Lecture 1

GRAPH THEORETIC MODELS OF PARALLEL COMPUTATION

R.E. Miller

Rapporteurs: Dr. P.E. Lauer
Dr. M.W. Shields

In the early 1960's under the direction of Professor G. Estrin a group at UCLA developed a "fixed plus variable" computer structure [41]. To study the scheduling and allocation problems of this parallel and flexible machine structure they developed some acyclic graph structures of the parallel computations [92]. At MIT various models of parallel computation were developed, some as Ph.D. theses under Professor Dennis. One of the early such models was that of Rodriguez [123]. One of the best known, and widely studied graphical models is one developed by Petri [116], now known as Petri nets. This structure still commands considerable research interest. In the early 1960's Karp and Miller started studying possible ways to speed up computers by adding special purpose units to machines, and through these studies developed a simple model known as a computation graph [69].

Rather than try to review all of these models in detail here I will briefly describe computation graphs and Petri nets (I presume most of you are already familiar with Petri nets). I will then discuss some relationships between computation graphs and Petri nets. Finally, I will introduce some synchronisation problems using semaphores, and show how these problems tie into our two models of parallel computation.

A: Petri Nets

Although I assume that most of you are familiar with Petri nets, a brief description - to introduce the terminology I use - will be given.

Definition 1: A Petri net $P = (\Pi, \Sigma, R, M_0)$ consists of:

(i) a finite set Π called places,
(ii) a finite set Σ called transitions,
(iii) a relation $R \subseteq (\Pi \times \Sigma) \cup (\Sigma \times \Pi)$, and
(iv) a mapping $M_0 : \Pi \to N$, called the initial marking, where N represent the set of nonnegative integers.

Usually a Petri net is represented by a graph in which places and transitions are represented by nodes, R is represented by directed edges, and M_0 is represented by dots in the place nodes. To distinguish the place and transition nodes, circles o are usually used for places and bars | are used for transitions. If $\pi \in \Pi$ and $\sigma \in \Sigma$ where $(\pi,\sigma) \in R$, then (π, σ) is represented by an edge directed from the node for π to the node for σ. Similarly for a $(\sigma, \pi) \in R$ by an edge from σ to π. Places are used to hold markers called tokens and M_0 assigns an initial number of tokens to each place.

For a given place π those transitions σ_i for which $(\sigma_i, \pi) \in R$ are called the input transitions of π and those σ_i for which $(\pi, \sigma_i) \in R$ are called the output transitions for π. Similarly, for a given $\sigma \in \Sigma$, those π_i for which $(\pi_i \sigma) \in R$ are called the input places of σ and those π_i for which $(\sigma, \pi_i) \in R$ are called the output places of σ. The Petri net is thus a fixed graphical structure which is supposed to represent the allowed sequencing of parallel processes. Usually the transitions are viewed as processes and the tokens on the input places of a transition are used to control the initiation of the process. A transition σ is called active or fireable if and only if each of its input places contains one or more tokens. An active transition σ may fire, and this can be interpreted as the execution of the process represented by σ. When σ fires it reduces by 1 the number of tokens in each of its input places, and increases by 1 the number of tokens in each of its output places. The firing of a transition thus changes the distribution of tokens on places. Such a distribution of tokens is called a marking. Through the marking change other transitions may become active. It is the sequence of transition firings that is used to represent the computation sequence in a Petri net. A sequence of transition firings is called a firing sequence. It also defines, given an initial marking, a marking sequence. Since a given place may be in the set of input places for more than one transition it is possible that a single token in a place causes more than one transition to be fireable. To prevent the number of tokens upon transition firing to become negative it is assumed that a token is used in only a single transition firing. This is assumed formally in yet another way, namely by defining firing sequences to be a sequence of transition labels, implying that even though several transitions are simultaneously fireable, no simultaneous firing is allowed in the formal study. Thus the next element in a firing sequence is one of the transition labels as picked arbitrarily from the current set of fireable transitions. This representation of simultaneous action by different sequences of action will be commented on further in my third lecture. It is not clear that this is always a good idealisation.

Some simple examples of Petri nets are helpful in understanding their operation.

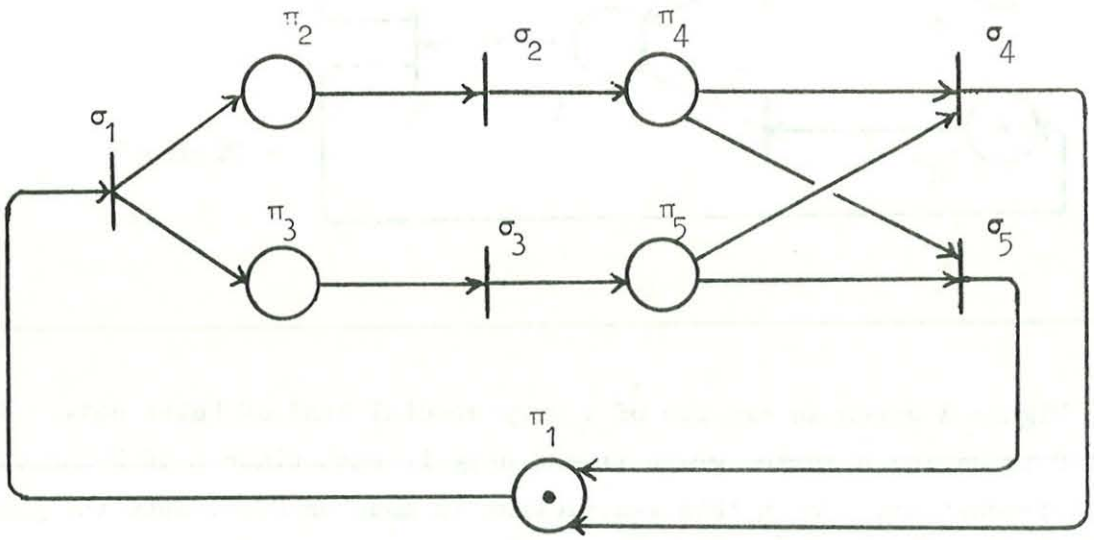


Figure 1

In Figure 1, with the initial marking having a token only in place π_1, the only fireable transition is σ_1. When σ_1 fires it removes the token from π_1 and places tokens in π_2 and π_3. Then σ_2 and σ_3 are fireable so parallel computation is represented. The structure π_4 and π_5 then both σ_4 and σ_5 are fireable, but because of the rule on a token being useable only in one firing not both σ_4 and σ_5 can fire. A choice must be made. We say in general that a pair of transitions σ_i and σ_j are in conflict under a given marking M if both σ_i and σ_j are active in M and there is some place π_k belonging to the input places of both σ_i and σ_j with $M(\pi_k) = 1$. It is precisely under the conflict situation that although both transitions are simultaneously active they cannot simultaneously fire.

The Petri net of Figure 2 is an example that shows that the number of tokens may grow unboundedly in a place. Here a single firing of both σ_1 and σ_2 causes π_3 to have two tokens. A single firing of σ_3 places tokens back in π_1 and π_2 leaving one token in π_3. Repeating this cycle of transition firings causes the number of tokens in π_3 to grow to as large a number as desired.

