MIT/LCS/TR-64

A GRAPH MODEL FOR PARALLEL COMPUTATIONS

Jorge E. Rodriguez

September 1969

September, 1969 Report Nos. ESL-R-398
MAC-TR-64

A GRAPH MODEL FOR PARALLEL COMPUTATIONS

by

Jorge E. Rodriguez

The work reported in this document has been jointly supported by the Air Force Manufacturing Technology Laboratory, RTD, Wright-Patterson Air Force Base under Contract F33615-67-C-1530, M.I.T. DSR Project 70429, Electronic Systems Laboratory and by Project MAC an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

Electronic Systems Laboratory
Department of Electrical Engineering

Project MAC
545 Technology Square

Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

This report presents a computational model called program graphs which makes possible a precise description of parallel computations of arbitrary complexity on non-structured data. In the model, the computation steps are represented by the nodes of a directed graph whose links represent the elements of storage and transmission of data and/or control information. The activation of the computation represented by a node depends only on the control information residing in each of the links incident into and out of the node. At any given time any number of nodes may be active, and there are no assumptions in the model regarding either the length of time required to perform the computation represented by a node or the length of time required to transmit data or control information from one node to another. Data dependent decisions are incorporated in the model in a novel way which makes a sharp distinction between the local sequencing requirements arising from the data dependency of the computation steps and the global sequencing requirements determined by the logical structure of the algorithm.

The concept of the state of a program graph is introduced and it is proved that every program graph represents a deterministic computation, i.e., that the final state of each computation started from the same initial state is unique. Computations which do not terminate properly are defined in terms of the concept of hang-up state. Methods of analysis are developed and necessary and sufficient conditions for the absence of hang-up states are obtained. These conditions are interpreted in terms of the structure of the graph and the manner in which the decision elements are imbedded in that structure. Finally, an equivalence problem for program graphs is formulated and a solution to this problem is presented.

ACKNOWLEDGEMENT

This report is the unaltered thesis submitted by Jorge E. Rodriguez in September, 1967, in partial fulfillment of the requirements for the degree of Doctor of Science in Electrical Engineering at the Massachusetts Institute of Technology.

The author wishes to thank the following: Professor Jack B. Dennis who, as thesis supervisor, provided invaluable advice and guidance throughout the course of this research; Professor Joseph Weizenbaum, Professor Chung L. Liu, and Mr. Douglas T. Ross for their assistance and suggestions as readers. (Mr. Ross was also Head of the Computer Applications Group at the Electronic Systems Laboratory, where the author worked as a Research Assistant, who throughout the years had supplied much insight, help, and encouragement); Professor Leonard A. Gould who, as his faculty advisor since his early days at the Graduate School, had provided invaluable academic and personal advice; Dr. Jacob Katzenelson for his timely help and encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

PREFACE

The goals of generalized computer-aided design, being synonymous with generalized man-machine problem-solving, place the most stringent requirements on underlying foundations and implementation techniques. As increasingly elaborate and complex applications are contemplated, it becomes clear that substantial inroads must be made to deepen our fundamental understanding of computation itself. Ultimately it must be possible to prove the correctness of a program, for no conceivable technique can provide an adequate basis for debugging; it must be possible to transform a proposed computation automatically from one formulation to another radically different formulation, with firm knowledge that the two forms are in a useful way equivalent; it must be possible to design, analyze, and compute using entire computational processes themselves as data, for manual composition of constructs of such vast complexity will be beyond human comprehension. It was in the spirit of these convictions that the research described in this report was undertaken.

Early in the preliminary investigation, it became clear that before any questions of equivalence or operations of transformation could meaningfully be posed, a rigorous, deterministic, and elegant model of a computational process itself was required. The model had to be independent of any artifacts of existing programming language characteristics and had to exhibit in an inherently simple and natural form only the essential relations between data and operators on data, from which any computational process is composed. The "program graph" model introduced here is a major contribution which meets the most basic criteria. Since the model is based directly upon "data dependency" relations, it enjoys the essential simplicity needed to assure its adequacy as a general model. Also, the rigorous formulation enables determinism of the model to be proved. Finally, some initial attempts to address questions of equivalence and transformation lend credence to the viewpoint that further elaborations and refinements can lead toward the desired basis for a mathematics for computational processes. Already the trends in this direction are taking shape in a number of related theses and studies listed at the end of this preface.

PREFACE (Contd.)

In view of the abstract nature of the model, and the fact that such important features as data structures are included only in the most degenerate form, it is clear that it still will be some time before these developments can have a direct impact on the practical matters of constructing man-machine systems. Many aspects can, however, be extracted and can be cast in terms compatible with some of the more advanced aspects of programming language semantics and compilation of optimized machine code. Hopefully such application attempts combined with the theoretical advancements will accelerate the pace at which these vital matters can be pursued.

Douglas T. Ross

Douglas T. Ross
Head, Computer Applications Group
January, 1969

1. Dennis, J.B. Programming Generality, Parallelism and Computer Achitecture. (Submitted for publication to the Journal of ACM)

2. Luconi, F.L. Asynchronous Computational Structures. MAC-TR-49, Project MAC, MIT (1968)

3. Slutz, D.R. The Flow Graph Schemata Model of Parallel Computation. MAC-TR-53, Project MAC, MIT (1969)

I. INTRODUCTION

A. SUMMARY

This paper presents a computational model called program graphs which makes possible a precise description of parallel computations of arbitrary complexity on non-structured data. In the model, the computation steps are represented by the nodes of a directed graph whose links represent the elements of storage and transmission of data and/or control information. The activation of the computation represented by a node depends only on the control information residing in each of the links incident into and out of the node. At any given time any number of nodes may be active, and there are no assumptions in the model regarding either the length of time required to perform the computation represented by a node or the length of time required to transmit data or control information from one node to another. Data dependent decisions are incorporated in the model in a novel way which makes a sharp distinction between the local sequencing requirements arising from the data dependency of the computation steps and the global sequencing requirements determined by the logical structure of the algorithm.

The concept of the state of a program graph is introduced and it is proved that every program graph represents a deterministic computation, i. e. that the final state of each computation started from the same initial state is unique. Computations which do not terminate properly are defined in terms of the concept of hang-up state. Methods of analysis are developed, and necessary and sufficient conditions for the absence of hang-up states are obtained. These conditions are interpreted in terms of the structure of the graph and the manner in which the decision elements are imbedded in that structure. Finally, an equivalence problem for program graphs is formulated and a solution to this problem is presented.

The model may be useful in a variety of problems including: the analysis and transformation of computer programs to meet some desired criterion, e.g. reduce the amount of space required, or increase the speed of operation, or both of these objectives; the assignment and sequencing of computations in parallel processor computer systems; and the design of sequencing and control units for parallel computation, in particular, the results of this paper are directly applicable to the design of macro-modular systems.[3, 18, 22]

B. REVIEW OF RELATED WORK

Graphs have been used to represent computations since the early days of computers. Most of these representations are strictly sequential and can be generally classified as flow charts. In a flow chart, a node represents either an operational element or a decision element, and an arc of the graph denotes flow of control from one node to another. At any one point, control resides in precisely one of the nodes. Flow charts have been studied by a number of workers in the field.[2, 4, 5, 8, 9, 11, 13, 20] In the context of this paper, these studies are not directly relevant and therefore we proceed to review only those models which have a direct bearing on the subject of parallel computations.

C. A. Petri[19] has proposed an approach to the description of transmission and transformation of information in discrete systems in which time is introduced only as a local relation among local states. In Petri's formalism, a system is represented by an undirected graph in which each node is a <u>connecting</u> element which <u>binds</u> together (relates) <u>objects</u> contained in places. Each arc of the graph is a place. A node represents a switching element of a type given by its label. The behavior of each type of element is given by a transition table and is influenced only by those objects assigned to places attached to the node. Petri claims that it is possible to construct conflict-free, deterministic networks corresponding to Turing machines using switching elements defined over the objects 0 and 1.

A. W. Holt[7] has introduced a formalism called -theory for describing discrete information systems. In a -theory the characteristics and behavior of a system are expressed by means of relations of parts. The state of a system is formalized as a finite undirected graph. The nodes of the graph represent system parts and every node is labelled with a node type. The arcs represent relations between two parts. A -theory consists of a *-grammar*, a list of *event types*, and a list of *observables*. The grammar establishes the laws of local context for the node types, i. e. what node types must or may relate and how. The list of event types establishes the laws of local change, i. e. which relations of parts bring about which changes in relations of parts. The list of observables defines which relations of parts are capable of conditioning events in the environment of a system of the class. Changes on the state of a system are defined by means of a *simulation rule* which effects the changes specified by an *admissible* subset of *applicable* event types. A subset of applicable event types is admissible if it is *consistent* and *lossless* which means that the state resulting from the changes obeys the rules of the grammar, no two events bring about conflicting changes, and every applicable event not contained in the admissible subset remains applicable after the changes are effected.

E. C. Van Horn[23] has proposed a class of abstract machines for coordinated multi-processing or MCM. An MCM consists of a set of *cells*, a *scheduler*, and a *count matrix*. The state of an MCM is defined to be the contents of the calls plus the contents of the count matrix. Each cell may behave either like a passive memory element or an active computing element. An active cell, called a clerk, may perform a sequence of transactions under the control of the scheduler. Each cell has its own table of transactions. There are five types of transactions; two transactions read and write on cells, three transactions modify the count matrix. Reading of the count matrix is performed by the scheduler to determine which clerk cells are enabled, i. e. can perform one transaction. Van Horn has shown that the behavior of any MCM is asynchronously reproducible.

R. M. Karp and R. E. Miller[10] have introduced a model for parallel computations, called computation graphs. A computation graph is a directed graph in which nodes denote operations and branches denote storage elements where results are placed in first-in-first-out queues. Associated with each branch are four non-negative integers A_p, U_p, W_p, and T_p where $T_p \geq W_p$. For a branch directed from node n_i to node n_j, these parameters are interpreted as follows: A_p is the number of data words initially in the queues; U_p is the number of words added to the queue upon completion of the operation associated with n_i; and T_p is a threshold giving the minimum queue length of the branch before the operation of n_j is initiated. Karp and Miller show that computations represented by these graphs are deterministic. They also give a test to determine whether a computation terminates, and study properties of the data queues associated with the branches, deriving conditions for the queue lengths to remain bounded.

G. Estrin and R. Turn,[6] and D. Martin[14] have introduced a directed graph model for computer programs in which the vertices represent computational tasks and the arcs represent data dependency between nodes. In this model, the conditions for the initiation of the computation denoted by a vertex is expressed by writing a boolean expression in terms of boolean variables associated with the arcs incident into the node. A boolean variable associated with an arc is true when the data in that arc becomes available. A computation may be initiated when the boolean expression of the corresponding node. called the vertex input control, is true. There are three types of vertex input control: 1) Conjunctive, 2) disjunctive, and 3) compound. Vertices with conjunctive input control may be initiated only when all input data are available. Vertices with disjunctive input control may be initiated only when precisely one set of input data (i. e. one arc) becomes available. The compound input control is a combination of the other two. Vertices also have output control which is used to specify the program flow from a vertex to a subset of its immediate successors. A vertex with conjunctive output control simultaneously makes data available at all of the arcs incident out of the vertex. A vertex with disjunctive output control makes data available at precisely one of its output arcs. Thus, it may be seen that vertices with disjunctive output control effectively perform data dependent decisions to control the program flow. The model can properly represent only cycle free graphs. It has been used primarily as a tool for the a-priori assignment and sequencing of computation in parallel processor systems.

The model presented in this paper is a direct extension and formalization of the model of Estrin and Turn using notational techniques introduced by Petri. In extending the model of Estrin and Turn, the concept of pure control information is introduced so that decision elements do not transmit any data but only enabled or disable computations. Furthermore, node types have been introduced which make possible the unambiguous specification of cycles. The notational techniques of Petri have been very useful for precisely describing the behavior of the model in terms of local information alone. In this respect, the similarities between this form of specification of events and that proposed by Holt should be noticed.

D. Muller and W. Bartky[17] have developed methods for the analysis of asynchronous sequential circuits. These methods proved of considerable value in the analysis of the determinism of program graphs.

C. OUTLINE

The material is organized as follows: in Chapter II the model is presented and it is proved that every computation represented by the model is deterministic. Chapter III begins with a detailed consideration of the function of each type of node together with reasons for the choices made in the specification of their behavior. This is followed with the introduction of the concept of hang-up state and a study of the conditions which give rise to these states. Chapter IV formulates an equivalence problem and presents a solution to it. This is followed with a brief consideration of some simple equivalence preserving transformations. Finally, Chapter V contains the conclusions and recommendations of this research.

II. THE MODEL

A. INTRODUCTION

This chapter presents a model for computational processes called program graphs. A program graph is both a denotation of an algorithm and a realization of this algorithm by a process. The linguistic device used to denote operations and the links of the graph denote data and/or control flow among the operations. The realization of the algorithm by a process is accomplished by assigning certain rules of behavior to the program graph elements (nodes and links). These rules of behavior are such that each program graph is a special-purpose deterministic machine which realizes an algorithmic process. The term deterministic machine as used in this paper means that the behavior of the machine is always the same whenever identical data is presented to it at its input terminals.

The material is organized as follows: Section B gives background for the model and the general viewpoint adopted in its formulation. Section C gives some necessary notation and introduces the elements used in constructing program graphs. Section D specifies the syntax for constructing program graphs. Section E specifies the interpretation of program graphs and gives illustrative examples. Finally Section F contains a proof of the determinism of program graphs.

B. BACKGROUND AND VIEWPOINT

The formulation of program graphs as a computational model has been motivated by the common observation that a large fraction of the sequential constraints of a process can be completely specified by explicitly indicating the data dependency among the different parts of the process.

In other words, if the results of sub-process A are data for sub-process B, then A must be performed before B. If control is defined as that quantity which determines the sequencing aspects of a process, then data flow always carries with it control flow information. An obvious advantage of this manner of process specification is the absence of unnecessary sequential constraints, and thus the immediate appearance of any parallelism inherent in the process. This can be seen in the example of Figure 2.1 which shows the representation of an algebraic computation by means of a directed graph in which the nodes denote operations and the directed links denote the data dependency. The potential parallelism of the two '+' nodes is clearly evident.

Data dependency is not sufficient to specify all the sequential constraints of a process, however. Most computations include decisions which affect the sequencing of the process without introducing explicit data dependency. We can think of the simplest form of decision, a binary decision, as generating pure control information which selects one out of two possible sequences. The decision does not affect the result of the computations of either sequence but only whether or not the sequences of any parts of them should be performed. Figure 2.2 shows how we might represent an ALGOL conditional expression. The diamond-shaped node denotes a decision selecting one of the additions to be performed as a prelude to the multiplication operation. (Open arrowheads denote pure control flow.)

A computational model which exploits the control aspects of data dependency and thereby places in sharp contrast the unique functions of pure control information is a potentially useful base for exploring transformations of algorithms which preserve input/output relations, logical design of asynchronous machines, and similar areas.

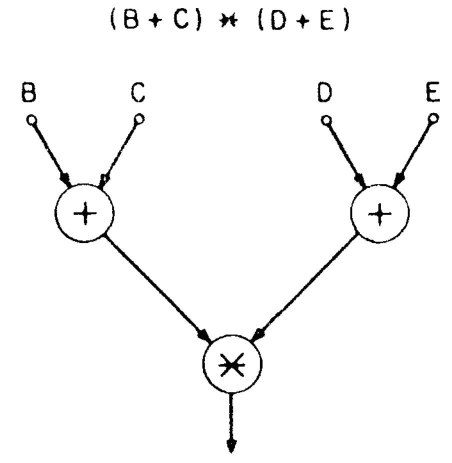


Fig. 2. 1 Representation of an Algebraic Computation by a Program Graph

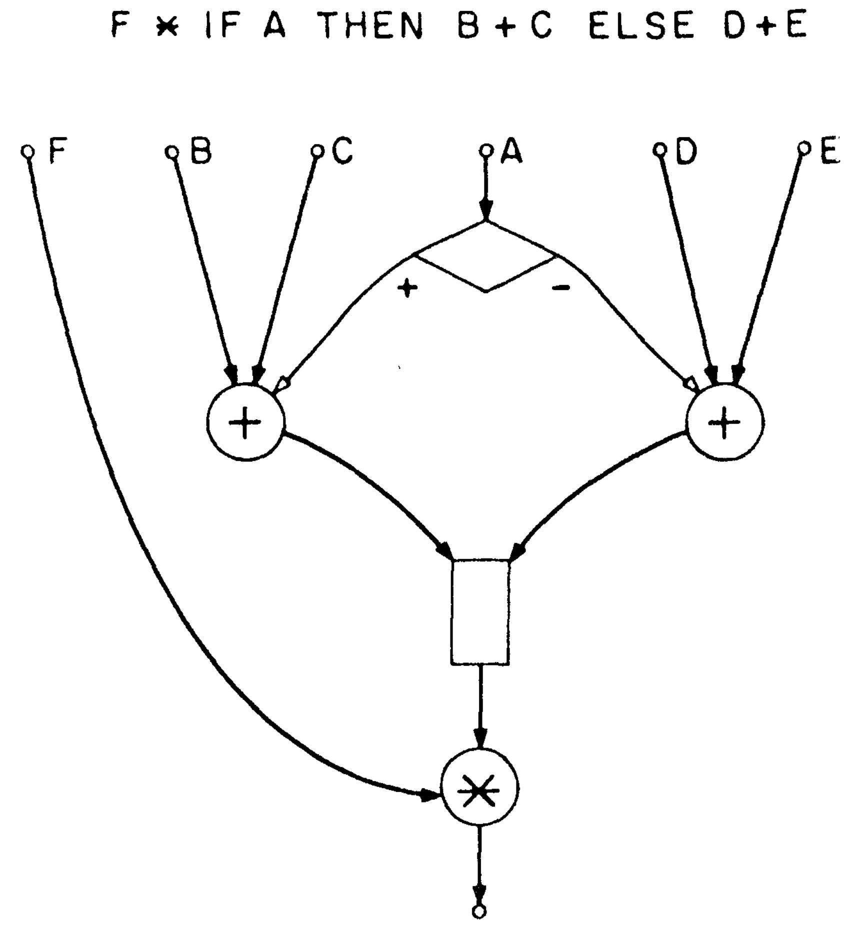


Fig. 2.2 Representation of a Conditional Expression by a Program Graph

Directed graphs are a natural choice for representing the dependency relations we are interested in, but the static relationships represented by a graph are not sufficient to unambiguously determine dynamic behavior, particularly when dealing with cycles. Dynamic behavior of the process represented by a graph is a crucial question which cannot adequately be handled by introducing a series of ad hoc global rules of interpretation. Therefore, it was decided to formalize a program graph as a formal machine, by having well-defined rules of behavior associated with each element of the graph, i. e. the nodes and links. By making these rules depend only on local information, i. e. by making them independent of the over-all structure of the graph, we achieve two things:

1. Any graph constructed following a minimum of local interconnection rules represents a process with unambiguous behavior.
2. Each operation proceeds asynchronously with all others, so that any degree of parallelism, anticipating computations or control, can be expressed by the formalism.

C. THE CONSTITUENTS OF PROGRAM GRAPHS

A program graph formally represents a computing machine. The computational elements of the machine are represented by the nodes of the graph, and elements for storage and transmission of data and control information are represented by the links of the graph. In what follows *node* shall be synonymous with computational element and *link* shall be synonymous with storage-and-transmission element.

There are two types of links - a *data link* and a *control link*. Associated with both data and control links is a quantity called the *link-status.* At any given time the link-status of a link assumes precisely one out of the four possible values -1, 0, 1, 2. These values will be called *disabled*, *idle*, *enabled*, and *blocked*, respectively.

Data links have in addition to link status, a property called *data contents*. *No restrictions are placed on the nature of the data contents of a link.* In the representation of program graphs data links are shown as heavy lines with black arrows, while control links are shown as light lines with open arrows.

There are seven types of nodes differing from each other either in the kind of computation performed or in the logic used to *activate* the node. Nodes have specific points of attachment called *connectors*. The connectors of a node are distinguished as being either *input* or *output* connectors. Furthermore an input or output connector may be a data connector or a control connector. Data connectors are attached only to data links. Similarly, control connectors are attached only to control links.

When the computation represented by a node is being performed, we say that the node is *active.* The activation of a node is determined by the link status of the data and control links attached to the connectors of the node. For brevity we often refer to the 'status of a connector' meaning the link status of the link attached to the connector, even though a connector does not properly have a status.

A node is in an active configuration when the statuses of its connectors is such that the node becomes active. The occurrence of an active configuration initiates a transition of arbitrary time length. Upon completion of the transition, the status of each connector (and perhaps the data contents of attached data links as well) are changed in a way specified by the transition table for that type of node.

The transition table for a type of node specifies all of the active configurations of the node in terms of the statuses of the input and output connectors. For each active configuration the transition table also specifies the final configuration i.e., the status of each connector after completion of the transition, and the change, if any, of the data contents of output data links.

The specification of the transition table is simplified by certain conventions and a notation adopted from the work of Petri.[19] Each connector of the node is assigned a sequential number. A configuration is then represented by writing the link status values from left to right in the sequence determined by the ordering assigned to the set of connectors. Thus the configuration 1 1 0 corresponds to a node with three connectors in which connector number 1 is in status 1, connector number 2 is in status 1, and connector number 3 is in status 0. A transition is denoted by writing the active configuration followed by ' $\rightarrow$ ', followed by the final configuration. If a transition changes the data contents of some link, an expression is written after the final configuration defining the new new value. Thus the expression $\$1 \leftarrow f_1 (\$2, \$3)$ means that the data contents of connector 1 is replaced with the result of applying function f_1 to the data contents of connectors 2 and 3.

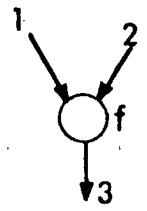
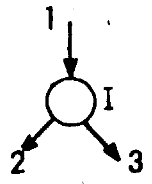
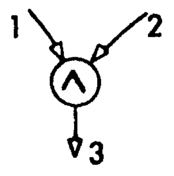
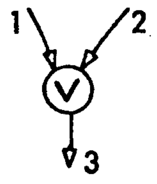
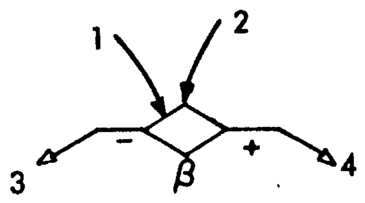
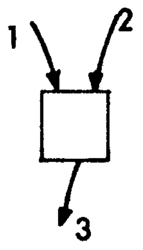
Operator			
Data operator			
Function		1 1 0 → 0 0 1	$\$3 \leftarrow f(\$1,\$2)$
		1 -1 0 → 0 0 -1	
		-1 1 0 → 0 0 -1	
		-1 -1 0 → 0 0 -1	
Identity		1 0 0 → 0 1 1	$\$2,\$3 \leftarrow \$1$
		-1 0 0 → 0 -1 -1	
Control operator			
And		1 1 0 → 0 0 1	
		1 -1 0 → 0 0 -1	
		-1 1 0 → 0 0 -1	
		-1 -1 0 → 0 0 -1	
Or		1 1 0 → 0 0 1	
		1 -1 0 → 0 0 1	
		-1 1 0 → 0 0 1	
		-1 -1 0 → 0 0 -1	
Selector		1 1 0 0 → 0 0 -1 1	if $\beta(\$1,\$2) = \underline{\text{true}}$
		1 1 0 0 → 0 0 1 -1	if $\beta(\$1,\$2) = \underline{\text{false}}$
		1 -1 0 0 → 0 0 -1 -1	
		-1 1 0 0 → 0 0 -1 -1	
		-1 -1 0 0 → 0 0 -1 -1	
Junction		1 -1 0 → 0 0 1	$\$3 \leftarrow \$1$
		-1 1 0 → 0 0 1	$\$3 \leftarrow \$2$
		-1 -1 0 → 0 0 -1	

Table 2.1 The Transition Tables of Program Graph Nodes

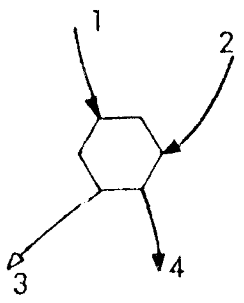
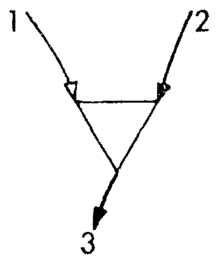
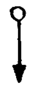
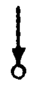
Loop Junction		1 0 0 0 → 2 0 2 1	\$4 ← \$1
		1 1 0 0 → 2 1 2 1	\$4 ← \$1
		1 -1 0 0 → 2 -1 2 1	\$4 ← \$1
		-1 0 0 0 → 2 0 2 -1	
		-1 1 0 0 → 2 1 2 -1	
		-1 -1 0 0 → 2 -1 2 -1	
		2 1 0 0 → 2 0 -1 1	\$4 ← \$2
		2 -1 0 0 → 0 0 1 0	
Loop Output		1 1 0 → 0 0 1	\$3 ← \$1
		1 -1 0 → 0 0 -1	
		-1 1 0 → 2 0 0	
		-1 -1 0 → 2 0 0	
		2 1 0 → 0 1 0	
		2 -1 0 → 0 -1 0	
Input Terminal			
Output Terminal		1 → 0	
		-1 → 0	

Table 2. 1 The Transition Tables of Program Graph Nodes

Table 2.1 contains the transition tables for each type of node. A brief description of the characteristics of each type follows.

1. Operators

Operator nodes denote functions of one or more input arguments and one or more output values. An operator node all of whose inputs and outputs are data links is called a data operator. Similarly an operator all of whose inputs and outputs are control links is called a control operator. Every operator is either a data or a control operator. The data operator representing the identity function will always be denoted by the symbol 'I'. The control operators representing logical disjunction and conjunction will always be denoted by the symbols '$\vee$' and '$\wedge$' respectively. These are the only control operators allowed in a program graph.

2. Selectors

Selector nodes denote decision-making elements. A selector is associated with a predicate function. All inputs of a selector are data links and there are precisely two control outputs. When the predicate associated with a selector is applied to the data contents of the inputs the result is to place one output connector in enabled status and the other in disabled status.

Selectors and data operators may have a control input connector in addition to their data connectors. The function denoted by a selector or data operator is applied only for those active configurations in which all input connectors are in enabled status.

3. Junctions

Junction nodes merge two or more sources of data. A junction transmits the data contents of at most one of its data inputs to its unique data output.

4. Loop Junction

Loop junction nodes are used to form cyclic structures in a program graph. Their function can be roughly described as follows: Suppose one has an iterative process and A, B, C represent the quantities on which the iteration depends, then to every oneof these variables there will correspond a loop junction. Input connector 1 is used to 'assign' the initial value of the variable and input connector 2 is used to 'assign' the new value of the variable on every iteration. I shall often refer to input connectors 1 and 2 as the initial and feedback connectors of the loop junction.

5. Loop Output

Loop output nodes are used in conjunction with loop junctions to precisely define what is to be considered the result of an iterative process. Proper usage of loop output nodes requires that their control input connector be attached to the control output connector of a loop junction. Some of the examples in the next section will serve to clarify the relationship between loop junctions and loop outputs.

6. Input Terminals

Input terminals are nodes with no inputs and precisely one output.

7. Output Terminals

Output terminals are nodes with no outputs and precisely one input.

