MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group Memo 64

Computation Structures Group
Progress Report 1970-71

January 1972

This research was done at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362-0001, and in part by the National Science Foundation under grant GJ-432.

A. Introduction

The Computation Structures Group is concerned with the study and analysis of fundamental issues arising in the design and construction of general-purpose computer systems. The research encompasses hardware and software aspects of computer systems, and much of the work has contributed toward establishing a common conceptual basis for both aspects. The accomplishments of the past year are principally in two areas: One is the theoretical study of Petri nets as a model for asynchronous systems of interacting parts, and the realization of Petri nets in the form of speed-independent modular switching systems. The goal of this work is to build a sound theory to serve as the basis of a new methodology for the design of asynchronous digital systems. The second area is the evolution of a base program language. This effort is expected to lead to a practical formal definition scheme for source programming languages and will provide a sound basis for the functional design of advanced computer systems.

B. Petri Nets

As reported last year, we have found Petri nets to be an elegant formalism for representation of concurrency in processes and for studying asynchronous systems. Petri nets stand out in relation to other schemes because of the preciseness and ease with which they can express parallel actions, resolution of conflicts, and interaction among processes. Moreover, they have the simple structure that is essential for analytic study. Simple as they are in their structure, study of the general class of Petri nets is difficult because of the variety of situations they can represent. A study of subclasses of Petri nets which represent simpler situations is a necessary step toward understanding the general class of Petri nets, and such study has been an important objective of the group in the past year. We have identified several subclasses of interest and have found useful results about them. Before discussing these results, we present a brief introduction to Petri nets and the subclasses of interest.

A Petri net [1,2] is a directed graph which can have two types of nodes, namely transitions and places, where the directed arcs can connect only transitions to places and places to transitions (Fig. 1.). In drawing the graph, places are represented by circles and the transitions by bars. The places from which arcs are incident on a transition are called input places of the transition, terminate are called the output places of the transition. Each place can have markers (sometimes called tokens) in them. A transition having markers in all of its input places is said to be enabled. Only enabled transitions can fire; in the act of firing, the transition picks one marker from each of its input places and puts a marker in each of its output places. The marking distribution in the net changes as transitions fire, and each new marking distribution makes firing of other transitions possible. With regard to the firing of transitions, an important situation is when

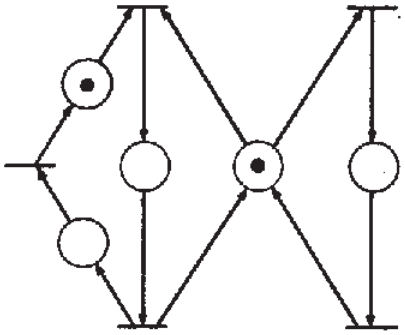


FIG. 1. A PETRI NET.

transitions share some input places. When two transitions which have a common input place are both enabled but the common input place has only one marker, the transitions are said to be in conflict because the firing of any one of the transitions disables the other. A net is said to be safe if no place in it will ever have more than one marker at a time. A net is said to be live if at no time in the operation of the net will any transition be ruled out as a transition that may fire some time in the future. Conflict, safety, and liveness in a net depend on the initial marking distribution. There are, however, some structural restrictions which can guarantee some of these properties. By structural restrictions, we mean restrictions with regard to the arrangements of transitions and places such as the restriction that transitions not have input places in common. The restrictions we use below to define subclasses of Petri nets are purely syntactic as they define local constraints on the arrangements of transitions and places. The subclasses are:

1) State Machines (SM)
2) Marked Graphs (MG)
3) Free Choice Petri Nets (FC)
4) Simple Petri Nets (SN)

The restrictions that define these subclasses are given below. The Petri nets without any restrictions will be referred to as general Petri nets to emphasize this fact. The following text should be read together with Figures 2 and 3. Figure 2 shows what kind of local configurations of transition and places are permitted for each subclass of nets.

1. State Machines (SM) -- A state machine is a Petri net in which every transition has exactly one input place and exactly

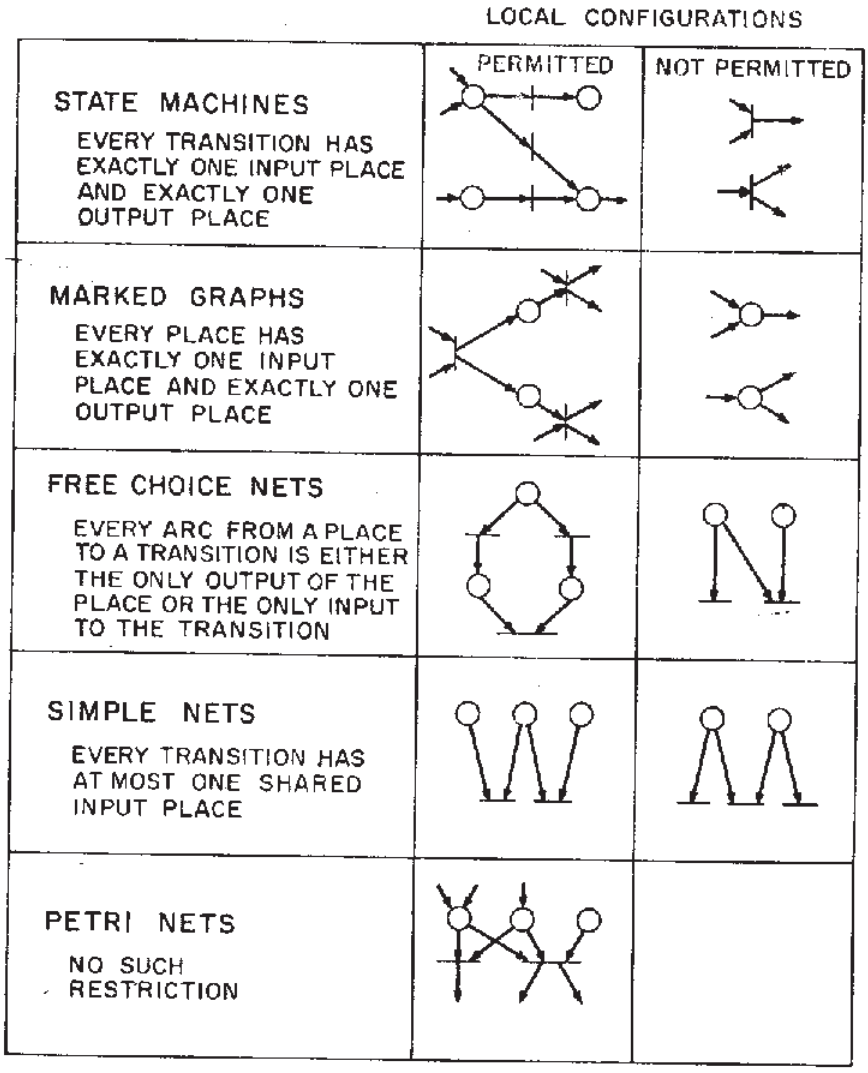


FIG. 2. THE SUBCLASSES OF PETRI NETS.

one output place. The state machines being discussed here are identical to the state machines of automata theory in their structure,(Fig. 4).

2. Marked Graphs (MG) -- A marked graph is a Petri net in which every place has exactly one input transition and exactly one output transition. Thus the restriction in this case is similar to the one for state machines but it applies to places instead of transitions. State machines have been studied extensively but the recognition of marked graphs and the study of their properties is recent. Genrich [3] started the study of marked graphs and his ideas led to a detailed study by Holt and Commoner [4]. The mathematics relating to marked graphs is fairly well understood now through these studies. In our previous report we showed a direct relationship between the elementary asynchronous modular control structures developed by us and the marked graphs. The study provided a simple way for obtaining hardware structures that mimic marked graphs, and also a method for determining if a control structure is free of any hangups. This year the study has been carried further to include a broader class of nets called free choice nets. The free choice nets and results relating to them are described below.

3. Free Choice Nets -- A Petri net in which every arc from a place to a transition is either the only output of the place or the only input to the transition is said to be a free choice Petri net. This condition on Petri nets is the same as requiring that when an input place is shared by some transitions, those transitions have no input places other than the one which is common to them. Thus when a marker arrives in the shared place, all of the transitions which share that place are enabled, and one of them may be freely chosen to fire. When the movement of a marker is regarded as flow of control, the situation just described represents a free choice with regard to where control flows from the shared place -- thus the name free choice nets. Free choice nets include both the state machines and the marked graphs.

A free choice Petri net can be used to represent the flow of control in a program as shown in Fig. 5. In this figure, the shared place x together with transitions T and F represent a decision element -- the if statement in the program. The direction in which control flows from place x is not arbitrary -- it conforms to the outcome of evaluating the predicate associated with the if statement. To the net considered alone the decision about the direction of flow is external to it because it is based on information outside the net; the information flows into the net by way of the interpretation which associates a certain if statement with the free choice transitions in the net. In the study of Petri nets and also in the study of computation schemata, it is important to distinguish what information is a part of the net and what is external to it.

Some important results about free choice nets have been found recently by Commoner of Applied Data Research and Hack of the

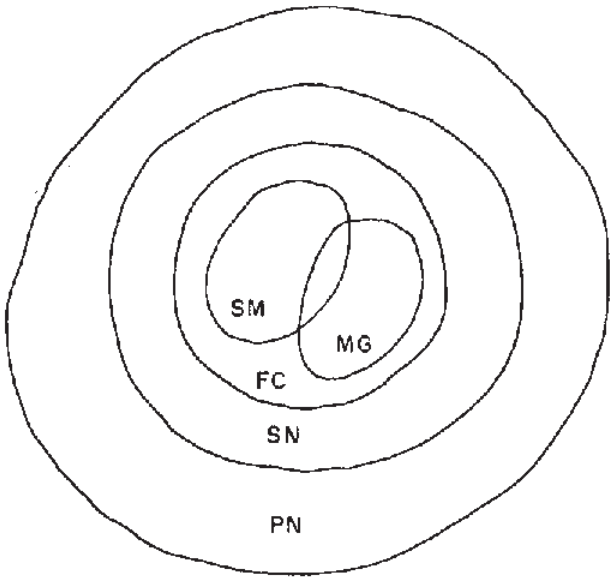
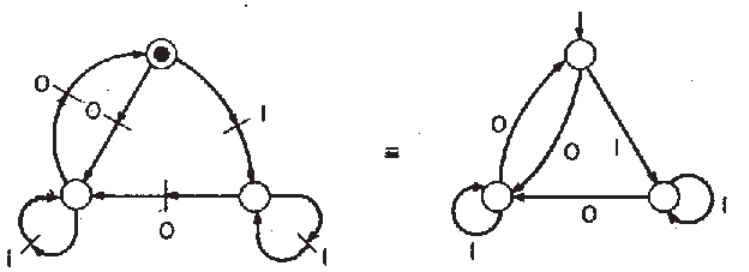


FIG. 3. THE INCLUSION RELATIONSHIP AMONG THE SUBCLASSES OF PETRI NETS.



STATE MACHINE AS A PETRI NET

STATE MACHINE AS A STATE DIAGRAM

FIG. 4. STATE MACHINES.

