $N^{k,r} \in A$ where $r > 0$
4. If all the elements of data vector $D^{k,r}$ are defined, then node $N^{k,r}$ becomes a known node and is removed from A and added to K

The definition of mapping above makes it impossible for any node to alter the results of a previously processed node. It is assumed that creation of copies is an indivisible operation. If two nodes simultaneously map to a common node and a copy does not exist, then only one copy would be created and placed in A.

Definition 2.13

The performance of the mapping as defined above will be deonted by $D^{k,m} \ominus f(D^{i,j})$ where $f:D^{i,j} \rightarrow D^{k,m}$.

The execution of a Program Structure is simply a matter of visiting the nodes in the set K. When there are no more nodes left to visit, execution terminates. When visiting a node, all of the mappings in the mapping vector are performed and information is passed to the specified nodes. If there are two or more nodes in the set K, then these nodes may be visited in any order. When visiting a node, it is first removed from the set K; after being visited, it is destroyed. The following definitions formalize these notations.

Definition 2.14

A node is said to be processed when that node is visited and all of its mappings are performed.

The next definition specifies the operation of a Program Structure. Before proceeding with it, note that whenever an unknown node has all of the elements of its data vector defined, it becomes a known node and is added to the set K.

Definition 2.15

A Program Structure is said to be executed when the nodes in the set K are processed. Execution is terminated when the set K becomes empty and there is no node currently being processed.

It is not necessary to prescribe a procedure for executing a Program Structure since only known nodes can be evaluated. However, the following proposition presents a possible algorithm for executing a Program Structure.

Proposition 2.1

Any Program Structure can be executed using the following algorithm:

1. If $K = \emptyset$ and there is no node currently being processed, then go to step 6
2. Delete a node $N^{i,k}$ from the set K
3. $\forall\ f \in M^{i,k}$ (w.l.o.g. assume $f:D^{i,k} \rightarrow D^{j,m}$ where $D^{j,m} \subseteq$ Range $(N^{i,k})$) perform $D^{j,m} \leftarrow f(D^{i,k})$
4. If any nodes in A become known, delete them from A and add them to K

5. Go to step 1

6. Terminate

Proof: Obvious from definition 2.15.

2.2 Examples

Several examples of Program Structures are now given in order to make clear some of the preceding definitions. A conventional flowchart is shown and compared to the equivalent Program Structure in each example.

The first example is the implementation of the Newton-Raphson root finding algorithm, i.e., given an initial guess x, "improve" x until a specified degree of accuracy a is met:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad \text{where the initial guess is } x_0$$

The values for the initial guess x, the accuracy a, and a control value n are input. The conventional flowchart appears in Figure 2.3 while the corresponding Program Structure appears in Figure 2.4.

In figure 2.4 the root of the equation will appear in node N^2 if a root is found within n iterations. Otherwise, the most recent value computed for x will appear in node N^4. At any rate, when execution is terminated either node N^2 or node N^4 (but not both) will appear in set A and that result could be used by still another Program Structure. Similarly, in Figure 2.3 the conventional flowchart could be a subprogram which is used to communicate with a main program or other subprograms.

As a numerical example, suppose that a root is to be found for the equation $x^2 - 5x + 3 = 0$. For illustrative purposes, it will be assumed that each node will require one time unit and only the contents of the

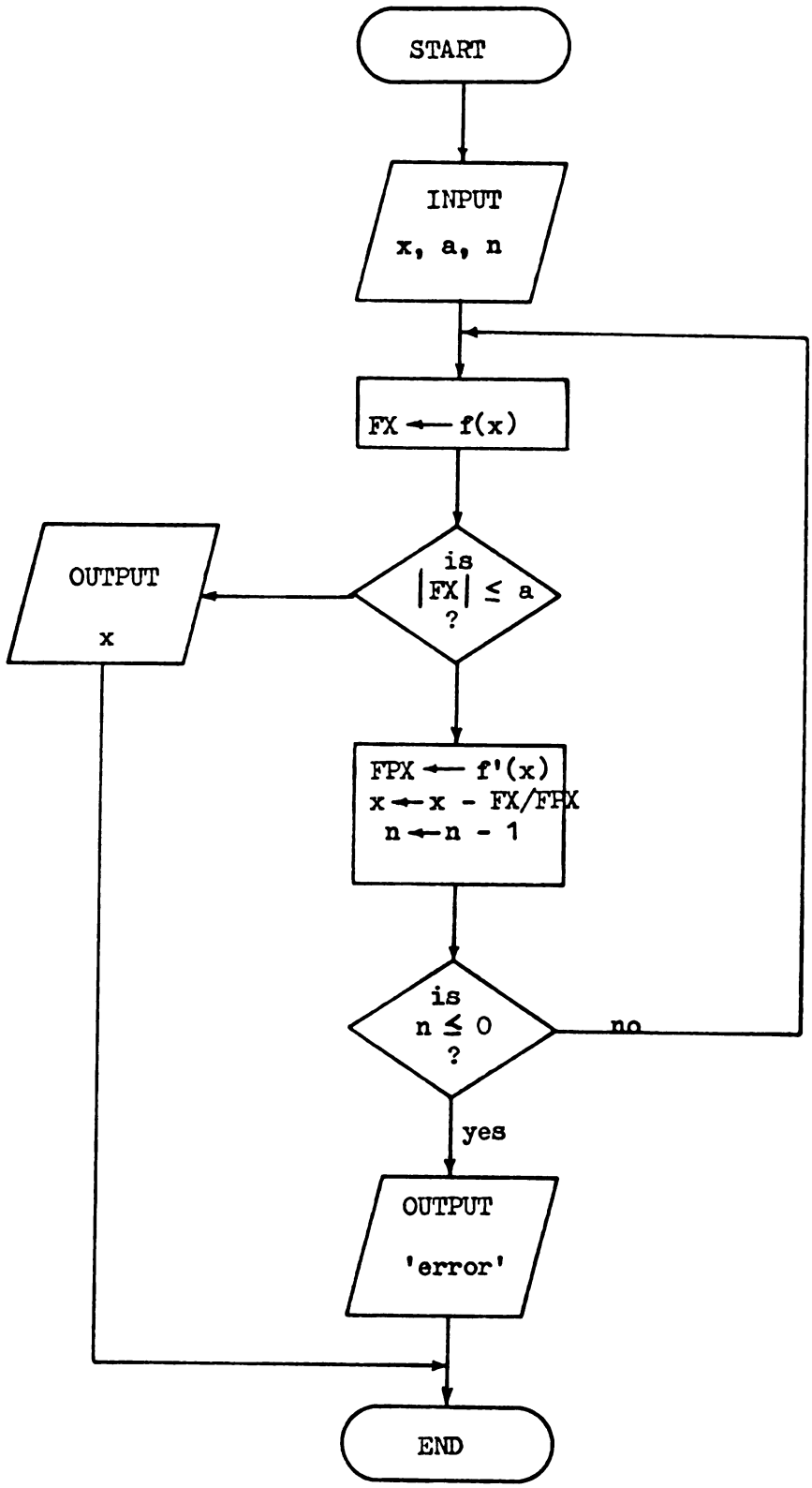
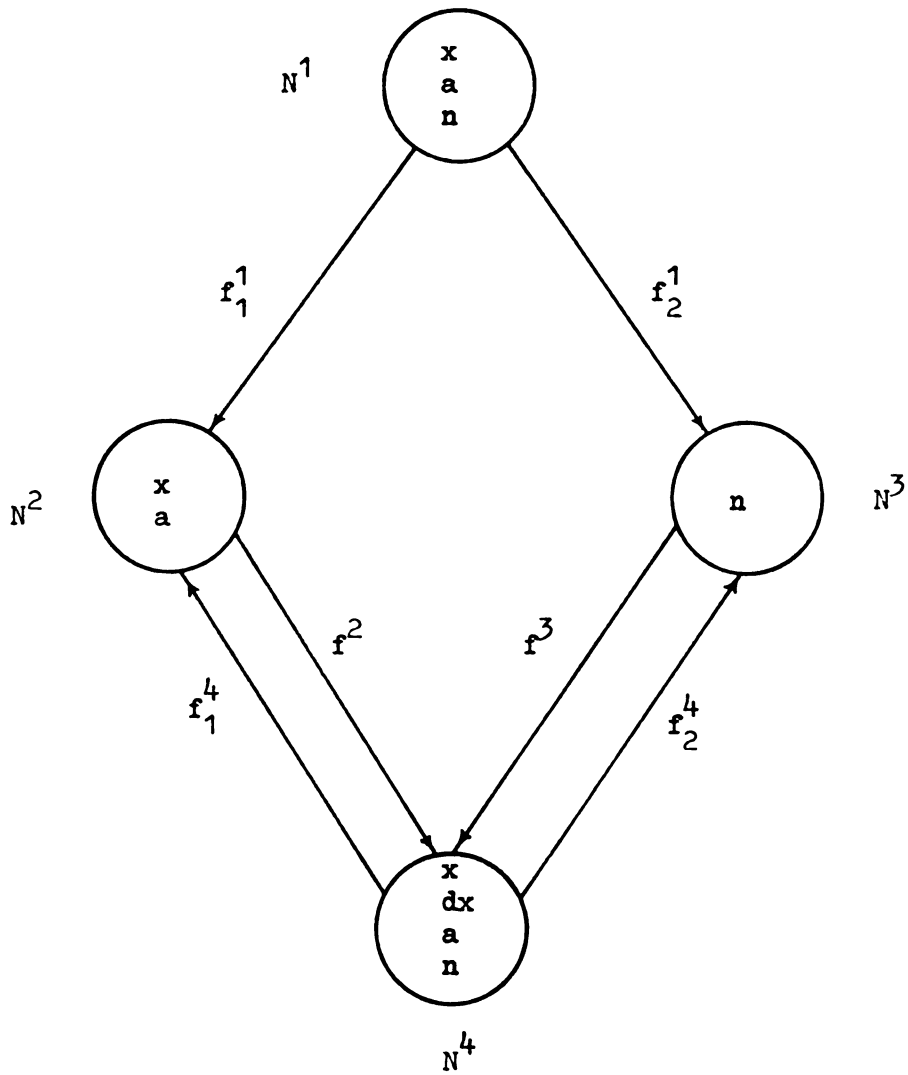


Figure 2.3 Conventional flowchart for Newton-Raphson algorithm



The mappings are defined as follows:

$$f_1^1(x, a, n) = (x, a)$$

$$f_2^1(x, a, n) = (n)$$

$$f^2(x, a) = \left(x, \frac{f(x)}{f'(x)}, a, ?\right)$$

$$f^3(n) = \begin{cases} (?, ?, ?, n-1) & \text{if } n > 0 \\ (?, ?, ?, ?) & \text{otherwise} \end{cases}$$

$$f_1^4(x, dx, a, n) = \begin{cases} (x-dx, a) & \text{if } |dx| > a \\ (x, ?) & \text{otherwise} \end{cases}$$

$$f_2^4(x, dx, a, n) = (n)$$

Figure 2.4 Program Structure for Newton-Raphson algorithm

data vector of a node will be displayed. The contents of set U will not be shown since it remains constant.

TIME		Set K		Set A	
0	$N^{1,1}$	5	(x)		
		.001	(a)		
		10	(n)		
1	$N^{2,1}$	5	(x)		
		.001	(a)		
	$N^{3,1}$	10	(n)		
2	$N^{4,1}$	5	(x)		
		.6	(dx)		
		.001	(a)		
		9	(n)		
3	$N^{2.2}$	4.4	(x)		
		.001	(a)		
	$N^{3,2}$	9	(n)		
4	$N^{4,2}$	4.4	(x)		
		.095	(dx)		
		.001	(a)		
		8	(n)		
5	$N^{2,3}$	4.305	(x)		
		.001	(a)		
	$N^{3,3}$	8	(n)		
6	$N^{4,3}$	4.305	(x)		
		.002	(dx)		
		.001	(a)		
		7	(n)		
7				$N^{2,4}$	4.303 (x)
					? (a)

In this example, the root is found after three iterations and the result is placed in $N^{2,4}$. If there were no real roots to the equation or if the root was not found after n iterations, the last computed value would be placed in a copy of N^4. Copies of nodes N^2 and N^3 are in the set K

at the same time on several occasions; when this happens it does not matter which node is processed first. This is characteristic of the asynchronous behavior of a Program Structure.

The Program Structure described in this example could be made "more parallel" † by breaking node N^2 into two nodes thus potentially allowing more nodes to be processed in parallel. The maps in each of the new nodes could compute f(x) and f'(x) respectively. A third node could be used to compute dx. The Program Structure in Figure 2.5 is an example that shows how this might be done. Although this Program Structure contains more nodes than the previous one, the execution time would undoubtedly decrease if more than one processor is available to process the nodes. This would be due to the fact that more operations could be performed simultaneously.

The next example of a Program Structure is the implementation of a Finite State Machine. A definition of Finite State Machines is first presented.

Definition 2.16

A Finite State Machine M is a quintuple $M = (S, I, s, \delta, F)$ where

S is a finite set of states

I is a finite set of input symbols

s is the start state $(s \in S)$

δ is the transition (next state) function

F is a set of final states $(F \subseteq S)$

---

† Maximal Parallelism will be discussed in Chapter IV.

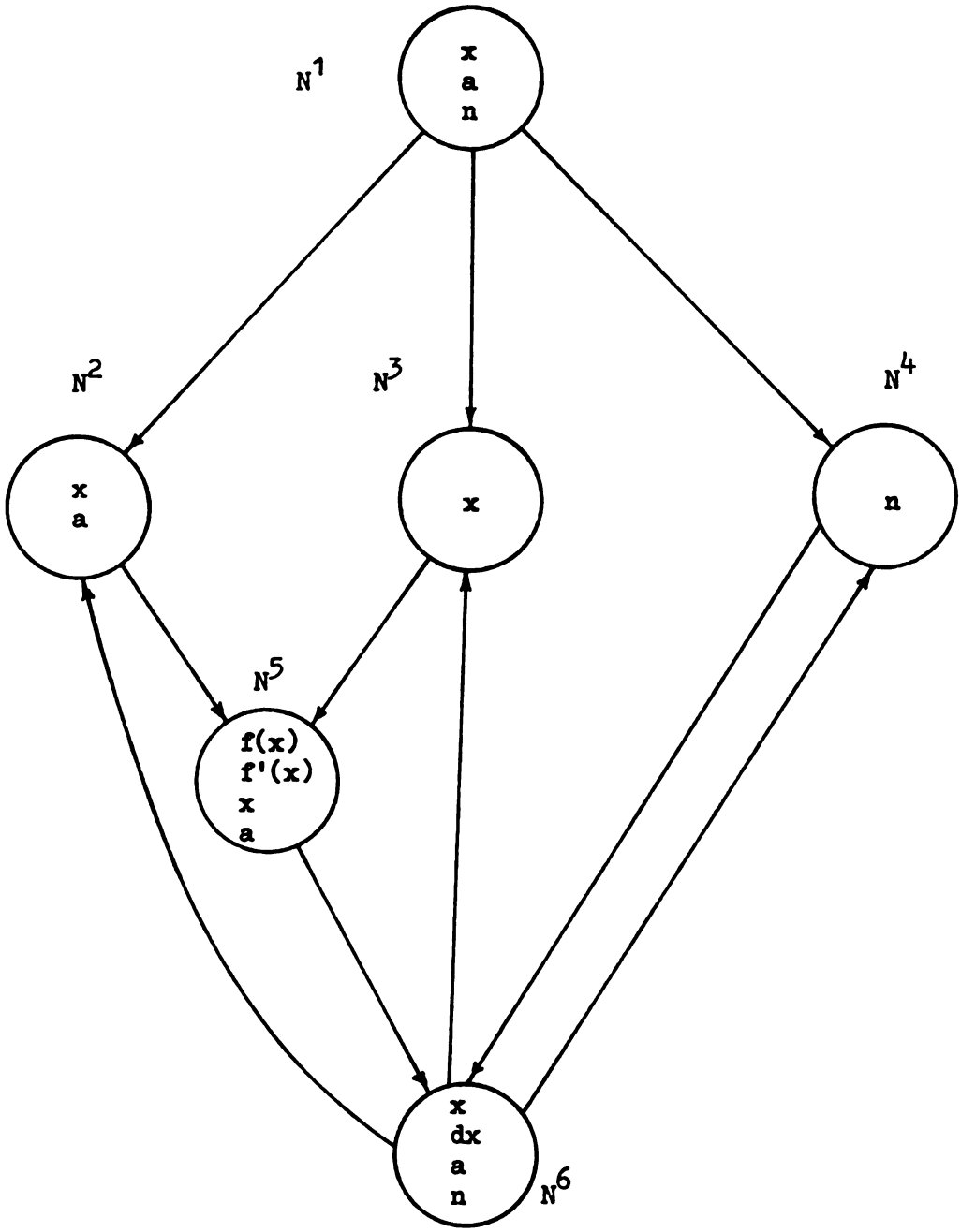


Figure 2.5 A "more parallel" representation of Figure 2.4

The input to the machine can be a sequence of symbols on a tape or simply a string of symbols. Since they are equivalent notions, the conventional flowchart will input from a tape and the Program Structure will input from a string. In the conventional flowchart in Figure 2.6 the tape is considered to be device one. The tape is assumed to be in position to read the first symbol prior to execution. Symbols are read from the tape one at a time, reading from left to right. Furthermore, each symbol x is read at most once and the tape will advance to the next symbol after each read. In this example, this process continues until the machine enters a final state or when there are no more symbols left to read. In the Program Structure of Figures 2.7 and 2.8, a string I is used rather than a tape so the symbols are read from the string in a similar fashion.