D. THE CONSTRUCTION OF PROGRAM GRAPHS

This section sets forth the rules for constructing program graphs and provides a few examples of the use and abuse of these rules.

A program graph is a finite set of input terminals, output terminals, and nodes interconnected by data links and control links according to the following rules:

1. The root of a data link must be attached to an input terminal or to a data output connector and its tip must be attached to an output terminal or to a data input connector.
2. The root of a control link must be attached to a control output connector and its tip must be attached to a control input connector.
3. Every input connector of a node must be attached to some link.

Figures 2.3, 2.4, 2.5, and 2.6 are examples of program graphs. Each example is provided with an Algol-like description of the algorithm.

Figure 2.3 illustrates how functional composition is represented by the interconnection of data operators. Figure 2.4 illustrates the use of loop junctions and loop outputs in a simple iterative process. The arrangement illustrated in the figure is used whenever an output value, e.g. "ans" is desired only upon completion of the iterative process. Quite often iterative processes are used which do not behave this way, but instead output values are produced on every iteration. A common example of this type of iteration is that of a program producing lines of output. Figure 2.5 shows another example of this situation which is perhaps not as obvious as the 'output feeder' case. The flow chart representation of this algorithm shows a single loop however, upon separating the data flow from the control flow in the program graph representation, two distinct loops arise: a counting loop (loop junction labelled i) and a summation loop (loop junction labellet t) with the counting loop effectively controlling the iterations of the summation loop. It should be noted that this type of relationship between two or more loops often gives a clue to one of two forms of parallelism among sub-parts of the process:

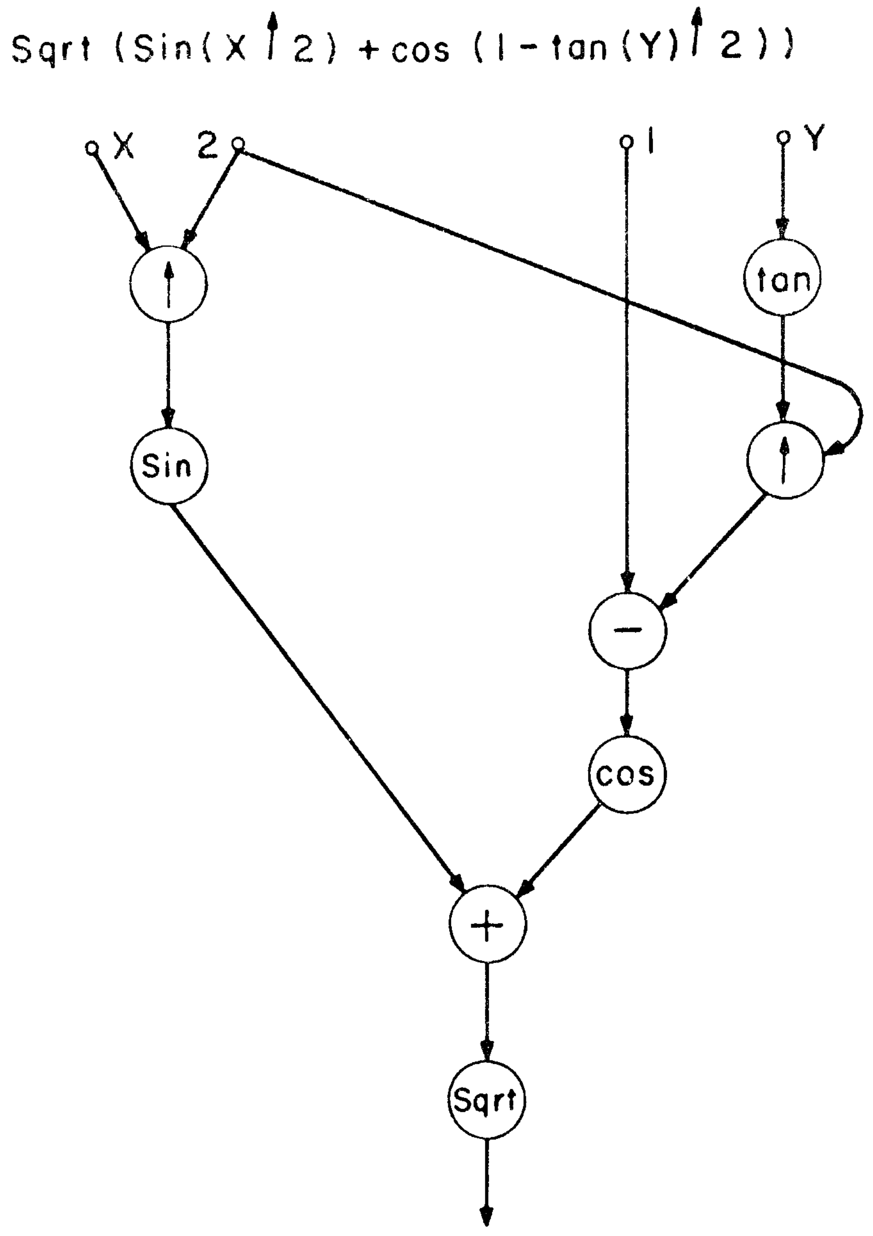
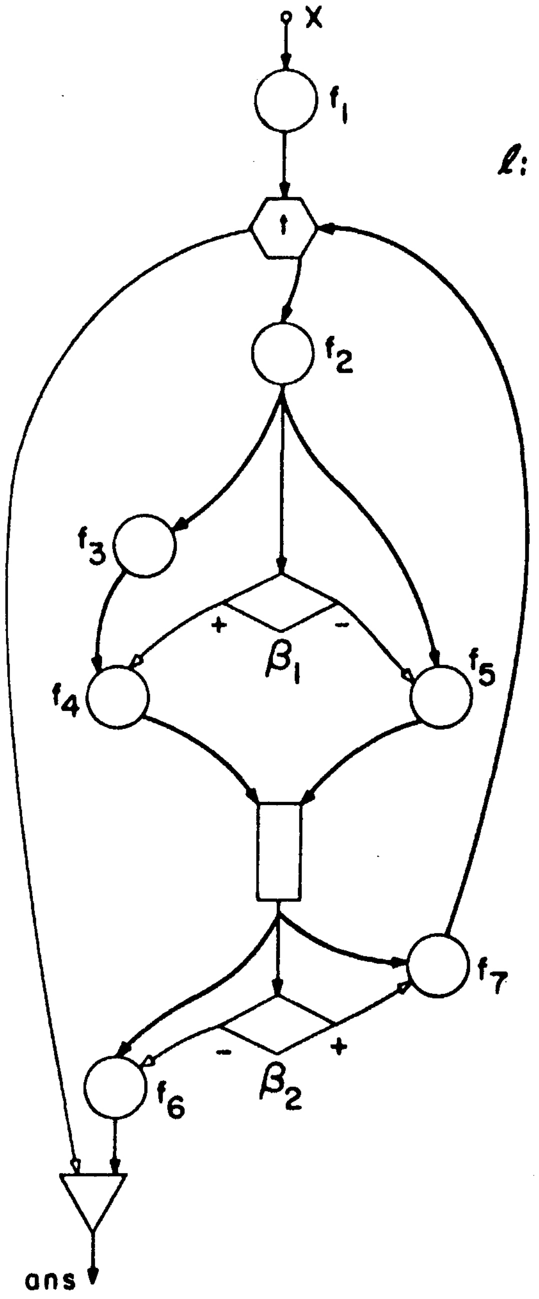


Fig. 2.3 Functional Composition in a Program Graph



```
     t = $f_1(x)$;
ℓ:   z = $f_2(t)$;
     v = $f_3(z)$;
     t = if $\beta_1(z)$ then $f_4(v)$ else $f_5(z)$;
          if $\beta_2(t)$ then begin
                    t = $f_7(t)$;
                    goto ℓ end
ans = $f_6(t)$;
```

Fig. 2.4 A Simple Iterative Process

1. Horizontal parallelism meaning simultaneous operation of individual computations without imposing requirements on the sequence of using the results.
2. Vertical parallelism meaning simultaneous computations with certain sequential constraints as to the use of the results of each computation.

In the example of Figure 2. 5 horizontal parallelism will be possible given the extra knowledge that addition is commutative and associative, otherwise we have to be content with vertical parallelism, i. e. initiation of as many functions as possible, but adding them in the specified sequence.

The final example of this section, shown in Figure 2. 6, illustrates one single loop (in the flow chart sense) with more than one loop junction. Recall that each loop junction represents one variable of the iterative process and a loop junction is required even for those quantities which are not changed within the loop because there is no permanent storage in a program graph, as is the case with the quantity denoted by A in Figure 2. 6.

E. THE EXECUTION OF PROGRAM GRAPHS

The transition tables for program graph nodes determine the dynamic behavior of a program graph by specifying whether or not a node should be activated, and if it is activated, whether or not certain data transformations should take place.

Unless otherwise specified, all links are initially in the IDLE(0) status. Execution of a program graph begins when enough input terminals have been placed in ENABLED(1) status to produce an active configuration on some node of the graph. For simplicity, however, we usually assume that there is time, t_o, at which all input terminals are placed in ENABLED status simultaneously.

```
t = o;
for i = 1 step 1 until N do t = t + A(i);  ans = t;
```

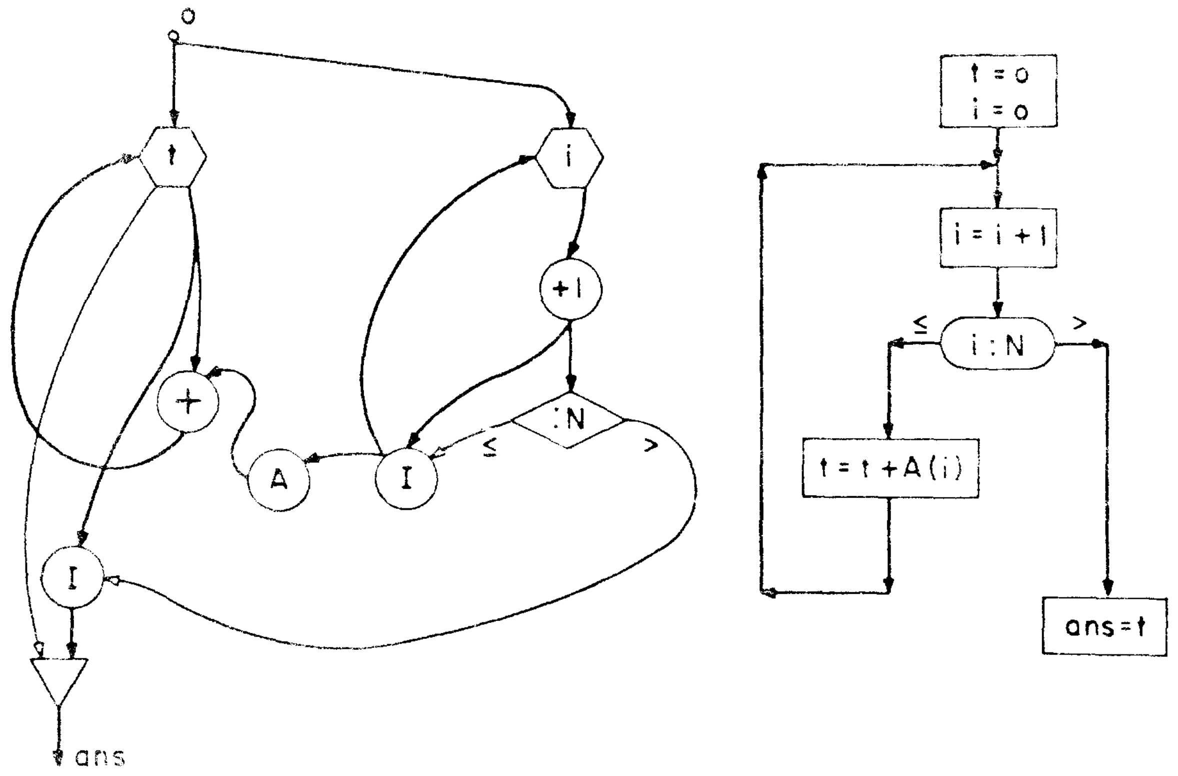
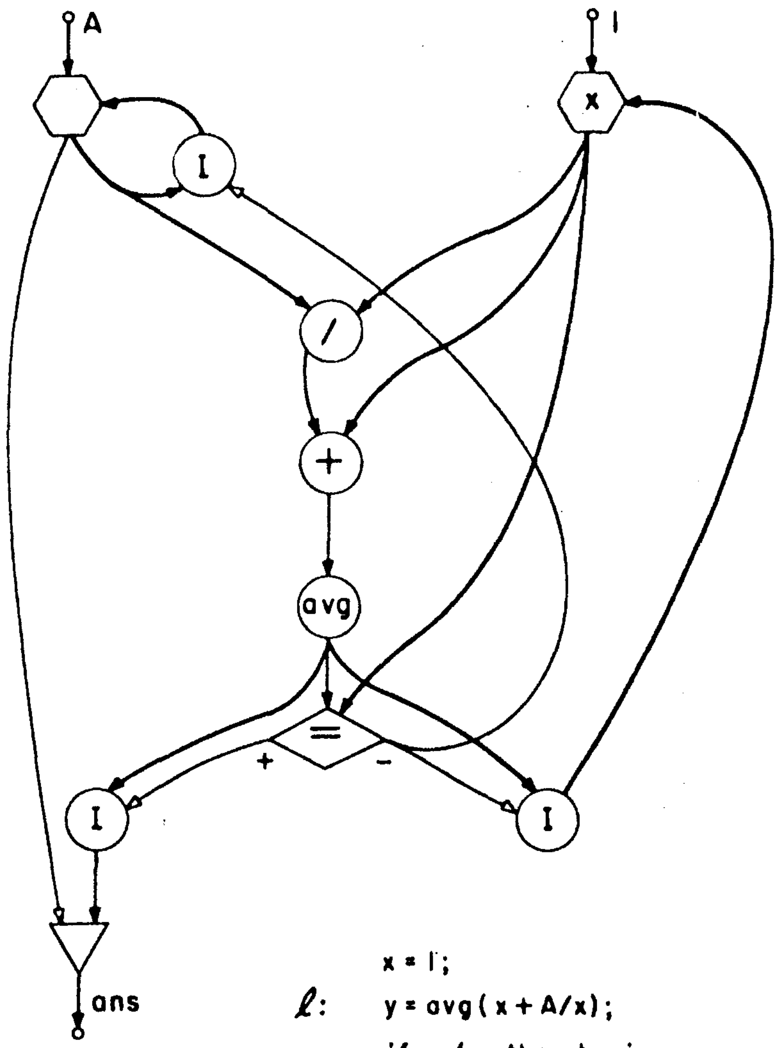


Fig. 2.5 Program Graph and Flow Chart of an Iterative Process



```
        x = I;
ℓ:      y = avg(x + A/x);
        if x ≠ y then begin
                    x = y;
                    goto ℓ end;
      ans = y;
```

Fig. 2. 6   A Single Loop with Multiple Loop Junctions

A simple example will serve to illustrate how the execution of a graph takes place. Figure 2.7 shows a program graph at various stages of execution. In these figures, the number written to the right of each link is the link-status at the time of the snapshot; the data contents of the link is written to the left of the link as a functional expression. A '*' next to a node signifies that the node is in an active configuration.

Figure 2.7a shows the state of the graph shortly after time, t_o, with operators labelled f_1 and f_2 active. In Figure 2.7b operator f_1 has completed its transition enabling f_3 to become active while f_2 still has not finished. The final snapshot Figure 2.7d shows the state of the graph upon operator f_6 having completed its transition. Note that all links have been restored to the IDLE status.

Figure 2.8 shows a complex situation, arising due to the presence of a loop. Figure 2.8a, b, c are snapshots during the execution of a non-final iteration showing the state of the graph before the activation of selector β_2 at two successive time intervals after the completion of the selector transition which enables the output link labelled '-'. Figure 2.8d, e, and f is a similar sequence, but in this case β_2 has enabled the output link labelled '+' signalling the end of the iterative loop. These sequences serve to clarify the purpose of the link-status value blocked which only affects loop junctions and loop outputs. While the loop is in progress, the initial connector of the loop junction is in blocked status, effectively blocking any attempt to initiate a new loop before finishing the current one. In the meantime the loop output blocks any signals to the outside world. When the loop junction receives an indication that the cycle has finished (DISABLED status of the feedback connector), it signals the loop output that it is alright to allow an output to be produced. Again, note in Figure 2.8f that upon completion of the iterative process all link-status values have been restored to the IDLE status.

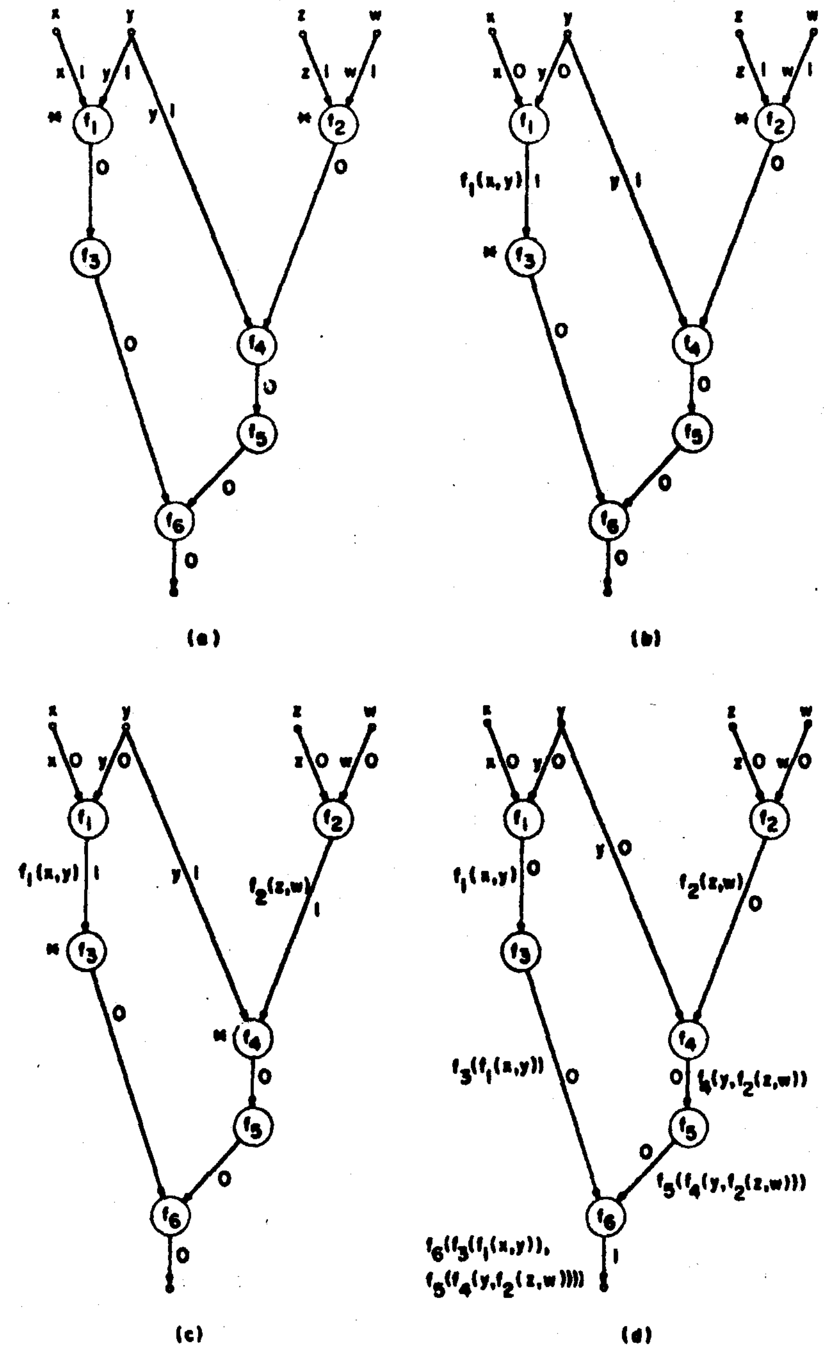


Fig. 2.7 States During the Execution of a Cycle Free Program Graph

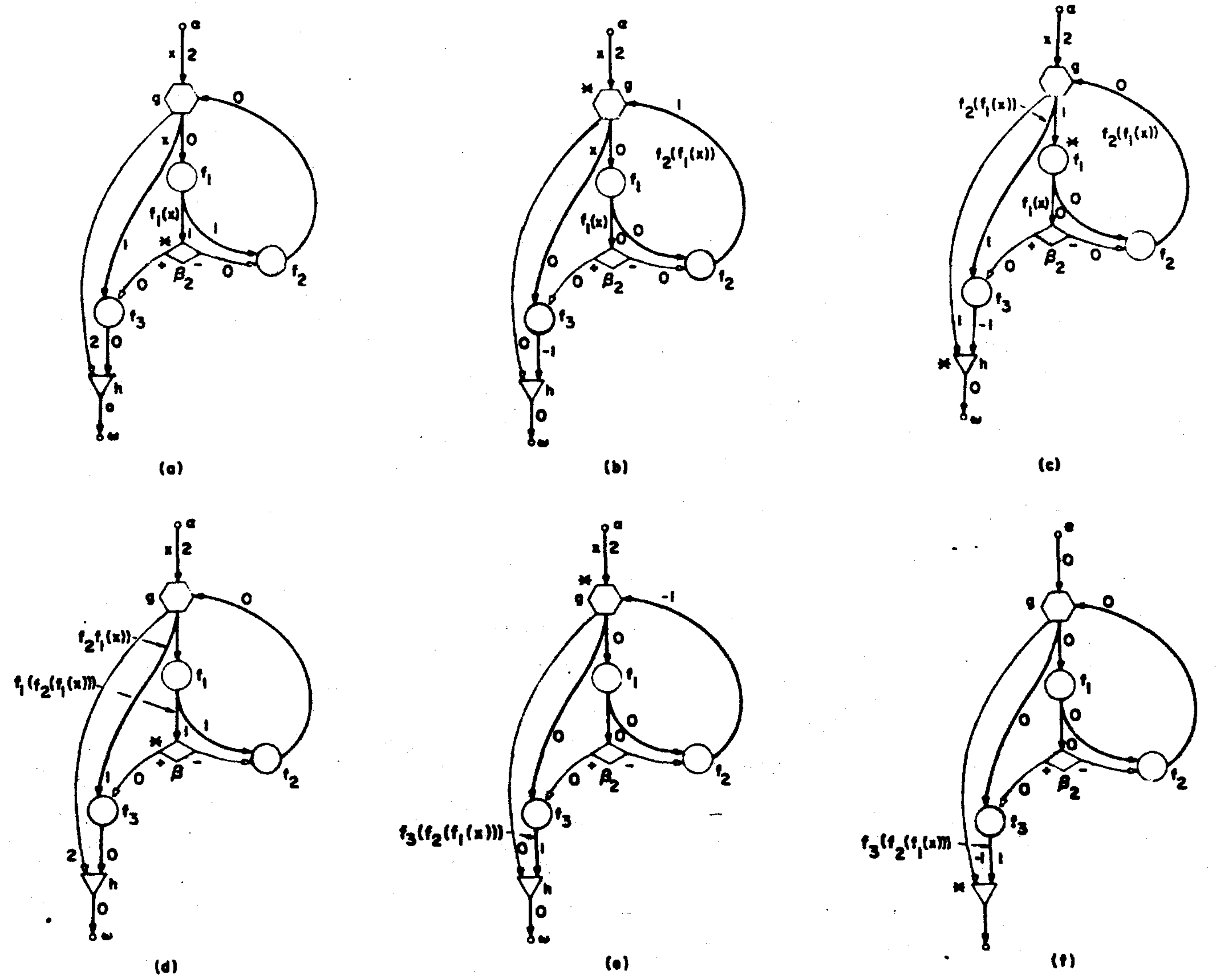


Fig. 2.8 States During the Execution of a Cyclic Program Graph

F. THE DETERMINISM OF PROGRAM GRAPHS

The rules for interconnecting nodes set forth in Section D do not place any restriction on the topology of the resulting program graph. Furthermore, it has been specifically assumed that we have no knowledge of the time elapsed between the initiation and the completion of a transition. These two situations combined can result in the specification of a process whose behavior is unpredictable in the sense that two distinct executions of the process with the same set of data values supplied at the input terminals may produce a different set of results.

The behavior of a program graph is determined by the link-status and the data contents of the links of the graph. We shall denote these two properties of a data link by an ordered pair (s, d) where s can take any of the link-status values -1, 0, 1, 2 and d is a functional expression, e.g. $f_1(x, f_2(y, z))$ denoting the value of the data contents. In the case of a control link, data contents is not defined so that only the link-status value s will be used.

If n is the number of links in a graph, then the state of the graph is an ordered n-tuple $A = (a_1, a_2, \ldots a_n)$ where each state variable a_i is either an (s, d) pair or an s depending on whether a_i is associated with a data link or a control link. Two states A and B are equal if and only if for all $1 \leq i \leq n$, $s_{a_i} = s_{b_i}$ and $d_{a_i} = d_{b_i}$. The state determines which nodes of the graph are in an active configuration. We make this explicit by associating with each node an n-tuple as follows:

> The ith element of the n-tuple for a node f is zero unless the link associated with the corresponding element of the state is attached to f. In this case the ith element is the number of the connector of f to which the link is attached.

For example if the third, fourth, and sixth links of the state vector are attached to the second, third, and first connectors of f, the n-tuple for f is (0, 0, 2, 3, 0, 1, 0, . . .). The n-tuple associated with a node f will be called the connectivity vector of f and is denoted by C_f.

A state A changes into a state A' upon completion of the transition of one of the active nodes of A. The components of the new state A' are determined by those of the old state A and the transition table for the node. If f is any node, the notation $A' = A \times C_f$ is interpreted as follows:

1. If the ith element of C_f is zero then the ith element of A' is the same as the corresponding element of A.
2. If elements of A corresponding to non-zero elements of C_f form an active configuration of f, the corresponding elements of A' are obtained by using the applicable transition of f.
3. Otherwise these elements are not changed and A' = A.

We now introduce the concept of the possible "next" states A' of a state A by means of the relation $\mathcal{R}$.

Definition 2.1 A state A' follows a state A iff A' results from the completion of the transition of none, one, or more active nodes of A. If A' follows A we write $A \mathcal{R} A'$. We say A' is a next state of A.

Definition 2.2 A final state of a program graph is a state in which no node is active.

From the definition of $\mathcal{R}$ and final state it is clear that a state A is final if and only if $A \mathcal{R} A'$ implies A = A'.

During execution, a program graph passes successively from one state to one of its next states.

Definition 2.3 An execution sequence of a graph is a sequence of states $A_0, A_1 \ldots A_k$ such that $A_i \mathcal{R} A_{i+1}$ and $A_i \neq A_{i+1}$.

Definition 2.4 An execution sequence A_0, A_1 ... A_k is terminal if A_k is a final state. If σ is an execution sequence, the length of σ denoted by $\ell(\sigma)$ is the number of states in the sequence.

For a given initial state, a program graph may exhibit several execution sequences depending on the relative speeds of the nodes. The problem of the speed-independence of the final state (when it exists) of a program graph with respect to an initial state A_0 is crucial to the justification of the model. The only requirement that we place on the behavior of a program graph is that every node transition is an indivisible operation, i.e., once a transition begins the indicated changes of status take place simultaneously. This assumption does not say that a node placed in active configuration performs the corresponding transition immediately. Quite to the contrary, we do not place any restrictions in the time interval elapsed from the time an active configuration occurs to the time a transition is actually performed. This of course raises, among others, the possibility of a node entering and leaving an active configuration without performing any transition.

