MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group Memo 53-1

Computation Structures Group
Progress Report 1969-70

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.

I. INTRODUCTION

Research in the Computation Structures Group has the objective of advancing knowledge and understanding of computer system organization through abstraction and analysis. Our activities have led us to some interesting ideas regarding appropriate directions for the evolution of general-purpose computer hardware. Much of our current activity explores the implications of these ideas concerning computer system organization. Areas under study include: the theory and practice of asynchronous systems; concurrency in computation -- its influence on computer structure and on the representation of algorithms; the concept of "programming generality" -- the property of a computer system that would permit unrestricted combination of independently written programs; the controlled access to programs and data bases; and an approach to formal semantics for programs based on an abstract model for information structures.

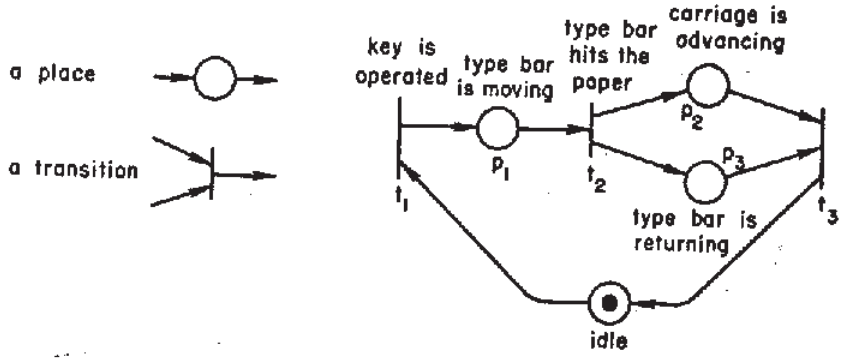
The past year has seen major advances in our understanding of modular asynchronous systems and the intimate relation of modular control structures to the Petri nets studied by Anatol Holt. We have found our knowledge of asynchronous systems sufficient to yield elegant and readily understood implementations of the control mechanisms of complex central processors. We have analyzed aspects of the concept of a hierarchical associative memory. Our understanding of the properties of uninterpreted schemes of programs has been improved through study of graphs that explicitly show data dependence. Finally, we have studied formal models of two aspects of advanced operating systems -- the controlled sharing of information, and the avoidance of deadlocks arising from resource sharing.

II. MODULAR ASYNCHRONOUS SYSTEMS

By "system" we mean an arrangement of parts that interact with one another by means of discrete signals. The essence of systems is activity: The parts of a system act at instants in consequence of earlier actions by other parts of the system. Most systems have many parts that act without immediate intercommunication. Such independent parts that may act simultaneously are said to have concurrent activity. Man-machine interaction involves concurrent activity of the man and the computer; a digital system operates through the concurrent activity of its individual circuits. The importance of concurrency goes far beyond the use of parallel actions to attain greater speed. A large system is usually constructed through interconnection of simpler systems which often operate without central control. The component systems must interact to make their presence felt and this interaction is inherently a concurrent activity. We shall review some aspects of our current

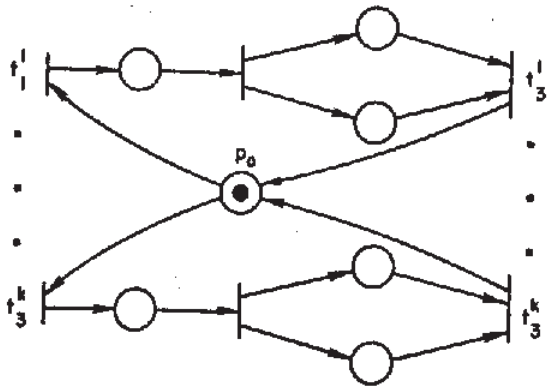
work on the representation of concurrent activity and the implementation of systems in the form of asynchronous modular hardware structures.

Consider what happens when a typewriter key is pressed. The type bar is initially idle. When the key is operated, the type bar starts moving toward the carriage; when it hits the paper, it starts to retreat and at the same time the carriage starts to advance. The key can be operated again only after the mechanism has returned to the idle condition, that is, the motion of the carriage has stopped and the type bar has returned to rest. This activity may be represented by a diagram called a Petri net:



The circles are called places and the solid bars are called transitions. The places are associated with conditions and the transitions with events. The condition associated with a place is said to hold when there is a token (sometimes called a marker or stone) at the place. A transition is enabled if all its input places have tokens. An enabled transition may fire, taking one token from each of its input places and putting a token on each of its output places. In terms of events and conditions, one can thus say an event occurs only when all conditions required for its occurrence hold, and the occurrence of an event causes the old conditions to cease and new conditions to hold. The activity continues as transitions fire, changing the distribution of tokens in the net and enabling other transitions.

In Petri nets, a place may be an input place of more than one transition. The situation where two transitions are active, but have one input place in common, is called a conflict because the transitions are in conflict over the token at the shared place. Only one of the transitions may fire even though both are active. Such a situation arises in a typewriter which does not permit more than one key to be operated at the same time. The Petri net below illustrates this situation.



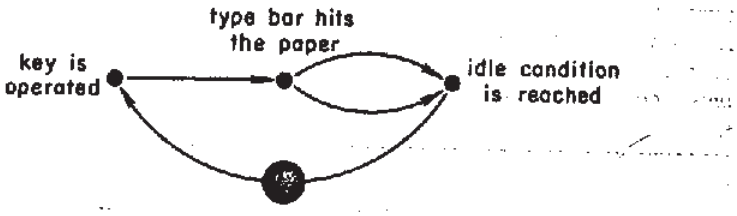
In this figure, transitions $t_1, \ldots, t_k$ are in conflict over the place p_0, and the conflict at this place prevents two or more keys from being operated concurrently.

Petri nets are a scheme for representing concurrent systems adopted by Anatol Holt of Applied Data Research [1] from the nets originally proposed by Carl Adam Petri of the University of Bonn [2]. In the Computation Structures Group, Suhas Patil has developed a generalization of Petri nets that simplifies the representation of interactions associated with resource sharing [3]; Jack Dennis has investigated the use of Petri nets to represent the control structures of a highly parallel computer processing unit [4]; and we have studied the implementation of nets in the form of asynchronous modular structures. A few aspects of these investigations are discussed briefly in the following paragraphs.

Marked graphs constitute a subclass of Petri nets in which each place is an input place of exactly one transition and an output place of exactly one transition. The net describing the operation of one key of a typewriter is a marked g aph. Marked graphs have many important properties, and there is a direct correspondence between marked graphs and elementary control structures for digital systems built by the interconnection of a set of primitive asynchronous control modules to be introduced shortly. This correspondence is useful in two ways: A computer control unit specified as a marked graph can be translated into an asynchronous control structure by a clerical procedure; and a control structure may be converted into a marked graph to facilitate analysis.

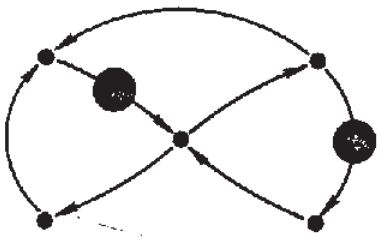
Since a place in a marked graph has only one incident arc and only one emergent arc, the circles representing the places are usually omitted -- an arc from one transition to another is understood to have

a place on it. Further conciseness is obtained by drawing the transitions as solid dots. In this simplified form, the marked graph describing the operation of a typewriter becomes:



In the new notation, the presence of tokens is indicated by placing markers on the arcs -- hence the name "marked graphs".

An important question about a marked graph is whether its activity continues forever or comes to a halt. The property of representing activity that goes on indefinitely is called liveness. A net is said to be live for some initial marking if, after any arbitrary activity has passed, a continuation of activity is possible that will fire any chosen transition. In other words, in a live net no transition is ever crossed off the list of transitions that may be called upon to fire. In general, it is difficult to determine whether an arbitrary Petri net is live. Yet marked graphs have the nice property that a marked graph is live if and only if cutting the marked edges of the graph leaves an acyclic graph. The marked graph shown below is live.