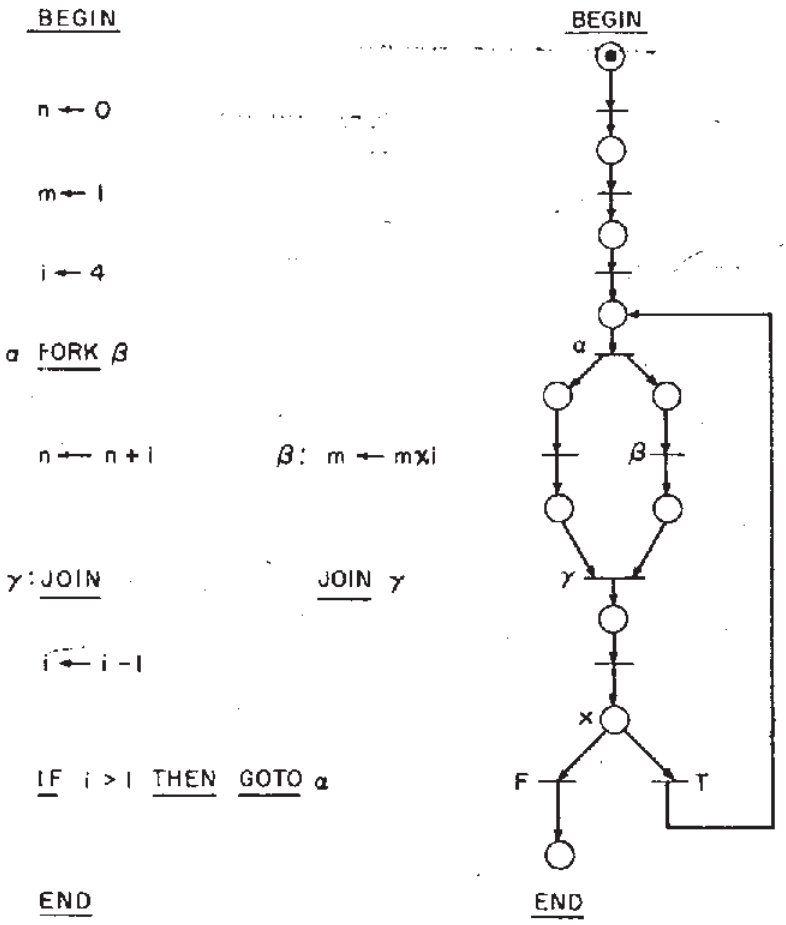


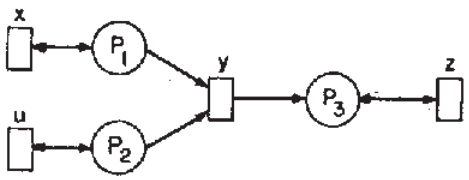
FIG. 5. FLOW OF CONTROL IN A PROGRAM.

Computation Structures Group. Commoner has found necessary and sufficient conditions for liveness and safety of a free choice net, and Hack has found conditions for the existence of a live and safe marking for a net. A live net is one in which the activity can continue indefinitely without any hangup. Hangup is a condition in which a part of the net enters into a state of inactivity from which it cannot recover. In our common experience a hangup for a machine is an unfortunate state in which its activity subsides and it fails to respond to stimulation because of some hopeless jam inside it. Safety on the other hand means that no more than one token will be in any place at any time. This is important where the places represent objects that cannot hold more than one of the things represented by the tokens. When places represent registers in a digital computer, safety means that a new piece of data will not be placed in a register until the previous one has been used up. In that way mixup of data can be avoided. Hack's work thus provides a way to determine if an uninterpreted parallel program which can be expressed as a flow diagram has a starting condition for which it will continue to operate without any hangups or mixups.

4. Simple Petri Nets -- A Petri net in which no more than one input place of any transition is a shared input place is called a simple Petri net; a transition in a simple Petri net may have any number of input places but at most one of those places may be an input place of some other transition. The class of simple Petri nets properly contains the free choice nets. There are situations which can be represented by simple Petri nets but not by free choice nets. Figure 6 shows such a situation which arises in representing flow of control in coordinating processes. An important aspect of simple nets is that they are able to represent interprocess coordination such as implemented by Dijkstra's semaphore primitives. A study of simple Petri nets has led to an understanding of the limitation and capabilities of the semaphore primitives. Details of this study are presented in the next section.

5. General Petri Nets -- The class of Petri nets without any of the restrictions is called general Petri nets. There are many Petri nets in the class of general Petri nets for which there are no equivalent nets in the subclasses defined. In particular, a Petri net which cannot be transformed into a simple net arises in the study discussed below.

Recent work by Patil [5] has shown some interesting facts about the semaphore primitives of Dijkstra [6] by establishing a correspondence between the flow of control in interacting processes and Petri nets. In Fig. 6, three processes coordinate their activities with the help of semaphores. The Petri net for each individual process is obtained by representing each instruction by a transition, connecting these transitions into a chain by means of places to indicate the flow of control in that process, and placing a token in the input place of a transition to indicate the present site of control. The Petri net for a collection of interacting processes is obtained by interconnecting the nets

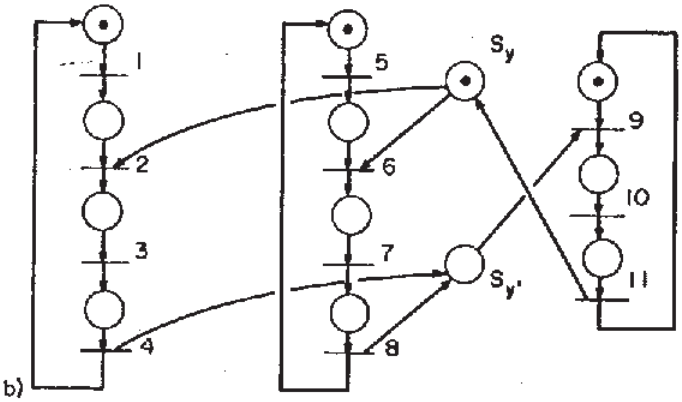


PROCESS

P_1		P_2		P_3	
1	x ← x + x	5	u ← u * u	9	P [$S_{y'}$]
2	P [S_y]	6	P [S_y]	10	z ← z+y
3	y ← x	7	y ← u	11	V [S_y]
4	V [$S_{y'}$]	8	V [$S_{y'}$]		GOTO 9
	GOTO 1		GOTO 5		

INITIALLY SEMAPHORE $S_y = 1$ AND $S_{y'} = 0$

a)



b)

FIG. 6. FLOW OF CONTROL IN PROCESSES AND THE CORRESPONDING SIMPLE PETRI NET.

for individual processes by means of places which represent the semaphores: A transition that represents an instruction P[S] is provided an input from the place that represents semaphore variable S, and each transition that represents an instruction V[S] feeds into the place representing the semaphore. The number of tokens in the place corresponding to a semaphore equals the value of the semaphore variable. Thus, corresponding to the fact that the control in a process can get past a P[S] instruction only when the process can decrement the semaphore variable S by 1, we have the phenomenon in the net that the transition corresponding to the instruction P[S] can fire only when it gets a token from the place representing the semaphore.

The above method of obtaining Petri nets for flow of control applies only to processes which do not have conditional statements. The Petri nets for such processes completely describe the flow of control. Moreover, these nets are simple Petri nets because the only transitions which can have any shared input places are the ones which correspond to the P[ ] instructions and each of these transitions has only one shared input place.

If there are any conditional instructions, they would have to be represented by two transitions, one for the outcome true and the other false, and these transitions would share the input place so that for any particular execution of the conditional instruction, only one of the transitions would fire. Which of the two transitions fires depends on the value of the predicate associated with the conditional instruction. Since this information is external to the net, the net only partially describes the flow of control in this case.

Interacting processes which do not contain any conditional statements are of particular interest to us because it would seem that the semaphore primitives would be adequate for describing their interaction, but our study has uncovered the surprising fact that the semaphore primitives are inadequate for this purpose. This fact is brought out by a study of a problem called the 2-out-of-3 problem which is discussed below.

The 2-out-of-3 problem can be explained in the framework of a message decoder. When viewed as a hardware device, the decoder has three input wires colored red, yellow and green, and three output wires called X, Y and Z. There are three different messages which can be sent to the decoder. Message X consists of signals on the red and yellow wires; message Y consists of signals on the red and green wires; and message Z consists of signals on the yellow and green wires. The decoder can be thought to have three processes inside it, one for each message. Process X waits for message X and responds on output wire X; the other processes are defined similarly. We will be concerned with the above decoder in its software form in which signals are represented through the use of semaphores; each wire is represented by a semaphore and incrementing the semaphore count by 1 corresponds to sending a signal. A

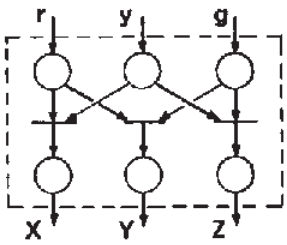


FIG. 7. THE 2-OUT-OF-3 NET.

signal is accepted by decrementing the semaphore count by 1. The question is: Can the three processes which decode the messages be so coordinated by semaphore primitives that the decoder functions correctly? Since each individual process just waits for the associated message to arrive, we insist that the processes not use any conditional instructions. Therefore, instead of asking the question in the form above, we ask: Is there any finite collection of processes not using conditional instructions that can specify the operation of the decoder with the help of the semaphore primitives? The answer to this question is negative.

The reason for the negative answer is that the decoder represents a net called 2-out-of-3 net, which is not a simple Petri net, and it has been possible to show that this net cannot be transformed into an equivalent simple Petri net [5]. Thus it is clear that the semaphore primitives need the help of conditional statements to carry out coordination among processes, (Fig. 7.). It should be recalled that the very purpose of introducing the semaphore primitives was to obtain a more direct means for coordinating processes and to do away with sneaky use of conditional statements to perform coordination. With the aid of conditional statements one can implement coordination of processes by such simple-minded schemes as repeated testing of a variable until it becomes, say, 1. Such schemes can implement coordination, but the implementation is very wasteful of computer resource because there is no limit to the number of times the variable may have to be checked. The semaphore primitives rectify this defect, but they are not able to implement all coordinations by themselves. Thus the question is, whether together with conditional statements they can express all conceivable coordinations without paying the price of unbounded computation. The study has shown that the answer to this question is affirmative.

At the root of the shortcomings of the semaphore primitives is the fact that a P[ ] instruction operates on only one semaphore. Unfortunately, a generalized instruction such as $P[S_1,\ldots,S_k]$, which simultaneously operates on semaphores $S_1, \ldots, S_k$, cannot be always expanded into a sequence of instructions $P[S_1], \ldots, P[S_k]$. But the generalized instruction can be expanded in terms of $P[S_1, S_2]$ instructions each of which operates on two semaphores. Even though $P[S_1, S_2]$ is adequate, one may wish to allow more arguments in instructions for the sake of efficiency.

