INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

University Microfilms International
300 North Zeeb Road
Ann Arbor, Michigan 48106 USA
St. John's Road, Tyler's Green
High Wycombe, Bucks, England HP10 8HR

A graph model for coordinating systems of tasks

by

Ralph Kendall White

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1976

TABLE OF CONTENTS

LIST OF FIGURES

THE ABSTRACT

The modeling of task oriented parallel processing systems is studied in this dissertation. The model found to be most useful is the UCLA Graph Model of Computation. Emphasis is placed on the modeling of synchronized parallel tasks.

The model is extended to incorporate an environment concept and its management through the use of the Data Flow Graph. This use of the environment gives the model dynamic aspects. Parameter passage is defined and the model restricted to force task separation except at definable points. The separation makes analysis for variable conflicts and deterministic behavior less complex.

Analysis is preserved for the deadlock free property by showing a condition called proper termination is preserved. Examples of various programming constructs are modeled showing the capability of modeling existing parallel programming systems.

I. INTRODUCTION

A. Properties and Definitions

In the past 25 years since the first computer was built, systems, both operating systems and programming systems, have been designed and programmed in unscientific and ad hoc ways requiring much wasted time and effort before a "working" system was evolved. Once it reached the "working" stage no guarantee existed that some strange input would not cause the system to fail by looping, giving bad answers, or simply terminating abnormally. Denning (18) gives a lengthy discussion of this point.

To begin to find a solution to this problem, one must first understand the basic building blocks from which computer systems are built. A system is a set of concurrent, cooperating processes called tasks. Tasks may be created by other tasks and may also terminate or be terminated by other tasks. We will use the terms ATTACH or FORK to represent the creation of tasks and the terms DETACH or JOIN (15,21,25,31) to represent the termination of tasks. A task may be represented by a pair of pointers labeled "i" and "e", designated "(i,e)". "i" is a pointer to an instruction stream and "e" is a pointer to an associated environment. The instructions of the task are performed in a designated or implied sequence and they operate on the memory locations in the associated environment.

Denning (18) states a set of characteristics for a system of tasks in his article on third generation computer systems. They are concurrency, automatic resource allocation, sharing, multiplexing, remote conversational access, non-determinacy, and long term storage. Concurrency is the parallel or simultaneous execution of tasks. Concurrency has been found to greatly increase the amount of work that a computer system can perform because it allows the central processor to service other tasks in the system while a task is in a wait state waiting for external interrupts or waiting for service from some other task in the system. For example, a task may be waiting on I/O for a device to be completed. During this time the central processor is freed to service other tasks in the system thus using time that would otherwise be idle time for the central processor. Automatic resource allocation is necessary for detecting and avoiding deadlock, for maintaining resource balance policies, and for aiding programmers in allocating resources. Programmers have been found to be more productive when they do not have to be concerned with resource allocation problems. Sharing of data, programs, and other resources decreases storage demands and removes redundancy. It also allows the use of a common data base by many programmers thus allowing programmers to build on each others work. Multiplexing is the dynamic reassignment of a preemptable resource such as the processor

over a relatively short interval of time so that many users can have opportunities to use that resource. Remote conversational access refers to users having on-line, remote access to executing tasks, a necessary characteristic for any kind of time sharing system. It is also useful as a debugging tool. Non-determinacy in the sense used here means the inability to determine the ordering of events during system operation. This characteristic stems largely from the desire to process external interrupts or signals on a demand basis. Without this characteristic systems would be very inflexible. Certainly local parts of systems should be determinant to the extent that the final result must be explicitly predictable from the input to that local program or system part. Finally, long term storage implies saving data files for use days or months later. This characteristic is necessary for storage of the system itself, compilers, and other data.

From these seven areas we glean five basic areas of abstraction which a model should address. Those are programming, storage allocation, concurrent tasks, resource allocation, and protection. The question now arises as to whether there are any theoretical principles which aid in the implementation of the above mentioned features and the control of concurrent tasks. Here Denning (18) lists four: determinacy, deadlock detection and avoidance, mutual

exclusion, and synchronization.

To define determinacy we must first define an interpretation for a task. An interpretation for a task is a mapping from the input to the output cells specified for the task or it is the algorithm performed by the task on its input and output cells. A system of tasks is said to be determinant if for every interpretation, the values of the output cells for the tasks in the system depend uniquely on the values of the input cells for the tasks in the system, and hence, the final values of the output cells do not depend on the order that the processor services tasks ready for execution. The implication here is that models should be invariant for any interpretation, a property called interpretation free; otherwise, determinacy would need to be proved for every interpretation, and hence, the utility of the model would be very limited. Determinacy is the property that frees the programmer from the time consuming and expensive task of debugging the program logic assuming the correct algorithm was selected initially (as is many times not the case). This property has, of course, not been achieved from a pragmatic point of view.

Deadlock is a logical problem in resource allocation in which two sets of tasks are requesting resources held by each other (25,27,35,36,38,39,41,50,59). The assumptions are that resources cannot be preempted and that each task may exclu-

sively control the resources it holds. Deadlock free is an operational constraint necessary for an operating system to maintain an optimum level of throughput.

Two tasks are said to be mutually exclusive if the tasks require sequential, exclusive control of the same resource (11,16,35,59). An example would be two tasks having sections of code called critical sections which access the same buffer and must not access the buffer at the same time. Mutual exclusion is a property necessary to achieve determinate systems in which data is shared among tasks.

Synchronization between tasks is necessary when one task must wait on another task to perform a function before it can continue. Both mutual exclusion and synchronization can be realized by the LOCK and UNLOCK or WAIT and SIGNAL (21,25,66,76,77) programming constructs or Dijkstra's semiphores, P and V (23,51). IBM's operating systems use ENQ and DEQ macros for mutual exclusion and WAIT and POST macros (44,77) for synchronization with some overlap. Synchronization is a property which greatly increases the flexibility of tasks. The terms used in this dissertation will be WAIT-SIGNAL and ENQ-DEQ. WAIT and SIGNAL are event oriented while ENQ and DEQ are more resource oriented.

A number of attempts (1,3,8,10,14,19,22,24,26,28,32,46,49,53,58,60,61,62,63,64,65,68,70,72,74,75, to indicate a few), have been made to model various aspects of parallel

processing systems. Although several have been shown to be useful in describing parts of parallel systems, it is this author's opinion that none have thus far been shown to be adequate to actually describe and analyze the programming constructs used in current operating systems which allow the communication of cooperating parallel tasks and the sequencing of events among those cooperating tasks. The intent of this dissertation is to bring model building for parallel tasks a step or two closer to being able to model existing systems with completeness.

B. Literature Review

There are two basic areas that model builders attack. Those two areas are the arithmetic or computational programming constructs and the sequencing of control programming constructs. These two may be addressed in one model but the undecidability of some properties results when the two are not separated (7,57). The model developed in this dissertation presents a way of separation of these two properties without the loss of modeling power.

Theoretical investigations generally take one of two approaches (57). The first is to restrict the model so that desired properties, such as determinacy and deadlock detection, hold. There is a whole area of research concerning the application of program proving techniques (17,52). The second is to design a model that is less restricted, and

then establish necessary and sufficient conditions for the desired properties to hold. For the studies presented here, the latter approach is more adaptable as the desire is to develop a model that addresses the varicus programming constructs and aspects of existing operating systems.

The useful existing models which address scme cr all of Denning's four principles are:

1. Vienna definition language (74,75)
2. Common base language (20)
3. Finite state machine models (9,10,29,30,42,56)
4. Petri Nets (37,40,45,58,60,61,62)
5. Graph models (1,2,6,53,64)
6. Parallel program schemata (43,47,48,49,68,69)
7. Message transmission systems (12,63),
8. Bilogic graph model of computation (3,4,5,8,13,14, 32,33,34,54,55,65,70,72,73)

These approaches to modeling may be grouped intc two categories, operational and theoretical. Vienna definition language (VDL) and the common base language are operational in the sense that they describe the desired ccnstructs of programming languages with a minimum cf consideration given to the four principles previously mentioned. The emphasis is on describing all the constructs characteristic to programming languages. The remainder of the approaches to modeling are designed such that some or all of the four principles can be proven within the constraints of the model. The sacrifice

here is that some of the desired programming constructs either cannot be represented in the model or are difficult to represent. For example, in the Petri net model a simple counter is very difficult to represent, depending on the size and generality of the counter. The theoretical approaches have taken the directed graph as the basic element while the operational approaches have taken a modified tree structure as the basic design element.

VDL (74,75) is directed specifically at an information-structure-oriented approach to modeling programming languages. Besides arithmetic expressions it models such structures as APPLY, ASSIGN, IF-THEN, BEGIN-END, PROCEDURE, and parameter passage. VDL uses a tree to define its data structures. The basic elements of the tree are the control, environment, denotation, attribute, dump, and integer or name. These are further subdivided. The control specifies the sequence of events or statements to be executed and the environment contains a set of names and associated indexes to elements in the denotation. The denotation contains values, procedures, and function definitions (the procedure bodies, parameters, and environments at the time of declaration are contained herein). The attribute specifies the types of variables found in the environment, and the dump is a push down stack onto which the control and environment are pushed when a procedure or function call is made. The integer or

name specifies a list of unique names. The VDL model is interpreted in that the tree structure completely defines a specific program. Each program has a different tree structure which must be analyzed separately for correctness.

Within VDL an interpreter exists which performs the functions specified in the control branch. Those include selecting the program from the denotation and performing the statements, managing the environments associated with each procedure in the program and performing the necessary parameter passage for procedures. The interpreter is specified in a modified Bachus-Naur Form (BNF) syntax.

VDL was designed and has been shown to be quite adequate to describe the syntax and semantics for programming languages. It does not attempt to address memory management, resource allocation, interrupts, and other constructs found in operating systems. Wallentine (74), however, has been able to show how multitasking can be defined in VDL with modifications to the data structure. VDL, as a vehicle for modeling operating systems, has several major difficulties. The strictly hierarchical nature of its objects makes the sharing of those objects difficult. Another difficulty is that state transitions are described as global transitions on the total system state. This notation makes the description of a system very awkward and obscure, i.e. the true system state is obscured by the notation (67). Multiple copies of

procedures and environments are obtained when multiple calls are made to the same procedure in the reentrant and recursive cases. This is not done in general in real systems because memories are limited in size. VDL has difficulty addressing determinacy and the detection of infinite looping.

Unlike VDL, the common base language (20) approach is not interpreted. It concerns itself with correctly modeling the data structures found in programming languages. The objective is to design a model in which the creation of correct programs is as convenient and easy as possible and to define a base language suited to the execution of computations in many languages around which computer architecture can be built. As in VDL, a tree structure with identifiers is used to select the branches of the tree and ultimately the objects at the ends of the branches. A state has a universe, local structure, and control. The universe contains the program's data structure and procedure structure. The local structure contains the data structures for a current activation of a procedure. The control is defined as the sites of activity of the program. The common base language is analogous to the contour model's (46) instruction/environment pointer in that a directed arc is used to associate a procedure with its local structure. Several activations of a procedure may exist concurrently using different local structure pointers. Instructions such as 'construct' an elementary object, 'link'

two objects together, 'delete' an arc or node, 'apply' a new procedure, 'add' two data objects, 'select' a data structure, 'return' to calling procedure, and 'move' data objects are defined. Within the data structure, attempts are made to discover cycles by looking at all external references from a procedure. The conclusion is that the elimination of cycles would be sufficient to generate correct programs. This model is cast into the data-flow schemata model (22,64,71) which models the flow of control and data. It was also found to be necessary to introduce AND, OR, and NOT logic to represent some programs so that a control signal for any Boolean function could be generated. The conclusion is that any program representable with a data-flow schemata is determinant. As with VDL the intent of the model is to describe programming constructs in higher level languages and not in systems.

Finite state machine models (9,10,29,30,42,56) was one of the first attempts at modeling computer systems and hence the model's properties and behavior have been well defined. The basic principle is defined in terms of a finite set of operations over a finite set of states. An initial state must be defined and a mapping function defining how to proceed from one state to the next must also be defined. The mapping function may be described in terms of a directed graph with the vertices of the graph representing states and the arcs representing the progression from state to state in

time. These systems are finite and hence the states may be enumerated allowing a complete analysis. A recent attempt to extend these finite systems to non-finite systems was relatively unsuccessful (9).

Petri nets were first introduced by Carl Adam Petri (62) and extended by Holt and Commoner (40) and later by Patil (60) and Peterson (61). Petri nets attempt to model the static and dynamic properties of concurrent conditions. A net consists of a set of arcs and vertices called conditions and events by Holt and Commoner. Arcs represent conditions and bars, found at the initiation and termination of arcs, represent events. Tokens placed on arcs represent the holding of conditions. When all the input conditions on the input arcs to an event or vertex are met, the vertex is said to fire taking one token off all the input arcs and placing one additional token on the output arcs of the fired vertex. Holt and Commoner (40) and Izbicki (45) have formally studied a subset of Petri nets called marked graphs which have the restriction that transitions may not share input or output arcs. A marked graph is a finite state machine in that any specific condition may begin or terminate one and only one event. Several conditions may begin or terminate a specific event but not multiple events. Petri nets are useful in that some complex structures may be represented very easily but also some very simple ones such as a counter are not easily

representable. Petri nets suffer somewhat from the one-token per-vertex restriction (safeness) and the disallowance of self loops (32). Determinacy has not been proven for the general class of Petri nets.

Noe and Nutt (58) use modified Petri nets to describe the SCOPE 3.2 operating system of the CDC 6400. Petri nets provide a good conceptual or pictorial description which easily allows the alteration of the level of detail (a necessary and very important aspect of systems programming according to Noe and Nutt). The model uses OR, exclusive OR, and AND logic in the nets to describe the desired conditions (a characteristic not found in Holt and Commoner). Noe and Nutt found the physical size of a page a serious problem in that the nets tended to get very large. This is a valid criticism of all graph models to date.

Peterson (61) has applied P-nets, which are modified Petri nets, to formal language theory and the analysis of grammers with success.

Adams (1,2), Luconi (53), and Rodriguez (64) have developed graph models for parallel computations. All three are similar in that the vertices represent computations and the arcs represent single variables, queues of data, or a specification of how control is to flow from one operation to another. Rodriguez's model has a rather complex control mechanism designed to facilitate pipelining operations. The arcs

in the model in addition to defining single variables necessary for the computations also carry status information such as disabled, idle, enabled, and blocked. The nodes of the graphs are operators such as 'and', 'or', 'selector', 'junction', 'loop junction', 'loop output', etc. Specific input and output arc states are required for an operator to be active. Rodriguez shows that computations expressed in his model are determinant.

Adams' model is less complicated than Rodriguez's model and is somewhat more general. The arcs in the directed graph represent queues of data along which data values move. As with Rodriguez's model the arcs carry status information although the status information is less complicated. Data dependent decisions can be made based on the status information. The model is designed to handle recursion. Adams has shown that every computation representable in his model is determinant.

In Luconi's model the status information is kept on each arc as with Rodriguez's and Adams'. Data values associated with each arc represent not only the data necessary for a computation but also the data values specify the flow of control from one operation to the next. Luconi has not shown computations expressed in his model to be determinant.

