HOPKINS COMPUTER RESEARCH REPORTS

REPORT #21

AUGUST 1972

STRUCTURAL PROPERTIES OF CONCURRENT PROCESSES

BY

RENAUD REGIS

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE

COMPUTER SCIENCE PROGRAM

THE JOHNS HOPKINS UNIVERSITY

BALTIMORE, MARYLAND

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

STRUCTURAL PROPERTIES OF CONCURRENT PROCESSES*

by Renaud Regis

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Mailing address:

Dr. R. C. Regis
c/o Mrs. Carol Rogers
Computer Science Dept.
The Johns Hopkins University
Baltimore, Maryland 21218

*This work supported in part by the U. S. Atomic Energy Commission under contract AT (11-1) 3288.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

leg

ABSTRACT

Characteristic of systems of concurrent processing activities operating in a common environment is their potential for non-deterministic (irreproducible) behavior. The notion of deterministic behavior is formalized in the context of a graph model of concurrent processes. In addition to complete determinacy, two less restrictive properties, partial and terminal determinacy, are introduced. Their decidability is constructively proved for acyclic graphs. Two extensions of the model are considered: recursively specified concurrent processes and mutually exclusive processes. With regard to the latter, it is suggested that they would be best studied in the context of partially interpreted models.

1. Introduction

A characteristic of asynchronous processing activities operating in a common environment is the potential for nondeterministic behavior. Since, by definition, time is a local relation, the results of two asynchronous processes may depend on the order of their execution.

This paper is concerned with characterizing the determinacy (deterministic behavior) of concurrent processes*. More precisely, an information structure for representing concurrent processes, called the Process Graph Model, is presented. Several properties of their dynamic behavior are defined and shown to be decidable.

The notion of deterministic behavior has been formalized into several models of concurrent processes. Karp and Miller [1] proposed a very general model, the parallel program schema, in which elementary operations communicate through and share some variables. The decidability of determinacy and equivalence of schema were the primary concern of the authors.

On the other hand, graph models explicitly represent the control structure of concurrent processes (whereas it is implicitly defined by a central transition function in the works along Karp and Miller's line). In Adams [2] and Rodriguez [3] models, arcs between computation nodes also represent data transmission lines (the model's storage locations); but sharing of variables is implicitly denied, thereby restricting the communication between elementary processes.

* By "concurrent process" we mean a process of which several parts can be executed concurrently (asynchronously).

The Process Graph Model is based on a graph model developed at UCLA by Martin, Estrin, et. al. ([4], [5]) to study processor assignment and memory allocation in multiprocessing systems and the automatic translation of sequential programs into parallel programs*. It differs from Adams and Rodriguez models in that storage locations can be shared by several elementary processes. Our primary concern in this paper will be to determine under which conditions this sharing of variables results in a deterministic behavior. It is similar, in that respect, to Karp and Miller's work.

Yet, in contrast with any other model of concurrent processes, we shall be interested in weaker, less restrictive forms of determinacy, called partial and terminal determinacy, and discuss their decidability.

Independently, Gostelow [6] recently completed a study of an essentially similar model. His main interest was to establish sufficient conditions for the proper termination of graphs, thereby complementing our results on partial and terminal determinacy.**

2. Graph Model of Concurrent Processes.

By the very nature of a concurrent process, time is only a local relation among elementary processes. It is therefore natural to express this partial ordering (or precedence) by a directed graph in which the nodes represent the elementary processes and the arcs, the local precedence. Yet, another important procedural aspect of a concurrent process is the specification of the objects "shared" (operated upon) by several elementary

*Rather than develop yet another graph model, it was felt more interesting to investigate the formal properties of a model which had been studied mainly and extensively from a practical point of view.

** Proper termination and determinacy are two independent properties in the sense that neither is a necessary nor a sufficient condition for the other.

processes and on behalf of which they should be synchronized. But, as far as the procedural (control) specification is concerned, the nature of these objects, as well as the functions performed on them, are irrelevant. What is needed is a set of identifiers to uniquely reference the objects. The notion of process graph, to be defined now, embeds these procedural aspects while the notion of interpretation provides the functional specification of a concurrent process.

2.1 Definitions.

A directed graph (N,A) consists of a set N of nodes, and a set A of ordered pairs of nodes, called arcs. The arc $(x,y) \in A$ is said to be incident from x and incident into y . A path from x_1 to x_n is an ordered sequence of nodes $\{x_1, \dots, x_i, \dots, x_n\}$ such that $(x_i, x_{i+1}) \in A$. Node y is an immediate successor of node x if and only if the arc $(x,y) \in A$; it is a successor of node x if there is a path from x to y . Similarly, node y is an immediate predecessor of node x if and only if the arc $(y,x) \in A$; it is a predecessor of node x if there is a path from y to x .

A process graph p is specified by:

- a directed graph (N,A)
- for every node $x \in N$, four attributes :
 - . an input logic
 - . an output logic
 - . a domain $D(x)$
 - . a range $R(x)$.