Note that $\delta(x,s) = ?$ if there is no next state associated with x and s. Also note that the null input is a valid input. If the machine ends in a final state, that state will be found in node N^8 of the Program Structure. If no node N^8 exists in set A when execution stops, then a final state was not reached.

2.3 Additional Features

No mention has been made of the concepts of input and output. This has been the case since Program Structures are inherently capable of input and output operations. The following definitions merely formalize this capability.

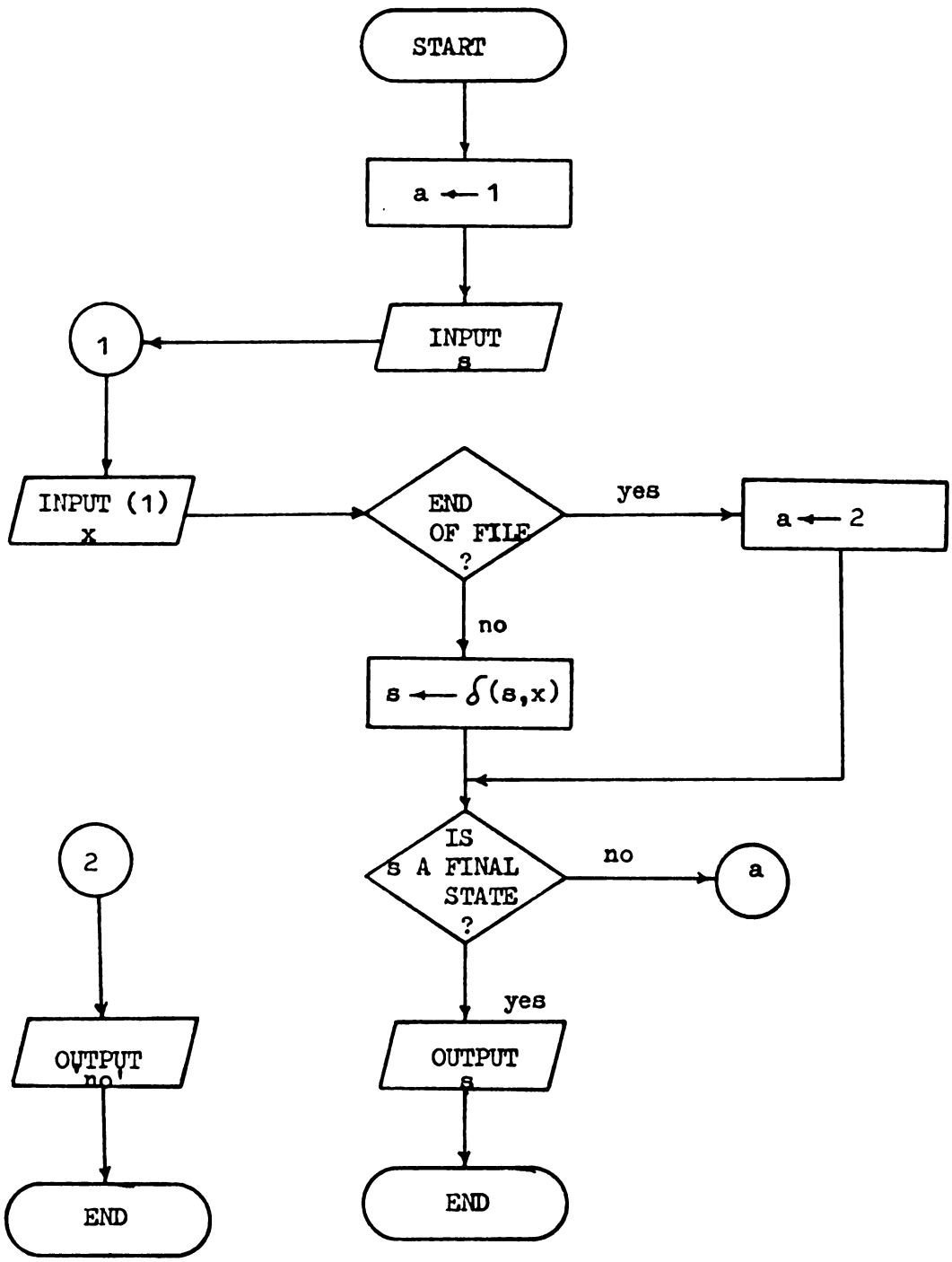


Figure 2.6 Implementation of Finite State Machine using a Conventional Flowchart

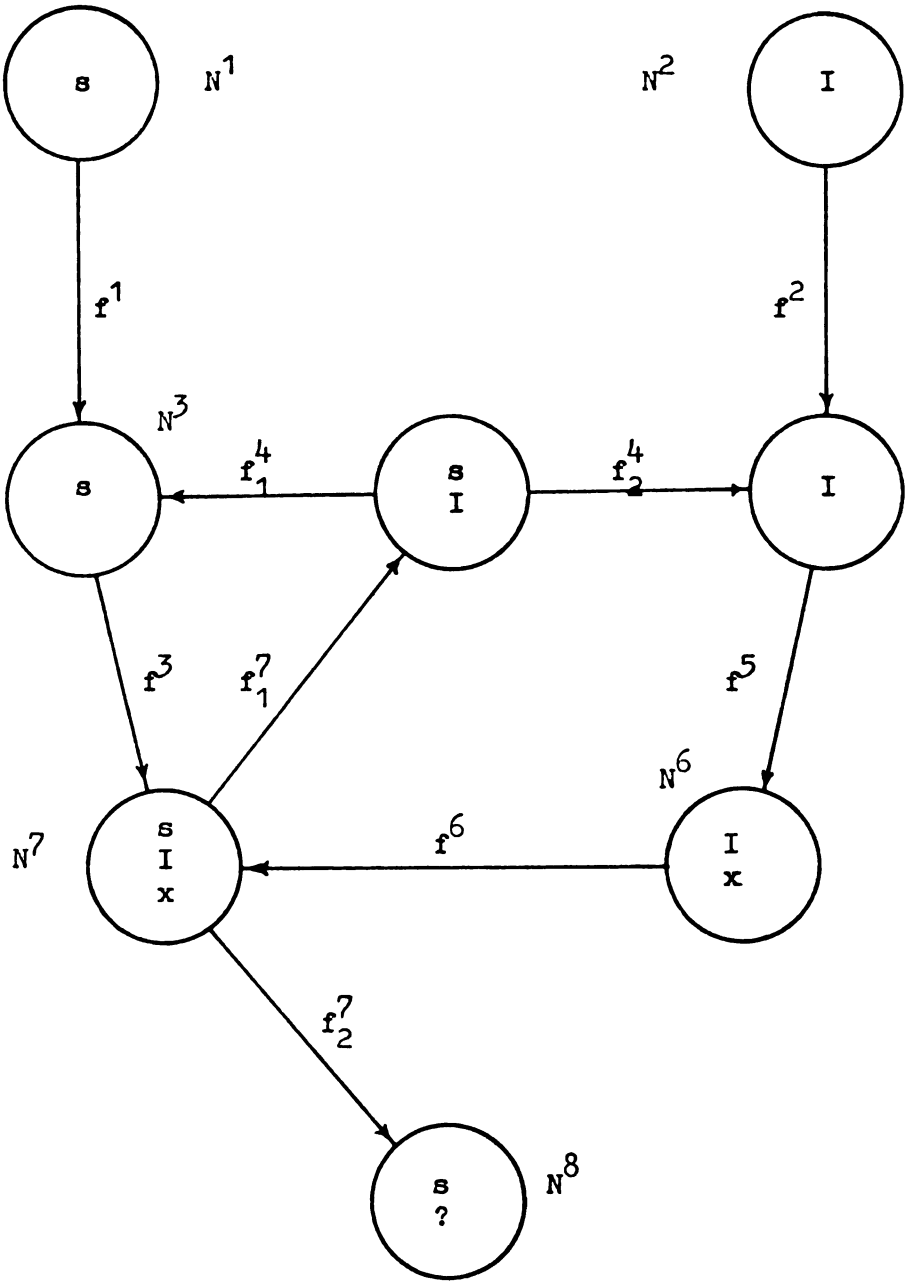


Figure 2.7 Implementation of a Finite State Machine using a Program Structure

$$f^1(s) = (s)$$

$$f^2(I) = (I)$$

$$f^3(s) = (s, ?, ?)$$

$$f_1^4(s, I) = (s)$$

$$f_2^4(s, I) = (I)$$

$$f^6(I, x) = (?, I, x)$$

$$f_1^7(s, I, x) = (\ \delta(s,x), I)$$

$$f_2^7(s, I, x) = \begin{cases} (s, ?) & \text{if } x \in F \\ (?, ?) & \text{otherwise} \end{cases}$$

Figure 2.8 Definition of mappings used in the Program Structure of Figure 2.7

Definition 2.17

The input to a Program Structure is a set I of nodes such that

$I = \{N^i \mid N^i \in K \text{ prior to execution of the Program Structure}\}$

In other words, the input to a Program Structure is the set of all nodes which are known before any node is processed. The output from a Program Structure is defined next.

Definition 2.18

The output from a Program Structure is a set O of nodes $N^{i,j}$ such that

1. $N^{i,j} \in A$ when execution is terminated
2. $N^{i,j}$ is the last copy of N^i
3. N^i is specified prior to execution

To give an example of input and output nodes a sequential table search algorithm is implemented. Given a table T which contains n elements and a key x, determine whether or not x is in T and if it is, output the position (or index i) that it occupies in the table T.

Figure 2.9 shows a Program Structure which performs table lookup. Nodes N^1 and N^2 are input nodes and nodes N^3 and N^4 are used as output nodes. If the table search is successful the position of x will be found in the last copy of N^4. If x is not in the table T, then the value n+1 will be found in the last copy of N^3.

In other words: $I = \{N^1, N^2\}$

$O = \{N^3, N^4\}$

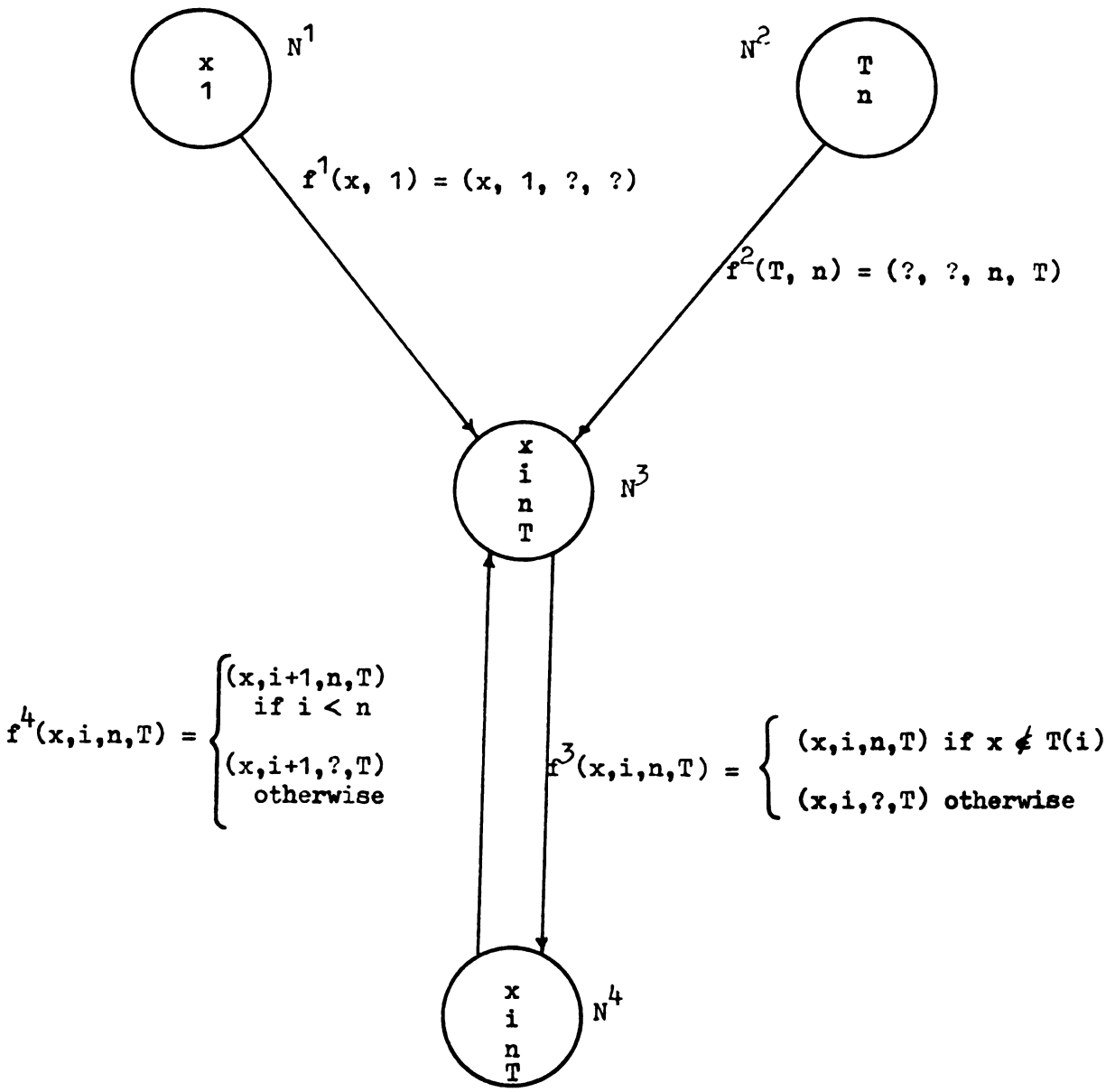


Figure 2.9 Use of a Program Structure to perform table lookup

This means that the output from a Program Structure can be any arbitrary set of nodes. These nodes are always the last copies of nodes residing in set A when execution is terminated.

There are no unconditional branching features in the nodes of a Program Structure. However, there is a capability for performing looping operations or iterations which has been apparent in most of the preceding examples. This capability involves the notion of cycle, which is defined next.

Definition 2.19

A Program Structure contains a cycle if there exist nodes N^i and N^j such that when N^i becomes known then N^j will become known only if N^i is processed and copies of N^i will become known only if N^j is processed, and so on. For example: $N^{i,1} \Rightarrow N^{j,1} \Rightarrow N^{i,2} \Rightarrow N^{j,2} \Rightarrow N^{i,3} \Rightarrow \ldots..$ †

The concept of cycle here is analogous to cycles in graph theory and loops in flowchart schemata. This analogy is depicted in Figure 2.10 and in Figure 2.11.

Given any Program Structure, it is not immediately apparent whether or not execution will halt in a finite number of steps. The next definition introduces terminology that will be used to discuss this problem.

Definition 2.20

A Program Structure is said to be solvable if execution of that Program Structure terminates in a finite number of steps. Otherwise, a Program Structure is said to be unsolvable.

---

† $N^{i,j} \Rightarrow N^{k,m}$ means $N^{k,m}$ will become known only if $N^{i,j}$ is processed.

