MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Electronic Systems Laboratory
Cambridge, Massachusetts 02139

Memorandum 9442-M-162 | Machine Structures Group Memorandum No. 22 | Memorandum MAC-M-301

March 7, 1966

APPROVED: [illegible]

ANALYSIS AND TRANSFORMATION OF COMPUTATIONAL PROCESSES

by

Jorge E. Rodriguez

ABSTRACT

Computational processes are represented by a class of directed graphs called 'program graphs'. Parallel as well as sequential processes are naturally represented in terms of this model. A procedure is developed by which the cycles of a program graph are destroyed, yielding a cycle free graph and a description of the cycles of the graph, called the N-cycle structure. Sufficient conditions for a deterministic process are obtained in terms of the cycle-free graph and N-cycle structure obtained by this procedure. A notion of equivalence among program graphs is defined and some simple equivalence-preserving transformation are briefly considered.

This memorandum is based on a proposal for thesis research in partial fulfillment of the requirements for the degree of Doctor Science submitted in the Electrical Engineering Department on March 7, 1966.

TABLE OF CONTENTS

A. INTRODUCTION

In the solution of problems by means of a digital computer a principal activity is the design of programs and their encoding in a suitable programming language.

During the past years there has been a great deal of interest and research in the area of automatic programming. This interest has been centered in the design of programming languages and the construction of processors for automatic translation of programs written in these languages into programs in some target language suitable for execution by a given machine or interpreter.

As the complexity of problems whose solution is tackled increases with increased sophistication in the use of digital computers, so the size and complexity of the resulting programs grows accordingly. This has given rise to programming systems formed by sub-systems interacting more or less heavily upon each other.

In the absence of systematic procedures of program analysis it becomes very difficult to determine the nature of these interactions as well as their effect upon total system behavior. With increased use of multi-programming and the advent of multi-processor computing systems another dimension is yet added to the interactive behavior of programming subsystems as well as whole independent programming systems.

Thus there arises a growing need for systematic methods of program analysis and synthesis which can be profitably applied at the design stage in order to optimize in some sense the nature of the interactions among program and system components. Potentially such techniques can be profitably used at every level of the programming activities by the program designer, the program encoder, and the translator or compiler which can in turn supply relevant information relating to the particular characteristics of the program to the supervisory system in charge of administering the program execution and allotment of the computing system resources.

We approach the problem of developing systematic procedures of program analysis by first formulating a model for computer programs and then studying the properties of the model with the hope that we will gain an understanding of the properties of the actual programs. We require such a model to be sufficiently close to actual programs, as we know them, so as to make its construction natural, e.g., there should be a reasonably simple procedure for expressing an actual program in a standard programming language in terms of the model; this of course does not imply that the model be bound by current programming languages. Consequently such things as sequencing requirements, interdependence among computations and powerful control structure should be easily expressed in the model; we should also be able to express independence among computations in a simple way, a facility not present or available in a rudimentary form in current programming languages. On the other hand the model shall not be bound by any ad-hoc assumption on the structure of machines or particular implementations. We may say that the model represents a program in an ideal way, i.e., without ties to any specific form of implementation. The reason for this requirement is fairly clear, a great number of the possible variations of a program arise from different forms of implementation such as choice of data structures, choice of what to put in the data structure and what to compute when needed, etc. These different implementations can be compared only if their effect shows explicitly in terms of structural changes in their respective model representations and this could hardly be done if the differences are hidden behind some implicit behavioral property.

From the foregoing it is clear that the developments in the mathematical theory of computability such as Turing machines, Markov algorithms and recursive function theory are too crude a representation of programs (when so viewed) to be useful in applications to actual programs. Similar remarks can be applied to the developments of mathematical machines theory McCarthy's [17,18] work is a first step towards the goal of studying program equivalence and program transformation.

The 'Applicative expressions' of Landin[10] based on the λ-calculus constitute a model in which program behavior is well defined only by the syntactic properties of the expressions involved. Landin's intent is to provide a vehicle suitable for formalizing the sematics of programming languages. In subsequent work[11] he proceeds to express Algol-60 in terms of an elaboration of the original applicative expressions.

Ianov[5,6,7] has studied the sequential and logical structure of what he calls 'program schemata'. In his work he shows that the equivalence problem for 'program schemata' is solvable and gives algorithms for reducing them to a canonical form as well as a complete sets of rules for transforming a scheme into any of its equivalent schemata. Ianov's schemata are very simplified models for actual programs, and his definition of equivalence among schemata turns out to be much stronger than one would desire in that many programs that we would like to consider equivalent, are not upon representing them as program shcemata. In a sequel, Rutledge[21] has shown the equivalence between Ianov's schemata and finite automata. Luckham,[12] Luckham and Marill[13] and Marill[15] have given a model for programs using a basic language together with interpretation rules. A form of equivalence is defined and it is shown by Luckham and Park [14] that the equivalence problem is undecidable even for this model. Rules or transformation with the intent of simplifying the programs have been given by Marill for the case where there are no 'branches' and by Luckham for programs involving non-intersecting loops.

The type of simplifications considered by Marill are the removal of 'vacuous statements' (e.g., those statements whose presence or absence do not affect the program) and the reassignment of variable names so as to minimize the number of names (and consequently storage locations needed. Luckham is concerned with removal of 'vacuous statements' as well as 'loop vacuous statements' from within loops (e.g., those statements inside a loop which only depend on statements outside the loop or which only influence statements outside the loop).

Both Ianov and Luckham, et. al., disregard the interim structure of the operators or functions used to represent the computations. McCarthy on the other hand relies on the known properties of functions in order to show equivalence by means of the recursion induction principle. A similar approach is taken by Cooper. [2]

Prosser [20] and Karp [8] have used a flow chart representation of programs and developed techniques for analysis of the structures of these flow charts.

B. SUMMARY

Our aims are to develop methods of program analysis and subsequently develop rules which can be applied to a given program so as to obtain another program which is in some sense equivalent to the original.

We start by presenting a model suitable for expressing a large class of programs and giving a set of rules which specify the manner in which these programs are executed. Since we are primarily interested in the behavioral equivalence of programs it is necessary to define a suitable criterion for comparing the behavior of two programs or the same program under different conditions; once such a criterion is obtained it becomes clear that two executions of the same program may not show the same behavior due to the possibility of concurrent computations during the execution of the program as represented by the model as well as the freedom to express through it all kinds of 'nonsense' programs. The concept of a deterministic program is precisely defined and analysis tools are presented which allow us to establish sufficient conditions for a program to be deterministic.

We subsequently consider the problem of obtaining rules for equivalence transformations and establish the validity of some relatively simple transformations.

The proposal concludes by indicating directions of further investigation.

C. THE MODEL

We will represent a program by means of a class of directed graphs which we will call program graphs.

We will distingush 3 types of nodes and 2 types of links in a program graph. Nodes may be operators, selectors or junctions; links may be control links or data links.*

* Nodes and links correspond to the vertices and branches in the literature of graph theory. We prefer the name link since a 'branch' has an established meaning in the programming field. We reserve the words vertex and branch for the definition of dynamic ancestry tree later on.

An operator is a node which has two or more input connectors and one or more output connectors. The set of input connectors of an operator shall be considered to be ordered so that we may uniquely identify a connector by its ordinal number in the set. Similarly for the set of output connectors. Operators will be represented in a program graph by a circle: (O)

A selector is a node which has two or more input connectors and precisely 2 output connectors. The sets of input connectors and the set of output connectors of a selector are also ordered sets. In particular one output connector of a selector will be labeled + and the other will be labeled -. Selectors will be represented in a program graph by a diamond-shaped symbol. (◇) Operators and selectors have a zeroth input connector which is distinguished in that it can only be the tip of a control link, furthermore it is not required for such a link to exist at all. All zeroth input connectors which remain unlinked are called free zeroth connectors.