The input logic of a node specifies under which conditions (in logical time) the elementary process represented by the node can be initiated.

We call t_e the instant in physical time when these conditions are satisfied and say that an activation of the node is defined.

No assumption will be made on the time elapsed between definition and actual initiation of the process: it depends on the number of processing units, their relative speeds, the state of other processes etc. In fact, we are precisely looking for concurrent processes which exhibit a determinate behavior, irrespective of these quantities. The only assumption is that once a process has been defined, it will eventually be executed (finite delay property). This property is necessary because control is permissive rather than imperative. Similarly, execution times are assumed to be finite.

We shall assume, unless otherwise specified, that once a process has been defined, it remains defined until it is actually executed.

When a node terminates (t_f denotes that instant), its output logic determines which of its successors should be notified.

The input logic of a node is either:

- . conjunctive (denoted by *): the node is defined when all its immediate predecessors have terminated.
- . inclusively disjunctive (denoted by +): the node is defined as soon as any one of its immediate predecessors has terminated; we assume that such a node may not be executed more than once.

Similarly, the output logic of a node is either:

- . conjunctive: upon completion, all its immediate successors are notified.
- . exclusively disjunctive (denoted by $\oplus$): upon completion, a unique successor is selected.

Thus, a $\oplus$ - output node indicates a decision. Each arc incident from a $\oplus$ -output node is uniquely labeled with a decision number. By convention, a *-output node has a unique decision number. Clearly, any positive input or output logic can be realized from the four types of logic previously defined. In particular, the concept of multi-headed and multi-tailed arcs (called complex graphs in [6]) is not strictly necessary. These re-

marks greatly simplify the theoretical treatment of process graphs, yet without loss of generality.

The domain $D(x)$ of a node x is a set of object identifiers, called variables, which specify the arguments of the elementary process, while the range $R(x)$ specifies, through their identifiers, its results.

Finally P is assumed to have a unique initial node (a node with no incoming arcs) and a unique terminal node (a node with no outgoing arcs). An activation of the concurrent process P is initiated by enabling the initial node. It is assumed for now that only one activation of a concurrent process is executing at anytime. Figure 1 depicts a typical process graph.

2.2 Interpretation of a process graph.

A process graph provides a specification of the control structure of a concurrent process. The functional information is provided by the interpretation of the process graph.

Definition.

An interpretation of a process graph is specified by:

- for each variable i , a set of values $V(i)$.

- for each node x :

- . a function $f: \prod_{i \in D(x)} V(i) \rightarrow \prod_{i \in R(x)} V(i)$ and

- . a function $g: \prod_{i \in D(x)} V(i) \rightarrow \{\text{decision numbers of } x\}$

where X denotes the cartesian product of sets.

The function f realizes a mapping from argument values to result values, while g selects one of several possible outcomes.

Note that a node with empty domain does not realize any function.

We will assume that if $D(x) = \emptyset$ then $R(x) = \emptyset$. Clearly, such a node cannot have a $\emptyset$ -output. This is a "dummy" node.

Thus, an interpreted process graph uniquely and completely specifies a concurrent process and defines a mapping $P_I: \prod_{x \in N} D(x) \rightarrow \prod_{x \in N} R(x)$ $V(i) \rightarrow V(j)$.

The object of this paper is to investigate under which conditions this mapping is indeed a deterministic function.

In the realizations of a concurrent process, we do not have any control over the resources of the servicing system and the scheduling algorithms: it was specifically assumed that initiation and execution delays are arbitrary, though finite. Further, the environment changes dynamically and may not be reproducible. Under these circumstances it is possible that different activations of a given process with the same initial values do not produce the same results.

Yet, the deterministic behavior of a concurrent process is undecidable if the process is completely specified. For example, consider the process represented in figure 2 where the two nodes f, g do not bear any precedence relation to each other. This process realizes the function: $x = f(x) \text{ or } g(x)$.

The problem of deciding whether x has a deterministic value then amounts to deciding whether the two functions f and g are equivalent, a well-known undecidable property.

In order to avoid this difficulty, we shall be interested in structural properties of concurrent processes which hold true irrespective of the interpretation, i.e. properties of the process graphs.

3. Complete determinacy.

Definition: A process graph is completely determinate if and only if, for every interpretation, the sequence of values appearing in each variable in $\bigcup_{x \in N} \{D(x) \cup R(x)\}$ depends only on the initial values of the variables.

In other words complete determinacy ensures that the modification of any variable's contents is uniquely determined by the interpretation and the initial values of the variables, although the sequence of initiations and terminations and the relative times at which the variables are modified may not be deterministic. The term "determinate" will be occasionally used to qualify such a variable.

Thus an interpreted determinate process graph realizes a (deterministic) function, whatever the interpretation.

