CGTM 45

A COMPUTATION MODEL WITH DATA-SEQUENCED CONTROL

Duane A. Adams

May 1968

Acknowledgment

I wish to thank Professor William F. Miller for his encouragement and guidance during the performance of this work.

D.A.A.

TABLE OF CONTENTS

I. INTRODUCTION

Several recent studies of parallel computation have used the directed graph as a model of the computation. In each of these studies the nodes of the graph represent computation steps and the edges represent transmission paths for data and control. An edge may be thought of as a queue of data produced by one node and waiting to be consumed by another. A computation step may be initiated whenever each edge directed into that node of the graph contains the amount of data required for the node to execute properly. The number of computation steps which may be executed at any given time is dynamically determined by the flow of the data. Thus unnecessary sequencing constraints may be eliminated. A program with the property that the results of the computation do not depend on the number of processors used, their relative speeds, or the possible sequencing of computation steps among those nodes which are ready to be initiated is said to be determinate.

The directed graph model has been used in connection with making a priori estimates of computation times in a parallel processor environment [3], showing the equivalence of certain programs [4], studying such properties of parallel computations as queue boundedness and conditions for a computation to terminate [1], and on-line pictorial specification of computer procedures [5]. In each of these studies a different model has been developed. The models of Karp and Miller [1] and Rodriguez [4] have been formally defined, and each program expressed in either of these models is determinate. However, it is not easy to express programs in either of these models. The Karp Miller model does not allow data dependent decisions. Neither of the

models has any facility for working with structured data nor for using procedures in writing programs. On the other hand, the less formally defined models of Sutherland [5] and Martin [3] are not determinate. Martin's model also does not allow the use of structured data or procedures, and Sutherland's work is more concerned with specifying computer procedures graphically than with the problems of parallel computation.

The model which is described in this report is actually a framework within which various classes of computations may be expressed. It is an interpreted model in the sense that each of the primitive nodes represents a specific, well-defined function. Structured data is introduced into the model by defining a set of data types, where each data type is specified as a production in a grammar. A graph procedure is defined recursively. This definition allows programs to be written in a hierarchical manner and also allows recursively defined programs to be written.

Included in the description of the model is a definition of the execution of a graph program. The execution of procedure nodes causes processes to be dynamically created and deleted, using a copy rule similar to the one in ALGOL 60. As a result, more operations may be carried out in parallel than in the previously defined models. The execution of a graph program is defined independently of any assumptions about computer organization, number of processors, or their relative speeds. It is only assumed that there are processors capable of realizing the functions defining the primitive nodes.

Each graph program expressed within the model is determinate. Furthermore, it is shown that all computable functions can be represented within the model. Emphasis has been placed on developing a model which can be programmed. Later reports will show how the model is used to represent algorithms for parallel computation and will discuss the selection of primitive nodes and data types to be used within the model.

II. REVIEW OF RELATED WORK

In the section we shall review some of the recent research in modeling parallel computations. The primary emphasis here is with a description of the models involved, and whether or not the models possess determinacy properties. Four of the models discussed use the directed graph to describe parallel computations. The remaining two are models in which the primitive operations defining the algorithm alter the contents of a memory in the course of the computation.

1. Karp and Miller Computation Graph

The Karp and Miller Computation Graph [1] is a directed graph consisting of:

(i) nodes $n_1,\ldots,n_\ell$;

(ii) branches $d_1,\ldots,d_t$, where any given branch d_p is directed from a specified node n_i to a specified node n_j;

(iii) four nonnegative integers, A_p, U_p, W_p, and T_p, where $T_p \geq W_p$, associated with each branch d_p.

The parameters on a branch have the following interpretation:

A_p - the number of data words initially in the first-in first-out (FIFO) queue associated with d_p;

U_p - the number of words added to the queue whenever operation O_i associated with n_i terminates;

W_p - the number of words removed from the queue whenever operation O_j is initiated;

T_p - a threshold giving the minimum queue length of d_p which permits the initiation of O_j.

The operation O_j associated with a given node n_j is eligible for initiation if and only if, for each branch d_p directed into n_j, the number of words in the queue associated with d_p is greater than or equal to T_p. When O_j becomes eligible for initiation, the first W_p words are removed from each branch d_p directed into n_j. When O_j terminates, U_q words are placed on each branch d_q directed out of n_j. There are no execution times specified for any nodes in the model, but no two performances of a node can be simultaneously initiated. The execution of a graph is a sequence of sets S_1, S_2,..., possibly infinite, where each set S_i indicates those nodes that are initiated at the i^{th} step. A proper execution is an execution in which any node that is ready to be initiated appears in some set S_k; that is, it will be initiated within a finite period of time. The sequence of data values placed on each edge of a computation graph is the same for each proper execution; thus the computation graph is said to be determinate.

2. Estrin - Turn - Martin Model

In the model described by Estrin and Turn [7] and Martin [3] the directed graph is also used to model parallel computations. The vertices of the graph represent "unambiguously defined computational statements", and the arcs joining the vertices represent data dependencies and sequencing requirements. The inputs to a vertex may be either conjunctive input in which data is required on all the inputs before the computation at that vertex can begin, or mutually exclusive input in which data is required from only one of several mutually exclusive origins. The outputs from a vertex may also be of two types. An AND type output allows data to be simultaneously placed on different arcs from a given vertex. An EXCLUSIVE OR output allows data dependent branching to take place by placing data on only one of a set of arcs.

Determinacy is not discussed, and, in fact, it appears that the programmer must be careful to insure that a vertex with mutually exclusive input requirements receives only such inputs. Cyclic graph structures are analyzed by transforming them to temporally-equivalent acyclic graphs. This research is concerned primarily with making a priori estimates of expected computation time for given problems on given processing systems.

3. Rodriguez Model

The Rodriguez model [4] may be considered an extension and formalization of the Estrin and Turn model. The nodes of the graph are computational elements and the links are storage and transmission elements. There are data links and control links. Associated with each link is a quantity called the link status. At any given time, the link-status of a link can assume one of four possible values. The data links also have a property called data contents.

There are seven types of nodes, differing in the types of computation they perform and the logic used to activate them. Each node has specific points of attachments called connectors - they may be either input or output connectors. The different node types are

1) Operator - a function of one of more inputs is realized and placed on the one or more outputs;

2) Selector - a decision making element which enables one of its outputs and disables the other;

3) Junction - merges two or more sources of data by transmitting at most one of the inputs on as a unique output;

4) Loop Junction - used to form cyclic structures by blocking the initial input to a node until the iteration has been completed;

5) Loop Output - used in connection with the loop junction to define the output of an iterative process;

6) Input Terminals - nodes with no inputs and one output;

7) Output Terminals - nodes with no output and one input.

Note that only the operator node performs transformations on input data values. The remaining node types perform various control functions. A transition table gives for each node type the set of link statuses for which the node is in an active configuration - i.e., the conditions required for the node to perform the indicated computation.

A program graph is constructed from a finite set of input terminals, output terminals and nodes interconnected by links, where data links must be connected to data connectors, control links to control connectors, and each input connector of a node must be connected to some link. When a program is executed, any node in active configuration may be activated. Its behavior is governed by the transition table for that node type.

The links are such that only one data value may be stored on a link at any given time. This condition is reflected in the transition table for the node types. Data dependent decisions are allowed in the model, but such decisions require disabling all operations on the unselected branch. All program graphs in the Rodriguez model are determinate. The model is only applicable to computations with non-structured data. In particular, the results are applicable to the design of macro-modular systems where the nodes of the graph may correspond to hardware components and the edges to actual data paths.

4. Sutherland's Graphical Specification of Computer Procedures

Sutherland's [5] aim was not to model parallel computations but rather to use computer-aided drawing techniques for graphically describing computer procedures. His format for a program is a graph. Computations are represented by box-line symbols with input and output terminals for connecting symbols

together. The terminals are portals through which variable values are introduced to and obtained from a function within the box. The terminals are classified by data type, and are connected together by lines which carry outputs of one symbol to the inputs of another. A symbol may have a primitive meaning, or may be a collection of symbols forming a subroutine.

An operator may be activated provided that all of its inputs are defined. When an operator is activated, each terminal may either keep the input data or reset the variable to "undefined" to wait for new data to arrive. Parallel operations are mentioned by Sutherland, but are rather briefly treated. The problem of determinacy is not discussed, and there remain unanswered questions regarding the merging of data from more than one source onto a single line and regarding timing considerations when an unlimited number of processors is not available.

5. Van Horn's MCM

Van Horn [6] has proposed a class of abstract machines called MCM, machines for coordinated multiprocessing. An MCM is a machine in the same sense that a Turing Machine is. An MCM consists of a set of cells, a scheduler, and a count matrix. Cells may be either clerk cells, which correspond to a CPU on a conventional machine, or value cells, which correspond to memory elements on a conventional machine. The count matrix has its rows and columns labeled by the cell names. If the contents at row x and column n of the count matrix are positive, then it means that clerk cell x may read the contents of cell n. If x is the only clerk which may read cell n, then it may also write in cell n. The operations which the clerk cells may perform correspond to a get in which the operand is read, a function of this operand evaluated, and the result left in the clerk cell; a put in

which the contents of a clerk cell are written into another cell; and 3 operations which affect the contents of the count matrix. The scheduler makes sure the execution strategy is reasonable (this is the same as a proper execution defined by Karp and Miller), and insures that no two clerks perform a transaction which alters the same element of the count matrix.

With the above definition of an MCM, Van Horn shows that an MCM is output-functional, that is, each symbol in the output stream is a function only of the initial input state. He also shows that the computation on an MCM is output-assured, i.e., the computation will not halt prematurely. This behavior of an MCM is referred to as asynchronously reproducible, and is equivalent to being determinate.

6. Karp and Miller Parallel Program Schemata

The parallel program schema described by Karp and Miller [2] is a very general model for representing algorithms containing parallel sequencing. The schema $\mathcal{S}$ is specified by:

(1) a set M of memory locations,

(2) a finite set $A = \{a, b, \ldots\}$ of operations, and for each operation a:

  (i) a positive integer K(a) called the number of outcomes of a;

  (ii) a set $D(a) \subseteq M$ whose elements are the domain locations of a;

  (iii) a set $R(a) \subseteq M$ whose elements are the range locations of a.

(3) a quadruple $\mathcal{J} = (Q, q_o, \Sigma, \tau)$ called the control where:

  (i) Q is a set of states;

  (ii) q_o is an initial state;

  (iii) $\Sigma = \bigcup_{a \in A} \{\bar{a}, a_1, \ldots, a_{K(a)}\}$ is the alphabet;

  $\bar{a}$ is called the initiation symbol and $a_1, \ldots a_{K(a)}$ are called termination symbols.