A junction is a node which has two or more input connectors and one output connector. The set of input connectors of a junction is not ordered. Junctions will be represented in a program graph by a rectangle. (□)

A link is a directed line segment having a root and a tip. The direction specified in a link by means of an arrowhead is from the root to the tip.

In a program graph a link always connects an output connector of a node to an input connector of some other node. That is to say that the root of a link lies at some output connector while its tip lies at some input connector.

A control link is one which is rooted at an output connector of a selector or at the output connector of a junction whose input connectors are tips of control links.

A data link is one which is rooted at an output connector of an operator or at the output connector of a junction whose input connectors are tips of data links.

A program graph is a finite set of operators, selectors and junctions interconnected by means of control and data links according to the following rules:

a. Every input connector of an operator or selector must be the tip of one data link except for the zeroth connector which may only be the tip of one control link.

b. Every input connector of a junction is the tip of one data link or one control link. However, for any given junction all input connectors must be tips of the same type of link. We shall accordingly distinguish between data junctions and control junctions.

c. Any output connector may be the root of any number of links.

For identification purposes we shall associate names with nodes as follows:

Operators will be denoted by subscripted 'f' i.e., f_1, f_2.

Selectors will be denoted by subscripted 'β' i.e., β_1, β_2.

Junctions will be denoted by subscripted 'g' i.e., g_1, g_2.

Input connectors will be denoted by a double subscript, thus the third input connector of f_1 is $f_{1,3}$.

Output connectors of selectors we denote by superscripting the selector name with + or - i.e., β_2^{+}.

We shall now give an interpretation for the elements of a program graph together with a set of rules which impose a well-defined dynamic behavior upon the graph interpretation. We shall require every program graph to contain a set of input terminals and a set of output terminals such that the input terminals are roots of data links but have no ancestors, while the output terminals are tips of data or control links but are roots of no links.

The following graphs are examples of program graphs:

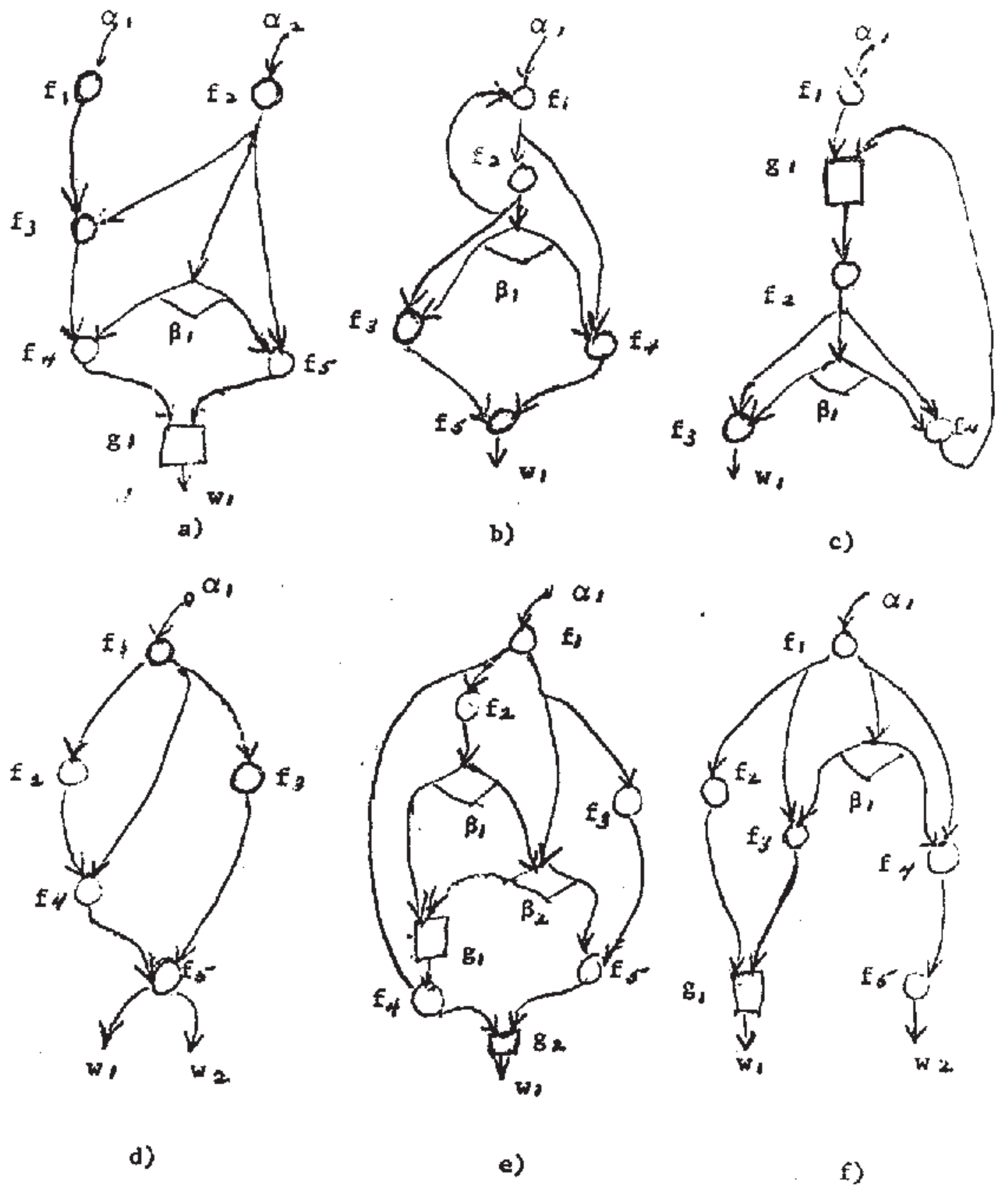


Figure 1. Examples of Program Graphs

We associate with input terminals, output terminals and output connectors of nodes the property value, and with input connectors of nodes we associate the property status. Links are capable of transmitting the value of a terminal or output connector to the input connectors at their tips.

The status of an input connector may assume 2 possible conditions which shall be denoted 0 and 1. Operators and nodes are interpreted as functions which are applied to the values associated with input connectors 1 through n. The process of applying an operator or selector yields a set of new values to be associated with the output connectors of the operator or selector. The application of an operator or selector is determined by the status of its input connectors as follows:

An operator or selector is applied only if all its input connectors are in status 1.

When this condition is met, we shall say that event A has occurred for the operator or selector. The effect of event A in addition to application of the function is to place all input connectors in status 0, except the zeroth connector (control link) which is placed in status 0 if a link to it exists and left undisturbed otherwise.

Event B occurs when the application of the operator or selector has yielded a new set of values. Event B has the additional effect of placing in status 1 all input connectors directly linked to output connectors in the case of an operator while in the case of a selector only those input connectors directly linked to one of the two output connectors are placed in status 1.

A junction is applied (event A) whenever one or more of its input connectors are in status 1. Event A for a junction has the effect of placing all input connectors in status 0. The application of a junction consists in associating with its output connector upon occurrence of event B the value of one of the input connectors in status 1 at the time of application. In addition all input connectors linked to the junction are placed in status 1.

Under the interpretation, events may occur only at discrete time intervals as determined by an independent clock. We shall associate with each operator, selector and junction an integer $t_i \geq 1$ specifying the number of intervals elapsed between the occurrence of event A and the occurrence of event B for the node.