In the remainder of this study, we focus on the properties of acyclic process graphs. Obviously, the model can represent very complex cyclic structures. However, their consideration would not provide any new insight into, but rather tend to obscure, the issues. It would, in particular, make the investigation of two very interesting new properties unduly complex.

A process graph represents all of the possible activations of a concurrent process. Yet, a particular activation may request only a subset of the nodes, in which case the flow of control will traverse a subgraph.

Definition: A conjunctive subgraph (denoted $\ast$ -subgraph) (N', A') of a process graph P is defined recursively by the rules:

- (i) The initial node $\in N'$
- (ii) For every node $x \in N'$:
 - if x has $\ast$ -output logic, all arcs incident from $x \in A'$
 - if x has $\oplus$ -output logic, only one arc incident from $x \in A'$
- (iii) A node x with $\ast$ -input logic $\in N'$ if and only if all arcs incident into $x \in A'$. A node x with $+$ -input logic $\in N'$ if and only if at least one arc incident into $x \in A'$.

The constructive rules (i),(ii),(iii) clearly imply that, in a *-subgraph, there is a path from the initial node to any other node.

Consider the graph of figure 3. Its three *-subgraphs are represented in figure 4.

Each *-subgraph represents one possible activation of the process, while the complete graph represents all possible activations. Figures 3 and 4 are particularly interesting on two counts. First, although node 5 appears in the process graph of figure 3, it is never executed in any of the *-subgraphs. This prevents node 2, in particular, from being related to the terminal node 7, in figure 4a and 4c. Thus node 7 can be terminated while 2 is still in progress and, in such a case, we cannot tell when a process has terminated. The notion of terminality of a procedure graph will be introduced later to cater to this problem. Secondly, in the activation represented by the *-subgraph (b), both arcs (3,6) and (4,6) will be enabled. This implies that, in spite of the precedence relation between node 3 and node 6 (for example), node 6 could actually be initiated (and terminated) before node 3 has terminated (or even initiated). In other words, although nodes 3 and 6 are related by a directed path, they are still mutually asynchronous (they may be executed asynchronously). For such nodes, the precedence is said to be a weak precedence relation and the *-subgraph (b) is said to be inconsistent as of node 6.

Such a situation was explicitly barred as "illegal" in [5]*. Yet, consistency is not necessary for determinacy and we shall not require it. Thus, in addition to allowing a more general input logic, we do not have to

*Their definition of "legality" required that every *-subgraph be consistent and include the initial and terminal nodes.

test the legality of a process graph.

As far as determinacy is concerned, the problem is to determine all the pairs of mutually asynchronous nodes, or equivalently, for every node x , all the nodes which must be completed before x can be initiated. Towards this end, we define a strong precedence relation between nodes of a $*$ -sub-graph by

Definition:

Node x is a strong predecessor of node y (denoted $x \preceq y$) if and only if:

- x is a predecessor of y
- x must be completed before y may be defined.

Clearly, strong precedence is a transitive relation. Note, further, that two nodes of a $*$ -subgraph are mutually asynchronous if and only if they are not related by a strong precedence path. The following theorem can then be established.

Theorem I:

An acyclic process graph P is completely determinate if and only if, for every $*$ -subgraph P_s of P , any two nodes, x and y , not related by a strong precedence path in P_s satisfy condition

$$(\alpha): [D(x) \cap R(y)] \cup [D(y) \cap R(x)] \cup [R(x) \cap R(y)] = \emptyset$$

Proof: For consistent $*$ -subgraphs, (normal) precedence is equivalent to strong precedence. I.e., two nodes of a consistent $*$ -subgraph are mutually asynchronous if and only if they are not related by a (normal) precedence path. Therefore the proof of Theorem I exactly parallels that given in [7, p.38] for consistent $*$ -subgraphs if "precedence" is replaced with "strong precedence".

In order to establish the decidability of the complete determinacy of a process graph, we must give an effective procedure to determine all the pairs of mutually asynchronous nodes of a given $*$ -subgraph P_s . Since P

has a finite number of nodes and a finite number of $*$ -subgraphs, Theorem I provides an effective decision procedure.

Let $P_s(x)$ be the set of strong predecessors of x in the $*$ -subgraph P_s . The set to be determined is $\{(x,y) \in P_s \text{ such that } y \notin P_x(x) \text{ and } x \notin P_s(y)\}$. Thus, it is necessary and sufficient to determine $P_s(x)$ for all $x \in P_s$. This is done by the following algorithm. Let C_s be the connectivity matrix of the $*$ -subgraph P_s , i.e. $c_{ij} = \begin{cases} 1 & \text{if } (i,j) \in P_s \\ 0 & \text{otherwise} \end{cases}$

Starting with the initial node, the algorithm inductively labels each node i with $P_s(i)$ until each node has been labeled.

Algorithm:

1. Label the column corresponding to the initial node with $P = \emptyset$ and mark every 1 element in the corresponding row as 1^* .