Theorem 2.1 establishes a property of the $\mathcal{R}$ relation which, as we shall see, is sufficient to guarantee the uniqueness of the final state of a program graph for any assignment of elapsed times to the nodes of the graph. This property of the $\mathcal{R}$ relation is closely connected to that existing among the states of semimodular asynchronous circuits as described by Muller and Bartky.

Lemma 2.1 Let A_0 be a non-final state of a program graph P and let A_1 and A_2 be any two states of P such that $A_1 = A_0 \times C_{f_1}$, $A_2 = A_0 \times C_{f_2}$. Then, there exists a state A_3 such that $A_i \mathcal{R} A_3$ $0 \leq i \leq 2$.

Proof: From the definition of the $\mathcal{A}$ relation, if $A_0 = A_1 = A_2$ then A_0 satisfies the requirements for A_3. Similarly if $A_0 \neq A_1 = A_2$ then $A_1 = A_2 = A_3$ satisfies the conditions of the lemma Therefore assume that A_0, A_1, and A_2 are distinct. This means that there are at least two active nodes f_1 and f_2 in A_0. We claim that the state arising from the simultaneous completion of the transitions of f_1 and f_2 satisfies the conditions for A_3.

First we note that the active configurations of program graph nodes as shown in Table 2.1 place certain restrictions on the possible inter-connection of nodes which are simultaneously active. Specifically, we have the following:

1. The status of an output connector is IDLE for any active configuration.
2. The only input connector which can be in IDLE status when a node is active is the feedback connector of a loop junction.

These two observations tell us that if two nodes are in an active configuration in the same state then either they do not have a common link or they have a common link which is attached to the feedback connector of a loop junction.

If the nodes f_1 and f_2 in A_0 do not have a common link it is clear that the state A_3 resulting from the simultaneous completion of their transitions is identical to the states $A_1 \times C_{f_2}$ and $A_2 \times C_{f_1}$.

Thus all it remains to show is that when f_1 or f_2 or both are loop junctions sharing a link, the order in which these nodes complete their transitions does not matter. Figure 2.9a and 2.9b show the two possible connections that may exist between f_1 and f_2 up to symmetry. In both of these cases an output of f_1 is attached to the feedback connector of f_2 and since f_1 is active this link must be in idle status. The only active

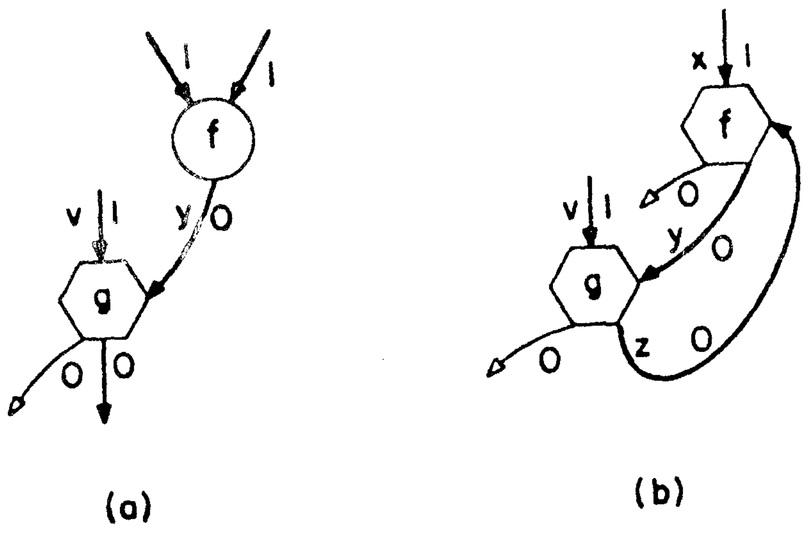


Fig. 2. 9 The Possible Configurations of Two Active Nodes Sharing a Link

configurations of loop junctions with a feedback link in idle status are, from Table 2.1, 1 0 0 0 and -1 0 0 0. By a straightforward use of Table 2.1 it can now be checked that the value and status of all links attached to f_1 and f_2 are identical independently of the sequence in which their transitions are completed.

Q. E. D.

Lemma 2.2 Let A_0 be a non-final state of a program graph P and let $A_i = A_0 \times C_{f_i}$, $A_i \neq A_0$ for $1 \leq i \leq k$. Then there exists a state A_{k+1} such that $A_i \mathcal{R} A_{k+1}$, $0 \leq i \leq k$.

Proof: Since $A_i \neq A_0$, A_0 has at least k-active nodes $f_1, f_2, \ldots f_k$. From the proof of Lemma 2.1, the completion of a transition of an active node cannot place any other active node in a non-active configuration. We claim that state resulting from the simultaneous transitions of f_1, f_2, $\ldots f_k$ satisfies the conditions of the state A_{k+1}.
From Lemma 2.1, there exists a state A_{nm} such that $A_0 \mathcal{R} A_{nm}$, $A_n \mathcal{R} A_{nm}$, and $A_m \mathcal{R} A_{nm}$ for all $1 \leq n$, $m \leq k$ and $n \neq m$. States A_{ni} and A_{im} are obtained from state A_i by the completion of precisely one transition, consequently by Lemma 2.1 we again conclude that there exists a state A_{nim} such that $A_0 \mathcal{R} A_{nim}$, $A_n \mathcal{R} A_{nim}$, $A_m \mathcal{R} A_{nim}$, $A_i \mathcal{R} A_{nim}$, $A_{n_i} \mathcal{R} A_{nim}$, and $A_{im} \mathcal{R} A_{nim}$ for all $1 \leq i \leq k$, $i \neq n$, $i \neq m$, and $n + m$. By repeating this process, we must eventually reach a state $A_{1,2\ldots k}$ satisfying the conditions of the theorem.

Q. E. D.

The construction of Lemma 2.2 is illustrated in Figure 2.10 for the case k = 3. In the process of proving this lemma we also proved the following.

Theorem 2.1 Let A_0 be a non-final state of a program graph P and let A_1, $A_2, \ldots A_k$ be states of P such that $A_0 \mathcal{R} A_i$, $1 \leq i \leq k$. Then there exists a state A_{k+1} such that $A_i \mathcal{R} A_{k+1}$ for $0 \leq i \leq k$.

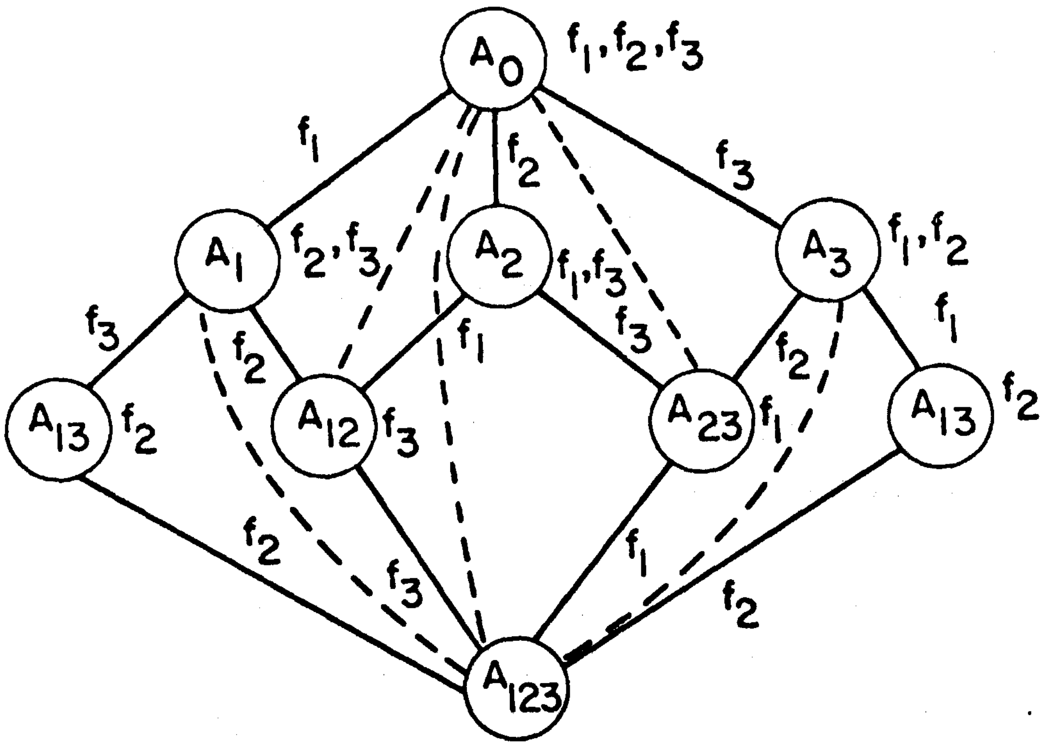


Fig. 2.10 The Relation Among the 'Next' States of a State A_0

Theorem 2.2 Let A_0 be the initial state of a program graph. If there exists an execution sequence $\sigma_1 = A_0 A_1 \ldots A_k$ such that A_k is a final state then A_k is unique.

Proof: To prove the theorem we will show that for any other execution sequence with the same initial state as σ_1, e.g. $\sigma_2 = A_0 B_1 \ldots B_m$, if $B_m \neq A_k$ then B_m is not a final state.

The proof consists of constructing two new execution sequences σ_1' and σ_2' such that $\ell(\sigma_2') < \ell(\sigma_2)$ and every state in σ_2' is also a state of σ_2. Since σ_2 is a finite sequence, successive applications of the construction yields sequences σ_1', σ_2', σ_1'', σ_2'', $\ldots \sigma_1^r$, σ_2^r with $\sigma_1^r = B_m A_1^r \ldots A_k$, $\sigma_2^r = B_m$, i.e. A_1^r is a next state of B_m.

To construct the execution sequences σ_1' and σ_2' we proceed as follows:

Let A_j be the last state in σ_1 which also appears in σ_2. There is at least one such common state, namely A_0.

$$A_j \mathcal{R} A_{j+1} \text{ in } \sigma_1$$

and

$$A_j \mathcal{R} B_{j+1} \text{ in } \sigma_2$$

From Theorem 2.1, there exists a state q such that $A_{j+1} \mathcal{R} q$ and $B_{n+1} \mathcal{R} q$. The state q can be either in sequence σ_2 or in sequence σ_1 or in neither sequence. Specifically, we must consider the following three cases, (Figure 2.11):

1. $q = B_n$ for some n $1 \leq n < m$
2. $q = A_n$ for some n $j + 1 \leq n < k$
3. $q \neq A_n$, $q \neq B_n$ for any n.

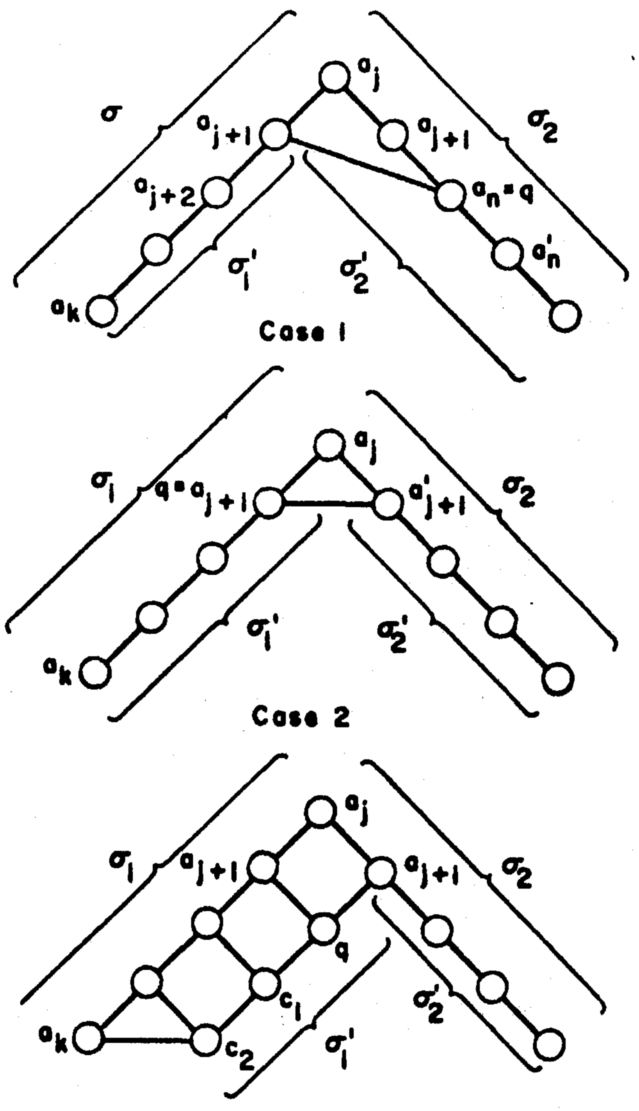


Fig. 2.11 The Three Cases in the Proof of Theorem 2.2

<u>Case 1</u>

Set $\sigma'_1 = A_{j+1}\ A_{j+2} \dots A_k \qquad \ell(\sigma'_1) < \ell(\sigma_1)$

$\sigma'_2 = A_{j+1}\ B_n\ B_{n+1} \dots B_m \qquad \ell(\sigma'_2) \leq \ell(\sigma_2)$

This case cannot occur indefinitely for if it did we could find two sequences σ_1^p, σ_2^p such that

$\sigma_1^p = A_k$

$\sigma_2^p = A_k \dots B_m$

which contradicts the assumption that A_k is a final state.

<u>Case 2</u>

Set $\sigma'_1 = B_{j+1}\ A_n \dots A_k \qquad \ell(\sigma'_1) \leq \ell(\sigma_1)$

$\sigma'_2 = B_{j+1}\ B_{i+2} \dots B_m \qquad \ell(\sigma'_2) < \ell(\sigma_2)$

<u>Case 3</u>

By repeated application of Theorem 2.1 there must exist states $C_1, C_2, \dots$ such that

$q\ \mathcal{R}\ C_1\ \mathcal{R}\ C_2 \dots \mathcal{R}\ A_k$

where C_1 is a state such that $A_{j+2}\ \mathcal{R}\ C_1$, $q\ \mathcal{R}\ C_1$

C_2 is a state such that $A_{j+3}\ \mathcal{R}\ C_2$, $C_1\ \mathcal{R}\ C_2$ etc. This must stop at A_k because it is terminal.

Thus we set

$\sigma'_1 = B_{j+1}\ q\ C_1\ C_2 \dots A_k$

$\sigma'_2 = B_{j+1}\ B_{i+2} \dots B_m \qquad \ell(\sigma'_2) < \ell(\sigma_2)$

Since Case 1 must eventually produce sequences σ'_1, σ'_2 such that either Case 2 or Case 3 applies, and in both cases $\ell(\sigma'_2) < \ell(\sigma_2)$ we conclude that eventually sequence σ_2^r will be B_m alone.

Q. E. D.

Theorem 2.3 Let A_0 be the initial state of a program graph. If there exists an execution sequence $\sigma_1 = A_0\, A_1\, A_2 \ldots A_k$ such that A_k is a final state, then every sequence with initial state A_0 is terminal on A_k.

Proof: All we need to show is that every execution sequence is terminal and by Theorem 2.2 it will follow that the final state is A_k.

Assume that there exists a non-terminal execution sequence

$$\sigma_2 = A_0\, A_1'\, A_2' \ldots A_n' \ldots .$$

Then we construct two new execution sequences σ_1' and σ_2' such that $\ell(\sigma_1') < \ell(\sigma_1)$ and σ_1' and σ_2' have the same initial state. Successive applications of the construction yields sequences $\sigma_1^r = A_k$ and $\sigma_2^r = A_k \ldots A_\ell' \ldots$, contradicting the assumption that A_k is a final state.

To construct the execution sequences σ_1' and σ_2' we proceed in a similar way as in the proof of Theorem 2.2.

Let A_j be the last state common to sequences σ_1 and σ_2.

$$A_j\, \mathcal{R} A_{j+1} \text{ in } \sigma_1$$
$$A_j\, \mathcal{R} A_{j+1}' \text{ in } \sigma_2$$

By Theorem 2.1 there exists state q such that $A_{j+1} \mathcal{R} q$ and $A_{j+1}' \mathcal{R} q$ consider the three cases (Figure 2.12):

1. $q = A_n'$ for some n, $j+1 \leq n$
2. $q = A_n$ for some n, $j+1 \leq n < k$
3. $q \neq A_n$ $q \neq A_n'$ for any n

Case 1

Set $\sigma_1' = A_{j+1}\, A_{j+2} \ldots A_k$ $\quad \ell(\sigma_1') < \ell(\sigma_1)$

$\sigma_2' = A_{j+1}\, A_n'\, A_{n+1}' \ldots$

$A_{j+1} \neq A_n'$ since A_j is the last common state in both sequences.

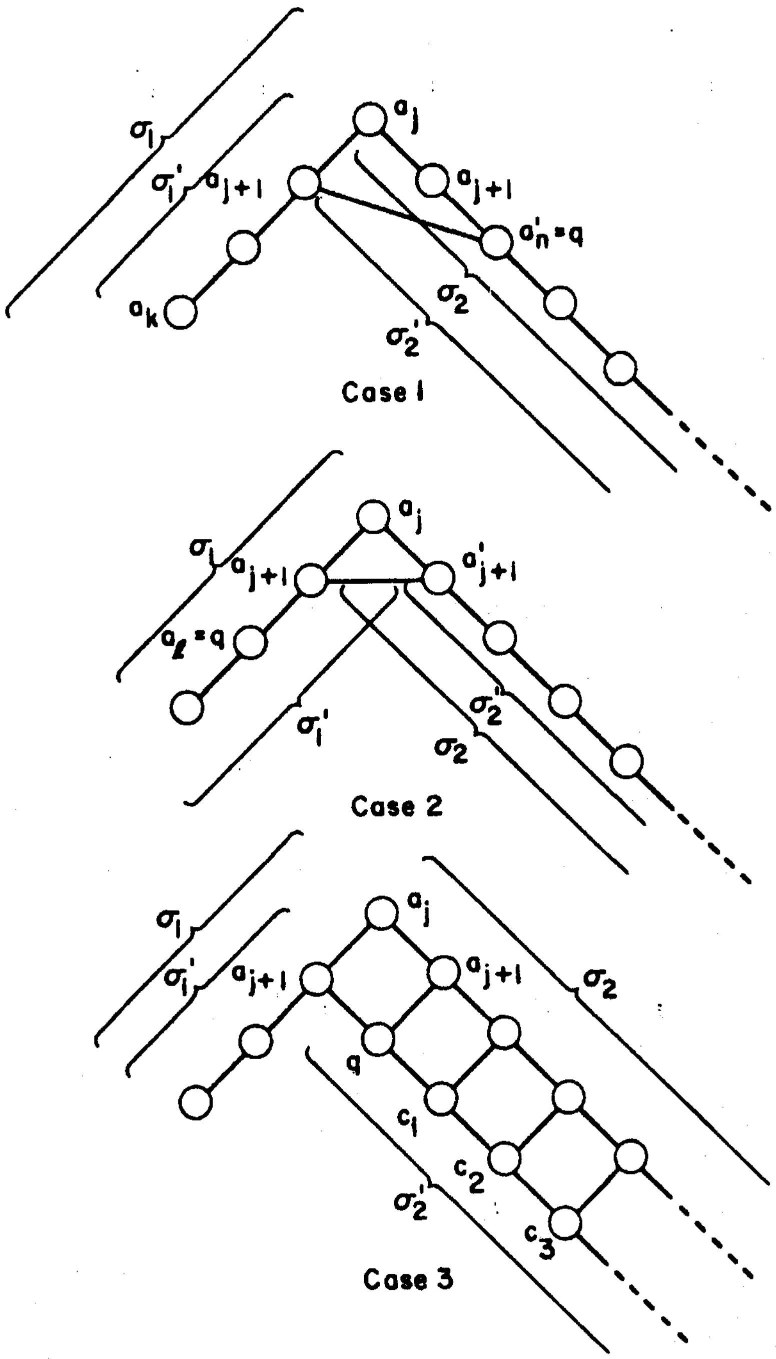


Fig. 2.12 The Three Cases in the Proof of Theorem 2.3

Case 2

Set $\sigma_1' = A_{j+1}' A_\ell \ldots A_k$ $\qquad \ell(\sigma_1') \leq \ell(\sigma_2)$

$\sigma_2' = A_{j+1}' A_{j+2}' \ldots$

$A_{j+1}' \neq A_\ell$ since A_j is the last common state in both sequences.

Case 3

By Theorem 2.1, there exists states $C_1, C_2, \ldots$ such that $q \mathcal{R} C_1 \mathcal{R} C_2 \ldots$. Therefore, set:

$\sigma_1' = A_{j+1} A_{j+2} \ldots A_k$ $\qquad \ell(\sigma_1') < \ell(\sigma_1)$

$\sigma_2' = A_{j+1}\, q\, C_1\, C_2 \ldots$

To complete the proof, we show that the condition $\ell(\sigma_1') = \ell(\sigma_1)$ in Case 2 cannot occur indefinitely. From the construction of the sequences σ_1' and σ_2'. if $\ell(\sigma_1') = \ell(\sigma_1)$ indefinitely, then there exists a state A_{j+1} in σ_1, and states $A_{j+1}', A_{j+2}', \ldots$ in σ_2 such that $A_j \mathcal{R} A_{j+1}$, $A_j \mathcal{R} A_{j+1}'$, $A_{j+1}' \mathcal{R} A_{j+1}$, $A_{j+2}' \mathcal{R} A_{j+1}, \ldots$ However, this implies that there are an infinite number of active nodes in state A_j which is impossible. Therefore, either there is a state A_{j+r}' in σ_2 such that $A_{j+r}' \mathcal{R} A_{j+s}$, $s > 1$, or Cases 1 or 3 apply. Whichever alternative occurs, $\ell(\sigma_1') < \ell(\sigma_1)$. Therefore, σ_2 must be terminal in A_k.

Q. E. D.

III. ANALYSIS OF PROGRAM GRAPHS

A. INTRODUCTION

The results of Chapter II tell us that every program graph represents a deterministic process. It is possible, however, to construct program graphs such that for some or all sets of input values no complete set of output values is ever produced, even when all execution sequences are terminal. If we view a useful computational process as a transformation of a set of input values into a set of output values, then not every program graph represents a useful computation.

The failure of a program graph to produce output values may be caused either by a never-ending cycle or by entering a final state prematurely. It is this second condition, which we call a hang-up state, that we are interested in because it is peculiar to program graphs, and because a study of the structure of graphs where it occurs provides insight into the properties of the model.

The occurrence of a hang-up state during an execution sequence is due to structural anomalies of the graph. These anomalies can arise from obvious misuse of a node for a purpose for which it was not intended, e. g. connecting a loop output control connector to a node other than a loop junction. Less obvious and more interesting structural anomalies arise in connection with communicating cycles. Section B introduces terminology and notation. In Section C we present several examples illustrating the proper use and the misuse of the various node types. Section D considers hang-up states in cycle free graphs. Finally Section E studies hang-up states in cyclic graphs.

B. NOTATION AND TERMINOLOGY

1. Notation

In this and subsequent chapters we will use the following notation when referring to nodes of program graphs:

Input terminals will be denoted by the letter $\underline{\alpha}$.

Output terminals will be denoted by the letter $\underline{\omega}$.

Data operators will be denoted by the letter $\underline{f}$.

Selectors will be denoted by the letter $\underline{\beta}$. The output connectors of a selector will be distinguished by writing β^+ or β^-.

Junctions will be denoted by the letter $\underline{j}$.

Loop junctions will be denoted by the letter $\underline{g}$.

Loop outputs will be denoted by the letter $\underline{h}$.

Links will be denoted by the letter $\underline{\ell}$.

Each of the above symbols will be used with a number subscripts when it is necessary to distinguish among two or more instances of a type, e.g. f_1, f_2, β_1, β_2. The letter $\underline{a}$ will be used to denote a node without specification of its type.

2. Paths, Cycles, and Connectivity

The following concepts and terms associated with directed graphs have been adapted from Busacker and Saaty.[1]

A finite sequence of links $\ell_1, \ell_2, \ldots \ell_k$ is said to constitute a path of length k in a program graph P if there are nodes $a_1, a_2, \ldots a_{k+1}$ in P such that ℓ_i is a link from an output connector of a_i to an input connector of a_{i+1}. The path is said to pass through the nodes $a_1, a_2, \ldots a_{k+1}$. The nodes a_1 and a_{k+1} are said to be the initial and final nodes of the path respectively, and it is said that there is a path from a_1 to a_{k+1}. If $a_1 = a_{k+1}$ the path is said to be a cycle. If all the k+1 nodes are distinct, the path is called a proper path. If $a_1 = a_{k+1}$, but otherwise all nodes are distinct, the path is said to be a proper cycle. Clearly, all links of a proper path or cycle are distinct. If all the k links are data links the path is called a data path. Similarly, a control path consists solely of control links. A program graph is said to be cyclic if it contains at least one cycle and cycle free otherwise.

Given two nodes a_i and a_j of a cycle free program graph P, it is said that a_i is an ancestor of a_j, or alternatively that a_j is a descendant of a_i, if there exists a proper path from a_i to a_j. If a_i is an ancestor (descendant) of a_j and there is a path of length 1 from $a_i(a_j)$ to $a_j(a_i)$ it is said that a_i is a direct ancestor (descendant) of a_j.

A subgraph P' of a graph P is a subset of the set of nodes of P together with all the links connected to these nodes. If for every pair of distinct nodes a_i and a_j of P' there is a path from a_i to a_j as well as one from a_j to a_i, it is said that the subgraph is strongly connected. If, in addition, this condition is not satisfied for any pair of nodes a_i and a_k when a_i but not a_k is contained in P', then it is said that P' is a maximal strongly connected subgraph (abbreviated mscs).

Two subgraphs P' and P'' of a program graph P are said to be disjoint if they do not share a common node.

3. Normal Sequences and Hang-up States

The last two definitions in this section are concerned with certain properties of the initial and final state of execution sequences of a program graph. In order to study the behavior of program graphs it is convenient to concentrate our attention to a limited, yet useful, class of execution sequences by normalizing the initial state of these sequences as follows:

Definition 3.1 A normal execution sequence of a program graph is an execution sequence with an initial state in which all links of the graph are in idle status except for those links attached to the input terminals. Henceforth, execution sequence will be used interchangeably with normal execution sequence. Furthermore, unless otherwise specified we will assume that the links attached to the input terminals are in enabled status in the initial state.

The final state of a terminal execution sequence will be called a hang-up state or a normal state according to whether or not it satisfies the following:

Definition 3.2 A state of a program graph is a hang-up state if

1. No node is in an active configuration.
2. At least one link is not in idle status.

C. EXAMPLES OF PROGRAM GRAPHS

1. Properties of Transition Tables

In this section we will consider in detail several examples of program graphs. We do this with a dual purpose. First, we want to acquaint the reader as much as possible with the manner of execution of a program graph. Second, we want to provide a better understanding of the properties of each type of node and how they should and should not be used. Concurrently we will indicate some of the reasons for the choices made in the specification of the transition tables.

During execution, the state of a graph changes as a result of the transitions specified by the active configurations of each type of node. The active configurations of a node are determined only by the status of the input and output connectors of the node. Upon completion of any transition these statuses are always changed. For some transitions the value of the output connectors may also be changed.