A program graph is executed as follows:

a. Initially all input connectors are placed in status 0 except for free zeroth connectors of operators and delectors which are placed in status 1.

b. Arbitrary values are placed at the input terminals and all input connectors linked to these terminals are placed in status 1.

c. The clock is started.

The execution terminates when event A has occurred for all data output terminals and some control terminal.

Let $f_1 \ldots f_r$ be the names associated with the operators of a program graph, and $\beta_1 \ldots \beta_q$ the names associated with its selectors $\alpha_1 \ldots \alpha_p$ the names of the input terminals and $w_1 \ldots w_s$ the names of the output terminals. No two nodes have the same name. The execution sequence associated with the execution of the program graph is a string defined as follows:

At time 0 write down, in any order, the names of the input terminals of the graph.

At time i, write to the right of the string in any order the names associated with all operators and selectors for which event B occurred at time i and the names associated with output terminals for which event A occurred.

Example 1

Consider the program graph in Figure 2.

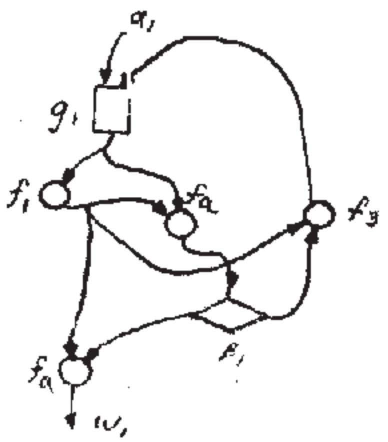


Figure 2. Program Graph used in Example 1

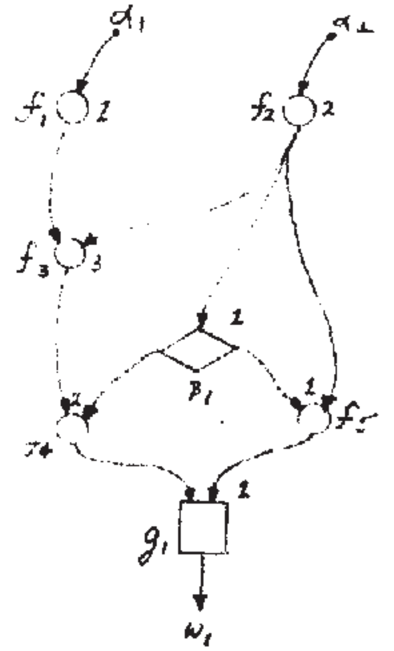
These are possible execution sequences for this graph

1) $\alpha_1 f_1 f_2 \beta_1 f_3 f_1 f_2 \beta_1 f_4 w_1$

2) $\alpha_1 f_1 f_2 \beta_1 f_4 w_1$

We illustrate the interpretation rules by applying them to the examples of Figure 1.

Consider the program graph of Figure 1a.



Numbers next to the nodes denote the time intervals which elapse between occurrence of events A and B for the node.

Figure 1a. (Repeated)

There are free zeroth input connectors in f_1, f_2, f_3 and β_1, these connectors are placed in status 1 and will remain in that status for all time. Initially the values of α_1,α_2 (inputs) are established and the input connectors of f_1 and f_2 are placed in status 1, all other input connectors are in status 0. We assume that no new α_1 and α_2 values are established until after execution terminates. The clock is started.

At the first clock event A occurs for both f_1 and f_2, the input values for f_1 and f_2 are obtained. The corresponding input connectors are placed in status 0 both f_1 and f_2 are applied. At the next clock, event B occurs for f_1 placing input connector 1 of f_3 in status 1 and making its output value

Available. At clock 3 event B occurs for f_2, placing input connector 2 of f_3, input connector 1 of β_1 and f_5 in status 1. At clock 4 event A occurs for both f_3 and β_1 and they are applied. Event B for β_1 occurs at the next clock. Since β_1 is a selector a value appears only at one output connector, say it is the + connector placing the zeroth connector of f_4 in status 1. Note that the zeroth connector of f_5 remains in status 0 and f_5 will never be applied during this execution. Execution continues by application of f_4 and later g_1 finally causing event A to occur for w_1 and thus terminating the process.

Let us now consider briefly the remaining program graphs of Figure 1.

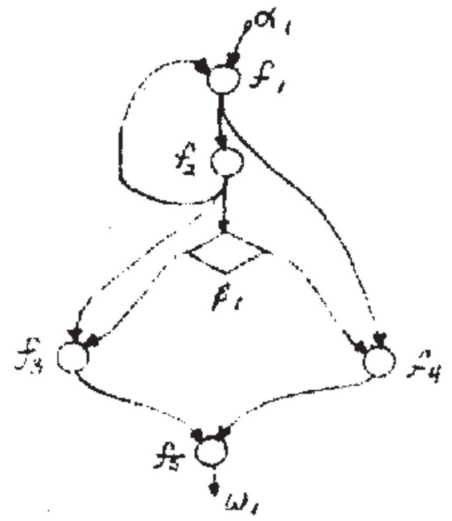


Figure 1b. (Repeated)

The graph of Figure 1b is peculiar on two counts. First, there is a sequence of data links starting at an output connector of f_1 and terminating at an input connector of the same node. Clearly f_1 will never be applied but f_1 cannot be applied until all its input connectors are in status 1. In this particular case no other node will ever be applied either. We may say that f_1 is 'hung-up'.

A similar situation shows for f_5 (independently of f_1) but for a different reason. There since selector β_1 produces a value at one output connector or the other, either f_3 or f_4 may be applied but not both, consequently f_5 which is linked to both f_3 and f_4 will never be applied.

Clearly in our study of program graphs we like to eliminate these pathological cases.

The program graph of Figure 1c shows a sequence of links starting and ending at a junction. This is a satisfactory situation since a junction needs only one of its input connectors in status 1 in order to be applied.

The xamples of Fig. 1d and 1e show program graphs without selectors and with both control and data junctions respectively. The graph of Fig. 1f illustrates another type of pathological condition. In this case function g_1 is so conditioned that for the time intervals assigned to f_2, f_3 and β_1, both input connectors of g_1 will be in status 1 at the same clock, consequently upon application of the junction we are faced with the dilemma of picking one of the input values as the output value of the junction. This is a simple example of a non-deterministic program.

The dynamic ancestry tree* of an instance of an operator, selector or output terminal in an execution sequence of a program graph is constructed as follows:

a. Label the lowest vertex of the tree with the name of the chosen instance of operator, selector or output terminal, say h.

b. Draw up from this vertex as many branches as there are input connectors of h excluding the zeroth connector.

c. Label these branches as follows:

   Let branch i correspond to input connector i of h,

   1. If branch i is directly linked to an input terminal α_j label it α_j^i

   2. If branch i is directly linked to output connector k of operator i_j, label it $f^i_{j,k}$

   3. If branch i is directly linked to a junction construct the set of immediate ancestors of g_i in the following manner: all operators, connectors, selector connectors and input terminals directly linked to g_i are immediate ancestors of g_i. All operator connectors, selector connectors and input terminals which are immediate ancestors of a junction which is a direct ancestor of g_i are immediate ancestors of g_i. In order to label branch i, scan the execution sequence from right to left starting at h until an input terminal α_j or an operator or selector f_j with output connector $f_{j,k}$, is found such that α_j or $f_{j,k}$ are immediate ancestors of g_i. Label the branch α^i_j or $f^i_{j,k}$ accordingly.

d. If the label of a branch corresponds to an operator or selector output connector $f_{j,k}$, append at the end point of the branch the dynamic ancestry tree of the first instance of f_j found by scanning the execution sequence from right to left starting at h.

---

* Luckham[10] introduced a very similar concept which he calls an 'Ancestry Tree'.