2. Find an unlabeled column, j , whose 1 elements are all marked (see Lemma 1). Since all the strong predecessors $(j_1 \dots j_m)$ of node j have been labeled, j can now be labeled with

$$P(j) = \begin{cases} \bigcup_{k=1}^m \{j_k \cup P_s(j_k)\} & \text{if } j \text{ has } *-input \\ \bigcap_{k=1}^m \{j_k \cup P_s(j_k)\} & \text{if } j \text{ has } +-input \end{cases}$$

Mark all 1 elements in the j th row as 1^* .

3. Repeat step 2 until all columns have been labeled.

The following two lemmas prove the correctness of the algorithm.

Lemma 1: If some columns are still unlabeled, there exists at least one which can be labeled.

Proof: Suppose Lemma 1 is false. Choose any yet unlabeled node, say j . Then, at least one immediate predecessor of j , i , cannot be labeled since row i contains an unmarked 1 (namely c_{ij}). If i is the initial node, this contradicts step 1. If i is not the initial node, by the same argument, it must have at least one unlabeled predecessor. Since the graph is acyclic and there is a path from the initial node to any other

node, repeating this argument a finite number of times will bring about the contradiction that the initial node cannot be labeled.

Lemma 2: When a node is labeled, its label contains all of its strong predecessors and only those.

Proof: The proof is by induction. Clearly, lemma 2 is satisfied for the initial node ($P_s = \emptyset$). Assume that Lemma 2 is true for every immediate predecessor of a yet unlabeled node, j , whose immediate predecessors have all been labeled (Lemma 1 ensures the existence of such a node if all nodes have not yet been labeled).

1. If j has *-input, then all its immediate predecessors must be completed before it can be executed: they are all strong predecessors of j . Since strong precedence is transitive, any $x \in P_s(j) = \bigcup_{k=1}^m \{j_k \cup P_s(j_k)\}$ a strong predecessor of j .

Conversely, if x is a strong predecessor of j , it is either an immediate predecessor of j , in which case $x \in P_s(j)$, or a strong predecessor of some immediate predecessor k , in which case $x \in P_s(k)$ by hypothesis, and therefore $x \in P_s(j)$. That these 2 cases are exhaustive is proved by contraposition. Suppose that a strong predecessor, k , of j is neither an immediate predecessor nor a strong predecessor of any immediate predecessor of j . Then, all the immediate predecessors of j are mutually asynchronous with k and therefore j and k are mutually asynchronous, contradicting the hypothesis. This concludes the proof of Lemma 2 for a *-input node.

2. If j has +-input logic, by definition of $P_s(j)$ any $x \in P_s(j)$ is either

(a) an immediate predecessor of j and a strong predecessor of every other immediate predecessor of j ; or

(b) a strong predecessor of every immediate predecessor of j .

Therefore x must be completed before any immediate predecessor in

case (b), or j in case (a), can be initiated. That is, x is a strong predecessor of j (Incidentally, for a $+$ -input node, at most one immediate predecessor can be a strong predecessor). Conversely, let k be any strong predecessor of j and suppose $k \notin P_s(j)$. Then there is at least one set $\{j_v \cup P_s(j_v)\}$ such that $k \notin \{j_v \cup P_s(j_v)\}$. Thus j and k are mutually asynchronous. Since the completion of j_v is sufficient for the initiation of j , k and j are mutually asynchronous, contradicting the hypothesis $k \in j$. This completes the proof of Lemma 2.

Note, incidentally, that the algorithm in fact determines the transitive closure of the strong precedence relation. This property will be used in the next sections.

4. Terminal Process Graphs.

Up to this point, we considered a single concurrent process. Yet, a given concurrent process may be only part of a larger system of concurrent processes sharing the resources of the processing system. When a concurrent process terminates and releases its resources, the system will perform management (housekeeping) tasks: de-allocation of resources, re-allocation to and initiation of delayed processes, etc. Thus, from an implementation point of view, execution of the terminal node is the only information available to the system to determine whether an activation of a process is terminated. However, not every process graph will reach its terminal node in every activation.

Consider as an example the graph depicted in figure 5. This graph has four $*$ -subgraphs, but notice that the $*$ -subgraphs (c) and (d) do not reach the terminal node; for these activations, the process will be hung-up.

There are other types of non-terminality. Consider the graph in figure 7, whose $*$ -subgraphs are given in figure 8. Each subgraph reaches the terminal node, but each of them contains a node which does not bear any strong precedence to the terminal node. This implies that such a node can terminate in any order with respect to the terminal node. Thus, we cannot

ensure a deterministic behavior of the system, even if the process graph is completely determinate, since housekeeping tasks could be started before all elementary processes have completely terminated.

We therefore define terminality of a process graph as follows:

Definition:

A process graph is terminal if and only if, in every *-subgraph, every node is a strong predecessor of the terminal node.