(iv) τ , the transition function, maps a subset of $Q \times \Sigma$ into Q.

The schema $\mathcal{S}$ is uninterpreted. An interpretation $\mathcal{I}$ is provided by stating the possible contents of each memory location i, the initial contents of memory, and a function F_a which determines the result which a stores upon completion and a function G_a which, upon completion of a, determines a K(a)-way conditional branch.

An $\mathcal{I}$-computation represents a possible sequence of primitive steps in applying the algorithm represented by $\mathcal{S}$. The transition function τ determines, on the basis of the current state q, whether a particular operation a may be initiated. If more than one performance of a has been initiated but not completed, the order of termination must be the same as the order of initiation. A schema $\mathcal{S}$ is determinate if whenever x and y are $\mathcal{I}$-computations for the same interpretation $\mathcal{I}$, the sequence of values place in each cell $i \in M$ is the same for x and y. Under certain hypotheses, determinacy is equivalent to a kind of commutativity between primitive steps in a computation.

III. DESCRIPTION OF THE MODEL

The model described below is an attempt to provide a framework within which various classes of computations can be represented. The model has been developed so that computations represented within it will be determinate. The main emphasis has been on developing a model that can be programmed. This latter goal is reflected in our attempt to deal with data structures and a hierarchical description of the program. Data structures are treated quite generally, and include the hardware defined structures such as bits and words and the structures usually defined in a programming language such as arrays, strings, and lists. Our hierarchical program description is achieved by being able to treat each node of our graph as representing some operation -- perhaps very complex -- and also being able to represent as a graph the sub-operations or instructions of which it is constructed. Thus a graph program will be defined recursively.

1. Elements of the Model

Our approach to defining a model for parallel computations is to provide a set of primitive elements (basic elements from which other structures can be constructed) and to define the rules for using these primitives. Various classes of computations can be described by varying the set of primitive elements. In the following definition we shall use standard mathematical notation for sets and functions. Other terminology and notation used here frequently appears in the study of formal languages. In describing a data structure grammar, we shall use the following meta symbols:

$\longrightarrow$ denotes a production;

$*$ denotes zero or more repetitions.

Definition 1. Within the model M a class of computations is defined by specifying the following:

i) A set of data types $d_1,\ldots,d_h$.

The syntax of the data types is described by a grammar with vocabulary $V=V_N+V_T$, where $V_N=\{d_1,\ldots d_h\}$, the nonterminal symbols, and $V_T=\{[,],Y_1,\ldots Y_k\}$, the terminal symbols. The data types are described by productions of the form $d_i \longrightarrow x$, with $d_i \epsilon V_N$, $d_i \neq x$, $x \neq \lambda$, the empty string, and either $x = \alpha_1\alpha_2\ldots\alpha_n$ or $x = [\alpha_1\alpha_2\ldots\alpha_n]$, where each α_j is one of

a) an element of V - {[,]} or

b) an element of V - {[,]} superscripted with * or an integer constant.

ii) A set of primitive nodes $p_1,\ldots,p_{k-1},p_k,\ldots,p_n$.

Each primitive node p_i, $1 \leq i \leq n$, has an ordered set of ℓ inputs requiring data of types $d_{i1},\ldots,d_{i\ell}$, an ordered set of $m - \ell$ outputs producing data of types $d_{i\ell+1},\ldots,d_{im}$, and a single-valued function f_i: $(V_{i1},\ldots,V_{i\ell}) \longrightarrow (v_{i\ell+1},\ldots,v_{im})$, where V_{ij} is either an element of type d_{ij} or φ, the null element; and v_{ij} is either a finite sequence of elements, each of type d_{ij}, or φ, the null element. Neither f_i nor g_i is defined when all of the arguments $V_1,\ldots,V_\ell$ are the null element. In addition, for $k \leq i \leq n$, each input of p_i has a binary-valued input status S_{ij}, $1 \leq j \leq \ell$, and a single-valued function g_i: $(S_{i1},\ldots,S_{i\ell},V_{i1},\ldots,V_{i\ell}) \longrightarrow (S'_{i1},\ldots,S'_{i\ell})$.

Those nodes defined only by the function f will be called r-nodes, and those nodes defined by both f and g will be called s-nodes.

iii) Directed edges.

Each edge is of data type d, where $d \in \{d_1,\ldots,d_h\}$, the set of data types.

Edges

The edges in the model M serve as temporary storage areas for data; they are first-in first-out queues of data. The parameter d identifies the syntactic structure of the elements found on the queue, and the data structure grammar describes the composition of these elements. Thus, depending on the data type of an edge, the elements on that edge may be very complex structures. Each element on a queue is implicitly separated from the remaining elements; that is, each element that is added to the queue may later be unambiguously removed from the queue. This implicit separation of elements on a queue is a generalization of the implicit separation of words in a computer memory. The length of each queue is assumed to be unbounded.

Data Types

As stated in definition 1, the primitive nodes are defined to operate on specific types of data, and each of the edges is defined to hold data of a particular type. For each data type $d_i \in V_N$, the grammar states precisely what strings and ordered sets of terminal symbols are elements of that type. The productions which define the data types fall into two distinct categories:

1) productions of the form $d_i \rightarrow \alpha_1\alpha_2\ldots\alpha_n$;

2) productions of the form $d_i \rightarrow [\alpha_1\alpha_2\ldots\alpha_n]$.

In the first case, an element of type d_i is the concatenation of the elements which correspond to the α's. This is the usual sense in which productions are used in the description of formal languages. In the second case, an element of type d_i is an ordered set of the elements $\alpha_1,\alpha_2,\ldots,\alpha_n$, enclosed in brackets. The brackets are used both as terminal symbols which enclose the remaining elements and to indicate that the enclosed elements form an ordered set. By treating the elements between the brackets as an ordered set there is an implicit separation of elements. This is meant to correspond to the notion of a queue of elements on an edge. A symbol superscripted by * denotes that zero or more copies of that symbol may appear in the productions, and a symbol superscripted by an integer constant denotes that a fixed number of copies of that symbol appear in the production.

A production of the form $d_j \rightarrow [d_i^*]$ would be interpreted to mean that each element of type d_j is a queue of elements, where each element on the queue is of type d_i. The queue may be of arbitrary length, including the queue with no elements at all. The ability to treat a queue of elements as a single element on another queue serves as a naming convention. It allows us to do what is done in ALGOL when a name is declared to be, say, an array of some given size. In procedure calls we can then pass a structure as a single element to the procedure, just as in ALGOL the name of a structure is used in the procedure call. Another reason for treating a queue of elements as a single element on another queue is that the only storage in our model is in the form of first-in first-out queues of data stored on the edges between nodes; there is no random access storage.

Thus, new structures cannot be formed by creating pointers to substructures, as is done in the implementation of LISP, for example. However, the use of pointers would be a natural way to simulate these data structures on a computer with a random access memory.

The set V_T of terminal symbols always includes the brackets, [and], although they need not be used. A special use of the brackets in connection with procedures will be discussed later. For the remaining symbols, $Y_1,\ldots,Y_k$, we might choose 0 and 1. These terminal symbols could be used to represent the data in a binary storage device. On the other hand, we may choose as terminal symbols the alphabet, decimal digits and perhaps some special symbols such as parentheses. This might be an appropriate set if one is concerned with symbolic manipulation.

In the following example of a set of data types $V_T = \{[,],0,1\}$ and $V_N = \{d_1,d_2,d_3,d_4\}$. The productions are given by:

$$d_1 \rightarrow 0 \mid 1$$
$$d_2 \rightarrow d_1^{32}$$
$$d_3 \rightarrow [d_2 d_2^*]$$
$$d_4 \rightarrow d_2 \mid [d_4^*]$$

The vertical bar used above is a meta symbol meaning or. In this grammar each element on a queue of type d_1 would be either a 0 or a 1. An element on a queue of type d_2 would be any string consisting of 32 zeros and ones, hence a 32-bit word. The elements on a queue of type d_3 would be 1-dimensional arrays of 32-bit words, each array containing at least one element. Data type d_4 is defined recursively and allows tree-structured data to be represented. Each of the branches eventually leads to either an ordered set of 32-bit words or the empty set. Some of the allowed structures are

[], [[]], [$d_2 d_2$] and [d_2[$d_2 d_2$] []], where in this example each d_2 represents a 32-bit word.

Note that there is a difference between the production $d_i \rightarrow [d_j]$ where $d_j \rightarrow \alpha_1 \ldots \alpha_n$ and the production $d_i \rightarrow [\alpha_1 \ldots \alpha_n]$, even though the same elements appear in each case. In the first case, a single element appears between the brackets; in the second it is an ordered set of n elements. When n=1 the two productions would be the same. The production $d_i \rightarrow [d_j^*]$ is well-defined, but the production $d_i \rightarrow d_j^*$ is not since it would admit the empty string λ as an element of type d_i. When we speak of the null element φ being placed on an edge, it is to be interpreted that no output is placed on the edge. φ is distinct from the empty set [].

Certain nonterminal symbols may be introduced as intermediate symbols in creating other structures, with no intention of having queues of those types. Besides being used to represent a rather general quantity such as the 32-bit word defined above, the data types may be used for such purposes as distinguishing between integers and floating point numbers. The use of structured data in programs within the model will be explained after the notions of a graph procedure and the execution of a graph program have been explained.

Primitive Nodes

The primitive nodes represent the "building blocks" from which the programs must be constructed. Since each of the primitive nodes has a particular well-defined function associated with it, our model is an interpreted model. The particular choice of primitive nodes determines the class of computations which can be represented within the model and the ease with which such computations can be programmed. The functions performed

by the primitive nodes might range from logical operations performed on bits to such complex operations as matrix multiply.

Each input status setting for an s-node may be either locked or unlocked, denoted by L and U, respectively. When the j^{th} input is in locked status, then the argument V_j is implicitly defined to be φ for both the functions f and g. That is, any values on the j^{th} edge directed into the s-node remain on the edge and are not used in the computation. The g function associated with an s-node allows the status settings to be changed as a function of the current settings and the input arguments to the node. The s-nodes allow data to be selected from mutually exclusive sources in a determinate manner, thus avoiding race conditions.

The function f associated with each primitive node is used in a very general sense. The domain of the function has been defined to permit the null element within the ℓ-tuple of arguments for f, as long as the entire ℓ-tuple does not consist of null elements. The null element is only allowed as an argument of f when the corresponding status setting is locked. Thus, for an r-node where there are no status settings, the null element is not allowed as an argument of f. The function f may be a conditional function which produces output on only some or possibly none of its edges. The null element φ will indicate that no output has been placed on an edge. More than one element may be placed on an edge as output. For a given ordered set of inputs, both f and g have uniquely defined outputs.