C. Asynchronous Speed-Independent Circuits

A digital system is often built as two interconnected parts -- a data flow structure containing registers, functional operators and data paths, and a control structure that generates signals that initiate actions by operators in the data flow structure.

In synchronous systems the operators may begin action only at certain time instants determined by a central generator of

clock signals. The design of the control structure involves choosing the appropriate number and duration of clock intervals, and realizing a switching circuit that routes the clock signals to operators as required to implement the system's function.

In an asynchronous control structure, each operator in the data flow structure sends an acknowledge signal to the control structure to indicate that action by the operator has been completed. The acknowledge signals from operators are used directly in the control structure to initiate action by operators that become eligible for execution. In this way, initiation of an operator is delayed only until completion of those actions upon which correct functioning of the operator depends. No special generator of timing signals is used, the timing of system operation being determined by the durations of actions by the operators.

If the control structure of an asynchronous system will function correctly regardless of delays in its components and their interconnecting wires, the control structure is called a speed-independent circuit.

A system described by a logic diagram for a synchronous realization of it is both overspecified and underspecified. The particular choice of clock instants is irrelevant to the function performed by the system, but is essential for the diagram to have any meaning. Yet understandings between the specifier and implementer about timing of actions are necessary for unambiguous interpretation of the description. These understandings are not usually represented in a logic diagram. That a synchronous system is overspecified makes understanding or altering its function difficult; that it is underspecified makes design verification impossible in the absence of oversimplifying assumptions. The description of a system as a speed-indepedent circuit does not suffer these problems. Two parts of a speed-independent circuit are interconnected if, and only if, some action by one part is dependent on completion of some action by the other.

This reasoning shows that speed-independent implementation of digital systems is of particular interest when one desires assurance that a paper design will yield a correctly functioning system when translated into hardware. Speed-independent implementation is also attractive where a system is built from several interacting parts (there are no clocks in the subsystems to be synchronized), or where a system has much concurrent activity (which could only be slowed up by synchronizing action to common clock signals). Computer systems developed in the future are likely to have all of these characteristics.

The group has been studying schemes for representing systems so that conversion of the description into a speed-independent realization may be accomplished by a mechanical procedure with a guarantee that the resulting hardware will function correctly according to the description. In this way, the onerous task of debugging the hardware (as opposed to debugging the system description) would be largely eliminated. In

particular the faults that appear in hardware systems because of misunderstandings about the timing of signals would be avoided.

We are considering two classes of speed-independent circuits based on two assumptions regarding the origin of delays which must not affect correctness of system operation. Both classes of circuits are interconnections of primitive modules which may be individual gates or specific circuits realized in turn by the interconnection of simpler modules or gates.

In a type 1 circuit we assume that all interconnecting wires are sources of arbitrary delays. Thus a signal sent out by one module to two others may reach one module arbitrarily earlier than the other. In a type 2 circuit we assume that the output of a module may be delayed arbitrarily, but when an output of a module changes, the change is observed immediately by all modules to which the output is connected. The type 2 assumption is less restrictive, and is appropriate for circuits in which delays on interconnecting leads are negligible compared to delays within gates. This is normally the case within a semiconductor chip, for example. The more general type 1 assumption is appropriate for interconnections between standard parts where the designer does not know the mechanical arrangement of the parts.

A principal goal of our work is to find a finite set of practical modules with which it is possible to implement any digital system as a type 1 speed-independent circuit. In last year's report we described a collection of control modules adequate to implement any marked graph as a type 1 circuit. The complete set of control modules are also adequate for implementing free choice and simple Petri nets in the form of type 1 speed-independent circuits, and are convenient for defining control structures for complex digital systems.

The C-element of Muller [7] is a very important gate type for the construction of control modules. We have shown that the C-element cannot be implemented as a type 1 interconnection of AND, OR and NOT gates. In fact, there is very little that can be done by a type 1 speed-independent circuit using only AND, OR and NOT gates. These results are included in a paper by Dennis and Patil [8]. Since several basic control modules have type 1 realizations using NOT gates and C-elements, these results emphasize the importance of the C-element as a fundamental gate type for speed-independent circuits. More recently, Fred Furtek has defined a complete set of basic modules for the realization of general Petri nets as type 1 speed-independent circuits.

Our success in applying speed-independent design to control structures for digital systems has led us to investigate the applicability of the concept to complete digital systems. As an experiment, Dennis and Plummer developed a design for a fast counter that could be sampled repeatedly without interfering with continuation of counting. The design is a type 1

interconnection of as many identical stages as desired, each stage being a type 2 circuit using OR-gates, NOT-gates and C-elements. Commands to 'count' or to 'sample' flow through the stages of the counter from the least significant end changing or reading the bit held by each stage. In this way the speed of the counter is independent of the number of stages. The details of the design have been reported [8]. Bill Plummer designed and constructed an arbiter module to resolve conflicts between 'count' and 'sample' commands, and has prepared a paper on his work [9].

D. Base Language

The Group is working toward the definition of a common base language that could serve as a target representation for procedures translated from a variety of practical source languages, for example, FORTRAN, ALGOL and LISP. By specifying a formal interpreter for the base language and giving a precise description of the translation of source programs into base language programs, we would have a complete scheme for the formal definition of the semantics of programming languages in terms of a common set of semantic notions (those of the base language).

The motivation for this work is the design of computer systems in which the creation of correct programs is as convenient and easy as possible. A major factor in the convenient synthesis of programs is the ability to build large programs by combining simpler procedures or program modules, written independently, and perhaps by different individuals using different source languages. This ability of a computer system to support modular programming is called programming generality [10,11]. Programming generality requires the communication of data among independently specified procedures, and thus that the semantics of the languages in which these procedures are expressed must be defined in terms of a common collection of data types and a common concept of data structure.

We have observed that the achievement of programming generality is very difficult in conventional computer systems, primarily because of the variety of data reference and access methods that must be used for the implementation of large programs with acceptable efficiency. For example, data structures that vary in size and form during a computation are given different representations from those that are static; data that reside in different storage media are accessed by different means of reference; clashes of identifiers appearing in different blocks or procedures are prevented by design in some source languages, but similar consideration has not been given to the naming and referencing of cataloged files and procedures in the operating environment of programs. These limitations, on the degree of generality possible in computer systems of conventional architecture have led us to study new concepts of computer system organization through which these limitations on programming generality might be overcome.

In this effort, we are working at the same time on developing

the base language and on developing concepts of computer architecture suited to the execution of computations specified by base language programs. Thus our work on the base language is strongly influenced by hardware concepts derived from the requirements of programming generality [10].

We have chosen trees with shared substructures as our universal representation for information structures because we have found attractive hardware realizations of memory systems for tree-structured data. Jeffery Gertz [12] has considered how such a memory system might be designed as a hierarchy of associative memories. Also, the base language is intended to represent the concurrency of parts of computations in a way that permits their execution in parallel. One reason for emphasizing concurrency is that it is essential to the description of certain computations; for example, when a response is required to whichever one of several independent events is first to occur. Furthermore, we believe that exploiting the potential concurrency in programs will be important in realizing efficient computer systems that offer programming generality. This is because concurrent execution of program parts increases the utilization of processing hardware by providing many activities that can be carried forward while other activities are blocked, pending retrieval of information from slower parts of the computer system memory.

When the meaning of algorithms, expressed in some programming language, has been specified in precise terms, we say that a formal semantics for the language has been given. A formal semantics for a programming language generally takes the form of two sets of rules; one set being a translator, and the second set being an interpreter. The translator specifies a transformation of any well-formed program expressed in the source languge (the concrete language) into an equivalent program expressed in a second language -- the abstract language of the definition. The interpreter expresses the meaning of programs in the abstract language by giving explicit directions for carrying out the computation specified by any well-formed abstract program.

It would be possible to specify the formal semantics of a programming language by giving an interpreter for the concrete programs of the source language; the translator is then the identity transformation. Yet the inclusion of a translator in the definition scheme has important advantages. For one, the phrase structure of a programming language, viewed as a set of strings on some alphabet, usually does not correspond well with the semantic structure of programs. Thus, it is desirable to give the semantic rules of interpretation for a representation of the program that more naturally represents its semantic structure. Furthermore, many constructs present in source languages are provided for convenience rather than as fundamental linguistic features. By arranging the translator to replace occurrences of these constructs with more basic constructs, a simpler abstract language is possible, and its interpreter can be made more readily understandable and, therefore, more useful as a tool for the design and specification of

computer languages and systems.

Our thoughts on the definition of programming languages in terms of a base language are closely related to the formal methods developed at the IBM Vienna Laboratory [13] and which derive from the ideas of McCarthy [14] and Landin [15].

For the formal semantics of programming languages, a general model is required for the data on which programs act. We regard data as consisting of elementary objects, and compound objects formed by combining elementary objects into data structures. Elementary objects are data items whose structure in terms of simpler objects is not relevant to the description of algorithms. For the purposes of this discussion, the class E of elementary objects is

$$E = Z \cup R \cup W$$

where

Z = the class of integers
R = a set of representations for real numbers
W = the set of all strings on some alphabet

Data structures are often represented by directed graphs in which elementary objects are associated with nodes, and each arc is labelled by a member of a set S of selectors. We will use integers and strings as selectors:

$$S = Z \cup W$$

In the class of objects used by the Vienna group, the graphs are restricted to be trees, and elementary objects are associated only with leaf nodes. We have used a less restricted class so an object may have distinct component objects that share some third object as a common component.

> Let E be a class of elementary objects, and let S be a class of selectors. An object is a directed acyclic graph having a single root node from which all other nodes may be reached over directed paths. Each arc is labelled with one selector in S, and an elementary object in E may be associated with each leaf node.

An example of an object is shown in Fig. 8. Leaf nodes having associated elementary objects are represented by circles with the element of E written inside: Integers are represented by numerals, strings are enclosed in single quotes, and reals have decimal points. Other nodes are represented by solid dots, with a horizontal bar if there is more than one emanating arc.

The node of an object reached by traversing an arc emanating from its root node is itself the root node of an object called a component of the original object. The component object consists of all nodes and arcs that can be reached by directed paths from its root node.

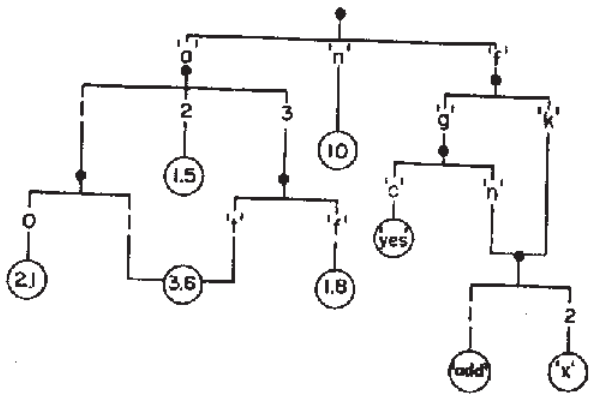


FIG. 8.

Some of us prefer to generalize this class of objects in two ways:

1) by permitting data values to be associated with any node of the graph of a structure

and

2) by permitting the graph to contain directed cycles.