An examination of Table 2.1 reveals the following facts:

1. In all active configurations of every node type except loop junctions the status of every input connector is never IDLE(0).
2. In all active configurations the status of every output connector is always IDLE.
3. Upon completion of each transition, except for loop outputs, the status of every output connector is always non-IDLE.
4. Upon completion of each transition, except for loop junctions, the status of every input connector is always IDLE.
5. Selectors are the only nodes which place one of their output connectors in disabled status when none of their inputs are DISABLED.
6. The function and predicate associated with operators and selectors respectively are applied to the input values only if no input is DISABLED.
7. The application of the function associated with a data operator to obtain a new value for an output link always results in placing the link in ENABLED status.

Items 1 and 3 suggest viewing the execution of a graph as effecting the flow of status and data information from one node to another in the direction indicated by the oriented links. The IDLE status signifies either that no activity has taken place on a link or that previous activity on a link has been properly accounted for. As a rule, activity must occur at each input to a node before the node can perform any action. The only exception to this rule is the loop junction which under some circumstances, only requires one of its input connectors to be in non-IDLE status to become active. Effectively this means that each node waits to receive information from each of its ancestors before it takes any action.

Items 2, 3, and 4 point out that new activation of a node cannot occur until the information transmitted to the immediate descendants of the node has been used by these descendants. This observation together with items 6 and 7 imply that, as a rule, both the status and data of a link attached to an input connector of a node are effective for precisely one activation of the node. The only exception to this rule can be found in loop junctions and loop outputs both of which have provisions for 'remembering' status and data information. This property of loop junctions and outputs appears to be needed in order to obtain a deterministic model in the presence of cycles.

Finally, items 5 and 6 point out the unique function of selectors as the arbiters which determine the functions and predicates that should be applied during the course of an execution sequence. Note that application of a function or predicate requires that the associated node be active, however the converse is not true. In fact, most of the active configurations shown in Table 2.1 do not require application of a function, predicate, or effecting a data transmission operation. Instead, their only purpose is to propagate through the graph the necessary status information. The enable and disable statuses get their names from the effect that a link exhibiting these statuses have on the data transformation and transmission action of the nodes to which the link is attached.

2. The Use of Selectors and Junctions

Selectors and junctions have complementary functions. If we view the action of a selector as choosing between two alternative sequences of data transformations, then the purpose of a junction is to transmit the result of whichever sequence was chosen to succeeding computations. Figure 3.1a illustrates the simplest form that this relationship between

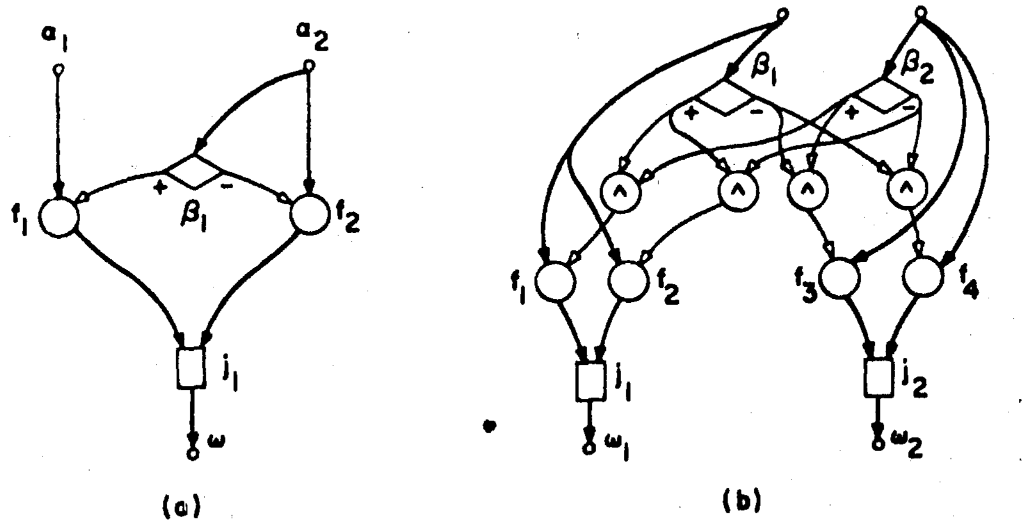


Fig. 3.1 Use of Selectors and Junctions

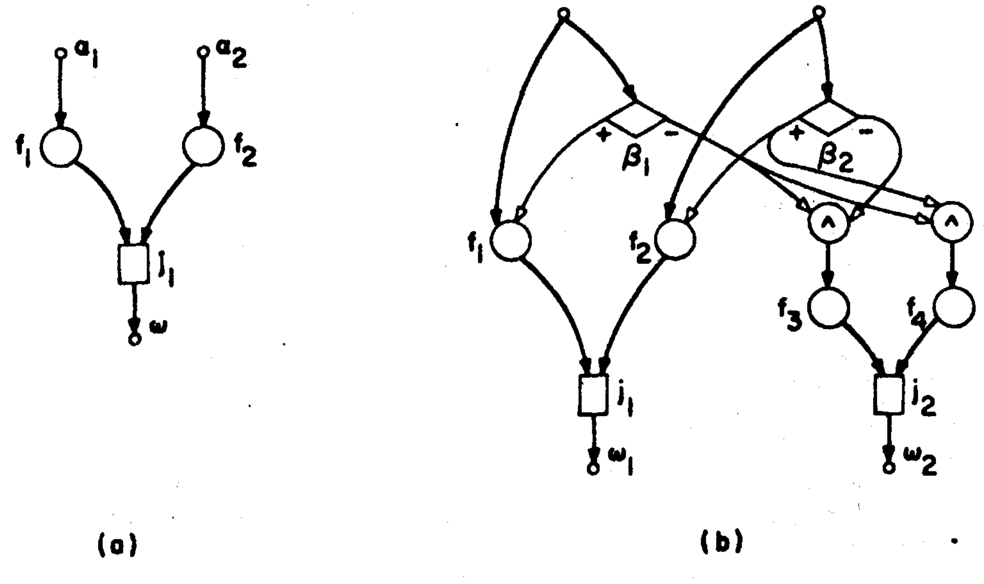


Fig. 3.2 Junctions Creating Hang-Up States

selectors and junctions can take. Clearly one selector can be used to choose one alternative from any number of pairs of sequences. Furthermore, several selectors may have to be invoked, either sequentially or in parallel, when more than two alternatives exist or when complex decision rules are needed. Figure 3. 1b illustrates some of these points.

Figure 3. 2a illustrates a typical misuse of a junction. In this graph, data junction j_1 will never become active, since Table 2. 1 requires that one or the other (or both) of its inputs be in the disabled status for any active configuration. Note that if the junction transition table did not have this characteristic, then the output of junction j_1 would depend on the relative speed of operators f_1 and f_2. Figure 3. 2b shows another example of this situation. In this case, junction j_1 is placed in a hang-up configuration if during an execution sequence the parallel selectors β_1 and β_2 enable their respective '+' connectors. Compare this with the arrangement of selectors β_1 and β_2 in Figure 3. 1b where no hang-up configuration arises.

The need for this type of behavior in junctions imply that in order to guarantee determinism we have to:

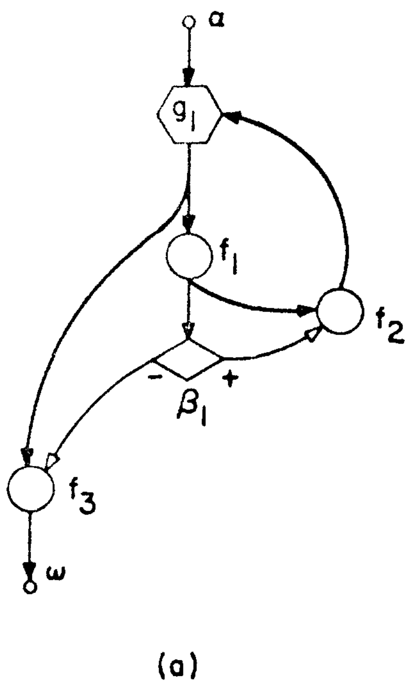
1. have three distinct link status values.
2. propagate a disable status throughout the graph.

3. The Use of Loop Junctions and Loop Outputs

Now let us turn our attention to the use of loop junctions and loop outputs. Loop junctions allow us to construct cycles in a program graph without necessarily introducing hang-up states. To see the necessity of loop junctions, consider the graph in Figure 3.7. If all links are initially in idle status, operators f_2 and f_3 will never become active because the idle output status of each prevents activation of the other. Clearly what is needed is a type of node which does not require each of its inputs to be in non-idle status before it can become active. We have already seen in the case of junctions that if the inputs of such a node were symmetric, non-deterministic behavior would arise even in the absence of cycles. It is not hard to convince ourselves that in order to have deterministic behavior given an n input node with an asymmetric transition table, the inputs must be arranged in a priority scheme. This priority must effectively dictate that an input cannot place the node in an active configuration unless the node had been previously activated by an input with a higher priority. For, unless this condition is satisfied the sequences of values at the outputs of the node would depend on the sequence of arrival of values at each of the inputs.

In order to implement this priority in a transition table it is necessary to introduce a fourth link status whose function is to remember the history of the activations of the node until such a time as this history becomes irrelevant. The link status blocked(2) serves this purpose in a program graph. Examination of the transition table for loop junctions reveals that the high priority input is the one labelled 1 which we call the initial input. The low priority input, labelled 2 will be called the feedback input. The behavior of a loop junction can be described as follows: The node becomes active as soon as the initial input becomes enabled or disabled independently of what the status of the feedback input is. The next active configuration of the node must be one in which the feedback input is enabled or disabled while the initial input is blocked.

Now we have to make a choice as to when to forget the history of activations of a loop junction. Since we want to allow an arbitrary number of repetitions of a cycle, the cue for this transition must come from the feedback input to the loop junction. The only two information



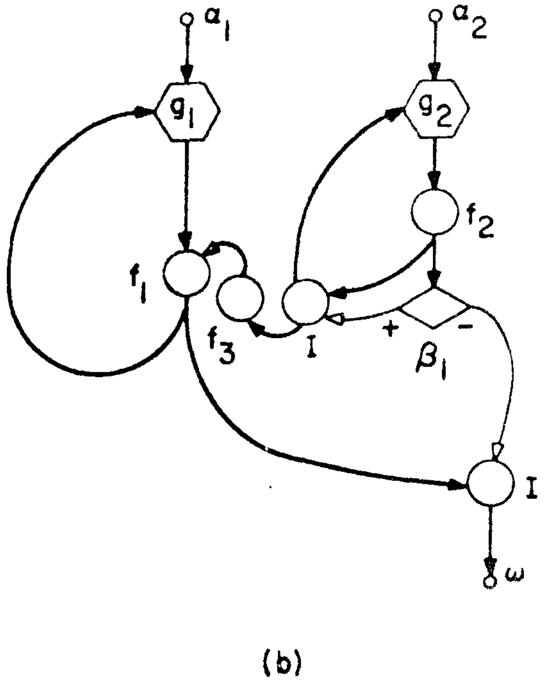
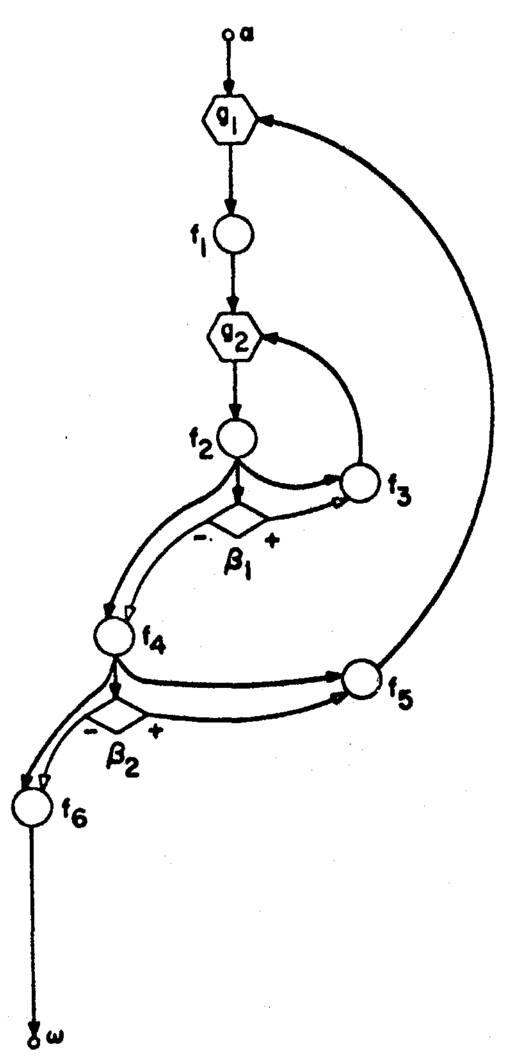


Fig. 3.3 Single and Parallel Cycles

statuses possible on a link attached to the feedback input are enabled and disabled. The obvious choice is the disabled status.

Figure 3.3a shows a simple example of a cyclic graph. Loop junction g_1 forms a cycle together with operators f_1 and f_2, and selector β_1. The cycle is initiated by enabling the input terminal α. Each time β_1^+ is enabled, f_2, g_2, f_1, and β_1 are reactivated. Note that since enabling β_1^+ implies disabling β_1^-, operator f_3, which is not a part of the cycle, will be activated on every iteration as well. This secondary effect may or may not be desirable. Figure 3.3b illustrates a cyclic graph with two parallel cycles. The cycle on the left, nodes g_1 and f_1, receives inputs from the cycle on the right. Both cycles repeat as long as β_1^+ is enabled. When β_1^+ is disabled, both cycles terminate. Operator f_3 is in neither cycle, yet it becomes active during each iteration, in this case performing a useful function. In the example of Figure 3.3a, the repeated activation of f_3 causes a sequence of disable statuses to appear at the output terminal $\underline{\omega}$. Upon termination of the cycle, the output terminal is enabled. In this instance, it is desirable to prevent the activation of f_3 while the cycle is in progress. In fact, if it is not possible to exert this type of control, we cannot construct graphs with nested cycles which are free of hang-up states. This difficulty is illustrated in Figure 3.4a where after four repetitions of the 'inner' cycle, formed by nodes g_2, f_2, β_1, and f_3, every node is unable to enter an active configuration because the output of $\beta_1^{(+)}$, f_4, f_5 are in non-idle status and the initial input of g_1 is in idle status.

Loop output nodes have been introduced to avoid such situations. Figure 3.4b shows the proper use of loop outputs to avoid hang-up states in the example of Figure 3.4a. A loop output should be connected only to a loop junction. The proper form of this connection is illustrated in Figure 3.5.



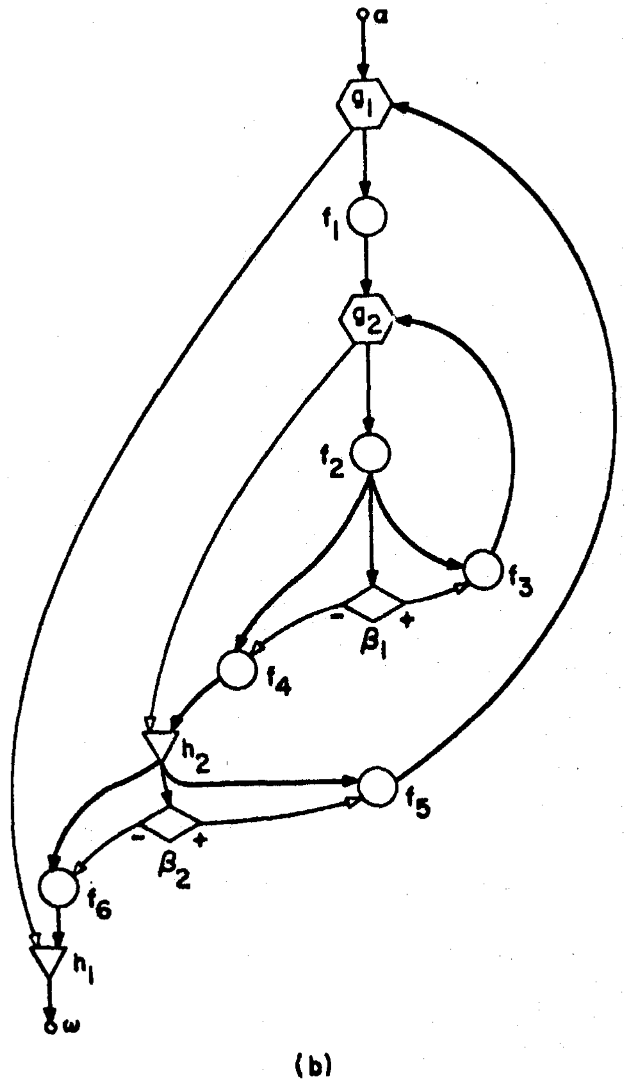


Fig. 3.4 Nested Cycles with and without Loop Output Nodes

By examining Table 2.1 we can verify that in this arrangement the output link of a loop output becomes non-idle only when the feedback input of the loop junction becomes disabled (provided the initial input had been enabled or disabled), i.e. when the cycle formed by the loop junction terminates.

D. ANALYSIS OF CYCLE FREE GRAPHS

1. The Role of Cycle Free Graphs

In this section we study certain properties of cycle free program graphs. The simplicity of this class of graphs relative to cyclic graphs makes them a natural starting point in the analysis of program graphs. Furthermore, some of the questions about cyclic graphs raised in the next section can be satisfactorily resolved by reducing them to similar questions about cycle free graphs.

2. Properties of Execution Sequences

Intuitively we expect each execution sequence of a cycle free graph to be terminal. In fact, at this point such a statement should not take us by surprise. However, the method we have chosen to specify the behavior of each node, i.e. the transition table, does not make this property obvious or even necessary. The following theorem and its corollaries are a justification for the choice of directed graphs for our representation.

Theorem 3.1 Every execution sequence of a cycle free program graph is a subsequence of a terminal sequence.

Proof: Assume that the theorem is false. Then we can find an execution sequence which never terminates. Since the number of active configurations in the transition table of each node is finite, it follows that there is at least one node which is placed in the same active configuration an infinite number of times. An examination of Table 2.1 indicates that after completion of the transition of most active configurations of a node, a new active configuration can occur only if the status of the input connectors of the node are changed by its direct ancestors. The only exceptions to this rule are the four active configurations of a loop junction: 1 1 0 0, 1 -1 0 0, -1 1 0 0, and -1 -1 0 0. For these configurations two transitions may occur while the direct ancestors of the loop junction are necessarily non-active. Thus, if a node is active an arbitrary number of times, the same is true of at least one of its direct ancestors. Since the graph is cycle free, by repeatedly applying this reasoning it must be that the initial terminal must be active an infinite number of times against the definition of an execution sequence. This shows that every node must be active a finite number of times. Therefore each execution sequence must eventually reach a final state and is a subsequence of a terminal sequence.

Q. E. D.

Corollary 3.1 A program graph with an infinite execution sequence must be cyclic.

Corollary 3.2 If a_i and a_j are two nodes of a cycle free program graph and there is a proper path from a_i to a_j which does not pass through any loop junction (except for a_i), then a_j can become active only after a_i has completed a transition.

Proof: In order for a_j to become active, all of its inputs must be non-idle. Since initially all links are in idle status, this change can take place only by completing a transition of all its immediate ancestors. By repeating this process, the immediate ancestors of the immediate ancestors, ... etc. must also complete a transition. But a_i is an ancestor of a_j. Thus eventually a_i must be encountered in the chain of direct ancestors which must complete transitions before a_j can become active.

Q. E. D.

Theorem 3.1 applies to an arbitrary cycle free graph. However, from a behavioral point of view, loop junctions and loop outputs do not perform any useful function in a cycle free graph while unnecessarily complicating the analysis. This motivates the following definition:

Definition 3.1 A simple cycle free program graph is a cycle free program graph which does not contain any loop junctions or loop outputs.

In Section C it is mentioned that the effect of most active configurations is to propagate the necessary status information throughout the graph even if no functional application (and presumably useful computations) take place. The following theorem shows that this is a necessary condition for the absence of hang-up states.

Theorem 3.2 The final state of a terminal execution sequence of a simple cycle free graph is not a hang-up state iff every node of the graph enters an active configuration exactly once during the execution sequence.

Proof: Assume the final state is not a hang-up state. Then every link is in idle status in the final state. Since initially all links attached to input terminals are placed in non-idle status, it follows that every direct descendant of an input terminal must have been in an active configuration. But every transition of a node appearing in the graph, i.e. operator, selector, and junction, places each of its outputs in non-idle status. Therefore the direct descendants of these nodes must also have been in an active configuration. This shows that every node becomes active at least once. However, by Table 2.1, each active configuration requires all input connectors in non-idle status, whereupon the completion of each transition reverts the status back to idle. Since there are no cycles, the status must remain idle thereafter which shows that each node becomes active at most once.

Q.E.D.

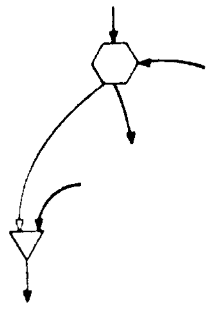


Fig. 3. 5 The Proper Way of Connecting Loop Junctions and Loop Outputs

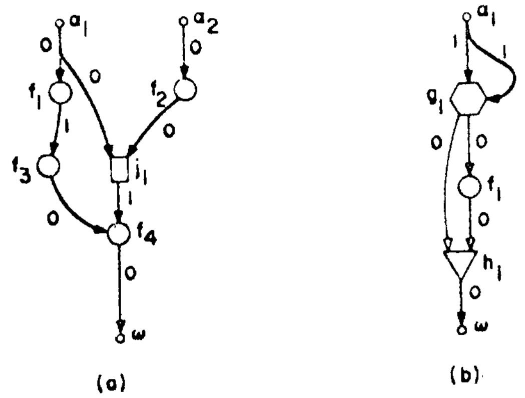


Fig. 3. 6 Initial States of Program Graphs that do not Satisfy Theorem 3. 2

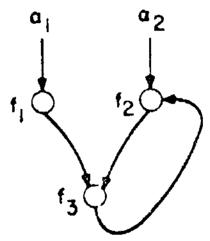


Fig. 3. 7 A Cycle that does not Pass Through a Loop Junction

Corollary 3.3 A node of a simple cycle free graph becomes active at most once during an execution sequence.

The simple characterization of hang-up states of Theorem 3.2 is not possible if we neither have a normal execution sequence nor are loop junctions forbidden from the cycle free graph. Figure 3.6a shows a program graph with an initial state such that the final state of every execution sequence is not a hang-up state, yet junction j_1 is never in an active configuration. In the program graph shown in Figure 3.6b, all nodes become active during each normal execution sequence, yet the final state is a hang-up state.

The following theorem shows that junctions constitute the source of all hang-up states in simple cycle free graphs.

Theorem 3.3 A node of a simple cycle free graph does not enter an active configuration during a terminal execution sequence iff the node is, or has as an ancestor, a junction with at least two input connectors in enabled status upon reaching the final state.

Proof: If a node is a junction and during the execution sequence two of its inputs become enabled, then by the transition table for junctions, it will never become active. By corollary 3.2, any descendant of this junction will never become active either.

Now assume that no two inputs of a junction are enabled during the execution sequence and that some other node does not become active. By the transition table of operators, selectors, and junctions this can only occur if a link remains in idle status throughout the execution sequence. This in turn implies that one or more of the input links of the node to which that link is attached must have remained in idle status also. By repeatedly applying this reasoning we must eventually reach a link attached to an input terminal and this link must have been idle. But this contradicts the definition of an execution sequence. Therefore every node must have been active during the execution sequence.

Q. E. D.

Theorem 3.4 A simple cycle free program graph has a hang-up state iff two or more input links of a junction are placed in enabled status.

Proof: Immediate from Theorem 3.2, Corollary, 3.3, and Theorem 3.3.

Q. E. D.

It is clear that the hang-up states of simple cycle free graphs arise only as a result of improper specification of the relationship between junctions and selectors.

3. The Enabling Function

The foregoing results suggest that it is desirable to have an easy way of ascertaining whether or not two links can be placed in enabled status during an execution sequence. To this end we associate with each link of a simple cycle free graph a boolean function, called the enabling function, with the property that the link cannot be enabled during an execution if its enabling function has the value false. The enabling function associated with a link ℓ_i will be denoted by E_i. If E_1 and E_2 are enabling functions then the union (or) is denoted by $E_1 \vee E_2$ and the intersection (and) is denoted by $E_1 \wedge E_2$. If B is a boolean variable, then $\overline{B}$ denotes its complement.

To obtain the enabling functions associated with the links of a graph we first assign a boolean variable B_i to each selector β_i[*]. The enabling function of a link ℓ is obtained by recursively applying the following rules:

[*] Throughout this discussion we assume that each selector in a graph has a label distinct from every other selector label.

1. If ℓ is the output link of an initial terminal, assign to it the identity element (true).
2. If ℓ is an output link of an n-ary operator, assign to it the function $E_1 \wedge E_2 \wedge \ldots \wedge E_n$, where E_i is the enabling function associated with the ith input link.
3. If ℓ is the output link of an n-ary junction, assign to it the function $E_1 \vee E_2 \vee \ldots \vee E_n$.
4. If ℓ is the '+' output link of n-ary selector β_i, assign to it the function $B_i \wedge E_1 \wedge E_2 \wedge \ldots \wedge E_n$.
5. If ℓ is the '-' output link of an n-ary selector β_i, assign to it the function $\bar{B}_i \wedge E_1 \wedge E_2 \wedge \ldots \wedge E_n$.

Clearly, the enabling function associated with a link involves only the variables B_i, their complement, and the identity element. In rules 4 and 5 we have effectively adopted the convention that a boolean variable B_i is true if the '+' output of the corresponding selector is enabled. We now define the relationship between the values of the B_i and an execution sequence. Let $\beta_1, \beta_2, \ldots \beta_n$ be the selectors of a graph P, and let $B = (b_1, b_2, \ldots b_n)$ be an n-tuple where b_i, $1 \leq i \leq n$ is either true or false. An execution sequence of P during which β_i^+ is enabled only if b_i = true is said to match the n-tuple B. Note that this definition does not require β_i to be active during the execution sequence. If ℓ_i is a link of P, $E_i(B)$ denotes the value of the enabling function of ℓ_i when b_i is substituted for B_i in E_i.

Theorem 3.5 Let P, β_i, B_i, B, ℓ_i, and E_i be as above. If $E_i(B)$ = false then ℓ_i cannot be enabled during any execution sequence of P which matches B.

Proof: In the transition tables for operators, selector, and junction in Table 2.1 note that the assignment enabled $\equiv$ true, disabled $\equiv$ false is consistent with rules 2, 3, 4, and 5. The transition tables are not complete in the sense that junctions do not have an active configuration 1 1 0. However every active configuration that can disable an output link of a junction is included in rule 2 and vice versa. Thus, the most that can be said is that a link cannot be enabled when its enabling function is false.

Q. E. D.

From Theorems 3.4 and 3.5 it follows that a simple cycle free graph does not have hang-up states if for each pair of links ℓ_i and ℓ_j where ℓ_i and ℓ_j are inputs of the same junction, $E_i \wedge E_j \equiv$ false.

E. ANALYSIS OF CYCLIC GRAPHS

1. Cycle Decomposition

In this section we undertake the study of cyclic graphs from the point of view of relating the cyclic structure of the graph to its behavior. The main results are that

1. for a large class of cyclic graphs necessary and sufficient conditions for the absence of hang-up states can be obtained based on properties of strongly-connected subgraphs, and
2. certain relations between cycles of a graph either introduce hang-up states or can be replaced by cycle free graphs.