The following example of an s-node illustrates some of the ideas which we have been discussing. Suppose the node has 3 inputs and one output, and its function is to accept a Boolean signal on its first input, and depending on the value of the Boolean, accept the next element of data from

either the second or third input and place it on the output. Suppose that the second and third inputs and the output are all of type word. The f and g functions for this node are given below.

g:		f:	
	$(U,L,L,T,\varphi,\varphi) \rightarrow (L,U,L)$		$(T,\varphi,\varphi) \rightarrow (\varphi)$
	$(U,L,L,F,\varphi,\varphi) \rightarrow (L,L,U)$		$(F,\varphi,\varphi) \rightarrow (\varphi)$
	$(L,U,L,\varphi,V,\varphi) \rightarrow (U,L,L)$		$(\varphi,V,\varphi) \rightarrow (V)$
	$(L,L,U,\varphi,\varphi,V) \rightarrow (U,L,L)$		$(\varphi,\varphi,V) \rightarrow (V)$

Note that not all combinations of status settings have been taken into account in the above functions. Those not listed are not meaningful for the above problem, and it is assumed that they would not be encountered. For completeness of definition, however, we will adopt the convention that for the combinations of status settings not listed, the g functions locks all inputs and the f function procedures all null outputs.

2. Composition of Graph Programs

Definition 2. A graph procedure G consists of a node set $\mathcal{N} = \{N_1,\ldots,N_\ell\}$ and an edge set $\mathcal{E} = \{e_1,\ldots,e_m\}$, where each edge $e_i \in \mathcal{E}$ must satisfy one of the following three conditions:

i) be directed from the output of a node $N_j \in \mathcal{N}$ to the input of a node $N_k \in \mathcal{N}$;

ii) only be directed to the input of a node $N_k \in \mathcal{N}$. The ordered set of such edges is called the input set of G, and is non-null;

iii) only be directed from the output of a node $N_j \in \mathcal{N}$. This ordered set of edges is called the output set of G, and is non-null.

Some $e_i \in \mathcal{E}$ must be connected to each input and each output of the nodes in $\mathcal{N}$. Each edge must be data type compatible with the node or nodes onto which it

is connected. The nodes of η may be either primitive nodes or procedure nodes.

Definition 3. A procedure node P is a node whose operation is described by a graph procedure G. The ordered set of inputs and the ordered set of outputs of the procedure node must agree in number and data type with the input set of G and the output set of G, respectively.

Definition 4. An edge e of data type d_i is compatible with the node or pair of nodes onto which it is connected when the following conditions are satisfied:

i) if e is directed into a node, then the input of that node must be of type d_i;

ii) if e is directed from a primitive node, the output of that node must be of type d_i;

iii) if e is directed from a procedure node, the output of that node may either be of type d_i, or if there is a production in the data type grammar of the form $d_i \rightarrow [d_j^*]$, it may be of type d_j.

Definition 5. A graph program $\mathcal{P} = G,X$ is a graph procedure G and the set X of all graph procedures such that

i) if P is a procedure node in G and Y is the graph procedure defining P, then $Y \in X$;

ii) if $Y \in X$, P is a procedure node in Y, and W is the graph procedure defining P, then $W \in X$;

iii) X is defined only by i) and ii), with no graph procedure appearing more than once in X.

Before discussing the graph procedure and the use of structured data in a graph program, it is perhaps worth while to say in a general and informal way how a program is to execute. The initial data for the program is placed on the appropriate edges. During the execution of a graph program, a node in the graph may be executed whenever there is at least one element of data on each of its unlocked input edges. When the node executes, it removes one element of data from each unlocked input, performs the indicated computation and places the results on the appropriate edges leading out from the node. For an s-node, the status settings may also be changed. When no further nodes can be executed, the computation is complete, and the results will be left on the edges designated as the output set of G.

The definition of a graph procedure allows us to write recursively defined programs. A procedure node may be defined by the graph procedure within which it appears. Thus it will not always be possible to simply replace each procedure node in a graph program by its defining graph procedure, since the process may be nonterminating. When a procedure node is ready to be executed, a copy of the defining graph procedure is created. This is similar to the ALGOL 60 copy rule for procedures. The arguments of the procedure are placed on the edges of the graph procedure which are designated as the input set, and the graph procedure is then allowed to perform the indicated computation. When the execution of the procedure has been completed, the results on the edges of the output set are transferred onto the corresponding edges directed out from the procedure node.

The definition of a graph program makes no reference to declared data structures, and, in fact, there are none. The data is dynamically structured during the execution of a program, as is done in LISP. There are two basic methods for structuring data in the model, whereas in LISP

there is only one. In LISP, the cons operation forms a new S-expression from two smaller S-expressions. Th operation is sequential, and to form a list, for example, requires the repeated consing of elements onto the partially formed structure. In the graph model, the first method of structuring data is to have primitive nodes which accept an operand on each of their input edges and output the structured result. A primitive node might be included whose operation is identical to the cons operation in LISP. More generality is allowed in the graph model since the number of operands need not be two, other operations besides the cons may be used, and it is possible to define structures other than S-expressions.

The second method for structuring data is referred to as bracketing, and is used only in connection with procedures. The elements on an edge in the output set of a graph procedure may logically form a unit of data, and we would like to treat these elements as a unit rather than as the individual elements on the queue. For example, a procedure may produce a sequence of words which we wish to consider as a vector, or a sequence of elements which forms a list. For determinacy reasons, it is not possible to have a primitive node remove a variable number of elements from a queue and form the desired structure. Bracketing can occur only after the procedure has finished executing, and thus it is know that all elements which are to be placed in the structure have been formed.

Whether or not the elements on an edge are to be bracketed is determined by the data type of the edge in the output set and the data type of the edge leading out from the procedure node. We note first of all that when a procedure terminates, the data is transferred from the edges of the graph procedure to the edges directed out from the procedure node. For each edge in the output set of the graph procedure, the corresponding edge

directed out from the procedure node may either be the same type or it may be a different data type; and for these two alternatives, the data types may or may not be related by a production of the form $d_i \rightarrow [d_j^*]$. We consider these four cases. For an edge of type d_j in the output set of the graph procedure with corresponding edge of type d_i directed from the procedure node;

1) $d_i \neq d_j$ and $d_i \not\rightarrow [d_j^*]$ -- this combination is not allowed by the definition of compatible;

2) $d_i \neq d_j$ and $d_i \rightarrow [d_j^*]$ -- bracketing occurs;

3) $d_i = d_j$ and $d_i \not\rightarrow [d_j^*]$ -- no bracketing occurs;

4) $d_i = d_j$ and $d_i \rightarrow [d_j^*]$ -- bracketing occurs.

Only in case 4) was there a choice of bracketing or not bracketing, and it was decided to bracket the output since the operation of removing unwanted brackets is easier than having to sequentially form the structured data. An alternative to 4) would be to provide some indication in the graph that the output is to be bracketed.

The input arguments to a procedure node are simply passed on to the defining graph procedure. There is no operation which can automatically remove the brackets from an input element. If the correct operation of the graph procedure requires that brackets be removed, then this must be done by a primitive node within the graph procedure. The consequence of this requirement is that any attempt to remove brackets from unbracketed data will cause an error within a primitive node and not an error in the initiation of a procedure node.

By requiring that the data type of an edge be compatible with the data type of the nodes onto which it is connected, we are insuring that the nodes always receive the type of data for which they are defined. Our definition of compatible could have been more general. For example,

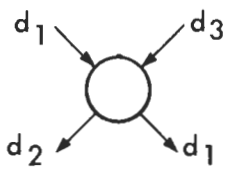
suppose our data type grammar were

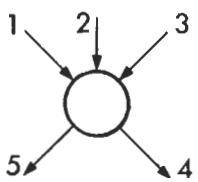
$B \rightarrow 0 \mid 1$ and $C \rightarrow B \mid BB$.

Since each element of type B is also of type C, we may have allowed a node producing data of type B to place its output on a queue of type C. In our model, however, a node would be needed to convert an element from type B to type C, even though there is no real transformation taking place. Thus, as a result of attempting to make some of the definitions simpler, we are forcing more operations to be performed by the primitive nodes.

3. Notation for Graph Programs

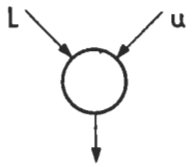
The definitions which we have given thus far have been mathematical in nature. The objects defined, however, are meant to have an intuitive appeal. Furthermore, we need to have a notation for representing the programs in our model.

A node is represented by a circle  . When discussing the properties of a node, the inputs and outputs will be designated by directed line segments,  . The data type will be indicated beside each directed line segment, 

. We shall assume that the inputs and the outputs each appear in consecutive order, and that they are ordered in a clockwise direction starting with the first input, 

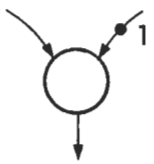
.

There must be at least one input and one output for each node, so that this ordering is unique. When the edges are not explicitly ordered, it is assumed this convention is being used. The initial status settings for an s-node will be denoted by writing an L or a U beside the corresponding input,



.

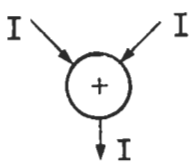
One drawback of the graph model as a means for expressing programs is that there is no convenient way for representing the data. The Karp and Miller model [1] used an edge parameter to show the amount of data initially placed on each edge, but there was no means of indicating the values of the operands. When an algorithm can be better understood by knowing certain values, especially constants, we shall denote it as



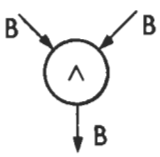
.

The interpretation here is that a 1 is initially placed on edge 2.

We will identify the nodes in a graph program, both primitive and procedure, by placing an identifier within the node. For the procedure node, this identifier will be the same as the name of the defining graph procedure. The identifiers may be alphabetic strings, numbers, or even mathematical symbols denoting the operations to be performed. For primitive nodes,



may be used to denote the addition of two integers, or



to denote the logical AND of two bits. A procedure node may appear as

, where SQRT is the name of

a graph procedure which calculates the square root of a real number. Here it is assumed that B, I, and R have all been precisely defined by a data type grammar, and that each primitive node has also been given a precise definition.

There is really no limitation as to the kind of functions the primitive nodes may be defined to perform. A primitive node is primitive in the sense that we are not interested in its structure below a certain level. This doesn't mean that it can't be described in terms of even more basic functions. This raises the question as to how we should describe the semantics for the primitive nodes. At present we have no specific set of nodes and data types from which all others can be constructed. Even if we did, it might be very laborious to define our semantics from such primitives. Neither ALGOL nor LISP are well suited for expressing the semantics of our primitive nodes, primarily because it is not easy to use ALGOL or LISP to manipulate the data structures allowed within the model. We shall not adhere to any specific formalism in expressing the semantics of the primitive nodes. Where possible, we shall try to use conditional expressions, but we may also use tables or verbal descriptions, if appropriate.