Whether to permit cycles in the structured data objects of the base language is an important unresolved issue. Some considerations bearing on this matter are discussed in a later paragraph of this report.

Figure 9 shows how source languages would be defined in terms of a common base language. Concrete programs in source languages (L1 and L2 in the Figure) are defined by translators into abstract programs of the base language. For this to be effectively possible, the structure of abstract programs cannot reflect the peculiarities of any particular source language, but must provide a set of fundamental linguistic constructs in terms of which the features of these source languages may be realized. The translators themselves should be specified in terms of the base language, probably by means of a specialized source language. Formally, abstract programs in the base language, and states of interpreter are elements of the class of objects defined above.

The structure of states of the interpreter for the base language is shown in Fig. 10. Since we regard the interpreter for the base language as a complete specification for the functional operation of a computer system, a state of the interpreter represents the totality of programs, data, and control information present in the computer system. The universe is an object that represents all information present in the computer system when the system is idle, that is, when no computation is in progress. The universe has data structures and procedure structures as constituent objects. Any object is a legitimate data structure; for example, a data structure may have components that are procedure structures. A procedure structure is an object that represents a procedure expressed in the base language. It has components which are instructions of the base language, data structures, or other procedure structures. So that multiple activations of procedures may be accommodated, a procedure structure remains unaltered during its interpretation.

The local structure of an interpreter state contains a local structure for each current activation of each base language procedure. Each local structure has as components, the local structures of all procedure activations initiated within it. Thus the hierarchy of local structures represents the dynamic relationship of procedure activations.

The control component of an interpreter state is an unordered set of sites of activity. A typical site of activity is

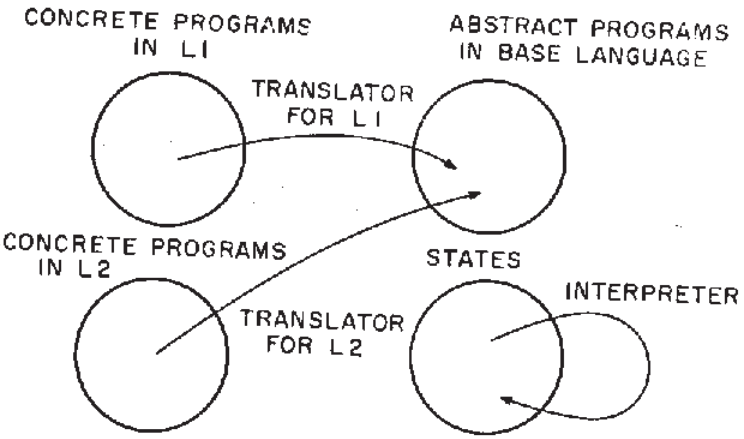

CONCRETE PROGRAMS
IN L1
ABSTRACT PROGRAMS
IN BASE LANGUAGE
TRANSLATOR
FOR L1
CONCRETE PROGRAMS
IN L2
STATES
INTERPRETER
TRANSLATOR
FOR L2

FIG. 9.

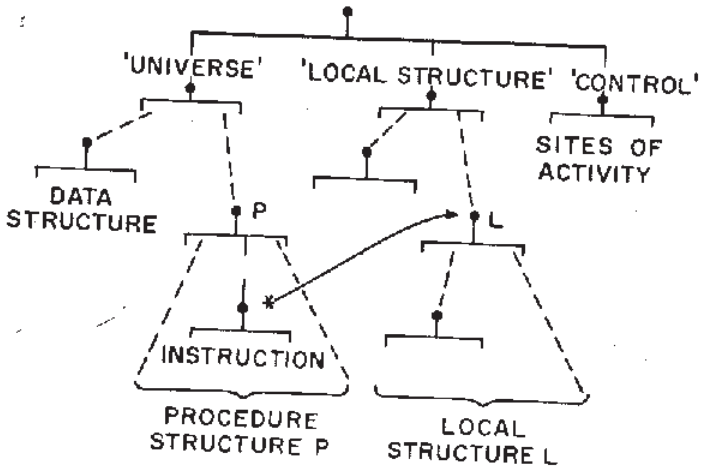

'UNIVERSE'
'LOCAL STRUCTURE'
'CONTROL'
DATA
STRUCTURE
P
L
SITES OF
ACTIVITY
INSTRUCTION
PROCEDURE
STRUCTURE P
LOCAL
STRUCTURE L

FIG. 10.

represented in the figure by an asterisk at an instruction of procedure P and an arrow to the local structure L for some activation of P. Since several activations of a procedure may exist concurrently, there may be two or more sites of activity involving the same instruction of some procedure, but designating different local structures. Also, within one activation of a procedure, several instructions may be active concurrently; thus asterisks on different instructions of a procedure may have arrows to the same local structure.

Each state transition of the interpreter executes one instruction for some procedure activation, at a site of activity selected arbitrarily from the control of the current state. Thus the interpreter is a nondeterministic transition system. In the state resulting from a transition, the chosen site of activity is replaced by zero or more new sites of activity according to the sequencing rules of the base language.

Interpretation of a procedure involves two objects, the procedure structure P and an argument structure A. The argument structure is formed by the calling procedure activation and contains, as component objects, all information (other than P) required by the activation of P. In particular, the actual parameters of the procedure activation are components of A. In this view of procedure execution, no meaning is given to nonlocal references occurring within a procedure structure. Thus no side effects of procedure executions are possible. Unless procedure P modifies part of its own procedure structure, it defines an algebraic operation on the class of all objects.

A subject of major importance to us is the representation of concurrent activities in the base language. Consideration of concurrency brings in the issue of nondeterminacy -- the possibility that computed results will depend on the relative timing with which the concurrent activities are carried forward. The ability of a computer user to direct the system to carry out computations with a guarantee of determinacy is very important. Most programs are intended to implement a functional dependence of results on inputs, and determinism is essential to the verification of their correctness.

There are two ways of providing a guarantee of determinacy to the user of a computer system. They are distinguished according to whether or not the class of base language programs is constrained through design of the interpreter to describe only determinate computations. If this is the case, then any abstract program resulting from compilation will be deterministic in execution. Furthermore, if the compiler is itself a determinate procedure, then each translatable source program represents a determinate procedure. On the other hand, if the design of the interpreter does not guarantee determinacy of abstract programs, determinacy of source programs, when desired, must be ensured by the translator.

E. Program Graphs

We are considering two approaches to represent the relationships

among instructions of a procedure structure:

1. A conventional form in which the instructions of each procedure structure are selected by successive integers, and instructions are executed sequentially except when a conditional transfer of control directs execution to a new instruction sequence.

In this form, concurrency is represented by fork instructions where activity splits into two concurrent streams and join instructions where two streams of activity merge into one.

2. A data flow form in which execution of an instruction is controlled by the availability of the data values required for its execution. For example, execution of an add instruction would be enabled as soon as the values of both operands have been computed.

Concurrency is inherent in a data flow representation since the creation of a computed value may enable several instructions. The data flow representations we are investigating are variations and extensions of the program graphs introduced by Rodriguez [16]. We shall illustrate our present thoughts regarding data flow representations by presenting program graphs for several programs. Consider the program

```
begin
     v := t - x; w := x - u
     if v > w then y := w - 2 else y := v + 3
     if y > 0 then z := y + 2 else z := 0
end
```

A conventional machine level representation would be:

```
begin
     fork ℓ1              ℓ3:   w - 2 → y
     t - x → v            ℓ4:   if y > 0 goto ℓ5
     goto ℓ2                    0 → z
ℓ1:  x - u → w                  goto ℓ6
ℓ2:  join                 ℓ5:   y + 2 → z
     if v > w goto ℓ3      ℓ6:   end
     v + 3 → y
     goto ℓ4
```

A program graph for this program is shown in Fig. 11. The nodes of the program graph include functional operators drawn as circles, predicate operators drawn as diamonds and two special node types, gate and merge, that perform control functions. The links may be thought of as conveying tokens

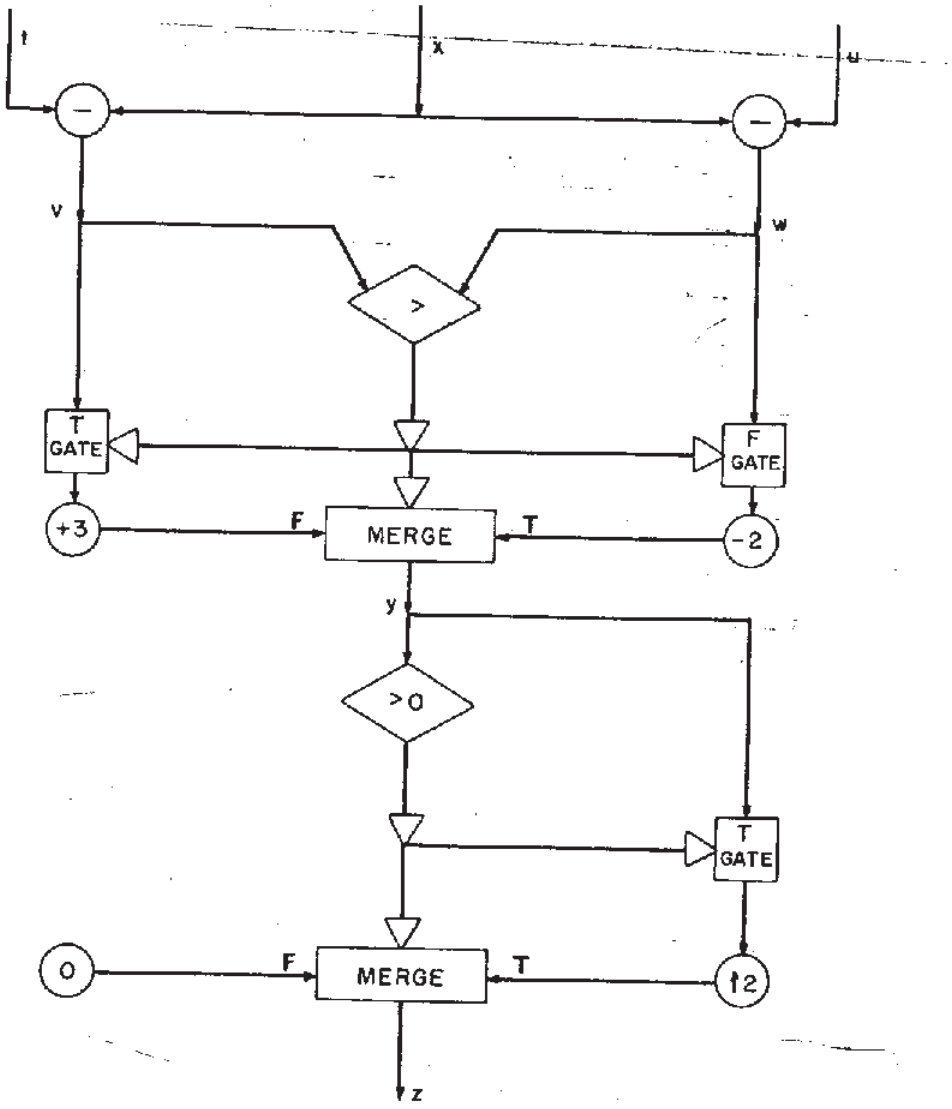


FIG. 11.

between nodes of the diagram as in a Petri net. Here the tokens have information associated with them. Tokens arriving at or leaving functional operators, and those arriving at predicate operators convey values (numbers for example); these links are drawn with small solid arrows. Tokens leaving a predicate operator convey decisions (true or false) to gate nodes of the diagram; these links are drawn with open arrowheads. We assume the net operates in a safe manner, that is, tokens do not overtake one another, nor do they accumulate at nodes. This may be ensured by acknowledge signals transmitted in the reverse direction over each link. Thus a value link may be represented in a Petri net by a pair of places: a place (drawn as a square box) through which tokens with attached values move from source node to destination, and an ordinary place through which "empty" tokens are returned to the source node. Decision links may be conveniently represented by three places through which ordinary tokens (not bearing values) move. A token arriving at the place labeled t signals a true decision; a token arriving at the place labeled f signals a false decision.