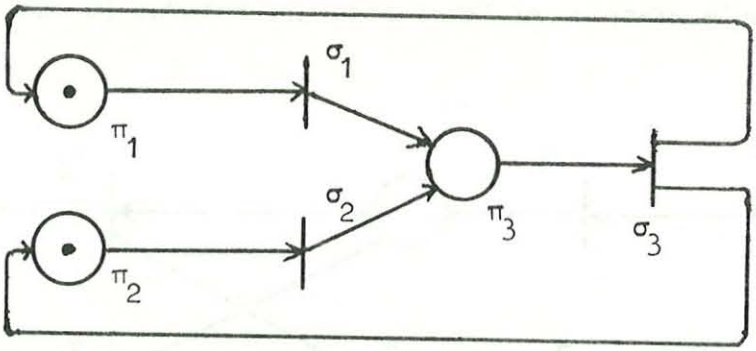


Figure 2

Figure 3 gives an example of a very special kind of Petri net. A Petri net P is called a marked graph if and only if each place π of P has exactly one output transition. When this restriction is made on Petri nets the graph can be simplified by absorbing each place into an edge and then letting the place marking be represented by a marking on the edge.

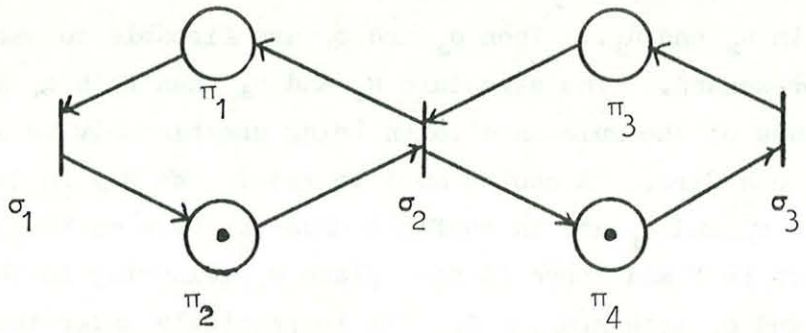


Figure 3

Similarly, restricting a Petri net so that each transition has exactly one input place and one output place gives a special class of Petri nets called state machines. This is readily seen by simplifying the graph as done by letting each transition now be represented by a directed edge from its input place to its output place. This then assumes the structure of a transition diagram of a finite state machine, but here the edges are not labelled. If one assumes an initial marking now as a single token in a single place (representing the start state) then state to state transitions correspond to transition firings. The analogy is too obvious to belabor. Both the marked graphs and state machines are subclasses of Petri nets that are considerably easier to analyse than general Petri nets. Other subclasses of Petri nets have also been defined and extensively studied. A number of properties of Petri nets are of interest and worth defining. First we note that any marking M of a Petri net P with n places can be viewed as an n-dimensional vector in which the value of the i^{th} coordinate of the vector is the number of tokens in the i^{th} place of P.

Definition 2: The reachable set of markings $R(P,M_o)$ of a Petri net $P = (\Pi,\Sigma,R,M_o) = \{M|$ a marking sequence starting with M_o and ending with $M\}$.

Definition 3: A Petri net P is called safe if $M \in R(P,M_o)$ implies that each coordinate of M is either zero or one.

Thus a safe net is a net in which the number of tokens in any place never exceeds one. This property is of interest when for some practical considerations one is interpreting the Petri net to represent a set of inter-related events and conditions, where conditions are represented as places. A condition is interpreted as holding if the place contains a token, and as not holding if the place does not contain a token. For such situations it is senseless to have more than one token in a place, so one wants to know that the net representing events and conditions is a safe net.

A natural extension of safeness is k-bounded or k-safe.

Definition 4: A Petri net P is called k-safe if $M \in R(P,M_o)$ implies that each coordinate of M takes on values from the set $\{0,1,2,\ldots,k\}$.

A second property of Petri nets is related to the current or eventual fireability of transitions.

Definition 5: A transition σ of a Petri net $P = \{\Pi,\Sigma,R,M_o\}$ is called live if and only if for every $M \in R(P,M_o)$ there is some firing sequence continuing from M which fires σ. The transition σ is called dead with respect to M if there is no firing sequence continuing from M which fires σ.

Definition 6: A Petri net P is called live if every transition of P is live.

The relevance of the property of liveness is evident when one interprets the transitions of the Petri net as representing processes. Liveness of a transition means that there is no way in which a sequence of process executions can cause the system to get into a state from which the given process can never again be executed. Thus both the liveness and deadness properties of Petri nets are related to the concept of deadlocks in operating systems.

Given any Petri net P we would like to know how to determine if P is safe, k-safe, live, or what transitions are dead, and with respect to what markings. We will approach these problems via vector addition systems in the next lecture.

One of the very intriguing aspects of Petri nets is the simple and illustrious way in which they represent parallel sequencing. Some researchers have enriched the model by various techniques. For example, by providing tokens of different colours, by inhibitor edges, and by timings. It appears that any such addition, although quite helpful for representing certain behaviours, turns the model into one that can simulate a Turing machine, and in that sense makes it hopeless to completely analyse.

B: Computation Graphs

We now switch to discussing a different graphical model of parallel computation called the computation graph. This was introduced in [69] and studied and extended in a number of further studies; e.g., [1, 96, 120].

Basic Definitions

Definition 7: A computation graph G is a finite directed graph consisting of:

(i) nodes $n_1, n_2, \ldots, n_\ell$.

(ii) edges $d_1, d_2, \ldots, d_t$, where any given edge d_p is directed from a specified node n_i to a specified node n_j.

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

In a computation graph each node n_i is used to represent an operation O_i and each edge is used to represent a first-in-first-out queue of data. Thus an edge d_p directed from n_i to n_j represents a queue of data flowing from n_i to n_j. Results of operation O_i represented by n_i are placed in the queue and may later be used as operands for operation O_j represented by n_j. The four parameters on edge d_p are interpreted as follows:

(1) A_p is the number of items initially in the queue from n_i to n_j.

(2) U_p is the number of items added to the queue each time operations O_i terminates.

(3) W_p is the number of items removed from the queue each time operation O_j initiates.

(4) T_p is a threshold giving the minimum number of items required in the queue

before operation O_j can initiate.

Computations are represented in a computation graph as sequences of operation performances. An operation O_j, associated with node n_j is said to be eligible for initiation if and only if each branch d_p directed into n_j contains at least T_p items in its queue. It is assumed that no two performances of a given operation O_j can be initiated simultaneously. When O_j is initiated W_p items are removed from the queue of edge d_p for each such edge directed into n_j. When O_j terminates each edge d_q directed out of n_j has U_q items added to its queue.

These definitions of operation initiation and termination describe how computations of the computation graph are sequenced. Note that the actual times required for operation performance are not specified. They are, in essence, asynchronous. The possible sequences of initiations for computation graphs are called executions. An execution is represented as a sequence of sets $E = S_1, S_2, \ldots, S_n, \ldots$ such that each S_n is a subset of $\{1, 2, \ldots, \ell\}$, the set of nodes indices. If $j \in S_n$ then this means that O_j is initiated at step n in execution E. To be more precise we define $x(j,n)$ for $j \in \{1, 2, \ldots, \ell\}$ and $n = 0, 1, 2, \ldots$ as:

$x(j,0) = 0$

$x(j,n)$ = the number of sets S_m, $1 \leq m \leq n$, for which j is an element.