4. Execution of a Graph Program

The program graph of Rodriguez [4] and the computation graph of Karp and Miller [1] may each be considered as formally representing a computing machine. In each case the nodes represent the computing units and the edges represent temporary storage areas for data. It is difficult to treat our graph programs as computing machines in the above sense. This is primarily because of the recursive nature of the graph procedure. When a node in

a graph procedure represents a recursive call upon the procedure of which it is a part, the only reasonable way to handle this seems to be to create a copy of the called graph procedure. Thus, during the execution of a graph program, an auxiliary graph referred to as the executing graph will be constructed. Initially the executing graph will consist of the main graph procedure G, with the initial data for the program placed on the edges of G. The initial data must be of the same type as the edges on which it is placed. Whenever a procedure node in the executing graph is ready for execution, a copy of the defining graph procedure will be created; and when the procedure terminates, the created copy will be deleted. The executing graph can thus expand and contract dynamically during the execution of the program.

We shall imagine that the nodes in the executing graph describe computations to be performed. We assume that for each primitive node p_i there is a processor F_i capable of realizing the functions f_i and g_i in a finite period of time. No particular organization or speeds of the processors is assumed. In fact, the nodes themselves may be the processing units if one wishes to assume that processing units may be created dynamically. In a realistic situation there will probably be some fixed number of processors which must be allocated to various computations as needed. Thus the dynamic expansion of the executing graph may be governed by both the structure of the program and the available resources.

In the following definitions relating to the execution of a graph program, there is an attempt to permit as much parallelism as possible in the execution, while at the same time maintaining determinacy.

Definition 6. Associated with each node N in the executing graph is a first-in first-out queue called the initiation queue, denoted IQ_N. At any

given time, the elements on IQ_N identify the executions of N which have been initiated but not terminated, and $n(IQ_N)$ gives the number of such elements on IQ_N. Initially, $n(IQ_N) = 0$ for all N.

As definition 6 indicates, more than one execution of a node may be in progress simultaneously. This is true for both primitive nodes and procedure nodes, and each has an initiation queue. The purpose of the initiation queue is to insure that when more than one execution of a node is simultaneously in progress, the executions terminate in the same order that they were initiated. This is necessary to preserve determinacy in the execution of $\mathcal{G}$, since we are making no assumptions about the relative speeds of any of the processors involved in the computation.

The execution of a node N in a graph program is a mapping of an ordered set of input values into an ordered set of output values; and if N is an s-node, the execution of N may cause the status settings to be reset. When N is a primitive node, this mapping is defined by the functions f and g, and for a procedure node by the defining graph procedure. We are now interested in the conditions under which a node may execute, and what effect the execution of a single node has on the executing graph. The execution of primitive nodes and procedure nodes is similar in many respects, but since there are some important differences, separate definitions will be given. To make some of the definitions a little simpler, it will be assumed that all primitive nodes have an input status on each input, but for r-nodes these status settings are always unlocked.

Definition 7. Let p be a primitive node and E(p) denote an execution of p. Then

a. E(p) is ready for initiation, written $E^I(p)$, when there is at least one data element on each edge directed into an unlocked

input of p. In addition, if p is an s-node, $n(IQ_p) = 0$.

b. The initiation of $E^I(p)$ consists of

i) removing the first element from each queue directed into p with unlocked input, and assigning these arguments to a processor F capable of realizing the functions f and g associated with p;

ii) adding to IQ_p an identifier k not already on the queue, and assigning to F the same identifier.

c. E(p) is ready for termination, written $E^T(p)$, after F has realized the functions f and g and the identifier k assigned to F is the first element on IQ_p.

d. The termination of $E^T(p)$ consists of

i) placing the results produced by F onto the appropriate outputs of p, and if p is an s-node, resetting the input status settings of p;

ii) removing the first element on IQ_p and releasing F.

Definition 8. Let P be a procedure node and E(P) denote an execution of P. Then

a. E(P) is ready for initiation, written $E^I(P)$, when there is at least one data element on each edge directed into P.

b. The initiation of $E^I(P)$ consists of

i) creating a copy of the graph procedure G which defines P;

ii) removing the first element on each input to P and placing on the corresponding edges of the copy of G;

iii) adding to IQ_p an identifier k not already on the queue and assigning the same identifier to the copy of G, denoted G_k.

c. $E(P)$ is ready for termination, written $E^T(P)$, when

i) no nodes of G_k are ready for initiation;

ii) no nodes of G_k have been initiated but not terminated;

iii) the identifier k is the first element on IQ_P.

d. The termination of $E^T(P)$ consists of

i) placing the values of the output set of G_k onto the corresponding output edges of P, bracketing if necessary;

ii) removing the first element of IQ_P and deleting G_k.

The main difference between the execution of a primitive node and a procedure node is that an execution of a procedure node causes the executing graph to dynamically expand and contract. A copy of a graph procedure, once created, behaves exactly as any other portion of the executing graph. Each node so created has its own initiation queue, and if it is a procedure node, it can cause other graph procedures to be copied. In the model, the initations and terminations of a node execution are assumed to be uninterruptible operations, during which time other initiations and terminations, respectively, of that node cannot take place. However, one execution of a node may be initiating while another execution of that same node is terminating.

The requirement in definition 7a that $n(IQ_p) = 0$ before an s-node is ready for initiation is there to insure determinacy. It prohibits a second execution of an s-node from being initiated until the first has terminated, since the operands for the following execution are determined by the g function of the execution in progress. Actually all that would be needed for determinacy would be for the g function to have determined the new status settings before the next execution were initiated.

More than one execution of any given r-node or procedure node may be in progress at the same time. This allows a greater degree of parallelism in the execution of a program. Since no assumptions have been made regarding the organization of any computer or the speeds of any processors, it was necessary to let each node have an initiation queue in order to preserve determinacy. For a particular computer, however, it may not be necessary to have an initiation queue for the primitive nodes. This assumes that the organization of the computer is such that determinacy is still preserved. For example, it may be that each processor takes a fixed length of time to perform a given operation, or that only one execution of a node may be in progress at a given time. The definition of an execution of a node requires that the executions be sequentially initiated and terminated. The simultaneous initiation or termination of more than one execution of a node could have been defined, as long as the order of the elements on the edges directed out from such a node would be the same as with the sequential initiations and terminations. For the general case, where such simultaneous initiations need not be followed by simultaneous terminations, the definitions become quite complex.

The intuitive notion of an execution of a graph program $\mathcal{G}$ is that of a sequence of primitive and procedure node executions, where more than one node may be executing simultaneously. The time taken to execute a given node may vary from one execution to the next. Also, it may turn out that a procedure is in an infinite loop. Thus the corresponding procedure node would never be able to terminate and would thus not satisfy the formal definition for a node execution. Yet we want to include such cases in the definition of a program execution. We can achieve the necessary conciseness by defining the execution of a program in terms of initiations and terminations of nodes. The definition will be similar to that used by Karp and Miller [1].

In the definition of a node execution we were only concerned with the conditions for a single node to execute. Also we were not talking about any particular node in the executing graph. However, during the execution of a graph program we may have created several copies of the same graph procedure for which we must be able to distinguish the different nodes and edges. We must be able to identify these nodes and edges independently of the order in which they were created, so that two different executions of a graph program $\mathcal{P}$ may be compared. Each node and edge which could possibly be created in the executing graph will be assigned a Gödel number. A schema will be given for creating an infinite number of node and edge numbers, of which only a finite subset would ever be used in a program which terminates. This schema will allow us to uniquely identify each node and edge which appears in an executing graph. Although such a schema as we are presenting here would not be necessary in actual practice, it does allow us to speak of a particular edge or node unambiguously. This will be especially important in the proofs of determinacy.

Let the nodes of $\mathcal{P} = G,X$ be uniquely numbered from the set of odd integers $H_1 = \{3,5,\ldots,2n+1\}$; let the edges of $\mathcal{P}$ be numbered from the set $H_2 = \{2n+3, 2n+5,\ldots,2m+1\}$; and let $H_3 = \{2m+3, 2m+5,\ldots\}$ be an ordered set of identifiers to be used with each initiation queue. It is assumed that for each initiation queue, each time the corresponding node in the executing graph is initiated, a new element from the set H_3 is added to the queue. The $j^{\underline{th}}$ initiation of node N will cause the $j^{\underline{th}}$ element of H_3 to be added to IQ_N. These numbers satisfy the uniqueness requirement for the identifiers on IQ_N required by definitions 7 and 8. Any other choice of identifiers can easily be mapped into this set without loss of generality.

The nodes of the executing graph are renumbered as follows: The Gödel number for each node of G is the corresponding number from H_1. Suppose now that gn is the Gödel number assigned to a procedure node P of the executing graph; $i_1,\ldots,i_\ell$ are the numbers of H_1 assigned to the nodes of the graph procedure defining P; and η is the next identifier to be placed on IQ_P. Then the nodes of the executing graph created by the initiation of P are given the Gödel numbers

$$(2^{gn})(3^{\eta})(5^{i_k}),\ 1 \leq k \leq \ell\ .$$

Edges are similarly numbered, the only difference being that the numbers $e_1,\ldots,e_m$ from the set H_2 would be used, giving

$$(2^{gn})(3^{\eta})(5^{e_k}),\ 1 \leq k \leq m\ .$$

This numbering is unique. First, each node of G has been assigned a distinct number, by definition, and each number assigned to a node of G is smaller than the number assigned to any other node in the executing graph. Now suppose that two procedure nodes in the executing graph with Gödel numbers gn_1 and gn_2, respectively, $gn_1 \neq gn_2$, each create a node with Gödel number gn. Then

$$gn = (2^{gn_1})(3^{\eta_1})(5^{j}) = (2^{gn_2})(3^{\eta_2})(5^{k})\ .$$

By the fundamental theorem of arithmetic, each positive integer has a prime factorization which is unique up to order. Hence $gn_1 = gn_2$, contradicting the hypothesis. Furthermore $\eta_1 = \eta_2$ and $j = k$ so that there is no way two distinct nodes in the executing graph may have the same name.

Thus far there has been no notion of time introduced into the model. The length of time required to perform any computation has been left unspecified, and may vary from execution to execution of a given node. We shall use time to mean those instances at which initiations and terminations of node executions occur. More precisely, when we speak of the time at which an initiation or termination takes place, we shall mean the instance at which an initiation or termination is started. It is not assumed that initiations and terminations take place instantaneously. The duration between consecutive time steps is left unspecified. All computations require some finite amount of time to perform, thus it is impossible to both initiate and terminate an execution at the same time. We shall let t denote time and t_k denote the k^{th} instance on our time scale. We are now ready to give a formal definition for the execution of a graph program. Let S_i be a non-empty set of symbols of the form I_k and T_k, where k uniquely identifies a node in the executing graph of $\mathcal{P}$. As was shown above, a Gödel numbering may be used to provide a unique naming of the nodes in the executing graph. Thus k may be assumed to be such a member, although it need not be.