When a link goes to two or more destinations, tokens are replicated at each branch point so that tokens with identical information are sent to each node. The branch points act like wye modules, and await acknowledgment signals from each destination before returning an empty token to the source node.

The gate and merge control nodes are needed so that decisions made by predicate operators may affect the pattern of data flow through functional operators of the program graph. A T-gate node permits a value-bearing token to pass through for each true decision received on the decision link. Whenever a false decision arrives the value-bearing token is not forwarded. In either case the gate node acknowledges both tokens received, and when a gate forwards a token, it waits for acknowledgment before forwarding another value-bearing token. The behavior of a gate node is described in Fig. 12. The arrival of a true decision leads to forwarding of a value token from link 1 to link 2. Arrival of a false decision causes a value arriving on link 1 to be acknowledged and discarded. An F-gate node is identical to the T-gate except that the sense of the decision is reversed.

A merge node permits values sent over its output link to originate from different sources according to decisions made during computation. The value sent over the output link is forwarded from the T- or F-labeled input value link according as the decision received is true or false. A Petri net for the switch node is shown in Fig. 13.

Next we give an example showing how iterative programs can be represented as program graphs:

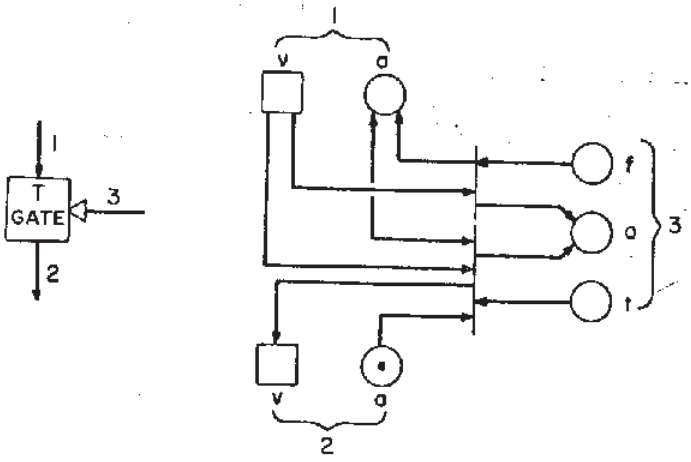


FIG. 12.

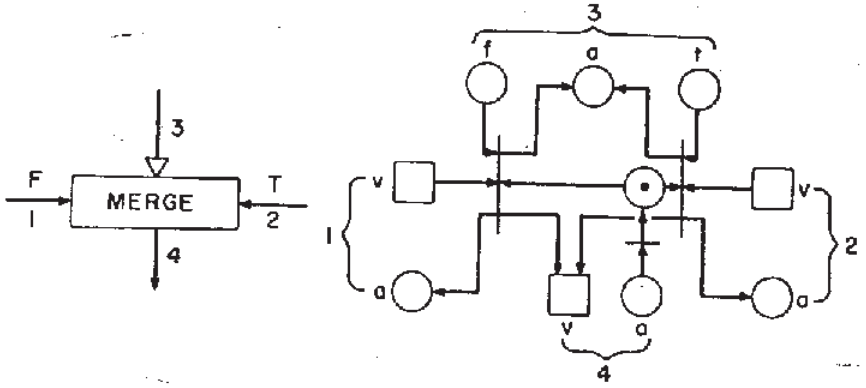


FIG. 13.

```
begin
     y := x
     v := 0
     while p(w,v) do
        begin
            v := f(v); y := g(y)
        end
     z := y
end
```

Noting that the two statements of the body of the iteration may be performed concurrently, a conventional representation would be similar to this:

```
begin
     x → y
     0 → v
ℓ1: if p(w,v)goto ℓ4
    fork ℓ2
    f(v) → v
    goto ℓ3
ℓ2: g(y) → y
ℓ3: join
    goto ℓ1
ℓ4: y → z
    end
```

A data flow version of the program is provided in Fig. 14. Two of the merge nodes serve as the junctions through which initial values and intermediate values flow to the functional operators of the body of the while loop. The predicate operator requires one copy of the value of variable w for each test of the predicate p. These copies are generated by the center merge node, and the associated gate node. Initiation of operation of the program graph requires arrival of a false decision at the decision input link of each of the three merge nodes. This is provided by the F-buff node which is a buffer for decisions that sends a false decision as its initial output, (Fig. 15.).

An important result of Suhas Patil [17] concerning interconnections of determinate systems can be applied to program graphs formed from the node types used in these two examples. We conclude that any such program graph is a determinate representation of a program. This class of program graphs is a revision of the class studied earlier by Rodriguez, and is

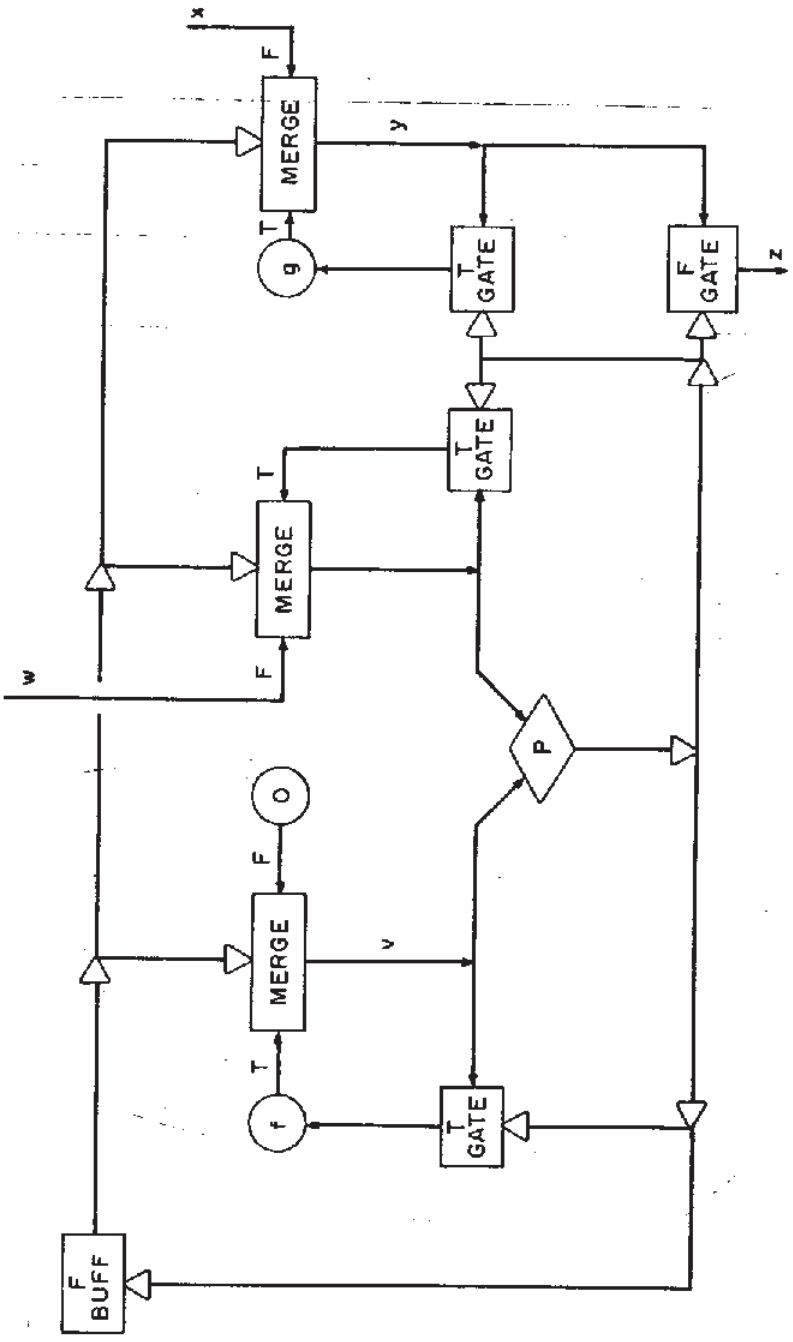

x
F
MERGE
y
T
g
T
GATE
F
GATE
z
T
GATE
T
MERGE
w
F
P
O
F
MERGE
v
T
f
T
GATE
F
BUFF

FIG. 14.

simpler as a result of our improved understanding of concurrent activities. We expect that future developments in the theoretical study of Petri nets will contribute significantly to the building of a satisfactory theory of program graphs.

Jack Dennis has formulated a class of program graphs suitable for representing certain computations on structured data [10]. These program graphs were limited in that no provisions were made for conditional execution of subgraphs or for iterative computation. We expect to combine the concepts developed in this class with those of Rodriquez to obtain a general class of program graphs encompassing,say,all ALGOL 60 programs. Our final example illustrates the form this class of program graphs may take.

```
procedure (a,b,n)
    begin
    y := 0
    for i := 1 step 1 through n do
        y := y + a[i] x b[i]
    return y
    end
```

The input data for this procedure will be represented by the argument structure shown in Fig. 16, having components for the three formal parameters of the procedure. In the program graph shown in Fig. 18, a third kind of link is used and is drawn as a heavy line with a solid arrowhead. Tokens passing on these links convey access to objects. Execution is initiated by arrival of a token at the root node P of the program graph. This token carries access to an argument structure of the form shown. Four new node types are used, (Fig. 17). The select x node converts access to an object into access to the x-component of the object. These nodes are used to obtain access to the components of the argument structure. The second form of select node uses the integer received on link 3 to select the component object. The value node converts access to an elementary object into the value of the object. Finally, the assign node receives a data value on link 2 and transforms the object conveyed on link 1 into an elementary object having that value.

The repeat nodes in this program graph generate multiple copies of tokens conveying access to the same object, in this case the actual parameters of the scalar product procedure. One token is sent over the output link for each true decision received on the decision link. Acknowledgment is not given on the input data link until a false decision is received, whereupon the node resets and waits for the arrival of new data.