That is, $x(j,n)$ is the number of initiations of operation j in the prefix $S_1, S_2, \ldots, S_n$ of execution E. With this notation we can define executions more precisely.

Definition 8: The sequence $E = S_1, S_2, \ldots, S_n, \ldots$ is an execution of the computation graph G if and only if, for all n, the following conditions hold:

(i) if $j \in S_{n+1}$ and G has an edge from n_i to n_j, then

$$A_p + U_p x(i,n) - W_p x(j,n) \geq T_p$$

(ii) if E is finite and of length r, then for each n_j there exists an n_i such that d_p is an edge from n_i to n_j and

$$A_p + U_p x(i,r) - W_p x(j,r) < T_p$$

Definition 9: An execution E is called proper if the following implication holds:

(iii) if, for all n_i and every edge d_p directed from n_i to n_j

$$A_p + U_p x(i,n) - W_p x(j,n) \geq T_p,$$

then $j \in S_r$ for some $r > n$.

In an execution the occurrence of a set S_n in the sequence denotes the simultaneous initiation of O_j for all $j \in S_n$. This model is one of the few that formally (rather than just informally) allows for simultaneous initiation of operations.

Thus, an execution E is viewed as a sequence of sets of events, not necessarily equally spaced in time, where an event is the initiation of an operation of G. As performances of operations in G proceed they generate an execution prefix. Each time an event, or set of simultaneous events, occurs a new element of the execution is generated.

The linear forms

$$A_p + U_p x(i,n) - W_p x(j,n)$$

associated with each edge d_p of G and each S_n of an execution gives the number of items in the queue associated with d_p at this point in the execution if we assume that all of the operations up to this point in E have actually terminated. Thus, part (i) of the definition for executions insures that sufficient items are in the queues for O_j to initiate. Condition (ii) insures that an execution will terminate only when no further operations are eligible for initiation. Part (iii), for proper executions, insures that if an operation becomes eligible for initiation at a certain step, then it will actually be initiated after some finite number of steps. This property, often called the "finite delay property," occurs in various forms in different models of parallel asynchronous computation and was apparently first introduced via asynchronous logic circuits by D.E. Muller.

In an execution E terminations are not explicitly mentioned. This does not mean, however, that an execution physically is a set S_n of operations that all initiate simultaneously and then all terminate before any further initiations. For example, if the inequality (stronger than that of (i) in Definition 8)

$$A_p + U_p(x(i,n)-1) - W_p x(j,n) \geq T_p,$$

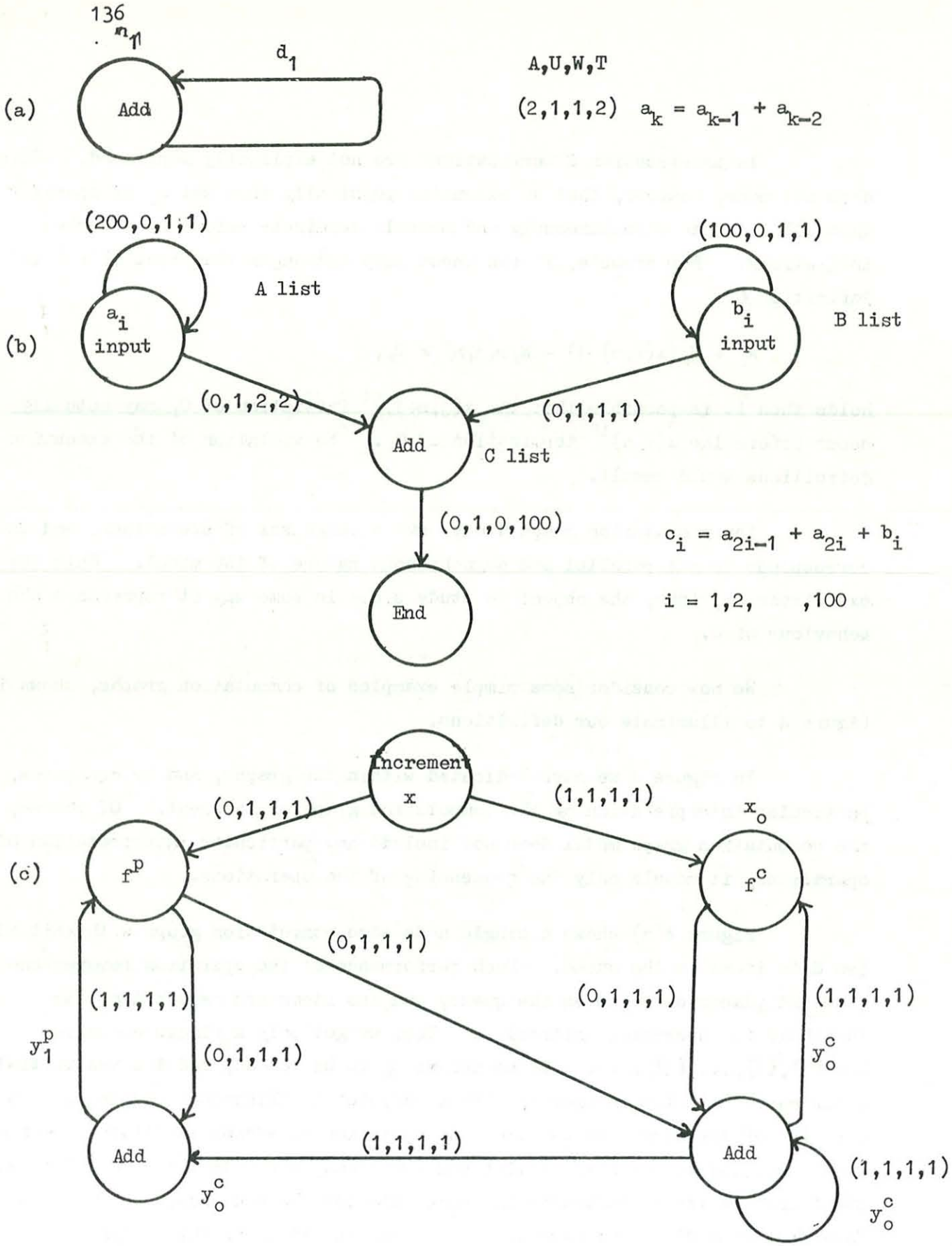
holds then it is possible that the $x(j,n+1)^{st}$ initiation of O_j may actually occur before the $x(i,n)^{th}$ termination of O_i. No violation of the execution definitions would result.

Any computation graph G may have a large set of executions, and this corresponds to the parallel and asynchronous nature of the model. This set of executions is, thus, the object to study since in some way it represents the behaviour of G.

We now consider some simple examples of computation graphs, shown in Figure 4 to illustrate our definitions.

In Figure 4 we have indicated within the graphs, and by equations, a particular interpretation of the computation graph of interest. Of course, the computation graph model does not include any particular interpretation of operations, it models only the sequencing of the operations.