Definition 9. The sequence $\mathcal{E} = S_1, S_2, \ldots, S_m, \ldots$ is an execution of $\mathcal{P}$ if and only if for each S_j in $\mathcal{E}$

i) there is an $I_k \in S_j$ for the execution of each node N_k initiated at time t_j ;

ii) there is a $T_k \in S_j$ for the execution of each node terminated at time t_j .

In addition, if $\mathcal{E}$ is finite with S_m the last set in the sequence, then

for all $t > t_m$, all nodes N in the executing graph, and each initiation queue associated with N, $\{N \mid E^I(N)\} = \varphi$ and $\{IQ \mid n(IQ) > 0\} = \varphi$.

The last condition assures us that an execution $\mathcal{E}$ cannot halt prematurely.

Definition 10. $\mathcal{E}$ is a proper execution of $\mathcal{P}$ if the following conditions also hold for each node N_k in the executing graph of $\mathcal{E}$:

i) If $E^I(N_k)$ at time t_i, then $\exists\ j \geq i$ such that $I_k \in S_j$;

ii) If $E^T(N_k)$ at time t_i, then $\exists\ j \geq i$ such that $T_k \in S_j$.

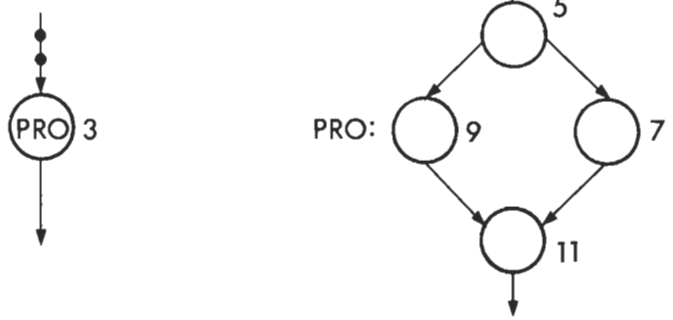
The above definition insures us that any execution of a node which is either ready to initiate or ready to terminate will be initiated or terminated, respectively, within some finite amount of time. If, for example, a program is in an infinite loop, then any node in the executing graph which at some time becomes ready to initiate or terminate will be attended to, regardless of how few or how many processors are available.

The execution of $\mathcal{P}$ can be visualized as a tree which would be constructed as follows:

1. Initially, at t_0, there is only the trunk of the tree.
2. Each node initiated at t_1 causes a branch to be added to the trunk at level 1.
3. At t_k any or all of the following may take place:
   i) each node execution which was initiated earlier and continues to execute causes the corresponding branch of the tree to be extended to level k+1.
   ii) each execution which terminates causes the corresponding branch of the tree to be terminated.
   iii) each execution which is initiated caused a new branch to be

added to the tree. The branch is added to the trunk of the tree if the initiated node is a node of G. Otherwise, the initiated node belongs to a graph procedure which was created when a procedure node P was initiated, and the new branch is added to the branch of the tree corresponding to the initiation of P.

The following example will illustrate the execution of a graph program. Let $\mathcal{P}$ be given by a one node main program and one procedure, and suppose that initially 2 elements of data are placed on the edge directed into the node numbered 3.



The Gödel numbering of nodes in the executing graph will be illustrated. The numbers beside the nodes form the set H_1. Assume the nodes numbered 5, 7, 9 and 11 are primitive nodes, each putting out one element on each output edge when it executes. $H_2 = \{13, 15, 17, 19, 21, 23, 25, 27\}$ and $H_3 = \{29, 31, \ldots\}$.

One possible execution sequence might be

$$\mathcal{E} = \{I_3\}\ ,\ \{I_3\ ,\ I_{2^3.3^{29}.5^5}\}\ ,$$

$$\{T_{2^3.3^{29}.5^5}\ ,\ I_{2^3.3^{31}.5^5}\}\ ,$$

$$\{T_{2^3.3^{31}.5^5}\ ,\ I_{2^3.3^{29}.5^7}\ ,\ I_{2^3.3^{29}.5^9}\}\ ,$$

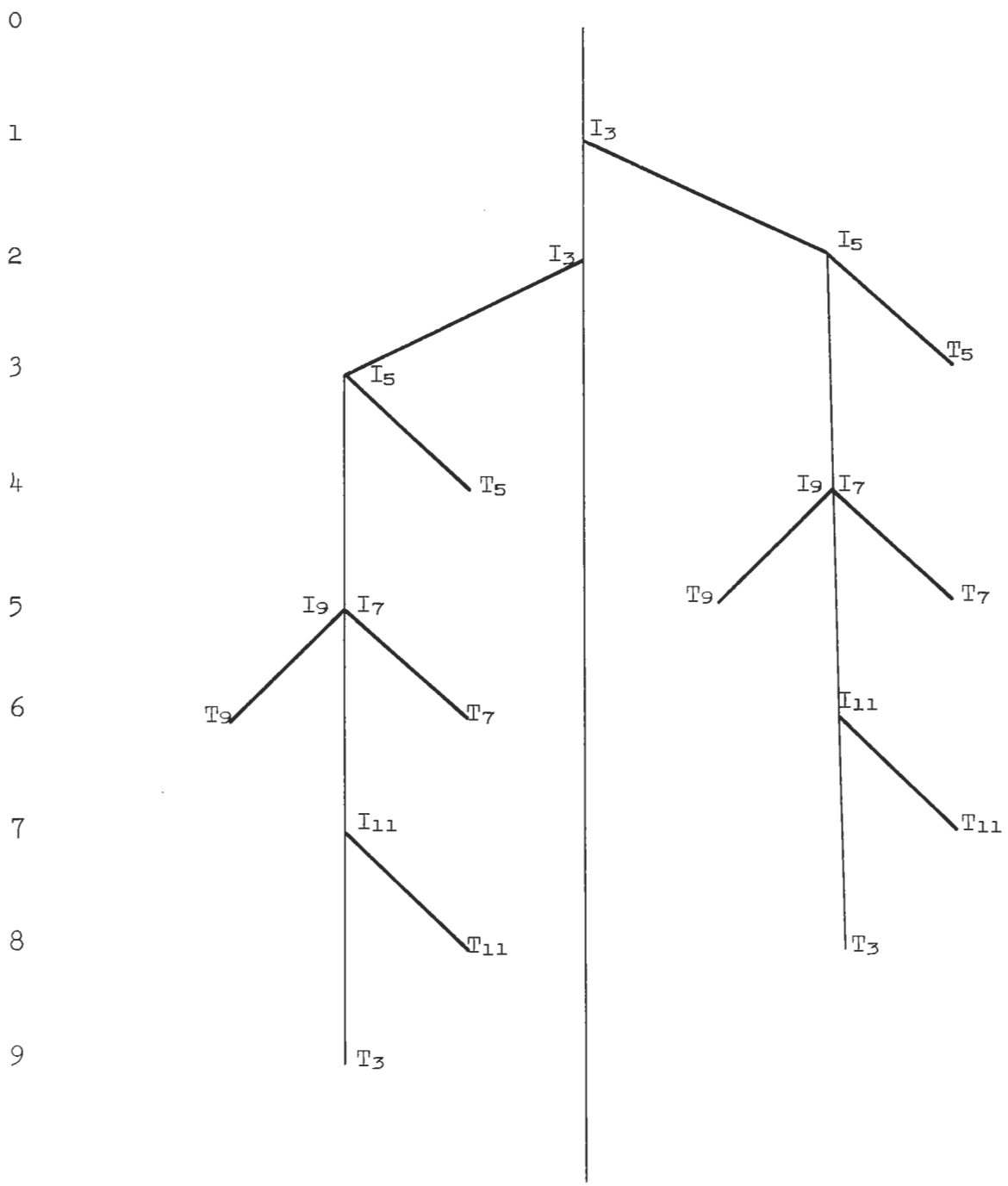
$$\{I_{2^3.3^{31}.5^7}\ ,\ I_{2^3.3^{31}.5^9}\ ,\ T_{2^3.3^{29}.5^7}\ ,\ T_{2^3.3^{29}.5^9}\}\ ,$$

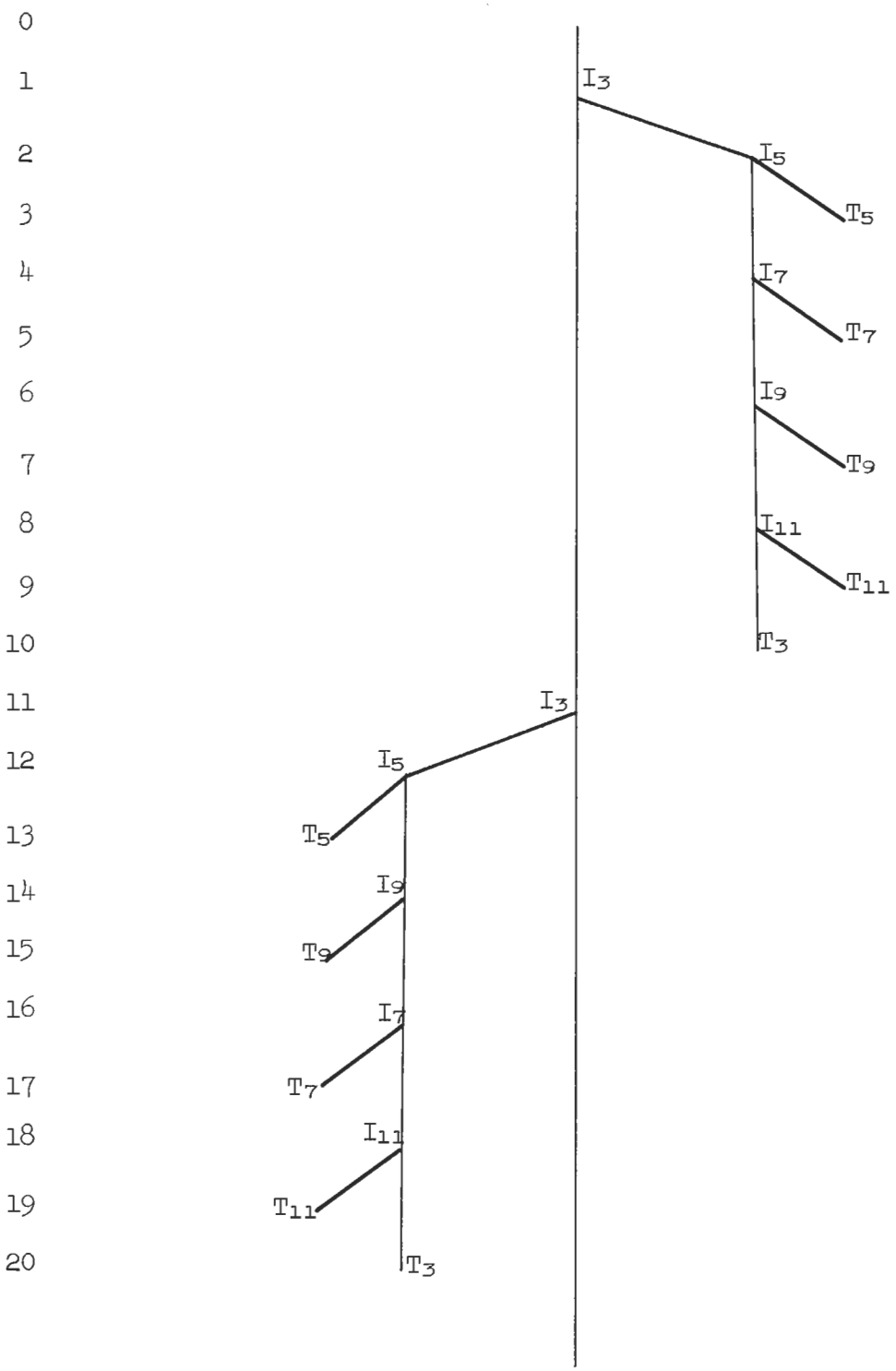
$$\{I_{2^3.3^{29}.5^{11}}\ ,\ T_{2^3.3^{31}.5^7}\ ,\ T_{2^3.3^{31}.5^9}\}\ ,$$