Karp and Miller (47,48,49) and Slutz (68,69) have developed general models called parallel program schemata and flow

graph schemata, respectively. The initiation and termination of a computation are distinguished so that the control state can change during a computation. In Slutz's model the termination of a computation can be made to wait on other computations in the graph while in Karp and Miller's model once a computation is initiated all conditions are also satisfied for it to terminate by definition. Arcs represent the flow of control and of data and also record the number of data values queued on each arc. Each input and output arc to a computation has a counter. When all input counters to a computation are greater than or equal to one the computation may initiate, compute for a finite time, and then terminate incrementing counters on output arcs. A number of results concerning equivalence of schematas and determinacy are shown.

Riddle (63) has developed a very interesting model for modeling parallel processes. Processes are modeled in terms of messages transmitted to and requested from link processes. The messages have only a type associated with them which must be specified when requesting or transmitting messages. Once a process requests a message, it is required to wait until the message is supplied to the link process by some other process in the program. A system of processes is called a program. Programs are represented in a program process modeling language (PPML). The PPML concentrates

principally on communication between processes and is not concerned with the modeling of arithmetic computations. Analysis of the programs expressed in PPML is based on a message transfer expression which defines how messages flow through the system and is the basis of the analysis of the system for correct operation. The analysis requires that program processes be finite and that the number of message types be finite.

The model with which this dissertation is most concerned is the UCLA graph model of computation (GMC) (3,8,14,32,65, 70). The model is directed toward a clear and concise expression of the flow of control for parallel, cooperating tasks or processes. The GMC models the flow of control for computations in the form of bigraphs. A bigraph consists of a set of arcs, vertices, and associated logic. The vertices represent operations and the arcs represent paths along which control in the form of tokens may pass. Each graph must have a single entry and single exit arc. The logic is represented in the graph by "*" (AND) and "+" (OR) indicating whether tokens are required on all or one the inputs to an operation before the operation can proceed or whether tokens are placed on all or one of the outputs when an operation completes. Informally, the execution of a bigraph must terminate with a finite number of operations. This is called proper termination, a property directly related to the deadlock free prop-

erty. More formally, following Gostelow (32), a bigraph is said to be proper terminating if:

1. Only a finite number of tokens are required to execute the bigraph, and
2. When execution halts, the final token distribution is one token on the exit arc of the graph and no other tokens on any other arc of the graph.

An important property which guarantees the possibility of having a properly terminating bigraph is repetition free (RF). A bigraph is said to be repetition free if no vertex can be executed twice. If the bigraph has a vertex that can initiate twice, it is still repetition free if between initiations there is some other vertex executed whose output affects the input of the vertex that can be initiated twice. This property guarantees that if v is a vertex in a loop which makes a decision, then there is always some interpretation which will cause v to branch arbitrarily, thus eventually allowing the loop to terminate. Cerf and Gostelow are able to derive a reduction procedure by which a repetition free bigraph may be determined to be proper terminating. They have shown that the GMC is able to model interrupts, mutual exclusion, synchronization, subroutines, and resource allocation. The static nature of the GMC has made it difficult to adequately model parameter passage particularly in

recursive and reentrant routines and also to model the dynamic nature of systems. Gostelow (32) has shown that any construct that can be represented with Petri nets can be represented with the GMC.

Karp and Miller (47,48), Slutz (68), Adams (1), and Rodriguez (64) have shown determinacy of computations expressed in their models. A change in the control state is allowed between the initiation and termination of a computation. Hence, these models are somewhat more general than the Cerf (14) and Gostelow model (32) in that they allow the "pipelining" or queuing of data. The control structures are cumbersome and complicated. Petri nets (62) are useful for expressing some complex program behavior with a simple control structure although others are difficult to represent. Noe and Nutt (58) have helped by introducing AND, OR, and NOT logic. Cerf (14) and Gostelow (32) have shown proper termination and its relation to the deadlock free property, mutual exclusion, synchronization and other programming constructs in a simpler and more concise form with fewer special constructs.

C. Organization of the Dissertation

Chapter II gives a description of a modified GMC and shows how additional structure in the data definition for the model provides new modeling power which can better model existing systems. It discusses the addition of the concept of

a changing environment in relatien to mcdeling the dynamic nature of systems. Chapter III gives a formal description and discusses formal properties in relatien to the extended GMC. Chapter IV presents examples of the new modeling power gained and points toward the development of an interactive system building program. Chapter V contains conclusions and further research.

II. DESCRIPTION OF THE MODEL

A. A Task Oriented Example

One of the intents of this dissertation is to show how existing programming systems or systems of tasks can be modeled and analyzed. Other authors have demonstrated the power of the GMC to model the arithmetic nature of tasks (14,32,72) with completeness. However, several aspects of the control mechanism need further study. For this reason we will focus on intertask communication in this dissertation and begin with an example of a multitask monitoring system.

In this example given in Figure 1a, a main task, which will be called MT, creates or attaches subtasks, called STi's, to perform various functions. The subtasks are given the ability to request specific services from MT through the use of event variables which are set and reset by WAIT and SIGNAL operations. The services performed by MT for the subtasks would typically be services which require correct exclusive control of specific resources to guarantee correct behavior and results for the subtasks. An example is performing input and output to a channel for a group of interactive terminals. Another is performing the management of specific resources such as the privilege to access a specific set of data or data set or the allocation of memory.

In the example given in Figure 1a, the event variables for subtasks (EVSTi's) are event variables used to signal the

```
DECLARE MT:

        CREATE ST1 (EVST1,EVIO1,EVRC1,RDATA1)
        CREATE ST2 (EVST2,EVIO2,EVRC2,RDATA2)

WAITLOOP:  WAIT (EVST1,EVST2,EVIO1,EVIO2,EVRC1,EVRC2)

        IF (EVST1) BEGIN <Detach ST1>; GO TO TEST; END

        IF (EVST2) BEGIN <Detach ST2>; GO TO TEST; END

        IF (EVIO1) BEGIN <Do I/O for ST1>;
                         GO TO WAITLOOP; END

        IF (EVIO2) BEGIN <Do I/O for ST2>;
                         GO TO WAITLOOP; END

        IF (EVRC1) BEGIN <Do resource management for ST1>;
                         GO TO WAITLOOP; END

        IF (EVRC2) BEGIN <Do resource management for ST2>;
                         GO TO WAITLOOP; END

TEST:   IF (ST1 & ST2 terminated) GO TO END
        GO TO WAITLOOP

END:    END MT
```

Figure 1a. A Simple Multitasking Monitor: MT.

termination of subtasks. The event variables for input and output (EVIOi's) are event variables used to signal main task (MT) to do input or output. The event variables for resource control (EVRCi's) are event variables used to signal MT to perform resource management. The resource data areas (RDATAi's) are data areas provided by MT to the subtasks for input and output communication and resource management data.

The significance of the multitasking monitor example is that operating systems operate fundamentally the way MT does

here to obtain the exclusion required to be able to allocate resources consistently, correctly, and without conflicts. A main task or tasks are given exclusive control, i.e. in a sequential, non-interruptible fashion, during which the tasks access the resource in a conflict free environment. Even user programs which operate as subtasks of a supervisor or operating system, which have agreed to cooperate for access to a resource, invoke mechanisms in operating systems, to accomplish the ccoperation, in which a main task receives control in a sequential, non-interruptible fashion. Sc, to be able to model the example given is to be able to model the basic mechanism behind operating systems and to model systems themselves. We present initially this simple example of the basic control structure. There are, however, many other problems to be solved to model operating systems in their totality. Some of these will be addressed later.

B. The Control Flow Graph (CTFG)

Figure 1b is a partial control flow graph, called a CTFG for short, of the main monitor task (MT) in Figure 1a. Only one function performed by MT is shown with its connection to the subtasks. The rest would be connected similarly. The arcs represent the flow of control and the circles, called vertices or nodes, represent events or operations. An operation may be the performance of a subroutine or it may be a single instruction. Vertices begin and end arcs. Arcs which

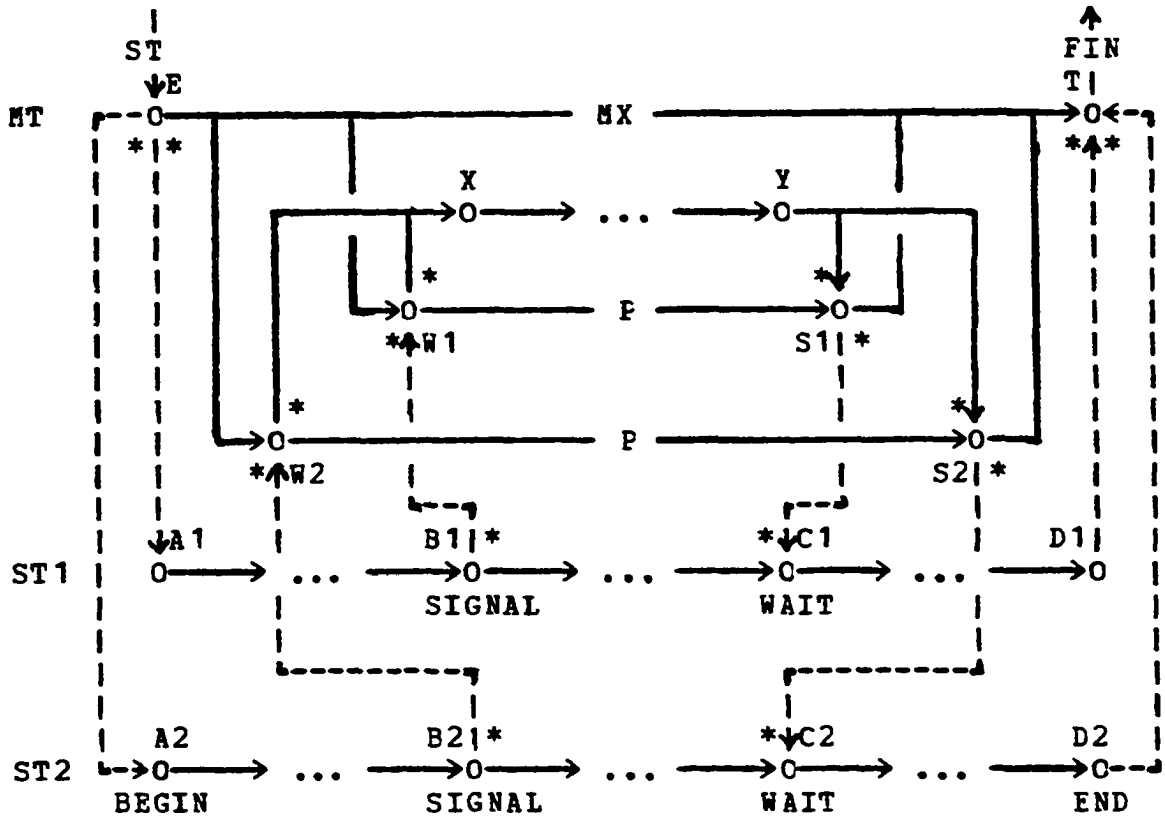

Figure 1b. The CTFG of MT and the STi's.

begin at a vertex are the output arcs of the vertex while arcs which end at a vertex are the input arcs of the vertex. The initiation of arcs are called tails and the termination of arcs are called heads. An arc may have many heads and tails. "+" logic in input or output represents an "or" condition and "*" logic represents an "and" condition. Tokens placed on the arcs represent an instance or point of execution in the graph and will be represented as a closed circle or dot on the arcs of graphs. For an operation with "*" logic on its inputs to be eligible for execution, it must

have tokens placed on all input arcs. In the case of "+" input logic, only one arc is required to have a token on it. A particular arrangement of tokens on a graph is called a marking (40) of that graph.

When an operation is performed, one token is taken off each of the input arcs in the case of "*" logic while one token is taken off only one of the arcs in the case of "+" logic. "+" and "*" logic may not be mixed for either input or for output although the input logic and the output logic need not be the same. In the case that two or more input arcs to an operation with "+" input logic have tokens, the choice of which token to take is arbitrary. Hence, we will have some problems with determinacy. "+" output logic represents a decision that must be made by the operation to which the "+" logic applies.

Just as the solid arcs represent the flow of control, the broken arcs also represent the flow of control except the control is through event variables. This type of WAIT-SIGNAL control is what might be called an indirect flow of control as opposed to a direct flow of control such as a LINK or CALL. One might also look at the broken arcs as representing timing constraints. We emphasize here that what timing constraints are needed for the correct operation of modeled systems of tasks, are included as control arcs in the model, and hence, the model is a time independent representation of

the tasks.

C. The Execution

To execute the control flow graph (CTFG) in Figure 1b, a token is placed on arc ST. At termination a token will be placed on arc FIN with no other tokens remaining on the graph. Vertex E represents a CREATE operation in main task (MT) which creates subtask 1 and subtask 2 (ST1 and ST2). We represent the creation of the subtasks to be simultaneous although that is not necessary. Arc MX is a special control arc which affects mutual exclusion in MT when the subtasks signal requests for services. The vertices marked Wi in MT represent the multiple wait conditions. A decision must be made at vertex X which is not shown in the control flow graph (CTFG) as to the service performed. The vertices marked Si represent signal operations and are used to signal the appropriate task that its service has been completed. The arcs marked P represent control information which determines the appropriate task to signal. The arcs labeled P could be regarded as 'return addresses or pointers' or as event variables used to pass control back to the correct subtask.

Gostelow (32) uses a similar solution to what we have used here for a reentrant subroutine for proper control and return. If the vertices marked Wi and Si are replaced by a multiarc, control information as to who requested service and who is to receive control back is lost. In the multiarc case

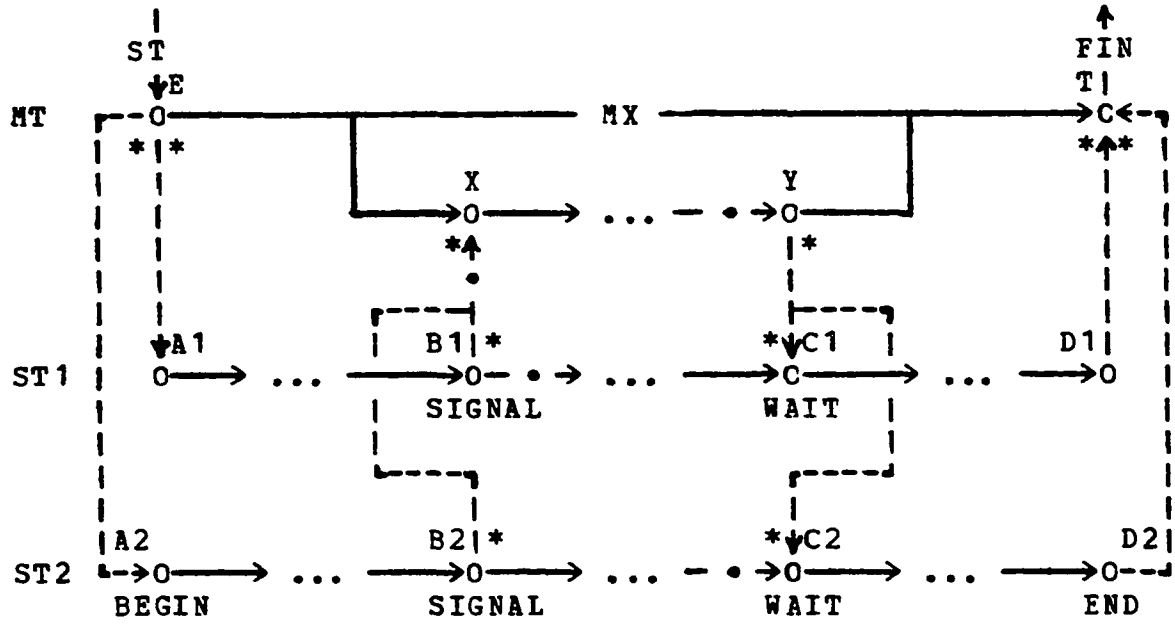

Figure 1c. An Incorrect CTFG for MT and the STi's

incorrect operation could result. Consider the example in Figure 1c with the specified token marking. Main task (MT) has just performed a service for subtask 1 (ST1). ST2 requested services and got to its WAIT for services to be complete before ST1 did. When vertex Y is executed, ST2 is enabled to proceed before its services are completed.