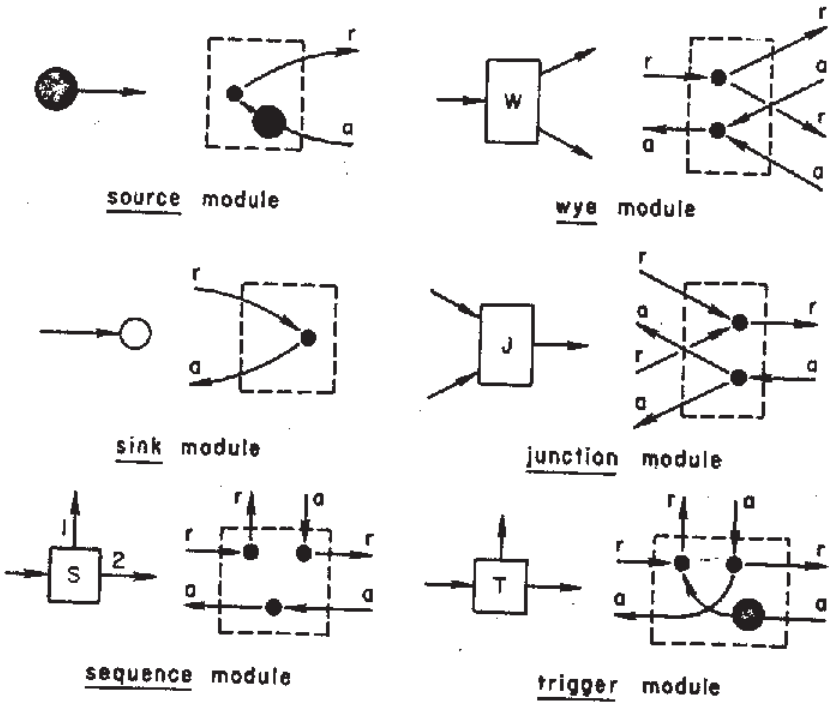


a live marked graph

The reader can check that, if any of the markers are removed, the activity of the graph will come to a halt. This property of marked graphs is very useful in determining whether an elementary control structure is free of hang-ups.

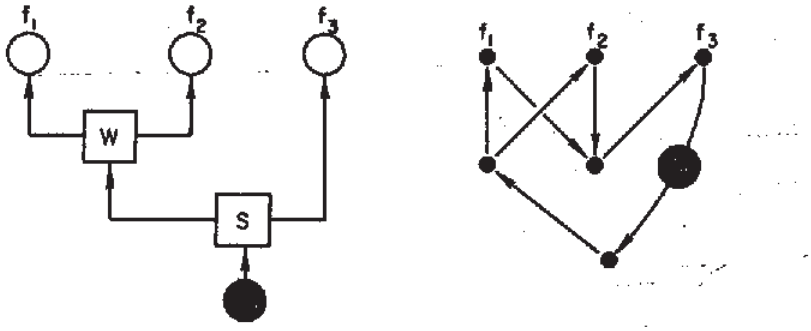
An elementary control structure is a digital system consisting of models of six types interconnected by directed links. Each link is able to transmit ready signals in the forward direction and acknowledge signals in the reverse direction. By associating two arcs with each

link of a module, the behavior of each module type may be specified by a marked graph fragment as follows:



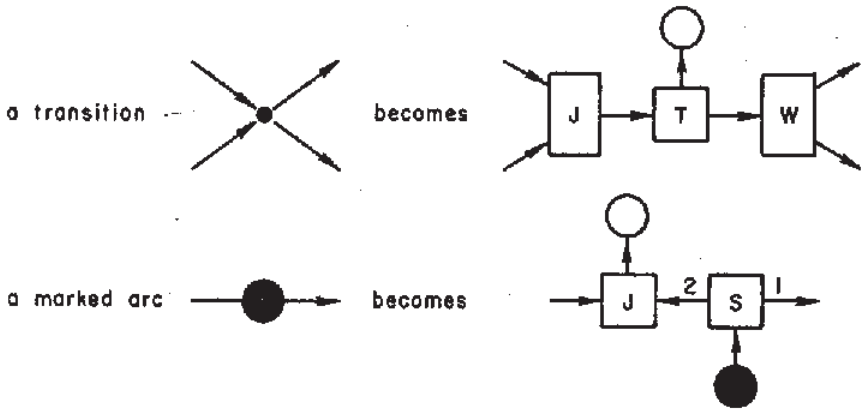
The arrival of a token on an arc in the marked graph corresponds to the transmission of a ready or acknowledge signal between two modules. A wye module, for example, sends a ready signal over the two emergent links when a ready signal is received on the incident link. Then, when acknowledge signals have been returned, an acknowledge signal is returned over the incident link.

Thus a wye module controls the concurrent execution of two independent operations. The sequence module controls the sequential execution of two operations. The junction module permits an action to take place only when the conjunction of two conditions becomes true. The control structure shown on the next page causes concurrent execution of activities f_1 and f_2, and causes activity f_3 to occur only when f_1 and f_2 have completed. The operators f_1, f_2, and f_3 are represented by sink modules, and a source module is included so that the control structure will have unceasing activity. The corresponding marked graph was found by substituting for control modules the marked graph fragments



given above, and simplifying the resulting graph by omitting certain redundant nodes. Since the marked graph is live, we can conclude that the control structure from which it was derived will not hang up.

It is also straightforward to obtain an elementary control structure that implements an arbitrary marked graph by making the following substitutions:



The resulting control structure is guaranteed to be hang-up free if the given marked graph is live.

Work is continuing on the problem of obtaining control structures for more general subclasses of Petri nets. We know, from the work of Suhas Patil [3], a systematic way of implementing any Petri net by an interconnection of asynchronous modules. However, this scheme seems unnecessarily complex, and we are studying what sets of simple primitive modules are sufficient to implement several intermediate classes of nets.

III. DESCRIPTION OF A HIGH PERFORMANCE PROCESSOR

We have looked into the suitability of Petri nets and asynchronous control structures for representing and implementing the control mechanisms of a high-performance processor. For this exercise, we chose a machine similar in principle to the Control Data 6600 but simpler in detail. The machine has several functional units that can perform different operations concurrently. The processor is so organized that instructions may be executed in a sequence different from their order of appearance in the instruction stream. A mechanism known as the scoreboard controls access of the functional units to values held in data registers so that each unit operates only when its operands are available.

Synchronous logic design techniques were used for the 6600. Thus it appeared to be an interesting challenge to see whether the control mechanisms of such a machine could be conveniently implemented by using the asynchronous modular techniques developed by the Computation Structures Group.

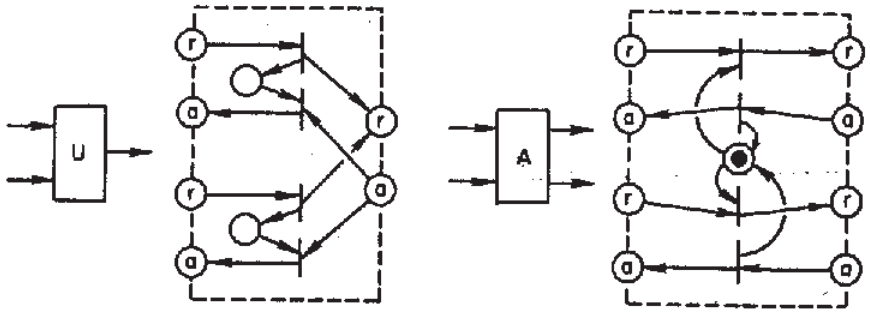
We divided the control problem into these parts: the instruction queue, the instruction allocator, the scoreboard, and control circuits for the functional units. Each was represented by a Petri net, and a control structure was devised to have exactly the behavior represented by each Petri net. It turned out that nine types of control modules were sufficient to give reasonable implementations of all six control structures:

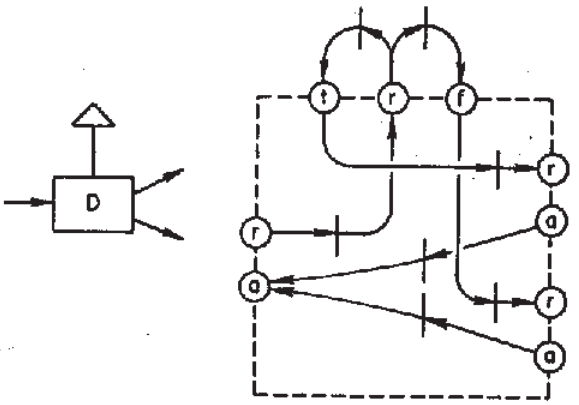
source	wye	decision
sink	junction	union
sequence	trigger	arbiter

The first six of these modules were specified earlier in terms of marked graphs. The three remaining modules are defined by the fragments of Petri nets shown on the following page.