$$\{T_{2^3.3^{29}.5^{11}}\ ,\ I_{2^3.3^{31}.5^{11}}\}\ ,\ \{T_{2^3.3^{31}.5^{11}}\ ,\ T_3\}\ ,$$

$$\{T_3\}\ .$$

For the tree, we don't need to use a renumbering, since the structure of the tree uniquely distinguishes nodes having the same name. The previous execution would be given by the tree shown on p. 36. Another possible execution of this same program in which only one initiation or termination occurs at each time step is given by the tree on p. 37. This execution represents the maximum possible length of $\mathcal{E}$ for this problem, since each set S_i in the sequence $\mathcal{E}$ must be non-null. Note that there are numerous executions of this length possible for this problem. A graph program for which there is only one execution sequence will be termed a <u>sequential program</u>.


0
1
2
3
4
5
6
7
8
9
I3
I3
I5
T5
I5
T5
I9
I7
T9
T7
I9
I7
T9
T7
I11
T11
I11
T11
T3
T3


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
I_3
I_5
T_5
I_7
T_7
I_9
T_9
I_{11}
T_{11}
T_3
I_3
I_5
T_5
I_9
T_9
I_7
T_7
I_{11}
T_{11}
T_3

IV. PROPERTIES OF THE MODEL

In this section it is proved that all graph programs in the model M are determinate, and that all computable functions can be represented within the model.

1. Determinacy

During the execution of a graph program $\mathcal{P}$, an auxiliary graph referred to as the executing graph is constructed. When a procedure node is initiated, a copy of the defining graph procedure is created and becomes a part of the executing graph. When a procedure node is terminated, the defining graph procedure is deleted. Thus, the extent of the executing graph varies dynamically during the course of the execution. If $\mathcal{E}$ and $\mathcal{E}'$ are executions of $\mathcal{P}$, the resulting executing graphs may be different. Thus, we will associate an executing graph for a graph program with a particular execution. We will assume, without further explicit mention of it, that the nodes and edges of the executing graph are uniquely numbered independently of the order in which they are created, and that the same numbering is used for each execution of $\mathcal{P}$. The Gödel numbering described earlier is such a numbering. However, in order to simplify the notation, we will distinguish different nodes by distinct names, and assume that any given name refers to the same node or edge in each execution.

The definition of determinacy and the theorems we will prove regarding determinacy of programs within the model require that we be able to compare two executions $\mathcal{E}$ and $\mathcal{E}'$ of a graph program $\mathcal{P}$. To do so, we shall introduce the notion of an edge history. The edge history of an edge e in the

executing graph of $\mathcal{E}$ is the sequence of values placed on edge e during the execution of the program. It also includes any initial data placed on e. We shall use the notation V_{e_j} to denote the j^{th} value in the edge history of e. Note that a value refers to a distinct element on a queue, regardless of the complexity of its structure.

Definition 11. A graph program $\mathcal{P}$ is said to be determinate if for any proper executions $\mathcal{E}$ and $\mathcal{E}'$, the edge histories for each edge in the executing graphs of $\mathcal{E}$ and $\mathcal{E}'$ are identical.

The definition for determinacy is trivially satisfied for a sequential program, since there is only one possible execution. Note that this is a strong form of determinacy. For example, it does not allow the order of elements on an edge to be varied even when the order does not affect the final results.

The following notation will be used in the theorems. Let I(N,j) and I'(N,j) denote the number of times I_N has appeared in the execution sequences $\mathcal{E}$ and $\mathcal{E}'$, respectively, up to and including time t_j. Let T(N,j) and T'(N,j) denote the number of times T_N has appeared in $\mathcal{E}$ and $\mathcal{E}'$, respectively, up to and including time t_j; and let X(e,j) and X'(e,j) denote the total number of data elements placed on edge e in $\mathcal{E}$ and $\mathcal{E}'$, respectively, up to and including time t_j.

Lemma 1. Let $\mathcal{P}$ be a graph program, $\mathcal{E}$ and $\mathcal{E}'$ executions of $\mathcal{P}$, and N a node which appears in the executing graphs for $\mathcal{E}$ and $\mathcal{E}'$. Suppose further that the edge histories for each edge e directed into N are identical in $\mathcal{E}$ and $\mathcal{E}'$. Then, if there is an i^{th} initiation of N in both $\mathcal{E}$ and $\mathcal{E}'$,

$i=1,2,\ldots$, the i^{th} ordered set of operands for N in $\mathcal{E}$ is identical with the i^{th} ordered set of operands for N in $\mathcal{E}'$.

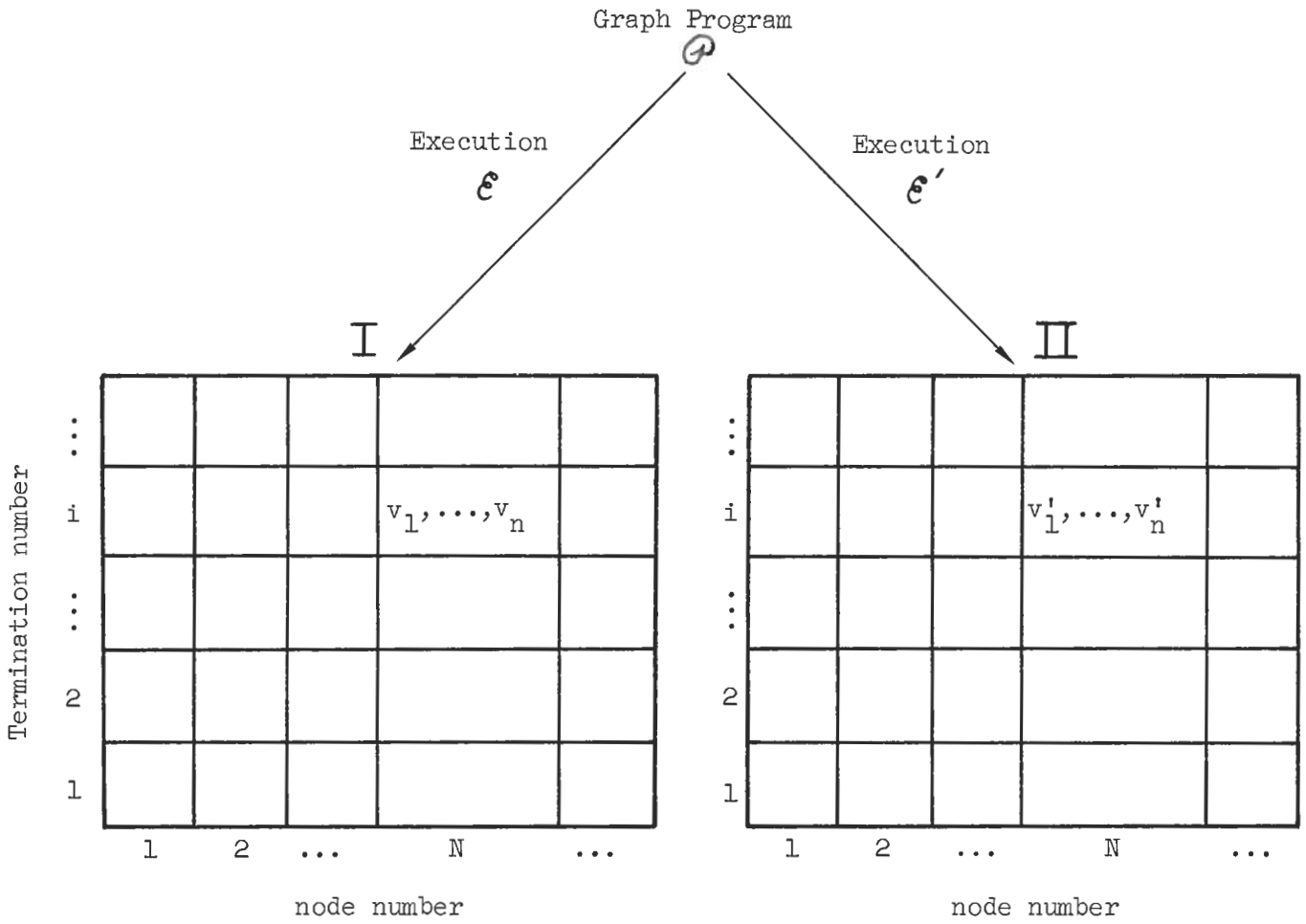
Proof: If N is an r-node or a procedure node, the lemma is true since an execution of N cannot be initiated until there is at least one element present on each edge directed into N, and when it is initiated, exactly one element is removed from each such edge. Now suppose N is an s-node. The initial status settings for N are the same in $\mathcal{E}$ and $\mathcal{E}'$, and when an execution of N is initiated, one element is removed from each unlocked input. Thus the first initiation of N in $\mathcal{E}$ and $\mathcal{E}'$ has the same ordered set of operands. Suppose now that the j^{th} initiation of N in $\mathcal{E}$ and $\mathcal{E}'$ is the first initiation of N where the ordered set of operands differ. Then the ordered set of operands for the $(j-1)^{st}$ initiation are identical, and hence so are the status settings, since the only way φ may be an operand is for the corresponding status to be locked. The conditions for initiation of an s-node require that $n(IQ_N) = 0$, hence the j^{th} execution of N can't be initiated until the $(j-1)^{st}$ execution terminates. g is a single-valued function so the status settings for the j^{th} initiation in $\mathcal{E}$ and $\mathcal{E}'$ are identical, and since the sequence of values on each edge is the same, the j^{th} initiation of N in $\mathcal{E}$ and $\mathcal{E}'$ will have the same ordered set of operands.