The basis for studying relations among cycles of a graph P is a decomposition of the graph into certain strongly-connected subgraphs $K_1, K_2, \ldots K_n$. The decomposition is unique and has the property that for any two strongly-connected subgraphs K_i and K_j, either* $K_i \subset K_j$, $K_j \subset K_i$, or $K_i \cap K_j \neq \emptyset$. Thus the K_i are partially ordered under the relation of proper set inclusion.

The decomposition of a graph P into its strongly-connected subgraphs $K_1, K_2, \ldots K_n$ is accomplished by the iterative application of a procedure which at each step of the iteration breaks certain cycles of P. If P^r is the graph after the rth iteration ($P = P^o$), then each of the K_j is a maximal strongly-connected subgraph (mscs) of some P^i, $0 \leq i \leq r$. The choice of which cycles are broken at each step is based on the remarks of section C that in order for a cycle of a program graph to be effective it has to pass through a loop junction. This motivates the following definition:

---

* A subgraph K_1 is contained in a subgraph K_2 if every node of K_1 is also a node of K_2. We write $K_1 \subseteq K_2$ to denote inclusion and $K_1 \subset K_2$ to denote proper inclusion. The notation $K_1 \cap K_2$ denotes the subgraph consisting of the nodes contained in both K_1 and K_2. If K_1 and K_2 are disjoint we write $K_1 \cap K_2 = \emptyset$.

Definition 3.2 Let K_i be a strongly-connected subgraph of a program graph P. The G-set of K_i is the set of loop junctions of P such that $g \epsilon K_i$ and the direct ancestor of the input connector of g is not in K_i. The G-set of K_i will be denoted by G_i.

In Figure 3.8, the G-set of mscs K_1 consists of g_1 and g_2. Now we show that certain hang-up states of a graph can be directly related to the G-sets of its strongly-connected subgraphs. First we need the following two lemmas:

Lemma 3.1 Let a_i and a_j be two nodes of a program graph P. If there is a path from a_i to a_j and a_j becomes active during an execution sequence of P, then a_i must have been active during that sequence.

Proof: The proof is entirely analogous to Corollary 3.2.

Lemma 3.2 If during a terminal execution sequence of a cyclic graph some node a does not enter an active configuration, then the final state is a hang-up state.

Proof: Assume the lemma is false. Then every link is in idle status in the final state. Using the transition tables and modifying the argument of Theorem 3.2 to take loop outputs into consideration it follows that every node must have been active at least once. Loop outputs deserve special attention because for some of their active configurations the status of the output link remains idle after the transition. However, these transitions leave an input link enabled or disabled; the only way that this link can become idle is if the active configuration enabling or disabling the output link occurs. Thus, if the final state is not a hang-up state all nodes have been active which contradicts the assumption.

Q. E. D.

Theorem 3.6 Let K be a strongly-connected subgraph of a program graph P. If the G-set of K is empty then P has a hang-up state.

Proof: Since input terminals do not have any input connectors, no cycle can pass through them. For any node of P, there is a path from at least one input terminal to the node. Therefore, K has a node with a direct ancestor not contained in K. Let a be such a node. Since K is strongly-connected, there is a cycle $\ell_1, \ldots, \ell_k$ such that ℓ_1 and ℓ_k are respectively an output link and an input link of a. If a is a loop junction, ℓ_k must be attached to its initial input for, otherwise the G-set of K would not be empty. Thus, for any a, ℓ_k must become enabled before ℓ_1 does and by lemma 3.1, a can never become active. By lemma 3.2, it follows that P has a hang-up state.

Q. E. D.

If $K_1, K_2, \ldots K_n$ are the maximal strongly-connected subgraphs of P with non-empty G-sets $G_1, G_2, \ldots G_n$, let P^1 be the graph obtained from P by disconnecting all links attached to the feedback connectors of each loop junction $g \in G_i$, $1 \leq i \leq n$. We say that P^1 is derived from P.

Theorem 3.7 Let P be a cyclic graph and P^1 its derived graph. Then

1. P^1 is unique.
2. P^1 is cycle free or every mscs of P^1 is properly contained within precisely one mscs of P.

Proof: That P^1 is unique follows directly from the fact that a directed graph can be uniquely decomposed into its maximal strongly-connected subgraphs.

To show part 2, we note that P^1 must necessarily have fewer cycles than P since every proper cycle of P which passes through $g \epsilon G_i$, $1 \leq i \leq n$ does not exist in P^1. Thus, P^1 may be cycle free. Alternatively, if P^1 is not cycle free, every proper cycle of P^1 is also a cycle of P and therefore every mscs of P^1 is contained in one mscs of P. The inclusion is proper because at least every $g \epsilon G_i$, $1 \leq i \leq n$ is not contained in any mscs of P^1.

Q. E. D.

Given a cyclic graph P, we can obtain a sequence of graphs $P = P^0 P^1 \ldots P^n$ such that P^{i+1} is derived from P^i. Since there are a finite number of proper cycles in P and each step of derivation destroys one or more cycles, P^n is either cycle free or it has an mscs with an empty G-set. If P^n is cycle free it is said that P is cyclic consistent. The process of obtaining the sequence $P^0P^1 \ldots P^n$ will be called cycle decomposition. If P is cyclic consistent, P^n is the cycle free graph generated by P. The mscs's of the graphs obtained during the cycle decomposition of P can be arranged in a tree structure. The root of the tree is labelled P and every other vertex of the tree is labelled with the name of one mscs in such a way that there is a branch joining vertex K_i to vertex K_j iff $K_i \subset K_j$. The tree obtained in this manner will be called the cycle structure of P. A node a is said to belong to an mscs K of the cycle structure iff K is the smallest mscs containing a. Figure 3.8 illustrates the application of the cycle decomposition procedure. The sequence $P = P^0P^1P^2$ appears in Figures 3.8a, b, and c respectively. The cycle structure is shown in Figure 3.8c. Node f_2 belongs to K_3, while node f_6 belongs to K_1.

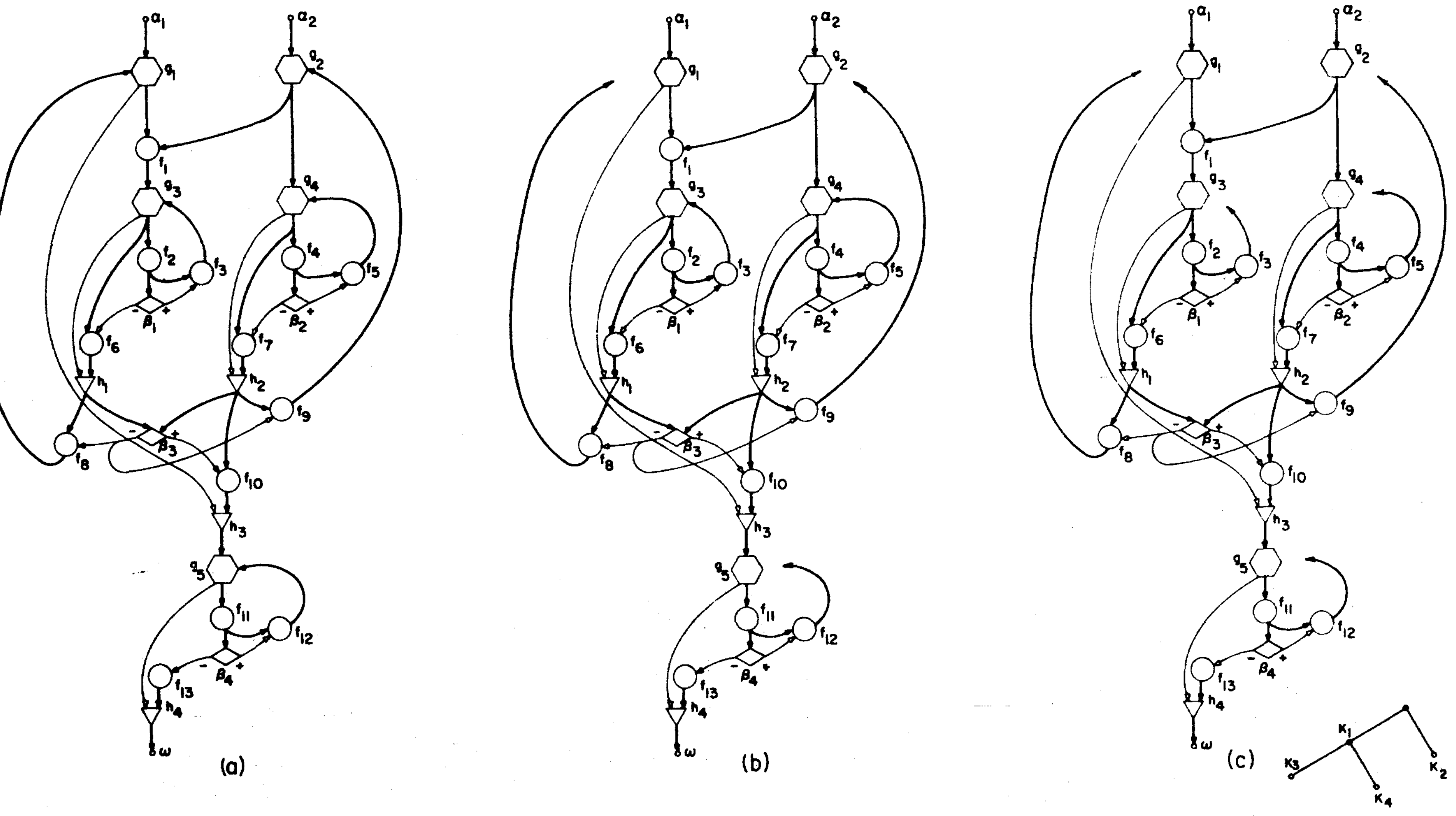


Fig. 3.8 Example of the Cycle Decomposition Procedure

2. Simple Cyclic Graphs

The concept of cyclic consistent graphs suggests that we restrict our attention to the study of cyclic graphs which meet certain minimal structural conditions in the formation of their cycles. In Section D we defined simple cycle free graphs by disallowing the use of loop junctions and loop outputs. Here we define simple graphs (scg) by only allowing 'proper' use of loop junctions and loop outputs in cyclic graphs. Simple graphs include simple cycle free graphs as a special case.

Definition 3.3 A graph P is simple iff

1. P is cyclic consistent.
2. Every loop junction of P is in the G-set of some mscs of the cycle structure of P.
3. Every loop output of P is properly connected to a loop junction.

From Theorem 3.2, we know that if all the paths from each ancestor of a node to the node pass only through operators, selectors, and junctions, then, in the absence of hang-up states the node becomes active iff its ancestors do. This suggests the following definition associating the links of the graphs with the mscs of the cycle structure:

Definition 3.4 A link ℓ belongs to an mscs K if there is a proper path $\ell_1 \ell_2 \ldots \ell_k \ell$ passing through nodes $a_1, a_2 \ldots a_k a_{k+1}$ such that

1. a_i, $1 < i \leq k+1$ is neither a loop junction or a loop output.
2. a_1 is a loop junction contained in the G-set of K, or a_1 is a loop output having as direct ancestor a loop junction whose initial input belongs to K.

A link may belong to zero, one, or more mscs's. For our purposes, we will consider that all links which do not belong to any mscs do in fact belong to a 'virtual' mscs.

The hang-up states of simple graphs can be characterized in terms of a counting property of links belonging to the same mscs and links attached to loop junction - loop output pairs. Figure 3.9 defines the link names used in Theorem 3.6. If ℓ is a link, $[\ell]$ denotes the <u>number of times</u> that ℓ is enabled or disabled during an execution sequence.

<u>Theorem 3.8</u> The final state of a terminal execution sequence of a simple program graph is not a hang-up state iff the following two conditions are satisfied:

1. For each loop junction - loop output pair $[\ell_2] = [\ell_3] = [\ell_4] = [\ell_5] \geq [\ell_1]$.

2. If ℓ_i and ℓ_j belong to the same mscs then $[\ell_i] = [\ell_j]$.

<u>Proof:</u> If P is cycle free, condition 1 does not apply and condition 2 becomes the statement of Theorem 3.2. Thus, it suffices to assume that P is cyclic. The proof invokes certain properties of the transition tables obtained by exhaustive case analysis. Because performing the case analysis each time becomes extremely tedious, we will just list these properties here and defer their detailed verification to Appendix B. The properties are grouped according to the node types to which they apply.

<u>Loop junctions - loop output pair</u>

a. $[\ell_1] = [\ell_6]$ iff ℓ_1 and ℓ_3 are idle

<u>Loop junctions</u>

b. If ℓ_1 is idle then:

$[\ell_2] = [\ell_4]$ iff $[\ell_2] = [\ell_3]$, ℓ_2 and ℓ_4 are idle.

c. If ℓ_1, ℓ_2, ℓ_3, ℓ_4 are idle then $[\ell_2] = [\ell_3] = [\ell_4]$

<u>Operators, selectors, junctions, and loop outputs</u>

d. Each input link has been enabled or disabled n times iff either all input links are idle or all are non-idle.

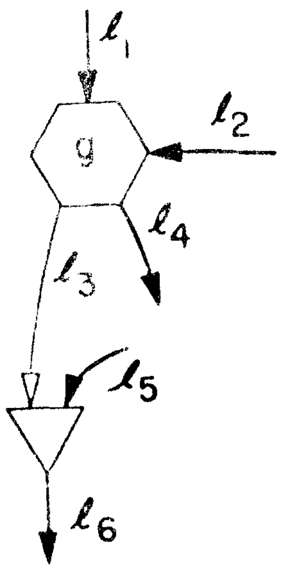


Fig. 3.9 The Link Names of Loop Junction-Loop Output Pairs

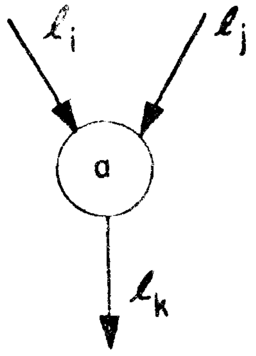


Fig. 3.10 The Link Names of Operators, Selectors, and Junctions

<u>Operators, selectors, and junctions</u>

e. If ℓ_i and ℓ_j are input and output links respectively then:

$[\ell_i] = [\ell_j]$ iff ℓ_i is idle.

To show that conditions 1 and 2 imply that all links are idle in the final state, we divide the nodes into two classes: 1) operators, selectors, and junctions, and 2) loop junctions and loop outputs. For each class we show that each of their input links has the desired property and thereby cover all the links of the graph, since links attached to output terminals are inconsequential.

In Figure 3.10, let the node represent an operator, selector, or junction. If the input links ℓ_i and ℓ_j belong to mscs's K_1 and K_2 respectively, then ℓ_j, the output link, belongs to both K_1 and K_2. By condition 2, $[\ell_i] = [\ell_k] = [\ell_j]$ and by property <u>e</u> both ℓ_i and ℓ_j must be idle. Clearly, the same argument can be applied for any number of inputs. Now consider Figure 3.9. Since the graph is simple, every loop output is paired with precisely one loop junction as illustrated in the Figure. By Definition 3, ℓ_1 and ℓ_6 belong to the same mscs, and by condition 2, $[\ell_1] = [\ell_6]$. Therefore, by property <u>a</u> both ℓ_1 and ℓ_3 are idle. By condition 1, $[\ell_2] = [\ell_4]$ and by property <u>b</u>, $[\ell_2] = [\ell_3]$ and ℓ_2 and ℓ_4 are idle. This shows that both inputs of the loop junction are idle. We also have that ℓ_3 is idle and by condition 1, $[\ell_3] = [\ell_5]$. Therefore, by property <u>d</u>, ℓ_5 must also be idle and this completes the first half of the proof.

Next, we show that conditions 1 and 2 are necessary as well. Thus, assume that all links are in idle status. In particular, for any loop junction $\ell_1, \ell_2, \ell_3, \ell_4$ are idle. Thus, by property <u>a</u>, $[\ell_1] = [\ell_6]$ and by property <u>c</u> $[\ell_2] = [\ell_3] = [\ell_4]$. Finally, by property <u>d</u> $[\ell_3] = [\ell_5]$ and thus follows condition 1. To show that condition 2 is also implied, we just reverse the argument using Figure 3.10 and condition <u>e</u>. This completes the proof.

Q. E. D.

Corollary 3.3 Let g_1 and g_2 be loop junctions contained in the G-set of an mscs of the cycle structure of a simple graph P. The final state of a terminal execution sequence is not a hang-up state, only if $[\ell_2(g_1)] = [\ell_4(g_1)] = [\ell_2(g_2)] = [\ell_4(g_2)]$* and $[\ell_1(g_1)] = [\ell_1(g_2)]$.

Proof: Immediate by observing that $\ell_2(g_1)$ and $\ell_2(g_2)$ belong to the same mscs.

Q. E. D.

Corollary 3.4 Let g_1 and g_2 be loop junctions contained in the G-sets of mscs's K_1 and K_2 of a simple graph P respectively. If there exists a link ℓ in P which belongs to both K_1 and K_2, then the final state of a terminal execution sequence is not a hang-up state only if $[\ell_2(g_1)] = [\ell_2(g_2)] = [\ell_3(g_1)] = [\ell_3(g_2)] = [\ell_4(g_1)] = [\ell_4(g_2)]$.

Proof: ℓ must be the output connector of a node as shown in Figure 3.10. By first applying condition 2 and then 1 we obtain the desired result.

Q. E. D.

If we view the links of the graph as providing channels for the flow of control information, Theorem 3.6 says that there must exist a certain balance of in-flow and out-flow at each node and at each cycle. The next theorem shows that in order to preserve this balance, and thereby avoid hang-up states, communication between cycles should be restricted when the cycles are not contained within mscs's of the cycle structure sharing common nodes (i. e. mscs's that are not disjoint). The essence of this constraint is shown in the proof of Lemma 3.5. We need two preliminary lemmas relating the initial and feedback inputs of loop junctions to the cycle structure of simple graphs.

* $\ell(g)$ denotes the named link of loop junction g with reference to Figure 3.9.

<u>Lemma 3.3</u> Let P be a simple graph, K_1 an mscs of its cycle structure and g_1 a member of the G-set of K_1. If ℓ_1 is the initial input of g_1, then ℓ_1 belongs to an mscs K_2, where $K_1 \subset K_2$ or $K_1 \cap K_2 = \emptyset$.

Proof: Assume the lemma is false, then $K_2 \subseteq K_1$. This implies that there is a path from g_1 to every $g_2 \in K_2$. Since ℓ_1 belongs to K_2, there is a path from some g_2 to g_1 and the last link in this path is ℓ_1. Therefore, there is a cycle in K_1 involving ℓ_1 and either g_1 is not in the G-set of K_1 or P is not simple. In either case we obtain a contradiction. Thus, either $K_1 \subset K_2$ or $K_1 \cap K_2 = \emptyset$.

Q. E. D.

<u>Lemma 3.4</u> Let P, K_1 and g_1 be as in Lemma 3.3. If ℓ_2 is the feedback link of g_1, then ℓ_2 belongs to K_2, where $K_2 \subseteq K_1$.

<u>Proof:</u> If $K_1 \subset K_2$ or $K_1 \cap K_2 = \emptyset$, both inputs of g_1 have direct ancestors not contained in K_1. Consequently $g_1 \epsilon K_1$ which contradicts the hypothesis. Thus, $K_2 \subseteq K_1$.

Q. E. D.

<u>Lemma 3.5</u> Let ℓ_i and ℓ_j be links belonging to mscs's K_i and K_j of a simple program graph P. Let g_i be a loop junction contained in the G-set of K_i. If $K_i \subset K_j$ and $[\ell_i] = [\ell_j]$, the final state of an execution sequence is not a hang-up state only if $[\ell_2(g_i)] = [\ell_1(g_i)]$.

<u>Proof:</u> Since $K_i \subset K_j$, there must exist a loop junction $g_r \in G_i$ and an mscs K'_s such that $\ell_1(g_r)$ belongs to K'_s and $K_s \subseteq K_j$. For, by lemma 3.3, if such g_r did not exist it follows that $K^1_s \cap K_i = \emptyset$ and therefore $K_i = K_j$. Now by Theorem 3.8 we have: (See Figure 3.11)

$$[\ell_2(g_r)] \geq [\ell_1(g_r)] \qquad (1)$$

$$[\ell_2(g_r)] = [\ell_4(g_r)] \qquad (2)$$

If g_s^1 is a loop junction in the G-set of K_s^1, since $\ell_4(g_s^1)$ belongs to the same mscs as $\ell_1(g_r)$ it follows that:

$$[\ell_1(g_r)] = [\ell_4(g_s^1)] \tag{3}$$

If $K_s \neq K_j$, we can apply the same process yielding at the mth step the relations (see Figure 3.11).

$$[\ell_2(g_s^{m-1})] \geq [\ell_1(g_s^{m-1})] \tag{4}$$

$$[\ell_2(g_s^{m-1})] = [\ell_4(g_s^{m-1})] \tag{5}$$

$$[\ell_1(g_s^{m-1})] = [\ell_4(g_s^{m})] \tag{6}$$

At the nth step K_j is reached, by Theorem 3.8

$$[\ell_i] = [\ell_j] = [\ell_4(g_s^n)] \tag{7}$$

Substituting back into each set of relations using (4), (6), and (7) we obtain:

$$[\ell_2(g_s^{m-1})] \geq [\ell_i] \tag{8}$$

Using (8), (5), and (3) we obtain:

$$[\ell_1(g_s^{m-2})] \geq [\ell_i] \tag{9}$$

This process is repeated as often as needed using the appropriate instances of (4), (5), and (3), finally yielding:

$$[\ell_2(g_r)] \geq [\ell_1(g_r)] \geq [\ell_i] \tag{10}$$

But

$$[\ell_i] = [\ell_4(g_r)] = [\ell_2(g_r)] \tag{11}$$

which when substituted in (10) yields

$$[\ell_2(g_r)] = [\ell_1(g_r)] \tag{12}$$

By Corollary 3.3

$$[\ell_2(g_r)] = [\ell_2(g_i)] \text{ and } [\ell_1(g_r)] = [\ell_1(g_i)]$$

From these two equalities and (12) we obtain

$$[\ell_2(g_i)] = [\ell_1(g_i)]$$

as required.

Q. E. D.

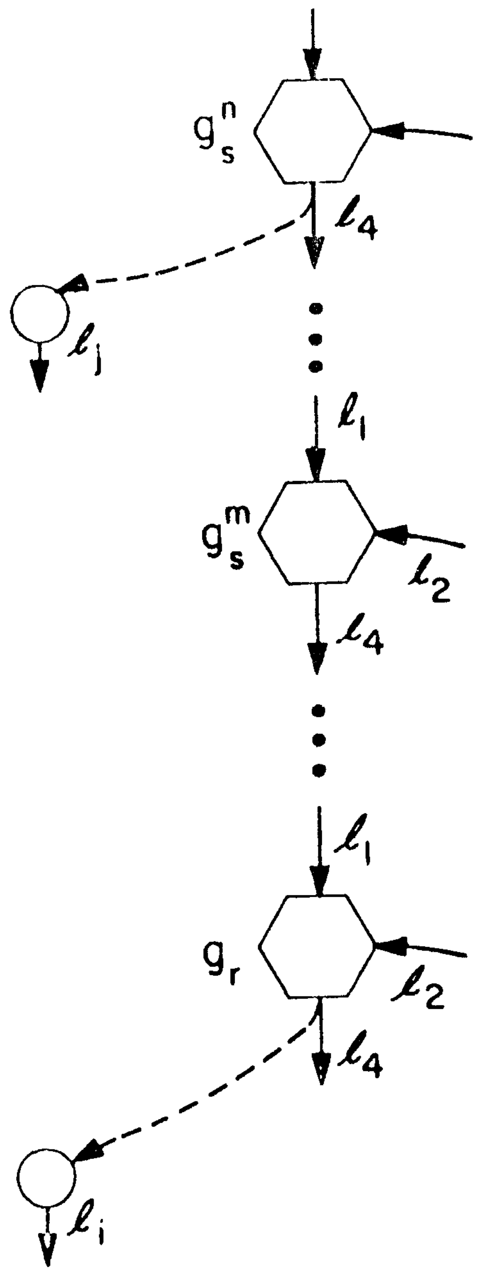


Fig. 3.11 The Link Names in the Iteration Step of Lemma 3.5

Theorem 3.9 Let g_1 and g_2 be loop junctions contained in the G-sets of mscs's K_1 and K_2 respectively, where $K_1 \subset K_2$. If there exists a link ℓ which belongs to both K_1 and K_2, then the final state of a terminal execution sequence of P is not a hang-up state only if $[\ell_2(g_1)] = [\ell_1(g_1)]$.

Proof: There must be an operator, selector, or junction which has inputs belonging to K_1 and K_2 respectively. Then the theorem follows immediately from Lemma 3.5.

The following corollaries of Theorem 3.9 relate the cycle structure to sources of hang-up states. The cycle structures are sketched in Figure 3.12 where each circle denotes an mscs.

Corollary 3.5 If a simple graph P has a node other than a loop junction with input links belonging to mscs's K_1 and K_2 and $K_1 \subset K_2$, then either no cycle of K_1 is ever effective or P has a hang-up state for some execution sequence.

Proof: By Theorem 3.9, for each loop junction g_1 of K_1 $[\ell_2(g_1)] = [\ell_1(g_1)]$ or there is a hang-up state. But this condition merely says that the feedback link of a loop junction is enabled or disabled the same number of times. This must occur for every execution sequence. Thus, for no execution sequence does the cycle repeat.

Q. E. D.

The proof of the remaining corollaries is completely analogous and will not be given.

Corollary 3.6 If a simple graph P has two nodes a_1 and a_2 with input links belonging to mscs's K_1 and K_3, and K_2 and K_4 respectively, where $K_1 \subset K_2$ and $K_4 \subseteq K_3$, then either no cycle of K_1 is ever effective or P has a hang-up state for some execution sequence.