This program graph is determinate, yet we cannot guarantee the determinacy of any program graph constructed from all node types introduced here. We would like to find a set of program

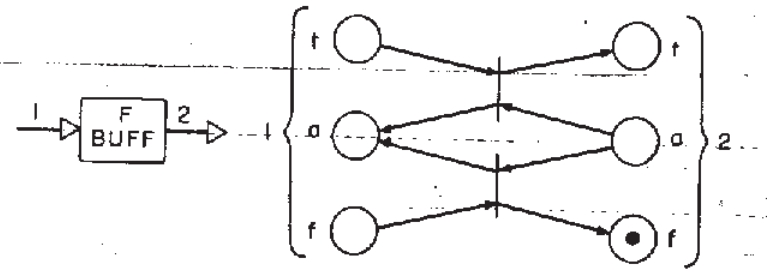


FIG. 15.

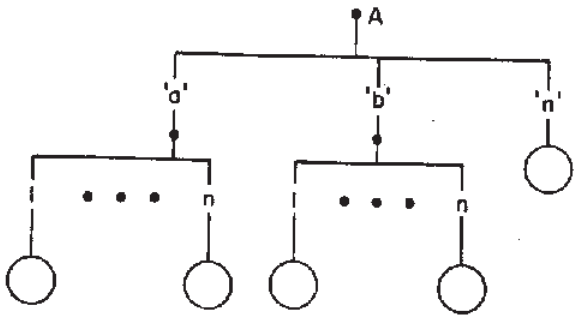


FIG. 16.

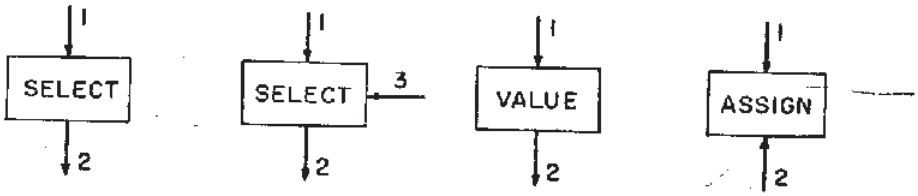


FIG. 17.

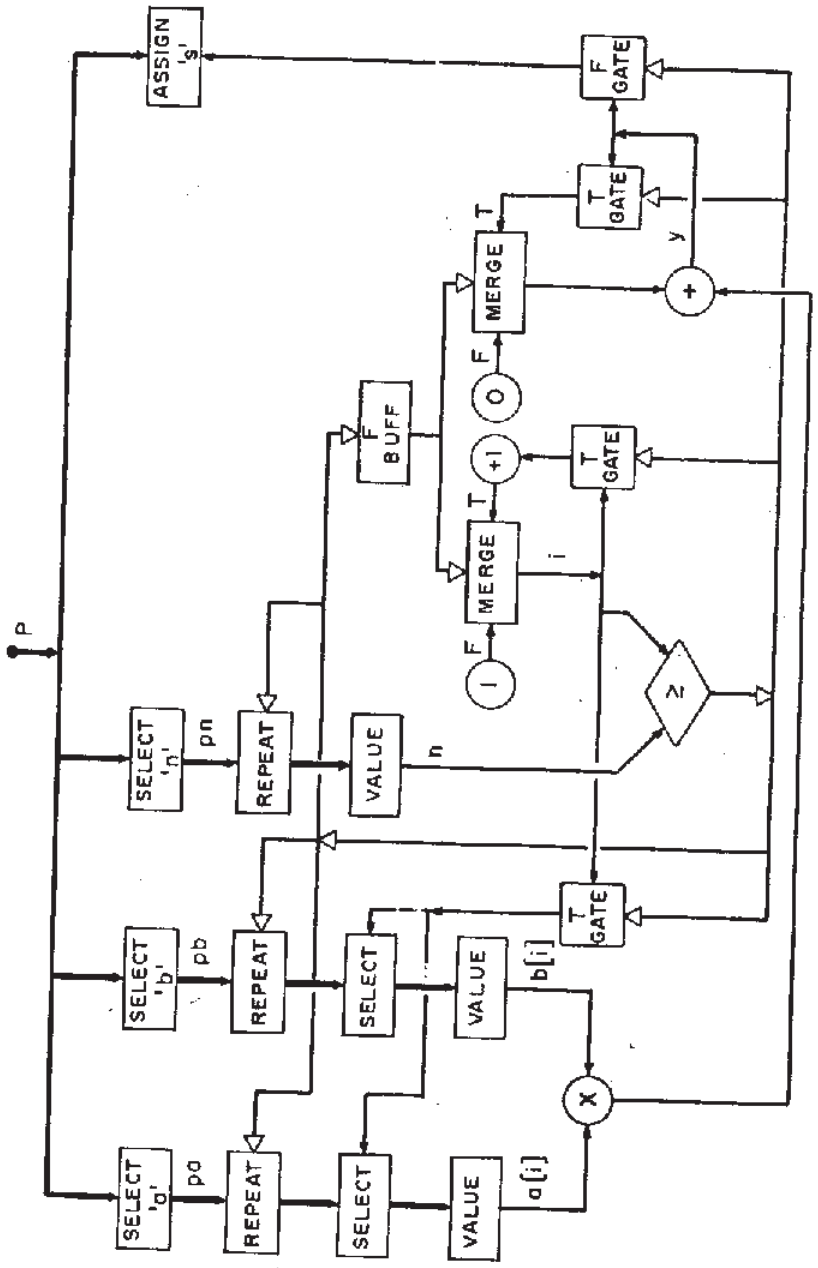

ASSIGN 's'
F GATE
T GATE
y
+
T
MERGE
F
0
F BUFF
+1
T GATE
T
MERGE
i
F
1
≥
P
SELECT 'n'
pn
REPEAT
VALUE
n
T GATE
SELECT 'b'
pb
REPEAT
SELECT
VALUE
b[i]
X
SELECT 'a'
pa
REPEAT
SELECT
VALUE
a[i]

FIG. 18.

graph node types and a condition on their interconnection, such that the program graphs satisfying the condition are determinate and include representations for a wide variety of programs.

Certain computations are more naturally expressed in data flow terms than in conventional form. A typical example is a situation in which several independent activities generate and consume units of data exchanged among themselves. Suppose a computation is performed by two interconnected modules, (Fig. 19.). Module 1 takes an initial value x from data cell a and generates a sequence of values $y_0, y_1, \ldots y_n$ that are forwarded to module 2 through data cell b. Module 2 processes these values as they become available, and, when all values have been processed, puts a cumulative result z in cell c. Let the computations performed by modules 1 and 2 be described by the following relations where f and g denote unspecified functions.

$$y_0 = f(x) \qquad w_0 = 0$$

$$y_1 = f(y_0) \qquad w_1 = g(y_0, w_0)$$

$$\vdots \qquad\qquad \vdots$$

$$y_k = f(y_{k-1}) \qquad w_k = g(y_{k-1}, w_{k-1})$$

$$z = g(y_k, w_k)$$

A program graph for this computation is shown in Fig. 20. The predicate p is applied to each value y_i by both modules to determine when the last value of a sequence has been processed:

$$p(y_i) = \underline{\text{true}}, \quad i = 1, \ldots, k-1$$

$$p(y_k) = \underline{\text{false}}$$

Note that this program graph allows the two modules to act concurrently and is formed simply by connecting together program graphs that represent the two modules. Furthermore, the incorporation of a first in-first out queue in the connecting link would permit module 1 to continue generating values even when module 2 has not had enough time to use up the previous values. The addition of queues does not require any change in the representations of the modules. These properties are not shared by other representations such as co-routines or processes inter-communicating by means of semaphores. Further discussion of these points appears in a recent paper by Jack Dennis [18].

Program graphs are an attractive representation for procedures expressed in the base language because the possibilities for concurrent execution of instructions are exhibited in a natural way. Program graphs represent many procedures in their maximum parallel form. Also, it is easy to impose constraints on

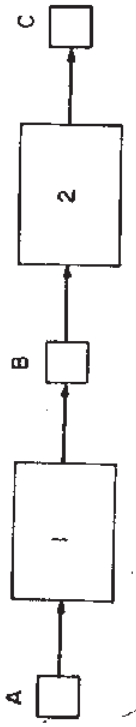


FIG. 19

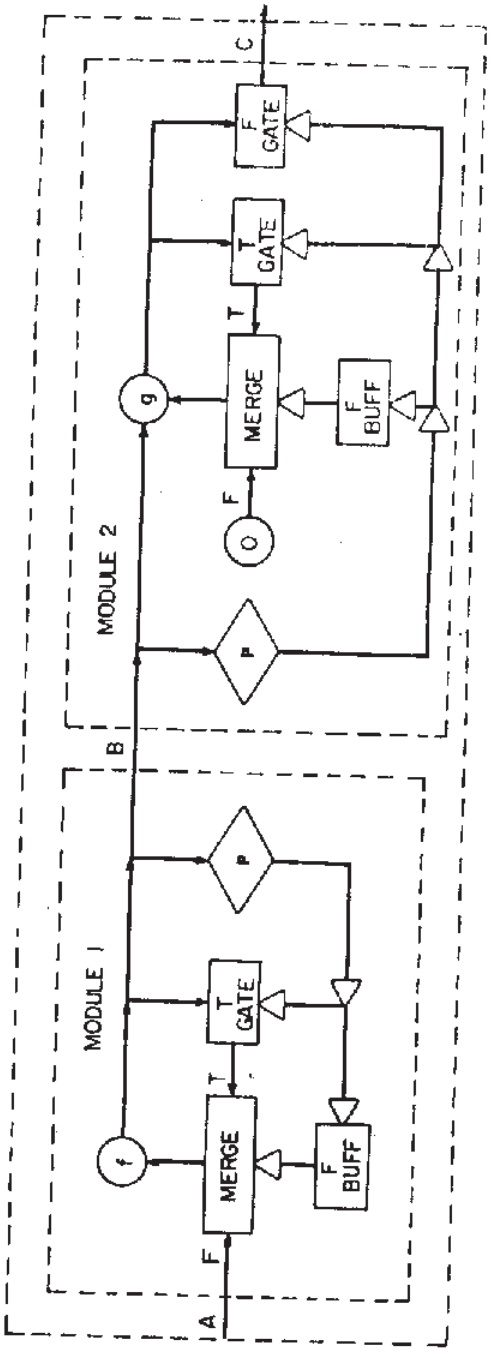


FIG. 20

program graphs such that determinate execution is assured without restricting the class of determinate procedures that can be expressed. Finally, we have found that considering program graphs as a machine level representation leads to interesting concepts for the structure of highly parallel computers [10].

F. Translation of Block-Structured Languages

Many important programming languages for practical computation are block structured; the texts of blocks and procedures are nested, and identifiers appearing in one text may refer to variables declared in other texts. We do not plan to include in the base language provision for directly representing reference by a procedure to external objects. Therefore, we must show how the execution of block-structured programs may be effected through translation into the base language and execution by the base language interpreter. The following discussion outlines one way in which this may be accomplished -- a way that seems attractive in view of the concepts of computer organization we are investigating.