If we modify the graph of figure 7 by adding two dummy nodes, 7a and 7b, it then becomes terminal, as indicated in figures 9 and 10. It is clear that the algorithm of the preceding section provides, also, a test for terminality.

5. Partial and Terminal Determinacy of Process Graphs.

Complete determinacy of a process graph requires that every variable referenced in the graph be determinate. There are, however, situations in which we are interested only in the determinate modification of a proper subset of these variables. We call this property partial determinacy.

Definition:

A graph is partially determinate with respect to the subset S (or S-determinate) if every variable in S is determinate.

This property is sometimes referred to as "output functionality" [8].

Clearly, complete determinacy implies partial determinacy with respect to any subset of the variables. An interesting question, proposed by Van Horn [8], is whether the converse is true. The answer is no.

Proof: Consider the graph of figure 11, where node 1 is the initial node and node 4 the terminal node. It is obvious that x and z are determinate and y is undeterminate. Therefore, the graph is (x,z) - determinate but not completely determinate.

If we replace the domain of node 3 by $D = \{y\}$, the new graph (figure 12) is not partially determinate with respect to any subset of $\{x,y,z\}$ since y and z and finally x in node 4 are undeterminate. These examples illustrate the different ways in which a variable can be undeterminate and show that the candidate condition: for every pair of mutually asynchronous nodes (i,j) of the same $\ast$ -subgraph,

$$(\beta) \{ [D(i) \cap R(j)] \cup [D(j) \cap R(i)] \cup [R(j) \cap R(i)] \} \wedge S = \emptyset$$

is necessary but no longer sufficient for S -determinacy of a process graph (whereas it is necessary and sufficient for complete determinacy).

For instance, in figures 11 and 12, (β) is satisfied for $S = \{x,z\}$ but the graph of figure 11 is $\{x,z\}$ -determinate while the graph of figure 12 is not.

A variable may indirectly become undeterminate by appearing in the range of a node of which one domain variable is undeterminate. However, it may be only temporarily non-deterministic: at some point in the $\ast$ -subgraph, a deterministic value may again be stored in that variable. For example, in figure 11, the value of y is deterministic after the execution of node 4 and if node 4 had a successor, it could use this value of y and still be $\{x,z\}$ -determinate.

Thus, it is not necessary that non- S variables be determinate.

We also note that in figure 11 the final values of x,y,z , after the execution of node 4, are deterministic: they depend only on their initial values. This is an extremely interesting property which we shall call terminal determinacy.

Definition:

A process graph is said to be S -terminally determinate if it is terminal and the final value of every S -variable is a function only of the initial values of its domain variables. Terminal determinacy is certainly

the most interesting and the least restrictive of the properties of process graphs. Yet, partial and terminal determinacy have received very little attention. We shall show that, in the context of the present model, they are decidable.

Since terminal determinacy emerged from studying partial determinacy, we will pursue their investigation in parallel.

Informally, undeterminate behavior can be created only by two unrelated nodes sharing a common variable, but can be propagated through the *-subgraph by execution of the nodes. The problem is to trace and determine the extent of this undeterminacy to see whether it affects S-variables.

When a variable becomes non-deterministic at a particular node, this information should be forwarded down to all of its successors so that those which use this variable can be notified that it was non-deterministic at some point. To that effect, we label each arc with the set of the non-deterministic variables at the instant when the arc is enabled.

A non-deterministic variable can become deterministic only if it appears in the range of a node whose domain variables are all deterministic at the time the node is enabled. Thus, if $N(x)$ represents the set of non-deterministic variables when node x is defined, the forwarded set $L(x)$ of non-deterministic variables when x terminates (i.e., the label on each arc incident from x) is:

$$L(x) := \text{if } N(x) \wedge D(x) \neq \emptyset \text{ then } N(x) \cup R(x); \text{ else } N(x) - R(x)$$

A further difficulty develops, however: since non-deterministic values can appear in the domains, the decision numbers may be non-deterministic as well and therefore, starting with the same initial values, different *-subgraphs may be activated.

Thus, if a node has a non-deterministic variable in its domain and a θ -output logic, the arc to be selected upon completion is

non-deterministically chosen. Since a successor of such a node may or may not be executed in different activations, the values stored by its successors are non-deterministic, independently of any other reason. In order to propagate this information, we flag (as non-deterministically enabled) the output arc with the node number.

If a node x has a flagged incoming arc, then its range should be added to $L(x)$ and the incident flag should be set on all arcs incident from x .

The only exception to this rule occurs when node x appears in all the $*$ -subgraphs containing the node numbered by the flag f : in this case, whatever the arc enabled upon completion of node f , node x will be executed in all the activations executing f ; the non-deterministic effect of node f ceases at node x and, as far as this effect is concerned, x will be deterministically enabled. However, the arcs incident into x may carry other flags and, unless the exception is also valid for the other flags, these should be forwarded.