Example

The dynamic ancestry trees of ω_1 in the sequences of the previous example are shown in Figure 6.

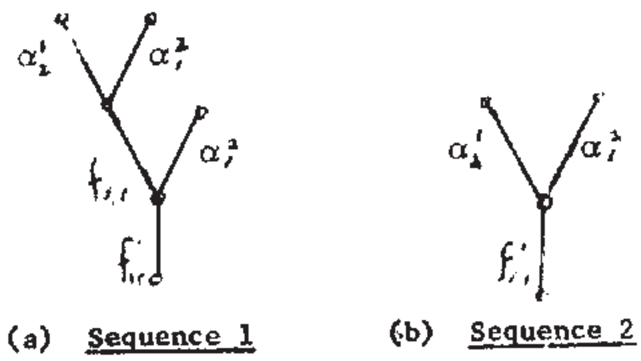


Figure 6. Dynamic Ancestry Trees for Sequences of Example 1

In the construction of programs it is useful to introduce an identity operator i.e., an operator having one input connector (in addition to the zeroth one) and one output connector and such that its application results in associating with the output connector the same value associated with the input connector. We shall denote the identity operator by I.

A minimal dynamic ancestry tree is a dynamic ancestry tree which does not contain any branches labelded with the name of the identity operator.

Two dynamic ancestry trees T1 and T2 are isomorphic if there is a one to one correspondence between the vertices of T1 and T2 which preserves the incidence relations and all branches so paired have the same label.

Two dynamic ancestry tree for an instance of a node in an execution sequence corresponding to a program graph P represents the precise sequence of operators and /or selectors applied to the input values of the program graph which yield the values associated with the inputs used in the chosen application of the node.

Under the assumption that all applications of a node yield the same output values whenever the values at the input connectors are the same, the concept of congruent dynamic ancestry trees provides an effective test for the equality of any value arising during the execution of a program graph.

We formalize this observation by means of Theorem 1 and a definition. An operator (selector)is <u>deterministic</u> if every application of the operator (selector) to the same set of input values yields the same set of output values.

<u>Theorem 1</u>:

Let T1 and T2 be two congruent dynamic ancestry trees derived from execution sequences generated by a program graph P whose operators and selectors are deterministic. If the values associated with the input terminals at the endpoints of the tree are the same for T1 and T2, then the output value associated with the instance of the operator, selector or output terminal at the root of the tree are the same for T1 and T2.

<u>Proof</u>:

Take the dynamic ancestry trees of any output terminal for any two execution sequences, the fact that they are congruent implies that we can place the minimal trees side by side, and establish n levels where n is the length of the longest path such that at level 0 is the set of vertices corresponding to input values, at level i+1 is the set of vertices having at least one incident branch connected to one vertex in level i-1. Because the trees are congruent for every vertex at level i in one of the trees, there is a corresponding vertex in the other tree such that the labels of all branches from vertices in level $k < i$ to the specified vertex are the same for both trees. This in turn implies that the labels of the branch from this vertex to a vertex in level $j > i$ are the same in both trees. Since vertices at level $0 < i < n$ represent operators or selectors, we can establish the identity of input values for the application of an instance of an operator or selector by induction as follows: At level 0, all input values are the same by hypothesis. Thus the input values to operators and selectors represented by vertices at level 1 are the same for both trees.

Consequently, since these vertices represent deterministic nodes, the output values of such operators or selectors are the same and the input values for the vertices in level 2 are identical for both trees. Now er repeat the same argument until all levels are exhausted and conclude that all input values to the only vertex at level n are the same for both trees and consequently both output values are identical. Q.E.D.

A program graph is deterministic if any two applications of the graph to the same set of input values yields execution sequences such that the dynamic ancestry trees of all output terminals are congruent for all time interval assignments to nodes of the program graph.

We start the development of the analysis of program graphs by defining the class of totally cyclic consistent graphs by eliminating all program graphs which shows some undesirable pathological condition in the formation of the graph cycles. For this purpose we develop the concept of N-cycle, N-cycle decomposition and N-cycle structure.

We then proceed to establish sufficient conditions for a cycle free program graph to be deterministic (theorem 5), and reduce the problem of determining sufficient conditions for an arbitrary program graph to conditions on a particular cycle free graph and the N-cycle structure associated with the given program graph (theorem 6).

We establish some terminology by the following definitions where nodes will be denoted by $\alpha_1, \alpha_2, \ldots$ and links by $\ell_1, \ell 2, \ldots$

A set of links $\ell_{k_1}, \ell_{k_2}, \ldots \ell_{k_m}$ is a path of a graph P if there are nodes $\alpha_{k_1}, \alpha_{k_2}, \ldots \alpha_{k_m}, \alpha_{k_{m+1}}$ in P such that ℓ_{k_1} is a link from an output connector of α_{k_i} to an input connector of $\alpha_{k_{i+1}}$. We say that the path $\ell_{k_1}, \ldots \ell_{k_m}$ passes through the nodes $\alpha_{k_1} \ldots \alpha_{k_{m+1}}$, or that there is a path from α_{k_1} to $\alpha_{k_{m+1}}$. If $\alpha_{k_1} = \alpha_{k_{m+1}}$ we say that the path is a cycle.

A data path (cycle) from α_{k_1} to α_{k_m} is a path (cycle) consisting of data links only.

A control path (cycle) from a_{k_1} to a_{k_m} is a path (cycle) consisting of control links only.

A mixed path (cycle) is one containing both data and control links.

A path P_1 is contained within a path P_2 if all links of P_1 are also links of P_2.

A graph is cycle free if it does not have any cycles.

An N-cycle* K is a set of nodes $a_1, a_2 \ldots a_m$ such that for all $i, j \leq m$ if $a_i, a_j \in K$, then there is a cycle of the graph which passes through a_i and a_j. The order of an N-cycle K is the number of nodes contained in K.

A maximal N-cycle of a given graph is an N-cycle K such that a_i K, $a_j \notin K$ implies that no cycle of the graph passes through a_i and a_j.

Two N-cycles are equal if they contain the same nodes.

Two N-cycles K_1, K_2 are disjoint if no node a_1 is in both K_1 and K_2.

Theorem 2:

Let K_1, K_2 be N-cycles of a graph. Then if $K_1 \neq K_2$ and $K_1 \cap K_2 \neq \emptyset$ there exists another N-cycle K_{12} which properly contains both K_1 and K_2.

Proof:

Let $a_i \in K_1$, $a_j \in K_2$ and $a_k \in K_1 \cap K_2$.

By the definition of N-cycle, there is a cycle of P passing through a_j and a_k and some other cycle passing through a_i and a_k. Therefore the path passing through $(a_i\ a_k\ a_j\ a_k\ a_i)$ is a cycle through a_i and a_j. Using a similar reasoning it follows that there is a cycle passing through $a_m \in K_1$, $a_n \in K_2$ for all m and n and therefore the sets of nodes $K_1 \cup K_2$ is also an N-cycle. Q.E.D.

* The term N-cycle has been used by Simões-Pereira.[23] His definition corresponds to our 'maximal N-cycle'.

It follows from Theorem 2 that a graph P can be <u>uniquely decomposed</u> in disjoint maximal N-cycles and a set of nodes through which no cycle of the graph passes.*

In the sequel, N-cycle shall always denote maximal N-cycle.

A node a_i is an <u>ancestor</u> of a_j if there is a path from a_i to a_j. If the path consists of exactly one link, then a_i is a <u>direct ancestor</u> of a_k.