It must be pointed out that the particular solution presented here to model proper return mechanisms, as in Figure 1b, is overly restrictive. Only one execution between X and Y may proceed at a time. If the code between X and Y was truly reentrant and did not require mutual exclusion for correct operation, then this would not be the best representation. We will discuss return mechanisms again later giving

a more detailed explanation.

D. The Environment

The set of nodes marked Wi and Si in Figure 1b have yet another purpose other than providing the proper control mechanism. When a task branches with "*" output logic, control is actually split in that a new task is created which may perform in parallel with the originating task. At these points where new tasks are created, the new tasks must be given access to a set of storage locations available for use. Some of these locations may be new, unused locations while others may be parameters or other types of storage known also to other tasks, i.e. shared variables. The storage may be storage that is allowed to be modified or it may be read-only storage which cannot be modified. What we have said is that when new tasks are created by "*" output logic, new environments must also be created. If a new environment is created for each task, a realistic, task level separation of data may be modeled, parameters may be passed, work area for the new tasks may be allocated, and control information such as event variables and return addresses may be stored. Similarly, task and environment deletion or deallocation occurs with "*" input logic. Not all vertices with "*" logic cause the creation of new environments. Hence, in the model, we use the combination of the "*" logic and the broken arcs to represent the environment delineation described here.

At this point a question arises as to the dynamic nature of a system of tasks. Most programs or tasks have the ability to select among a set of algorithms based on a specific value assignment to variables in their environment or they may modify an algorithm based on the environment in which the tasks are running. Control structures are generally not modified dynamically. They are modified by something external to the control structure. Control structures may, however, behave differently as a result of a change in the environment. The dynamic nature of systems is contained principally in the environment. The exception to this statement is in the area of artificial intelligence and in the area of compiler-compiler techniques which is heuristic in nature. Even in these areas a major portion of the dynamic nature is based on a selection of alternatives from an environment. Compiler-compilers can restructure or reshape themselves in a way which changes the flow of control. The GMC is not able to treat thoses cases directly in which the control structure is dynamically changed. However, the GMC can be made to treat the cases where the environment is different from task to task. Nodes can be made to stand for whole subroutines or other sections of the control program on a substitution basis. Yavne (78) and Gostelow (32) have treated this type of substitution in the CTFG of the GMC. When the graphs are analyzed for correctness, the assumption

is made that the subroutine or control flow graph (CTFG) that is substituted for a node or vertex has the desired properties of correctness, i.e. it is analyzed separately. Yavne and Gostelow prove that the composite CTFG has the desired properties.

There are two types of data which compose the environment. They are arithmetic and logic data. Arithmetic data is data which is used in arithmetic computations and does not directly effect the logic of the control graph. It may indirectly, however, as a decision may be based on an arithmetic computation. Logic data is used directly in making decisions and in task synchronization. Decision vertices use decision logic data for making decisions. Decision logic data is often used as arithmetic data also. There is some overlap, and hence, the type of data depends on its use. Synchronization logic data includes mutual exclusion event variables and wait and signal event variables. This type of data must be included as arcs of the control flow graph for analysis purposes. It must also be included as a type of data because it is implemented in the software of a computer system as part of the environment.

In Figure 1b the dotted arcs represent the passing of control through event variables and the arcs labeled P represent control information associated with return mechanisms. In this case an old existing environment of main task (MT)

would be combined with the control information passed by the task requesting services to form a new environment. New storage locations may be added. At termination of the services performed by MT any new storage could be deallocated and the environment returned to its original condition which existed before the call. If a resource was allocated, perhaps an allocation counter variable would be updated leaving the original environment of MT modified. In the case we mention here an old environment is added to or modified. A WAIT-SIGNAL mechanism would result in the interface of tasks and their environments while an ATTACH-DETACH or FORK-JOIN mechanism would create or destroy tasks and their environments. Both are represented with broken arcs and "*" logic.

E. The Data Flow Graph (DFG)

Just as control flows from operation (or node) to operation (or node) in the control flow graph, data can be viewed as flowing from one operation to the next. A graph called a data flow graph (22) can be drawn which expresses the flow of data. The data flow graph (or DFG) could be combined with the control flow graph (or CTFG) in one graph except for two reasons. One is clarity. Distinguishing which arcs are control and which are data flow becomes difficult and the graph becomes too large and cumbersome much more quickly. The second is that control structures are inherently static in

nature while environments are inherently dynamic in nature. Environments are continually changing and being created and destroyed. Each "*" output logic from an operation with a broken arc represents the creation of a new task and a new environment or the modification cf an old task and its environment. The first operation of each new task must have associated with it necessary informaticn from which to create a new environment. The necessary informatien is a data flow graph (DFG) from which a new environment may be created. Storage may or may not need to be allccated. So, a data flow graph and a ccntrol flow graph are associated with each task. The DGF is used to create a new envircnment at execution time and to show the flow of data from operation to operation. The CTFG represents the flow cf ccntrol from operation to operation.

Consider the example in Figure 1b. The subtasks, ST1 and ST2 and main task, MT, are three separate tasks. Let the services performed by MT between X and Y be for input and output. The event variables for input and cutput (EVIOi's) are the variables used by MT to signal the operation and the resource data areas (RDATAi's) are the communicatien areas. MX is the mutual exclusion variable known to MT. Figure 1d represents MT's DFG.

One way of looking at the data flow graph is as describing which operations have access to which variables and

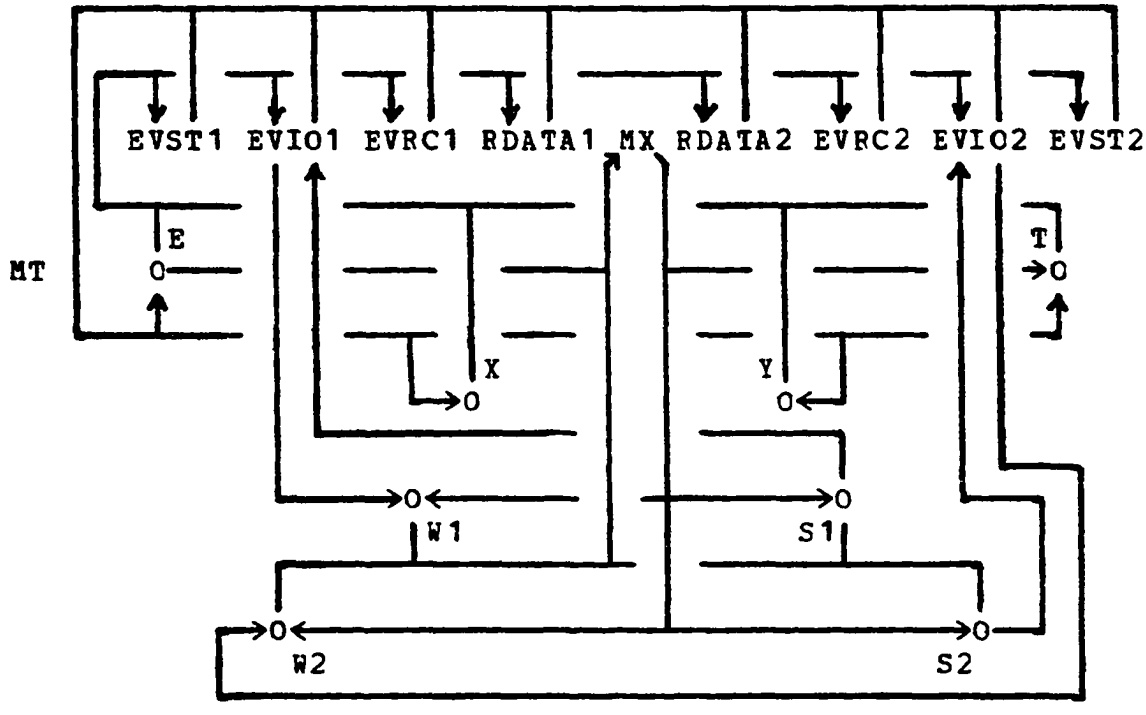

Figure 1d. The Data Flow Graph for MT.

whether the access is as an input, an output, or both. E must have access to all variables as it makes known the variables to the subtasks (STi's). X, Y, and the operations between must have access to the RDATAi's to perform the input and output operations. T must have access to the termination variables to determine when termination occurs, and so on.

Figure 1e represents the environment and data flow graph for the subtasks requesting input and output services of MT. Operations Ai and Di represent the entrance and exit processing, respectively, and also the data manipulation before and after input or output is performed. Bi and Ci represent the

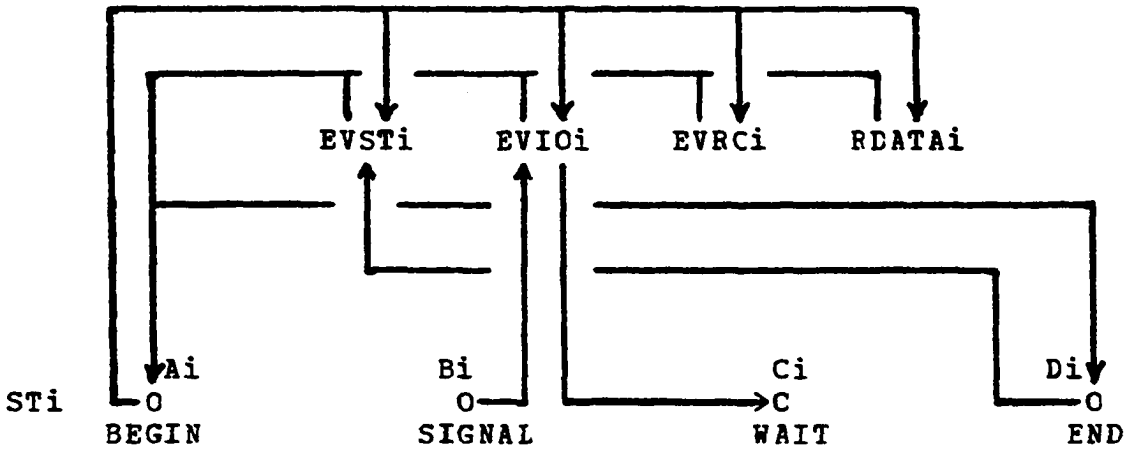

Figure 1e. The Data Flow Graph for the STi's.

signal and wait mechanism used to signal MT to perform services. In a real implementation of the example here, EVIOi is actually two separate variables, one for STi to signal MT for services and one for MT tc signal STi that the services are ccmplete.

F. The Token Machine (TM)

At execution, MT is given an environment as defined in Figure 1d. An "(i,e)" pair (46) of pcinters, one fcr the CTFG or an instruction pointer and one fcr the DFG cr an environment pointer is set up. As more tasks are created by "*" logic, new "(i,e)" pairs are created and collected in a list. The list is a list of all the active tasks in the system along with their control flow graph (CTFG) and data flow graph (DFG) pointers. Cerf (14) defines a "token machine". The tcken machine is responsible fcr perfcrming

operations and moving tokens, representing instances of control or execution, through the CTFG. We extend the notion of a token machine to include environment management here. The token machine is responsible for making parameters supplied by calling or attaching routines available to be included in the environments of called or attached routines. Earlier we indicated that the first operation of a called or attached routine was responsible for establishing a new environment from its own DFG and from passed parameters. Space must be allowed for work areas, data areas, and return pointers. It is necessary to also include return pointers in the CTFG for analysis purposes as we pointed out earlier.

We present next an informal definition for the token machine. It is a modification of Cerf's token machine (14).

1. Clear all tokens from the graph, place a token on the entry arc of the single, initially active task (there must be only one), and enter that task in the "(i,e)" pair list.
2. Randomly select a task from the list of tasks represented by "(i,e)" pairs.
3. Check the logic for the vertex i to see if it is enabled. If not, repeat step 2 until no vertices can be executed, else continue with step 4.
4. Perform the operation specified by the inter-

pretation for vertex i placing tokens on the output arcs of vertex i according to the output logic and updating the environment pointed to by "e" in the "(i,e)" pair according to the interpretation.

5. Then update the instruction pointer "i" and add or delete any "(i,e)" pairs as dictated by the logic, the arc type, and the interpretation for vertex i. Continue at step 2.

A token machine represents a processor operating on the list of active tasks. More than one machine can operate on this list of tasks so long as no two processors are allowed to select and execute a vertex simultaneously.

G. Summary of the Modified GMC

With the environment and data flow graph concepts that have been outlined here, it is possible to consider modeling programs with block structure. Modeling programs which use virtual storage concepts are possible where complete separation of environments for programs is achieved. A mechanism exists for separating environments such that it is possible to analyze small pieces of programs, characterized by a control flow graph and a series of data flow graphs, for variable conflicts. The DFG concept points toward a complete analysis for determinacy. The problem of a correct representation for calling and return mechanisms is solved as well as

a correct modeling of parameter passage.

In the GMC it is possible tc model parallelism, task synchronization, mutual exclusion, and perform an analysis of the CTFG for a deadlock free condition. Standard programming constructs such as loops, branches, and decisions are also modeled. With the extended model a ccrrect representation is found for calling subroutines, handling proper return, passing parameters and managing envircnments, thus giving the GMC a dynamic aspect. Gostelow (32) has shown hcw interrupt structures could be implemented, resources allocated, and vertex replacement accomplished. Yavne (78) has extended the notion cf vertex replacement.

III. FORMAL DEFINITION AND PROPERTIES

A. Definition of the Modified GMC

This chapter gives a formal definition of the model and discusses the various properties of the GMC such as repetition freeness (RF) and proper termination (PT) with its relation to deadlock freeness. Also, the computation flow graph (CFG) is introduced with its associated transformation expression set (TE set) and reduction procedure.

Before we begin our definitions we must introduce several concepts. An arc may have multiple tokens on it at any one time. For example, several tasks may wait on the completion of the same event. A vertex may place more than one token on an arc as a result of its execution. The Q value of an input or output arc defines the number of tokens placed on or taken from an arc. Arcs, called multiarcs, may have more than one head and more than one tail. SESX refers to a single entry, single exit graph. SESX graphs have one and only one entry arc and one and only one exit arc. We generally follow Cerf's definitions (14) with modifications to allow for the additional structure in the environment.

A digraph is defined first, then a bigraph, and then a GMC. A digraph or directed graph (6) is simply a set of vertices connected by arcs. Some or all of the arcs may be multiarcs.