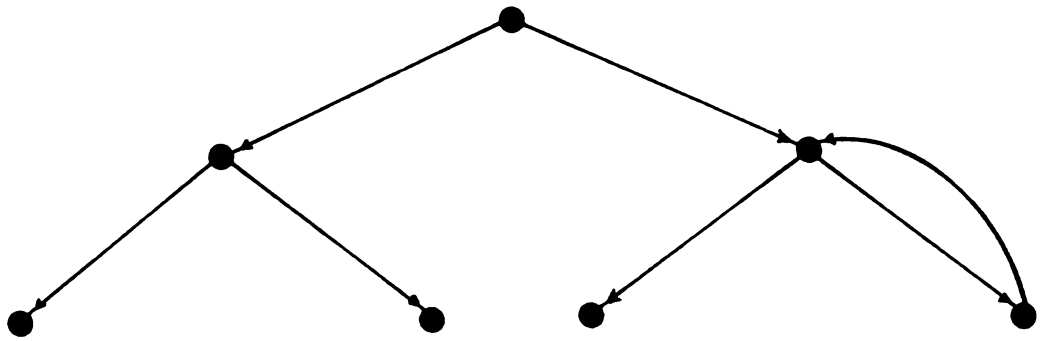


Figure 2.10 Directed Graph containing a cycle

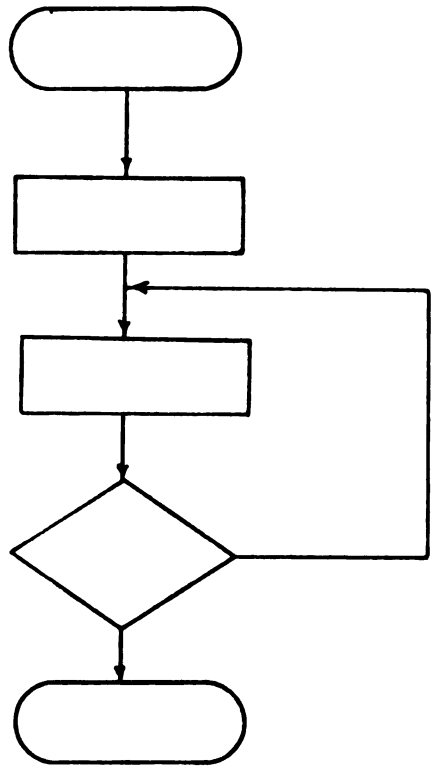


Figure 2.11 Flowchart Schema containing a loop

In general, Program Structures may not necessarily be solvable, but there is one type that is.

Proposition 2.2

Any Program Structure which is cycle-free (contains no cycles) is solvable.

Proof:

Assume that there are a total of n nodes in the Program Structure. Since the Program Structure is cycle-free, this means that each node will be processed at most one time. Hence, the Program Structure will terminate execution after processing a maximum of n nodes.

CHAPTER III

COMPUTABILITY

3.1 Implementation of Flowchart Schema

Most of the preceding discussion has concentrated on the internal aspects of Program Structures. In this section, it is shown that Program Structures can be used to implement any Flowchart Schema as well as incorporating any inherent parallelism in the Flowchart Schema.

Definition 3.1

A Flowchart Schema (FS) is a 4-tuple (M, N, p, f) where

M is a finite set of memory cells

$N = \{s, e, B, A\}$ is a finite set of nodes where

s is the starting node (i.e., the node to be processed first)

e is the ending node (i.e., the last node to be processed)

$B = (b_1, b_2, b_3, \ldots.)$ is an n-tuple of branching nodes

$A = (a_1, a_2, a_3, \ldots.)$ is an m-tuple of assignment nodes and each assignment node modifies the value of one or more memory cells.

p is an n-tuple of predicates corresponding with the elements of B

$f: N \times M \rightarrow N$ where f is a control function which specifies which node is to be processed next and where

$f(x,d) = y$ where $y \in N - e$ (d means don't care)

and $x = s$ or $s \in A$

$$f(x, p_i(m)) = \begin{cases} y_1 \in N - e \text{ where } x \in B,\ m \in M,\ p_i(m) \text{ is T} \\ y_2 \in N - e \text{ where } p_i(m) \text{ is F} \end{cases}$$

Theorem 3.1

A Program Structure can implement any Flowchart Schema.

Proof:

Since only known nodes can be processed in a Program Structure, the data vector of each node will be M. The input node will have every element in its data vector specified. Each node N^i in the Program Structure that corresponds to an assignment node will have one mapping and it will be of the form $f^i:D^i \rightarrow D^j$ and where $f^i(M) = M'$. M' denotes M after assignments have been made. Each node N^i in the Program Structure that corresponds to a branching node will have two mappings of the form

$$f_1^i:D^i \rightarrow D^j \text{ where } f_1^i(M) = \begin{cases} M' \text{ if } p_i(m) \text{ is T} \\ ? \text{ otherwise} \end{cases}$$

$$f_2^i:D^i \rightarrow D^k \text{ where } f_2^i(M) = \begin{cases} M' \text{ if } p_i(m) \text{ is F} \\ ? \text{ otherwise} \end{cases}$$

The mapping for the ending node N^i would have the form $f^i:D^i \rightarrow D^i$ where $f^i(M) = ?$. Hence, any Flowchart Schema can be rewritten as a Program Structure.

In practice, the implementation of a specific FS could be much less cumbersome. In general, nothing would be gained by such an implementation unless some degree of parallelism were introduced.

Definition 3.2

The set of all memory cells referenced by assignment node a_i will be denoted by C_i.

Proposition 3.2

If a_i and a_j are any two sequential assignment nodes of a Flowchart Schema and $C_i \cap C_j = \emptyset$, then a_i and a_j can be processed in parallel without altering the final outcome.

Proof:

Since the processing of one node does not affect the other, the nodes may be processed in any order.

Thus any parallelism detected at the node level can be implemented in the Program Structure. In Figure 3.1(a) a_i and a_j are sequential assignment nodes of a Flowchart Schema where $C_i \cap C_j = \emptyset$. When the Program Structure is created, this parallelism is incorporated by changing several mappings as indicated by Figure 3.1(b).

3.2 Equivalence Classes of Data

In this section, the set of all inputs are placed into equivalence classes. To accomplish this, some preliminary definitions are first made.

Definition 3.3

A node is called a junction if it must receive information from two or more nodes in order to become known.

Figure 3.2 provides an illustration of a node that is a junction. In case node N^5 is a junction since it must receive information from both nodes N^3 and N^4 before it will become known. Notice also that if node N^i is a junction, then $\#(D^i) \geq 2$. However, the converse is not true.

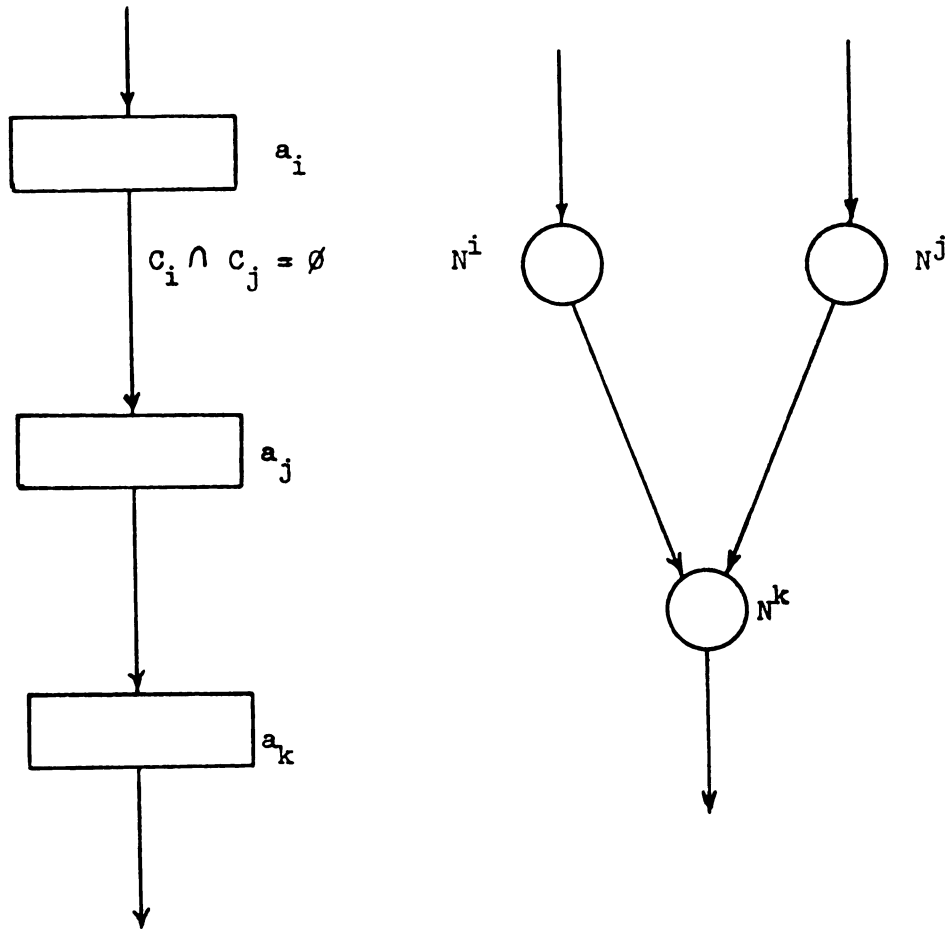


Figure 3.1 (a) Segment of a Flowchart Schema which can be processed in parallel

(b) Implementation of parallelism in (a)

Definition 3.4

If node N^i is a junction, then it is denoted by $\$N^i$.

Definition 3.5

Given an input set I, an execution sequence is the series of nodes which must be processed in order to produce values in a given output set O.

Definition 3.6

Two execution sequences are said to be equivalent if they always produce identical values in the output set O from a given input set I.

The processing of a cycle can be regarded as the processing of a single node when relating to execution sequences. For notational purposes, if a series of nodes N^i, ... , N^j are to be processed n times (a cycle), then that cycle can be referred to as $\overline{N^i, \ldots, N^j}$.

Proposition 3.3

If N^i and N^j are non-junction nodes that precede a junction $\$N^k$, then the following execution sequences are equivalent:

(1) N^i, N^j, $\$N^k$

(2) N^j, N^i, $\$N^k$

Proof:

If N^i and N^j are both known nodes, then the order in which they are processed is immaterial.

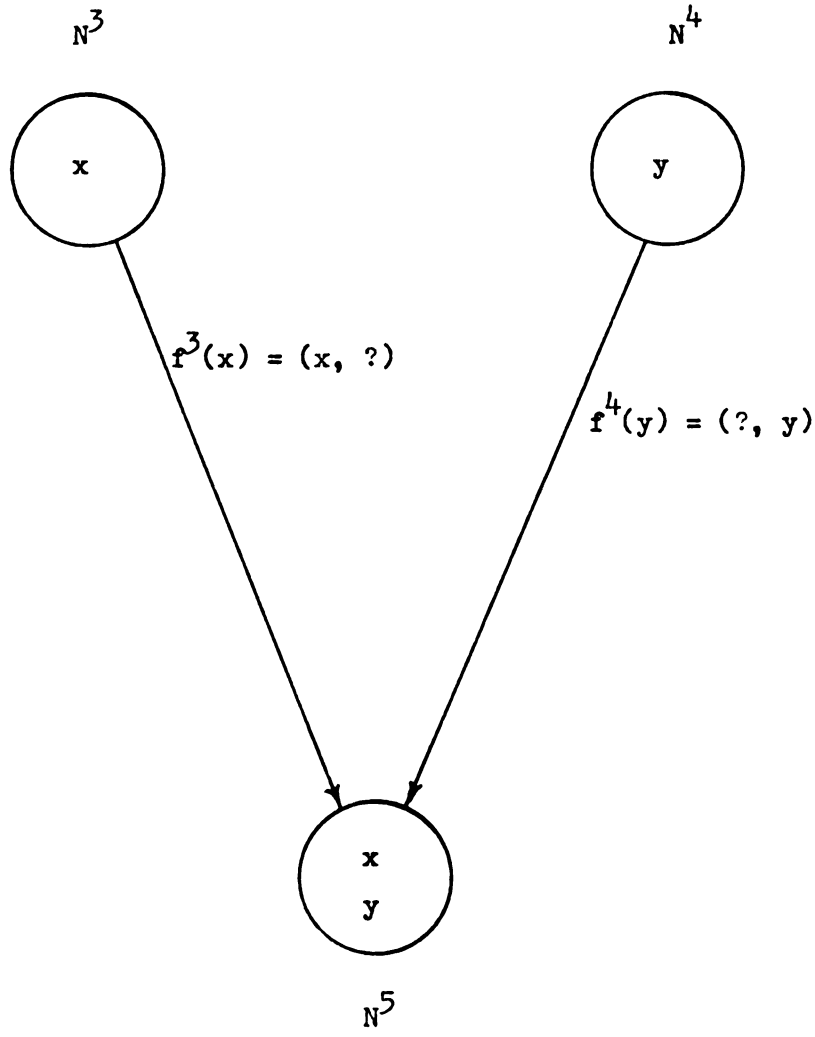


Figure 3.2 Example of a node that is also a junction

The next definition helps in the discussion of equivalence classes of input data which follows shortly.

Definition 3.7

R is the relation on inputs which has the meaning "has an equivalent execution sequence as".

Thus for any two inputs I^i and I^j to a Program Structure, I^i R I^j means that the execution sequence generated by I^i is equivalent to the execution sequence generated by I^j.

Theorem 3.4

The relation R is an equivalence relation if and only if there is a unique output set associated with each input set.

Proof:

Follows immediately from definition 3.7 and the definition of an equivalence relation.

Definition 3.8

$$[I^i] = \{I^j \mid I^i \; R \; I^j\}$$ where R is an equivalence relation.

This means that $[I^i]$ is the set of all inputs that have equivalent execution sequences as I^i. This also means that each input to a Program Structure is placed in one and only one such set.

That an equivalence relation partitions its field is a well known property. Consequently, the collection of sets $[I^i]$ forms a partition on the set of all inputs to a Program Structure.

Pictorially, the set of all inputs can be regarded as a data space and the individual inputs can be visualized as data points. The disjoint collection of data points can be thought of as partitions. A similar type of analogy can be drawn for the set of all outputs. The major difference is that the set of all outputs can be rigorously predefined.

It has been an easy matter to show that the set of inputs can be partitioned. However, it is difficult to show how these partitions can be formed. The following comment and example are intended to add insight to this problem. From the definition, it follows immediately that the number of equivalence classes of an input set is the number of distinct (non-equivalent) executions sequences.

An example is now being given that shows the appearance of an execution sequence as well as its meaning. The example uses the Newton-Raphson root finding algorithm of Figure 2.4.
There are three distinct execution sequences possible in that Program Structure:

1. $N^1\ N^2\ N^3$ - if this execution sequence is obtained then the input value of n was 0
2. $N^1\ \overline{N^2\ N^3\ \$N^4}\ N^3$ - If this execution sequence is obtained then a root was not found after n iterations
3. $N^1\ \overline{N^2\ N^3\ \$N^4}\ N^2$ - if this execution sequence is obtained then a root was found within n interations

Even though there are an infinite number of possible inputs to the Program Structure, all of these inputs can be partitioned into three equivalence classes. The three equivalence classes are:

1. $[x, n \leq 0]$ where x is any value
2. $[a, n > 0]$ where a is an element of the set of all inputs that will not result in finding a root
3. $[b, n > 0]$ where b is an element of the set of all inputs that will result in finding a root

In this example, it is not very difficult to partition the set of inputs. The difficulty arises when a specific input is to be assigned to an equivalence class. Obviously, there are two conditions which determine which equivalence class an input is to be placed into: Nodes which contain two or more mappings and nodes which contain "complex" mappings. An example of a complex mapping is as follows:

$$f^5(x,y) = \begin{cases} (x,\ x+y) & \text{if } x < 3 \\ (?,\ x+y) & \text{otherwise} \end{cases}$$

This makes the following proposition obvious.

Proposition 3.5

Every cycle which terminates in a finite number of steps has at least one node which contains a complex mapping.

The problem of partitioning input data would be greatly simplified if Program Structures contained no cycles. In the following proposition, that assumption is made.

Proposition 3.6

If a cycle free Program Structure contains a total of n nodes and

has no junctions, then the maximum number of possible execution sequences

is $$1 + \sum_{i=1}^{n-1} \binom{n-1}{i} .$$

Proof:

In a cycle free Program Structure no node is processed more than once. Since there must be at least one input node, i.e., a node that is known prior to execution, then there is at least one node to be processed which leaves n-1 nodes left to form execution sequences. The number of permutations which can be formed is (n-1)! but the number of execution sequences is less than this because an execution sequence and its permutation are regarded as the same. To properly count the maximum number of possible execution sequences, combinations of n-1 nodes must be calculated for the possibilities of processing 1, 2, 3, ... n-1 additonal nodes. To state this more formally, the maximum number of possible execution sequences is:

$$1 + \binom{n-1}{1} + \binom{n-1}{2} + \ldots\ldots + \binom{n-1}{n-1}$$

3.3 An Analysis Formalism

There are a number of aspects of Program Structures which can be investigated and analyzed. However, due to the graph-like appearance of Program Structures, it becomes difficult to perform any analysis on them. In order to circumvent this difficulty, a formalism is now introduced.

Definition 3.9

The index set of all nodes will be denoted by $N = \{1, 2, 3, \ldots, n\}$

Definition 3.10

The classificiation set for all nodes will be denoted by

$$C = \left\{ x \mid 0 \leq x \leq 1 \text{ where } \begin{array}{l} x < 1 \Rightarrow \text{the node is unknown} \\ \text{and} \\ x = 1 \Rightarrow \text{the node is known} \end{array} \right\}$$

Definition 3.11

The set of test values on the mappings of any particular node will be denoted by

$$T = \{0, 1, 3, 4, \ldots\ldots\}$$

A series of transition statements using the above definitions can be made which describe the behavior of some Program Structure. In the illustration below, when node N^3 is processed, it causes node N^4 to become known.