Figure 4(a) shows a single node edge computation graph with initially two data items in the queue. Each performance of the operation removes one item and places one item on the queue, and two items are required as the threshold for operation initiation. Here we get only a single execution $E = \{1\},\{1\},\ldots,\{1\},\ldots$. If we assume O_1 to be an add, and the two initial items each to be the integer 1, then E computes the Fibonacci sequence. In part (b) of the figure we can view the operation as adding two lists together (see equation) in which the A list has 200 items, the B list has 100 items and the C list, which is formed on the edge entering the end node, has 100 items. Note that many different sequences of execution exist for this graph.



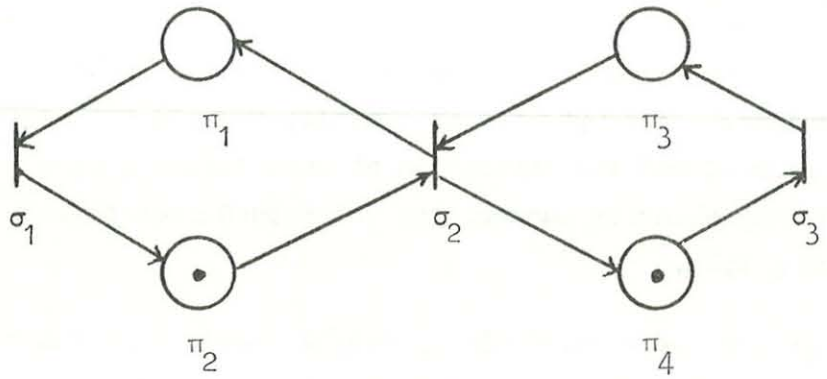
$$y^p_{n+1} = y^p_{n-1} + 2hf^p_n(x_n, y^p_n)$$

$$y^c_n = y^c_{n-1} + \frac{h}{2} f^p_n (x_n, y^p_n) + \frac{h}{2} f^c_{n-1} (x_{n-1}, y^c_{n-1})$$

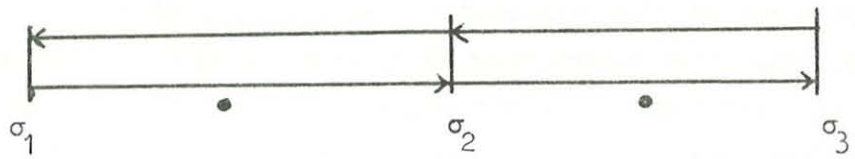
Figure 4

Part (c) of Figure 4 depicts a parallel predictor-corrector scheme of computation for an ordinary differential equation devised by Miranker [101]. The computation graph can be analysed to determine the amount of parallelism possible in this computation.

Previously we defined a marked graph to be a special type of Petri net in which each place $\pi \in \Pi$ has exactly one input transition and one output transition. Thus, the places in a marked graph can be absorbed into edges from transition to transition where tokens are then thought of lying on the edges. Our example marked graph

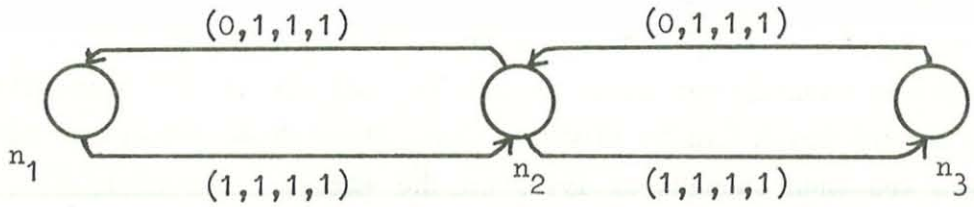


then becomes:



This graph can now be considered to be a computation graph, of the same node and edge structure. The number of tokens on an edge become the number of items in the queue associated with the edge, and the transition firing rules directly transform into the restriction that for any edge d_p of the computation graph $U_p = W_p = T_p = 1$. The A_p values correspond to the initial marking M_0. A formal correspondence, which should be obvious from this informal discussion, thus could be given. Thereby each firing sequence of the marked graph would correspond to an execution of the computation graph. Executions of the computation graphs having sets S_n with $|S_n| > 1$ would not, however, correspond

directly to a single firing sequence but rather a subset of firing sequences where each such S_n would result in an arbitrary ordering of firings. Our example marked graph then becomes the following computation graph,



where n_i corresponds to σ_i, $i = 1,2,3$.

Through this correspondence results of computation graphs can be directly applied to marked graphs. See [96] for an example; we will not amplify this here. S.C. Meyer has generalised the definition of computation graphs and investigated, in a much more comprehensive manner, the correspondences between computation graphs and marked graphs.

Since computation graphs are a particularly simple model, much can be proved about them. For example, if each operation is assumed to perform a specific function (from input variables to output variables) then it can be shown that the sequences of data on the queues is independent of the particular execution chosen. That is, computation graphs are determinate. Also, using appropriate algebraic manipulation of the inequalities associated with a computation graph algorithms exist for determining which operations of a computation graph terminate, and with how many performances; which operations have an unbounded number of performances, and what the bounds on queue lengths of data are during executions.

C: Relation of Synchronisation Problems to Computation Graphs and Petri Nets.

To introduce some simple synchronisation problems it is convenient to discuss semaphores.

The concept of semaphores was introduced by Dijkstra to provide a means of coordinating cooperative sequential processes. Several mutual exclusion problems and producer-consumer problems were also discussed there as prime examples for the use of semaphores.

Definition 10: A semaphore is a nonnegative integer valued variable which can be accessed by program processes only by the two types of instructions P(s) and V(s) defined below.

This definition is not made completely precise here; e.g., we leave undefined some of the concepts, such as "processes" used in the definition. It is hoped, however, that the notions will be clear from the examples we discuss.

Definition 11: P(s) is an indivisible operation on a semaphore s. P(s) at location L is defined as:

L: if $s < 1$ go to L else $s - 1$.

Definition 12: V(s) is an indivisible operation on a semaphore s defined by:

$s \leftarrow s + 1$.

The indivisibility of the P and V operations means that once started the operation must be completed with interaction or interference by any other P or V operation. Thus if $V_1(s)$ and $V_2(s)$ were acting on a single semaphore "simultaneously" the semaphore value would be increased by 2 after completion of both $V_1(s)$ and $V_2(s)$. If a semaphore value was 1 and two P operations were attempting to operate on it, only one of the P operations (arbitrarily determined) would be allowed to proceed, decreasing the value to 0. The other P operation would have to wait until the semaphore was again increased and then it would have to compete with any other P operations on that semaphore that were attempting to operate. If the original value had been 3, then both P operations could proceed simultaneously with the semaphore value ending up at value 1.