Let G = (V,A) be a SESX digraph.

V is a finite set of vertices.

A is a finite set of arcs.

$A^-(v)$ is defined as the set of arcs which have heads at vertex v, called inbranching arcs of v.

$A^+(v)$ is defined as the set of arcs which have tails at vertex v, called outbranching arcs of v.

The arcs and vertices are connected tc form a graph in the following manner:

Let I and J be subsets of V.

Let a be an element of A.

Then a = (I,J) is an ordered pair of sets.

I or J may be null but not both.

A single entry-single exit (SESX) graph has one and only one vertex of each of the following forms:

$a = (\phi, J)$ an entry arc and

$a' = (I, \phi)$ an exit arc.

When $|J| > 1$ an arc is said to be multi-headed.

When $|I| > 1$ an arc is said to be multi-tailed.

We now define a bigraph. A bigraph is a digraph with the addition of inbranching and outbranching vertex logic to define which input and output arcs are required in the execution of each vertex.

Let B = (G,L,Q) be a bigraph.

G = (V,A) is a SESX digraph.

L = (L^-,L^+) is the vertex logic for the digraph.

$L^-(v)$ is the inbranching arc logic for vertex v.

$L^+(v)$ is the outbranching arc logic for vertex v.

The logic may be either * (AND) or + (OR).

It may not be mixed either for input or for output.

$L^-(v)$ is not required to equal $L^+(v)$ although it may.

* (AND) logic requires that a token be taken off or placed on all inbranching or outbranching arcs when the vertex is executed.

+ (OR) logic requires that a token be taken off or placed on only one inbranching or cutbranching arc when the vertex is executed. The choice for the outbranching case is determined by the interpretation given a vertex with this kind of logic. The inbranching case is an arbitrary choice.

Q = (Q^-,Q^+) is a partial, single-valued mapping for V x A into the integers.

$Q^-(v,a)$ is the number of tokens vertex v takes off arc a when vertex v is executed.

$Q^+(v,a)$ is the number of tokens vertex v places on arc a when vertex v is executed.

Entry and exit vertices are required tc have Q

values of 1 for the entry and exit arcs.

We now define environments with data flow graphs (DFG's).

Let E = (V,M,D) be a set of environments with data flow graphs.

V is the same set of vertices as in the digraph G.

M is a finite set of storage identifiers.

D = (D^-,D^+) is a pair of mappings from M into V.

$D^-(m,v)$ = m if m is an input to v.

$D^-(m,v)$ = ϕ otherwise.

$D^+(m,v)$ = m if m is an output to v.

$D^+(m,v)$ = ϕ otherwise.

$D^-(m,v)$ also represents an arc from m to v in the DFG.

$D^+(m,v)$ also represents an arc from v to m in the DFG.

Summing over the m variable yields:

$D^-(v)$ is the set of input identifiers or arcs in the DFG to v.

$D^+(v)$ is the set of output identifiers or arcs in the DFG to v.

We now define environment delineators and passed parameters.

Let H = (R,P) be a set of environment delineators and passed parameters.

Let P be a set of parameter lists.

Let p be an element of P and m be an element of M. Then every p is a set of elements of M, m.

P(a) is a paramenter list for arc a where a is an element of A.

If P(a) is not null, then arc a must not be a multiarc and $Q^-(v,a)$ and $Q^+(v,a)$ must both be 1 for any v for which arc a is an inbranching or an outbranching arc. If either is the case, subroutine linkage would be unclear. There would be a choice left to the token machine as to which subroutine to call. This is not desirable. Two executions of the same subroutine could occur at the same time. This is also an undesirable result.

Let R be a subset of A. R is a set of broken or environment arcs.

If arc a is an element of R then P(a) cannot be null. At least an event variable is a passed parameter.

If arc a is an element of R then it cannot be a multiarc and $Q^-(v,a)$ and $Q^+(v,a)$ must both be 1 for any v for which arc a is an inbranching or an outbranching arc. If either is the case, the desired task separation would not be achieved, return mechanisms would be more dif-

ficult to model and task linkage would be confusing.

The arcs in R represent the passing of control through a WAIT-SIGNAL mechanism where old tasks interface or an ATTACH-DETACH (FORK-JOIN) mechanism where new tasks are created and old tasks destroyed. This is a restriction on the interpretation that may be placed on vertices connected by arcs in R.

Entry and exit arcs of graphs must be elements of R.

If arc a is an element of R, v an element of V, and arc a an element of $A^-(v)$, then $L^-(v)$ = "*".

Similarly, if arc a is an element of R, and v an element of V, and arc a an element of $A^+(v)$, then $I^+(v)$ = "*".

That is, if arc a is a broken arc and is not the only inbranching or outbranching arc of a vertex, then the associated logic must be "*". This forces multiple tasks with broken arcs as desired.

A path in a CTFG is any sequence of connected arcs and vertices such that for any two adjacent vertices in the sequence, there is a common arc having a tail at the vertex which appears first

in the sequence and a head at the vertex that appears next in the sequence.

If any two vertices can be enabled at the same time, then they can be performed in parallel.

If any two vertices can be performed in parallel, then there must be some path that split to form two paths allowing the two vertices to be enabled in parallel.

At the point of the split of the original path, we require broken arcs causing separate tasks to be created (the classical FORK operation). Likewise, when two paths from separate tasks join to form one path (the classical JOIN operation), we also require broken arcs because it is here that two tasks interface or communicate and their environments restructured, deleted, etc.

Tasks may not interface with each other except through broken arcs. That is, two vertices which can be performed in parallel may not be connected by an arc unless the arc is a broken arc where task interface is allowed to take place through a WAIT-SIGNAL mechanism or an ATTACH-DETACH (FORK-JOIN) mechanism.

Data flow graphs (DFG's) for separate tasks may not

intersect except through parameter lists.

Hence, variable usage conflicts are found only in parameter lists, P.

We now define the GMC.

The GMC is a triple C = (B,E,H).

B is a bigraph.

E is a set of environments with DFG's.

H is a set of environment delineators and passed parameters.

Some additional comments as to why the restrictions listed above have been placed on the GMC must be made at this time. Parameters are both keyword and positional. They are associated with output locations or parameters of the vertex which is at the head of the parameter arc. This is the only "special" feature of the entry vertex to a task, subroutine, or block. Additional interpretation may, of course, be placed on that entry vertex.

We build in no requirement for environment separation except in the case of broken arcs. Any arc may pass parameters. However, it is possible to separate environments and CTFG's of subroutines from those of calling routines and communicate from one routine to the next with parameters only. The possibility of passing parameters on any single arc as opposed to on just broken arcs is necessary to facilitate the modeling of block structured languages where

parameters of blocks may be held in ccmmcn cr passed.

"*" logic is restricted so that it must either be used to create or interface with a new task or it may be used within a task to facilitate control mechanisms such as mutual exclusion and to avoid parallel execution. Parallel execution is not allowed except through the use cf broken arcs.

The implications of requiring parameter arcs tc be single arcs and the vertices of parameter arcs to have Q values of 1 are that subroutine linkage would be unclear and multiple executions of the same subroutine could occur for a given task. With multiarcs as parameter arcs, the token machine wculd be required to make an arbitrary choice as to which subroutine to call. Parameterless subroutines are assumed to have at least one parameter, namely, a return parameter. If subroutines are not setup in the structure as described, incorrect operation could result. Q values greater than one are generally useful fcr ccntrol mechanisms such as resource allocation and deallccation.

It must be pointed out that with the GMC we prcvide a way of modeling programming structures correctly but there are also many incorrect ways which cannct be entirely eliminated. Analysis for properties such as proper termination, which is described later, will help.

B. Definition of the Token Machine (TM)

Token machines represent a set of identical processors operating on the task list defined previously. The task list contains an entry for each task in the system. Each entry contains a vertex or instruction pointer labeled i(p) for processor p and an environment pointer labeled e(p) for processor p. C is a GMC and I(C) its corresponding interpretation. I(C) is a quadruple which is a set of initial values for the environment (Mo), a set of functions for each vertex which do not have "+" output logic (F), a set of decision functions for vertices which do have "+" logic (Fd), and a set of operations or computations (O) performed by each vertex. The functions F and Fd describe how tokens are to be taken from input arcs and placed on output arcs. All elements of the interpretation for each vertex must correspond to the DFG and CTFG logic and arc definitions for the vertex. No other restriction is placed on the interpretation. x is the sequence of vertices executed by the token machine. It is called a computation sequence. The definitions presented here generally follow Cerf (14) and Gostelow (32) with some modifications.

1. Set $x = \phi$.
2. Clear all the tokens from the arcs of C, place a single token on the entry arc of C called E, and enter E and its associated environment

pointer as the first and only entry of the task list.

3. Each processor independently and asynchronously processes the vertices in C following steps 4 thru 7 with the condition that only one processor at a time may proceed with steps 4 or 6.

4. Arbitrarily select a task from the task list, remove the instruction pointed to by the task entry, i(p), from the vertices available for execution for this processor, and

a) If the input logic $L^-(i(p))$ is "+",

Do the following:

Let a be an element of $A^-(i(p))$;

If there are not at least $Q^-(i(p),a)$ tokens on arc a for some arc a in $A^-(i(p))$,

Then go to step 4c;

If there are, remove $Q^-(i(p),a)$ tokens from arc a, add the initiation of instruction i for processor p, $\bar{i}(p)$, to x, and go to step 5;

b) If the input logic $L^-(i(p))$ is "*",

Do the following:

Let a be an element of $A^-(i(p))$;

If there are not at least $Q^-(i(p),a)$

tokens on arc a for every arc a in $A^-(i(p))$,

Then go to step 4c;

If there are, remove $Q^-(i(p))$ tokens from arc a for all a in $A^-(i(p))$, add the initiation of instruction i for processor p, $\bar{i}(p)$, to x, and go to step 5;

c) Here a check for the termination of the token machine is made.

If all vertices looked at by processor p have not been checked go to step 4.

If there are other processors which have vertices which can be considered then go into a wait state.

If not, then terminate the computation with computation sequence x.

5. Perform the computation specified by the interpretation, $O(i(p))$, for vertex $i(p)$ updating the environment pointed to by "e" in the task list entry, passing any parameters as specified by $P(a)$, making any decisions as required by the interpretation, and adding and deleting tasks from the task list as dictated by broken arcs.

6. Following the output logic $L^+(i(p))$:

   a) If the output logic $L^+(i(p))$ is "+",

      Do the following:

      Use $Fd(i(p))$ to select an arc in $A^+(i(p))$;

      Place $Q^+(i(p),a)$ tokens on the selected arc;

      Add the termination of instruction i for processor p, i(p), to the computation sequence x, and go to step 7;

   b) If the output logic $L^+(i(p))$ is "*",

      Do the following:

      For every arc a in $A^+(i(p))$;

      Place $Q^+(i(p),a)$ tokens on each arc;

      Add the termination of instruction i for processor p, i(p), to the computation sequence x, and go to step 7;

7. Make i(p) again eligible for selection by a token machine and go to step 4.

It must be pointed out that the token machine might not terminate. However, if it does, x is the sequence of vertices executed. Since an arbitrary selection is made in step 4, x is not unique.

C. The Computation Flow Graph (CFG)

Analysis of the GMC is dependent on the generation of the computation flow graph or CFG. If all computation sequences were known for a given GMC, information

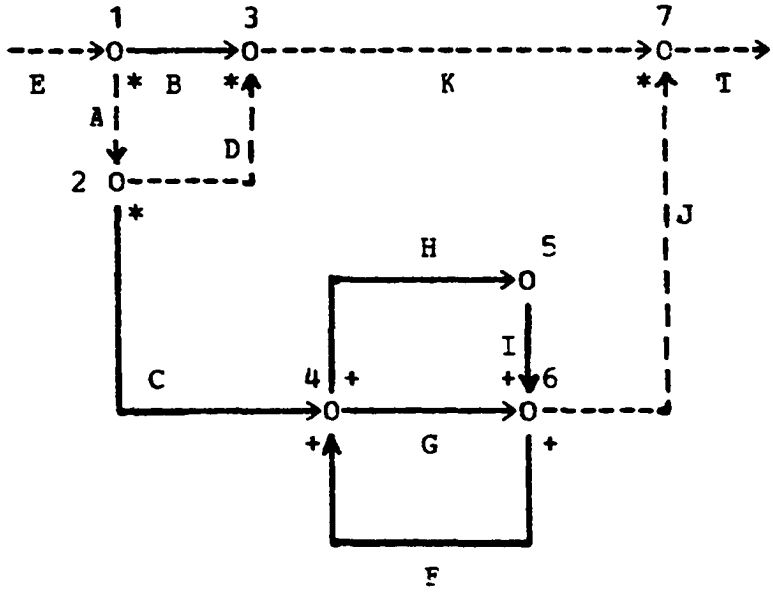

Figure 2a. A Control Flow Graph (CTFG).

would be available for this generation. A list of all possible token states could be generated from the computation sequences. If information concerning the order of the possible states is represented by directed arcs and the states represented by circles, we have just generated a computation flow graph (CFG). It is a state flow graph. The CFG reveals loop states or states that can be repeated in the CFG. It also will reveal infinite states. These are states that may

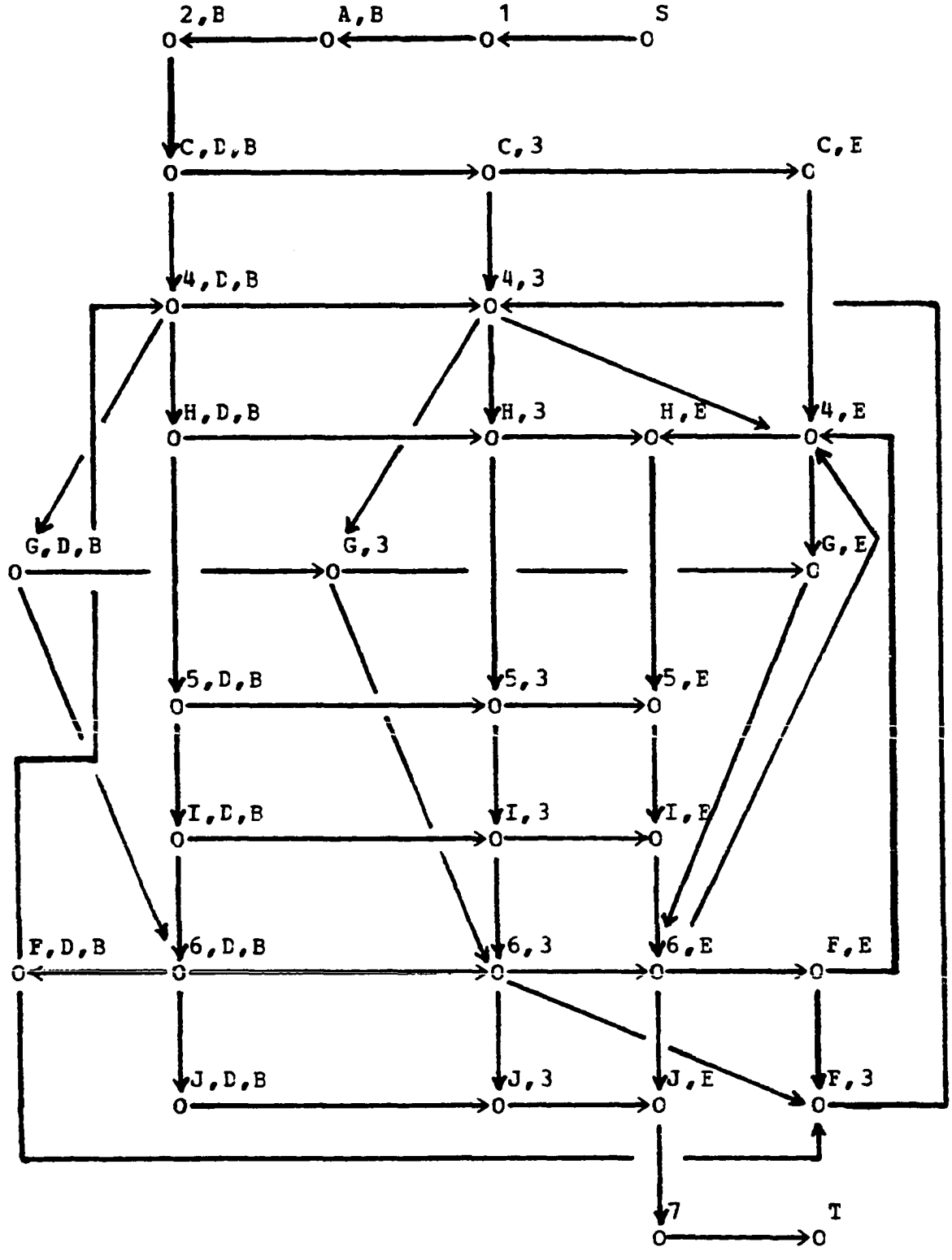

Figure 2b. A Computation Flow Graph (CFG) fcr Figure 2a.

occur an infinite number of times in a path through the CFG. Loop states are not necessarily infinite states. A state with no possible transitions into the state is an initial state and one with no possible transiticns cut of that state is a terminal state.