A program graph is <u>cyclic consistent</u> if for every N-cycle K there is at least one data junction $g_i \in K$ which has one or more direct ancestors not contained in K and no node of K which is not a data junction has a direct ancestor not contained in K. These junctions will be called the <u>loop junctions</u> generated by K. Intuituvely, the conditions for a cyclic consistent program graph imply that every data cycle of the graph passes through a junction which is not selely dependent upon itself. In programming terms this is equivalent to require every loop in the program to be 'initialized'. The other implication of the cyclic consistency conditions is that a value which remains unaltered by any node of an N-cycle and is used repetitively must show as such explicitly in the program graph. This property is useful in obtaining simplifying transformations which 'clean up' the loops of a program among other things. Its main reason for existence however, is the simplification of the interpretation rules of the model.

<u>Theorem 3:</u>

Let P be a program graph, $K_1, K_2, \ldots K_n$ its N-cycles and $G_1, G_2, \ldots G_n$ the corresponding sets of generated loop junctions.

Let P^1 be the program graph obtained by deleting from P all links (data links) from direct ancestors of $g \in G_1$ contained in K_1 and connecting these links to a new junction g'. The newly created junction does not have any links rooted at its output connector. We will say that g and g' are

* This result is the same as the uniqueness of the decomposition of a separable graph into its components applied to directed graphs. See for example Seshu and Reed Th. 3-6 and 3-7, page 38.

conjugate. Then

$$P^1 \text{ is cycle free}$$

or the following is true.

Let the N-cycles of P^1 be $K_1^{\ 1}, K_2^{\ 1}, \ldots K_g^{\ 1}$ then every N-cycle of P^1 is properly contained within some N-cycle of P.

i.e., for every i there exists a j such that $K_k^{\ 1} \subset K_j$ and for every N-cycle of K_j, of P, the union of N-cycles of P^1 contained in K_j is properly contained in K_j.

i.e., for every j

$\cup K_i^{\ 1} \subset K_j$ where the union is taken over all i such that $K_i^{\ 1} \subset K_j$.

<u>Proof:</u>

Since all links from $a_j \in K_i$ to $g_i \in K_i$ have been removed, there is no path in P from a $\in K_i$ to g_i, thus no cycle of P^1 passes through g_i, the number of cycles of P^1 is less than the number of cycles of P, and P^1 may be cycle free.

If P^1 is not cycle free, $K_i^{\ 1} \subset K_j$ for some j since no new links have been added and thus every cycle appearing in P^1 also appears in P. The inclusion is proper because at least one node contained in K_j is not in $K_i^{\ 1}$ namely the junction g_i.

The second part of the theorem follows from the fact that all N-cycles of P^1 are disjoint. Q.E.D.

<u>Corollary 1.1</u>

The sum of the orders of the N-cycles of P^1 is less than the sum of the orders of the N-cycles of P.

Let P and P^1 be as in <u>Theorem 3</u>. Total <u>cyclic consistency</u> is defined recursively as follows:

P is <u>totally cyclic consistent</u> if P is cyclic consistent and P^1 is cycle free or <u>totally cyclic consistent</u>.

The definition is effective since there is a finite number of nodes and at every step of the process the sum of the orders of the resulting N-cycles is less than the sum of the orders of the original N-cycl.

The process of obtaining the N-cycles of a program graph by successive application of the procedure of Theorem 3 will be called N-cycle decomposition. The series of N-cycles obtained by N-cycle decomposition will be called the N-cycle structure of the program graph.

The N-Cycle can be represented by a forest, where the N-cycles of P constitute the root of the trees in the forest and the N-cycles of P1 are the nodes of branches emanating of the roots, similarly for P^2, ect.

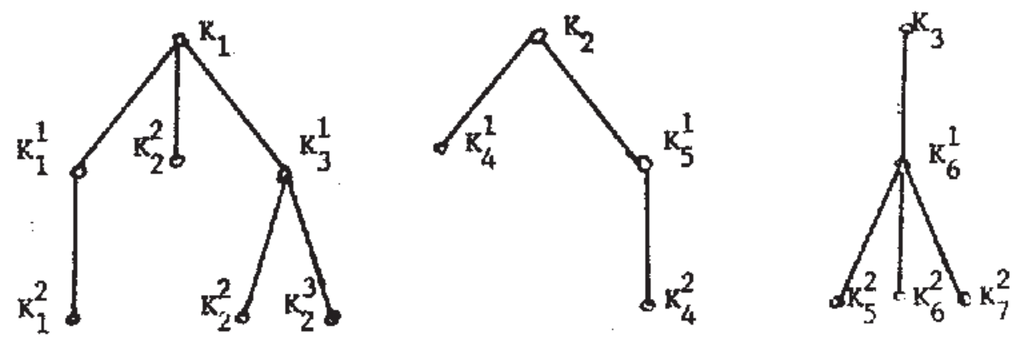
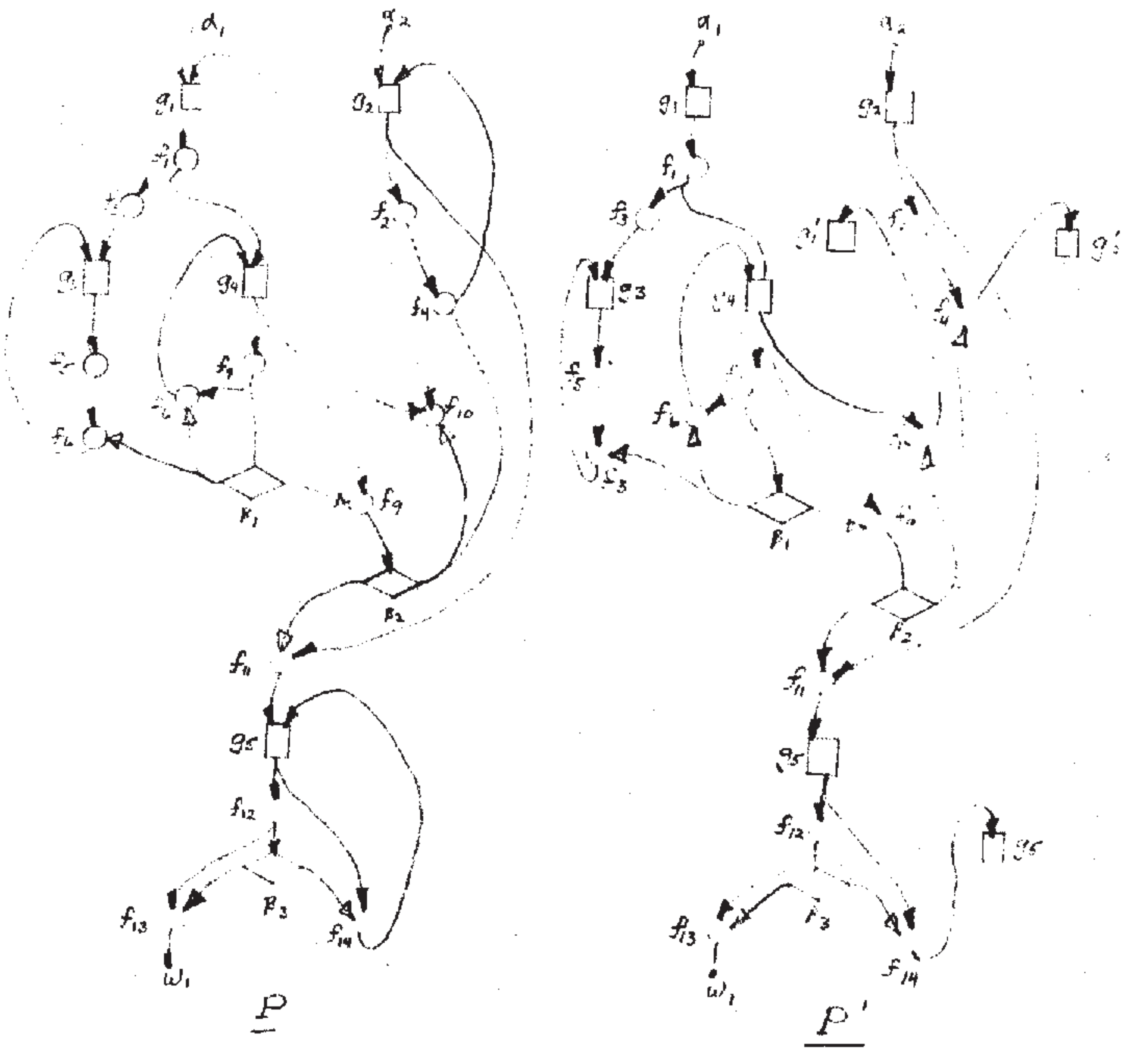