As an example of semaphore usage consider the two process example depicted in Figure 5

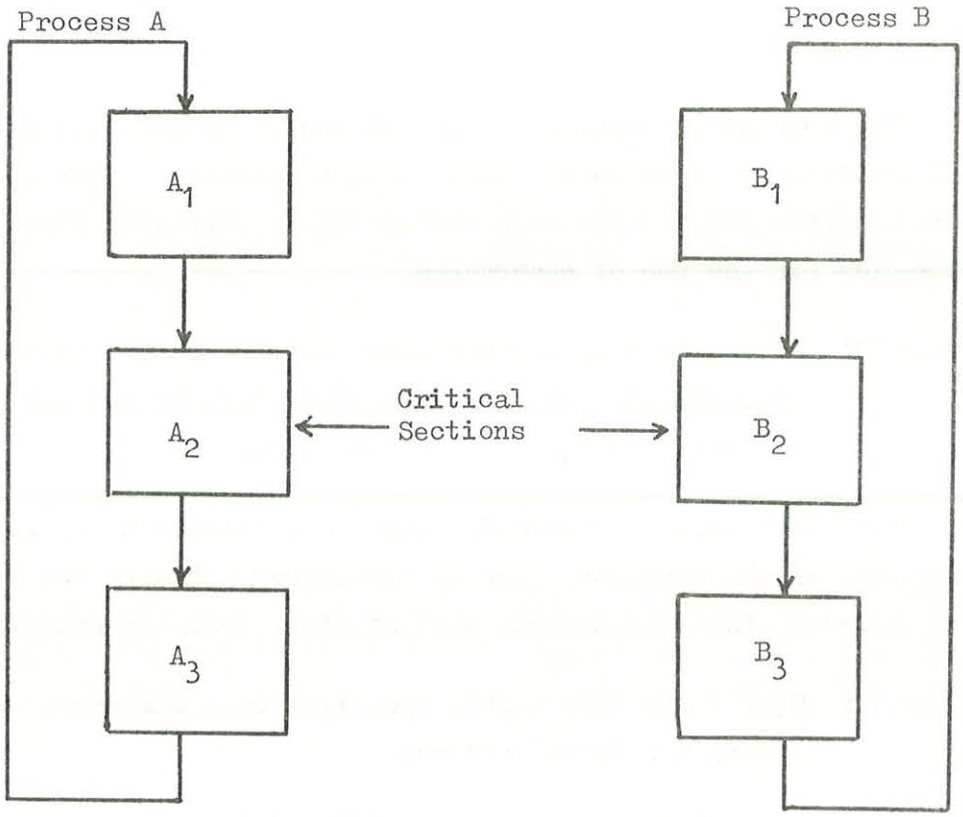


Figure 5
Mutual
Exclusion
Example

In this example we consider the two processes A and B, each having three subprocesses which are performed cyclically. The two processes can be performed in parallel with the constraint that the two critical sections A_2 and B_2 are constrained so as not to run simultaneously. That is, if A_2 is being performed, then B_1 or B_3 could also be in some stage of performance, but B_2 is prohibited from starting before A_2 finishes. The constraint is symmetrically imposed for B_2 being performed. Other than that, no constraints beyond the individual cyclic action are desired. This mutual exclusion problem is a common one in practice. Assume, for example, that A_2 and B_2 are performing different functions on a common file, then concurrent operation of these two functions could cause a malfunction.

The desired sequencing constraint is readily implemented with a single semaphore s, initially set to value 1. Then each subprocess A_2 and B_2 can be started with a P(s) instruction and ended with a V(s) instruction. Of course, the arbitrary choice of which P operation is performed when two are attempting to be performed allows cases in which just one of the processes, either A or B, dominates the situation so that the other process is eternally waiting at the

start of its critical section without ever actually getting a chance to be performed. Some more complicated solutions prohibit such "eternal" waits.

This simple mutual exclusion problem is readily modelled by a Petri net - as shown in Figure 6.

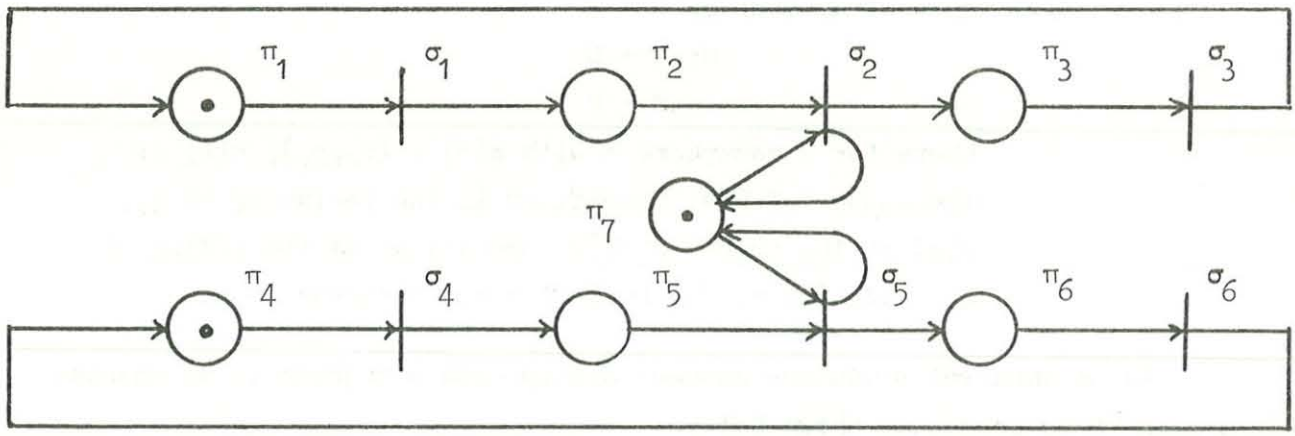


Figure 6

Here transitions σ_1, σ_2, and σ_3 are used to represent the subprocesses A_1, A_2, and A_3, respectively, of process A; and σ_4, σ_5, and σ_6 represent B_1, B_2, and B_3 respectively. Also, place π_7 represents semaphore s, and it is tokens in this place that control the mutually exclusive firings of σ_2 and σ_5.

We now turn to another class of synchronisation problems called producer-consumer problems.

Producer-consumer problems are an important class of synchronisation problems that arise when one considers the interconnection of a set of processes. Essentially, the idea of a producer-consumer system is that a given process of the system produces results that are used (consumed) by some other process. Such problems should be common in distributed systems. We first define a restricted system we call unshared.

Definition 13: An unshared producer-consumer system S consists of:

(i) a finite set $B = \{p_1, p_2, \ldots, p_\ell\}$ of processes,

(ii) a finite set $S = \{s_1, s_2, \ldots, s_t\}$ of semaphores,

(iii) a function $\alpha: S \to B \times B$ which associates an ordered pair of processes with each semaphore,

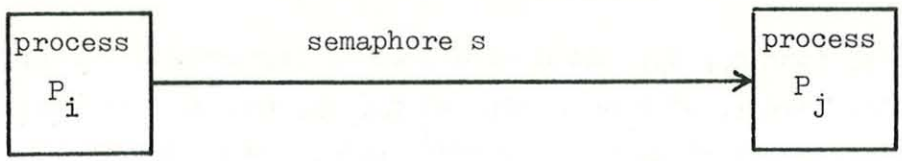
(iv) three functions $\mu: S \to N$

$\pi: S \to N$

$\nu: S \to N$

where for a semaphore s with $\alpha(s) = (p_i, p_j)$, $\pi(s)$ is the number of P(s) operations in the beginning of p_j, $\nu(s)$ is the number of V(s) operations at the ending of p_i, and $\mu(s)$ is the initial value assigned to s.

In an unshared producer-consumer system each semaphore is associated with a pair of processes as shown below:



Here the process p_i is thought of as the "producer" of results for "consumer" p_j, where P and V operations are used to indicate to the consumer when sufficient items have been produced for the consumer to start.

This unshared producer-consumer is a very restricted usage of semaphores. The semaphore is "private" to the producer-consumer pair rather than being shared by several producers or several consumers. However a process may be considered to be a producer (or consumer) for several processes, just as long as one semaphore is used for each producer-consumer pair.