Consider the CTFG in Figure 2a. Vertex T is a terminal state. Vertex E is an initial state. Arc F in the CTFG creates a number of loop states. There are no infinite loop states in this CFG. There are a total of thirty-three possible token states.

D. Transformation Expressions (TE's)

Since the computation flow graph (CFG) is the major analysis tool, it would be desirable to be able to generate it without finding all possible execution sequences. Transformation expressions (TE's) are a method of generating the CFG from a control flow graph (CTFG). The CTFG does, in fact, have enough information to generate the associated TE set and the CFG. Figure 2c is the TE set for the CTFG in Figure 2a.

The notation used is the following:

A bar over a vertex number means the initiation of that vertex while the vertex number without a bar means the termination of the vertex.

$t(\bar{7})$ means the transformation expression for

vertex 7 to initiate.

$t(\bar{7})$: J,K->7 means for vertex 7 to initiate requires a token on arc J and arc K.

$t(\bar{7})$: J,K,K->7 would mean two tokens are required on arc K and one on arc J.

$t(\bar{1})$:	E->1	t(4):	4->G
t(1):	1->A,B	$t(\bar{5})$:	H->5
$t(\bar{2})$:	A->2	t(5):	5->I
t(2):	2->C,D	$t(\bar{6})$:	G->6
$t(\bar{3})$:	D,B->3	$t(\bar{6})$:	I->6
t(3):	3->K	t(6):	6->F
$t(\bar{4})$:	C->4	t(6):	6->J
$t(\bar{4})$:	F->4	$t(\bar{7})$:	J,K->7
t(4):	4->H	t(7):	7->T

Figure 2c. The TE set for the CTFG in Figure 2a.

An informal definition of how to generate TE's from the CTFG is simply to write down all the conditions possible for each vertex to initiate and then for each to terminate. A formal definition can be found in Cerf (14). To then find all the token states, a token is placed on arc E and the rules followed. As the rules are applied, one at a time, to the right hand side (RHS) of the first rule to initiate vertex 1, the token states in the CFG are generated and

P(0)	P(1) b	P(2) b	P(3) b	P(4) b	P(5) b
E->1	E->A,B	E->B,2	E->B,C,D	E->C,3	E->C,K
1->A,B	A->2	2->C,D	D,B->3	3->K	C->4
A->2	2->C,D	D,B->3	3->K	C->4	F->4
2->C,D	D,B->3	3->K	C->4	F->4	4->H
D,B->3	3->K	C->4	F->4	4->H	4->G
3->K	C->4	F->4	4->H	4->G	H->5
C->4	F->4	4->H	4->G	H->5	5->I
F->4	4->H	4->G	H->5	5->I	G->6
4->H	4->G	H->5	5->I	G->6	I->6
4->G	H->5	5->I	G->6	I->6	6->F
H->5	5->I	G->6	I->6	6->F	6->J
5->I	G->6	I->6	6->F	6->J	J,K->7
G->6	I->6	6->F	6->J	J,K->7	7->T
I->6	6->F	6->J	J,K->7	7->T	
6->F	6->J	J,K->7	7->T		
6->J	J,K->7	7->T			
J,K->7	7->T				
7->T					

P(6) b	P(7) b	P(8) b	P(9) b	P(10) b	P(11) b
E->K,4	E->K,H	E->K,5	E->K,I	E->K,6	E->K,F
F->4	E->K,G	E->K,G	E->K,G	E->K,G	E->K,J
4->H	F->H	F->5	F->I	F->6	E->K,G
4->G	F->G	F->G	F->G	F->G	F->F
H->5	H->5	5->I	G->6	G->6	F->J
5->I	5->I	G->6	I->6	6->F	F->G
G->6	G->6	I->6	6->F	G->J	G->F
I->6	I->6	6->F	6->J	J,K->7	G->J
6->F	6->F	6->J	J,K->7	7->T	J,K->7
6->J	6->J	J,K->7	7->T		7->T
J,K->7	J,K->7	7->T			
7->T	7->T				

P(12) a	P(13) b	P(14) a	P(15) b	P(16) b	P(17) b
E->K,F	E->K,J	E->K,J	E->K,J	E->7	E->T
E->K,J	E->K,G	E->K,G	J,K->7	7->T	
E->K,G	G->J	G->J	7->T		
F->J	G->G	J,K->7			
F->G	J,K->7	7->T			
G->F	7->T				
G->J					
J,K->7					
7->T					

Figure 2d. The Reduction Procedure for Figure 2c.

hence, the CFG is generated. Writing a program to find all token states would require some care as an infinite number of states can result. Loops would have to be detected and eliminated. The reverse process of generating the CTFG from the TE set is also possible so that from the TE set both the CFG and the CTFG can be generated.

E. Repetition Free Graphs (RF graphs)

We have shown how the TE sets can generate computation flow graphs (CFG's) and control flow graphs (CTFG's) with loops. In analyzing GMC's it would be desirable to know whether those loops can be expected to terminate. The repetition free property which is required of computation sequences guarantees that an interpretation can be found that will allow the token machine and hence the execution of the CTFG to terminate.

Let $\bar{x}, c(1), c(2), \ldots, c(n), \bar{x}$ be a part of a computation sequence in which $\bar{x}$ represents the initiation of some vertex x. x can be initiated twice in this partial sequence.

A GMC is said to be repetition free if for all sequences of the type $\bar{x}, c(1), c(2), \ldots, c(n), \bar{x}$ in which a vertex can be initiated twice, there exists a c(j), j between 1 and n, such that:

$$D^-(x) \text{ intersect } D^+(c(j)) \neq \phi$$

or the output of some vertex c(j) can modify

the input of the vertex x.

Thus, an interpretation can be found which will cause the loop, $\bar{x}$ to $\bar{x}$, to have a finite number of iterations. For an arbitrary interpretation (which still conforms to the GMC with a loop), the repetition free property is a necessary condition and not a sufficient one for loops to have a possibility of terminating. It provides the opportunity for some interpretation to be found which will not loop infinitely. Certainly, many interpretations can be found which will loop infinitely.

Another interesting point concerning the CFG can be easily understood at this point. The CFG is more general than the CTFG with its interpretation, as the token machine may not generate all the computation sequences possible in the CFG. The interpretation for a decision vertex may dictate that one or more paths may never be taken. When generating the CFG from the CTFG, arbitrary branching must be considered.

Loops are not the only way for a GMC to be non-repetition free. Consider a computation sequence of the form $\bar{x}\bar{x}$. There can be no vertex between two initiations of x which can modify inputs to x. Hence, the GMC cannot be repetition free. One way to create this situation is with Q values greater than one. Care must be taken in regard to the use of Q values.

A decision vertex is said to be a vertex with "+" output logic. We require the interpretation for a decision vertex to make a choice based on the input values to the vertex. Clearly, a loop in a CTFG can only be terminated by a decision vertex. Therefore, it would be tempting to conclude that we only need consider vertices whose output data cells intersect with the input data cells of decision vertices when considering the repetition free property.

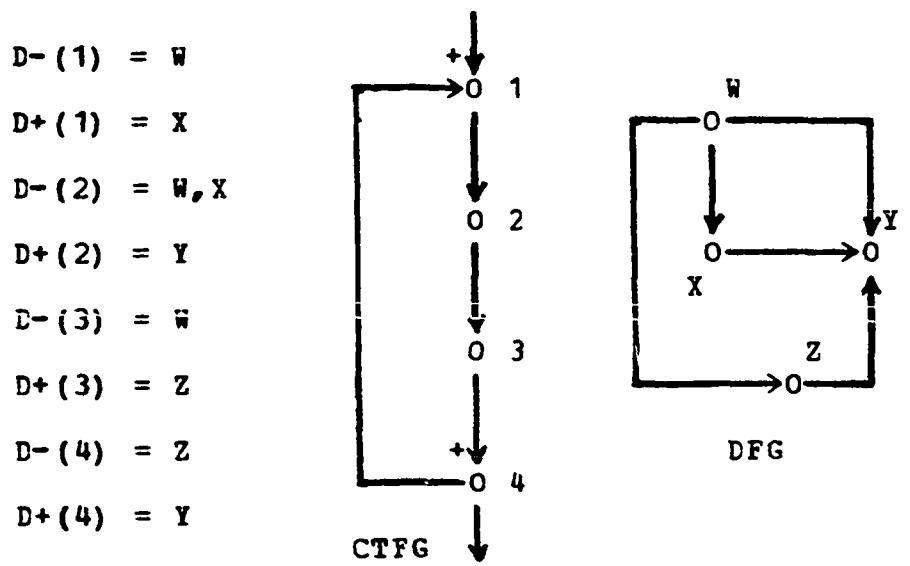

Figure 3. A Non-repetition Free GMC.

Consider the example in Figure 3. In this example $D^+(3)$ intersect $D^-(4)$ is Z. If we only considered decision vertices, we would say that the graph is repetition free. However, since W is input to vertex 3 and W is not in the output data set of any other vertex in the loop, vertex 3

would always compute the same value for Z and vertex 4 would make the same decision every time. Hence, the control flow graph (CTFG) is not repetition free (RF).

The original definition for repetition free is somewhat restrictive, however. Remembering that the purpose of the repetition free property is to eliminate infinite loops, vertices in a loop which do not affect the outcome of any decision vertices in the loop need not be considered. We do not need to consider vertices which are not decision vertices and whose output data sets do not intersect with the input data set of any other vertex in the loop. All other vertices in the loop must be considered. Another way of defining repetition free (RF) is that there must be a loop in the DFG which includes one or more cells of the input data set for at least one decision vertex in the CTFG loop. There is no such loop in the DFG in Figure 3 which includes Z. Hence the example in Figure 3 is not RF.

F. Proper Termination (PT)

Proper termination (PT) is a property which describes the state of the token machine when it halts. It is an interesting property in and of itself but is most useful to the GMC in its relation to deadlock freeness (14,25,27,35,36,38, 39,41,50,59). Deadlock freeness is, in turn, a desirable property for determining the correct operation of the system being modeled by the GMC and hence the GMC itself. Gostelow

(32) has shown that if resource allocation and deallocation arcs are included in the control flow graph (CTFG), then a proper terminating (PT) computation flow graph (CFG) implies a deadlock free system.

A CFG is said to be proper terminating if the token machine halts with a token on the exit arc of the CTFG and no extraneous tokens left on the CTFG. This is a way of saying that there is a path from every vertex in the CFG to the exit (terminal) vertex called T. There must be only one T. Yet another way to say that the CFG terminates properly (has the PT property) is to say that there exists some vertex T of the CFG which is the maximum of the CFG. A vertex in the CFG (which is a token state) is said to be maximum if it can be reached from every other vertex (state) in the CFG. Following Cerf (14):

A CFG is said to be proper terminating

(1) if it is finite and

(2) if T is the maximum of the CFG.

The token machine is proper terminating if it halts and leaves a token on arc T, the maximum of the CFG.

G. The Reduction Procedure

We have now determined that it is desirable to have properly terminating control flow graphs. So far, the only way to determine the PT condition is to execute the CTFG to find all possible execution sequences. Just generation of

the CTFG and particularly the CFG is a costly process in time and space. The CFG may be described by use of what are called transformation expressions (TE's). They define the rules by which one state of the CFG can evolve intc another. The transformation expressions associated with a CFG are called a TE set. When the following reductien procedure is applied to a TE set, a new TE set is prcduced. The reduction procedure as applied to TE sets is a much more efficient and hence more desirable method for analyzing the CFG and the CTFG. We will give only an overview since a thorough treatment is found in Cerf (14).

Since PT is not defined for multiple entry-multiple exit (MEMX) graphs, the first task is to determine whether the TE set represents a single entry-single exit (SESX) CTFG. This step consists simply of checking the left and right hand sides (LHS and RHS) of the productions in the TE set to see that there is one and only cne production of the form "E->a" and "a'->T" for some E, T, a, and a' in the CTFG. E and a' appear only once in the LHS and not in the RHS. T and a appear only once in the RHS and not in the LHS. E is the entry arc and vertex a the entry vertex. Vertex a' is the exit vertex and T the exit arc.

The reduction procedure operates on transformation expression sets (TE sets) to produce new TE sets. If a TE set can be reduced to the point that "E->T" is the only produc-

tion left, then the original TE set is said to be completely reducible (CR) and is also PT. It is possible to describe more complicated control structures with the transformation expressions then with the computation flow graph. Consequently, all completely reducible transformation expression sets are proper terminating but not all transformation expression sets that are properly terminating are also completely reducible. This unfortunate consequence will be explained further later.

The procedure itself is a process of substituting one TE into another to produce a new TE set. Cerf has a whole collection of theorems which lead up to three rules which describe under what conditions a TE can be substituted and not create any new terminal states. Infinite and potentially infinite TE sets are noted. The procedure is to apply the three rules in order until no further reductions can be made. If only one production is left and it has the form "E->T", then the original set was CR, completely reducible.

Rule (a) of the reduction procedure describes elimination of transformation expressions (TE's) by cancellation. LHS stands for left hand side and RHS stands for right hand side.

Let P(c) be the original set of TE's and Q the new set of TE's.