Note that since a flagged arc will be enabled only in some activations of the graph, if a node has $*$ -input logic and a flagged arc incident into it, it cannot possibly appear in all activations containing the flag node. Therefore, the exception, with respect to one flag, may occur only for $+$ -input nodes.

Thus a flag, f , is suppressable from the input flag set of a $+$ -input node x if and only if every $*$ -subgraph containing node f also contains node x . Otherwise f should be forwarded, along with all the non-suppressable flags of the incoming arcs.

Procedure A, whose flowchart is given in figure 13, then determines the output flag-set and the set $L(x)$, which constitutes the label of all arcs incident from node x . In particular, $L(x)$ specifies which variables

have a non-deterministic value when x terminates, i.e. when the output arcs are enabled. For the propagation rules to be complete, it remains to determine $N(x)$, the set of non-deterministic variables when x is defined.

A node with $*$ -input will be defined when all incident arcs are enabled and it is therefore possible that between the instant when an incident arc is enabled and the instant when the node is defined, some non-deterministic variables became deterministic. Figure 14 illustrates such a case (flag-sets are not represented for the sake of simplicity). Node 3 will be defined only when 2 has terminated, and although y was undeterminate on arc (1,3) it has a deterministic value when node 3 is defined. Further, arc (1,2) carries the same flag-set and L-set as (1,3) and therefore the suppression of arc (1,3) will not suppress any information.

This is generalized by the following rule.

Rule 1:

For every set of arcs incident into a $*$ -input node and whose origins are linked by a single path, retain only the arc whose origin is the end of the path (i.e. which will be enabled last).

Thus when dealing with a $*$ -input node, we first "suppress" conceptually all redundant arcs according to Rule 1, so that there is no precedence relation between any two remaining immediate predecessors of the node. Since these predecessors can terminate in any order, thereby enabling the incident arcs in any order, the set $N(x)$ (and the flag-set) is obtained by taking the union of the L-sets (respectively flag-sets) of the non-redundant incident arcs. Figure 15 illustrates this determination.

For a $+$ -input node, the input flag-set and L-set are simply the union of those of the incoming arcs, since the latter can be enabled asynchronously (if more than one in the $*$ -subgraph considered).

The propagation rules are now complete. It remains to generate the initial undeterminacies and propagate them properly in the graph.

Undeterminacies are created only by mutually asynchronous nodes sharing a common variable. Indeed, Theorem 1 states that this is a necessary and sufficient condition for the existence of undeterminacies: if such nodes did not exist the graph would be completely determinate and therefore S-determinate for all S.

The initial undeterminacies are generated and marked, for every *-subgraph, by the rule:

Rule 2:

For every pair of mutually asynchronous nodes x and y:

- if $R(x) \cap R(y) \neq \emptyset$ mark, in both $R(x)$ and $R(y)$, all the common variables.
- if $D(x) \cap R(y) \neq \emptyset$ mark all common variables in $D(x)$ only
- if $D(y) \cap R(x) \neq \emptyset$ mark all common variables in $D(y)$ only.

Rule 5 is illustrated in figure 16. Note that in figure 16b it is not necessary to mark y in $R(1)$: upon completion of node 1, y has a deterministic value; but since node 2 may initiate before or after node 1, at the time 2 is defined y does not have a deterministic value.

The generation and propagation rules are consolidated into a procedure which decides whether a given graph is S-determinate. For every *-subgraph, the procedure first generates all initial undeterminacies and then, propagates this information from top to bottom.

A node is said to be processable when all the arcs incident into it have been labeled, but it has not yet been processed. Once a node is processed, the arcs incident from it are labeled and the node is checked (i.e. marked as "processed").

Procedure P:

1. Determine all marked variables according to Rule 2.
2. Propagation:
 - a. Label all arcs incident from the initial node with ($L = \emptyset$; flag-set = $\emptyset$). Check the initial node.
 - b. For every processable node, x :
 - Determine $N(x)$ according to Rule 1 and the associated procedure.
 - Set $N(x) = N(x) \cup \{\text{marked variables in } D(x)\}$
 - Apply Procedure A to determine $L(x)$ and the output flag-set of x
 - Set $L(x) = L(x) \cup \{\text{marked variables in } R(x)\}$
 - Label every arc incident from x with $L(x)$ and the output flag-set.
 - Check node x .
3. Record all variables that appeared in some $L(x)$. (These are the non-deterministic variables for this *-subgraph).

It is clear that the procedure terminates when all the nodes have been checked. We now have:

Theorem II:

The graph is S-determinate if no S-variable ever appeared in some L-set for some *-subgraph.

Of course, as soon as an S-variable either is marked or appears in an L-set, we can conclude that the graph is not S-determinate.

But the procedure, when applied to all *-subgraphs, yields more information: it tells us that the graph is partially determinate with respect to the subset of variables which never appeared in an L-set.