The union module permits control of an activity from either of two points within a control structure. The arbiter interlocks two activities so that only one of them may be in progress at a time. The decider module makes it possible for the control structure to effect different activities, depending on information residing outside the control structure -- for instance, the operation code of an instruction.

The design of the scoreboard turned out to be particularly elegant, and it seems clearly preferable to a synchronous design in regard to complexity and speed. Details are given in a recent paper by Jack Dennis [4].





IV. DETERMINACY OF SYSTEMS

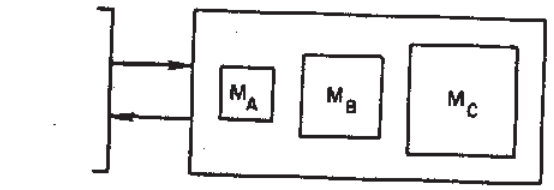
To keep the design complexity of a large system within manageable limits, the system is generally conceived as a combination of simpler systems. Unfortunately, even if the subsystems are known to work correctly, one cannot conclude that the interconnection of the subsystems to form the complete system will operate as intended. Therefore, it is important to obtain a better understanding of the problems which arise when systems are interconnected. In this direction we have achieved some important results concerning interconnections of determinate systems -- systems whose input-output relations are functions. A computer system which gives the same results for two runs of a given program for given data is a determinate system, a system that does not is not determinate. In constructing a large system from simpler determinate systems one would like to know how to ensure that the interconnection will result in a determinate system. Suhas Patil [5] has shown that, if the intercommunication discipline is chosen properly, any interconnection of a number of determinate systems may be

guaranteed to be determinate. This work provides a theoretical basis for elementary control structures: The elementary control structures form a class which is closed under interconnection. Moreover, since each of the elementary control modules discussed earlier is determinate, each member of the class of elementary control structures is guaranteed to be a determinate system. Correspondingly, the marked graphs form a class of determinate systems.

This work on the interconnection of systems may have significant application to networks of computers in which one would like to ensure correctness of a computation even though parts of it are carried out at different installations.

V. HIERARCHICAL ASSOCIATIVE MEMORY

The use of location-independent addressing is essential in a computer system that offers programming generality. In contemporary computer systems, where the memory consists of several physical storage media (solid-state, magnetic-core, drum, etc.), combinations of software and hardware mechanisms (paging, for example) have generally been used to realize location-independent addressing. Nevertheless, it is recognized that these implementations suffer from gross inefficiencies in the form of wasted processor time and poorly utilized memory space. In 1968, we outlined a radical concept of computer organization, and proposed the concept of a hierarchical associative memory [6].



In such a memory system each level is arranged as an associative memory with value fields of n bits and key fields of p bits; M_A is small and fast, M_C is slow by comparison but large. Reference to an item is made by presenting its name to the memory system. A match is first sought in M_A; if successful, the required item has been located and is read out or altered. If the search in M_A is unsuccessful, the key is used for a search of M_B, and then a search of M_C. When an item is found, it is moved to the highest level M_A, possibly together with other items known likely to be required in conjunction with it. In each level, we suggested that the age of items since their last instance of use be used to determine which items should be moved down in the hierarchy to maintain a suitable number of vacant locations for newly referenced items.

As in conventional memory systems, an organization is desired that permits a large throughput (average number of references completed per unit time). In contemporary high-performance systems, high throughput is achieved by building the memory in several modules each of which can be performing memory accesses concurrently with the others. In a location-addressed memory, this scheme works well because each name (address) always designates the same location in the same module, and action by more than one module is never required to complete a reference.

The construction of a modular associative memory poses some new problems. Since an item may occupy different locations in the memory at different times, one does not know in general which module will contain an item when access to it is required. Unless some provision is made for organized assignment of items to modules, an access request to a modular associative memory must be presented to each of the modules either in sequence or concurrently. If this is done sequentially, an average of half the modules will have to be interrogated before the item is found. If the item is not present in this level of the memory hierarchy, all modules must be interrogated before this fact is known. If all modules are interrogated concurrently, each one will be activated whether or not the item is present in the level, but the average time required to complete an access may be less. In either scheme, the speed advantage of using a modular memory is lost.

Jeffrey Gertz has investigated two alternate schemes for avoiding the necessity of searching all modules [7]. Both schemes assign each item to a specific module according to some readily tested property of the item:

(1) By ownership -- all items belonging to the same computation are assigned to the same module.

(2) By transformation -- a transformation of the key (a hash code) determines the module to which an item is assigned.

If items are assigned to memory modules by ownership, a search of more than one module is required only when reference is made to the information owned by another computation. If the key includes unique identification of an item's owner, only one module need be searched. If the key does not indicate ownership, the module containing owned information can be interrogated first -- on the assumption that items referenced are more likely to be owned items than shared items. The use of this scheme implies there are many more active computations than modules because it is unreasonable to expect one module to exactly fit the memory requirement of any one computation.

The assignment of items to modules according to a hash code of their keys is attractive where one expects most information to be shared among active computations. Only one interrogation is required to locate an item or to find that it must be retrieved from a lower level.

However, if an item may be referenced by two distinct keys, either the item would have to be duplicated in two modules, or all modules would have to be interrogated to effect reference by one of the two keys.

VI. BASE LANGUAGE RESEARCH

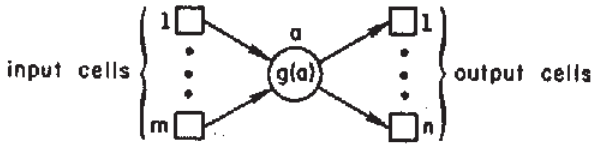
During the past year, work has begun toward the definition of a base program language. We think of the base language as a representation scheme for programs intermediate between source programming languages such as Algol and Snobol, and a machine-level representation. In its design, we hope to achieve three goals: to create a general-purpose language that is entirely consistent with the requirements of programming generality; to find a representation that expresses all possibilities for concurrent execution of parts of algorithms; and to obtain a language that can be used as a functional specification for an advanced highly parallel computer design.

We have made major gains in our understanding of the properties of certain mathematical models of the structure of programs; we call these models computation schemata. Our theoretical work with computation schemata has so far been restricted to computations that operate on simple variables -- variables whose structure as a collection of simpler entities is not relevant to the scheme (the flowchart) of the computation. Yet it is important to thoroughly understand this subject as a basis for building a more general theory for programs that operate on structured data.

VII. COMPUTATION SCHEMATA

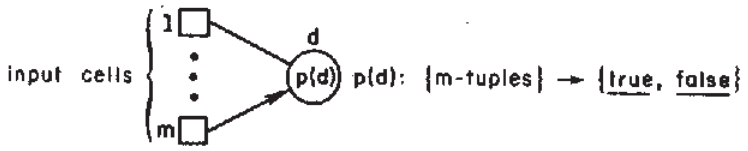
Our work on computation schemata has evolved from the thesis research done by Van Horn [8], Rodriguez [9], Luconi [10], and Slutz [11] at Project MAC, and has been considerably influenced by the original studies of Yanov [12] and, more recently, the work of Karp and Miller [13], and the work of Paterson [14]. Two questions have been of greatest interest to us: What sort of constraints must be met in the representation of parallel computations so that unique results of computations may be guaranteed? Under what conditions is it possible to determine whether two representations (schemata) describe identical classes of computations? For the class of schemata we have considered, we now have satisfactory answers to the first question, and have gained a better understanding of the second.