Theorem 1. Let $\mathcal{G}$ be a graph program, $\mathcal{E} = S_1, S_2, \ldots$ and $\mathcal{E}' = S_1', S_2', \ldots$ be executions of $\mathcal{G}$, and N an arbitrary node in the executing graphs for both $\mathcal{E}$ and $\mathcal{E}'$. If there is an i^{th} T_N in both $\mathcal{E}$ and $\mathcal{E}'$, $i=1,2,\ldots$, then $(v_1,\ldots,v_n)=(v_1',\ldots,v_n')$ where $(v_1,\ldots,v_n)$ is the i^{th} output of N in $\mathcal{E}$ and $(v_1',\ldots,v_n')$ is the i^{th} output of N in $\mathcal{E}'$.

Comment -- The notation $(v_1,\ldots,v_n)$ is used in the same way as it was in definition 1. Each v_j, $1 \leq j \leq n$, is the output placed on an edge directed from N and may be a single element, a sequence of elements of the null element φ. The figure on page 42 is another way of viewing the statement of theorem 1. The columns of the matrices labeled I and II represent the nodes in the executing graphs of $\mathcal{E}$ and $\mathcal{E}'$, respectively. Note the renumbering. An entry in the matrix at row i, column N denotes the output produced at the i^{th} termination of node N. The order in which the entries are placed in I and II may differ for $\mathcal{E} \neq \mathcal{E}'$, and, in fact, for an entry at i,N in I there need not be an entry at i,N in II. This could happen, for example, when there is an infinite loop in the computation and there was an execution of some node N in $\mathcal{E}$ for which there was no corresponding execution in $\mathcal{E}'$. However, the theorem states that whenever there is an entry in corresponding locations of I and II, that such entries will be identical.

We shall use the term corresponding values to refer to the values V_{ej} and V'_{ej}, where V_{ej} is the j^{th} value in the edge history for edge e in execution $\mathcal{E}$, and V'_{ej} is the j^{th} value in the edge history for e in $\mathcal{E}'$. In the event that only V_{ej}, say, is defined, then we shall say there is no value in $\mathcal{E}'$ corresponding to V_{ej} in $\mathcal{E}$. When we speak of the edge histories for e in $\mathcal{E}$ and $\mathcal{E}'$ which have been formed up to some time t_k, then there may be cases where a value has been defined for one execution but not the other. We shall also speak of the corresponding outputs for some node N and the corresponding terminations of an execution, and they are to be interpreted in a manner similar to corresponding values.

Proof of theorem 1: The proof is by contradiction. For each $S_j \epsilon \mathcal{E}$, starting with j=2, and for each T_N in S_j, we compare the output from N with the



Entries in the matrices I and II denote the output produced when the i^{th} execution of node N terminates in $\mathcal{E}$ and $\mathcal{E}'$, respectively, i=1,2,..., N=1,2,... .

output for the corresponding T_N in $\mathcal{E}'$, provided such a T_N exists in $\mathcal{E}'$. Let t_m denote the earliest time in $\mathcal{E}$ for which the output from the $(T(N_k,m))^{th}$ termination of some node N_k differs in $\mathcal{E}$ and $\mathcal{E}'$. Thus, for the edges directed into N_k, the values in the edge histories at t_{m-1} in $\mathcal{E}$ must be identical with the corresponding values in the edge histories in $\mathcal{E}'$.

Case 1. N_k is a primitive node.

By lemma 1, each execution of N_k initiated at $t \leq t_{m-1}$ in $\mathcal{E}$ had the same ordered set of operands as the corresponding initiation in $\mathcal{E}'$. The initiation queue for N_k insures that for both $\mathcal{E}$ and $\mathcal{E}'$, terminations occur in the same order as the initiations. Finally, since the function f defining N_k is single-valued, the output from the $(T(N_k,m))^{th}$ termination of N_k in $\mathcal{E}$ must be identical with the corresponding output of N_k in $\mathcal{E}'$, contrary to hypothesis.

Case 2. N_k is a procedure node.

Let $i = T(N_k,m)$. The i^{th} initiation of N_k in both $\mathcal{E}$ and $\mathcal{E}'$ creates an identical copy of the graph procedure defining N_k, which we denote by G. Since the i^{th} initiation of N_k in $\mathcal{E}$ occurred at $t < t_m$, the operands placed on the input set of G will be the same as for the i^{th} initiation of N_k in $\mathcal{E}'$. In $\mathcal{E}$, all executions of nodes in G must terminate at $t < t_m$, since all such nodes must have terminated before the procedure node N_k may terminate; and by the minimality of t_m, corresponding terminations in $\mathcal{E}'$ must produce identical outputs. Furthermore, the initiation queue for N_k insures us that each execution of N_k in $\mathcal{E}$ and $\mathcal{E}'$ must terminate in the same order as initiated. Thus, the outputs from N_k in $\mathcal{E}$ and $\mathcal{E}'$ can only differ if the number of executions for each node of G is not the same.

Suppose now that for some node N_o in G and some time t_j in $\mathcal{E}$ that

$$T(N_o,j) > T'(N_o,m') \quad (1)$$

where

$t_{m'}$ is the time the i^{th} termination of N_k occurs in $\mathcal{E}'$ and

t_j is the earliest time in $\mathcal{E}$ when (1) is true for any node in G.

By the minimality of t_j, there exists an $\ell \leq m'$ such that

$$T'(N,\ell) \geq T(N,j-1) \quad (2)$$

for all nodes N in G. Now $t_{j-1} < t_m$ since in execution $\mathcal{E}$, all executions of nodes in G must have terminated before the procedure node N_k can terminate; hence, all terminations in $\mathcal{E}$ at $t \leq t_{j-1}$ produce outputs which are identical with those produced by corresponding terminations in $\mathcal{E}'$. This, combined with (2) gives

$$X'(e,\ell) \geq X(e,j-1) \quad (3)$$

for all edges in e and G. In particular, it is true for each edge directed into N_o. The i^{th} execution of N_k in $\mathcal{E}'$ can only terminate when there are no nodes of G waiting to initiate or which have been initiated but not terminated. Hence,

$$T'(N_o,m') \geq T(N_o,j) \quad (4)$$

contrary to the assumption in (1). Similarly, there can't be fewer executions in $\mathcal{E}$ than in $\mathcal{E}'$ for any node N of G. Thus the i^{th} termination of N_k in $\mathcal{E}$ and $\mathcal{E}'$ must produce identical outputs on each edge directed out from N_k, and the theorem is proved.

Corollary 1. Let V_{ej} denote the j^{th} value in the edge history for edge e in $\mathcal{E}$ and let V'_{ej} be similarly defined for $\mathcal{E}'$. For each edge e in the executing graphs of $\mathcal{E}$ and $\mathcal{E}'$, if there is a j^{th} value placed on e in both executions, then $V_{ej} = V'_{ej}$.

Proof: Since corresponding node terminations in $\mathcal{E}$ and $\mathcal{E}'$ result in identical outputs, we obtain, by renumbering the non-null outputs placed on each edge, that corresponding values in the edge histories must also be identical.

Note that we have not yet proved that the entire edge histories are identical since we are not assured that each node in the executing graph for $\mathcal{E}$ will have executed the same number of times as the corresponding node in $\mathcal{E}'$. If the computation terminates, however, then an execution is the same as a proper execution and corresponding edge histories will be identical.

Corollary 2. Let $\mathcal{E}$ and $\mathcal{E}'$ be executions of $\mathcal{P}$. Then $\mathcal{E}'$ is finite if and only if $\mathcal{E}$ is finite.

Proof: Suppose $\mathcal{E}$ is finite with S_m the last set in the sequence, and suppose $\mathcal{E}'$ is infinite. Let t_j denote the first initiation of a node, say N_o, in $\mathcal{E}'$ for which there is no corresponding initiation of N_o in $\mathcal{E}$. Since $\mathcal{E}$ is finite, all initiated executions must terminate, hence $T(N,m) \geq T'(N,j-1)$ for all nodes N in the executing graphs of $\mathcal{E}$ and $\mathcal{E}'$. Consequently $X(e,m) \geq X'(e,j-1)$ for all edges e directed into N_o. Corresponding initiations of N_o in $\mathcal{E}$ and $\mathcal{E}'$ have the same ordered set of operands, and since N_o is not ready to initiate at any $t \geq t_m$ in $\mathcal{E}$, there can be no initiation of N_o at t_j in $\mathcal{E}'$, and it is finite.

If it is assumed that $\mathcal{E}$ is infinite but that $\mathcal{E}'$ is finite it is similarly established that there can be no last set S'_m in $\mathcal{E}'$, and hence it also must be infinite.

Corollary 3. Let P be a procedure node in the executing graphs of $\mathcal{E}$ and $\mathcal{E}'$, and suppose that an execution of P terminates in $\mathcal{E}$. Then for each node N created during a corresponding execution of P in $\mathcal{E}'$, the number of terminations of N in $\mathcal{E}'$ cannot exceed the number of terminations of N in $\mathcal{E}$, and if the execution of P terminates in $\mathcal{E}'$, the number of executions of N will be identical.

Theorem 2. Let $\mathcal{P}$ be a graph program, let $\mathcal{E}$ and $\mathcal{E}'$ be proper executions of $\mathcal{P}$, and let N be an arbitrary node in the executing graph of $\mathcal{E}$. Then for each S_j in $\mathcal{E}$ there exists an S'_i in $\mathcal{E}'$ such that

$$I(N,j) = I'(n,i) ,$$

and for each S_j in $\mathcal{E}$ there is an S'_ℓ in $\mathcal{E}'$ such that

$$T(N,j) = T'(N,\ell) .$$

Proof: Suppose the contrary, and let t_j denote the earliest time such that for some node N_o in the executing graph of $\mathcal{E}$, one of the following is true:

(1) $$I(N_o,j) > I'(N_o,m) \quad \text{for all } S_m \text{ in } \mathcal{E}'$$

(2) $$T(N_o,j) > T'(N_o,m) \quad \text{for all } S_m \text{ in } \mathcal{E}' .$$

Since t_j is the first time that either (1) or (2) is assumed to be true, then there is a k' such that $I'(N,k') \geq I(N,j-1)$ and also a k" such that $T'(N,k") \geq T(N,j-1)$ for all nodes N in the executing graph of $\mathcal{E}$. Let $k = \max(k',k")$. Then

(3) $$I'(N,k) \geq I(N,j-1) \qquad \underline{\text{and}}$$

(4) $$T'(N,k) \geq T(N,j-1)$$

for all nodes N in the executing graph of $\mathcal{E}$.