Let N = |P(c)| and P(c) = [t(1),t(2),...,t(N)].

```
1.  i = 0; Q = P(c);
2.  i = i + 1; If i > N then halt;
3.  a = LHS(t(i)) intersect RHS(t(i));
4.  If a = ϕ then go to 2;
5.  If LHS(t(i)) = RHS(t(i))
    Then do;
       Q = Q - [t(i)];
       If there exists a b such that for every
          t, b is an element of a, t is an ele-
          ment of Q, and the intersection of b
          and the RHS(t) is ϕ, then go to 2;
       If there exists a t such that t is an
          element of Q and LHS(t) = a,
       Then go to 2;
       Q = A union [t(i)]; go to 2;
    End;
6.  If LHS(t(i)) = a then go to 2;
7.  If RHS(t(i)) = a then go to 2;
8.  If there exists a t such that t is an ele-
          ment of Q and
    LHS(t) = LHS(t(i)) - a and
    RHS(t) = RHS(t(i)) - a
       Then do;
          Q = Q - [t(i)];
          Go to 2;
```

End;

In steps 3 and 4, transformation expressions (TE's) which do not have some subset of the RHS and LHS which is identical are skipped. Step 5 eliminates TE's with identical left and right hand sides. Steps 6 and 7 skip TE's in which one side is a subset of the other side and has not been eliminated in step 5. Step 8 tests the remaining TE's for cancellation. Rule (a) is based on two theorems proved in Cerf.

Theorem 6.2.5 Cerf(14).

Let P be a set of TE's, and let t be an element of P such that t: ab->ag. It is a necessary condition for cancellation to leave the CFG unaffected that at least one of two conditions hold:

(1) For all states, S, in the CFG b is a proper subset of S implies ab is a proper subset of S or

(2) There exists a t' which is an element of P such that t': b->g.

Theorem 6.2.7 Cerf(14).

A sufficient condition for a transformation expression (TE) of the form

t: a->a to be eliminated from a TE set P with-

out affecting the terminal states of the CFG is that either

(1) There exists a b such that for every t', b is an element and t' is an element of P - [t] implies b intersect RHS(t') = ϕ or

(2) There exists a t' such that t' is an element of P - [t] and LHS(t') = a.

Rule (b) of the reduction procedure describes the elimination of TE's with identical LHS's in P(c).

If there exists a D and an R such that

D $\neq \phi$, R $\neq \phi$, D is a proper subset of P(c),

R is a proper subset of P(c),

For every d and for every d' such that d and d' are elements of D implies LHS(d) = LHS(d'),

And for every d and for every r such that d is an element of D and r is an element of R implies LHS(d) is a subset of RHS(r),

Then do;

For every r such that r is an element of R implies T(r) = the set r(D);

T = union over r of T(r);

Q = P(c) - D - R union T;

If for every d and for every t such that d is an element of D and t is an element of

Q implies LHS(d) intersect LHS(t) = ¢ and

LHS(d) intersect RHS(t) = ¢,

Then P(c+1) = Q;

Else fail;

End;

Else fail;

P(c+1) consists of P(c), less the set of TE's (D) with identical left hand sides, less the set of TE's (R) for which the LHS(d) is a proper subset, and included are the new TE's with the LHS(d) substituted in the RHS(r) for each d and r in D and R, respectively. D and R may have one or more elements.

Rule (c) of the reduction procedure describes substitutions performed on the RHS of a TE whose LHS is E where E is the entry arc. E, I, and L lists (ELIST, ILIST, LLIST) are defined. ELIST holds transformation expressions whose left hand sides contain the entry arc. The ILIST contains transformation expressions that have a potential of being infinite. The LLIST contains transformation expressions that are in a loop. ELIST, ILIST, and LLIST are set to null before the reduction procedure starts.

1. If there exists a t and a d such that t and d are elements of P(c), LHS(t) = E, RHS(t) is not an element of the union of ILIST and LLIST, and LHS(d) is a subset

of the RHS(t),

Then for every d,

If d is an element of P(c) and the LHS(d) is a proper subset of the RHS(t) then set T(t) to T(t) union the set E->RHS(t)RHS(d)-LHS(d) and mark d to show that rule (c) was used;

2. For every t',

If there exists an e such that e is an element of ELIST and t' is an element of T(t) and RHS(t') = e,

Then do;

Set LLIST TO LLIST union [RHS(t')];

Mark t' to show RHS(t') is a loop state;

End;

3. Set P(c+1) to (P(c) - [t]) union T(t);

4. For every t',

If there exists an e such that e is an element of ELIST and t' an element of T(t) and e a subset of RHS(t),

Then do;

Set ILIST to ILIST union [e];

Mark t' to show it is potentially infinite;

5. For every t,

   If t is an element of P(c+1) and LHS(t) = E

   Then set ELIST to ELIST union [RHS(t)];

ELIST, ILIST, and LLIST are initialized to zero before the reduction procedure begins. ELIST keeps track of the RHS of all TE's with a LHS of E. The ILIST records RHS's of TE's that are potentially infinite and the LLIST holds the RHS's of TE's that are in a loop. In step 1 a TE is found whose LHS is E, whose RHS is in neither the ILIST or the LLIST, and for which a TE exists that can be substituted into its RHS. T(t) is the collection of RHS's for the single substitutions. These TE's are marked but not deleted from P(c). Step 2 checks T(t) for states previously created in this substitution. Any found are placed on the LLIST. In step 3 P(c+1) is created. Step 4 puts states on the ILIST which are subsets of previously generated states. Step 5 places the RHS of TE's in P(c+1) with a LHS of E on the ELIST.

The reduction procedure consists of applying rules (a) through (c) to a transformation expression set which represents a single entry-single exit (SESX) control flow graph. The prccedure preserves the terminal states of the CFG and stops when no rules can be applied.

If the TE set can be reduced to a single TE, E→T where E is the entry arc and T the terminal arc, the TE set is completely reducible. Completely reducible (CR) TE set must be

proper terminating (PT) since the reduction procedure shows that there is a path from every vertex to T, the terminal vertex, and that the CFG is finite (14). All possible infinite states and infinite loop states have been found by the reduction procedure and the TE set marked non-completely reducible (non-CR) if any exist. Figure 2d is the reduction of the TE set in Figure 2c.

H. Analysis in the Extended GMC

In designing the extensions to the GMC to make it more adaptable to modeling existing systems, care was taken to preserve the properties and analysis so eloquently presented in Cerf (14) and Gostelow (32). The GMC presented is extended and somewhat more restrictive. For proper termination analysis and the reduction procedure, the broken arcs may be treated no differently from any other arc. Broken arcs are restricted to having Q values of 1. The data flow graph (DFG) has no part in the analysis for proper termination (PT) nor do parameter lists defining the intersection of DFG's. Parameter arcs are restricted to being single arcs with Q values of 1. Intertask communication is restricted to broken arcs. Broken arcs are required to have "*" logic. The computation flow graph (CFG) and control flow graph (CTFG) are unchanged except the CTFG is somewhat more restricted in order to define more precisely how tasks interact.

I. Determinacy

There are several interesting aspects to determinacy. Within the confines of the extended GMC it is possible to easily analyze for conflicts in variable usage among parallel tasks. This is one of the reasons for defining parameter lists in the manner described in the extended model. However, in addition to resolving conflicts, it would be desirable to be able to prove for a GMC that a given set of input variables will always produce the same output. This property is called output determinacy. Output determinacy is a less restrictive form of determinacy.

A sufficient set of conditions for determinacy is that for every pair of vertices, call them i and j, either the execution of one follows the execution of the other or they may execute in parallel. If they can execute in parallel, then all of the following three conditions must hold:

1. $A^-(i)$ intersect $A^+(j) = \phi$
2. $A^+(i)$ intersect $A^-(j) = \phi$
3. $A^+(i)$ intersect $A^+(j) = \phi$

These are known as Bernstein's conditions (7). It should be obvious that these are sufficient and not necessary. However, determining necessity requires a partial or complete specification of the interpretation. For example, Adams (1) and Rodriguez (64) have designed their models such that the equivalent of tokens, representing instances of execution,

flow along the arcs or edges of a graph which is essentially a data flow graph with additional control structures specified to force sequencing in the data flow graph (DFG). They both show determinacy in their respective models. A complete or partial specification of the interpretation seems undesirable in that a change in the interpretation requires a new complete analysis. Likewise, an exhaustive execution with various inputs is undesirable as a tool to design programs. Their approach does achieve determinacy however.

Just showing determinate output is an interesting problem. Cerf (14) brings up several conflicts. The desirability of allowing mutual exclusion in the model implies the existence of variables in conflict among parallel tasks. The order of the use of these variables in conflict, even though mutually excluded, is indeterminate. Since variables are allowed to be in conflict, Bernstein's conditions are violated. Hence, we cannot rely on Bernstein's conditions to show determinacy. To actually show determinate program behavior we must know something about the computations performed by the mutually excluded vertices. Consider also a linked list. In most cases, the order of the elements in a linked list is unimportant. Because of the order of execution, two identical sets of input could cause an algorithm to operate on a linked list in such a way as to leave the same elements in the list in both cases but in a different order. Consider

two tasks that compete for the use of a resource such as core memory for which the allocation list is linked. There is nothing in the model to guarantee a specific ordered pattern for allocation and deallocation of elements in the list, yet the list, regardless of the order, still describes the same resources and would be considered correct in both cases though not identical. The arbitrary choice of vertices in step 4 of the token machine allows this desirable (or undesirable depending on your point of view) anomaly.

IV. THE MODELING POWER OF THE EXTENDED GMC

A. The Modeled Constructs

In this chapter we demonstrate the modeling of various programming constructs used in operating systems. Constructs found in programming which are not task oriented include sequential execution, decisions, loops, branches, subroutines, recursion, parameter passage and environments. In addition to the constructs found in sequential programming, constructs found in task oriented programming include parallel execution, both ENQ-DEQ (mutual exclusion) and WAIT-SIGNAL types of task synchronization, FORK-JOIN or ATTACH-DETACH task creation and deletion, parameter passage at task creation and deletion, mutual exclusion, resource allocation, interrupts, and environment management.

B. Basic Programming Constructs

Sequential execution is modeled with vertices or computations connected with arcs to indicate flow of control. Each vertex has an input and an output data set. A DFG represents the flow of data between vertices. "+" output vertex logic is used for decisions and branches. "+" input vertex logic is used for loops. All these have been demonstrated previously.

C. Reentrant and Non-reentrant Subroutines

There are several possibilities for modeling subroutines. Let A be a subroutine in Figure 4a. This figure represents a close approximation to what is frequently implemented in programming. The vertices marked C represent calls to A and the vertices marked R are returns from A. Xi represents return information. F is the entry vertex to the subroutine and G is the exit vertex. In actual fact the arcs marked Xi are redundant because vertex G must make a decision by virtue of its output logic as to the arc on which to place a token. If an incorrect decision is made, improper return will result; or, if arcs Xi are left in, deadlock will result. The reduction procedure produces these transformation expressions (TE's) which cannot be reduced:

$$E \rightarrow (X1, PE2) \text{ and } E \rightarrow (X2, PE1)$$

X1 and X2 may be eliminated but then the construct does not operate as desired.

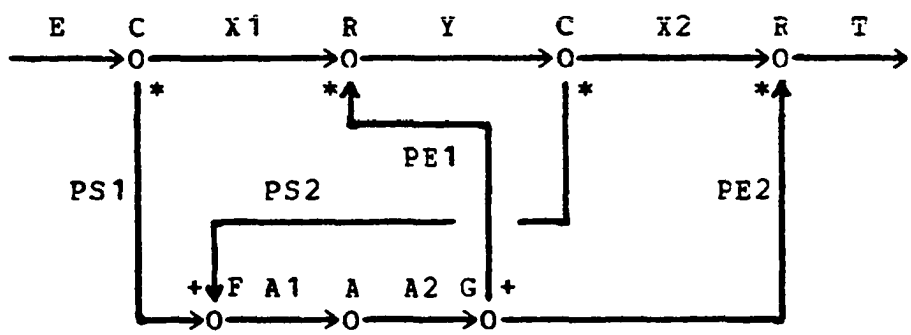

Figure 4a. An Incorrect Model of a Subroutine.

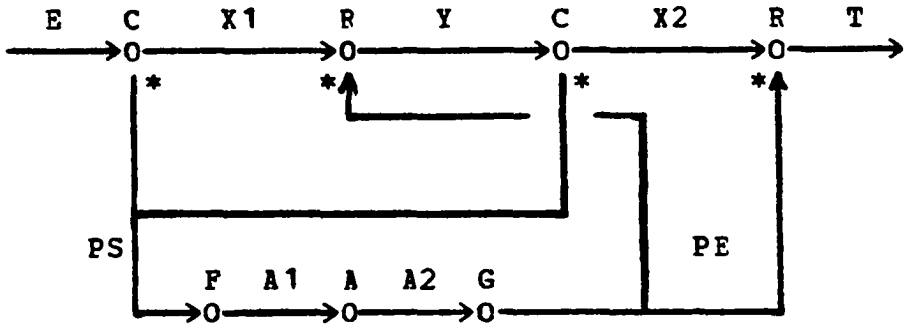

Figure 4b. The CTFG for a Parameterless Subroutine.

The solution to the non-completely reducible (non-CR) problem is to combine PS1 with PS2 and PE1 with PE2 to make multiarcs as in Figure 4b. This graph is completely reducible (CR) and hence is proper terminating (PT). It does operate as desired. PS and PE are parameter arcs and F and G the entry and exit vertices, respectively, to subroutine A.

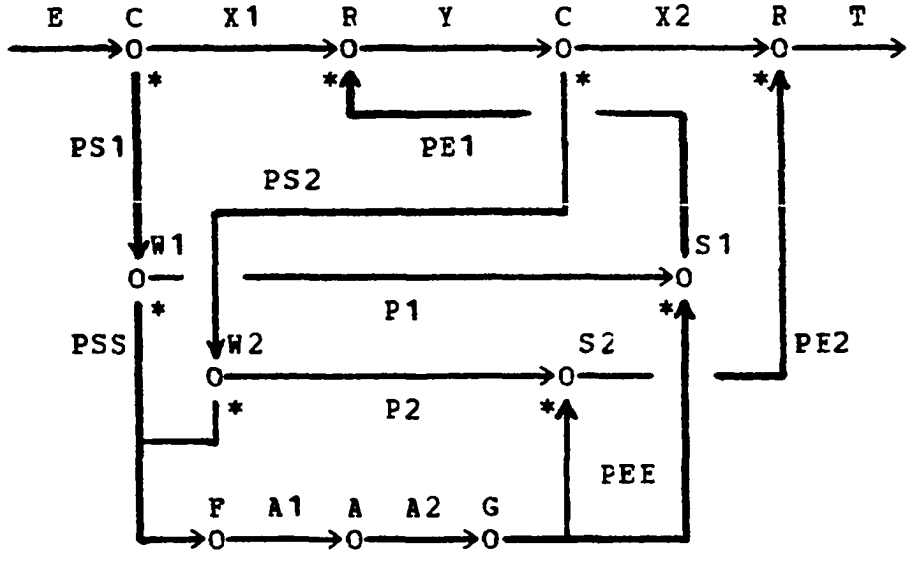

Figure 4c. The CTFG for a Subroutine.