This action can be described by the transition statement

$$(3, 1, 0) \rightarrow (4, 1, 0)$$

Definition 3.12

The transition statement $(n, c, t) \rightarrow (n', c', t')$ describes the action taken when node N^n is processed and where

$n, n' \in N$

$c, c' \in C$

$t, t' \in T$

Each 3-tuple of a transition statement will be referred to as a state.

Since unknown nodes are not processed, they must not affect the contents of any other node. The following definition specifies the transition statement for that possibility.

Definition 3.13

If node N^n is unknown, then $(n, 0, d) \rightarrow (n, 0, d)$ where the symbol d represents "don't care".

The above definition can be made more general by testing the classification of a particular node.

Definition 3.14

If $x < 1$ for any node N^n, then $(n, x, d) \rightarrow (n, x, d)$

A more difficult problem arises when transition statements are used to describe the transmittal of information to a junction or the action by a node with a multiple mapping. For example, in Figure 3.3 node N^8 will become known only if it receives information from both

nodes N^6 and node N^7. One possible way of describing this notion is by using the following two transition statements:

$$(6, 1, d) \longrightarrow (8, 1, d)$$

$$(7, 1, d) \longrightarrow (8, 1, d)$$

Clearly this is unacceptable since it implies that either node N^6 or node N^7 can make node N^8 become known.

Another possible way of describing this action is by using the following two transition statements:

$$(6, 1, d) \longrightarrow (8, 0, d)$$

$$(7, 1, d) \longrightarrow (8, 0, d)$$

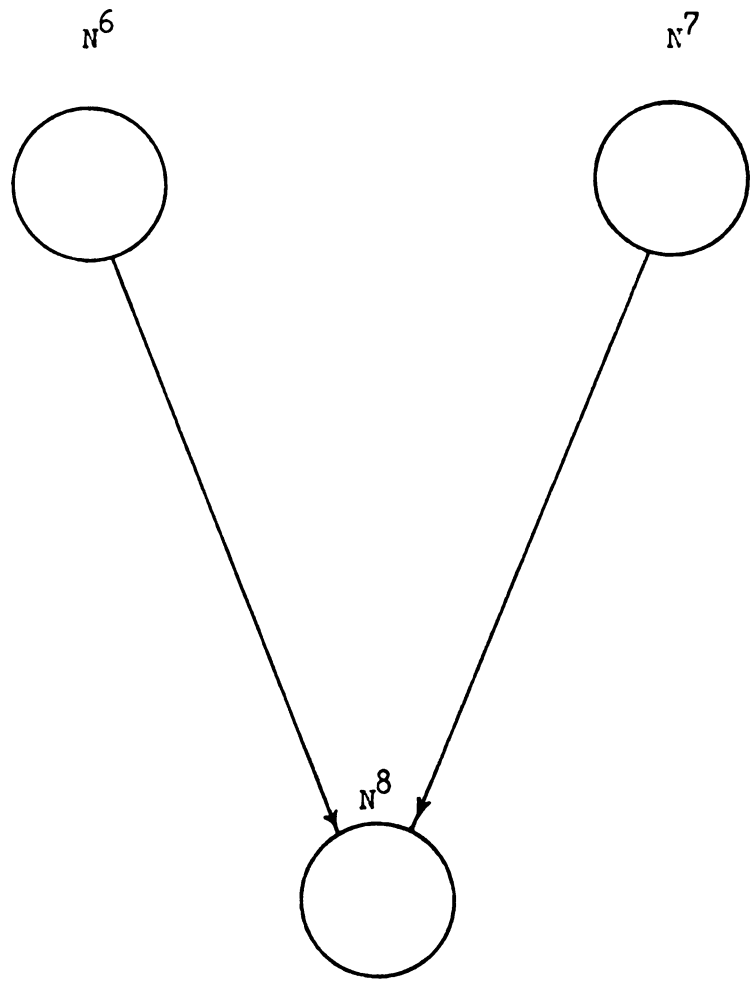
This is perhaps a better solution, but it too is unacceptable since it fails to communicate any information about the behavior of the nodes

To represent a junction, notation will be used that is similar to "conditional expressions".

Definition 3.15

If node N^i is a junction and it must receive information from nodes N^j and N^k to become known, then this action is described by the following transition statement: $(j, 1, d) \wedge (k, 1, d) \longrightarrow (i, 1, d)$ where $\wedge$ denotes logical and. In other words, (i, 1, d) will be realized only if the condition indicated is true.

Figure 3.4 gives an example showing how a transition statement is used to handle a multiple mapping.



$$(6, 1, d) \wedge (7, 1, d) \longrightarrow (8, 1, d)$$

Figure 3.3 Use of a transition statement to describe the transmittal of information to a junction

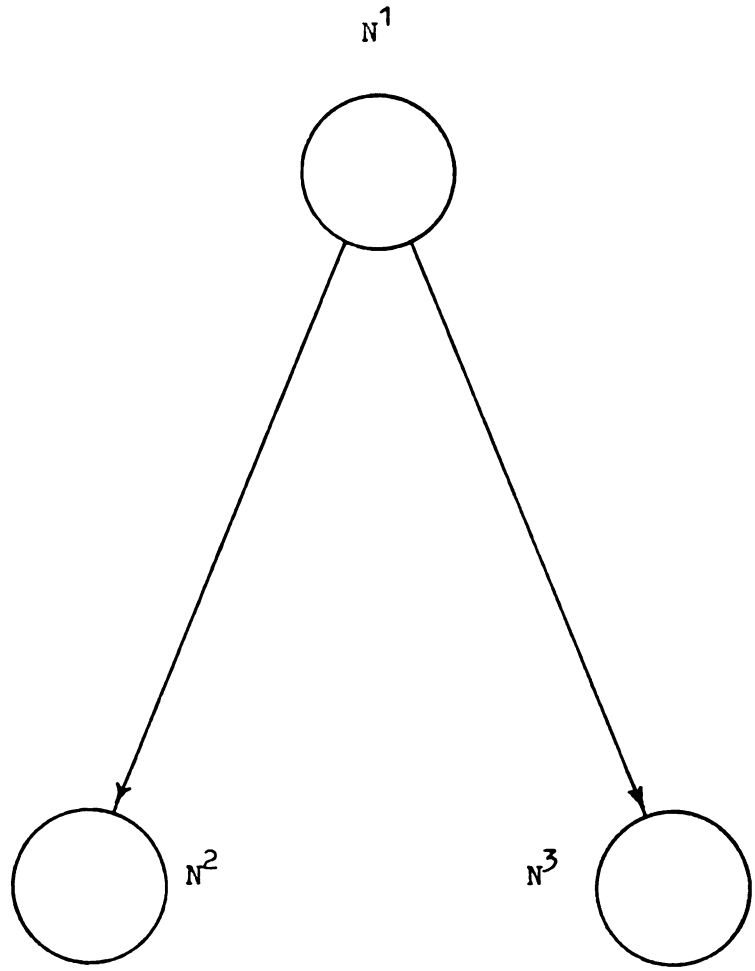


Figure 3.4 Use of a transition statement to describe the action taken by a node containing a multiple mapping

Definition 3.16

If node N^i contains a multiple mapping to nodes N^j and N^k then the action taken by N^i can be described by the following transition statement:

$$(i, 1, d) \longrightarrow (j, d, d) \wedge (k, d, d)$$

It is assumed that definitions 3.14 and 3.15 can be generalized to handle more than two nodes.

Figure 2.4 displays a Program Structure which is used to implement the Newton-Raphson root finding algorithms. The following transition statements describe the behavior of that Program Structure.

$$(d, 0, d) \longrightarrow (d, 0, d)$$
$$(1, 1, d) \longrightarrow (2, 1, d) \wedge (3, 1, d)$$
$$(2, 1, d) \longrightarrow (4, 0, 2)$$
$$(3, 1, 1) \longrightarrow (4, 0, 3)$$
$$(3, 1, 0) \longrightarrow (4, 0, 0)$$
$$(4, 0, 2) \wedge (4, 0, 3) \longrightarrow (4, 1, d)$$
$$(4, 1, 1) \longrightarrow (2, 1, d) \wedge (3, 1, d)$$
$$(4, 1, 0) \longrightarrow (3, 0, d)$$

Note that the third value of the triple can be used to describe the behavior of a complex mapping. A choice of "next states" is possible depending on that test value.

Figure 3.5 shows how a transition diagram can be drawn based on the above transition statements.

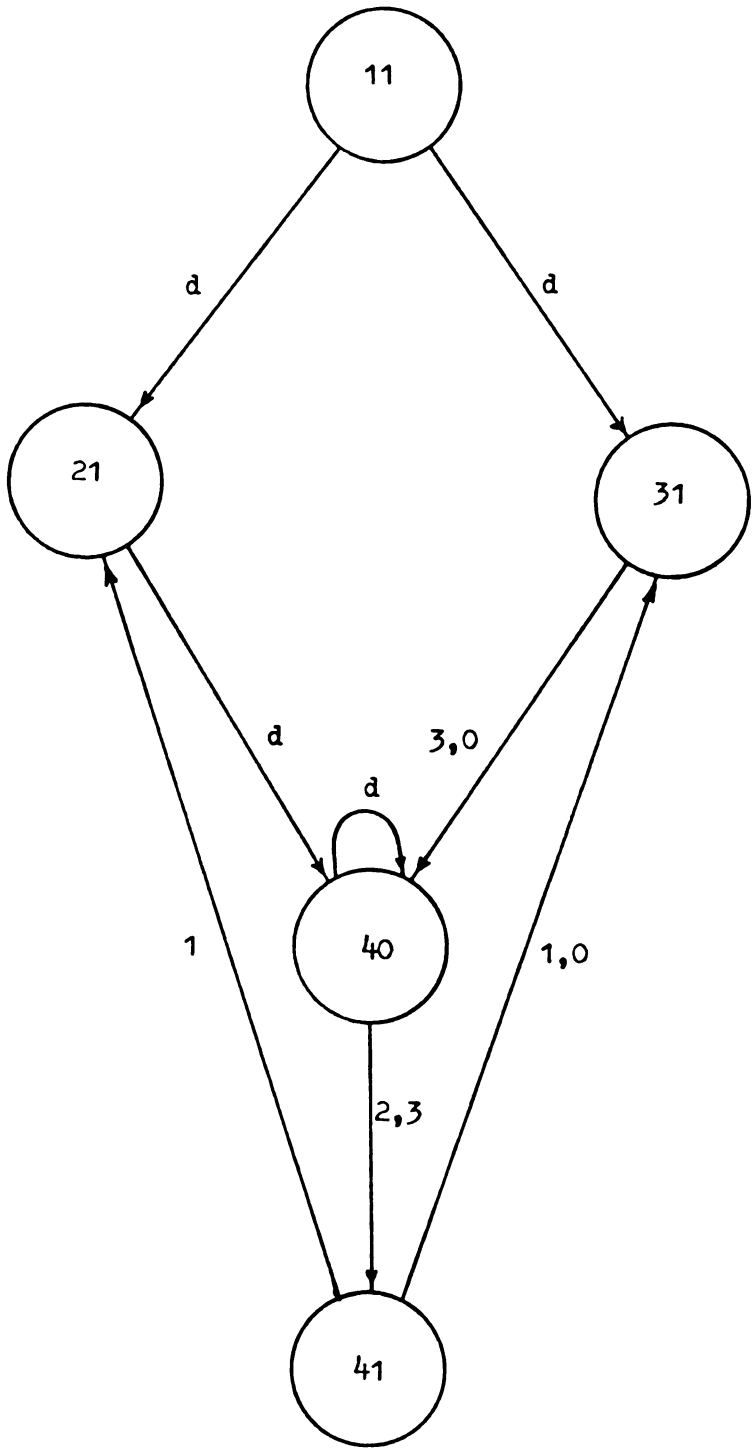


Figure 3.5 A transition diagram made form transition statements of Program Structure in Figure 2.4

CHAPTER IV

CONDITIONS FOR REPEATABILITY

The purpose of this chapter is to investigate ways in which the uniqueness of the output of a Program Structure can be assured. In addition to discussing determinism and repeatability, maximal parallelism is studied.

Definition 4.1

A Program Structure is said to be deterministic if the execution sequence generated by a particular set of values in the input set is unique. For the purpose of this definition, execution sequences $N^1 N^2 N^3$ and $N^1 N^3 N^2$ are considered to be different even though one execution sequence is a permutation of the other.

It should be obvious that in most cases Program Structures are not deterministic. Determinism is not necessarily a desirable property of Program Structures. In fact, a deterministic Program Structure has very little capability for parallel processing.

Proposition 4.1

Any Program Structure can be modified so that it is deterministic.

Proof:

If there is more than one input node, rewrite the Program Structure so that there is only one input node. If any node contains two or more mappings, then rewrite the mappings so that only one node can become known as a result of processing any one node.

A lesser form of determinism, call repeatability, is frequently the more desirable property of Program Structures because parallelism is achieved and yet the end result is unique.

Definition 4.2

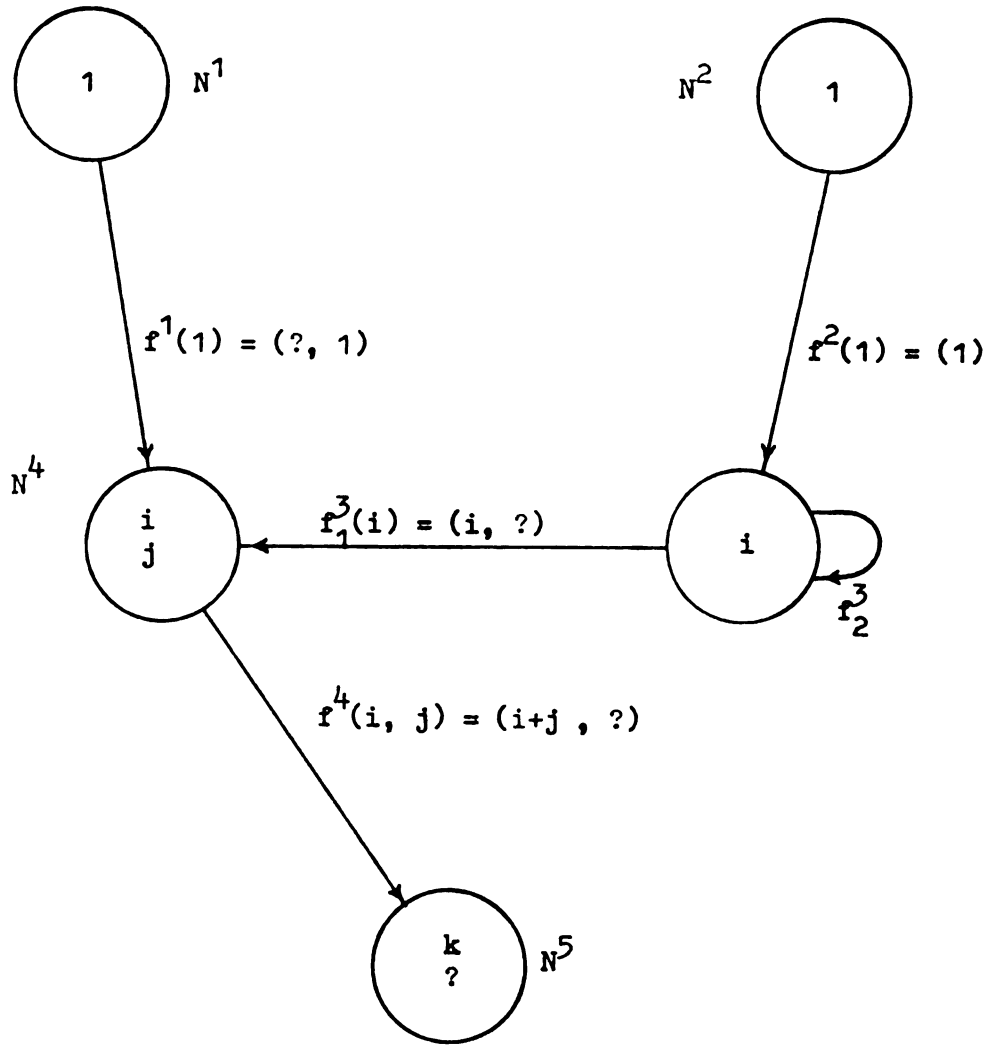
A solvable Program Structure is said to be repeatable if the values in the output set are always identical regardless of the number of times the Program Structure is executed using a given set of values in the input set.

The notion of repeatability has been called "weakly determinate" (14), "output functional" (34), or even "deterministic" (43) in other models. Clearly, a deterministic Program Structure is repeatable.