A fairly direct representation of unshared producer-consumer systems by computation graphs should be evident. For an unshared producer-consumer system S of ℓ processes and t semaphores we can construct a computation graph G_S with ℓ nodes and t edges.

Each process p_i of S is represented by a node n_i of G_S, and each semaphore s_k is represented by an edge d_k of G_S directed from n_i to n_j if $\alpha(s_k) = (p_i, p_j)$. The parameters A_k, U_k, W_k, and T_k are defined as:

$$A_k = \mu(s_k)$$
$$U_k = \nu(s_k)$$
$$W_k = T_k = \pi(s_k).$$

With this representation, the performance of an operation O_j associated with n_j of G_S corresponds to the performance of process p_j of S. An execution of G_S corresponds to an allowed sequence of process performances in S, where termination properties of the two systems correspond, and where queue length of d_k corresponds to attained semaphore value of s_k.

This correspondence also shows why the generalised P and V operations often called PV Chunk, are natural extensions of P's and V's to consider.

An example of the computation graph G_S for an unshared producer-consumer system is shown in Figure 7.

This system S, by Definition 13 is:

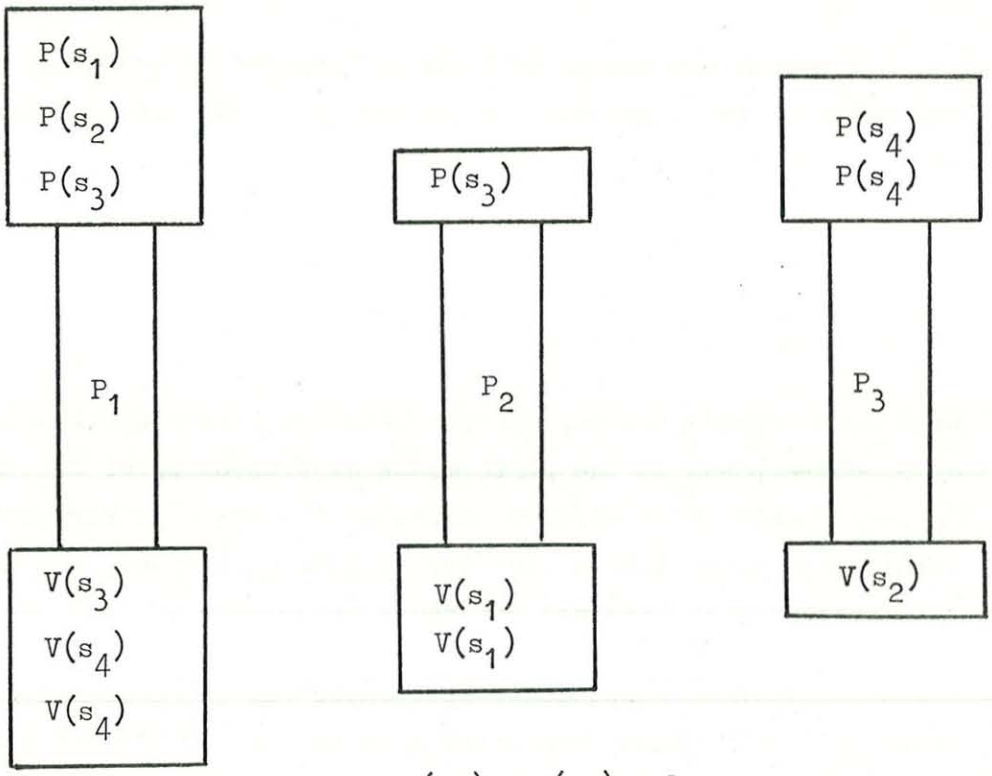
$B = \{p_1, p_2, p_3\}$ $\quad S = \{s_1, s_2, s_3, s_4\}$

$\alpha(s_1) = (p_2, p_1\}$	$\pi(s_1) = 2$	$\nu(s_1) = 2$
$\alpha(s_2) = (p_3, p_1,)$	$\pi(s_2) = 1$	$\nu(s_2) = 1$
$\alpha(s_3) = (p_1, p_2)$	$\pi(s_3) = 1$	$\nu(s_3) = 1$
$\alpha(s_4) = (p_1, p_3)$	$\pi(s_4) = 2$	$\nu(s_4) = 2$

We see that in this example initially only process p_3 can start. When p_3 terminates and updates s_2 then p_1 can start. When p_1 finishes and updates s_3 and s_4 then both p_2 and p_3 can start. Process p_1 can initiate again only when both p_2 and p_3 have finished.

The "unshared" aspect of the systems we have just defined is quite restrictive. We generalise.

(a) S



$\mu(s_1) = \mu(s_4) = 2$

$\mu(s_2) = \mu(s_3) = 0$

(b) G_S

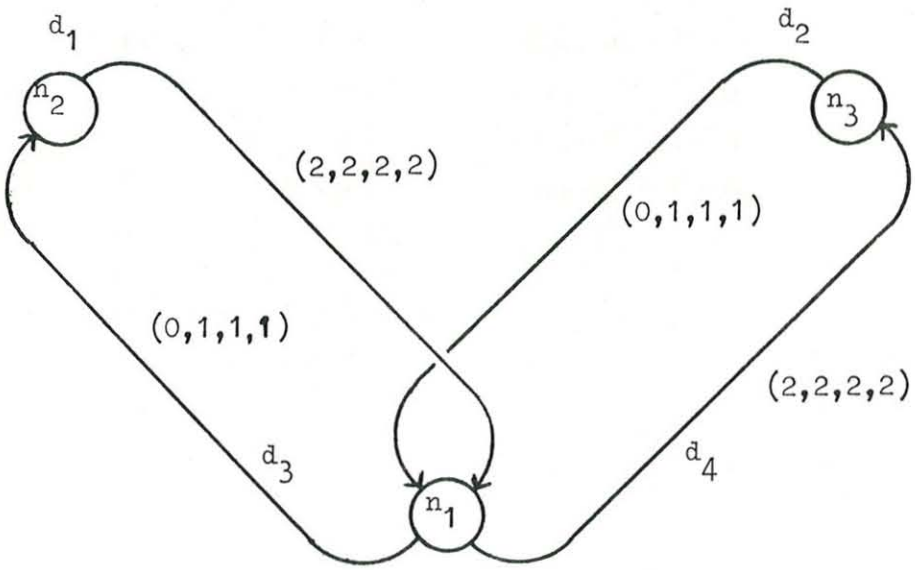


Figure 7

Unshared Producer – Consumer System S and Corresponding Computation Graph G_S

Definition 14: A producer-consumer system S, consiste of:

(i) a finite set $B = \{p_1, p_2, \ldots p_l\}$ of processes,

(ii) a finite set $S = \{s_1, s_2, \ldots, s_t\}$ of semaphores,

(iii) three functions $\mu: S \rightarrow N$

$\pi': S \times B \rightarrow N$

$\nu': S \times B \rightarrow N$

where for any $s \in S$ and $p \in B$, $\mu(s)$ is the initial value of s, $\pi'(s,p)$ is the number of P(s) operations at the beginning of p, and $\nu'(s,p)$ is the number of V(s) operations at the end of p.