Another significant property of the procedure is its potential as a debugging aid: it indicates exactly why and at which points the graph starts to be undeterminate. Since one major objection to parallel

programming is precisely the lack of debugging tools, this feature of the procedure described here is substantially instrumental in validating the parallel programming approach.

Yet, probably its most interesting property resides in that it provides a test for S-terminal determinacy of a terminal process graph:

Theorem III:

A terminal process graph is S-terminally determinate if no S-variable ever appears in the L-set of the terminal node.

This is clear since the L-sets of the terminal node contain all the non-deterministic (and none of the deterministic) variables of the graph.

Note that since the terminal node belongs to every *-subgraph in a terminal graph, all flags (if any) are suppressable when the terminal node is processed.

The procedure P provides us with a constructive proof of the decidability of partial and terminal determinacy for process graphs.

Certainly, this procedure could be refined so as to avoid total enumeration of the *-subgraphs and to be more computationally effective, but this is not our concern in this paper.

To summarize briefly, the control structure of the concurrent process can be represented by an uninterpreted process graph, whose interpretations, in terms of the primitive functions defined at the level of interest, realize a mapping from the domain to the range. The correctness of the dynamic behavior of a concurrent process is characterized by three properties of its graph - complete, partial and terminal determinacy - which are shown to be decidable.

To conclude this study of the properties of the process graph model, we briefly consider two interesting extensions of the model.

6. Mutual Exclusion.

Two processes are said to be mutually exclusive if they may be executed in any order but not simultaneously; i.e. they are still related by a precedence relation but it is determined dynamically.

Many variations of Dijkstra's semaphores [9] have been proposed to implement mutual exclusion. In [7] we retain Dijkstra's original notation to implement a straightforward mutual exclusion scheme and demonstrate the model's capability.

As far as determinacy is concerned, mutual exclusion of processes which do not share common variables does not add any new constraints and therefore, from this point of view, such mutually exclusive processes are treated as mutually asynchronous. This is a "trivial" situation.

A more interesting situation occurs when they do share some common variables. In this case, the associated process graph cannot be completely determinate since the common variables can be modified in a different order in different activations. However, it may still be partially determinate (the procedure of the previous section will decide this). Thus, the notion of partial determinacy is of prime importance if we want to consider mutual exclusion. Yet, the fact that mutually exclusive processes execute in a strict sequence will not be used in deciding partial determinacy, since the order is arbitrary. However, this characteristic of mutually exclusive processes would be used if we had some additional information on the functions performed by the processes, e.g. that they commute with respect to some of their variables. This information could then be taken into account by the procedure P and provide less restrictive conditions for partial (or terminal) determinacy. Thus, we are suggesting that the behavior of concurrent processes exhibiting mutually exclusive elementary processes will be best investigated in the context of a partially

interpreted process graph model, i.e. one in which we have some information about the function performed by the nodes.

7. Generalized Process Graphs and Their Specification.

7.1 Generalized Process Graphs.

The concept of process graph naturally lends itself to an interesting generalization: the individual nodes of a process graph could represent other (concurrent) processes whose structure would in turn be represented by process graphs.

In other words, the process graph can be considered as partially interpreted: although the functions performed by the individual nodes are not completely specified, their structure is given, in terms of other partially interpreted process graphs. As we shall see, this additional information substantially modifies the problems involved.

Clearly, this recursive description of processes requires a more elaborate set of definitions and conventions.

First of all, two types of nodes should be distinguished:

- the primitive nodes, which are assigned primitive functions in any complete interpretation of the process graph.
- the non-primitive nodes, whose structure will be specified by partially interpreted process graphs.

Definition:

A partially interpreted process graph is a process graph uniquely identified by its name and whose nodes either are primitive or are assigned the name of a partially interpreted process graph, except for the initial and terminal nodes.

In the context of process graphs, the initial and terminal nodes play a special role: they are responsible for the proper "passing" of

the parameters, the management of temporary variables and the flow of control from and to the calling node.

This is, of course, implementation dependent and will not be discussed here. For the sake of completeness, it is assumed that:

- (i) any particular interpretation of a process graph is a reentrant, pure procedure.
- (ii) the decision function is realized by the terminal node of the graph.
- (iii) the binding of the variables, the flow of control and the management of temporary variables are properly realized by the initial and terminal nodes.

The notion of process graph can now be generalized and recursively defined as the set of partially interpreted procedure graphs resulting of the union of the initial partially interpreted process graph and the process graphs assigned to every non-primitive node of the initial graph.

Thus, a process graph has a rooted-tree structure in which a vertex names a process graph and its immediate successors, the component process graphs. The leaves of the tree are either all-primitive process graphs or process graphs whose name appeared already once in the directed path from the root to the leaf. A process graph is said to be non-recursive if every leaf of the tree identifies an all-primitive process graph.

7.2 Specification of process graphs.

Corresponding to the notion of interpretation a particular specification of a process graph assigns a primitive function to every primitive node of every partially interpreted process graph.