Program Structures are not always repeatable as the example in Figure 4.1 illustrates. Although the purpose of that Program Structure may not be clear, it is clear that the output node N^5 may contain the value 2, 3, 4, 5, 6, or 7 depending on when nodes N^1 and N^4 are processed. The reason for this apparently chaotic situation is that it is possible to have up to six copies of node N^4 in the active set A at some instant in time. If node N^1 is processed after these six copies are placed in A, then there are six known nodes - all copies of node N^4. The order in which these known nodes are processed will determine the final output value in node N^5.

However, all is not lost. The Program Structure in Figure 4.1 can be made repeatable by changing one mapping:

$$f_1^3(i) = \begin{cases} (?, ?) \text{ if } i < 6 \\ (i, ?) \text{ otherwise} \end{cases}$$



$$I = \{ N^1, N^2 \}$$

$$O = \{ N^5 \}$$

$$f_2^3(i) = \begin{cases} (i+1) \text{ if } i < 6 \\ (?) \text{ otherwise} \end{cases}$$

Figure 4.1 Example of a non-Repeatable Program Structure

The effect of the above mapping is that node N^3 passes information to node N^4 only when the cycle is completely finished. Non-repeatability frequently occurs when a node receives information (which makes it known) both from a cycle and a node outside the cycle. This suggests a condition to be applied which can make all Program Structures repeatable. If a cycle could be "unraveled" so that execution of a cycle could be regarded as execution of a series of nodes which then passes information on to another node, then that Program Structure can be thought of as having no cycles.

The following defintion suggests a type of cycle which is very useful in developing repeatable Program Structures.

Definition 4.3

A closed cycle is a cycle such that a node outside the cycle will receive information from that cycle at most once each time the cycle is executed. Also, there is one node in the cycle which contains a complex mapping and this will determine whether the cycle will be executed once again or whether information is to be passed to a node outside the cycle.

A Program Structure which contains only closed cycles behaves like a cycle-free Program Structure. Although use of closed cycles can be helpful in creating repeatable Program Structures, it is not in itself a complete assurance of repeatability. For example, in Figure 4.2 a cycle-free Program Structure is displayed that is not repeatable. The output value in node N^4 will depend on the order in which nodes N^1, N^2, and N^3 are executed. The execution sequence N^1 N^3 N^2 N^3 will

produce a different output value than the execution sequence $N^2\ N^3\ N^1\ N^3$. Note also in this example that the Program Structure does not contain a junction even though node N^3 may graphically appear as such.

This suggests a condition to be applied to cycle-free Program Structures which will guarantee repeatability.

Theorem 4.2

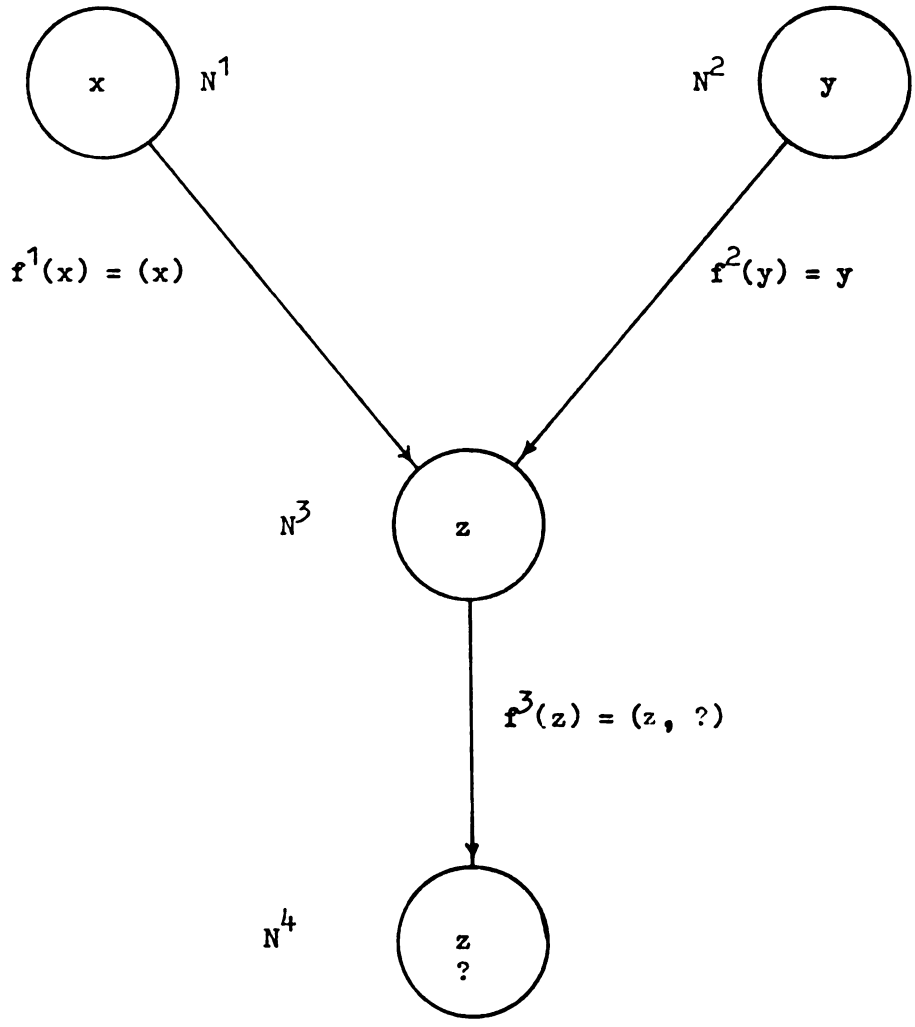
A cycle-free solvable Program Structure (or one that contains only closed cycles) is repeatable if for any node N^i where $f^j : D^j \rightarrow D^i$ and $f^k : D^k \rightarrow D^i$, then $f^j(D^j)$ & $f^k(D^k) \neq f^j(D^j)$.

Proof:

To guarantee repeatability of a cycle-free Program Structure, the theorem simply states that no two mappings may map values into the same element of a data vector.

Case 1. The Program Structure contains no junctions. Each time a node is processed, it causes 0 or more nodes to become known. Since there is no communication or sharing of information between the nodes, the set of output values is not affected by the order in which the nodes are processed.

Case 2. The Program Structure contains one or more junctions. Because of the condition stated in the theorem, the information that must be placed in a junction to make it become known is unique regardless of the order in which the nodes that precede the junction are processed. This has the effect of ensuring that certain nodes will be



$I = \{ N^1, N^2 \}$ assume $x \neq y$

$O = \{ N^4 \}$

Figure 4.2 Example of a cycle-free, junction-free Program Structure which is not repeatable.

processed before the junction itself is processed. Consequently, this will have no effect on the set of values placed in the output set O.

Note that in both Case 1 and Case 2 above there is only one execution sequence for any given set of values in the input set I. This is a necessary, but not a sufficient condition for repeatability.

Although the above theorem adds insight into the nature of repeatability of Program Structures, it contains very restrictive conditions. So that repeatability can be more easily described, the next concept is introduced.

Definition 4.4

A critical race occurs whenever there are two or more copies of the same node in the set A or the set K during execution of a Program Structure.

The term "critical race" is chosen because it resembles a like situation described in switching theory. The purpose is to show that if execution of a Program Structure can result in no critical races, then that Program Structure is repeatable. Note that the Program Structure illustrated in Figure 4.2 does contain a critical race and it is not repeatable. The critical race occurs because it is possible to have two copies of node N^3 in the set K at the same time. The value placed in the output set will depend on the order in which the nodes are processed.

Definition 4.5

The value vector of a node N^i will be denoted by $V^i = (d^i_1, d^i_2, \ldots, d^i_n)$

where d^i_j is used to represent the value of $D^{i,j}$.

Lemma 4.3

Given an input set I, if the value vector V^i is unique for any node N^i, then that Program Structure is repeatable.

Proof:

Suppose N^j is any output node. Since V^j is unique, the values in the output set are unique.

Theorem 4.4

If critical races cannot oocur during the execution of a solvable Program Structure, then that Program Structure is repeatable.

Proof:

The absence of critical races guarantees that all copies of any node N^i are processed in a unique order, i.e., $N^{i,1}$, $N^{i,2}$, ..., $N^{i,m}$. Thus the values mapped to each copy are also unique. This further implies that the value vector V^i for any node N^i is unique, and by Lemma 4.3, the Program Structure is repeatable.

The analysis formalism which was developed in the previous chapter can be used to determine whether or not a Program Structure contains any critical races. For example, consider Figure 4.1. The transition statements for that Program Structure are:

$$(1,1,d) \longrightarrow (4,0,1)$$

$$(2,1,d) \longrightarrow (3,1,d)$$

$$(3,1,1) \longrightarrow (3,1,d) \quad \wedge \quad (4,0,3)$$

$$(3,1,0) \longrightarrow (4,0,3)$$

$$(4,0,1) \quad \wedge \quad (4,0,3) \longrightarrow (5,0,d)$$

If it is possible to write a series of transitions from the above statements that will lead to a node more than once before that node is transformed to another, then the Program Structure contains a critical race. From the above transition statements, the following series of transitions is possible:

$$(2,1,d) \longrightarrow (3,1,d) \longrightarrow (3,1,1) \longrightarrow (3,1,d) \quad \wedge \quad (4,0,3)$$

$$\longrightarrow (3,1,1) \longrightarrow (3,1,d) \quad \wedge \quad (4,0,3) \quad \ldots\ldots\ldots$$

The state $(4,0,3)$ has been referenced more than once before being transformed to another node. This means that there can be two or more copies of node N^4 in the set A at the same time. This implies that a critical race exists. Results using the above appear at the end of the chapter.

The analysis formalism can also be used to help determine the number of equivalence classes present in a Program Structure. The number of equivalence classes is closely related to the number of output nodes. In particular, consider the Program Structure of Figure 2.4. The transition statements are:

$$(1,1,d) \longrightarrow (2,1,d) \quad \wedge \quad (3,1,d)$$

$$(2,1,d) \longrightarrow (4,0,2)$$

$$(3,1,1) \longrightarrow (4,0,3)$$

$$(3,1,0) \longrightarrow (3,0,d)$$

$$(4,0,2) \quad \wedge \quad (4,0,3) \longrightarrow (4,1,d)$$

$$(4,1,1) \longrightarrow (2,1,d) \quad \wedge \quad (3,1,d)$$

$$(4,1,0) \longrightarrow (2,0,d) \quad \wedge \quad (3,1,d)$$

The following series is possible given the above transition statements:

(1) $(1,1,d) \longrightarrow (2,1,d) \wedge (3,1,d) \longrightarrow (4,0,2) \wedge (4,0,3)$

$\longrightarrow (4,1,d) \longrightarrow (4,1,1) \longrightarrow (2,1,d) \wedge (3,1,d) \longrightarrow \ldots\ldots$

(Infinite cycle)

(2) $(1,1,d) \longrightarrow (2,1,d) \wedge (3,1,d) \longrightarrow (4,0,2) \wedge (3,0,d)$

(3) $(1,1,d) \longrightarrow (2,1,d) \wedge (3,1,d) \longrightarrow (4,0,2) \wedge (4,0,3)$

$\longrightarrow (4,1,d) \longrightarrow (2,0,d) \wedge (3,1,d) \longrightarrow (4,0,3)$

Using the analysis formalism, it appears that the Program Structure can enter an infinite cycle. However, after studying the Program Structure, this should not happen since there is a control built in to prevent such a possibility. This may appear to be a serious anomaly, but in practice it could be very worthwhile in detecting possible infinite cycles.

At any rate, the analysis formalism indicates that the Program Structure can terminate in one of three ways. This further suggests that there are only three distinct execution sequences, and hence three equivalence classes.

If a Program Structure does contain a critical race, it may be possible to make an alteration to it in order to insure repeatability. In Figure 4.3 there is an example of a Program Structure which contains a critical race. The purpose of the Program Structure is to implement a Finite State Machine. Upon close inspection, it should be apparent that nodes N^5 and N^6 could be processed repeatedly. This could cause multiple copies of node N^7 to appear in the set A. When these copies become known,

the order in which they are processed will determine the set of values in the output set. This Program Structure can be made repeatable by adding a "delay" node N^4 to insure that each input symbol properly correlates with the current state. The "fixed up" Program Structure is the one which appears in Figure 2.7.

Definition 4.6

Two execution sequences are said to be similar if each contains exactly the same occurences of nodes as the other.

Proposition 4.5

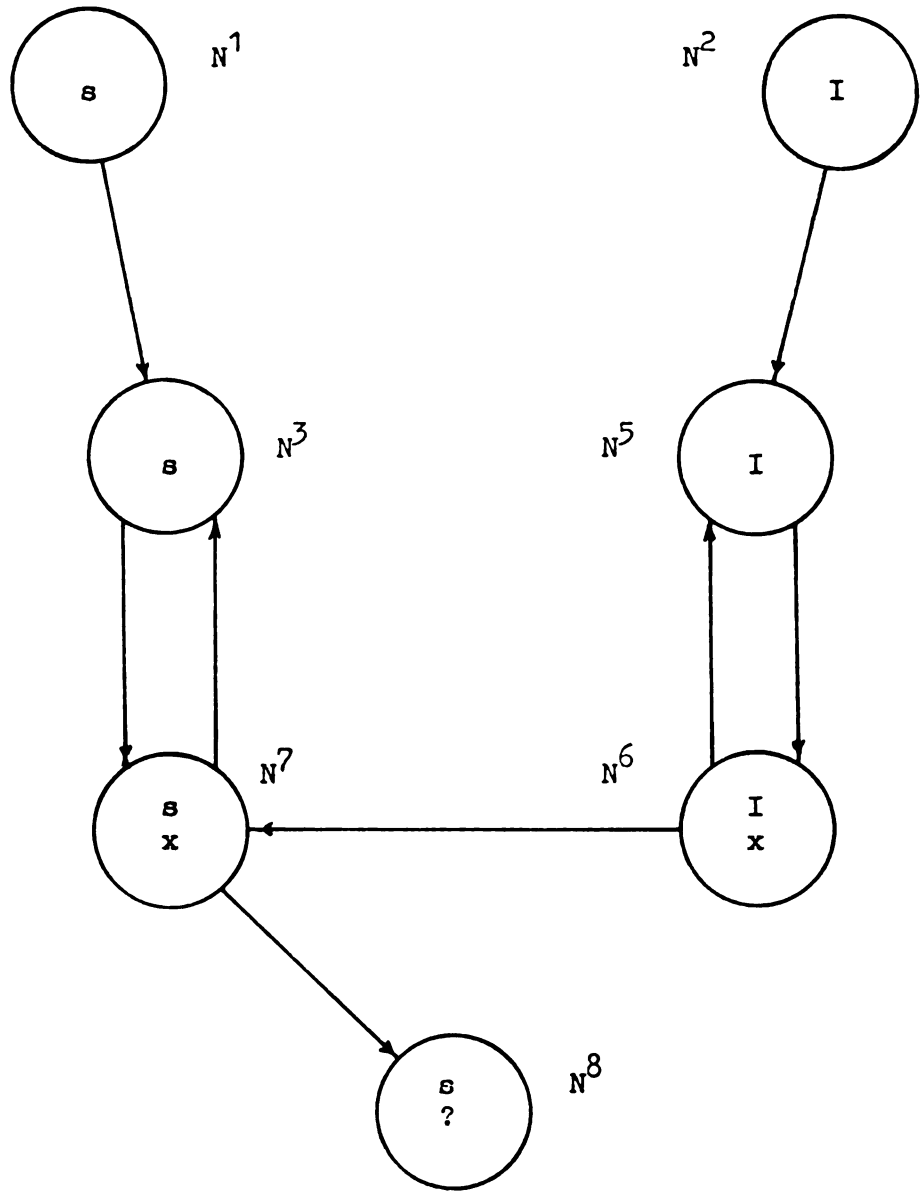
A solvable Program Structure is repeatable if for any given input set the following properties hold:

1. All possible execution sequences are similar to each other
2. For any nodes $N^{i,p}$ and $N^{i,q}$ where $p < q$, $N^{i,p}$ always precedes $N^{i,q}$ in all execution sequences.

Proof:

Clearly if the above properties hold for any Program Structure, then that Program Structure does not have any critical races. Hence, repeatability is assured.

One of the objectives in using multiprocessing is to decrease the processing time of programs. To achieve this, as many nodes as possible should be processed concurrently. This naturally leads to some considerations about the "degree of parallelism" in a Program Structure. In other words, a high degree of parallelism would imply less processing time.



Mappings are assumed to be similar to those in Figure 2.7

Figure 4.3 Program Structure which contains a critical race

Definition 4.7

A Program Structure is maximally parallel if for every node which contains more than one mapping, then all of the mappings of that node are identity mappings.