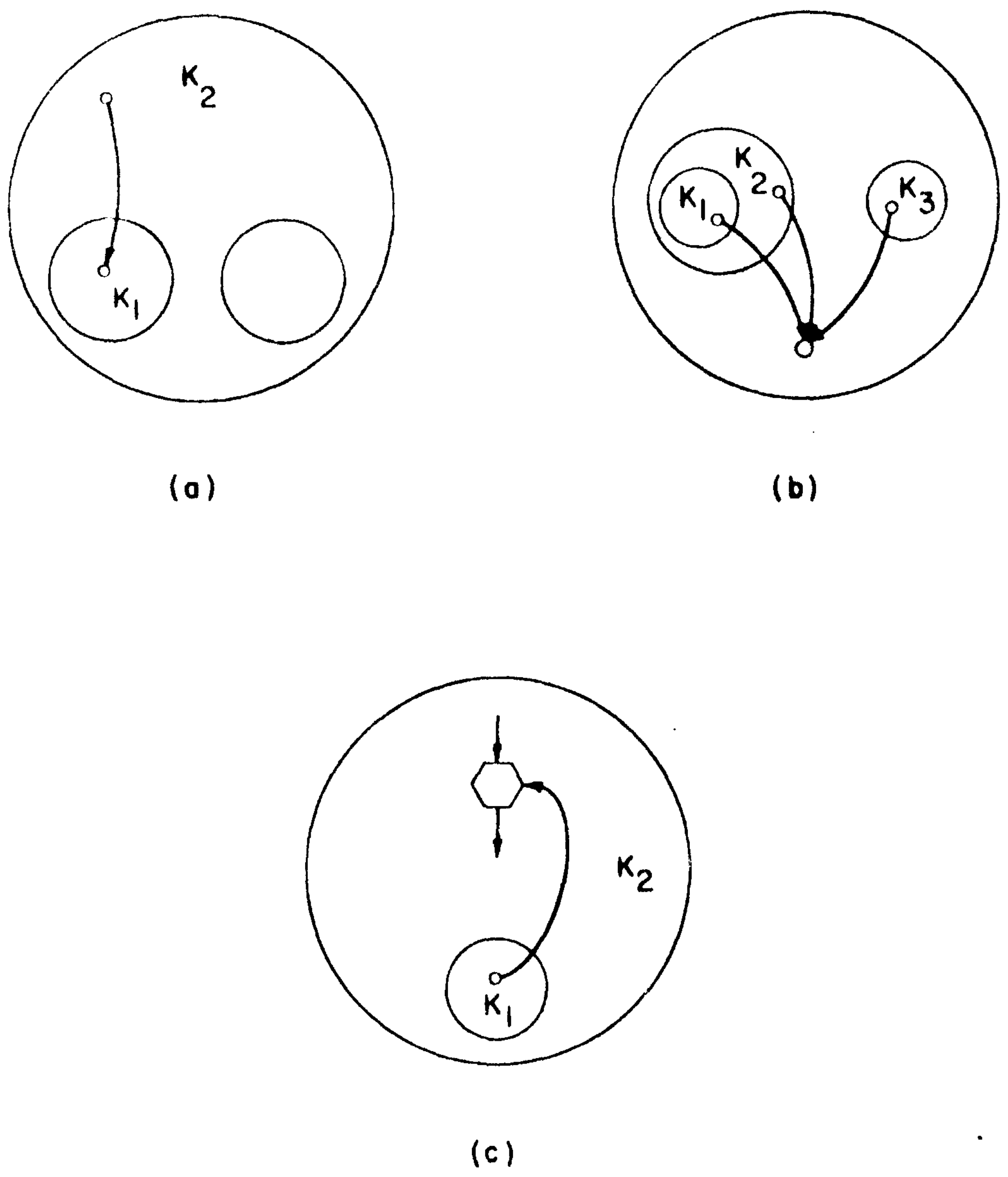


Fig. 3.12 Diagram of Paths Between mscs's which May Cause Hang-Up States

Corollary 3.7 Let P be a simple graph, K_2 an mscs of P, and g_1 a loop junction in the G-set of K_2. If the feedback link of g_1 belongs to an mscs K_1 and $K_1 \subset K_2$, then either no cycle of K_1 is ever effective or P has a hang-up state for some execution sequence.

Combining Lemma 3.4 and Corollary 3.6 we obtain:

Corollary 3.8 Let P, K_2, and g_1 be as in Corollary 3.6. If the feedback link of g_1 does not belong to K_2, then either no cycle of any mscs K_1 where $K_1 \subset K_2$ is ever effective or P has a hang-up state for some execution sequence.

Corollaries 3.5 and 3.6 tell us that a node in an inner cycle should not depend directly on a node in a containing cycle unless the node in the inner cycle is a loop junction. Corollary 3.7 tells us that loop outputs have to be used in order to nest cycles. For example, in the graph of Figure 3.4a loop junction g_1 does not satisfy the condition of Corollary 3.7. We have already seen in Section C how the hang-up states of this graph arise.

3. Graphs of Type I

We will now restrict our attention to a class of program graphs having the property that a hang-up state can arise only if two input links of a junction are placed in enabled status during an execution sequence. Graphs with this property will be called graphs of type I. Under suitable assumptions on the behavior of the selectors, we will give a procedure to test whether or not hang-up states can arise in graphs of Type I.

The motivation for the following definition comes from Corollary 3.8 which states that if the feedback input of a loop junction in the G-set of an mscs does not belong to the mscs, then all cycles of inner mscs's are useless. Thus, if we require that the feedback input of a loop junction belong to the same mscs as the outputs of g, we do not seriously restrict the class of program graphs. If, in addition, we require that both inputs to a loop output belong to the same mscs, then condition 2 of Theorem 3. 6 implies condition 1.

Definition 3. 5 A simple graph is said to be of Type I only if:

1. The initial inputs of all loop junctions in the same G-set belong to the same mscs.
2. If a link ℓ belongs to mscs's K_1 and K_2, then the initial inputs of g_1 and g_2, where $g_1 \epsilon G_1$ and $g_2 \epsilon G_2$, belong to the same mscs.
3. The input links of every loop output belong to the same mscs.

Corollary 3. 9 The final state of a terminal execution sequence of a graph of Type I is not a hang-up state iff whenever ℓ_i and ℓ_j belong to the same mscs, $[\ell_i] = [\ell_j]$.

Proof: Immediate from the definition and Theorem 3. 8.

If P is a graph of Type I, we associate enabling functions with the links of P in the following manner: Let P^n be the cycle free graph generated by the cycle decomposition of P. Every link of P is also a link of P^n. To obtain the enabling functions, we apply, in P^n, rules 1 through 5 of Section C. 3 plus the following three additional rules to deal with loop junction and loop outputs.

6. If ℓ is the data output link of a loop junction g assign to it the function E_1 associated with the initial input of g.
7. If ℓ is the control output link of a loop junction g assign to it the function E_2 associated with the feedback input* of g.

* The feedback input of g is not attached to the loop junction in the graph P^n.

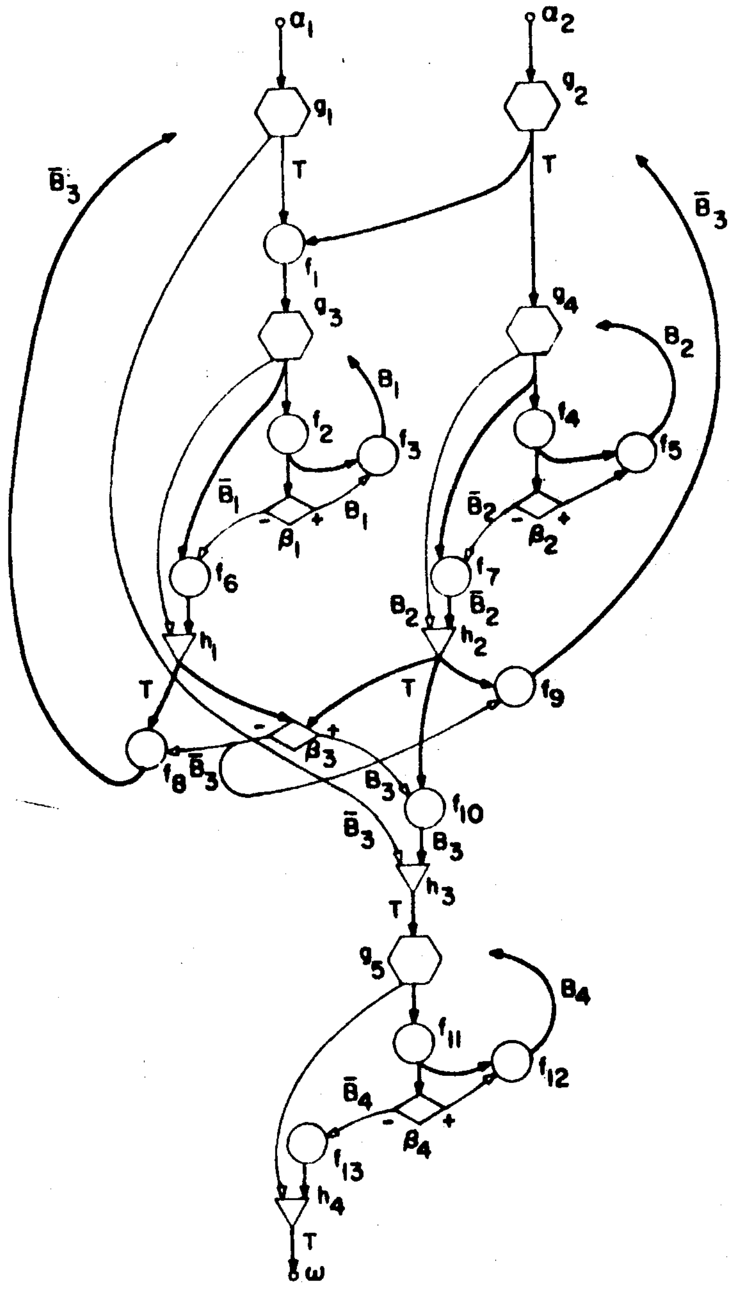


Fig. 3.13 The Enabling Functions of a Cyclic Graph

8. If ℓ is the output link of a loop output assign to it the function $E_1 \vee E_2$ where E_1 and E_2 are the enabling functions associated with the input links.

Rules 7 and 8 define the enabling function of the output link of a loop output in terms of the function of the feedback input of the associated loop junction. Since the definition of graphs of Types I require that the initial and feedback inputs of a loop junction belong to different mscs's, it follows, by Lemma 3. 3, that the enabling function of each link is uniquely defined, i. e. application of rules 6, 7, and 8 will not give rise to an endless loop. Figure 3. 13 illustrates the application of these rules for the graph previously decomposed in Figure 3. 8. The interpretation of the enabling function associated with links of loop outputs by rule 8 is, as with all other enabling functions, that the link cannot be enabled when the value of the function is false. By checking the transition table for loop outputs and keeping in mind rule 7, it can be verified that the assignment of rule 8 is consistent with this interpretation and the conventions established in Section C. 3.

Strictly speaking, the enabling functions associated with the links of P^n are applicable only to those execution sequences during which no cycle of P repeats. However, in order to guarantee that no hang-up states can occur, the enabling functions of feedback inputs of loop junctions covered by 1 and 2 of Definition 3.5 have to be equivalent (i. e. they must be false, and therefore the link disabled, under the same conditions). For, it is clear that if this condition is not met, then we could find execution sequences for which corollary 3. 9 is not satisfied. This is illustrated in the example of Figure 3. 14 where the output link of the operator labelled f_3 belongs to both mscs of the graph. The cycles of each mscs are independent of the other. Thus, during some execution sequence one cycle is enabled more times than the other and a hang-up state thereby arises.

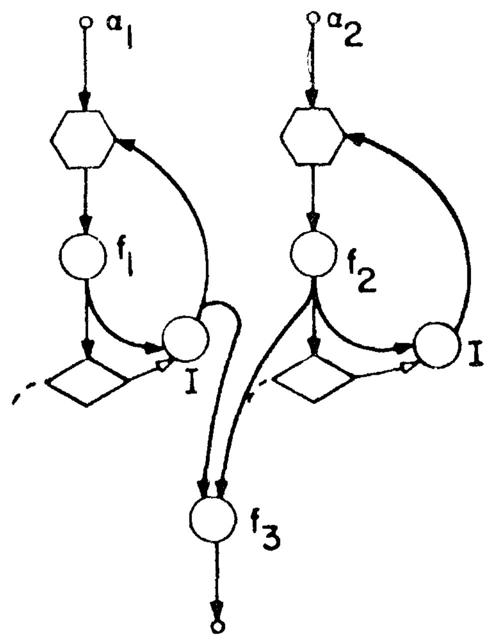


Fig. 3.14 A Possible Hang-Up State Arising from mscs with Different Feedback Enabling Functions

We formalize this observation as follows:

Lemma 3.6 The final state of each terminal execution sequence of a graph of Type I is not a hang-up state only if the enabling functions of the feedback input links of loop junction satisfying 1 and/or 2 of Definition 3.5 are equivalent.

It is convenient to introduce the term G*-set to denote the union of the G-sets of the mscs's K_1 and K_2 such that there exists a link belonging to both K_1 and K_2. Clearly, any two G*-sets are disjoint. To extend the enabling function to those execution sequences where cycles repeat, we observe that whenever the feedback input of the loop junctions of a G*-set become enabled, the next value of the enabling function can be obtained by simply setting the data output link of all members of the G*-set to the identity function. The enabling functions so obtained applies to all succeeding repetitions of the cycles, and should also satisfy Lemma 3.6. We will call these functions the $P^{n'}$ enabling functions.

Theorem 3.8 Let P be a graph of Type I. The final state of all terminal execution sequences of P is not a hang-up state only if*:

1. The P^n and $P^{n'}$ enabling functions of the feedback inputs of all members of each G*-set are equivalent and different from the identity element.
2. If E_i and E_j are the $P^n(P^{n'})$ enabling functions of two inputs of a junction, then $E_i \wedge E_j \equiv \text{false}$.

Proof: The first half of part 1 follows directly from Lemma 3.6. The second half, i.e. that the enabling function be different from the identity element, is also needed. For, otherwise the feedback links of the corresponding G*-set will never be disabled and therefore they could not be in idle status in the final state.

* As in Section C.3, here we assume that no two selectors of P have the same label.

In order to show the necessity of part 2, consider the possible execution sequences of a graph in the light of its cycle structure. First, if the condition $E_i \quad E_j$ false is not satisfied for some junction in P^n, then, by Theorem 3.5, there is at least one execution sequence during which a hang-up state occurs, i.e. that sequence during which no cycle of the graph repeats. Now, let $K_1, K_2, \ldots K_r$ be the mscs's in the first level of the cycle structure. Set the output links of the loop junctions in their G-sets and compute the $P^{n'}$ enabling functions for all links belonging to an mscs K, where $K \quad K_i$, $1 \leq i \leq r$. All other links of the graph are discarded. By the definition of the G*-sets, the result is a set of disjoint graphs such that the data output links of a G*-set obtained from $G_1, G_2, \ldots G_r$ correspond to the input terminals of a graph. The argument used for the P^n enabling function is valid for each of these graphs and consequently $E_i \wedge E_j \equiv$ false at each junction as well. This process is repeated until the cycle structure is exhausted.

Q. E. D.

The proof of Theorem 3.8 contains the essence of a procedure to test whether a graph of Type I has hang-up states under the assumptions that all selectors are distinct and independent of one another.

IV. AN EQUIVALENCE PROBLEM

A. INTRODUCTION

The determinism of program graphs guarantee not only the uniqueness of the final state of terminal execution sequences with the same initial state, but also the uniqueness of the sequence of value and status pairs at any link. It is often the case that we are interested in observing identical value and status pairs in a subset of the links of the graph. Under these circumstances, the following two problems are of interest:

1. To determine of two program graphs P_1 and P_2 whether or not the value and status of a given link of P_1 is the same as the value and status of a given link of P_2.
2. To determine what kind of transformations can be performed on a program graph P_1 so that the transformed graph P_2 is equivalent (in the sense of 1) to the original graph P_1.

In this chapter we consider both of these problems when two values are to be considered the same if and only if they have identical functional expressions after elimination of identity functions. For example, if I denotes the identity function, then

$$f_1(I(f_2(x, y)), z) \equiv f_1(f_2(x, y), I(z))$$

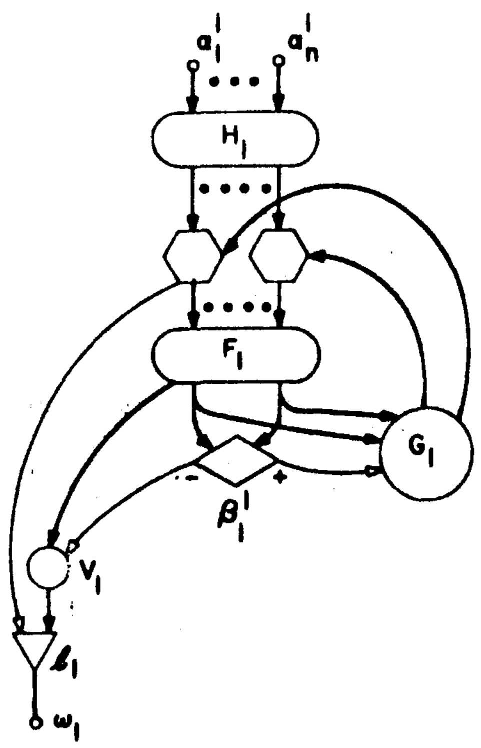
Section B presents a solution to the equivalence problem for graphs with a known, simple structure. Section C formalizes and generalizes the reasoning of Section B to program graphs with almost arbitrary structure. Theorem 4.1 in that section establishes necessary and sufficient conditions for the equivalence of two program graphs. Finally, Section D considers certain simple equivalence-preserving transformations.

B. A SIMPLE EQUIVALENCE PROBLEM

Let us consider the following problem: we are given two program graphs P_1 and P_2 -- whose structure is known and is as shown in Figure 4.1. Specifically, we know that both graphs have precisely one maximal strongly-connected subgraph, i.e. the boxes labelled H_1, H_2, F_1, F_2, G_1, G_2, V_1, and V_2 in Figure 4.1 are cycle free. Furthermore, we know that a cycle occurs in either graph only if the selectors labelled β_1^1 and β_1^2 respectively enable the connectors labelled '+'. Both graphs have the same number of input terminals and loop junctions, and there are no hang-up states in either one. We are asked to determine whether or not the value and status of the links labelled ℓ_1 and ℓ_2 in P_1 and P_2 respectively are the same for all sets of input values when the input terminals of both graphs are identified as indicated by the dotted lines in the figure. Assume for the moment that β_1^1 and β_1^2 are the only selectors in both P_1 and P_2. Now assume that it is possible to find execution sequences of P_2 which yields a value at ℓ_2 identical to the value obtained at ℓ_1 when the first activation of β_1^1 in P_1 enables the connector labelled '-', and when the first activation of β_1^1 enables its '+' connector and the second activation enables its '-' connector. Similarly, assume the same thing holds true when the roles of P_1 and P_2 are reversed. We claim that these two conditions imply the following:

1. $F_1H_1(x) \equiv F_2H_2(x)$
2. $F_1G_1(y) \equiv F_2G_2(y)$
3. $V_1(z) \equiv V_2(z)$

and it is clear that 1, 2, and 3 imply that to each execution sequence of P_1 producing any value at ℓ_1 there corresponds an execution sequence of P_2 producing the same value at ℓ_2 and vice versa.



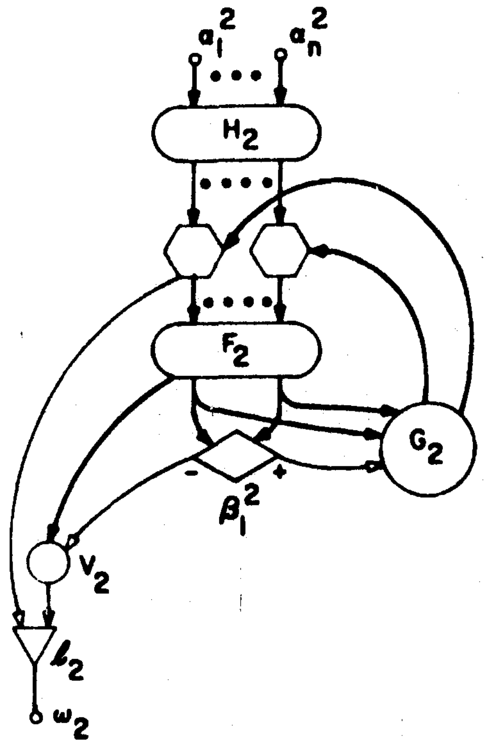


Fig. 4.1 Cyclic Graphs with the Same Cycle Structure

Let σ_1^1, σ_1^2 denote the two chosen execution sequences of P_1 and σ_2^1, σ_2^2 those of P_2.

For each of those sequences let us write the expression corresponding to the value and status of ℓ_1, ℓ_2 as follows:

$$\sigma_1^1 : \beta_1^-(F_1(H_1(x))) \rightarrow V_1(F_1(H_1(x))) \tag{1}$$

$$\sigma_1^2 : \beta_1^+(F_1(H_1(x))) \wedge \beta_1^-(F_1(G_1(F_1(H_1(x))))) \rightarrow V_1(F_1(G_1(F_1(H_1(x))))) \tag{2}$$

$$\sigma_2^1 : \beta_1^-(F_2(H_2(x))) \rightarrow V_2(F_2(H_2(x))) \tag{3}$$

$$\sigma_2^2 : \beta_1^+(F_2(H_2(x))) \wedge \beta_1^-(F_2(G_2(F_2(H_2(x))))) \rightarrow V_2(F_2(G_2(F_2(H_2(x))))) \tag{4}$$

The expression on the left hand side of the arrow denotes what sequence of selector outputs must be enabled in order for the value on the right hand side of the arrow to appear at the chosen link.

Now suppose that σ_1^1 is matched with σ_2^1 and σ_1^2 is matched with σ_2^2, i.e., $V_1(F_1(H_1(x))) \equiv V_2(F_2(H_2(x)))$ and $V_1(F_1(G_1(F_1(H_1(x))))) \equiv V_2(F_2(G_2(F_2(H_2(x)))))$.

In order for the left hand side of the arrows in (1) and (3) to be enabled under all circumstances we must have $F_1H_1 \equiv F_2H_2$. From this identity and the identity of the right hand sides it follows that $V_1 \equiv V_2$. A similar argument using the identity of (2) and (4) yields $F_1G_1 \equiv F_2G_2$.

On the other hand, if $F_1H_1 \equiv F_2H_2$ then (1) and (3) cannot be matched even if $V_1(F_1(H_1(x))) \equiv V_2(F_2(H_2(x)))$ for we have no assurance that $\beta_1^-(F_1(H_1(x))) = \beta_1^-(F_2(H_2(x)))$. Analogously, (1) and (4) cannot be matched either; in fact (1) cannot be matched with any execution sequence of P_2. Similarly if $F_1G_1 \equiv F_2G_2$, (2) cannot be matched with any execution sequence of P_2 either. This, however, contradicts the hypothesis and we must conclude that the only possible match is (1) with (3) and (2) with (4) which proves our original claim.

If we now allow $H_1, H_2, F_1, F_2, G_1, G_2, V_1$, and V_2 to be arbitrary cycle free graphs, the same reasoning still holds by considering every execution sequence which results in a different combination of selector connectors being enabled. The detailed justification of this assertion is best left to the next section.

C. THE EQUIVALENCE PROBLEM FOR ARBITRARY GRAPHS

In order to apply the foregoing reasoning to arbitrary program graphs we must be able to express the same concepts without any knowlege of the cycle structure. The concept of base sequences is introduced with this aim.

Definition 4.1 A base sequence of a program graph P is a normal execution sequence during which at least one link of every proper cycle of P is enabled at most once.

Corollary 4.1 Every program graph has a finite number of base sequences.

Proof: Immediate from the definition and the fact that there are a finite number of proper cycles in a program graph.

The definition of base sequences immediately yields the following:

Corollary 4.2 If a_1, a_2 are two nodes of a program graph P and there exists a proper path from a_1 to a_2 all of whose links are enabled for some execution sequence of P then there is a base sequence of P during which these links are also enabled.

Corollary 4.3 Every execution sequence of a cycle free program graph is a base sequence.

Definition 4.2 An infinite cycle is a data cycle all of whose links are enabled for all execution sequences.

A program graph with an infinite cycle never reaches a final state. Infinite cycles can be detected by examining the relationship between the cycles and selectors of a graph.

Appendix A contains an algorithm to determine whether or not a program graph has infinite cycles and/or hang-up states. Furthermore the algorithm generates all the base sequences of the graph if neither of the above conditions are present.

We shall henceforth consider program graphs without infinite cycles and/or hang-up states.

If $\underline{\ell}$ is any link of a program graph, its value and status are, according to Theorem 2. 2, uniquely defined for every execution sequence. Accordingly, the execution sequences can be grouped into equivalence classes with respect to $\underline{\ell}$ as follows:

Definition 4. 3 Two execution sequences are in the same equivalence class with respect to a link $\underline{\ell}$ if and only if the value and status of $\underline{\ell}$ are identical upon completion of both sequences.

Since the value of a link is considered to be undefined when the link status is not the enabled status, there is one equivalence class for all sequences during which the link is not enabled.

In order to study the equivalence between program graphs we need only consider a suitable representative of each equivalence class. For this purpose we need a precise description of the circumstances under which a particular value is assigned to a link. It is not enough to know that the link is enabled. We also need to know what selector terminals are enabled when a value is obtained at the link. We will represent each equivalence class of execution sequences by a pair of functional expressions. The first member of the pair denotes the condition under which that value is obtained. The second member of the pair denotes the value assigned to the link. The condition part is a boolean expression using the connectives ' $\wedge$ ' (AND) and ' ' (OR) among terms of the form: $\beta^+(y_1, \ldots y_n)$ or $\beta^-(y_1, \ldots y_n)$ where y_i are any functional expressions denoting the values of the input links of the n-place selector β.

The notation is best clarified by an example. The pair:

$$\ell : \beta_1^+(a_1) \wedge \beta_2^-(f_2(a_1, a_2)) \vee \beta_1^-(f_3(a_1)) \rightarrow f_1(f_2(a_1, a_2), f_3(a_1))$$

denotes that link $\underline{\ell}$ is assigned the value $f_1(f_2(a_1, a_2), f_3(a_1))$ if either selector β_1 enables its '+' connector when its input has the value a_1 and selector β_2 enables its '-' connector when its input has a value $f_2(a_1, a_2)$, or when selector β_1 enables its '-' connector when its input has a value $f_3(a_1)$.

Note that the first member of the pair is closely related to the enabling function used in Chapter III. There it was specifically assumed that all selectors of a program graph were distinct and therefore it was sufficient to consider only the selector label. In the context of this chapter such an assumption is too restrictive, yet the need to uniquely distinguish each selector value is preserved by the use of the functional notation as above.

The two members of a pair characterizing an equivalence class of execution sequences will be called dynamic enabling function (def) and value respectively.

Given two dynamic enabling functions it is possible to determine whether or not they are equivalent by replacing each expression of the form $\beta_i^+(\ldots)$, $\beta_i^-(\ldots)$ with single boolean variables according to the following rules:

1. Two expressions $\beta_i^{\pm}(y_1, y_2, \ldots y_n)$ and $\beta_j^{\pm}(y_1', y_2', \ldots y_n')$ are assigned the same variable iff $i = j$, $y_i \equiv y_i'$, $1 \leq i \leq n$.

2. If $\beta_i^+(y_1, y_2, \ldots y_n)$ is assigned the variable B_i, then $\beta_i^-(y_1, y_2, \ldots y_n)$ is assigned its negation $\bar{B}_i$

From the foregoing and the definition of equivalence class we obtain:

<u>Corollary 4.4</u> Two execution sequences belong to the same equivalence class with respect to a link $\underline{\ell}$ iff their characterizing pairs exhibit identical values and equivalent dynamic enabling functions.

Now we give a precise formulation of the equivalence problem for program graphs.

Let P_1 and P_2 be two program graphs without infinite cycles and/or hang-up states. Nodes of P_1 and P_2 tagged with the same label denote the same function or predicate. Furthermore, we assume that if β_i is an n-place predicate and y_1 and y_2 are n-place value expressions, the only constraint on the value of β_i is that $\beta_i^+(y_1)$ and $\beta_i^-(y_2)$ may occur during the same execution sequence only if $y_1 \not\equiv y_2$. Let the input terminals of P_1 and P_2 be denoted by $\alpha_1, \alpha_2, \ldots \alpha_n$ and $\alpha_1', \alpha_2', \ldots \alpha_m'$, respectively. If ℓ_1 and ℓ_2 are links of P_1 and P_2, the notation $\ell_1(\alpha_1, \alpha_2, \ldots \alpha_n)$ and $\ell_1(\alpha_1', \alpha_2', \ldots \alpha_m')$ denotes the value assigned to the links as a function of the values assigned to the input terminals of P_1 and P_2, respectively. Let X denote the set of values which can be assigned to these input terminals and Φ a mapping which associates each α_i, α_i' with a member of X.