A computation schema represents the manner in which functional elements and decision elements are interconnected, and their action sequenced, to define an algorithm. The functional elements of a schema are called operators: Each operator a evaluates some unspecified function of an m-tuple of input variables and assigns values to an n-tuple of output variables.



g(a): {m-tuples} → {n-tuples}

The unspecified function associated with an operator a is denoted by g(a). The decision elements of a schema are called deciders: Each decider d tests some unspecified predicate p(d) for an m-tuple of input variables.

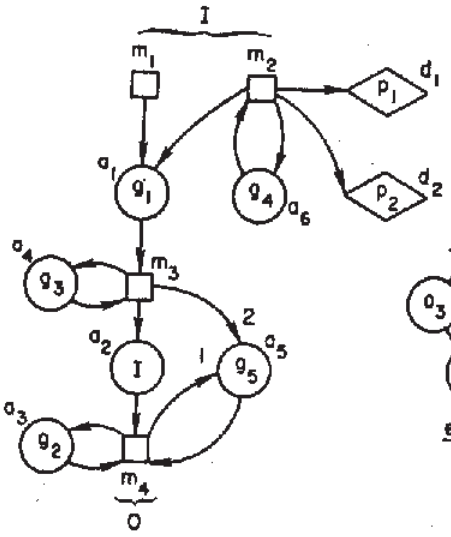


A computation schema has two parts -- a data flow graph and a control. The data flow graph defines the interconnections through which results of each operator application are passed on as arguments for further transformations and tests. The variables of a schema are represented in the data flow graph by boxes called cells. There is also a circle for each operator and a diamond for each decider. Directed arcs join the operators to their output cells and represent the connections to each operator and decider from its input cells.

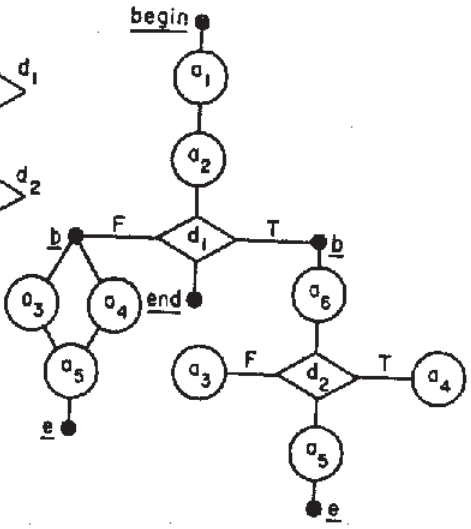
The cells of the schema are identified by the letters $m_1, m_2, \ldots$. Certain cells are designated as input or output cells. Values are assigned to the input cells before a computation begins; upon completion, the result is the set of values present in the output cells. Several operators, a and b, say, may have the same associated function letter: g(a) = g(b). In this way, a schema may require that two operators, a and b, always implement the same transformation, although the particular transformation is unspecified. Similarly, each decider designates a predicate letter p(d). The function letters and predicate letters of a schema make up two finite sets G and P.

The control of a computation schema is a specification of the order in which the operators and deciders of the data flow graph are permitted to act. In particular, the control indicates how further progress of computations is affected by the actions of the deciders. For the examples of computation schemata given below, we shall represent the control by precedence graphs. An example of a computation schema is the following.

S_1: data flow graph



precedence graph



Each diamond node in the precedence graph connects to two subordinate precedence graphs that specify alternative computations according to whether the designated decider has a true or false outcome. Operator a_2 in the data flow graph is an identity operator; the associated function $g(a_2)$ is always the identity function.

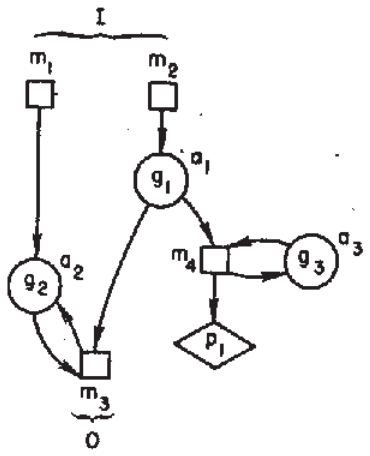
For schema S_1, the precedence graph allows just four distinct sequences of action by the operators and deciders of the schema. These sequences comprise the control set C of the schema

$$C_1: \left\{ \begin{array}{l} (a_1\ a_2\ f_1\ a_3\ a_4\ a_5) \\ (a_1\ a_2\ f_1\ a_4\ a_3\ a_5) \\ (a_1\ a_2\ t_1\ a_6\ f_2\ a_3\ a_5) \\ (a_1\ a_2\ t_1\ a_6\ t_2\ a_4\ a_5) \end{array} \right\}$$

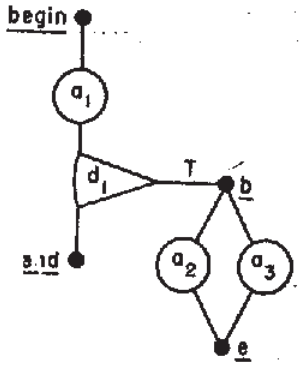
In these sequences, a_i stands for an action by operator a_i; f_i stands for an action by decider d_i for which the outcome is false; and t_i stands for an action by decider d_i for which the outcome is true. Since no iteration is present, the control set C_1 is finite.

Iteration is represented in a precedence graph by a pie-shaped node connected to a single subordinate precedence graph.

S_2: data flow graph



precedence graph



The computation specified by the subgraph is repeated until the decider acts with a false outcome. The control set for schema S_2 is C_2.

$$C_2: \left\{ \begin{array}{l} (a_1\ f_1) \\ (a_1\ t_1\ a_2\ a_3\ f_1) \\ (a_1\ t_1\ a_3\ a_2\ f_1) \\ (a_1\ t_1\ a_2\ a_3\ t_1\ a_2\ a_3\ f_1) \\ \quad \cdot \qquad \cdot \qquad \cdot \\ \quad \cdot \qquad \cdot \qquad \cdot \\ \quad \cdot \qquad \cdot \qquad \cdot \end{array} \right\}$$

To convert a computation schema into a specification of a particular algorithm, it is necessary to specify the functions and predicates designated by the function letters in G and the predicate letters in P. Of course, the specified functions and predicates must have domains and ranges consistent with the topology of the data flow graph, and must be in agreement whenever the value of a function may be the argument of a function or predicate. Such a specification of functions and predicates is called (after Yanov) an interpretation of a schema.

Two properties of schemata are of particular interest to us. A schema S is determinate if, for any interpretation of the function and predicate letters, S determines a functional relation of output tuples to input tuples. In order to say whether two schemata S_1 and S_2 describe the

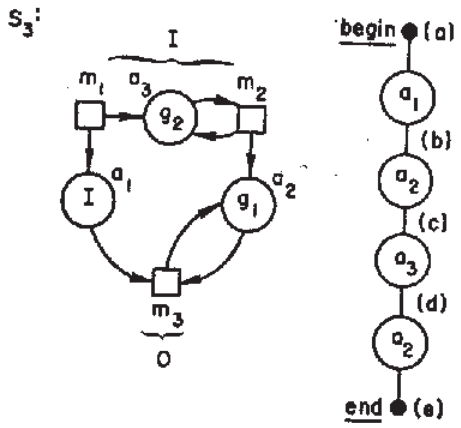
same computations, we must be able to relate the interpretations of the function and predicate letters in S_1 and S_2. For this purpose, let

$$G = G_1 \cup G_2 \qquad P = P_1 \cup P_2 \quad .$$

Then S_1 and S_2 are <u>equivalent</u> schemata if, for any interpretation of the functions and predicate letters in G and P, S_1 and S_2 determine precisely the same relation of output tuples to input tuples.