The reader will remark that a specification of a process graph may or may not provide a valid interpretation of the initial process graph since the component processes may or may not realize a deterministic function from domain to range variables.

Properties similar to those investigated in the previous sections can be defined for generalized process graphs:

Definition:

A process graph is completely functional if and only if for every specification, the sequence of values taken by every referenced variable is uniquely determined by the initial values of its domain variables.

Definition:

A process graph is S-functional (or partially functional with respect to the set S) if and only if, for every specification, the sequence of values taken by every S-variable is uniquely determined by the initial values of the domain variables.

A theorem similar to Theorem I can now be stated:

Theorem IV: A generalized process graph is completely functional if and only if each of its process graph is completely determinate.

The proof is given in the author's dissertation [7]. However, when we address the partial functionality of a process graph, its potential recursiveness create some difficult problems. We shall only briefly mention them.

In a graph, if a node has an undeterminate domain variable then, upon completion, all its domain variables are undeterminate (since there clearly exists at least one interpretation for which they would not be deterministic). This need not be the case in a generalized process graph, since the components process graphs are already partially interpreted: only those range variables whose computation uses an undeterminate domain variable will be undeterminate.

On the other hand, a component process graph may create some new undeterminacies. For a given component process graph, it may be the case that, although all of its domain and range variables are left determinate

by the other component process graphs, some of its range variables may be non-deterministically modified by the process graph itself. Note, however, that these newly created undeterminacies will not affect the predecessors nor the unrealized nodes, but only the successors of such a node.

Clearly, these peculiarities of process graphs can easily be embedded into a modified version of the procedure P given in section 5. If we call P' such a modified procedure, then a recursive application of P' to the components of the process graph considered will theoretically determine its partial functionality.

It is clear that P' would terminate for non-recursive process graphs, thereby providing a decision procedure for such graphs. Yet, for the more interesting recursive graphs, P' would not terminate. Although we shall not dwell on this problem any longer, but rather leave it for further investigation, it is this author's conjecture that partial and terminal functionality are indeed decidable for recursive process graphs.

8. Conclusions.

In contrast with strictly sequential processing, the determinism of the results of concurrent processes is a significant and difficult question. In this paper, the notion of deterministic behavior of concurrent processes has been formalized in the context of a graph model, the Process Graph Model, and characterized by successively broader properties. In addition to the now standard notion of (complete) determinacy, the notions of partial and terminal determinacy were introduced as alternate and more general formalization^s of the intuitive concept of deterministic behavior. Their decidability was established for acyclic graphs, but remains an open problem for recursively defined concurrent processes.

The Process Graph Model can easily represent the mutual exclusion relationship; but it was suggested that concurrent processes having this property would be best studied in the context of partially interpreted process graphs, if the full power of this relation is to be efficiently used.

Acknowledgments.

The author wishes to thank Professor Michael J. Flynn for his continuing support. I am also indebted to Mr. Jayadev Misra for many interesting suggestions.

REFERENCES

- [1] Karp, R. M. & Miller, R. E., "Parallel Program Schemata", Journal of Computer and System Sciences, Vol 3 (1969) pp. 112-195.
- [2] Adams, D. A., "A model for parallel computation", Parallel Processor Systems, Technologies and Applications, Ed. L. C. Hobbs (1969).
- [3] Rodriguez, J. E., "A graph model for parallel computation", Ph.D. Dissertation, MIT (1967) Project MAC - TR -64.
- [4] Martin, D. F. & Estrin, G., "Experiments on models of computation and systems", IEEE Transactions on Computers, Vol EC-16 (1967) pp. 59-69.
- [5] Baer J. L, Bovet, D. P. & Estrin, G., "Legality and other properties of graph models of computations", JACM, Vol. 17 n^o3, July 1970, pp.543-554.
- [6] Gostelow, K. P., "Flow of control, resource allocation and the proper termination of programs", Ph. D. Dissertation, University of California, Los Angeles, December 1971, UCLA Report #7179.
- [7] Regis, R. C., "Systems of concurrent processes: structural properties and stochastic analysis", Ph. D. Dissertation, Johns Hopkins University, June 1972.
- [8] Van Horn, E. C., "Computer design for asynchronously reproducible multiprocessing", Ph. D. Dissertation, MIT, November 1966, Project MAC TR-34.
- [9] Dijkstra, E. W., "A constructive approach to the problem of program correctness", BIT, Vol 8, (1968) pp. 174-188.

Figure 1

Figure 2

Figure 3

Figure 4

Figure 5

(a)

(b)

(c)

(d)

Figure 6

Figure 7

Figure 8

Figure 9

Figure 10

Figure 11

Figure 12

Figure 13

Figure 14

$N(1) = \{y\}$
 $N(2) = \{y, u\}$
 $N(3) = \{u, t\}$
 $N(4) = \{y, t\}$
 $N(5) = \{t, u, v, y\}$

Figure 15

Figure 16