The problem encountered with the control flow graph (CTFG) in Figure 4b is not with the correctness of the model but with the accuracy with which we model subroutine calls with parameters in operating systems. Suppose PS had been multi-headed as well as multi-tailed. The other head might perhaps be a call to a subroutine B or another entry point to subroutine A. The choice is arbitrary as to which of the several vertices at the heads of arc PS is chosen, a rather unhappy programming condition. The same is true if PE could be multi-tailed. What is actually desired is that an entry arc cannot be multi-headed and an exit arc cannot be multi-tailed. This would work fine as long as we have to define only one set of parameters to associate with PS or PE. However, consider the case with a reentrant subroutine where PS may have multiple tokens on it representing several calls with different environments and parameter lists to the same subroutine. What do we do with all the parameter lists since we only have one arc? The simplest way is to force separate arcs for each call with separate parameter lists. There are many cases where multi-headed and multi-tailed arcs are desirable but not in this context. Figure 4b is correct as long as PS and PE are not parameter arcs. However, Figure 4c is the best choice for modeling calls where parameters are passed. The vertices marked Si and Wi are the initiation and termination vertices for the subroutine. P1 and P2 force

proper return.

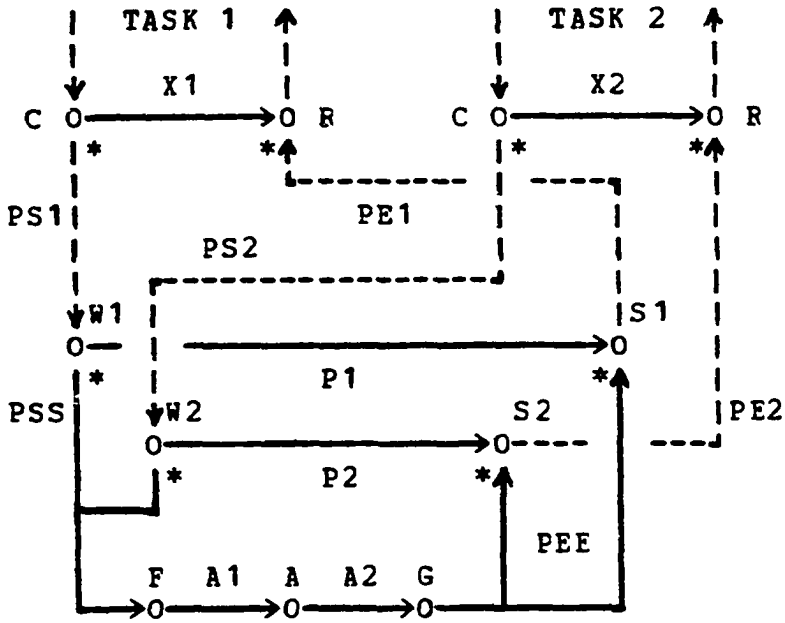

Figure 4d. The CTFG for a Reentrant Subroutine.

Figure 4c is repetition free because of the sequential nature of the calls at the vertices marked C. Consider, however, the reentrant case in Figure 4d. PSS can be initiated twice without terminating. Therefore, the control flow graph (CTFG) in Figure 4d is not repetition free (RF). A mutual exclusion arc, MX, as in Figure 4e is required to force sequencing even though the sequencing is arbitrary. Mutual exclusion over the whole subroutine could be accomplished by not having MX connected to every vertex. A desirable property, however, is to allow as much parallelism as possible. The arc, MX, as drawn allows only one vertex in the subrou-

time to execute at a time while what is really desired is for any vertex to execute in parallel with any other for the separate tasks and, of course, separate environments. This does, however, violate the repetition free (RF) property as it is defined. Perhaps a redefinition would be possible which includes some indication of the environment. The restriction of repetition free (RF) as it is defined is not a serious hindrance to the modeling of reentrant subroutines.

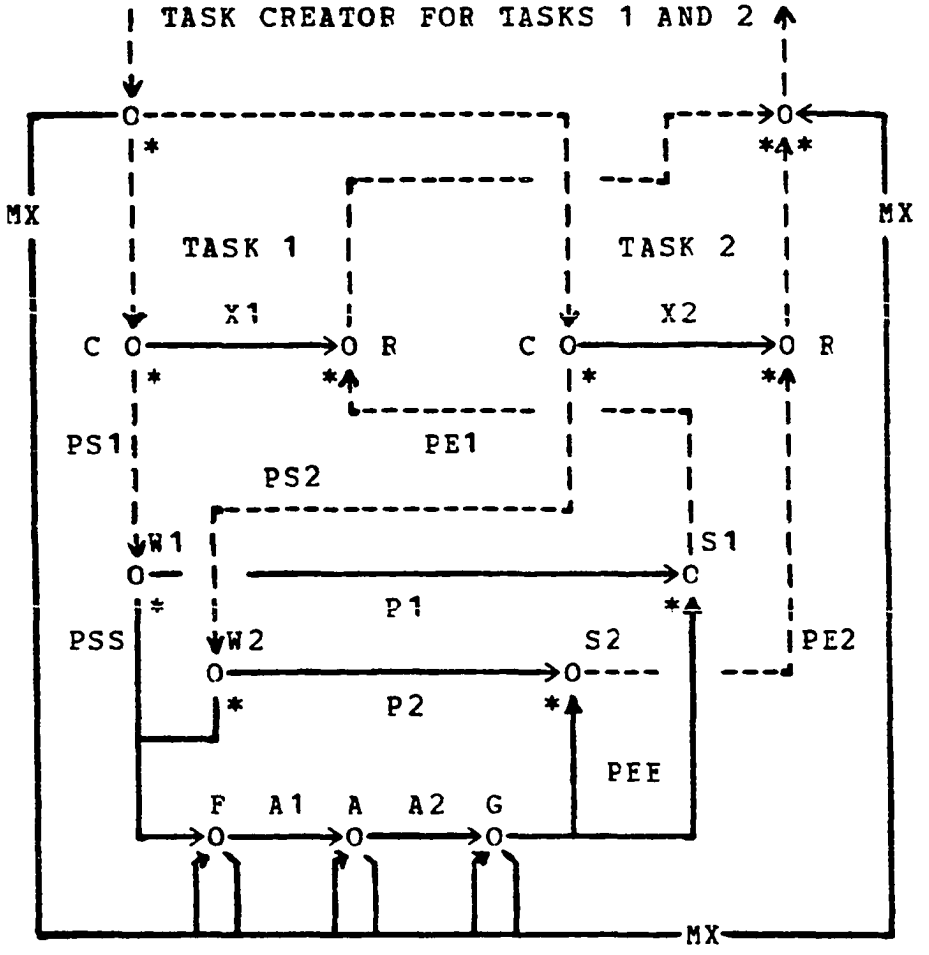

Figure 4e. A Reentrant Subroutine with the MX arc.

D. Environments and Parameter Passage

In currently existing operating systems, environments play a very important role in determining the flow of control for a program or system. For example, if two tasks agree to mutually exclude each other for access to specific resources, each task must request a service of the operating system called ENQ-DEQ. Of course, mutual exclusion may be accomplished with WAIT-SIGNAL or with some kind of a test and set operation. All, however, make use of data in fundamentally the same way. A mutually agreed upon memory location, logical name, etc. is tested and then set one way for one task to have access to or control of the resource or set another way for the other task or tasks to have control of the resource. Hence, these types of control mechanisms do not alter the flow of control of a task but represent timing constraints. Modeling such task dependencies becomes a problem without the inclusion of the environment. Hence, the environment has several functions which vary from data for arithmetic operations to control data. Program control is difficult to model without environments and vice versa. However, for analysis such as for a proper termination (PT) condition, these data dependencies need to be placed in the CTFG.

Return mechanisms from subroutines or tasks, in point of fact, are data areas (possibly a hardware or software stack) where return pointers or addresses are stored. These return

pointers or mechanisms must be modeled as part of the CTFG so analysis for a proper termination (PT) condition can be performed.

We have problems trying to model situations where a WAIT is performed on a variable that will be signaled by a task which is not in memory or even created yet. The choice made here is that the CTFG, the DFG, etc. for every task must be known to do a PT analysis. About the most that can be done to solve this problem is that the token machine may be modified to substitute a properly terminating GMC for a vertex. Cerf (14), Gostelow (32), and Yavne (78) have worked on a set of replacement rules for vertices. The static nature of the CTFG does not represent a serious restriction to modeling operating systems, particularly with the environment as defined. Very few if any routines or pieces of code for operating systems are built at execution time. Consequently the whcle system is known for analysis purposes.

Parameter passage is yet another case of a control related mechanism which is actually in the environment. In the modified GMC, executing a vertex which is a call sets the parameter list for the output arc from the vertex. If the arc is a broken arc, we know that the call is an inter-task type call. The next vertex logically follcwing the call (the first vertex of the subroutine) must take the parameters from the parameter arc and store them in his DFG environment data

area (one may have to be allocated). A new DFG environment may need to be established. The modified CMC is defined so that the parameter arcs and lists are the only variables in common between subroutines or tasks. In this way an automatic separation of the environments is achieved. Environments may vary with CTFG's as the CTFG's are executed multiple times. Each time a task is entered, a new environment is created for it from its associated DFG. Since this is the case, there is no need to model reentrant or recursive subroutines with multiple copies or multiple CTFG's. We still must have repetition free CTFG's, however. The environment separation concept is necessary to be able to model parameter passage, to easily analyze for variable conflicts, and for the modeling of the scope of variables such as in recursive or block structured programming.

E. Recursion

The problem of modeling recursion or programming which can call itself is related to the repetition free property and multiple control levels in the process of recursing. Local variables must be allocated again each time a routine calls itself to avoid reusing these variables. The same is true in the case of a reentrant subroutine as was pointed out earlier. Proper termination can be shown as long as we know that the recursion will stop. This is guaranteed by the repetition free property. The structure exists in the DFG to

allocate local variables each time the recursive subroutine is entered and reuse the same CTFG without having to modify the CTFG and tack on a copy of the recursive subroutine. Repetition free would have to be redefined somewhat, however, to allow the same vertex to be initiated twice in succession as long as the vertex had different sets of input and output variables.

F. Task Creation and Communication

In general "*" output logic with broken arcs interfaces to or creates tasks and "*" input logic with broken arcs interfaces to or deletes tasks. In the definition of the modified GMC we force broken arcs to have "*" logic. If the vertex at the head of a broken arc has only the broken arc for input to the vertex, then the broken arc creates a task. Otherwise, the broken arc represents task interface. Likewise, if the vertex at the tail of a broken arc has only the broken arc for output to the vertex, then the broken arc deletes a task. Otherwise, the broken arc represents task interface. Figure 5 shows the four possibilities. With these four constructs and parameter passage, the constructs ENQ-DEQ, WAIT-SIGNAL, and FORK-JOIN or ATTACH-DETACH are readily modeled.

"*" logic is not restricted to tasks. As we have seen, it is used in modeling subroutines and in other control mechanisms such as mutual exclusion. Mutual exclusion is not

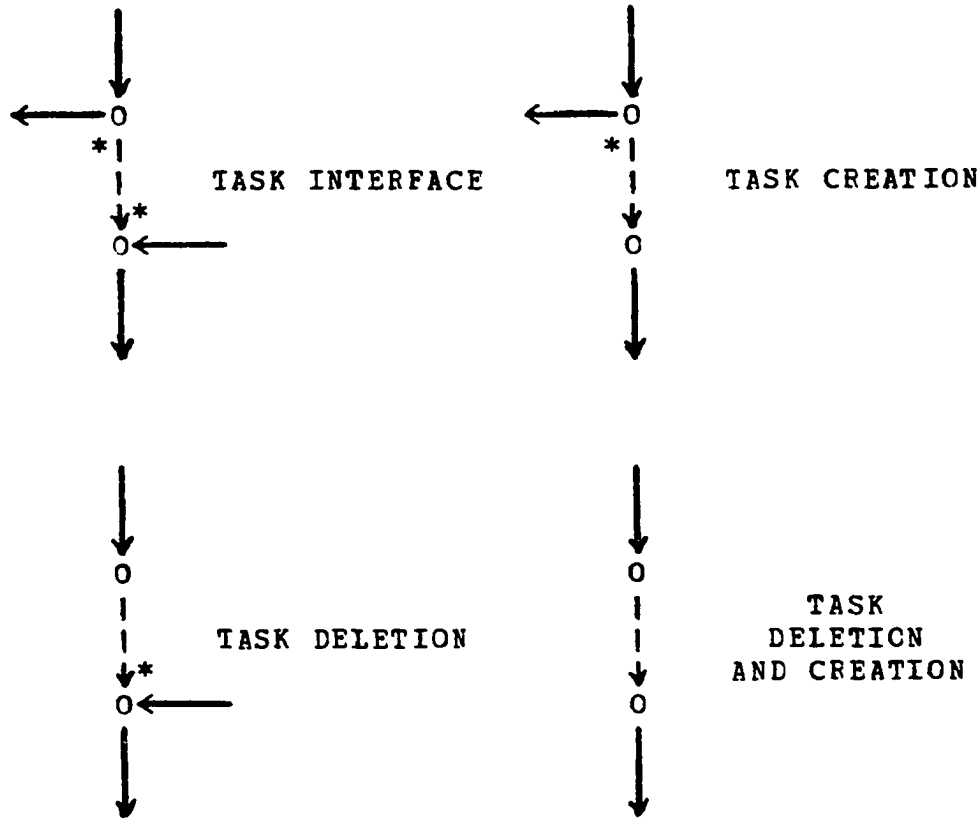

Figure 5. The Four Constructs for Task Interface.

required except between tasks since a task, itself, can only perform one operation at a time. The ENQ-DEQ or WAIT-SIGNAL constructs may be used to mutually exclude. It may take the form of a multiarc with a token which is used and replaced as various tasks use a resource. Broken arcs may not be multiarcs so to accomplish mutual exclusion we must turn to a construct such as the mutual exclusion arc, MX, in Figure 1b. This is again another type of task communication.

G. Resource Allocation

Resource allocation is covered to some extent by Gostelow (32). In resource allocation a multiarc connected to a task at all the vertices where resources are allocated and deallocated is used to hold an available resource count. Resources are used (taken from) and then replaced on the arc. Since resource allocation is generally done by one task, we have only to interface with a broken arc to a routine that allocates resources from resource count holders or multiarcs. Resources with unique names would have to have an arc for each resource.

H. Interrupts

As with resource allocation Gostelow (32) has suggested a way to model interrupts. This consists essentially of performing a little subroutine at the vertex that caused or requested the interrupt and then returning control to the task. We could think of an interrupt as a task requesting a service from the operating system task. Programmable interrupts can be interfaced to the operating system with broken arcs. For the non-programmable ones, the token machine can be modified to connect the interrupting vertex to the operating system task in a predefined way, much as is done by the current hardware. Hardware locations are generally reserved in one way or another for fielding interrupts.

I. Environment Management

To this point in time the structure for environment management has been defined as a task list of active tasks which point to CTFG's and DFG's. The first vertex to be executed in each task is required to accept the parameters passed from the calling task and set up a new environment. The subset of items actually done by the first vertex and the subset done by the token machine for the new task makes no difference. The things that must be accomplished at task interaction time are well defined. The task list must be updated, parameters passed and accepted, a DFG created or modified, and storage allocated.