Figure 7. Example of an N-cycle Structure

Example

We illustrate the N-cycle decomposition with the following example:



$K_1 = \{g_1, g_2, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, g_3, g_4, \beta_1, \beta_2\}$

$K_2 = \{g_5, f_{12}, f_{13}, f_{14}, \beta_3\}$ $\qquad$ $K_1' = \{g_3, g_4, f_6, f_7, f_8, \beta_1\}$

Figure 8. Example of N-cycle Decomposition

Theorem 4:

Every cycle of a totally cyclic consistent program graph passes through a loop junction.

This is just a re-statement of total cyclic consistency since by definition all cycles of such a graph can be removed by deleting selected links to input connectors of loop junctions which implies that such links are in the cycles and therefore the cycles pass through the loop junction.

Let P be a totally cyclic consistent graph, $P = P^0\ P^1\ P^2\ \ldots\ P^r$ the sequence of graphs generated by the N-cycle decomposition of P. P^r is the cycle free graph generated by P.

So far we have dealt with nodes of program graph and established some concepts involving nodes and paths through them. Now we turn our attention to the connectors of the nodes inasmuch as our main concern is to study the values that arise during the execution of a program graph, and such values reside or appear at the output and input connectors of nodes.

In particular we will consider input connectors for, as we shall see, the properties of output connectors relevant to our program graph analysis can be deduced from the properties of the input connectors of the corresponding node. Let c_1 and c_2 denote any two connectors of a program graph.

c_1 is an a-ancestor of c_2 if there is a path from c_1 to c_2 which does not pass through any junction.

c_1 is exclusive with c_2 if any of the following conditions are met.

a. c_1 is the '+' output connector of a selector β and c_2 is the '-' output connector of β.

b. There exists at least two exclusive connectors c_1' and c_2' in P such that c_1' and c_2' are a-ancestors of c_1 and c_2 respectively.

c. c_1 and c_2 are the output connectors of junctions g_1 and g_2 in P such that every input connector of g_1 is exclusive with all input connectors of g_2.

Note that the given definition of exclusive connection is recursive. The following lemma guarantees that the definition is effective for the class of cycle free program graphs.

<u>Lemma 5.1</u> If P is a cycle free program graph it is always possible to determine whether or not any two connectors of P are exclusive.

<u>Proof Omitted</u>

Roughly speaking, the concept of exclusive connectors means that values cannot be 'simultaneously' available at both connectors during execution of a program graph. We formalize this notion by means of the following lemma.

<u>Lemma 5.2</u> If c_1 and c_2 are exclusive connectors of a program graph P, then there is no execution of P during which values are associated with both c_1 and c_2.

<u>Proof Omitted</u>

The following definition will be useful in the subsequent theorems:

A set of connectors C is <u>mutually exclusive if for all</u> $c_i, c_j \in C$, c_i is exclusive with c_j.

Theorem 5:

A cycle free program graph is deterministic if

a. P does not have any junctions.

or b. P has junctions and for every junction the set of its input connectors is mutually exclusive.

<u>Proof</u>:

Assign a unique name to every node of P. Place all nodes P in levels as follows:

Level 0 contains the input terminals

Level i contains all nodes such that every input connector of the node linked to nodes in levels $j < i$ and at least one link comes from a node in level i-1.

The number of levels, n, is the length of the longest path of P.

Assume P does not have any junctions, then every node of P appears at most once in any execution sequence of P if the nodes at level 0 (input terminals) appear only once.

This is clear for the input connectors of nodes at level 1 are placed in status 1, and thus the nodes applied, only once; this in turn implies that the input connector of nodes at level 2 satisfy the same condition and so on. If a node at level $i > 0$ is a selector, these nodes at level $j > i$ connected to one of the outputs of the selector will not be applied.

Since every input connector of a node has precisely one link directed into it, the name of the node which is the root of this link can be uniquely identified in any execution sequence and consequently the dynamic ancestry tree for any node, including the output terminals, are isomorphic for all execution sequences and P is deterministic.
Assume P has junctions. We show that condition (b) is sufficient for P to be deterministic. Since all junctions have mutually exclusive sets of input connectors, no junction is directly linked to an input terminal. There is a set of junctions G^o such that no $g \in G^o$ has a junction as one of its ancestors. Therefore we may consider the sub-graph consisting of those operators and selectors which are ancestors of members of G^o as a program graph whose output germinals are connected to the input connectors of the junctions in G^o.

By part (a) this sub-graph is deterministic. By Theorem 1, for a fixed set of input values, the values available at the input connectors of the junctions in G^o is the same for all executions of the sub-graph.

Because the junctions in G^o do not have any function as an ancestor, if input connectors c_1 and c_2 of $g \in G^o$ are exclusive for any given execution either c_1 or c_2 or neither may be placed in status 1. If any of c_1 or c_2 are so activated, it occurs only once by part a. Since by hypothesis the set of input connectors of all $g \in G^o$ is mutually exclusive, at most one input connector of g may be placed in status 1 and the value of the junction is deterministic. Now we repeat the same argument as above, using the set of junctions G^1 whose only junction ancestors are members of G^o.

In order to show the validity of this iterative step, we note that since for a fixed set of input values the same input connector of g G^o will always be placed in status 1, the direct ancestors of all other input connectors of g will never appear in any of the corresponding execution sequences and therefore do not appear in any dynamic ancestry tree. Consequently as far as the member of G^1 are concerned, all links from such nodes to g may be deleted and g nodes replaced by one <u>identity</u> operator. With this construction G^1 satisfies the same conditions as G^o. Q.E.D.

Throughout the proof of Theorem 5 we assumed that every operator and selector of the program graph had a unique name associated with it. This was mainly for convenience. The proof applies with minor changes whenever there appear duplicate names provided the ancestries of any two nodes with the same name are congruent (in addition to being of the same type).

<u>Theorem 6</u>:

Let P be a totally consistent program graph P^r the cycle free graph generated by P, K the N-cycle structure obtained by the N-cycle decomposition of P and let K be any N-cycle in K.

P is deterministic if

a. P^r is deterministic

b. Let g be a loop junction of P generated by the N-cycle K, g^1 its conjugate in P^r and α any node of P contained in K (thus α cannot be the conjugate of a loop junction). If there is a path from g to α in P^r, then there is a path from α to g^1 in P^r.

c. Let α_1, α_2 be any two nodes in P such that α_1 K, α_2 ' K, if there is a path in P^r from α_1 to α_2 then some input connector of α_2 is exclusive with all input connectors of the conjugate of the loop junctions of K.

We first show that (a), (b) and (c) taken together are sufficient to render a program graph deterministic. Clearly we are only concerned with program graphs which contain cycles for otherwise (b) and (c) do not apply and (a) says nothing.