An identity mapping is one that performs no manipulation of information, but rather copies information into another node. For the purpose of the above definition, complex mappings will be considered as identity mappings. The following is an example of an identity mapping:

$$f^3(a, b, c, d) = (?, c, ?, a)$$

In Figures 4.4 and 4.5 a comparison is made between two segments of a Program Structure. For the sake of comparison, assume that identity mappings require one unit of time and the mappings of Figure 4.4 each require ten units of time. This means that the program segment in Figure 4.4 could be executed in a minimum of thirty time units. By comparison, the program segment of Figure 4.5 could be executed in a minimum of thirteen time units. If a number of processors are available, it then becomes very advantageous to have a Program Structure in maximally parallel form.

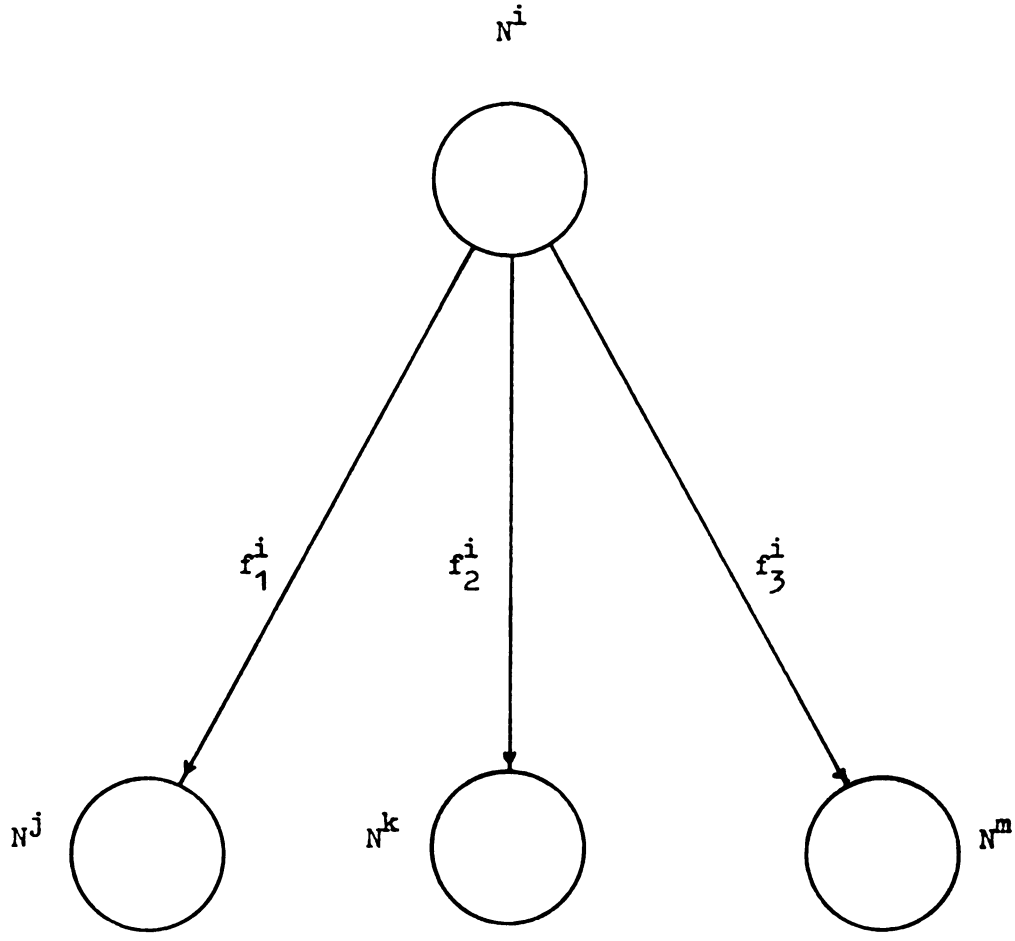
Proposition 4.6

Any Program Structure can be rewritten so that it is in maximally parallel form.

Proof:

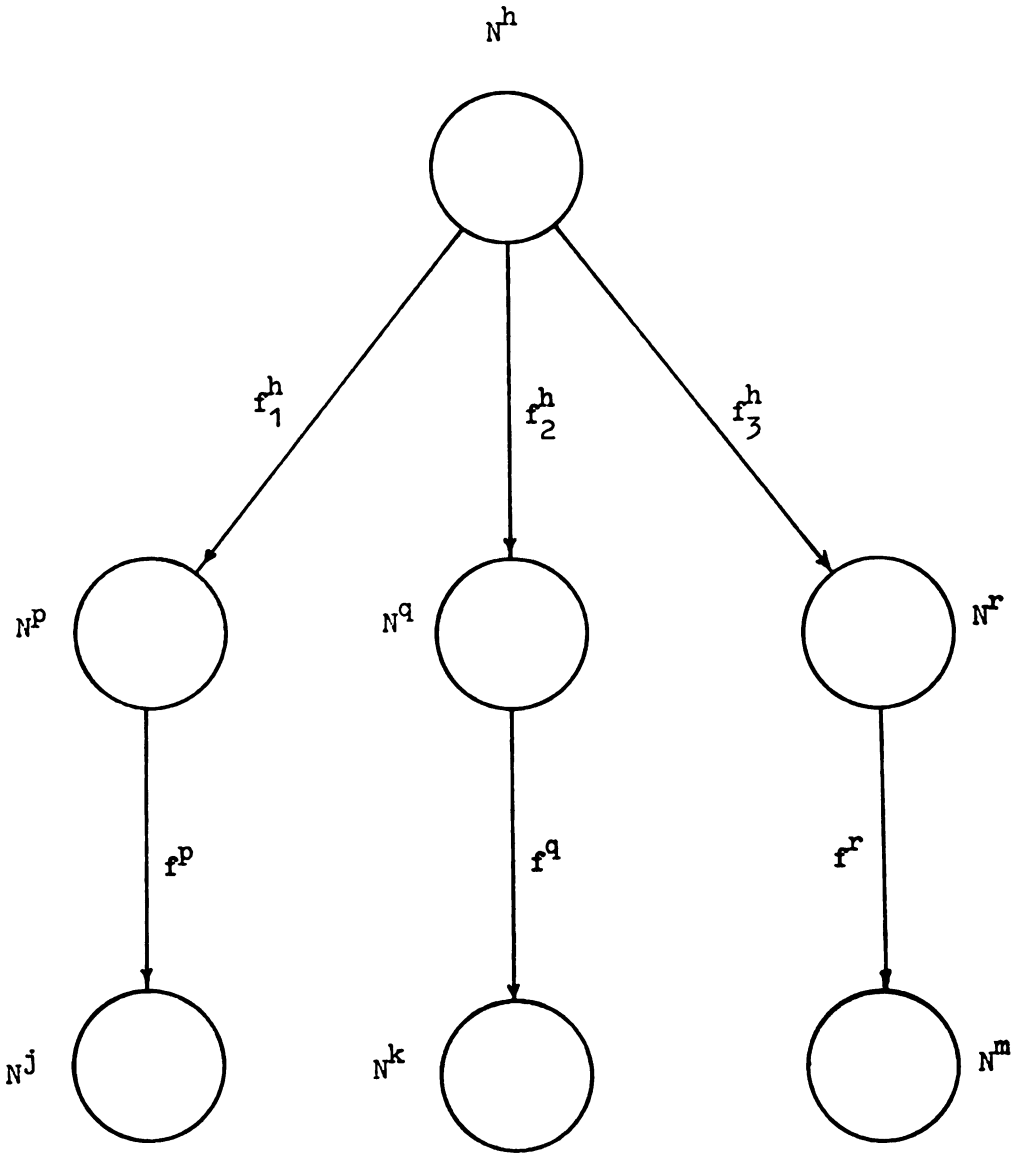
The proof consists of a procedure which will effectively force any Program Structure to be in maximally parallel form.

For each node N^i that contains a multiple mapping and n of the



f_1^i , f_2^i , f_3^i are non-identity mappings

Figure 4.4 Example of "poor" parallelism



Assume f_1^h , f_2^h , f_3^h , are identity mappings and

f^p, f^q , f^r are the same as f_1^i , f_2^i , f_3^i , in Figure 4.4

Figure 4.5 Example of Maximally Parallel Form

mappings ($n \geq 1$) are non-identity mappings, perform the following:

1. Create n new nodes that have the same data vector as N^i
2. Each of the new nodes will have only one mapping, i.e., each new node will have one of the n non-identity mappings
3. Replace the non-identity mappings in N^i with identity mappings that map the data vector of N^i into each of the n new nodes

When this procedure is completed, the Program Structure will be in maximally parallel form.

Proposition 4.7

The maximally parallel form of any repeatable Program Structure is also repeatable.

Proof:

Assume the proposition is false and then show a contradiction. For any node N^j in the original Program Structure, the value vector V^j is identical to that of the corresponding node in the rewritten Program Structure. This is true because the only alteration to the original node was to change some of the mappings to identify mappings. Then for at least one node N^i, the value vector V^i is not unique. Suppose N^i is a new node that was created from original node N^j. Then the value vector V^j is not unique either since the mapping from N^j to N^i is an identity mapping. Hence a contradiction.

Much of the previous discussion of repeatability has utilized various aspects of a Program Structure after it has been executed. It would be of great practical value if it were known that a particular Program Structure was repeatable before it was executed. The analysis formalism will be used

for this purpose.

Definition 4.8

When writing transition statements, a state is considered current under either of the following circumstances:

1. The node being described is an input node and that state has not yet appeared on the left side of a transition statement
2. Any state referenced by the right side of a transition statement until that state appears on the left side of a transition statement

Definition 4.9

A path is a series of transiton statements which leads to a particular current state.

Proposition 4.8

If x is any state and there exists one or more paths to x while x is current, then that Program Structure contains a critical race.

Proof:

If n represents any node, then x is of the form (n, 0, a) or (n, 1, a) where a is some test value. Since there are two or more paths to x, this means that during execution of the Program Structure there can be two or more copies of node n in set A or set K concurrently. Hence, a critical race exists.

Similar arguments can be made for the following propositions.

Proposition 4.9

If n represents any node, $a \neq b$, and states (n, 1, a) and (n, 1, b) are both current, then that Program Structure contains a critical race.

Proposition 4.10

If n represents any node, test values a and b are not necessarily equal, and states (n, 1, a) and (n, 0, b) are both current, then that Program Structure contains a critical race.

Thus it has been shown that by using the analysis formalism, a mechanical procedure can be developed which can detect the presence of critical races in any Program Structure before it has been executed. This can be of considerable value when dealing with large programs.

CHAPTER V

IMPLICATIONS

5.1 Elimination of Classical Problems

In this section, solutions to such problems as interference among nodes and deadlock will be presented. The problem of interference has received a great deal of attention in the literature, particularly by Dijkstra (19) and Gilbert and Chandler (22).

Definition 5.1

Two nodes N^i and N^j are said to interfere with node N^k if all of the following hold:

1. $f^i:D^i \rightarrow D^k$ and $f^j:D^j \rightarrow D^k$
2. $f^i(D^i)$ & $f^j(D^j) = f^i(D^i)$
3. N^i and N^j can be in set K concurrently

Interference is a problem unique to multiprocessing in that it can only arise when two or more processors are trying to simultaneously change the value of some location in memory. The copy concept of Program Structures reduces the seriousness of this problem but it does not entirely eliminate it.

There are only two ways in which interference can arise in the execution of a Program Structure. In the first case, two copies of the same node are processed concurrently. Note that more than one node can be mapping information into a particular node simultaneously. Consequently, the timing can be such that both copies are mapping into the same node. The second case is similar to the first except different nodes map into

the same node concurrently.

Even in Program Structures the problem of interference is a potentially dangerous one. Not only does this situation prevent repeatability, it may also destroy any value produced by the execution of a Program Structure. If two nodes map into the same data element at the same time, the result will be indeterminate; in fact the result may cause erroneous action to be taken.

Solution of this problem is accomplished in two steps. The first step establishes a mechanism by which all potentially interfering nodes are detected. The second step develops a procedure to eliminate interfering nodes. Most other approaches to this problem simply do not allow more than one processor to simultaneously access shared memory. The mechanism used to insure this is called mutual exclusion. The approach taken here is not so restrictive; interference will be eliminated and yet many processors can be accessing shared memory simultaneously.

Definition 5.1 specifies the criteria which causes interference so the problem of detecting possible interference becomes one of determining whether or not there are any nodes which meet that criteria. This determiniation can be accomplished in large part by using the analysis formalism. All nodes which meet the first criterion (i.e., all N^i, N^j that map information into N^k) can be detected as follows:

$(i, d, d) \longrightarrow (k, d, d)$

$(j, d, d) \longrightarrow (k, d, d)$

This amounts to a checking of transition statements to find all nodes that map into the same node. This of course does not guarantee interference. In fact, there is no problem if each data element in N^k has one or less

mappings that reference it. For an example of this, consider junctions. there are always two or more nodes that map into a junction and this in itself does not necessarily lead to interference.

The second criterion of the definition determines whether or not there is one or less mapping to each data element of node N^k. If $f^i(D^i)$ & $f^j(D^j) = f^i(D^i)$ and criterion one has been met, then there is at least one data element in N^k that is referenced by both N^i and N^j. Even though both of the first two criteria have been met there is still no assurance that interference is even possible. The third criterion establishes that possibility. If both N^i and N^j are known concurrently, then they can be processed concurrently and furthermore it is possible that they will map information into the same data element of N^k concurrently.

By using the analysis formalism, it is possible to determine whether or not nodes N^i and N^j can be known concurrently. If there exist paths to N^i and N^j where the path to N^i does not depend on N^j and the path to N^j does not depend on N^i, then N^i and N^j can be known concurrently. As in the first criterion, this becomes a matter of mechanically checking the paths to N^i and N^j and making certain the above condition holds.

The preceding discussion is summarized in the following theorem.

Theorem 5.1

There exists an effective procedure which can detect all potentially interfering nodes.

The solution to this problem is actually a matter of properly synchronizing the processing of certain nodes. When a Program Structure is being executed, the nodes in the set K can be processed in any order.

To eliminate interference, all that is necessary is to insure that potentially interfering nodes are not processed concurrently. Since it is possible to detect all potentially interfering nodes, suppose a list L of these nodes is created. Furthermore, suppose that this list L is created such that each element of L is a set of nodes which potentially interfere with the same node. Interference will be eliminated if the following change is made in the processing of known nodes.

For any known node N^i, that node can be processed only if either of the following conditions hold:

a. For all $x \in L$, $N^i \notin x$ OR

b. For all $x \in L$ where $N^i \in x$, no other element of x is currently being processed when N^i is being processed (i.e., prevent concurrent execution of two or more potentially interfering nodes)

The above discussion constitutes the proof for the following theorem.

Theorem 5.2

The execution of any Program Structure can be synchronized so that interference of nodes is eliminated.

There is another way in which interference can be eliminated. This method has the advantage of not altering the way in which nodes are processed. Suppose nodes N^i and N^j potentially interfere with node N^p. All that is necessary to prevent interference is to create a new node N^q that is identical to N^p and for all $f^i \in N^i$ that map into N^p, change the mappings so that they map into N^q. It may be necessary to repeat this process a number of times and it may substantially increase the

size of the Program Structure. However, execution time would most likely decrease since the resulting Program Structure would be in a more parallel form.

Although it is possible that a repeatable Program Structure may contain interfering nodes, it is not necessary to synchronize execution in order to eliminate interference. By definition, a repeatable Program Structure will yield the same output set each time it is executed. Consequently, if there are interfering nodes, then the interference they create is incidental and does not contribute in any way to the values obtained in the output set.

Quite often, the solution to one problem leads to the creation of another problem. In the situation created by Theorem 5.2, it must be known whether or not the restraints imposed will reslut in perpetual blocking of all processors.

Definition 5.2

A Program Structure is said to be deadlocked when it becomes impossible to process any of the known nodes.

The concept of deadlock is analogous to the "deadly embrace" described by Dijkstra (19) and Luconi (34) as well as the "hang up state" discussed by Petri (45) and Rodriguez (41).

Suppose a Program Structure contains interfering nodes and the synchronizing scheme suggested by Theorem 5.2 is used to eliminate them. Processing of any interfering node N^i will be delayed only as long as there are one or more known nodes, then it will always be

possible to process at least one node. Execution time may be increased, but deadlock will not occur.

Before discussing the deadlock problem for Program Structures, this problem will be first viewed with respect to other models. In particular, attention is given to problems arising in the use of shared memory and shared resources.

Traditionally, when a processor has access to shared memory, it can read or write anywhere in that memory. Since the memory is used as a means of communication between processors, safeguards must be maintained to prevent two or more processors from accessing shared memory concurrently. However, such a mechanism can have the effect of preventing all processors from ever accessing the shared memory and deadlock results.

Another way deadlock can occur is when processors share resources, such as input or output devices. For example, suppose one processor has control over the one available reader and needs to get control of the printer before it can finish. At the same time, another processor has control of the one available printer and needs to get control of the reader before it can finish. Neither processor can proceed and deadlock results.