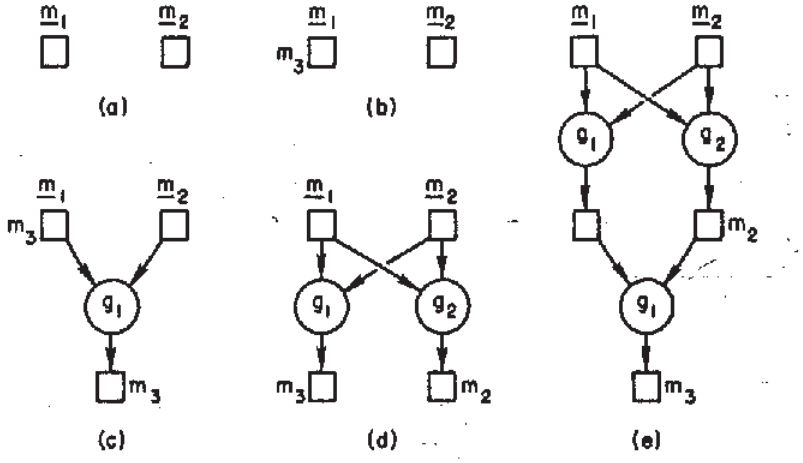
To develop insight into the questions of determinism and equivalence, we have devised the notion of <u>data-dependence graph</u> or <u>dadep graph</u> for short. A dadep graph of a schema sets forth separately each action by an operator or decider. For a particular control sequence of a schema, the final value placed in each output cell will be the result of some cascaded composition of functions. A dadep graph is just a graph representation of the cascade composition of operators associated with each output cell.

Let us construct the dadep graph for schema S_3 from its unique control sequence $\alpha = (a_1\ a_2\ a_3\ a_4)$.

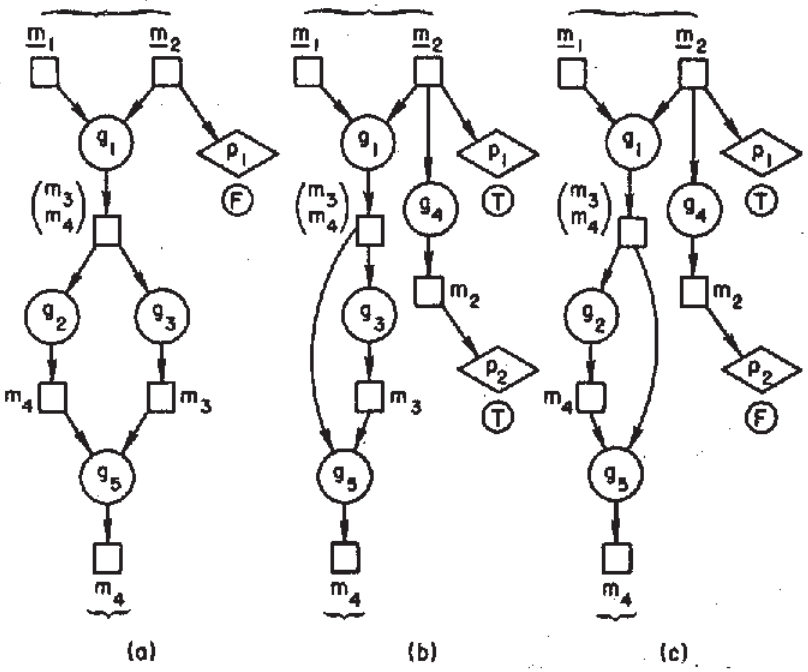


The construction is shown on the next page. We start by setting down a copy of each input cell of the schema. (The letters denoting these cells are underlined.) Then we add a copy of an operator and its output cells for each succeeding element of the control sequence. Each cell added to the dadep graph is labeled as in the data flow graph, and this label is erased from the cell copy previously bearing it. In the case of an identity operator, a second label is given to the most-recent copy of its input cell, and no copy of the operator is made.

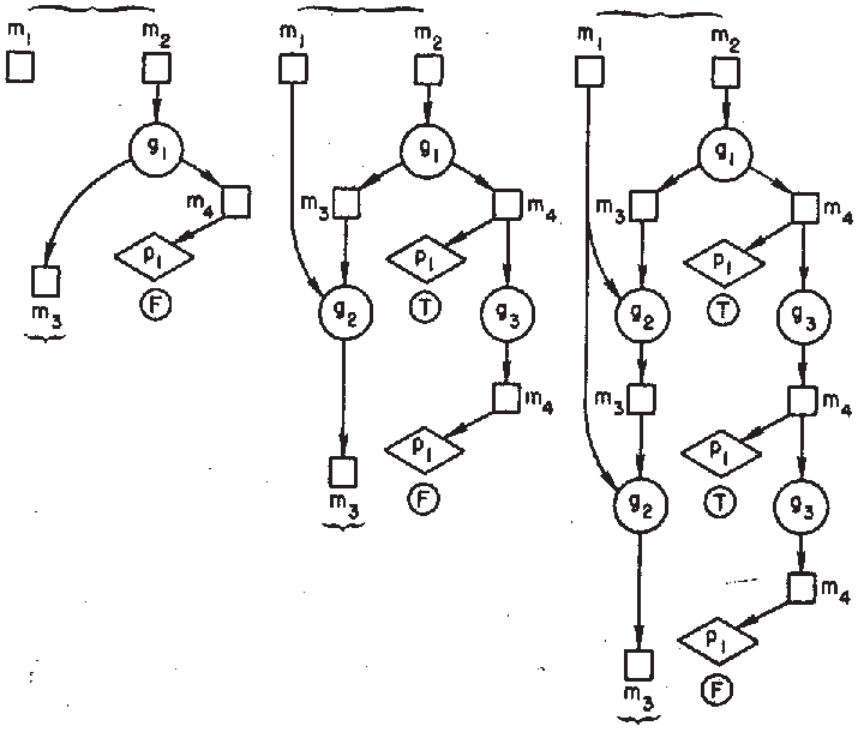
For schemata that include deciders, there will be a cascade composition of functions associated with each action of a decider as well as each



output cell. A determinate schema with k deciders could have 2^k distinct dadep graphs -- one for each combination of decisions that might occur in the course of some computation. For the schema S_1, there are just three dadep graphs because a decision of <u>false</u> by d_1 results in the absence of any action by d_2.



In general, a schema that represents an iteration defines an infinite set of dadep graphs. In the case of S_2, the three simplest dadep graphs are:

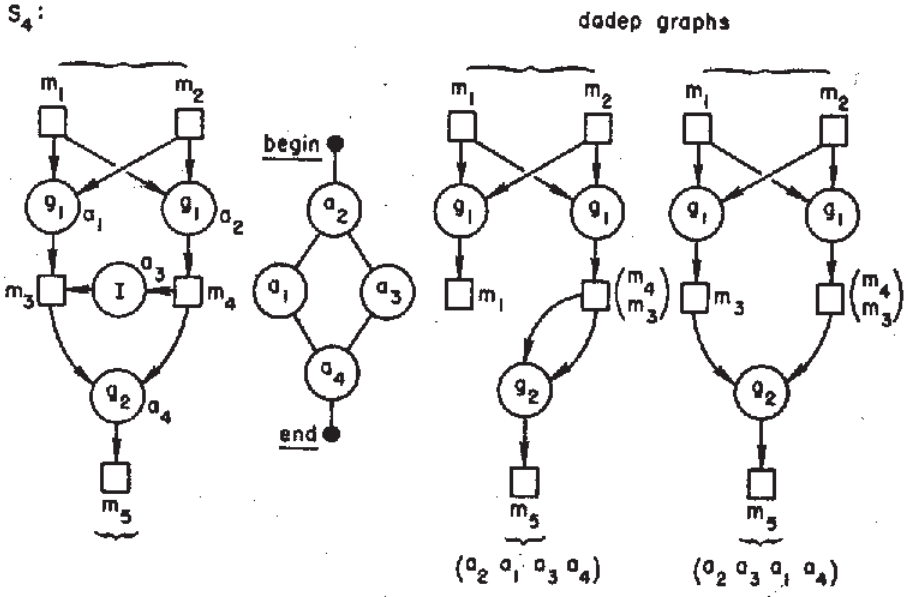


Certain properties are important in the study of schemata: A schema is <u>persistent</u> if the occurrence of one of two actions that could proceed concurrently does not inhibit or block the other action. Furthermore, a schema is <u>commutative</u> if the order in which two concurrent actions occur has no effect on the subsequent course of the computation.