Consider the program shown in Fig. 21. This program has the block structure shown; the main block P encloses a procedure declaration F and a block Q. Upper case letters are used to identify the texts of blocks or procedures.

If T is a text (block or procedure declaration) of a program, let B(T) be the set of identifiers occurring in T that are locally declared. Let X(T) be the set of identifiers occurring in T, or any text nested within T, that refer to variables declared outside T. For the above program we have

$B(P) = \{y, z, f\}$ $B(F) = \{x\}$ $B(Q) = \{y\}$

$X(P) = \emptyset$ $X(F) = \{y\}$ $X(Q) = \{f\}$

Since non-local references are excluded in the base language, we need a scheme for making variables accessed by non-local reference in the block-structured program accessible through the argument structure in the base language representation. We will discuss one method of doing this, details of which are given in a recent paper by Jack Dennis [19]. To illustrate this scheme consider the computation of apply p (4). As objects, the procedure structure P and the local structure L(P) at the beginning of the computation will be as shown in Fig. 22. Texts F and Q are represented as components of the object representing text P. The local structure for the activation of P has one component for each identifier in the set $B(F) \cup X(F)$.

The first step is execution of the declaration of text F. This gives the procedure identifier f a value called a closure of the text F (Fig. 23). The C·T-component of the closure is the text of procedure F and is shared with the procedure structure P. The C·E-component of the closure links identifiers in X(F) to the value these identifiers have in the current procedure activation. Thus the identifier y shares the value 4 with y in L(P).

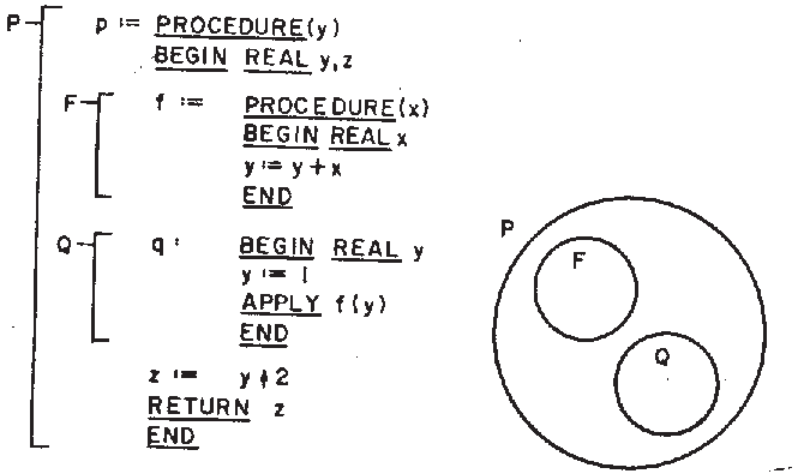

P
p := PROCEDURE(y)
BEGIN REAL y,z
F
f := PROCEDURE(x)
BEGIN REAL x
y := y + x
END
Q
q := BEGIN REAL y
y := 1
APPLY f(y)
END
z := y ↑ 2
RETURN z
END
P
F
Q

FIG. 21.

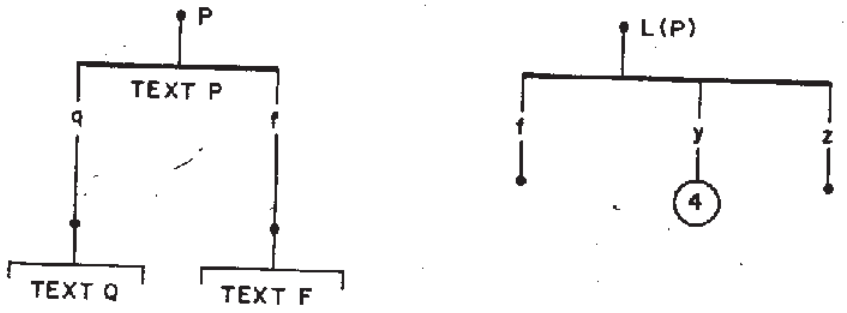

P
TEXT P
q
f
TEXT Q
TEXT F
L(P)
f
y
z
4

FIG. 22.

Entering block Q may be treated as though it were a procedure without parameters. A new local structure L(Q) is formed and made inferior to L(P), (Fig. 24.). This new local structure has a component for each identifier in $B(Q) \cup X(Q) = \{f, y\}$. Identifier f is external, so it is given the same meaning as f in L(P).

After y in L(Q) is assigned the value 1, the closure of F is applied to an argument structure having a 1-component of 1 and an E·y - component that conveys to F the correct meaning of its external identifier, (Fig. 25.).

The meanings of identifiers x and y in text F are established as in the case of Text Q. Since y is in X(F) it is linked to the E·y - component of the argument structure. Since x identifies the first formal parameter of text F, it is linked to the 1-component of the argument structure. In this way, execution of the assignment in text F correctly updates the value of y in the local structure L(P), (Fig. 26.).

G. Cycles in Structures

The class of objects defined earlier does not permit directed cycles to occur in the graph of an object. The desirability of this restriction on the class of objects has been the subject of considerable study and discussion. Arguments against permitting cycles include these:

1. Cyclic structures do not seem essential to the representation of the structured data types of current important source languages.

2. When cycles occur in linked list structures, they can usually be considered part of an implementation rather than an essential aspect of the data structures being represented.

3. The presence of cycles in objects makes it difficult to exploit the concurrency of parts of an algorithm.

The principal arguments in favor of permitting cycles are:

1. Generality of data structures should not be arbitrarily restricted.

2. Cyclic structures are important for representing certain kinds of data.

3. Implementation of a base language using cyclic structures will not present great difficulty.

We have studied two definitive questions to develop better understanding of the importance of cyclic data structures to the base language.

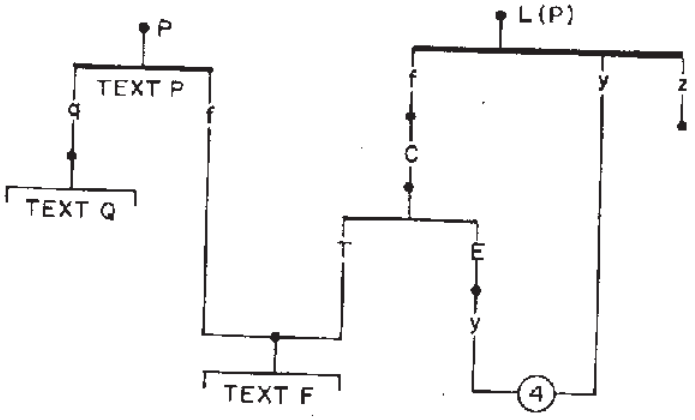


FIG. 23.

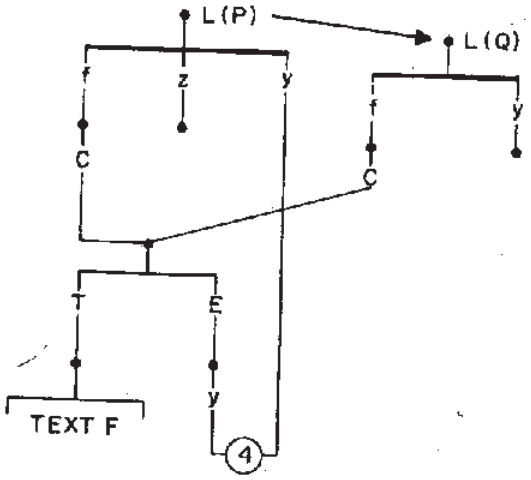


FIG. 24.

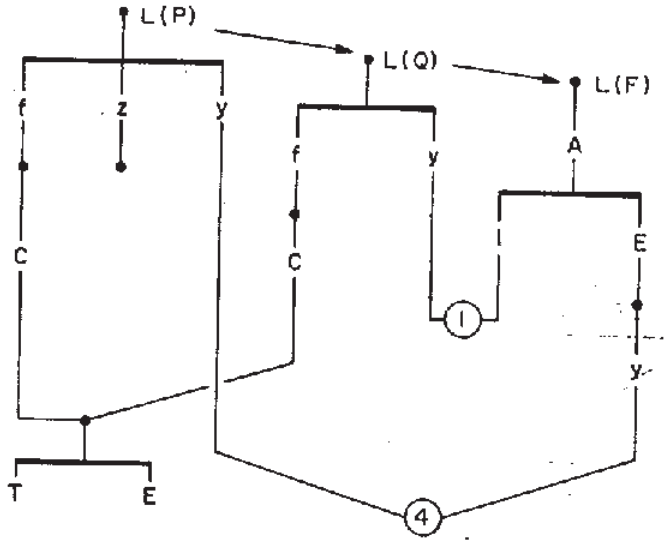


FIG. 25.

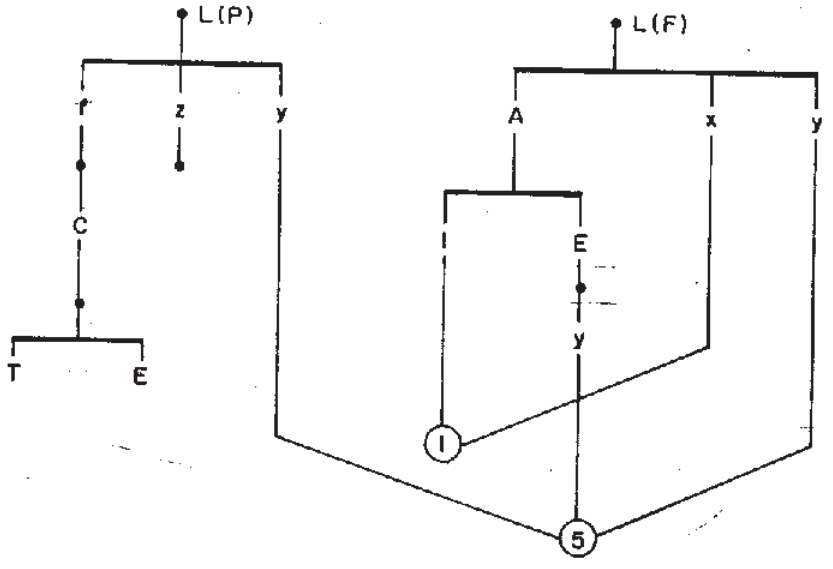


FIG. 26.

```
P  p := PROCEDURE(u)
   BEGIN
      F  f := PROCEDURE(x); INTEGER x
         BEGIN
            IF x = 0 THEN RETURN 1
            x := g(x)
            z := APPLY f(x)
            RETURN z
         END
      APPLY f(u)
   END
```

FIG. 27.