So we assume that P contains some cycle. Condition (a) guarantees that for all possible time intervals assignment the values in the forward and 'feedback' portions of a cycle are the same for all executions. The fact that P has cycles opens up the possibility of a given operator or selector appearing more than once in an execution sequence even with condition (a) satisfied. If P is not properly conditioned, there may be time intervals assignments to the nodes of the graph for which the number of occurrences of events A (start of a process) and B (termination of a process) differs from one assignment to the other. Clearly if this occurs the corresponding dynamic ancestry trees of some results will not be congruent.

Conditions (b) and (c) guarantee that this situation can never occur. Condition (b) imposes requirements on nodes contained within some N-cycle while condition (c) constrains nodes outside every N-cycle.

Note that the basic problem is to prevent the occurrence of event A at a node while the node is being applied to a previous set of input values (i.e., before the occurrence of event B corresponding to a previous event A)

If condition (b) is satisfied, the re-occurrence of event A for a loop junction g implies that event B has occurred for all nodes of the associated N-cycle which have the loop junction as an ancestor (and for which event A had occurred). The re-occurrence of event A for g in P is the same as the occurrence of even t A for g^1 in P^r. Since P^r is deterministic the sub-trees of dynamic ancestry trees corresponding to every repetition of the cycle are congruent. For nodes a outside an N-cycle K, condition (c) guarantees that if there is a path from a node in K to a, then some input connector of a is exclusive with all input connectors of the conjugate of the loop junctions of K in P^r. Consequently the occurrence of event A for precludes the occurrence of such an event for g^1 which is sufficient. Q.E.D.

The converse of theorems 5 and 6 could be obtained by requiring certain auxiliary conditions to hold on a program graph; roughly speaking what is needed is to guarantee that all possible combinations of paths be allowed to occur; this seems to be a rather strong requirement since often this is not the case; we offer two typical situations. For specific operators and selectors in a program graph there exists some known relationships among them with respect to some body of data which precludes some combinations from occurring under all circumstances. As a consequence some connectors which are not exclusive on the light of topological conditions alone become exclusive when these relationships (which acrue from detailed knowledge of what the specific operators and selectors 'mean') are also considered. It transpires from the previous remark that Theorems 5 and 6 carry naturally to this situation by appropriately re-defining the concept of exclusive connectors.

Another situation of interest is the case where the assignment of time intervals to the nodes of the graph is not completely arbitrary (as it is assumed in Theorems 5 and 6) but either they are completely known or can be suitably bound for all pllications of interest. For example we may know that operator 'add.two' always takes 1 time interval, operator 'add. matrix' may require 1, 4, 9,...i^2 time intervals, operator 'do.this' may require time intervals bound by a double series like 1 to 3, 4 to 6, i^2 to $i^2 + 2$, etc. When such knowledge of actual time interval assignments is available, it may be possible to relax the conditions of Theorems 5 and 6.

We now turn our attention to the problem of obtaining program graphs which are equivalent to a given program graph. We assume hereafter that the given program graph is deterministic.

We start out by defining one form of equivalence among program graphs, which we shall call 1-equivalence as follows:

Let P_1 and P_2 be program graphs. Let $\omega_1^2 \ldots \omega_r^2$ the output terminals of P_2.

P_1 is 1=equivalent to P_2 if for every execution sequence, S_1 of P_1 there is some execution sequence, S_2. of P_2. such that the dynamic ancestry tree of ω_i^1 generated by S_1 is congruent with the dynamic ancestry tree of ω_i^2 generated by S_2 and vice versa.

In order to make this definition effective, we must be assured that all data output terminals names and one control terminal name appear precisely once in all terminating execution sequence of P^1 and P^2. For this purpose we restrict the class of deterministic program graphs and considered only <u>well-formed</u> graphs which are defined as follows:

The deterministic program graph P is <u>well formed</u> if the following conditions are satisfied.

a. No output terminal is contained within an N-cycle of P.

b. No data output terminal is exclusive with any output connector of any selector.

c. The set of control output terminals is mutually exclusive: furthermore if this set is considered as the set of output connectors of a junction g., then the output connector of g satisfies condition (b).

There are several types of equivalence preserving transformations which may be considered of interest, some of them will be the subject of future work, in this section we indicate the nature of one of the simplest type of transformations, namely those modifications of a well formed programed graph which add and/or delete control links, control junctions and identity operators, resulting in another well formed program graph. In general, we are interested in discovering rules which when mechanically applied to a program graph there results an equivalent graph. The process of finding such rules involves looking for a property of the graph whose invariance under the application of a rule is sufficient to guarantee the equivalence between the original and the transformed graphs; in addition it is of interest to know what other properties of the graph change and how they change by virtue of the transformation since generally we are interested in measuring what is gained or lost by such an operation.

Upon considering the type of transformations mentioned above, we conjecture that invariance of the exclusive relations of <u>data junction connectors</u> preserves 1-equivalence under these transformations. In order to obtain some insight into the nature of such transformations we consider several simple examples. Figure 9 depicts a program graph P_1 and an equivalent graph P_2 obtained from P_1 by altering the control links β_1^+, β_1^- and β_2^-. By way of comparison possible flow charts for P_1 and P_2 have been drawn next to the program graphs. The flow charts show clearly that the effect of the transformations is, as we may have guessed, to redistribute operation boxes with respect to decision boxes. In actual programs this kind of redistribution affects both execution time and storage requirements. Considering execution time, there is a simple test to determine whether or not an operator may be applied but its results never used. Clearly all such situations contribute to execution time without affecting the results of the computations and their elimination is desirable. We assert that by adding or deleting control links, control junctions and/or identity operators, <u>all</u> such situations may be eliminated from a program graph; we also note that in other representations not admitting parallel computations such as flow charts there are circumstances which force us to <u>duplicate</u> some operator(s) or selector(s) in order to achieve the same effect.

In Figure 9, f_3 in graph P_1 and f_4, f_7 in graph P_2 are examples of operators which may be applied without having any use for their values. This condition can be remedied by connecting β_1^- to f_4 in P_2. The graph in Fig. 10 is an example of a program which if represented by a flow chart, we must duplicate operator f_8 or some arrangement of selectors in order to reduce it to the 'minimal' form.

D. RELATION WITH PREVIOUS WORK

Representation of computer programs by means of directed graphs have been proposed in the past. In this section we will consider the relationship between program graphs and other graph representations, namely flow charts the graph schemes of Kaluzhnin[9] and the graphs of Estrin and Turn.[4]

Flow charts are widely known and have been used by Prosser, Karp and others to analyze flow of control in programs. Essentially the nodes in a flow chart represent sequence of commands while the branches indicate transfer of control from one node to another either undconditionally or conditional depending on the kind of node. Flow charts thus are intrinsically sequential, furthermore the data flow is not explicity but can only be deduced by a total analysis of control flow with identification of names of variables. Note that as it is generally used, a flow chart does not introduce any new elements in the representation of a program but instead it adopts the syntactic and semantic units of some other language, generally a linear language, and merely uses the graph as a means of showing explicitly the transfer of control relations. Kaluzhnin has proposed a representation of algorithms by means of what he calls graph schemes which are defined as follows:*

"Let there be given a finite set of objects

$$\mathcal{U} = \{\mathcal{U}_1, \mathcal{U}_2, \dots, \mathcal{U}_n\}$$

called operators, and a second finite set of objects

$$\emptyset = \{\phi_1, \phi_2, \dots \phi_m\}$$

called discriminators. A graph scheme Γ or, more precisely, a $\mathcal{U}$-$\emptyset$ graph scheme (with the given operator-sets $\mathcal{U}$ and discriminator-sets $\emptyset$) is a finite, connected and directed linear complex [i.e., a finite number of points (vertices), some of which are linked by directed line segments (arrows), and such that, starting from any point, we can reach any other, following the connected line segments (not necessarily in the direction of the arrows], satisfying the following conditions:

* The quoted text is taken verbatim from Kaluzhnin's paper.