Nondeterminate computation can occur only when a schema has a cell that could be assigned a value by one operator either before or after a value is assigned to or read from the cell by the action of another operator or decider. When this can happen we say the schema has a <u>conflict</u>.

By means of known methods it is not difficult to show that any computation schema that is persistent, commutative, and free of conflict is

guaranteed to be determinate. A more interesting problem is to determine the circumstances for which the conflict free property is a necessary condition for schemata to be determinate. We have studied two natural restrictions on schemata such that any determinate schema meeting the restrictions is necessarily conflict free. The first of these restrictions amounts to requiring that each action by any operator or decider in a schema participate in determining some output value. A schema meeting this restriction is said to be normal. The second restriction disallows control sets that permit repetition of a computation or test for the same m-tuple of input values. A schema meeting this restriction is said to be repetition-free. In schema S_4, repetition of the function designated by g_1 occurs. Because of the repetition, the conflict between operators a_1 and a_3 at cell m_3 fails to yield nondeterminate computations -- both dadep graphs define the same composition of functions.



For an elementary schema that is well defined, normal, repetition-free and determinate, all execution sequences yield the same dadep graph. In fact the dadep graph is a canonical form for this class of schemata. Thus the equivalence of any two elementary schemata can be tested by constructing their dadep graphs.

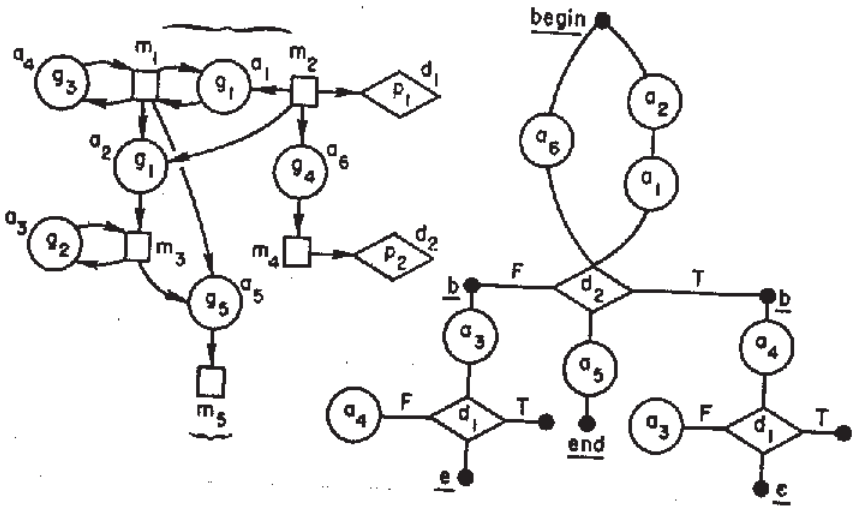
In the case of a normal, repetition-free schema that has deciders but no iteration, the class of computations represented is described by a finite set of dadep graphs, as shown for the schema S_1 earlier. Each pair of input values will be processed as shown in that one of the dadep graphs for which the evaluation of predicates agrees with the truth values given at decision points of the graph.

We can construct a table of two columns, called a conditional expression list, that characterizes the computations represented by a schema. Each row of the table corresponds to one dadep graph. In the left-hand column, we write a conjunction of the predicates that must be satisfied by the input variables for the corresponding dadep graph to describe the computation. In the right-hand column, we write the compositions of functions that specify the corresponding dependence of output values on input values. For S_1 we have:

Condition	Expression
$\overline{p}_1(x_2)$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_1(x_1, x_2), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot \overline{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_1(x_1, x_2))$

Now consider the schema S_5:

S_5:



Schema S_5 has four dadep graphs and is characterized by a conditional expression list with four entries:

Condition	Expression
$\bar{p}_1(x_2) \cdot \bar{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$\bar{p}_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_3(g_1(x_1, x_2)))$
$p_1(x_2) \cdot \bar{p}_2(g_4(x_2))$	$g_5(g_2(g_1(x_1, x_2)), g_1(x_1, x_2))$
$p_1(x_2) \cdot p_2(g_4(x_2))$	$g_5(g_1(x_1, x_2), g_3(g_1(x_1, x_2)))$

This table specifies the same class of computations as the conditional expression list for S_1, for we have the logical equivalence

$$\bar{p}_1(x_2) \equiv \bar{p}_1(x_2) \cdot \bar{p}_2(g_4(x_2)) + \bar{p}_1(x_2) \cdot p_2(g_4(x_2)) \quad .$$

Thus schemata S_1 and S_5 are equivalent. In general, noniterative schemata may be tested for equivalence by constructing their conditional expression lists.

Since an iterative schema has an infinite set of dadep graphs, its conditional expression list is infinitely long. For schema S_2 we have

Condition	Expression
$\bar{p}_1(g_1^2(x_2))$	$g_1^1(x_2)$
$p_1(g_1^2(x_2)) \cdot \bar{p}_1(g_3(g_1^2(x_2)))$	$g_2(x_1, g_1^1(x_2))$
$p_1(g_1^2(x_2)) \cdot p_1(g_3(g_1^2(x_2))) \cdot \bar{p}_1(g_3(g_3(g_1^2(x_2))))$	$g_2(x_1, g_2(x_1, g_1^1(x_2)))$
• • •	• •
• • •	• •
• • •	• •

We can show that, in general, two normal and repetition-free schemata are equivalent if and only if their conditional expression lists agree in the sense illustrated by our demonstration of equivalence for S_1 and S_5. When the lists are finite the existence of a decision procedure is

clear. At this time it is not known whether or not a decision procedure can be found for the more general equivalence problem.

VIII. CONTROLLED INFORMATION SHARING

The merit of the computer utility concept [15], lies in the ability of the users of the utility to build on each other's work. Thus the utility must provide orderly means for sharing access to procedures and data bases. We believe [16] that, to be successful, a utility must provide an environment in which a variety of information services may flourish and compete as private enterprises. Because proprietary and personal data will reside in the memory of a computer utility, access of users to stored information must be controlled so that only legitimate access is permitted.

Dean Vanderbilt has studied the implications of these requirements for sharing and access control on the organization and representation of procedures and data bases in a computer utility [17]. A computer utility must allow the owner of a program to authorize its use by other users without giving them the ability to view its internal structure. The execution of a program involves access to data and access to other programs. This additional information falls into two categories -- information that is associated with (shared by) all activations of the program; and the information that is associated with a particular activation (and not shared by several activations). The former category (Category I) consists of subprograms considered to be part of the program, subprograms of these subprograms, etc., and any data that are common to all activations of the programs. Category II information consists of all information passed as arguments to and from the program, and all temporary information generated during the particular activation.

During execution of a program, access to Category I and Category II information must be provided. Two aspects must be dealt with: First, the names used by the program to refer to this additional information must be bound to be the correct information. Second, the access control mechanisms of the utility must allow access to the information when it is needed.

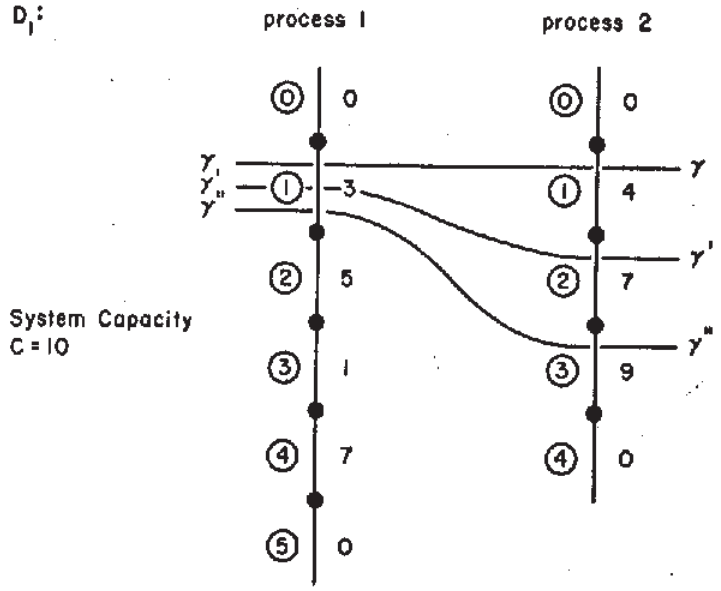
The Category I information is known to the owner -- the creator -- of the program, but not to the borrower. Thus the owner must specify the binding of names in the program to that information, and ensure that the information may be effectively referenced when needed during execution of the program. Since the program borrower should be granted no more access abilities than necessary, it must be possible for the owner to give the borrower the ability to access Category I information only in conjunction with use of the program. Thus, access abilities and binding information must be associated with the shared program so that the appropriate Category I information is available each time the program is executed.