Here the π' and ν' functions let a semaphore be used by any process. As before, however, we assume all P operations to occur at the start of a process and all V operations to occur at the end of a process.

There is a correspondence between producer-consumer systems and a generalized form of Petri net, which we now define.

Definition 15: A generalized Petri net $P = (\Pi, \Sigma, R, M_o, \Delta_I, \Delta_O)$ consists of:

(i) a finite set Π called places,

(ii) a finite set Σ called transitions,

(iii) a relation $R \subseteq (\Pi \times \Sigma) \cup (\Sigma \times \Pi)$,

(iv) a mapping $M_o: \Pi \rightarrow N$, called the initial marking, and

(v) two functions $\Delta_I: (\Pi \times \Sigma) \rightarrow N$ and $\Delta_O: (\Sigma \times \Pi) \rightarrow N$, where for $\pi \in \Pi$ and $\sigma \in \Sigma$, $\Delta_I(\pi, \sigma) = 0$ if and only if $(\pi, \sigma) \notin R$ and $\Delta_O(\sigma, \pi) = 0$ if and only if $(\sigma, \pi) \notin R$.

A generalized Petri net is like a Petri net (condition (i) through (iv)) with added functions Δ_I and Δ_O. These functions define the amount by which the number of tokens on a place π change by the firing of a transition σ. A transition is called active or fireable in a generalized Petri net if each input place π to σ, contains at least $\Delta_I(\pi, \sigma)$ tokens. The firing of an active transition σ changes the number of tokens on a place π by the amount $\Delta_O(\sigma, \pi) - \Delta_I(\pi, \sigma)$. We use the same terminology and concepts developed for Petri nets to discuss generalized Petri nets. The only extension being generalizing the removal and addition of tokens by transition firing to be other than single token changes. See [54,74,99] for further discussion of generalized Petri nets.

The next two definitions describe structural restrictions on generalized Petri nets.

Definition 16: Two transitions $\sigma \neq \sigma'$ of a generalized Petri net P are call equivalent transitions if and only if, for all $\pi \in \Pi$, $\Delta_I(\pi, \sigma) = \Delta_I(\pi, \sigma')$ and $\Delta_O(\sigma, \pi) = \Delta_O(\sigma', \pi)$.

Definition 17: A generalized Petri net P is called irreflexive if and only if there does not exist any $\pi \in \Pi$ and $\sigma \in \Pi$ such that $\Delta_I(\pi, \sigma) > 0$ and $\Delta_O(\sigma, \pi) > 0$.

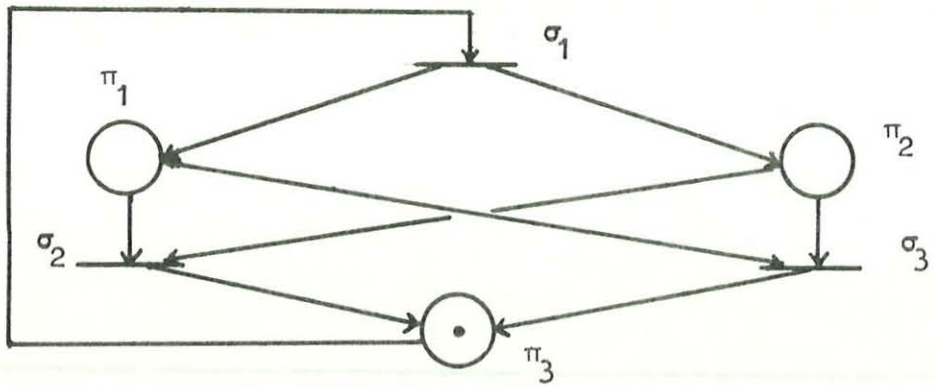
A formal correspondence between producer-consumer systems and generalized Petri nets is depicted below:

Producer-Consumer System S with		Generalized Petri net P with
$B = \{p_1, p_2, \ldots, p_l\}$		$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_l\}$
$S = \{s_1, s_2, \ldots, s_t\}$		$\Pi = \{\pi_1, \pi_2, \ldots, \pi_t\}$
p_j	$\sim$	σ_j
s_i	$\sim$	π_i
$\pi'(s_i, p_j) \neq 0$	$\sim$	$(\pi_i, \sigma_j) \in R$
$\nu'(s_i, p_j) \neq 0$	$\sim$	$(\sigma_j, \pi_i) \in R$
$\mu(s_i)$	$\sim$	$M_o(\pi_i)$
$\pi'(s_i, p_j)$	$\sim$	$\Delta_I(\pi_i, \sigma_j)$
$\nu'(s_i, p_j)$	$\sim$	$\Delta_O(\sigma_j, \pi_i)$

Although this correspondence between producer-consumer systems and generalized Petri nets give an isomorphism between the two models, we will show that it does not automatically provide an isomorphism between behaviours. This is shown by the next sample. Consider the three process producer-consumer system with $\mu(s_1) = \mu(s_2) = 0$ and $\mu(s_3) = 1$:

P1:	P2:	P3:
$P(s_3)$	$P(s_1)$	$P(s_2)$
$P(s_1)$	$P(s_2)$	$P(s_1)$
$V(s_1)$	$V(s_3)$	$V(s_3)$
$V(s_2)$		

This corresponds to the Petri net:



This producer-consumer system has a deadlock. Note that after process p_1 is performed both s_1 and s_2 change to a value of 1. Then p_2 can execute $P(s_1)$ and p_2 can execute $P(s_2)$ which deadlocks the system. No deadlock occurs in the corresponding Petri net, however. Rather, after σ_1 fires then both σ_2 and σ_3 become active. There is a conflict between σ_2 and σ_3, but the global rules for firing transitions do not allow both σ_2 and σ_3 to fire. Thus, the conflict situation in the Petri net is related to the deadlock in the producer-consumer system. More complex examples, like the Cigarette Smokers Problem of Patil show that even a rearrangement of P(s) operations in the processes cannot always circumvent the deadlocking problem. The simultaneous taking of tokens from several places by a transition firing, which prevents the firing of conflicting transitions, is what gives rise to the desire to generalize P operations to operate simultaneously (or in an indivisible manner) on arbitrary subsets of semaphores.

This example should amply demonstrate that one needs to carefully analyze correspondence between models to be sure that the desired properties carry over in the correspondence from one model to the other. Here we see they did not. A weak relationship between conflicts and deadlocks was noted but this has not been precisely described.

In our next lecture we will introduce another formalization called vector addition systems (VAS) which originally arose in the study of parallel program schemata. VAS have been shown equivalent (in some sense) to Petri nets and correspondences with other models also have been developed.

Lecture 2

Schemata Models for Parallel Computation

In this lecture I beiefly describe some of the schemata models and their results. Since vector addition systems, which first arose in parallel program schemata, have been shown to play a role in so many of the different studies of parallelism I will spend considerable time on these, and a related system called vector replacement systems.

A: Schemata Models.

Two basic types of schemata models exist. One is based on having a finite set of operations operating on a common memory, and whose control of the operations is done by some of automata theoretic construct [72,75,90,136]. Thus we have a schema S = (M,A.T) where M is the memory, A is the set of operations and T is the control. The models are usually uninterpreted models or partially uninterpreted models meaning that the particular functions and decisions associated with the operations are not specified.