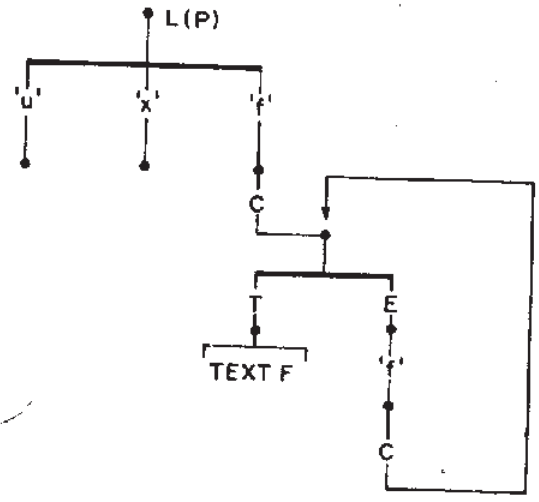


FIG. 28.

One study [19] concerns how cycles can arise during execution of block-structured programs according to the scheme outlined earlier. Consider the program shown in Fig. 27.

This program consists of a procedure declaration F which contains an application of itself. Interpretation of the declaration as described above assigns identifier f a value which is a closure of F, and in which f appears as an external reference. This creates a cycle in the local structure L(P), (Fig. 28).

We have found that many block-structured programs can be rewritten so they accomplish the original computation, but without the creation of cycles. The principle is to convey closures to and from a procedure activation by passing them as parameters or results rather than by external references. For example, the program given above becomes:

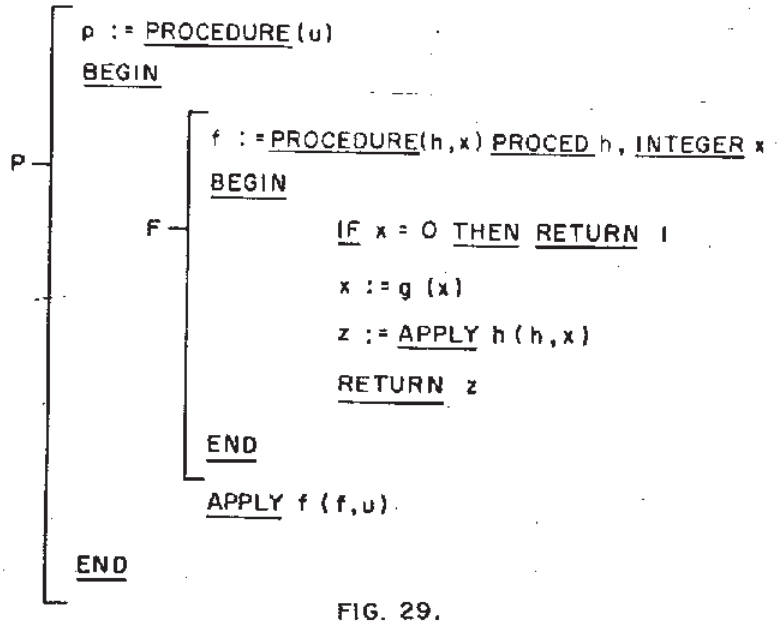


FIG. 29.

This raises some interesting questions. In particular, we would like to develop a general method for rewriting lock-structured programs so that cycles will not arise during execution.

The second study by Ian Campbell-Grant [20] investigated an execution model for multiprocess computations that operate on a data base represented as an arbitrary directed graph. The arcs of the graph represent structural relations among data items associated with the nodes. In this model each process may hold several pointers by which it may access the data base. Each pointer has an associated access control indicator having one of the three values:

R read access
WD write data access
WS write structure access

If a pointer carries R-access to a node, the process may apply the pointer to read (but not alter) the data associated with the node. The process may also obtain a pointer with R-access to any node that can be reached over a directed path in the data base from a node for which it holds R-access. A pointer carrying WD-access to a node permits the process to alter the data associated with the node, and to obtain a pointer with WD-access to any node accessible from the given node. A pointer carrying WS-access to a node permits a process to modify the graph of the data base by adding or deleting arcs within the subgraph formed by all arcs that can be traversed via directed paths starting from the given node. The three kinds of access are cumulative, that is, WD-access includes the privileges of R-access, and WS-access includes the privileges of R-access and WD-access.

The objective of this study was to show how constraints can be implemented in an execution model so that any computation carried on by a set of interacting processes would be determinate. For this purpose, a computation is regarded as determinate if it can never happen that two processes apply pointers to the same data base node concurrently, unless both processes possess only R-access.

The scheme used to ensure determinism involves a set of constraints. Each constraint is an ordered pair (A, B) where A and B are pointers held by distinct processes 1 and 2. The constraint (A, B) signifies that application of pointer B by process 2 must wait until process 1 reduces its access privilege for pointer A.

By executing certain instructions defined for the model, a process may: access nodes by following directed paths in the data base; create and terminate subsidiary processes; and apply pointers to read and write the data associated with accessible nodes of the data base. The execution rules for each instruction type includes specification of how the constraint set must be modified. Campbell-Grant has shown that the relation graph defined by the set of constraints will always be acyclic throughout any multiprocess computation by his model. In consequence, the following condition will always be satisfied, where the predicate struct (X,Y) is true, if and only if, there is a node in the data base reachable over directed paths from the nodes designated by pointers X and Y:

> If pointers A and B are held by distinct processes and struct (A,B) = true then access (A) = R and access (B) = R or one of (A, B) or (B, A) is in the constraint set.

This is sufficient to guarantee determinate computation.

H. Computers and People

When computers are used in any facet of the operation of society, the specific technical characteristics and capabilities of the computer system employed constrain and significantly influence the behavior of the larger system comprising hardware, software and people. We have learned by now that computer hardware should be designed and evaluated in the context of the software that provides the interface with the users. We must now learn how to design and evaluate computer systems in the context of the community of people that is affected by their use.

Two related problem areas require attention. One concerns the interaction between characteristics of computer systems and the individual and collective behavior of the people affected by their use. The other concerns the design of computer systems possessing whatever characteristics are necessary to implement modes of operation that are, at the very least, not objectionable from a human standpoint.

Work in the first area has been in progress during the last three years, although at a low level of intensity. A few papers by Prof. Robert M. Fano and by some of his students are listed below. In addition, Prof. Fano is preparing a short monograph based on the Centennial Lectures he gave during the Spring, 1970, at the Stevens Institute of Technology.

Work in the second area consists mainly of doctoral research by Leo J. Rotenberg. He has developed a model of the protection structures and access-control mechanisms of a multi-access computer system capable of preventing unauthorized releases of information. The model includes spheres of protection constructed out of abilities to reference programs and data segments. Processes can make calls and return from sphere to sphere through inter-sphere links. It can be shown that, under appropriate conditions, calling spheres cannot spy on their callees, nor the callees on their callers. The model includes also facilities for keeping records of critical actions (by system programmers, for instance) and for allocating responsibility for whatever a process does. Such facilities are essential to implement and enforce law regulations and contractual agreements existing in the user community. A brief summary of some of this work is presented in one of the papers listed below ("Surveillance Mechanisms in a Secure Computer Utility").

References

1. A. W. Holt and F. Commoner, Events and Conditions, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp 3-52.

2. C. A. Petri, Communication With Automata, Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1, Griffiss Air Force Base, New York 1966. [Originally published in German: Kommunikation mit Automaten, University of Bonn, 1962.]

3. H. J. Genrich, Simple Nonsequential Processes, Gesellschaft fur Mathematik und Datenverarbeitung, Bonn, 1971.

4. A. W. Holt and F. Commoner, Events and Conditions, Part 2, Applied Data Research, Inc., New York, N. Y.

5. S. S. Patil, Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes, Computation Structures Group Memo 57, Project MAC, M.I.T., Cambridge, Mass., February 1971.

6. E. W. Dijkstra, Co-operating Sequential Processes, Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968.

7. D. E. Muller, Asynchronous Logics and Application to Information Processing, Switching Theory in Space Technology, Stanford University Press, Stanford, California, 1963.

8. J. B. Dennis and S. S. Patil, Speed Independent Asynchronous Circuits, Proceedings of the Fourth Hawaii International Conference on System Sciences, 1971.

9. W. W. Plummer, Asynchronous Arbiters, Computation Structures Group Memo 56, Project MAC, M.I.T., Cambridge, Mass., February 1971.

10. J. B. Dennis, Programming Generality, Parallelism and Computer Architecture, Information Processing 68, North-Holland, Amsterdam 1969, pp 484-492.

11. J. B. Dennis, Future Trends in Time Sharing Systems, Time-Sharing Innovation for Operations Research and Decision-Making, Washington Operations Research Council, 1969, pp 229-235.

12. J. L. Gertz, Hierarchical Associative Memories for Parallel Computation, Report MAC-TR-69, Project MAC, M.I.T., Cambridge, Mass, June 1970.

13. P. Lucas and K. Walk, On the Formal Description of PL/I, Annual Review in Automatic Programming, Vol. 6, Part 3, Pergamon Press 1969, pp 105-182.

14. J. McCarthy, A Formal Description of a Subset of Algol, Formal Language Description Languages for Computer Programming, North-Holland, Amsterdam, 1966, pp 1-12.

References (cont.)

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

16. J. E. Rodriguez, A Graph Model for Parallel Computations, Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.

17. S. S. Patil, Closure Properties of Interconnections of Determinate System, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970, pp. 107-116.

18. J. B. Dennis, Coroutines and Parallel Computation, Princeton Conference on Information Sciences and Systems, Princeton, N.J., March 1971.

19. J. B. Dennis, On the Design and Specification of a Common Base Language, Proceedings of a Symposium on Computers and Automata, Polytechnic Institute of Brooklyn. To be published.

20. I. Campbell-Grant, "The Controlled Execution of Parallel Programs Operating on Structured Data", S.M. Thesis, Dept. of Electrical Engineering, January 1971.

Publications 1970-1971

Campbell-Grant, I., "The Controlled Execution of Parallel Programs Operating on Structured Data", S.M. Thesis, Dept. of Electrical Engineering, January 1971.

Dennis, J. B., Coroutines and Parallel Computation, Princeton Conference on Information Sciences and Systems, Princeton, N. J., March 1971.

Dennis, J.B., On the Design and Specification of a Common Base Language, Proceedings of a Symposium on Computers and Automata, Polytechnic Institute of Brooklyn. To be published.

Dennis, J. B., and Patil, S. S., Speed Independent Asynchronous Circuits, Proceedings of the Fourth Hawaii International Conference on System Sciences, 1971.

Fano, R. M., "Computers in Human Society -- For Good or Ill?", Technology Review, March 1970, pp. 25-31.

Publications (cont.)

Fano, R. M., "Computers in Society", to be published in the Proceedings of the Symposium "L'Informatica, La Cultura e La Societa Italiana", held at the Fondazione Giovanni Agnelli, Torino, Italy, December 9-11, 1970.

Patil, S. S., Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes, Computation Structures Group Memo 57, Project MAC, M.I.T., Cambridge, Mass., February 1971.

Plummer, W.W., Asynchronous Arbiters, Computation Structures Group Memo 56, Project MAC, M.I.T., Cambridge, Mass., February 1971.

Rotenberg, Leo J., "Surveillance Mechanisms in a Secure Computer Utility", Computers and Society, Vol. 2, No. 1, April 1971, ACM Special Interest Group on Computers and Society.

Vogt, Carla, "Making Computerized Knowledge Safe for People", Technology Review, March 1970, pp. 33-39.