<u>Definition 4.4</u> P_1 and P_2 are <u>equivalent with respect to ℓ_1 and ℓ_2</u> under the mapping Φ iff to every equivalence class of the execution sequences of P_1 there corresponds an equivalence class of the sequences of P_2 such that $\ell_1(\Phi(\alpha_1), \Phi(\alpha_2), \ldots \Phi(\alpha_n)) \equiv \ell_2(\Phi(\alpha_1'), \Phi(\alpha_2'), \ldots \Phi(\alpha_m'))$ and the corresponding dynamic enabling functions are equivalent.

This is a very strong definition of equivalence. It requires that to each sequence of functional applications of one graph there corresponds an identical sequence of the other graph. Furthermore, it assumes that it is always possible to assign to a selector β a predicate function such that if y_1 and y_2 are suitable arguments of this predicate and $y_1 \neq y_2$ then either $\beta^+(y_1)$ or $\beta^-(y_1)$ may be enabled concurrently with $\beta^+(y_2)$ or $\beta^-(y_2)$.

This form of equivalence is closely related to the equivalence of program schemata as defined by Ianov. There are two important differences, however, which makes our definition a non-trivial generalization of Ianov's. First, we specifically require that during no execution sequence it occurs that $\beta_i^+(x)$ and $\beta_i^-(x)$ are enabled; second, we allow functions of any number of arguments and parallel evaluation of functions.

Neither of these situations can be expressed in Ianov's program schemata which can only represent strictly sequential application of single-input, single-output functions and where the predicate variables, which perform the selection function in program schemata, are distinct and their values at any one time areindependent of the past history of the process. In fact, Rutledge has shown that program schemata are equivalent to finite state devices whose inputs are sequences of allowable predicate variable vectors and whose outputs are sequences of operators (i. e. values so defined by Ianov). This implies that the allowable input and output sequences of program schemata are regular sets. This assertion does not hold, in general, for program graphs even if the first condition mentioned above were not required.

The following theorem gives necessary and sufficient conditions for the equivalence of two program graphs in terms of the base sequences of each graph.

Theorem 4.1 Let P_1, P_2, Φ, ℓ_1, and ℓ_2 be as before. P_1 and P_2 are equivalent iff to every equivalence class of the base sequences of P_1 there can be found an equivalent class of execution sequences of P_2 and vice versa.

Proof: If P_1 and P_2 are equivalent, the condition of the theorem must be satisfied since otherwise we would have found an execution sequence of P_1 not equivalent to any execution sequence of P_2 or vice versa.

Next, we show that if the condition of the theorem is satisfied, to every execution sequence of P_1, there can be found an equivalent execution sequence of P_2 and vice versa.

If none of the links of P_1 and P_2 are enabled more than once for any of their base sequences, then by Corollary 4.2, neither P_1 nor P_2 have any effective cycles and by Corollary 4.3 it follows that the condition of the theorem considers all execution sequences.

Now assume that there is a base sequence of P_1 during which some link is enabled more than once.

We recall from Chapter III that every cycle of a program graph without hang-up states must pass through a loop junction. Since all inputs of loop junctions are data links, it follows that every cyclic graph without hang-up states does not have control cycles. If during any execution sequence any link is enabled more than once, all links of some cycle must have been enabled.

P_1 and P_2 do not have any infinite cycles. This can occur only if for each proper cycle there is at least one selector capable of disabling (and therefore also enabling) a link of the cycle.

Let $B = \beta_{i_1}, \ldots, \beta_{i_n}$ be the set of selectors of P_1 which has the afore mentioned property for a given cycle.

From the definition of base sequences, there are such sequences during which the values assumed by members of B disable this cycle, and there are other base sequences during which the cycle is first enabled, then disabled. We shall refer to these base sequences as <u>acyclic</u> and <u>cyclic</u> respectively. Note that a sequence acyclic with respect to a cycle may be acyclic or cyclic with respect to another cycle. If a cyclic base sequence is in the same equivalence class as an acyclic one, then there is no execution sequence during which all links of the corresponding cycle are enabled more than once, i.e., all execution sequences involving the cycle are base sequences (e.g., the example of Figure 4.3). On the other hand, if a cyclic base sequence is not in the same equivalence class as any acyclic one then the cycle may repeat any number of times.

Our first task is to show that the condition of the theorem implies that to every such cycle in P_1 there corresponds an identical cycle in P_2. The dynamic enabling function of the cyclic base sequence of P_1 must have two instances of each of the members of the set B of selectors. Each of these instances must have at least one of its arguments different from the corresponding argument of the other, for otherwise the graph has infinite cycles. Since by hypothesis we have found an equivalent execution sequence σ of P_2, there are instances of members of B in P_2 which either are in cycles having the same relationship to a cycle as in P_1 or are not contained in any cycles. If it is the latter case, then there must exist a base sequence of P_2 identical to the sequence σ up to the second activation of members of B at which point, one or more members of B enable the complementary connectors instead. This second activation, therefore, yields values corresponding to the repetition of the cycle of P_1. It follows that the sequence of P_1 equivalent to this base sequence of P_2 enables the cycle once more and therefore the corresponding def has a third distinct instance of members of B. By repeating this argument as often as needed we conclude that either there are an infinite number of instances of B in P_2 or there is a set of these instances contained in some cycle. But the first alternative is not possible since a program graph is finite.

An entirely parallel argument can be used when the set B determines a data cycle of P_2.

The foregoing construction also shows that to any execution sequence of P_1 which enables a cycle any number of times, there corresponds an equivalent sequence of P_2 and vice versa. Thus, now we have shown that in addition to the base sequences, all sequences of one graph enabling one of the cycles an arbitrary number of times and all other cycles at most once have an equivalent sequence in the other graph. Using the correspondence of cycles and an analogous reasoning we conclude that the set of equivalence sequences can be extended to include those in which two cycles are repeated an arbitrary number of times and so on until all cycles are considered and therefore all possible execution sequences.

Q. E. D.

We illustrate the application of Theorem 4.1 by working out several examples. Throughout these examples we shall omit parentheses from functional expressions.

<u>Example 1</u>. We want to determine whether or not the graphs P_1 and P_2 shown in Figure 4.2 are equivalent with respect to the links ω_1 and ω_2.

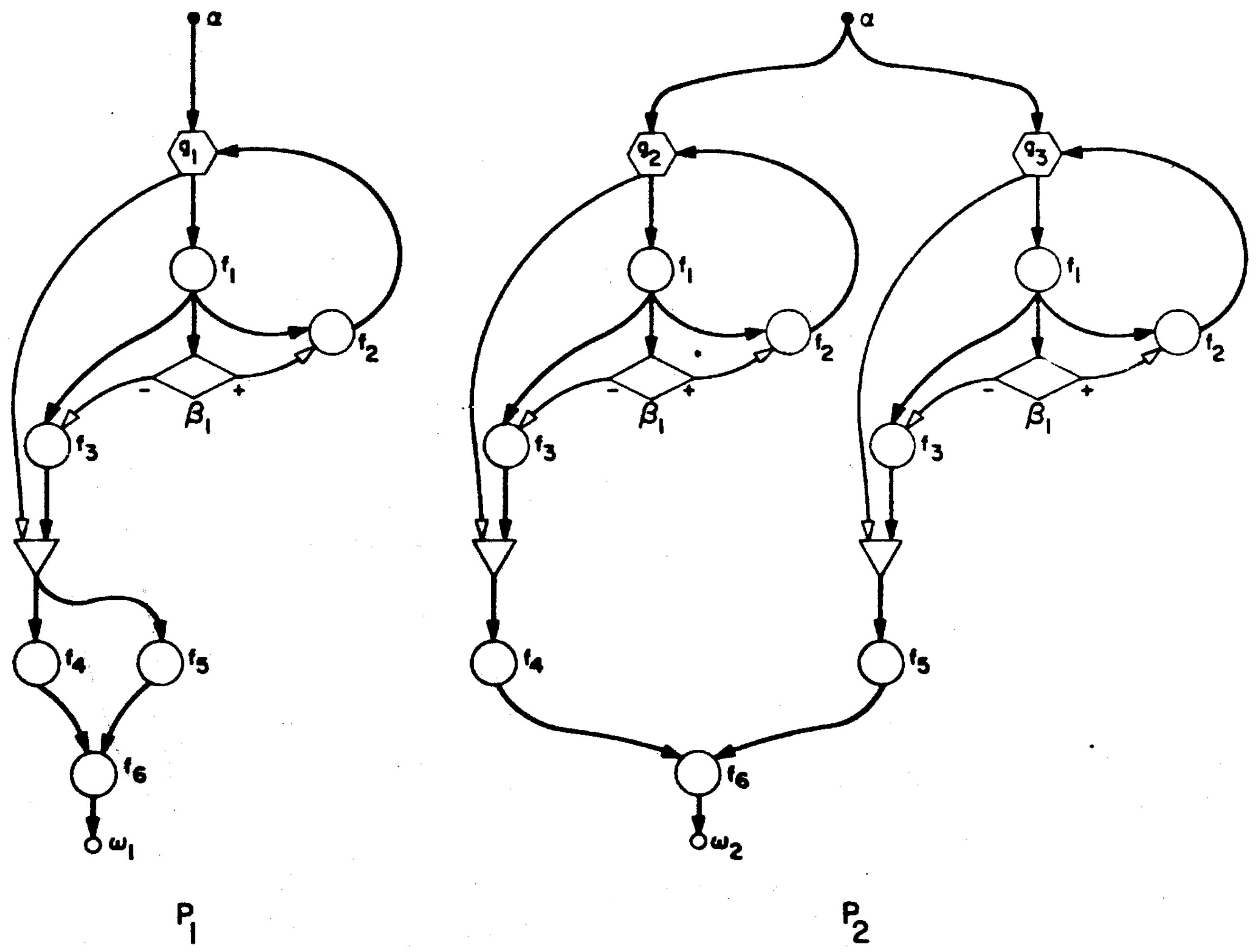


Fig. 4.2 The Two Program Graphs of Example 1

P_1 has two base sequences corresponding to the proper cycle passing through nodes g_1, f_1, and f_2.

$\sigma_1^1 : \beta_1^- f_1 a \rightarrow f_6 f_5 f_3 f_1 a f_4 f_3 f_1 a$

and

$\sigma_1^2 : \beta_1^+ f_1 a \wedge \beta_1^- f_1 f_2 f_1 a \rightarrow f_6 f_5 f_3 f_2 f_1 a f_3 f_1 f_2 f_1 a$

P_2 has four possible base sequences corresponding to the proper cycles passing through nodes g_2, f_1, and g_2 and g_3, f_1, and f_2.

$\sigma_2^1 : \beta_1^- f_1 a \wedge \beta_1^- f_1 a \rightarrow f_6 f_5 f_3 f_1 a f_4 f_3 f_1 a$

$\sigma_2^2 : \beta_1^- f_1 f_2 f_1 a \wedge \beta_1^+ f_1 a \wedge \beta_1^- f_1 a \rightarrow f_6 f_5 f_3 f_1 f_2 f_1 a f_4 f_3 f_1 a$

$\sigma_2^3 : \beta_1^- f_1 a \wedge \beta_1^- f_1 f_2 f_1 a \wedge \beta_1^+ f_1 a \rightarrow f_6 f_5 f_3 f_1 a f_4 f_3 f_1 f_2 f_1 a$

$\sigma_2^4 : \beta_1^+ f_1 a \wedge \beta_1^- f_1 f_2 f_1 a \wedge \beta_1^+ f_1 f_2 f_1 a \rightarrow f_6 f_5 f_3 f_1 f_2 f_1 a f_4 f_3 f_1 f_2 f_1 a$

However, σ_2^2 and σ_2^3 can not be execution sequences since both $\beta_1^+ f_1 a$ and $\beta_1^- f_1 a$ occur in their defs. It is readily checked that $\sigma_1^1 \equiv \sigma_2^1$ and $\sigma_1^2 \equiv \sigma_2^4$ and thus P_1 and P_2 are equivalent.

Example 2. P_1 and P_2 are as shown in Figure 4.3. P_1 has eight possible base sequences corresponding to the proper cycles passing through nodes g_1, f_2, I and g_2, f_2, f_3.

$\sigma_1^1 : \beta_2^+ a \wedge \beta_1^+ f_1 a \rightarrow f_4 f_1 a$

$a_1^2 : \beta_2^+ a \wedge \beta_1^- f_1 a \wedge \beta_1^- f_2 f_1 aa \rightarrow f_4 f_2 f_1 aa$

$\sigma_1^3 : \beta_2^+ a \wedge \beta_1^- f_1 a \wedge \beta_1^+ f_2 f_1 aa \wedge \beta_1^+ I f_2 f_1 aa \rightarrow f_4 I f_2 f_1 aa$

$\sigma_1^4 : \beta_2^+ a \wedge \beta_1^- f_1 a \wedge \beta_1^+ f_2 f_1 aa \wedge \beta_1^- I f_2 f_1 aa \wedge \beta_1^- f_2 I f_2 f_1 aa f_3 f_2 f_1 aaa$
$\rightarrow f_4 f_2 I f_2 f_1 aa f_3 f_2 f_1 aaa$

$\sigma_1^5 : \beta_2^- a \wedge \beta_1^+ f_1 a \rightarrow f_4 f_1 a$

$\sigma_1^6 : \beta_2^- a \wedge \beta_1^- f_1 a \wedge \beta_1^- f_2 f_1 aa \rightarrow f_4 f_2 f_1 aa$

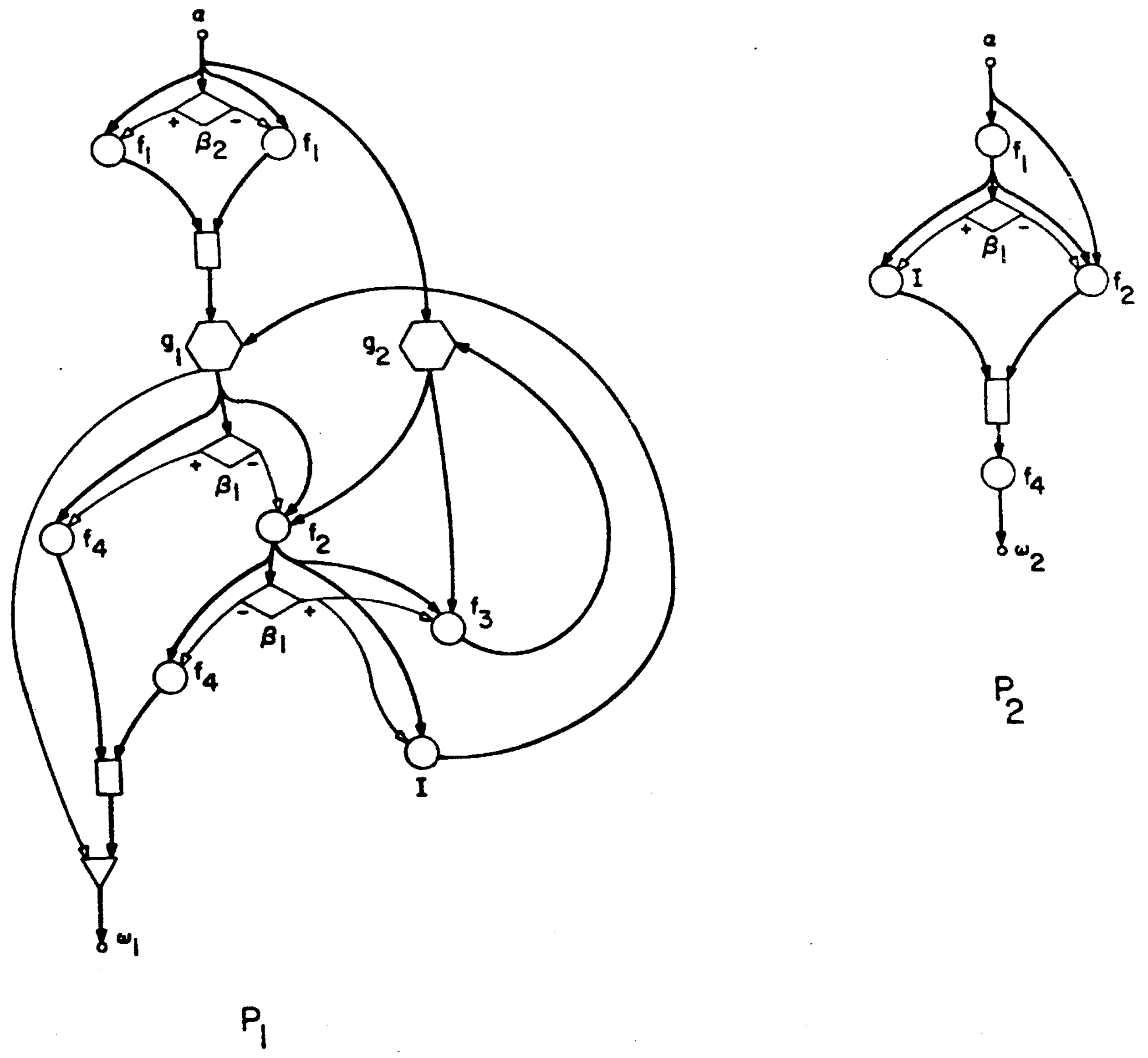


Fig. 4.3 The Two Program Graphs of Example 2

$$\sigma_1^7 : \beta_2^- a \wedge \beta_1^- f_1 a \wedge \beta_1^+ f_2 f_1 aa \wedge \beta_1^+ I f_2 f_1 aa \rightarrow f_4 I f_2 f_1 aa$$

$$\sigma_1^8 : \beta_2^- a \wedge \beta_1^- f_1 a \wedge \beta_1^+ f_2 f_1 aa \wedge \beta_1^- I f_2 f_1 aa \quad \beta_1^- f_2 I f_2 f_1 aa f_3 f_2 f_1 aaa$$
$$\rightarrow f_4 f_2 I f_2 f_1 aa f_3 f_2 f_1 aaa$$

After eliminating identity functions (I) from all expressions, there results the following equivalence classes:

$$\{\sigma_1^1, \sigma_1^5\} : \beta_1^+ f_1 a \rightarrow f_4 f_1 a \qquad (1)$$

$$\{\sigma_1^2, \sigma_1^3, \sigma_1^6, \sigma_1^7\} : \beta_1^- f_1 a \rightarrow f_4 f_2 f_1 aa \qquad (2)$$

where σ_1^4 and σ_1^8 have been eliminated because both $\beta_1^+ f_2 f_1 aa$ and $\beta_1^- f_2 f_1 aa$ appear in their defs. The equivalence classes in (1) and (2) are in one to one correspondence with the two base sequences of P_2.

In order to establish that the equivalence problem is solvable, all that remains is to show that there is an effective procedure to test the conditions of Theorem 4.1.

The algorithm of Appendix A generates all the base sequences of a program graph. Determining the equivalence of two dynamic enabling functions is a solved problem of the propositional calculus. The only remaining difficulty is to find an execution sequence of a graph equivalent to a base sequence of the other. If such a sequence cannot be found, it is conceivable that we may never know when to stop the search. That this is not the case is easily discernible from the fact that for two sequences to be equivalent, their values have to be identical. Therefore, all we have to do is generate all possible values whose lengths are less than or equal to the length of the longest value in any base sequence. By a simple modification, the algorithm of Appendix A can be used for this purpose. All thatis needed is to place a higher upper bound in the number of times a

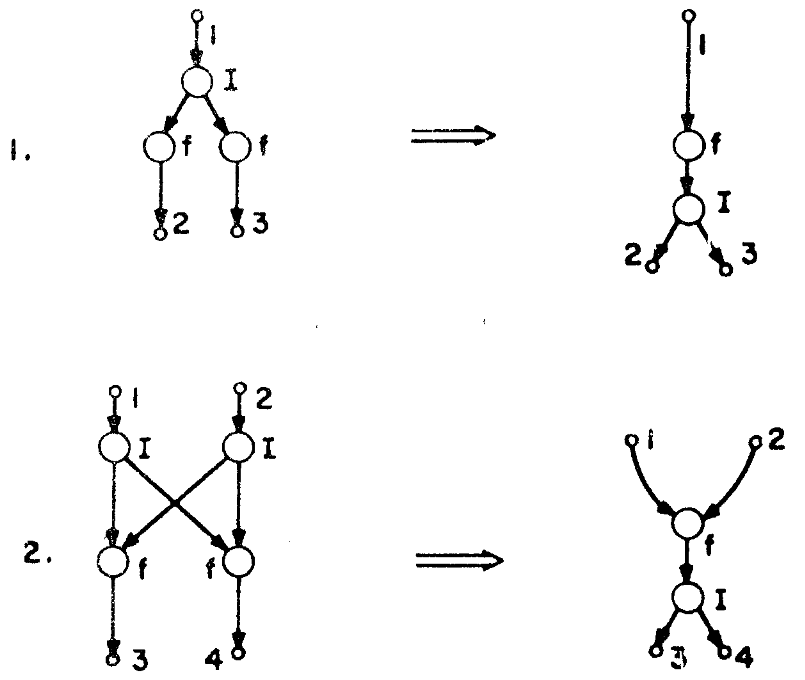


Fig. 4.4 Transformations to Identify Common Subgraphs Disregarding Selectors and Junctions

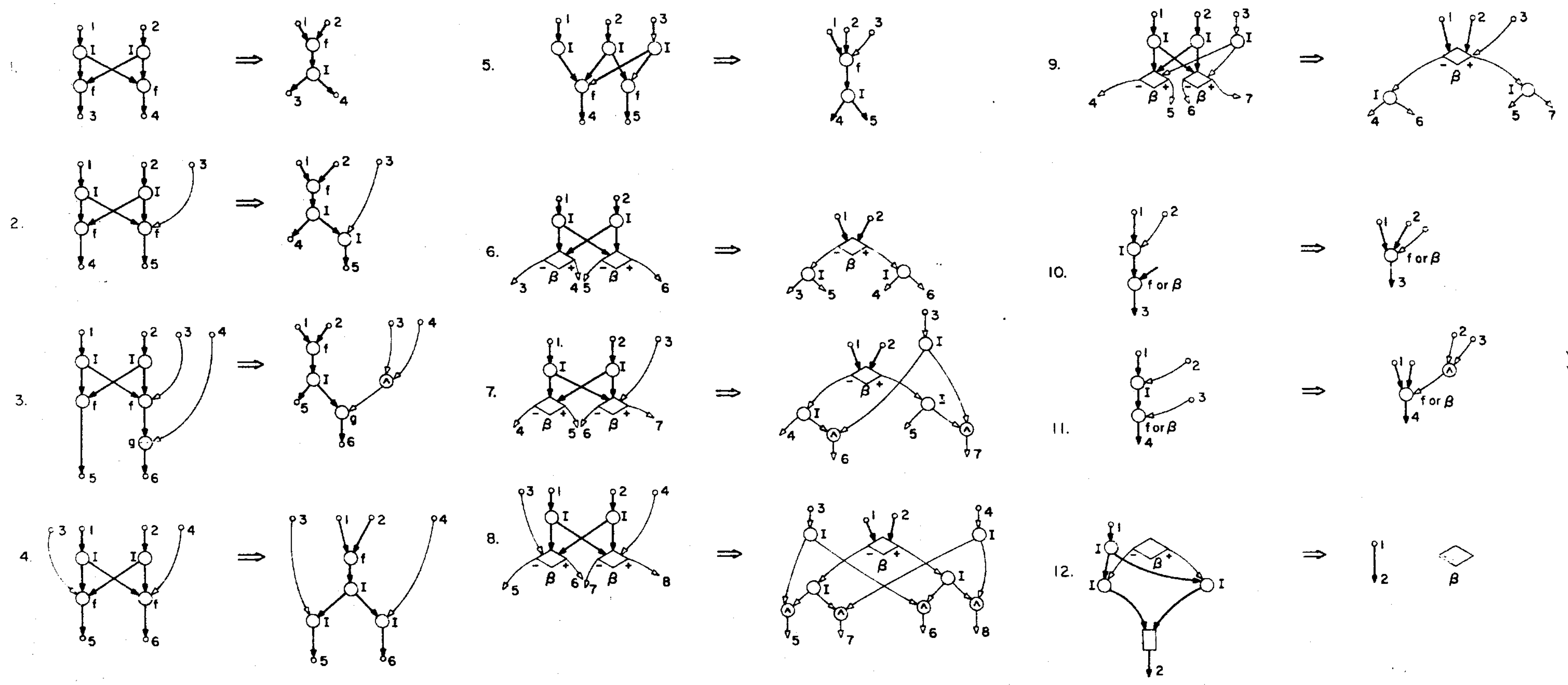


Fig. 4.5 Transformations to Identify Common Subgraphs Including Selectors and Junctions

cycle is allowed to repeat. If L is the length of the longest value in any base sequence and ℓ is the length of the shortest cycle in the program graph then $L/\ell + 1$ is a suitable upper bound.

We formalize these observations in the following theorem.

Theorem 4.2 The equivalence problem for program graphs is solvable.

D. SIMPLE TRANSFORMATIONS

In this section we will briefly indicate by means of examples the application of the results of the previous section to obtain equivalence preserving transformations of program graphs. In general, we seek to transform a graph in order to optimize a given criterion. For example, we may wish to minimize the number of operators and selectors associated with functions and predicates by identifying all identical subcomputations, or we may wish to transform a graph so that a function or predicate is applied only if the results of each application will actually be used by another computation during the course of the execution sequence; alternatively, we may want to speed up the average time of an execution sequence by performing as many computations as possible in anticipation of the possible utilization of their results, etc.