V. CONCLUSIONS

The intent of the research was to develop a model that could be used to model task oriented systems and further model essentially all the programming constructs. A model, the UCLA Graph Model of Computation was chosen as a base and modified to accomplish the intended purpose. The GMC is relatively simple, concise and easily understood. Some unsolved problems still remain.

The extended GMC incorporates the concepts of environments and data flow within environments. The GMC control structure (CTFG) is static in nature with the dynamic part of the model being in the environment. With the environment came a definition for parameter arcs and the ability to separate tasks on an environmental basis. Hence, parameter passage and multiple executions of a CTFG with different environments became possible.

In chapter III we formally defined the modified GMC and the token machine. Various properties such as repetition free, proper termination, and the reduction procedure were defined showing the ability to do a partial analysis for correct operation and for a property called deadlock free. In chapters II, III, and IV we have addressed task communication mechanisms such as WAIT-SIGNAL, ENQ-DEQ, and FORK-JOIN, and we have addressed programming constructs such as parallelism, mutual exclusion, subroutines, return

mechanisms, recursion, reentrancy, loops, and decisions. We have included environment management, resource allocation, interrupts, and much time was spent defining environments with parameter passage. Thus, we concluded that essentially all the necessary programming constructs required to model sequential execution and parallel oriented task systems can be modeled with the extended GMC.

The next step would be to design and implement a parallel processing, perhaps interactive, language in which one could design systems of programs and have them analyzed for correctness.

A better definition which is less restrictive is needed for repetition free in relation to reentrant and recursive subroutines. Determinacy, not only the definition, but also the analysis for the property is difficult. The modified GMC separates environments and forces a better definition of how environments interact but still the problem of determinacy versus mutual exclusion is not solved. If output determinacy is what is desired, then what properties of the GMC produce it? Over all the GMC provides a good foundation for modeling work for parallel task oriented systems. Although the discussion has been oriented toward computer operating systems, no assumptions have been made in the design of the extended GMC which restrict its use specifically to operating systems.

Any type of system which is parallel task oriented should be readily modeled in the extended GMC.

VI. BIBLIOGRAPHY

1. Adams, D. A. A computation model with data-flow sequencing. Ph.D. dissertation. Stanford University, 1968. (Technical Rep. No. STAN-CS-117)

2. Adams, D. A. A model for parallel computations. In Parallel processor systems, technologies, and applicatons, pp. 311-334. Edited by L. C. Hobbs. New York: Spartan Books, 1970.

3. Baer, J. L. Graph models of computaticns in computer systems. Ph.D. dissertation. University of California at Los Angeles, 1968. (Technical Rep. No. UCLA-10P14-51)

4. Baer, J. L. A survey of some theoretical aspects of multiprocessing. Computing Surveys 5, No. 1 (Mar. 1973): 31-80.

5. Baer, J. L., Bovet, D. B., and Estrin, G. Legality and other properties of graph models of computation. Journal of the ACM 17, No. 3 (1970): 543-554.

6. Berge, C. Theory of graphs. New York: Methuen & Co., Ltd. (Barnes and Noble), 1962.

7. Bernstein, A. J. Analysis of programs for parallel processing. IEEE Transactions on Computers C-17 (Aug. 1968): 746-757.

8. Bovet, D. B. Memory allocation in computer systems. Ph.D. dissertation. University of California at Los Angeles, 1968. (Technical Rep. No. UCLA-10P14-Y68-17)

9. Bredt, T. H. Analysis of operating system interactions. Proceedings of the AICA Congress on Theoretical Informatics, Institute for the Elaboration of Informaticn, University of Pisa, Mar. 1973. pp. 253-281.

10. Bredt, T. H. Control of parallel processes. Ph.D. dissertation. Stanford University, 1970.

11. Bredt, T. H. The mutual exclusicn problem. Technical Rep. No. 9. Digital Systems Laboratory, Stanford University, Aug. 1970.

12. Brinch Hansen, P. The nucleus of a multiprogramming system. Communications of the ACM 13, No. 4 (Apr. 1970): 238-241, 250.

13. Cerf, V. G., Fernandex, E. B., Gostelow, K. P., and Volansky, S. A. Formal control flow properties of a graph model of computation. Technical Rep. No. ENG-7178 (UCLA-10P14-105). Computer Science Department, University of California at Los Angeles, Dec. 1971.

14. Cerf V. G. Multiprocessors, semaphores, and a graph model of computation. Ph.D. dissertation. University of California at Los Angeles, 1972. (Technical Rep. No. UCLA-10P14-110)

15. Conway M. E. A multiprocessor system design. In 1963 Fall Joint Computer Conference, AFIPS conference proceedings 24, pp. 139-146. Washington, D. C.: Spartan, 1963.

16. Courtois, P. J., Heymans, R., and Parnas, D. L. Concurrent control with 'readers' and 'writers'. Communications of the ACM 14, No. 10 (Oct. 1971): 667-668.

17. Davis, M. Computability and unsolvability. New York: McGraw-Hill, 1958.

18. Denning, P. J. Third generation computer systems. ACM Computing Surveys 3, No. 4, Dec. 1971.

19. Denning, P. J. Working set model for program behavior. Communications of the ACM 11, No. 5 (May 1968): 323-333.

20. Dennis, J. B. On the design and specification of a common base language. Computation Structures Group Memo No. 60. Project MAC, Massachusetts Institute of Technology, July 1971.

21. Dennis, J. B., and VanHorn, E. C. Programming Semantics for multiprogrammed computations. Communication of the ACM 9, No. 3 (Mar. 1966): 143-155.

22. Dennis, J. B., and Patil, S. Computation structures course notes for Course 6.232. Department of Electrical Engineering, Massachusetts Institute of Technology, 1970.

23. Dijkstra, E. W. Cooperating sequential processes. In Programming languages, pp. 43-112. Edited by F. Genuys. New York: Academic Press, 1968.

24. Dijkstra, E. W. Hierarchial ordering of sequential processes. Acta Informatica 1 (1971): 115-138.

25. Dijkstra, E. W. Solution of a problem in concurrent programming control. Communications of the ACM 8, No. 9 (Sept. 1965): 569-570.

26. Dijkstra, E. W. The structure of the programming system. Communications of the ACM 11, No. 5 (May 1968): 341-346.

27. Eisenberg, M. A. and Mcguire, M. R. Further comments on Dijkstra's concurrent programming control problem. Communications of the ACM 15, No. 11 (Nov. 1972): 999.

28. Firestone, R. M. Parallel programming: Operational model and detection of parallelism. Ph.D. dissertation. New York University, Oct. 1971.

29. Gilbert, P., and Chandler, W. J. Interference between communicating processes. Communications of the ACM 15, No. 3 (Mar. 1972): 171-176.

30. Gill, A. Introduction to the theory of finite state machines. New York: McGraw-Hill, 1962.

31. Gill, S. Parallel programming. Computer Journal (Apr. 1958): 2-10.

32. Gostelow K. P. Flow of control, resource allocation, and the proper termination of programs. Ph.D. dissertation. University of California at Los Angeles, Dec. 1971. (Technical Rep. No. UCLA-10P14-106)

33. Gostelow, K. P. Proper termination of flow-of-control in programs involving concurrent processes. Proceedings of the ACM 2 (Aug. 1972): 742-754.

34. Gostelow, K. P. Transformation systems. Internal Memorandum No. 114. Computer Science Department, University of California at Los Angeles, May 1973.

35. Habermann, A. N. On a solution and a generalization of the cigarette smokers' problem. Computer Science Department, Carnegie-Mellon University, Aug. 1972.

36. Habermann, A. N. Prevention of system deadlocks. Communications of the ACM 12, No. 7 (July 1969): 373-377.

37. Hack, M. H. T. Analysis of production schemata by Petri nets. Technical Rep. No. MAC TR-94. Project MAC, Massachusetts Institute of Technolcgy, Feb. 1972.

38. Havender, J. W. Avoiding deadlock in multitasking systems. IBM Systems Journal 7, No. 2 (1968): 74-84.

39. Helbalkar, P. G. Deadlock-free sharing cf resources in asynchroncus systems. Ph.D. dissertation. Massachusetts Institute of Technolcgy, Sept. 1970. (Technical Rep. No. MAC-TR-75)

40. Holt, A. W., and Commoner, F. Events and conditions. New York: Applied Data Research Inc., 1969.

41. Holt, R. C. Some deadlock properties cf computer systems. ACM Computing Surveys 4, No. 3 (Sept. 1972): 179-196.

42. Hopcroft, J. E., and Ullman, J. D. Formal languages and their relation to automata. Reading, Massachusetts: Addison-Wesley, 1969. ACM Computing Surveys 5, No. 1 (Mar. 1973): 5-30.

43. Ianov, I. I. The logical schemes cf algcrithms. Problems of Cybernetics 1 (1968): 75-127.

44. IBM SYSTEM/360 Operating System Supervisor Services and Macro Instructions. Form GC28-6646-6. Poughkeepsie, N.Y.: IBM Corporation, 1972.

45. Izbicki, H. On marked graphs. Technical Rep. No. LR 25.6.023. Vienna, Austria: IBM Vienna Lab., 1971.

46. Johnston, J. B. The contour model of block structured processes. Proceedings ACM SIGPIAN Symposium on Data Structures and Programming Languages (Feb. 1971). ACM SIGPLAN Notices 6, No. 2 (Feb. 1971): 55-82.

47. Karp, R. M., and Miller, R. E. Properties cf a model for parallel computations: Determinacy, termination, queuing. SIAM Journal of Applied Math 14, No. 6 (Nov. 1966): 1390-1411.

48. Karp, R. M., and Miller, R. E. Parallel program schemata: A mathematical model for parallel computations. New York: IEEE Conference Record 1967 8th Annual Symposium on Switching and Automata Theory, Oct. 1967. pp. 55-61.

49. Karp, R. M., and Miller, R. E. Parallel program schemata. Journal of Computer and Systems Science 3, No. 4 (May 1969): 167-195.

50. Knuth, D. E. Additional comments on a problem in concurrent programming control. Communications of the ACM 9, No. 5 (May 1966): 321-322.

51. Kosaraju, S. R. Limitations of Dijkstra's semaphore primitives and Petri nets. ACM Operating Systems Review 7, No. 4 (Oct. 1973): 122-126.

52. Levitt, K. N. The application of program proving techniques to the verification of synchronization processes. In Research in interactive program-proving techniques, pp. 84-122. Edited by B. Elspas, C. Green, K. N. Levitt, and R. J. Waldinger. Menlo Park, California: Stanford Research Institute, May 1972.

53. Luconi, F. L. Asynchronous computational structures. Ph.D. dissertation. Massachusetts Institute of Technology, Feb. 1968. (Technical Rep. No. MAC-TR-49)

54. Martin, D. F. The automatic assignment and sequencing of computations on parallel processor systems. Ph.D. dissertation. University of California at Los Angeles, Jan. 1966. (Technical Rep. No. 66-4)

55. Martin, D., and Estrin, G. Experiments on models of computations and systems. IEEE Transactions of Computers EC-16, No. 1 (Feb. 1967): 59-69.

56. McCluskey, E. J. Introduction to the Theory of Switching Circuits. New York: McGraw-Hill, 1965.

57. Miller, R. E. A comparison of some theoretical models of parallel computation. IEEE Transactions on Computers C-22, No. 8 (Aug. 1973): 710-717.

58. Noe, J. D., and Nutt, G. J. Macro E-nets for representation of parallel systems. IEEE Transactions of Computers C-22, No. 8 (Aug. 1973): 718-727.

59. Parnas, D. L. On a solution to the cigarette smokers' problem (without conditional statements). Department of Computer Science, Carnegie-Mellon University, July 1972.

60. Patil, S. S. Coordination of Asynchronous Events. Ph.D. dissertation. Massachusetts Institute of Technology, Sept. 1970. (Technical Rep. No. MAC-TR-72)

61. Peterson, J. L. Modelling of parallel systems. Ph.D. dissertation. Stanford University, Dec. 1973.

62. Petri, C. A. Kommunikation mit Automaten. Ph.D. dissertation. Germany: University of Bonn, 1962.

63. Riddle, W. E. The modelling and analysis of supervisory systems. Ph.D. dissertation. Stanford University, Mar. 1972. (Technical Rep. No. STAN-CS-72-271)

64. Rodriguez, J. E. A graph model for parallel computation. Ph.D. dissertation. Massachusetts Institute of Technology, Sept. 1967. (Technical Rep. No. MAC-TR-64)

65. Russell, E. C. Automatic program analysis. Ph.D. dissertation. University of California at Los Angeles, Mar. 1969. (Technical Rep. No. UCLA-10P10-72)

66. Saltzer, J. H. Traffic control in a multiplexed computer system. Technical Rep. No. MAC-TR-30. Project MAC, Massachusetts Institute of Technology, July 1966.

67. Schroeder, M. D. A brief report on the SIGPLAN/SIGOPS interface meeting. ACM Operating Systems Review 7, No. 3, July 1973.

68. Slutz, D. R. The flow graph schemata model of parallel computation. Ph.D. dissertation. Massachusetts Institute of Technology, Sept. 1968. (Technical Rep. No. MAC-TR-53)

69. Slutz, D. R. Flow graph schemata. New York: Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, 1970. pp. 129-141.

70. Turn, R. Assignment of inventory of a variable structure computer. Ph.D. dissertation. University of California at Los Angeles, Jan. 1963. (Technical Rep. No. 63-5)

71. VanHorn, E. C. Computer design for asynchronously Reproducible Multiprocessing. Ph.D. dissertation. Massachusetts Institute of Technology, Nov. 1966. (Technical Rep. No. MAC-TR-34)

72. Volansky, S. A. Graph model analysis and implementation of computational sequences. Ph.D. dissertation. University of California at Los Angeles, June 1970. (Technical Rep. No. UCLA-10P14-93)

73. Volansky, S. A. Synthesis of properly terminating graphs. Internal Memorandum No. 113. Computer Science Department, University of California at Los Angeles, Jan. 1973.

74. Wallentine, V. E. An abstract machine to control the execution of semi-independent concurrent computations. Ph.D. dissertation. Iowa State University, Sept. 1972.

75. Wegner, P. The Vienna Definition Language. ACM Computing Surveys 1, No. 2 (1972): 5-63.

76. Wirth, N. On multiprogramming, machine coding and computer organization. Communications of the ACM 12, No.9 (Sept. 1969): 489-498.

77. Witt, B. I. Part II job and task management. IBM Systems Journal 5, No. 1 (1966): 12-29.

78. Yavne, M. Synthesis of properly termination graphs. Technical Rep. No. UCLA-34P214-2. Computer Science Department, University of California at Los Angeles, Apr. 1974.

VII. ACKNOWLEDGEMENTS

Most importantly I would like to thank Dr. Clair G. Maple, my thesis advisor, for his continued support and encouragement, patience and understanding throughout the research for this work and during its writing.

I would like to thank Dr. Arthur E. Oldehoeft for his guidance, suggestions, and aid in finding materials during the research. Mr. George F. Covert, Dr. Dale D. Grosvenor, Dr. E. Jerome Niebaum, Mr. Christian J. Reschly, Dr. George O. Strawn, and Dr. Charles T. Wright have provided much encouragement and aid in finding research materials.

Finally, I sincerely thank my wife, Linda, for putting up with me and my idiosyncrasies throughout the years of my graduate studies. Without her support and confidence it never would have been completed.