The most common approaches to the deadlock problem have been to either develop an algorithm to prevent it or to simply allow it to happen and then take corrective action. The former approach has received much attention; one such widely known algorithm is the "banker's algorithm" of Dijkstra (19).

The concept of shared resources is not relevant when discussing Program Structures. The mappings in each node perform manipulations on information only; furthermore, the concept of input and output does not rely on external devices.

Program Structures do use shared memory. However, the concept is somewhat different than previous approaches. When a processor is accessing shared memory, it can only access the information contained in the node it is currently processing. Consequently, many processors can be accessing shared memory simultaneously and not affect the final outcome. It is also possible for two processors to access identical information simultaneously. The "copy of a node" concept makes this possible. This modularity makes it unnecessary to create algorithms whose purpose is to prevent two or more processors from accessing shared memory concurrently.

The problem of deadlock is a serious matter in previous multiprocessing models. However, this problem is of no consequence for general Program Structures.† No restrictions have been made as to the amount of available memory or even the number of available processors. There is no configuration which can prevent the processing of nodes in the set K. Hence, the next proposition follows immediately.

Proposition 5.3

A Program Structure with no restriction on the amount of available resources cannot be deadlocked.

5.2 Finite Program Structures

This thesis has investigated Program Structures in their most general form. In this section, a restricted form of Program Structures will be studied with an emphasis on problems associated with this form.

---

† The next sections deal with special cases of Program Structures where deadlocks can occur.

Definition 5.3

A Finite Program Structure (FPS) is a Program Structure which has access to a finite amount of memory and a finite number of processors.

The definition does not imply that an FPS must deal with a finite number of nodes. Indeed, an unsolvable FPS is infinite in this sense. Besides limiting the number of processors, the definition states that there is a limitation on the total amount of memory available to store all the unknown, active, and known nodes.

Definition 5.4

The amount of memory required by any node N^i will be denoted by Size (N^i). The total amount of memory available to a particular FPS will be denoted by the mnemonic Mem.

There are several trivial ways in which deadlock can occur in an FPS. If the number of available processors is zero, then nodes in the set K cannot be processed. Also, if the amount of memory required by any node exceeds Mem, then deadlock results because the set U cannot be placed in memory. These trivial cases will not be considered further.

The "copy of a node" concept has proved to be a very powerful feature in that it can help to eliminate interference between nodes. There is a negative aspect to this in that improper usage can cause deadlocks. Creation of unnecessary copies can occupy all available memory and lead to deadlock.

Assumed Conditions regarding FPS

1. There is at least one processor available for use.
2. Mem $\geq$ 3 $\sum_{N^i \in U}$ Size (N^i)

The only way that deadlock can occur in an FPS is by attempting to occupy more memory than is available. This can happen if too many copies of nodes are created. It is possible to state that for certain FPS deadlocks can not occur. For the others, a dilemma is faced. If these FPS are executed, then some will become deadlocked and some will not. Some care must be exercised when dealing with FPS that might lead to deadlock. If an FPS does become deadlocked, recovering from it can become a very expensive propositon.

This suggests two approaches to the problem. If an FPS cannot become deadlocked, then there is no problem and no restrictions need to be placed on the execution of that FPS. This implies that for any FPS there must be some method of determining whether or not a given FPS is deadlock free. A method of detecting deadlock free FPS is presented shortly. If an FPS cannot be classified as deadlock free then it is necessary to alter the execution of that FPS. By properly processing the nodes, it may be possible for such an FPS to not end in deadlock. Of course, there are some FPS that are hopeless. That is, no matter what precautions are taken, deadlock will result. This topic is also discussed shortly.

Theorem 5.4

If a Finite Program Structure contains no critical races and assumed conditions 1 and 2 hold, then that Finite Program Structure is free from deadlocks.

Proof:

Elimination of critical races implies that there cannot be more than one copy of any node in set A and/or set K. However, when a known node is being processed it is no longer in the set K but it still requires use of memory. Since it is permissible for a node to map information into a copy of itself, it is possible that memory would be required for the amount of two copies of that node. Since it is assumed that $\text{Mem} \geq 3 \sum_{N^i \in U} \text{Size}(N^i)$ then there is an adequate amount of memory and it is impossible for deadlock to occur.

By using the analysis formalism it can be readily determined whether any FPS contains a critical race. According to proposition 4.10 all that is necessary is to examine the transition statements. If an FPS contains a critical race, then there exists one or more paths to a current state. In other words, if the transition statement $(x, 1, d) \longrightarrow (i, 0, x)$ is possible while $(i, 0, x)$ is a current state, then a critical race exists.

Theorem 5.4 has shown that it is possible to determine whether any FPS is free from deadlock. However, this is not adequate since many FPS may not fall into this category. If an FPS cannot be classified as deadlock free, this does not necessarily mean that it is certain to end in deadlock. A stronger result is needed in order to determine whether it is possible for any given FPS to be executed and avoid deadlock.

There will be few restrictions placed on the FPS under consideration. Although properties of execution sequences are utilized, it is not necessary to make any assumptions about them. Since a general solution to the deadlock problem is sought, the results must hold regardless of whether the FPS is repeatable or even solvable.

Cycles and junctions are permitted and it is required that all cycles be closed cycles. This will have the effect of assuring a finite number of execution sequences for unsolvable FPS. Note that every unsolvable FPS contains at least one cycle. The presence of a cycle or even an infinite cycle does not pose any difficulty in studying the deadlock problem. The nodes of a closed cycle can be treated as nodes outside a cycle when considering memory usage. For example, given the following execution sequence which contains a closed cycle: $N^1\ N^2\ N^4\ \overline{N^5\ N^6\ N^7\ N^8}\ N^{10}\ N^{11}$

The nodes in the closed cycle do not have to receive any special consideration. Each node in the cycle would have to be checked (just as any other node) to determine whether memory had been exhausted. Further, it is immaterial how many times the cycle would be executed. If deadlock does not occur during the first pass through the cycle, then deadlock cannot occur after n passes where $1 \leq n \leq \infty$.

Definition 5.5

t_j = $\{(x, y) \mid$ x is a node designator, y is a set of node designators such that for all $i \in y$ there exists $f^i: D^i \longrightarrow D^x\}$

T^i = $\{t_j \mid$ where t_j is a description of the sets A and K after the jth node in execution sequence i has been processed$\}$

Since t_j is a description of the sets A and K, for any $x \in t_j$ where x = (a, b), the elements of b are unique. However, it is possible to have $y \in t_j$ where y = (c, d) and a = c. For each node in A or K during execution sequence i there is one or more corresponding entries in T^i. It is a relatively easy matter to determine whether or not a node is known. If

$x \in t_j$ where $x = (a, b)$ and $b = (b_1, b_2, \ldots, b_m)$ then N^a is a known node if there exists a transition statement of the form:

$(b_1, d, d) \wedge (b_2, d, d) \wedge \ldots \wedge (b_m, d, d) \longrightarrow (a, 1, d)$

Proposition 5.5

A critical race does not occur in execution sequence i if for all $t_j \in T^i$ and for all $x \in t_j$ where $x = (a, b)$ there does not exist $y \in t_j$ where $y = (c, d)$ and $a = c$.

Proof:

Follows immediately from the definition of a critical race.

Definition 5.6

$D = \{ i \mid \text{execution sequence i does not lead to deadlock} \}$

The elements of D can be determined by computing the amount of memory required by nodes represented in each T^i. For convenience when performing this computation, it can be assumed that only one processor is being used.

Proposition 5.6

If a Finite Program Structure is deadlock free when using one processor, then it is deadlock free when using two or more processors.

Proof:

There must always be at least one processor able to completely process some node. If this were not true, deadlock could arise when using only one processor. When multiple processors are used, it could be possible for all but one to be idle. However, all nodes will ultimately

be processed and deadlock cannot occur.

Proposition 5.7

If there are n execution sequences for a given Finite Program Structure and #(D) = n then that Finite Program Structure is free from deadlock.

Proof:

Since every execution sequence generated by the FPS does not lead to deadlock, then clearly the Finite Program Structure is free from deadlock.

Proposition 5.8

Suppose there are m equivalence classes of input data for any given repeatable Finite Program Structure. Let E^i represent the set of execution sequences for equivalence class i ($1 \leq i \leq m$). For any equivalence class i, if there exists $x \in E^i$ such that $x \in D$ then that Finite Program Structure can avoid deadlock.

Proof:

By definition, any two execution sequences in E^i are equivalent. Then for all inputs in equivalence class i, there exists an execution sequence which will achieve the desired output and be able to avoid deadlock. Since the proposition states that this holds for any equivalence class, then it is possible to process any input and not end in deadlock.

A corollary to proposition 5.8 is now presented which considers both repeatable and non-repeatable FPS.

Corollary 5.9

A Finite Program Structure can avoid deadlock if for any execution sequence either it or an equivalent execution sequence does not lead to deadlock.

Proof:

Follows immediately from proposition 5.8.

The previous discussion and propositions can be summarized in the following theorem.

Theorem 5.10

It is decidable whether any Finite Program Structure can avoid deadlock if all cycles are closed cycles.

Much of the previous discussion has centered on the detection of potential deadlock. When it is possible for an FPS to avoid deadlock then processing of nodes must be synchronized properly. This implies that the order of processing the nodes is predetermined (i.e., knowing the execution sequences) or making an adjustment to the processing algorithm so that the proper order of processing nodes would always be observed.

When a deadlock does arise, it may be possible to preempt the processing of some node in an attempt to recover from this situation. In other words, when a processor is about to perform its mappings and finds no available memory it can return that node to the set K and attempt to process a different node. However, if proper synchronization is used, preemption of nodes will not be necessary.

5.3 Copy Free Program Structures

In this section, properties of a Program Structure in which copies are not allowed are investigated.

Definition 5.7

A Copy Free Program Structure (CFPS) is an FPS where each node appears exactly once and must be in Set U, A, or K.

In order to utilize the properties of the "copy" concept, an analogous property is now defined.

Definition 5.8

Each element of the data vector of any node of a Copy Free Program Structure is a finite length first-in first-out data queue.

Many of the results established for Program Structures are applicable here. For example, a node is known if each of its data queues contains one or more elements. The current value is always the value at the front of the queue. In the mapping $f^i: D^i \longrightarrow D^j$ where $f^i(D^i) =$ (a, ?, b), the first and third elements of D^j are referred to as defined data queues for this mapping. When a mapping is performed at least one element is added to the rear of each defined data queue. Entries are duplicated so that the queue length of each defined data queue equals the maximum queue length of the data vector.

Proposition 5.11

A Copy Free Program Structure is repeatable if for any input set the

data values arrive at the data queues in a unique predetermined order.

Proof:

Assume that the value of an output node is composed of the elements at the rear of the data queues. This proposition then follows immediately from the definition of repeatability.

A special case of the above proposition would be a situation where the maximum length of any data queue is one. Of course, if this condition is met the CFPS is repeatable.

Although proposition 5.11 may seem very restrictive, in practice it would be a more desirable condition to attain rather than the special case discussed above. This would be the case since it would be possible to achieve greater utilization of the processors. In fact, the condition expressed in the special case implies that there is no need to use data queues, but rather use single valued variables.

In the previous two sections, the problems of interference and deadlock were discussed. Although the results obtained could be applied here as well, an additional comment is necessary with respect to deadlock.

Proposition 5.12

A Copy Free Program Structure can be deadlocked only if the maximum length of one or more data queues is exceeded.

Proof:

Deadlock can only occur when it becomes impossible to process any of the known nodes. There are two ways that this can happen:

1. All available processors are delayed because they are processing nodes which are attempting to map information into one or more

nodes which have exceeded their maximum queue length.

2. All currently known nodes are being processed and are delayed for the same reason as (1).

In either case, deadlock occurs as a result of attempting to exceed the maximum length for one or more data queues.

Note that the exceeding of a maximum queue length does not of itself guarantee deadlock. It is possible for a CFPS to "recover" when this situation arises simply by processing the node(s) with maximum queue length a number of times while there are processors still available. If this situation does arise, it can seriously degrade system performance since at least one processor must be idle as long as that situation exists.

Although Program Structures are intended to represent programming processes rather than a scheme for machine design, the implementation of CFPS resembles existing hardware systems. Such a system is the CDC STAR which is a pipeline processor and is described by Hill and Peterson (25). In the pipeline computer (depicted in Figure 5.1) there is a shared memory and a number of arithmetic units which communicate with the memory via the input and output pipelines. There are many technical considerations that are overlooked such as methods of insuring the determinacy of the system. Also, the pipeline processor might have a number of special purpose elements in the arithmetic unit.

When considering functional characteristics, the pipeline processor bears a great deal of resemblence to CFPS. In fact, the pipeline processor is a special case of CFPS. The data queues represent the input pipeline and the mappings represent the output pipeline. The major difference is that the processors of a CFPS are all identical. This suggests that the

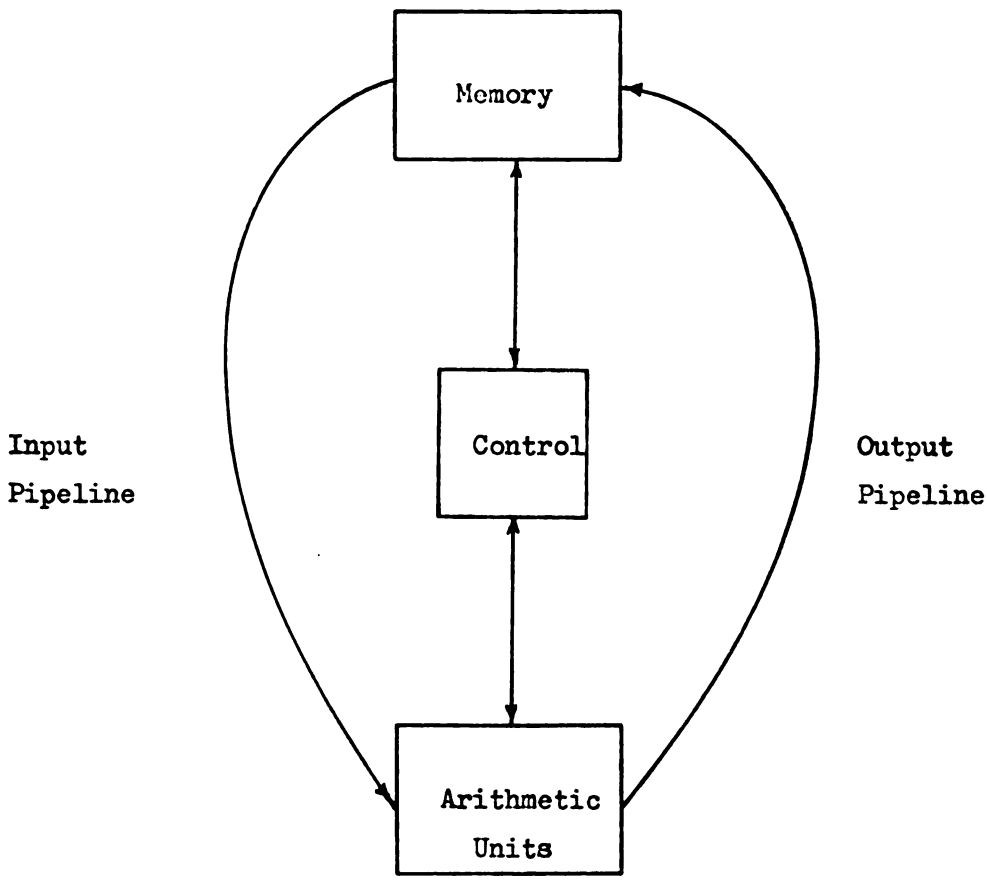


Figure 5.1 Functional Diagram of a Pipeline Processor

cost of a CFPS would be higher but this would be compensated by added flexibility and possibly better thruput.

5.4 Comparison of FPS and CFPS

The Program Structure in its most general form is clearly free from the deadlock problems exhibited by FPS and CFPS. However, in a practical sense, it is impossible to implement the general Program Structure due primarily to limitations on memory size. Consequently, the two special cases of Program Structure are contrasted in this section.

In both cases, repeatability is assured based on similar conditions established for general Program Structures. Deadlock can be detected based on finiteness conditions although it is easier to predict for FPS.

The two major areas of comparison are speed of execution and amount of memory required. If there are n nodes in a CFPS, then the maximum number of nodes which can be processed concurrently is n assuming that processors are available. For FPS, the maximum could be much higher because a number of copies of the same node could be processed simultaneously assuming available processors and memory.

The advantage gained by increased speed of FPS is somewhat compensated by increased use of memory. Although additional memory is needed for the data queues of a CFPS, it would in general be less than that required to create copies of nodes.