Ideally, we would like to obtain a set of elementary transformation schemes such that : 1) each scheme can be applied independently of each other to yield an equivalent graph, 2) the value of the criterion function in the transformed graph is not less (greater) than the corresponding value in the original graph, 3) each transformation is local, and 4) the set of transformations is complete. A transformation is said to be local if it can be applied to any subgraph without any knowledge of the structure within which the graph is embedded. Otherwise we say that the transformation is global. A set of transformations is said to be complete if whenever the

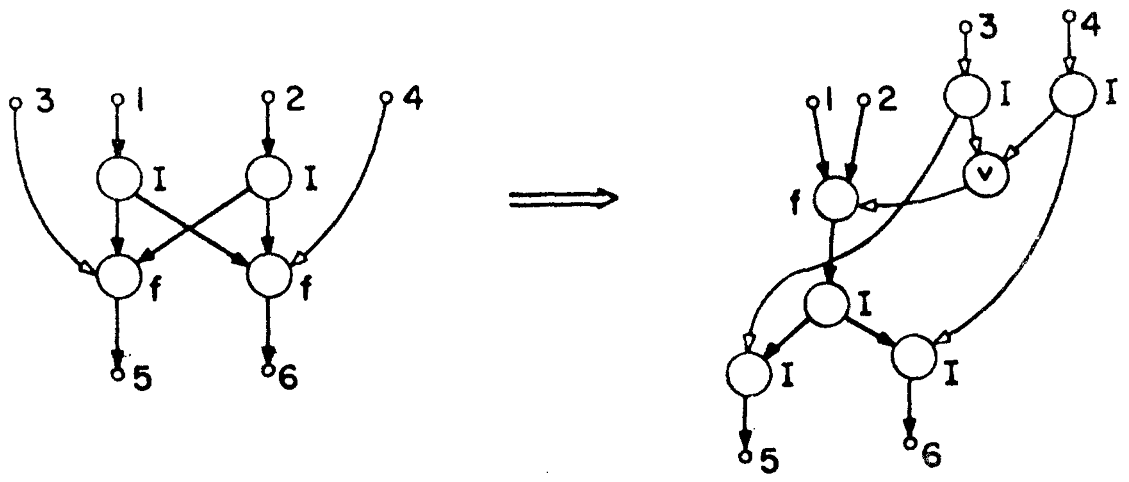


Fig. 4. 6 Alternative Form of a Transformation Scheme Which Does Not Preserve Equivalence

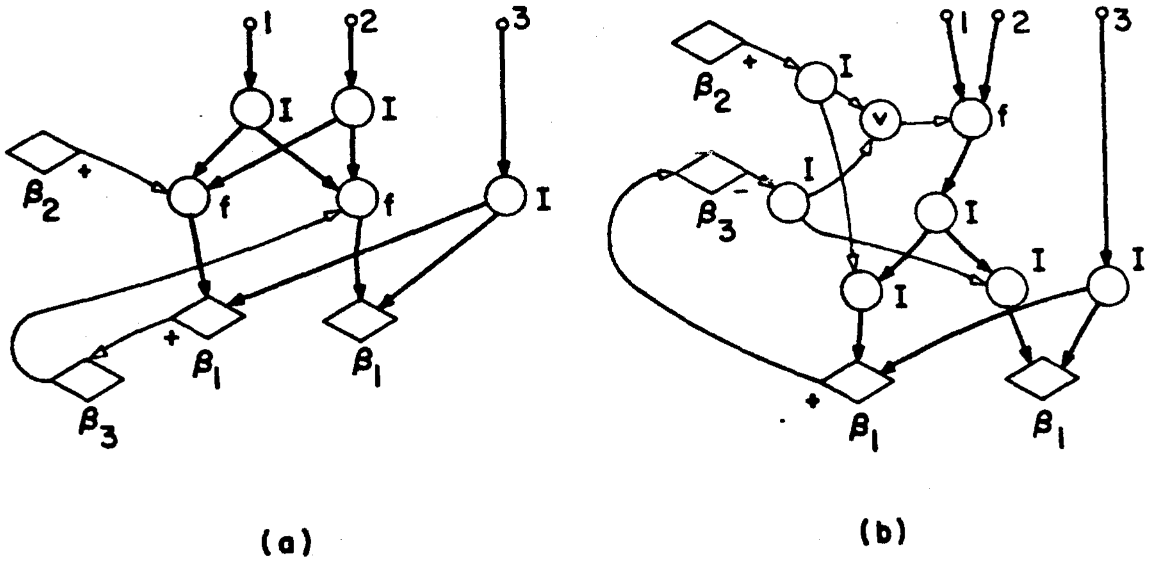
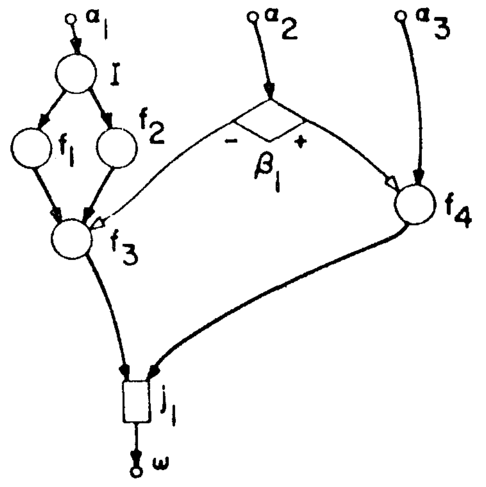


Fig. 4.7 Subgraph in which the Transformation of Figure 4.6 Introduces a Cycle

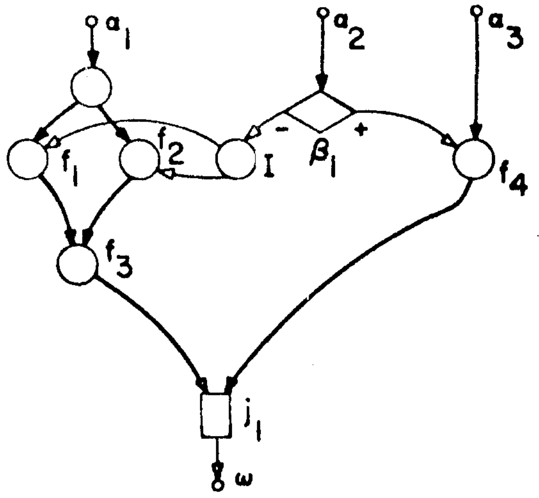
criterion function has a minimum (maximum), there is a sequence of transformations which obtains a graph exhibiting this minimum (maximum) value.

The transformation schema shown in Figure 4.4 satisfy each of the above conditions when the problem is to minimize the number of functional data operators in a simple cycle free graph consisting solely of nodes with one output and either one or two inputs. In order to include cycle free graphs with selectors and junctions, we may augment the set of transformations to include subgraphs with control links with or without selector nodes. The transformation scheme 4 in Figure 4.5 serves to illustrate a case in which care should be exercised in order to guarantee that the transformations are in fact local. An apparently reasonable alternative to this scheme is shown in Figure 4.6, however, this transformation is not local as can be verified by considering the graph of Figure 4.7. When transformation 4 is applied to both instances of selector β_1, there results the graph of Figure 4.7b which obviously cannot be equivalent to the original since it contains a cycle without any loop junction and therefore has hang-up states. Scheme 4 in Figure 4.5 avoids this difficulty by applying the control links labelled 3 and 4 in such a way that every path of the transformed graph is also a path of the original.

As an example of a problem which appears to be inherently global consider the transformation of a graph with the objective that a function or predicate is applied in an execution sequence only if the results of its application will be used by another computation during the execution. This objective can be achieved simply by moving the point of application of chosen control links. For example, in the graph of Figure 4.8a, operators f_1 and f_2 will be applied under all circumstances. However, their results



(a)



(b)

Fig. 4.8 Two Equivalent Program Graphs Differing on the Way a Control Link is Applied to Operators

will be used by operator f_3 only if β_1^- is enabled. To eliminate this condition, the control link β_1^- is applied at both f_1 and f_2 as shown in Figure 4.8b. The equivalence of both graphs is readily established. To see that this transformation is in fact global, imagine that there is a path from f_2 to β_1. If this is the case, applying the control link β_1^- to f_2 creates a cycle and the resulting graph is not equivalent to the original. Since the path from f_2 to β_1 can be of arbitrary length, there is no single finite rule that can accomplish this transformation.

V. CONCLUSIONS AND RECOMMENDATIONS

We have presented a deterministic model for the representation of parallel computations on non-structured data. The model incorporates data-dependent decisions and sufficient apparatus for precisely defining cyclic structures. Methods of analysis have been developed and a simple characterization of the hang-up states of a computation has been given. An equivalence problem for the model has been formulated and solved.

The main weakness of the model is its inability to represent computations on structured data. Further research is needed to determine whether the same conceptual framework used in this paper can be adapted to represent relations between an unbounded memory and a finite sequencing and control structure.

The results obtained in Chapter IV regarding the equivalence of program graphs suggest several areas of future research. McCarthy[15] has proposed that the formulation of a theory of equivalence is a basic step towards the development of a theory of programming. Very little progress has been made towards a satisfactory selection of this problem. Undoubtedly the difficulty of the general problem is related to the known unsolvability results of every computational model so far proposed, e.g. Turing machines, λ-calculus, Markov algorithms, etc. If we separate the computational aspects of a program from the pure control aspects, we can identify at least two sources of unsolvable problems. On one hand, it may be that the decision problem of the functional calculus for a given set of primitive functions and predicates is itself unsolvable. In this case, the equivalence problem is unsolvable even for the simplest programs, i.e. cycle-free programs. On the other hand, even if the aforementioned decision problem is solvable, it may be that the iterative or recursive structures expressible in the model give rise to unsolvable combinational problems. We have often looked at the combination of these two effects as permitting just too many ways of representing processes that do nothing. In our formulation both of these difficulties are avoided by 1) defining equivalence in terms of the identity of certain strings, and 2) allowing just one way of doing nothing, the identity function, and this in a manner which is easily detected. The equivalence obtained is too strong and ways must be sought to obtain weaker conditions.

It appears that under certain circumstances, the criterion given in Chapter IV may supply sufficient conditions for weaker forms of equivalence, i.e. if one can show that to each base sequence of one graph one can find an equivalent sequence of the other and vice versa, then the two programs are equivalent. It would be of interest to determine under what circumstances, if any, this conjecture holds. Also, there are some similarities between that criterion and the recursion-induction principle formulated by McCarthy.[15]

From the point of view of computational linguistics, a program graph may be considered as the definition of a grammar whose terminal symbols are the labels associated with the operators of the graph. The languages generated by these grammars include the finite-state languages but are not limited to that class. McNaughton[16] has studied the class of languages generated by parentheses grammars which are in turn a subset of the backward deterministic grammars. He has shown that the equivalence problem for parentheses grammars is solvable. It would be of interest to investigate the relationship between program graph-like grammars and either parentheses or backwards deterministic grammars.

Finally, the model and methods of analysis developed in this paper should be useful in the study of problems arising in the design of asynchronous multiprocessor computer systems.[11] In particular, the relatively small number of node types with which determinism is achieved constitute a workable basis for the design of macromodular systems.[3, 18, 22]

AN ALGORITHM TO GENERATE BASE SEQUENCES

APPENDIX A

The algorithm described in this section generates all base sequences of a program graph without hang-up states and/or infinite cycles. If the program graph does not satisfy these conditions, an appropriate diagnostic is produced and the algorithm stops.

The generation of the base sequences is accomplished by simulating all possible execution sequences during which no proper cycle is reported more than once. The finiteness of the graph guarantees that the process eventually stops.

We assume a suitable representation of the program graph which allows for storing the state of the graph, i. e. the value and status of every link. In addition, five separate structures are used to keep track of the state of the simulation. These five structures are:

1. The selector choice list (C-list).
2. The selector value list (V-list).
3. The proper cycle list (P-list).
4. The dynamic enabling function (DEF).
5. The environment stack (E-stack).

C-list The C-list contains an entry for every selector in the graph. Associated with each entry is a list containing the possible outcomes of the application of the selector function, i. e. which output connector to enable. Initially, this list contains '+' and '-' for every entry in the C-list.

V-list The V-list contains an entry for every distinct selector label in the graph. Associated with each entry is a list of all outcomes of a selector having this label. The entries of these sublists also contain the input values corresponding to the outcome, e. g. $+(f_1(f_2(x, y)), z)$. Initially, these sublists are empty for every entry in the V-list.

P-list The P-list contains an entry for every proper cycle of the graph. Associated with each entry is a list of all links contained in the cycle. Each entry in these sublists have a count field which is initially set to zero.

DEF The DEF is a variable containing the dynamic enabling function for the simulated execution sequence.

The state of the graph, C-list, V-list, P-list, and DEF are collectively referred to as the environment.

E-stack The E-stack is a last-in-first-out list. . Each entry in the E-stack is an environment.

It is convenient to define the following operations:

1. Scan i, n_1

   This operation scans the graph looking for a node in an active configuration. For this purpose it is assumed that the nodes are ordered. The scan always starts at the first node. When an active node is found, the variable i is set to point to this node and control is transferred to the step following the scan. If no active node is found, control is transferred to the step labelled n_1.

2. Stack

   This operation obtains a copy of the environment and places it on top of the E-stack. The current environment remains unchanged.

3. Pop n_1, n_2

   This operation removes the top element from the E-stack and installs it as the current environment. A successful performance of the operation transfers control to the step labelled n_1. If the operation cannot be performed because the E-stack is empty, control is transferred to the step labelled n_2.

4. Outcome i, n_1, n_2

This operation picks a number from the list of outcomes found in the entry of the C-list corresponding to i which must point to a selector. If the list of outcomes is empty, control is transferred to the step labelled n_2. Otherwise, the outcome chosen is deleted from the list and the V-list is checked to verify that it is an allowable outcome for the selector i, i.e. if the outcome chosen is '+' and the input value is y, -(y) should not appear in the V-list. If the outcome is not allowed, the process is repeated. If the outcome is allowed the following sequence of operations is performed:

a. if outcome list is not empty, stack.

b. the outcome and input values are added to the V-list entry of i.

c. the DEF is augmented with the chosen selector value.

d. an indication of the desired outcome is set in the node.

e. control is transferred to the step labelled n_1.

5. Do transition i, n_1, n_2

This operation updates the state of the graph according to the transition table for the node pointed to by i. If i is a selector, the information set by step 4 of outcome is used. After updating the status and possibly the value of the links involved, the sub-lists of the P-list are searched for all instances of links which have been newly enabled. For each such instance found the count field is incremented by one. If this causes all members of a sublist to have a count greater than one, control is transferred to label n_2. Otherwise, control goes to label n_1.

6. Reset C The list of outcomes for all members of the C-list is initialized.

In terms of the above operations the algorithm is written as follows:

```
 1.  scan i, 13
 2.  if i is not a selector, loop junction, or final node, go to step 8
 3.  if i is a final node, go to step 11
 4.  if is is a loop junction, go to step 7
 5.  if the active configuration of the selector does not require a
     functional application, go to step 8
 6.  outcome i, 8, 15
 7.  if the status of the loop junction inputs is blocked and enabled
     respectively, reset C
 8.  do transition i, 1, 9
 9.  pop E 1, 15
10.  pop E 1, 16
11.  write DEF and value as a base sequence
12.  go to step 8
13.  if all links are in IDLE status, go to step 10
14.  report hang-up state and halt
15.  report infinite cycle and halt
16.  report successful completion and halt.
```

The algorithm generates all the base sequences by trying all allowable combinations of selector values. This is accomplished in step 6 by stacking the environment as it existed prior to every selector application. Before the stacking is performed, a note is made that a certain branch has been taken by removing the chosen selector outcome from the corresponding

C-list entry. Upon successful completion of a base sequence (step 13), th environment is restored to the point of the last selector application and the process continues. If at this point the E-stack is empty, all alternatives have been tried. At step 7, all outcomes are rehabilitated upon the occurrence of any cycle. All sequences which are not base sequences will eventually be caught in step 8. When this occurs, step 9 discards the current sequence and tries an alternative one. Note that if at this point no alternative is present, it can only mean that under no circumstances can the cycle be disabled and this is reported as an infinite cycle.

A similar situation occurs if in step 6 no allowable outcome can be found. In this case however, the infinite cycle may be caused by previously-chosen selector outcomes and need not occur for all execution sequences.

A hang-up state is reported whenever an active node cannot be found, yet these are links in a status other than IDLE.

VERIFICATION OF PROPERTIES OF THE TRANSITION TABLES

APPENDIX B

In this Appendix we verify properties a, b, c, d, and e of the transition tables in Table 2.1. These properties have been used in the proof of Theorem 3.8. If ℓ is a link, $[\ell]$ denotes the number of times that ℓ is enabled or disabled during an execution sequence.

Property a. For every loop junction-loop output pair of a simple graph (see Figure 3.9), $[\ell_1] = [\ell_e]$ iff ℓ_1 and ℓ_3 are idle.

Proof: By lines 1 and 2 of the transition table for loop outputs, ℓ_6 can be enabled or disabled only if ℓ_3 is enabled. By line 8 of the transition table for loop junctions, ℓ_3 can be enabled only if ℓ_2 is disabled and ℓ_1 is blocked. By lines 1 through 6 of this same table, ℓ_1 is enabled or disabled. Therefore, ℓ_6 is enabled or disabled at most once for each time that ℓ_1 is enabled or disabled.

First, assume that $[\ell_1] = [\ell_6]$. If $[\ell_1] = 0$ then ℓ_1 and ℓ_3 are necessarily idle since their status must be identical to that occurring in the initial state. Therefore, assume $[\ell_1] = [\ell_6] \neq 0$. The next to the last status of ℓ_3 must have been enabled. For, otherwise, by lines 3, 4, 5, and 6 of the loop output transition table and the argument of the previous paragraph $[\ell_1] = [\ell_6] + 1$. Thus, the last transition of the loop output was either line 1 or line 2 of the table and ℓ_3 is in idle status. Also, if ℓ_3 was last enabled, by line 8 of the loop junction table, ℓ_1 is also in idle status. This verifies the first part of the property. Now assume both ℓ_1 and ℓ_3 are in idle status. If ℓ_1 has been in enabled or disabled status at all, the last transition of the loop junction must have been line 8 of its table, which we have seen can occur at most once for every time ℓ_1 is enabled or disabled. Thus, the last transition of the loop output must have been either line 1 or line 2 of its table. Thus, ℓ_6 must have been enabled or disabled once for each time ℓ_1 has been enabled or disabled, i. e. $[\ell_1] = [\ell_6]$. This verifies the second part of the property.

Q. E. D.

Property b. For every loop junction of a simple graph, if ℓ_1 is idle, then $[\ell_2] = [\ell_4]$ iff $[\ell_2] = [\ell_3]$ and both ℓ_2 and ℓ_4 are idle.

Proof: First, assume $[\ell_2] = [\ell_4] = 0$. By lines 1 through 6 of the transition table, it is clear that $[\ell_1] = 0$ and, therefore, $[\ell_3] = 0$. Furthermore, ℓ_4 must be idle and since the graph is simple, by Lemma 3. 4, there is a proper path containing ℓ_4 and ℓ_2, and, by Corollary 3. 2, ℓ_2 is idle. Now, assume $[\ell_2] = [\ell_4] \neq 0$. By Lemma 3. 4 and Corollary 3. 2, ℓ_4 is enabled or disabled after ℓ_2, so that the only possible configurations of the loop junction are $\underline{2\text{-}1\ -\ -}$, $\underline{2\ 1\ -\ -}$, or $\underline{0\ 0\ -\ 0}$. But since ℓ_1 is idle, only the last configurations, i. e. 0 0 0 0 or 0 0 1 0 are allowed. In any event, ℓ_3 is disabled or enabled and, by lines 2, 3, 5, 6, 7, and 8 of the loop junction table, ℓ_3 is enabled or disabled every time that ℓ_2 had been in those statuses. Therefore, $[\ell_2] = [\ell_3]$ and ℓ_2 and ℓ_4 are idle. This shows the first part of the property. Next, assume $[\ell_2] = [\ell_3]$ and both ℓ_2 and ℓ_4 are idle. By the transition table, the last transition of the loop junction must have been line 8. Therefore, by the same argument used above, $[\ell_2] = [\ell_4]$.

Property c. For every loop junction of a simple graph, if ℓ_1, ℓ_2, ℓ_3, and ℓ_4 are idle, then $[\ell_2] = [\ell_3] = [\ell_4]$.

Proof: First, assume $[\ell_1] = 0$. Then $[\ell_3] = [\ell_4] = 0$ and by Lemma 3. 4 and Corollary 3. 2, $[\ell_2] = 0$ also. Now, assume $[\ell_1] \neq 0$. Since ℓ_1, ℓ_2, and ℓ_4 are idle, the last transition of the loop junction must have been line 8 of its table. By property b, $[\ell_2] = [\ell_4]$, and by lines 2, 3, 5, 6, 7, and 8 of the loop junction table, $[\ell_2] = [\ell_3]$. This shows that $[\ell_2] = [\ell_3] = [\ell_4]$.

Q. E. D.

Property d. All input links of an operator, selector, junction, or loop output have been enabled or disabled the same number of times iff each input link is idle or each is enabled or disabled.

Proof: For operators, selectors, and junctions the property is easily verified by noting, in Table 2.1, that all active configurations of these nodes require all inputs to be enabled or disabled, and that the completion of a transition places all the inputs in idle status.

In the case of loop outputs, lines 3, 4, 5, and 6 do not have this property. Lines 3 and 4 satisfy an equivalent condition since the transition places the input links in a status other than enabled or disabled. Thus, all it remains is to verify the property for lines 5 and 6, i.e. the active configurations $\underline{2\ 1\ 0}$ and $\underline{2\ {-1}\ 0}$. In both of these cases the link in enabled or disabled status remains in that condition after the transition. It follows that if all the input links are enabled or disabled the same number of times one of the active configurations 1, 2, 3, or 4 must eventually occur. Conversely, if all input links are idle, or either enabled or disabled this last condition must have occurred the same number of times, since consecutive occurrences of the configurations of lines 5 or 6 must eventually yield to one of the others.

Q. E. D.

Property e. If ℓ_i and ℓ_j are input and output links, respectively, of the same operator, selector, or junction, then $[\ell_i] = [\ell_j]$ iff ℓ_i is idle.

Proof: An examination of Table 2.1 yields that an output link of a node can become enabled or disabled only if all inputs to that node have been enabled or disabled. Furthermore, the transitions that effect a change of output status place the input links in idle status. This is sufficient to verify both parts of the property.

BIBLIOGRAPHY

1. Busacker, R. G. and Saaty, T. L., Finite Graphs and Networks. McGraw Hill, New York (1965).

2. Bohm, E. and Jacopini, G., Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules, Comm. ACM Vol. 9, No. 5 (May, 1966), pp. 366-371.

3. Clark, W. A., Macromodular Computer Systems, AFIPS Conference Proceedings, (SJCC) Vol. 30, Thompson Books, Washington, D. C. (1967) pp. 335-336.

4. Cooper, D. C., Computer Programs and Graph Transformations, Center for the Study of Information Processing, Carnegie Institute of Technology (1966).

5. Ershov, A. P., Operator Algorithms I, Problems of Cybernetics III, Pergamon Press (1962) pp. 697-763.

6. Estrin G. and Turn, R., Automatic Assignment of Computations in a Variable Structure Computer System, IEE Transactions on Electronic Computers, Vol EC-12, No. 5 (Dec. 1963), pp. 755-773.

7. Holt, A. W., Notes for Computer and Program Organization, Engineering Summer Conference, University of Michigan. (June 1966).

8. Ianov, Y. I., On the Logical Schemata of Algorithms, Problems of Cybernetics I, Pergamon Press (1960), pp. 75-127.

9. Karp, R. M., A Note on The Application of Graph Theory to Digital Computer Programming, Information and Control, Vol. I, (June, 1960), pp. 179-190.

10. Karp, R. M. and Miller, R. E., Properties of a Model for Parallel Computations: Determinary, Termination, Queueing, SIAM J. Appl. Math., Vol. 14, No. 6 (Nov. 1966), pp. 1390-1411.

11. Kaluzhnin, L. A., Algorithmization of Mathematical Problems, Problems of Cybernetics II, Pergamon Press (1961), pp. 371-391.

12. Malhotra, A., Asynchronous Control of Computer Operations, S.M. Thesis, Sloan School of Management, M.I.T. (Feb. 1967).

13. Marimout, R. B., Application of Graphs and Boolean Matrices to Computer Programming, SIAM Review,. Vol. 2, No. 4 (Oct. 1960) pp. 259-268.

14. Martin, D. F., The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems, Report No. 66-4, Department of Engineering, University of California, Los Angeles, (January 1966).

15. McCarthy, J., A Basis for a Mathematical Theory of Computation, Proc. Western Joint Computer Conference, Vol. 19, (1961), pp. 225-238.

16. McNaughton, R., Parenthesis Grammars, Journal ACM, Vol. 14, No. 3 (July 1967), pp. 490-500.

17. Muller, D. E. and Bartky W. S., A Theory of Asynchronous Circuits, Proc. of an International Symposium on The Theory of Switching, The Annals of the Computation Laboratory of Harvard University, Vol. 29, Part I, Harvard University Press (1959), pp. 204-243.

18. Ornstein, S. M., Stucki, M. J., and Clark, W. A., A Functional Description of Macromodules, AFIPS Conference Proceedings (SJCC) Vol. 30, Thompson Books, Washington, D.C. (1967), pp. 337-355.

19. Petri, C. A., Communication with Automata, Memorandum MAC-M-212, Project MAC, M.I.T., Translation of: Kommunikation mit Automaten, Institut fur Angewandte Mathematik der Universitat Bonn, Wegelerstrasse 10, Bonn (1962).

20. Prosser, R. T., Application of Boolean Matrices to the Analysis of Flow Diagrams, Proc. Eastern Joint Computer Conference Spartan Books (1959), pp. 133-138.

21. Rutledge, J. D., On Ianov's Program Schemata, Journal ACM, Vol. 11, No. 1 (Jan. 1964), pp. 1-9.

22. Stucki, M. J., Ornstein, S. M., and Clark, W. A., Logical Design of Macromodules, AFIPS Conference Proceedings (SJCC) Vol. 30, Thompson Books, Washington, D. C. (1967), pp. 357-363.

23. Van Horn, E. C., Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., (Sept. 1966). Also Report MAC-TR-34, Project MAC, M.I.T.

CS-TR Scanning Project
Document Control Form

Date : 2 / 2 / 96

Report # LCS-TR-64

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 130 (136-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other:

Page Data:

Blank Pages (by page number): Follows title page

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:
Image map: (1-130) un#'ed title & blank pages, iii-x, 1-120
(131-136) Scancontrol, cover, DOD, Trgts (3)

Scanning Agent Signoff:
Date Received: 2/2/96 Date Scanned: 2/2/96 Date Returned: 2/9/96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

Project MAC — Technical Report Abstract

1. ORIGINATING ACTIVITY Massachusetts Institute of Technology Project MAC	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
3. REPORT TITLE A Graph Model for Parallel Computations	
4. DESCRIPTIVE NOTES Technical Report(the unaltered MIT Doctor of Science thesis, submitted Sept. 1969)	
5. AUTHOR(S) Rodriguez, Jorge E.	
6. REPORT DATE September, 1969	7a. TOTAL NO. OF PAGES 133 7b. NO. OF REFS 23
8a. CONTRACT OR GRANT NO. Office of Naval Research, Nonr-4102(01) b. PROJECT NO. NR-048-189 c. RR 003-09-01	9a. ORIGINATOR'S REPORT NUMBER MAC TR-64 ESL-R-398 9b. OTHER REPORT NO. AD 697 759
10. AVAILABILITY/LIMITATION NOTICES Defense Contractors may obtain from: Defense Documentation Center, Defense Supply Agency, Cameron Station, Alexandria, VA 22314* Others from: Clearinghouse for Federal Scientific and Technical Information (CFSTI) Sills Building, 5285 Port Royal Road, Springfield, VA 22151	
11. SUPPLEMENTARY NOTES None Air Force Manufacturing Technology Laboratory, RTD Wright-Patterson AFB	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency, 3D-200 Pentagon Washington, D.C. 20301

13. ABSTRACT

This report presents a computational model called program graphs which makes possible a precise description of parallel computations of arbitrary complexity on non-structured data. In the model, the computation steps are represented by the nodes of a directed graph whose links represent the elements of storage and transmission of data and/or control information. The activation of the computation represented by a node depends only on the control information residing in each of the links incident into and out of the node. At any given time any number of nodes may be active, and there are no assumptions in the model regarding either the length of time required to perform the computation represented by a node or the length of time required to transmit data or control information from one node to another. Data dependent decisions are incorporated in the model in a novel way which makes a sharp distinction between the local sequencing requirements arising from the data dependency of the computation steps and the global sequencing requirements determined by the logical structure of the algorithm.

14. KEY WORDS

Program Graphs
Parallel Computations
Computation Models

*No copies are available from Project MAC.