Case 1. Suppose (1) is true.

We note first of all that the node N_o must be in the executing graph for $\mathcal{E}'$, for if N_o was created in $\mathcal{E}$ by the initiation of a procedure node N_1 at $t \leq t_{j-1}$, we have by (3) that N_1 would have been initiated in $\mathcal{E}'$ at $t \leq t_k$. For each edge e directed into N_o and for each value placed on e in $\mathcal{E}$ at $t \leq t_{j-1}$, there is a corresponding and identical value placed on e in $\mathcal{E}'$ at $t \leq t_k$. This follows from (3) for any edge e in the input set of a graph procedure, otherwise it follows from (4) and theorem 1. Thus, for the execution of N_o initiated at t_j in $\mathcal{E}$, there is a corresponding execution of N_o in $\mathcal{E}'$ which is ready for initiation at t_{k+1}. Since $\mathcal{E}'$ is a proper execution there exists an $i > k$ such that $I_{N_o} \in S'_i$. Hence $I(N_o,j) = I'(N_o,i)$, contrary to (1).

Case 2. Suppose (2) is true.

Each node which terminates at t_j in $\mathcal{E}$ must have been initiated at $t \leq t_{j-1}$, and by (3) for each initiation at $t \leq t_{j-1}$ in $\mathcal{E}$ there is a corresponding initiation at $t \leq t_k$ in $\mathcal{E}'$. In particular this is true for N_o. If $n = I(N_o,j-1)$, there is also an n^{th} initiation of N_o at $t \leq t_k$ in $\mathcal{E}'$.

a. Suppose N_o is a primitive node.

N_o will be ready for termination when the processor assigned to N_o realizes the defining function, and since $\mathcal{E}'$ is a proper execution, there is some $\ell > k$ such that $T_{N_o} \in S_\ell$. Hence $T'(N_o,\ell) = T(N_o,j)$, contrary to the assumption in (2).

b. Suppose N_o is a procedure node.

Since the n^{th} execution of N_o is ready for termination at t_j in $\mathcal{E}$, then at t_{j-1} all node executions in the defining graph procedure must have

terminated, with no further nodes ready for initiation. But by (4) and corollary 3, similar conditions hold at t_k for the n^{th} execution of N_o in $\mathcal{E}'$. Also by (4), at t_k the n^{th} execution of N_o is the next execution of N_o to be terminated in $\mathcal{E}'$. Since $\mathcal{E}'$ is a proper execution there exists an $\ell > k$ such that $T_{N_o} \in S'_\ell$, and hence $T'(N_o,\ell) = T(N_o,j)$, contrary to (2). This proves the theorem.

Corollary 4. Let $\mathcal{E}$ and $\mathcal{E}'$ be proper executions of $\mathcal{G}$. Then each node N appearing in the executing graph for $\mathcal{E}$ will also appear in the executing graph for $\mathcal{E}'$.

Theorem 3. Each graph program in the model M is determinate.

Proof: This theorem follows immediately from theorems 1 and 2 by renumbering the non-null elements placed on each edge of the executing graphs for the graph program, thus obtaining identical edge histories for corresponding edges in the executing graphs.

2. Universal Computing Capability within M

There are several formulations of what is meant by a function being effectively computable, and all have been shown to be equivalent [8]. Turing proposed that the effectively computable functions could be performed by a class of abstract machines now called Turing machines. We shall show that any computation which can be performed on a Turing machine may be performed by a graph program within the model M. This proof will not provide much of an indication as to how programs should be written within the model, but rather will show that with the properly chosen primitive nodes

we can expect to represent all computable functions within the model. The interesting question then is whether or not a particular set of primitive nodes is both useful from a programmer's point of view and complete in terms of being able to represent all computable functions. This question will be discussed in a later report.

Suppose that a Turing machine is represented as a finite set of quintuples of the form

$$q_i S_j S_k \left|\begin{matrix} L \\ R \end{matrix}\right| q_\ell$$

where $q_1,\ldots,q_m$ are the internal states and $S_o,\ldots,S_n$ are the tape symbols. a machine in state q_i scanning the symbol S_j would erase S_j and write the symbol S_k, move either left or right one position, and change to state q_ℓ. The machine halts if and only if for any $q_i S_j$ there is no specification of $S_k \left|\begin{matrix} L \\ R \end{matrix}\right| q_\ell$. For a given Turing machine we have the following data type grammar:

$$V_T = \{[,],q_1,\ldots,q_m,S_o,\ldots,S_n,0,1\}$$

$$V_N = \{C, ST, AL, SC, LH, RH, TA\}$$

C → 0 | 1

ST → $q_1 \mid q_2 \mid \ldots \mid q_m$

AL → $S_o \mid S_1 \mid \ldots \mid S_n$

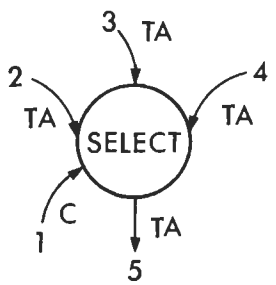
SC → ST AL

LH → $[AL^*]$

RH → $[AL^*]$

TA → LH SC RH

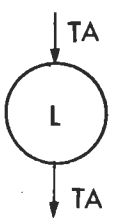
SC denotes the current state of the machine and tape symbol being scanned. LH and RH denote, respectively, the tape to the left of the scanned symbol and the tape to the right of the scanned symbol. When LH is given by [ ], it is to be interpreted that all symbols to the left of the scanned cell are blank. RH has a similar interpretation. There are four primitive nodes, performing the following operations:



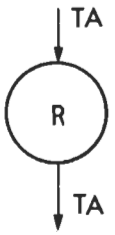
Only one input for the SELECT node is unlocked at a time. When an element appears on edge 2, 3, or 4, it is routed onto edge 5. An element on edge 1 is a control which causes either edge 2 or 4 to be unlocked. The operation of the SELECT node is summarized below.

g: LLUL → ULLL
ULLL → LULL if $v_1 = 0$
ULLL → LLLU if $v_1 = 1$
LULL → ULLL
LLLU → ULLL

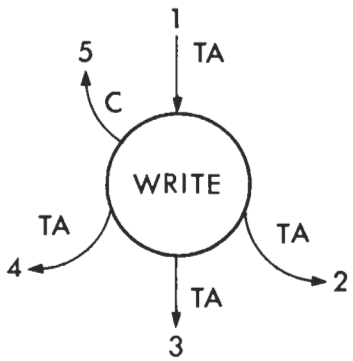
f: $v_5 \leftarrow v_3$
$v_5 \leftarrow \varphi$
$v_5 \leftarrow \varphi$
$v_5 \leftarrow v_2$
$v_5 \leftarrow v_4$



The operation of this node corresponds to moving the reading head of the Turing machine left one square. The AL of SC is added onto RH as the first element. The last element of LH replaces the AL of SC, if LH is non-null. Otherwise S_o replaces the AL of SC.



The operation of this node corresponds to moving the reading head right one square. The AL of SC is added onto LH as the last element. The first element of RH replaces the AL of SC, if RH is non-null. Otherwise S_o replaces the AL of SC.



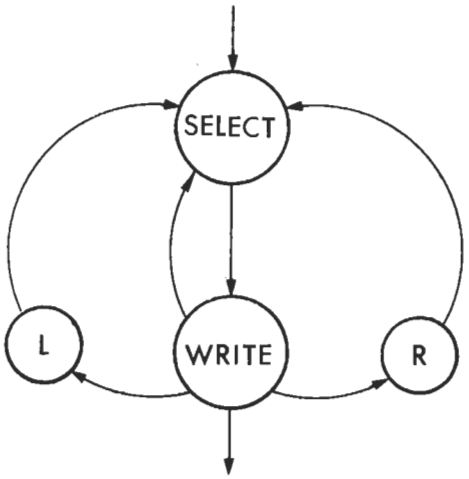
The WRITE node accepts an element of type TA and either 1) changes the state-scanned symbol pair in SC; routes the resulting element to 4 or 2; and places a 0 or 1 on edge 5, depending on whether it was routed to 2 or 4, or 2) routes the input directly onto edge 3.

The following matrix determines the operation performed by the WRITE node. It corresponds to the functional matrix of a Turing machine.

f:

	S_o	S_1	...	S_j	...	S_n
q_1						
q_2						
⋮						
q_i				$S_k \left\lvert \begin{matrix} L \\ R \end{matrix} \right\rvert q_\ell$		
⋮						
q_m						

The graph program to perform the Turing machine computation is given by



The element initially placed on edge 3 into the SELECT node corresponds to the initial tape configuration and initial state of the Turing machine. It would be of the form TA = LH SC RH, where

LH = [ ], corresponding to a blank tape to the left of the initial symbol,

SC = initial-state initial-symbol-on-tape,

RH = [remainder of initial tape symbols] .

The initial status settings for the SELECT node are LLUL. There is only one data element in the computation, and it is routed from node to node. Each arrival of the element back at the SELECT node corresponds to the Turing machine having moved from one instantaneous description to the next. The computation halts if the data is ever placed on edge 3 of the write node.

We could have chosen primitive nodes which perform more elementary functions than the ones we did choose. It would be of interest to know what constitutes a sufficient set of more primitive nodes.

V. REFERENCES

1. Karp, R. M. and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Appl. Math. 14 (Nov. 1966), 1390-1411.

2. Karp, R. M. and Miller, R. E. Parallel program schemata: a mathematical model for parallel computation. presented at Eighth Annual Symposium on Switching and Automata Theory (Oct. 1967).

3. Martin, D. F. The automatic assignment and sequencing of computations on parallel processor systems. UCLA Report No. 66-4 (Jan. 1966).

4. Rodriguez, Jorge E. A graph model for parallel computations. MIT Ph.D. thesis (Sept. 1967).

5. Sutherland, W. R. On-line graphic specification of computer procedures. Lincoln Laboratory Technical Report 405 (May 1966).

6. Van Horn, E. C., Jr. Computer design for asynchronously reproducible multiprocessing. MAC-TR-34 (Nov. 1966).

7. Estrin, G. and Turn, R. Automatic assignment of computations in a variable structure computer system. IEEE Transactions on Electronic Computers, EC-12, No. 5 (Dec. 1963) p. 755-773.

8. Mendelson, E. Introduction to mathematical logic. Van Nostrand, Princeton, 1964.