When a person describes a solution to a problem by means of a Program Structure, he may not be aware of whether he is using an FPS or a CFPS. It may be possible to obtain different results depending on which was used.

Definition 5.9

A Finite Program Structure and a Copy Free Program Structure are equivalent if for any input set the output set is identical for both.

It may seem that if two structures contain exactly the same components then they would be equivalent. This is not always the case with FPS and CFPS. Suppose there is a node N^4 which has exactly three mappings to it defined as follows:

$f^1(a, b, c) = (a, ?, b, c)$

$f^2(d, e) = (?, d, e, ?)$

$f^3(g, h) = (g, h, ?, h)$

In an FPS, there would be three copies of node N^4 created and none of those copies would become known. However, in the CFPS enough information has been mapped so that node N^4 can become known twice. Clearly, this can effect the values placed in the output set.

Theorem 5.13

A Finite Program Structure is equivalent to a Copy Free Program Structure if both of the following hold:

1. Any node N^i is contained in both structures
2. Neither structure contains any critical races

Proof:

Having the same node in both structures implies that not only are the node designators identical, but the mapping vectors and the capacity of the data vectors are identical as well. Showing that for any given input set the execution sequence generated by the Finite Program Structure

must be equivalent to that generated by the Copy Free Program Structure is sufficient to prove this theorem.

Suppose it is possible for the FPS to generate an execution sequence that is not equivalent to that generated by the Copy Free Program Structure using the same input set. Then there must exist at least one node $N^{i,j}$ in the execution sequences of the Finite Program Structure and the Copy Free Program Structure which has one or more data elements whose values are not identical. This can only happen if at some time there are two or more copies of the same node in the set K or A or if there are two or more paths to $N^{i,j}$. In either case, this contradicts the fact that the Finite Program Structure and the Copy Free Program Structure are repeatable (absence of critical races implies repeatability). Hence given any input set the execution sequences generated by the Finite Program Structure are equivalent to the execution sequences generated by the Copy Free Program Structure.

It is not always possible to create a structure which can be used interchangeably as either an FPS or a CFPS. The last two propositions in this section show that it is possible to take any repeatable FPS and construct an equivalent CFPS and vice versa.

Proposition 5.14

For any repeatable Finite Program Structure there exists an equivalent Copy Free Program Structure.

Proof:

An equivalent Copy Free Program Structure will be constructed

from some or all of the nodes of the given Finite Program Structure. If the Finite Program Structure does not contain any critical races then an equivalent Copy Free Program Structure would contain exactly the nodes of the Finite Program Structure. Assuming that critical races do exist, it may be necessary to alter some of the nodes of the Finite Program Structure in order to construct an equivalent Copy Free Program Structure. This may be necessary because there are no copies of nodes in a Copy Free Program Structure. An example has been presented earlier in this section showing how this can be a problem.

A Copy Free Program Structure can be made to behave like an Finite Program Structure by changing the mappings so that a special distinguishable marker (say *) would replace all undefined elements. The multiple elements in the data queues would simulate copies of nodes. Every node that had at least one element in its data queue would be regarded as a known node. The processing of nodes would have to properly account for this.

Nodes would only be processed when all elements of the data vector would be defined by merging one or more disjoint rows of the data queues. After the node is processed all rows involved in the merging would be deleted. These alterations provide the capability of allowing the Copy Free Program Structure to mimic the operation of the Finite Program Structure. The only operation the Copy Free Program Structure will not be capable of implementing is processing two copies of the same node concurrently. Clearly, this operation is not required to achieve equivalence.

As an example of how the above alterations could be carried out,

consider node N^5 which has information mapped to it from nodes N^3 , N^7 , N^9:

$f^3(D^3) = (a, ?, b)$

$f^7(D^7) = (?, c, d)$

$f^9(D^9) = (?, e, ?)$

After nodes N^3 , N^7 , and N^9 have been processed, the data vector of node N^5 would appear as:

D^5

a	*	b
*	c	d
*	e	*

The first and third rows of D^5 could be merged to provide a data vector whose elements are all defined. After node N^5 has been processed, D^5 would contain only one row; the first and third would be deleted.

Note that if the mappings were not modified as suggested in the proposition, D^5 would appear as follows after nodes N^3, N^7 , and N^9 have been processed:

D^5

a	c	b
?	c	d
?	e	?

In this case, when node N^5 is processed the data vector would contain (a, c, b) rather than (a, e, b).

Proposition 5.15

For any repeatable Copy Free Program Structure there exists an equivalent Finite Program Structure.

Proof:

If there is never more than one entry in any data queue in the Copy Free Program Structure, then critical races do not exist and an equivalent Finite Program Structure can be constructed by using exactly those nodes of the Copy Free Program Structure.

Assuming that multiple entries in the data queues are possible, an equivalent Finite Program Structure will be constructed. The nodes of the Copy Free Program Structure will be used and various alterations will be made to them. Performance of mappings can cause different actions to be taken depending on which structure is being used. Another problem which can arise is that it is possible for more than one copy of a node to be known at the same time in a Finite Program Structure whereas the queueing discipline of the Copy Free Program Structure prevents this. The multiple copy problem can be eliminated by changing the execution algorithm so that the oldest copy of a node is always processed first. This forces the nodes of a Finite Program Structure to be processed in a first-in, first-out manner which is identical to the manner they would be processed in a Copy Free Program Structure.

The other problem to be resolved is forcing the mappings of the structure to behave identically. If any mapping to any node maps to n elements of the data vector ($n > 1$) all that is necessary is to rewrite that mapping in terms of n mappings where each maps to exactly one element. The original mapping would then be replaced by n distinct

mappings to the same node. Although there may be many more mappings in the Finite Program Structure, these additional mappings will assure duplicate results for the Finite Program Structure. Clearly, a Finite Program Structure has been constructed that behaves as though it were a repeatable Copy Free Program Structure.

Note that there are other ways in which an equivalent FPS can be constructed that do not involve the changing of the execution algorithm. One such method would be the invocation of proposition 4.1, i.e., making the FPS deterministic. The second step of this process would be to rewrite the mappings as was done in the previous proposition. In practice, it would not be desirable to make an FPS deterministic since only one node could be processed at any time.

CHAPTER VI

SUMMARY AND CONCLUSIONS

6.1 Summary

This dissertation has presented a novel approach to the problem of multiprocessing. The notions of shared memory and asynchronous operation are widely used in other models. However, in most cases, these techniques are limited because of the necessity to prevent two or more processors from concurrently accessing the same memory location.

The concept of the copy of a node is a new notion. Although seemingly innocuous, it is this feature that eliminates many of the classical problems of parallel processing as well as simplifying the entire process.

Chapter I presents an introduction and discusses previous work. A statement of the problem is also included.

A formal definition of Program Structures is presented in Chapter II interspersed with several motivational examples.

Some of the capabilities of Program Structures are discussed in Chapter III and it is shown how the set of all inputs of certain Program Structures can be partitioned in equivalence classes. A formalism is introduced which can be used in the analysis of Program Structures. This formalism proves to be most helpful as a tool and as a notational aid when studying various aspects of Program Structures.

Chapter IV discusses criteria necessary to ensure that the output produced by a Program Structure is unique. This property is called repeatability and it is assured based on the absence of any critical races. The maximally parallel form is investigated and it is shown that any Program Structure can be written in this form.

The problems of interference and deadlock are presented in Chapter V and solutions to these problems are discussed. Finite Program Structures and Copy Free Program Structures are defined and their properties are studied with an emphasis on elimination of deadlock. Finally, Finite Program Structures and Copy Free Program Structures are compared and conditions for equivalence of these structures are established.

6.2 Conclusions

The intention of this dissertation was to develop a multiprocessing model that was relatively easy to use and free from the problem associated with earlier models. Whether or not the model discussed here is easy to use may be a value judgement but clearly many of the classical problems have been either eliminated or diminished. A major innovation in this dissertation is the development of the concept of a copy of a node. It is this property that makes the notion of shared memory take on a new meaning.

It is no longer necessary to be concerned with the possibility of two or more processors accessing shared memory simultaneously. Consequently, there is no need to establish procedures for mutual exclusion. It is the copy concept that reduces the seriousness of problems such as repeatability, interference, and deadlock as well as aiding in finding solutions for these problems.

Other innovations are the creation of a single node type and the elimination of a separate control structure. Some models have over ten node types each with precisely defined input, output and computation, see (1, 3, 36, 41). Many models have employed two structures to represent parallel computation; one structure to represent data relationships and

the other to represent the control mechanism of the model. In this dissertation, there is only one structure and it serves as both a representation of data and control. Not only is the model highly readable, it is also very usable in a wide variety of applications as suggested by the examples in Chapter II.

6.3 Future Work

A topic which has received a great deal of attention in recent years is "Structured Programming". This is intended to be a technique for sequential programming where only three types of control structures are allowed: Linear sequence, selection, and repetition. Selection refers to conditional operation and repetition refers to looping. These three types of operations are closely related to Program Structures and further investigation could formalize these relationships. If it can be shown that Program Structures can implement and structured program then many of the results any techniques recently developed in this area can be utilized. In particular, study of translator writing systems using this marriage could prove most interesting. The paper by Lincoln (32) presents some ideas in this direction.

Another worthy area of research with respect to Program Structures is the study of efficiency. No material has been presented which considers the speed of execution of a Program Structure or even the amount of memory required. There can be numerous "bottlenecks" which can degrade system performance. If these bottlenecks can be detected, various scheduling schemes could be developed that would either eliminate them or at least minimize their effect. Since the amount of memory required by an FPS or a CFPS is critical, a study of minimum memory requirements would also be in order.

Finally, another possible area of investigation is proving program correctness of Program Structures. The problem of proving correctness should be simplified somewhat due to the modular construction of Program Structures. There have been a number of tools developed in this dissertation such as the analysis formalism and equivalence classes of data which could be most useful. Assertions about the behavior of Program Structures could be made at the node level and this should have the effect of reducing the complexity of the problem and making it possible to verify large scale programs.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. Adams, D. A., "A Model for Parallel Computations", Parallel Processor Systems, Technologies and Applications, L. C. Hobbs, et al, Spartan Books, New York, pp. 311-333, 1970.

2. Anderson, James P., "Program Structures for Parallel Processing", Comm. ACM, Vol. 8, pp. 786-788, December 1965.

3. Baer, J. L., "A Survey of some Theoretical Aspects of Multiprocessing", Computing Surveys, Vol. 5, pp. 31-80, March 1973.

4. Barnes, G. H., et al, "The ILLIAC IV Computer", IEEE Transactions on Computers, pg. 746, August 1968.

5. Bredt, T. H., et al, "Analysis and Synthesis of Control Mechanisms for Parallel Processes", Parallel Processor Systems, Technologies and Applications, L. C. Hobbs, et al, Spartan Books, New York, pp. 287-295, 1970.

6. Brinch Hansen, Per, "Concurrent Programming Concepts", Computing Surveys, Vol. 5, pp. 223-245, December 1973,

7. Brinch Hansen, Per, Operating System Principles, Prentice-Hall, New Jersey, 1973.

8. Brinsfield, W. A. and Miller, R. E., "On the Composition of Parallel Program Schemata", IEEE Conference Record of the 1971 Twelfth Annual Symposium on Switching and Automata Theory, pp. 20-23, 1971.

9. Cadiou, J. M. and Levy, J. J., "Mechanizable Proofs about Parallel Processes", IEEE Conference Record of the 1973 Fourteenth Annual Symposium on Switching and Automata Theory, pp. 34-48, 1973.

10. Campeau, J. O., "Communication and the Sequential Problems in the Parallel Processor", Parallel Processor Systems, Technologies and Applications, L. C. Hobbs, et al, Spartan Books, New York, pp. 215-234, 1970.

11. Chen, T. C., "Parallelism, Pipelining, and Computer Efficiency", Computer Design, Vol. 10, pp. 69-76, January 1971.

12. Chen, Y. E. And Epley, D. L., "Memory Requirements in a Multiprocessing Environment", Journal of the ACM, Vol. 19, pp. 57-69, January 1972.

13. CODASYL Development Committee, "An Information Algebra", Comm. ACM, Vol. 5, pp. 190-204, April 1962.

14. Coffman, E. G. Jr. and Denning, P. J., Operating Systems Theory, Prentice-Hall, New Jersey, 1973.

15. Constable, R. L. and Gries, D., "On Classes of Program Schemata", IEEE Conference Record of the 1971 Twelfth Annual Symposium on Switching and Automata Theory, pp. 5-19, 1971.

16. Conway, M. E., "A Multiprocessor System Design", AFIPS Conference Proceedings 1963 Fall Joint Computer Conference, Vol. 24, Spartan Books, New York, pp. 140-146, 1963.

17. Denning, P. J., "The Working Set Model for Program Behavior", Comm. ACM, Vol. 11, pp. 323-333, May 1968.

18. Dennis, J. B. and Van Horn, E. C., "Programming Semantics for Multiprogrammed Computations", Comm, ACM, Vol. 9, pp. 143-155, March 1966.

19. Dijkstra, E. W., "Cooperating Sequential Processes", Programming Languages, Gunuys, F., ed, Academic Press, New York, 1968.

20. Dijkstra, E. W., "The Structure of "THE" - Multiprogramming System", Comm. ACM, Vol. 11, pp. 341-346, May 1968.

21. Fleck, A. C., "Towards a Theory of Data Structures", Journal of Computer and System Sciences, Vol. 5, pp. 475-488, 1971.

22. Gilbert, P. and Chandler, W. J., "Interference between Communicating Parallel Processes", Comm. ACM, Vol. 15, pp. 427-437, June 1972.

23. Habermann, A. N., "Synchronization of Communicating Processes", Comm. ACM, Vol. 15, pp. 171-176, March 1972.

24. Hellerman, H., Digital Computer System Principles, McGraw-Hill, New York, 1973.

25. Hill, F. J. and Peterson, G. R., Digital Systems: Hardware Organization and Design, John Wiley and Sons, New York, 1973.

26. Hoare, C. A. R., "A Structured Paging System", Computer Journal, Vol. 16, pp. 209-214, August 1973.

27. Holt, R. C., "Some Deadlock Properties of Computer Systems", Computing Surveys, Vol. 4, pp. 179-195, March 1972.

28. Joyner, W. H. Jr., "Automatic Theorem-Proving and the Decision Problem", IEEE Conference Record of the 1973 Fourteenth Annual Symposium on Switching and Automata Theory, pp. 159-166, 1973.

29. Karp, R. M. and Miller, R. E., "Parallel Program Schemata", Journal of Computer and System Sciences, Vol. 3, pp. 147-195. May 1969.

30. Keller, R. M., "On Maximally Parallel Schemata", IEEE Conference Record of the 1970 Eleventh Annual Symposium on Switching and Automata Theory, pp. 32-50, 1970.

31. Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley Massachusetts, 1969.

32. Lincoln, N., "Parallel Programming Techniques for Compilers", SIGPLAN Notices, pp. 18-31, October 1970.

33. Llewellyn, J. A., "The Deadly Embrace - a Finite State Model Approach", Computer Journal, Vol. 16, pp. 223-225, August 1973.

34. Luconi, F. L., Asynchronous Control Structures, M. I. T. Project MAC report MAC-TR-49, M.I.T., Massachusetts, 1968.

35. McIntyre, D. E., "An Introduction to the ILLIAC IV Computer", Datamation, April 1970.

36. Miller, R. E., "A Comparison of some Theoretical Models of Parallel Computation", IEEE Transactions on Computers, pp. 710-717, 1973.

37. Organick, E. I., The Multics System, M.I.T. Press, Massachusetts, 1972.

38. Rabin, M. O. and Scott, D., "Finite Automata and their Decision Problems", IBM Journal of Research and Development, Vol. 3, pp. 114-125, April 1959.

39. Randell, B. and Kuehner, C. J., "Dynamic Storage Allocation Systems", Comm. ACM, Vol. 11, pp. 297-306, May 1968.

40. Regis, R. C., "Multiserver Queueing Models of Multiprocessing Systems", IEEE Transactions on Computers, pp. 736-745, 1973.

41. Rodriguez, J. E., A Graph Model for Parallel Computations, M.I.T. Project MAC report MAC-TR-64, M.I.T., Massachusetts, 1969.

42. Rutledge, J. D., "On Ianov's Program Schema", Journal of the ACM, pp. 1-9, January 1964.

43. Slutz, D. R., The Flow Graph Schemata Model of Parallel Computation, M.I.T. Project MAC report MAC-TR-53, M.I.T., Massachusetts, 1968.

44. Stone, H. S., "A Pipeline Push-down Stack Computer", Parallel Processor Systems, Technologies and Applications, L. C. Hobbs, et al, Spartan Books, New York, pp. 235-249, 1970.

45. Tsichritzis, D. C. and Bernstein, P. A., Operating Systems, Academic Press, New York, 1974.

MICHIGAN STATE UNIV. LIBRARIES
31293106418761