The Category II information consists of information supplied by the program user and information created by the program. For the former, the supplied arguments must be bound to the program's call parameter names. The access abilities pose no problem since this information belongs to the program user. For the latter information, the process executing the program must be allowed to create information and to have it automatically bound to the appropriate names appearing in the program.

Dean Vanderbilt has designed an abstract program-execution environment [17] which offers one solution to the problems of implementing controlled access to shared information in a computer utility. This work uses a directed graph model of structured information that is closely related to the abstract information structures that form the foundation of our development of a base language, and is similar to the abstract "objects" used by the IBM Vienna Laboratory [18] for their work in formal semantics.

IX. RESOURCE SHARING WITHOUT DEADLOCK

Another form of concurrency is the cooperative activity of interacting computational processes, as in a multiprocess computer system. One form of interaction among processes is the implicit interaction arising from the sharing of limited resources. Consider, for example, two independent, sequential processes that progress through several distinct phases of activity.

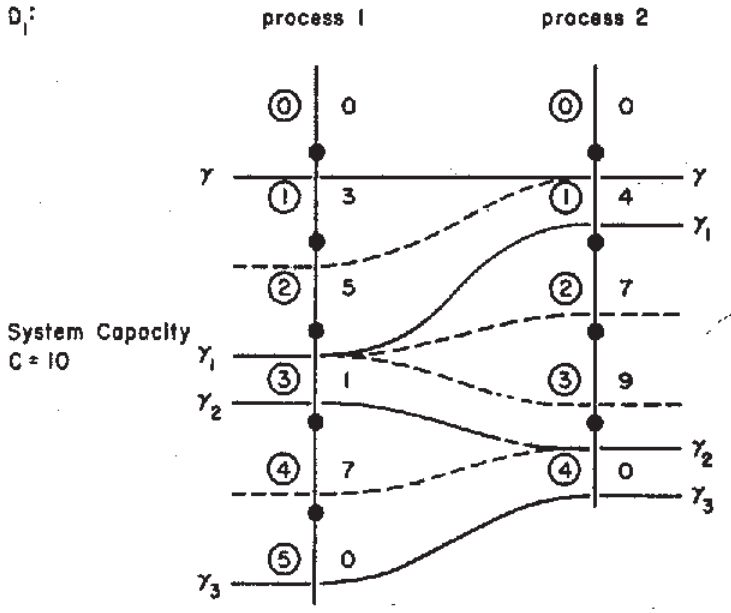


For each of its phases (identified by the circled numbers), a process requires the specified amount of a single resource type. The number of available units of the resource type (the system capacity) is C = 10. This representation of the resource requirements of a system of concurrent processes is called a demand graph. It is convenient to represent the composite state of the processes by a slice through the demand graph. For example, in the slice $\gamma = (①, ①)$ of demand graph D_1, both processes are engaged in phase 1 of their activity. Slice γ is feasible because the total resource units required is seven, which is less than the system resource capacity. If process 2 should complete phase 1, it could immediately proceed with phase 2, for the slice $\gamma' = (①, ②)$ is also feasible. However, process 2 could not continue into phase 3 of its activity because slice $\gamma'' = (①, ③)$ has a total resource requirement larger than the system capacity -- we say that slice γ'' is not feasible. The resource-allocation mechanism of a system should operate so that all processes can complete all phases of their activities, if possible, by a sequence of feasible slices. This kind of scheduling is not simply implemented if processes are assumed to retain control over resources during their transitions to new phases of activity. For instance, we must allow process 2 to retain the four units allocated to it for phase 1 while awaiting the release of three more units for its use in phase 2. Such hoarding occurs in computer systems where the resource may be memory areas, access to locked data bases, tape units, etc. When such hoarding is practiced, deadlocks can occur: Slice γ' in the demand graph D_1 is feasible, and represents a system state reachable by a sequence of feasible slices. Yet neither process can proceed beyond its phase in slice γ' for the lack of needed resource units -- the two processes are deadlocked. To avoid deadlock, the allocator must prevent the system from reaching the state corresponding to slice γ' even though the slice is feasible.

We call a slice γ in a demand graph safe if it is feasible and there is a sequence of phase transitions leading to a succession of feasible slices so that each process completes all remaining phases of its activity. If there is no such sequence of feasible slices, then slice γ is unsafe. Slice γ' in D_1 is unsafe. That slice γ is safe is demonstrated by showing, on the next page, a sequence of phase transitions to successive feasible slices that takes both processes through all remaining phases.

In these terms, the task of the resource allocator is to regulate the transitions of processes to new phases so that each slice attained is safe. It is not adequate to start the system of processes in a safe slice, for unsafe slices may be reached from a safe slice.

For demand graph D_1, we can discover that slice γ is safe by observing that process 1 goes through a phase of reduced demand during which



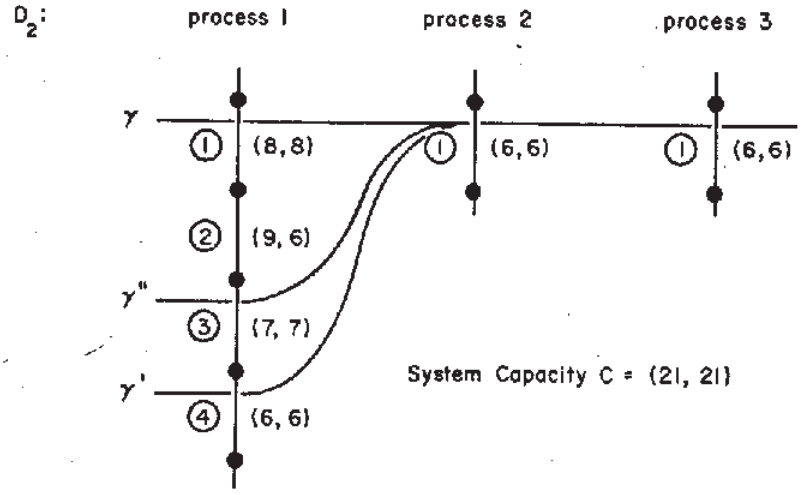
process 2 may advance to phase 4. In the absence of the detailed demand data given by the demand graph, this sort of reasoning cannot be applied, and less-complete resource usage is possible. For instance, if it is only known that processes 1 and 2 require maximum demands of 7 and 9 units, respectively, the system state represented by slice γ could not be permitted to occur.

In principle, one could examine all possible slices of a demand graph and determine whether each is safe before initiating any activity. However, this technique lacks flexibility, since adding a new process to a system of processes would require a suspension of activity while a redetermination of safeness of slices is carried out. It is also wasteful in that few of the slices of a demand graph will occur during a run of the system of processes. Incremental algorithms, which only test for safeness slices that are candidates for becoming the new current slice, are therefore of interest. Prakash Hebalkar [19] has formulated an algorithm for testing safeness that is non-enumerative and in which the amount of backtracking is minimal. This Safeness Algorithm attempts to construct a sequence of feasible slices from the slice under test to the terminal slice of the demand graph. The construction proceeds in steps, each of which consists of a series of phase transitions by one process. A step ends at the first phase that has a demand no greater than the demand for the process at the initial phase of the step. For demand graph D_1, the Safeness Algorithm produces

the sequence of steps $\gamma \longrightarrow \gamma_1 \longrightarrow \gamma_2 \longrightarrow \gamma_3$ to verify the safeness of slice γ. We have shown that failure of the algorithm to generate a sequence of feasible slices by which all processes complete their activity implies the slice under test is unsafe; conversely, success of the algorithm implies safeness. A resource allocator that uses the Safeness Algorithm to restrict system operation to only safe allocation states would make better use of resources without the possibility of deadlock.