1. One vertex of the complex is identified as the <u>input</u>(B): the input is a unique vertex, which is not the end-point of any arrow, and only one arrow starts from it.

2. The complex has one specified vertex, called the output (θ); no arrow starts out from the output.

3. To each vertex, other than the input and the output, corresponds unequivocally either a certain operator U_i, in which case the vertex will be called a U-vertex, or a discriminator, $\emptyset_j$, in which case it is a $\emptyset$-vertex. (It is not necessary that each operator U_i and each discriminator $\emptyset_j$ should correspond to a certain vertex; on the contrary, some operators or discriminators may correspond to several different vertices). Meanwhile:

   (a) If a is a U-vertex, then exactly one arrow starts out from it.

   (b) If a if a $\emptyset$vertex, then exactly two arrows start out from it, marked with plus and minus signs respectively.

An interpretation of a graph scheme is a set M, mapping $U \longrightarrow A$ and a mapping $\emptyset \longrightarrow F$, where members of A are mappings from M into M and members of F correspond to properties of elements of the set M. The operation of executing a graph is carried out as follows:

"Let the element $m \in M$ appear at the input of the scheme: then it runs through the scheme following the arrows, and is transformed each time whenever it passes through a U-vertex. This passage takes place according to the following rules:

(a) Let the initial element, already transformed into the element m', enter, following the arrows, the vertex corresponding to the operator U_i; then on emerging from a, the element follows its path indicated by the single arrow issuing from the vertex a, as $m''=A_i(m')$.

(b) Again, as above, let the transformed element m' enter a certain vertex , corresponding to the discriminator ϕ_j; then this element m' leaves this vertex along that arrow marked either by a plus or minus sign, depending on whether m' has the property F_j or not.

(c) If at a certain stage the transformed element $\bar{m}$ is the result of applying the U-ϕ-algorithm, defined by the U-ϕ graph-scheme Γ with the interpretation $\{M: U \rightarrow A: \phi \rightarrow F\}$..."

Upon study of these operating rules it becomes evident that graph schemes are a simplified form of flow charts. The key is rule (b) and conditions 3a and 3b which specify that the transformed element has to move <u>through</u> discriminator vertices and there is only one possible path out of any vertex. Graph schemes as extended by Ershov[3] have precisely the same structure as standard flow charts.

Estrin and Turn (apparently following an earlier suggestion of Marimont) have used a graph representation of programs closely related to program graphs. Conceptually, the significant difference between this model and program graphs, lies in the introduction of junctions in program graphs. That the concept of junction is a non-trivial one seems to be supported by the results of Theorems 5 and 6 which specify sufficient conditions for a deterministic program graph based on certain simple properties of the graph junctions. In addition, the concept of N-cycle structure which seems to provide a good handle on the analysis of the loop structure of a program is a direct by-product of the introduction of junctions.

E. DIRECTION OF FURTHER RESEARCH

It is proposed to continue this research along two main avenues namely on the study of transformations of program graphs which preserve equivalence (1-equivalence or others) and on the analysis of program graphs.

On the tópic of program graph transformations we can enumerate various types of transformations which are of interest:

(a) Simplification of the graph by removal of 'vacuous statements' and 'lcop vacuous statements' in the sense of Luckham et. al. Vacuous statements are easily recognized in a program graph as non-terminal nodes whose output connectors are not roots of any links and there is a simple iterative procedure to remove all such nodes from the graph. 'Loop vacuous statements' are also easily recognized and extracted from the pertinent N-cycle(s).

(b) Transformations which merge duplicate nodes. This is a generalization of the problem of recognizeing ' common subexpressions'.

(c) Transformations which introduce duplicate nodes. This type of transformations are useful when altering the control structure of the program graph as one may do when attempting to minimize expected execution time given the relative frequencies of control branches or when one is interested in partitioning the program graph into sub-graphs as in the segmentation problem.

(d) Transformations which modify the N-cycle structure of the graph. Such transformations in conjunction with the merging and introduction of duplicate nodes are useful in the study of programs with intersecting loops.

On the subject of program graph analysis we are roughly interested in finding useful properties of the graph and establishing effective tests for ascertaining the presence or absence of such properties in specific graphs as well as relating such properties to patterns of behavior during the program graph execution. In particular we hope to study in further detail the property which has been called a 'deterministic graph', when there is a knowledge of the possible time interval assignment of some or all nodes of the graph.

Finally, some effort will be applied to extend the program graph model to include data structures which at the present stage of development have been deliberately omitted. In this connection some results of automata theory, particularly those concerning multitape automata might prove helpful.

BIBLIOGRAPHY

1. Berge, C., Theory of Graphs and its Applications, John Wiley and Sons, New York (1962).

2. Cooper, D. C., The Equivalence of Certain Computations, Computation Center, Carnegie Institute of Technology (1965).

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

4. 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.

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

6. Ianov, Y. I., On the Equivalence and Transformation of Program Schemes, Comm. ACM, Vol. I., No. 10, (October 1958), pp. 8-12.

7. Ianov, Y. I., On Matrix Program Schemes, Comm. ACM, Vol. I, No. 12, (Dec. 1958), pp. 3-6.

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

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

10. Landin, P., The Mechanical Evaluation of Expressions, Computer Journal, Vol. 6, No. 4, (Jan. 1964) pp. 308-320.

11. Landin, P., A Correspondence between Algol-60 and Church's Lambda Notation, Parts I and II, (Comm. ACM Vol. 8, No. 2 (Feb. 1965) pp. 89-101 and Vol. 8, No. 3 (March 1965) pp. 158-165.

12. Luckham D., Investigation of the Theory of Algorithms with Emphasis on Program Simplification, Bolt Beranek and Newman, Inc., Report No. 1225, AFCRL-65-170 (Feb. 1965).

13. Luckham D., and Marill T., Techniques of Simplification, Bolt Beranek and Newman, Inc., Report No. 1006, AFCRL-63-112, (March 1963).

14. Luckham D. and Park., The Undecidability of the Equivalence Problem for Program Schemata, Bolt Beranek and Newman, inc., Report No. 1141, AFCRL-64-664, (August 1964).

BIBLIOGRAPHY (Cont.)

15. Marill, T., Computational Chains and the Simplification of Computer Programs, IRE Transactions on Electronic Somputers. Vol. EC-11, No. 2, (April 1962), pp. 173-180.

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

17. McCarthy, J., Recursive Functions of Symbolic Expressions and their Computation by Machine, Comm. ACM. Vol 3, No. 4 (April 1960), pp. 184-195

18. McCarthy, J., A Basis for a Mathematical Theory of Computation, Proc. Western Joint Computer Conf., May 9-11, 1961. Los Angeles, pp. 225-238.

19. Nievergelt, J., On the Automatic Simplification of Computer Programs Comm. ACM, Vol. 8, No. 6 (June 1965) pp. 366-370.

20. Prosser, R. T., Application of Boolean Matrices to the Analysis of Flow Diagrams, Proc. Eastern Joint Computer Conference, Spartan Books, (Dec. 1-3 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. Seshu S. and Reed, M. B., Linear Graphs and Electrical Networks, Addison-Wesley Publishing Co., Reading (1961).

23. Simões Pereira J. M. S., On the Boolean Matrix Equation M = VM Journal ACM, Vol. 12, No. 3, (July 1965), pp. 376-382.