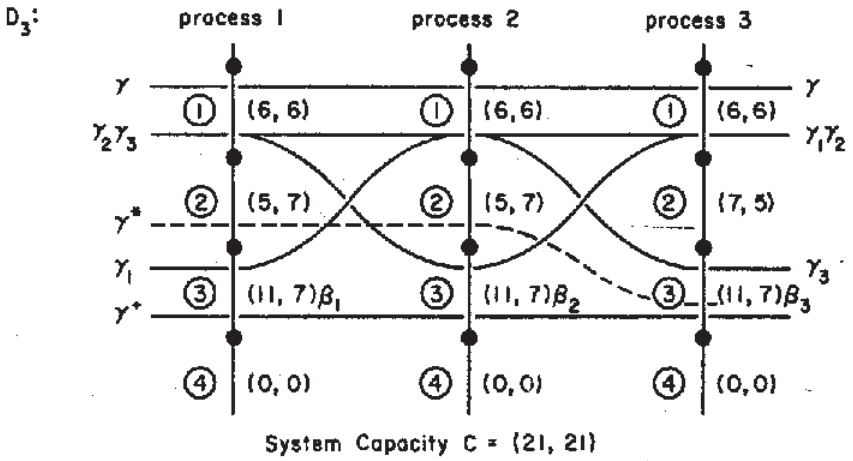
For systems in which more than one type of resource is shared, we have formulated an extension of the Safeness Algorithm and established its validity. However, the amount of computation can have a nonlinear dependence on the number of phases of the processes in the demand graph -- a problem that does not arise for systems with a single resource type. This is not a failing of our particular algorithm: We have shown that a local algorithm (one that is not permitted a bird's-eye view of the entire demand graph) will, for some cases, have to exhaustively search a large set of slices to determine that a slice is safe. The following example illustrates why this is so.

The extended Safeness Algorithm generates feasible slices in steps, as before. However, the series of phase transitions making up a step now ends only at a slice in which each component of demand is no greater than the same component in <u>each</u> prior phase of the series.



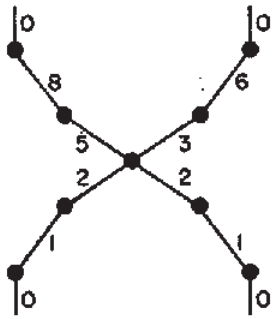
Thus in D_2, a step will consist of moving from γ to γ' rather than from γ to γ''.

Now consider the demand graph D_3.



Without further modification, using the Safeness Algorithm to search for a step from slice γ leads to failure for processes 1, 2 and 3 at slices γ_1, γ_2 and γ_3, respectively. Thus the algorithm would conclude (falsely) that γ is unsafe. A limited backtracking algorithm must discover some way of getting past the slice $\gamma+ = (③, ③, ③)$ consisting of the barrier arcs β_1, β_2 and β_3. From the study of the demand graph, it is evident that the slice $\gamma* = (②, ②, ③)$ must be used. But a local algorithm can determine this only through an exhaustive search of slices. The number of futile trials can be quite large.

The Safeness Algorithm can also be extended to systems in which explicit interactions between processes take place as well as the implicit interactions arising from resource-sharing. In studying this situation, we have discovered an interesting phenomenon, called intrinsic deadlock: There are demand graphs for which no schedule can permit the processes to complete their activity, for example:



This is a result of excessive hoarding of resources at points of (explicit) interaction. For this reason, among others, hoarding of resources at points of interaction should be held to the minimum.

The study of demand graphs is a perfectly general one that is not restricted to computer processes. Deadlock situations can arise from sharing of resources in road transportation, aircraft maintenance, and so on, and these operations can profit from analysis for the prevention of deadlocks.

X. WOODS HOLE CONFERENCE

The culmination of the year's activities of the Computation Structures Group was the sponsoring of an informal conference on Concurrent Systems and Parallel Computation. It was held at the National Academy of Sciences' Conference Center in Woods Hole, Massachusetts, during the first week of June 1970. Participants in the conference included six members of Project MAC and twenty-one persons representing most institutions in the United States that are carrying on theoretical research related to parallelism and concurrency.

The objective of the conference was to bring together people working along four distinct conceptual lines that we have found to be intimately related:

Representations of systems of concurrent events.
Speed-independent switching circuits.
Uninterpreted schemes of programs.
Cooperating sequential processes.

The conference was most successful in acquainting the participants with each other's ideas and in catalyzing many stimulating discussions.

Eleven technical papers were prepared for the conference, and were of such quality that they have been published collectively as a Conference Record [20]. For the conference we assembled an extensive collection of papers and reports related to the concepts of concurrency and parallelism. With the inclusion of a bibliography of this collection, the Record should be a valuable introduction to the field for interested research scientists.

Publications 1969-1970

Dennis, J. B., "Asynchronous Control Structures for a High Performance Processor", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, N.Y., 1970, pp. 55-80.

Gertz, J. L., Hierarchical Associative Memories for Parallel Computation, Ph.D. Thesis, Dept. of Electrical Engineering, June 1970, also MAC TR-69, AD-711-091.

(continued)

Publications 1969-1970 (cont.)

Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, N.Y., 1970, pp. 107-116.

Patil, S. S., Coordination of Asynchronous Events, Ph.D. Thesis, Dept. of Electrical Engineering, June 1970, also MAC-TR-72, AD-711-763.

Vanderbilt, D. H., Controlled Information Sharing in a Computer Utility, Ph.D. Thesis, Dept. of Electrical Engineering, October 1969, also MAC TR-67, AD-699-503.

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. S. S. Patil, Coordination of Asynchronous Events. Report MAC-TR-72, Project MAC, M.I.T., Cambridge, Massachusetts, June 1970.

4. J. B. Dennis, "Modular, Asynchronous Control Structures for a High Performance Processor". Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp. 55-80.

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

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

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

8. E. C. Van Horn, Computer Design for Asynchronously Reproducible Multiprocessing. Report MAC-TR-34, Project MAC, M.I.T., Cambridge, Massachusetts, 1966.

9. J. E. Rodriguez, A Graph Model for Parallel Computation. Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Massachusetts, 1969.

(continued)

References (cont.)

10. F. L. Luconi, Asynchronous Computational Structures. Report MAC-TR-49, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

11. D. R. Slutz, The Flow Graph Schemata Model of Parallel Computation. Report MAC-TR-53, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

12. Y. I. Yanov, "The Logical Schemes of Algorithms", Problems of Cybernetics, Vol. 1, Pergamon Press (1960), pp. 82-140.

13. R. M. Karp and R. E. Miller, "Parallel Program Schemata", J. of Computer and System Sciences, Vol. 3, No. 2 (May 1969), pp. 147-195.

14. M. S. Paterson, "Program Schemata", Machine Intelligence, Vol. 3, American Elsevier, New York (1968), pp. 18-31.

15. R. M. Fano, "The MAC System: The Computer Utility Approach", I.E.E.E. Spectrum, Vol. 2, No. 1 (January 1965), pp. 56-64.

16. J. B. Dennis, "A Position Paper on Computing and Communications", Comm. of the ACM, Vol. 11, No. 5 (May 1968), pp. 370-377.

17. D. H. Vanderbilt, Controlled Information Sharing in a Computer Utility. Report MAC-TR-67, Project MAC, M.I.T., Cambridge, Massachusetts, 1969.

18. P. Lucas, P. Lauer and H. Stigleitner, Method and Notation for the Formal Definition of Programming Languages. Technical Report TR 25.087, IBM Laboratory, Vienna, June 1968.

19. P. G. Hebalkar, Deadlock-Free Sharing of Resources in Asynchronous Systems. Report MAC-TR-75, Project MAC, M.I.T., September 1970.

20. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM. New York (1970).