A second type of schemata model is based upon elementary operation schemas (usually a finite set of them) which are interconnected to form a data-flow schema [37,80,129]. In these, rules of interconnection are often specified in order to insure determinacy of the interconnected schema. That is, we have sufficient conditions for determinacy. In contrast, in the (M,A,T) schemata one develops constraints on the schemata (usually global in nature) from which necessary and sufficiency of determinacy follow.

The more purely automata type models vary anywhere from finite automata forms [17,18] to parallel random access programmed machine in nature. A special iterative form has been studied [87,88] in which some complexity types of results have been obtained.

B: Basic Properties and Proof Techniques.

As we have remarked earlier, determinacy is one of the better understood properties of parallel computation. It takes several different forms in the different models, but in essence it means that the outcome of the computation is unique and does not depend upon the particular relative times that operations are allowed to be performed. The computation graph is by its structure always

determinate, as are some of the data-flow schemata. In terms of schemata one can envision different types of determinacy. It means that for any memory location the complete sequence of values that appear in the location during computation under a given interpretation is independent of how the individual operations were sequenced. Necessary and sufficient conditions are developed for such determinacy and they are shown to be essentially the Bernstein conditiions [15] on overlap on domain and range locations of operations. Also, for a broad class of schemata, namely repetition-free, lossless, persistent, commutative, counter schema it is shown that determinacy is decidable. The technique for showing this is a more-or-less standard sliding argument which is used in Church-Rosser type theorems which allows one to slide symbols of one sequence of operations to match another sequence without changing memory values. Another aspect of the proof involves vector addition systems of which we say more later. A rather surprising aspect of the decidability of determinacy (as well as other properties) is its lack of "stability." It has been shown [94] that if the single property of repetition-free is removed from the hypothesis then determinacy becomes undecidable. This boundary between the decidability and undecidability can be viewed as the most rudimentary measure of complexity, although some of the properties are known to be quite complex [85] even though they are decidable.

Normally, this strong form of determinacy is more than really desired. Often one would only require the final values (assuming termination) of two computation sequences to match on either all, or a specified subset, of memory. The strong determinacy of course implies this weaker "output determinacy" but little is known how to obtain output determinacy without requiring determinacy throughout the sequence.

The determinacy property does not arise directly in terms of Petri nets. This is because the Petri net does not have interpreted functional operations nor does it have a formal way, like interpretations for schemata, of adding them. Thus any such questions must be dealt with outside the Petri net model. The conflict situation in Petri nets does, however, give rise to an obvious situation that looks like it would lead to indeterminacy. Also, it has been shown to be intimately connected with deadlocks — as was shown in the first lecture.

Other properties of interest include: termination, i.e. how many times the operations of the model are performed; boundedness, i.e., the number of operation performances that can be done concurrently; and the number of control

states that are reachable in computations. For schemata all of these properties are decidable in a manner similar to determinacy, and become undecidable without repetition-freeness assumed. For computation graphs rather straightforward algorithms for boundedness and termination can be derived. In Petri nets boundedness is defined in terms of the maximum number of tokens that can reside in any place at any moment. A net is called "safe" if this bound is one. Termination is expressed by the term "liveness" in a Petri net. A transition in a Petri net is called "live" if from any reachable token distribution it is possible to reach a situation in which the transition is fireable. Boundedness and safeness follow directly from the decidability of a problem in vector addition systems whereas liveness is equivalent to the "reachability problem" in vector addition systems.

Since vector addition systems are a simple mathematical construct, and since they underlie many problems concerning parallel computation, I have decided to discuss vector addition systems at some length here.

C: Vector Addition Systems and Vector Replacement Systems

Vector addition systems were originally formulated and studied to prove that certain properties of parallel program schemata were decidable [71,72]. Subsequently they were seen to be an appropriate formalation of previously studied problems [53,55,70,99], and have been widely applied to various problems since then [51,63,113]. Keller [74] generalized VAS to vector replacement systems to extend their modelling capability to other classes of asynchronous systems.

In this section we introduce VAS and VRS as purely mathematical objects, and state some of the known results about these systems. Later we will see how these are applied to problems in parallelism and asynchronism.

We first discuss vector addition systems as found in [72].

Definition 1: An r-dimensional vector addition system is a pair $W = (d,W)$ where d is an r-dimensional vector of nonnegative integers, and W is a finite set of r-dimensional integer vectors.

The reachability set $R(W)$ is the set of all vectors of the form $d+w_1+w_2+\ldots+w_i \geq 0$ for $i = 1,2,\ldots,s$. That is, $R(W)$ is the set of points that can be reached from d by successively adding elements of W such that the path of points so formed always remains in the first orthant.

A simple example : r = 2, d = (1,1), W = {(-2,1),(0,1),(3,-1)}. Note that $(4,1)\in R(W)$ since (4,1) = (1,1) + (3,-1) + (0,1) and the successive points (1,1), (4,0) and (4,1) are all in the first orthant.

We use the following terminology:

(1) For r-dimensional vectors x and y, $x \leq y$ if and only if $x_i \leq y_i$ for i = 1,2,...,r.

(2) We sometimes use 0 to denote the r-dimensional vector of zeroes.

(3) ω is a symbol such that if n is an integer then $n<\omega$ and $n+\omega = \omega$. In some sense ω intutively means "as large as desired."

(4) A rooted tree is a directed graph with some designated node, δ, called the root, which has no edges directed into it, each other node has one edge directed into it, and each vertex can be reached through a directed path from the root. If ζ and η are distinct nodes of the rooted tree having a directed path from ζ to η then we say $\zeta<\eta$. If there is a directed edge from ζ to η then η is called a successor of ζ. If η is a node with no edge directed out of it, then η is called an end.

For W we construct a rooted tree T(W) with labelled nodes $l(\zeta)$ for each node ζ, where $l(\zeta)$ is an r-dimensional vector label having components from $N \cup \{\omega\}$.

Definition 2: T(W) consists of:

(1) a root δ with label $l(\delta) = d$.

(2) let η be a node of T(W)

(a) if for some vertex $\zeta \prec \eta$ $l(\zeta) = l(\eta)$ then η is an end.

(b) otherwise successors of η are formed (one for each $w\in W$ for which $l(\eta) + w \geq 0$). Let η_w denote the successor of η associated with $w\in W$. then $l(\eta_w)$ is determined as follows:

(i) if there is a $\zeta \prec \eta_w$ such that
$l(\zeta) \leq l(\eta) + w$ and
$(l(\zeta))_i < (l(\eta) + w)_i$
then $(l(\eta_w))_i = \omega$

(ii) if no such ζ exists, then $(l(\eta_w))_i = (l(\eta) + w)_i$.

This is a complicated definition which needs some explaining. The recursive form of the definition for T(W) provides a means for recursively constructing T(W) starting with the root with label d. Given any node ζ of T(W) that has not yet been shown to be an end we first construct trial successors to ζ, one for each $w_i \in W$ with temporary label $l(\zeta) + w_i$. If $l(\zeta) + w_i \not\geq 0$ then it

is not a node of T(W), otherwise parts 2b(i) and (ii) of the definition are used to obtain the permanent label for this node, component by component. Having the permanent label one can check to see if the node is an end. The initial portion of the tree T(W) for our example vector addition system is shown in Figure 1.

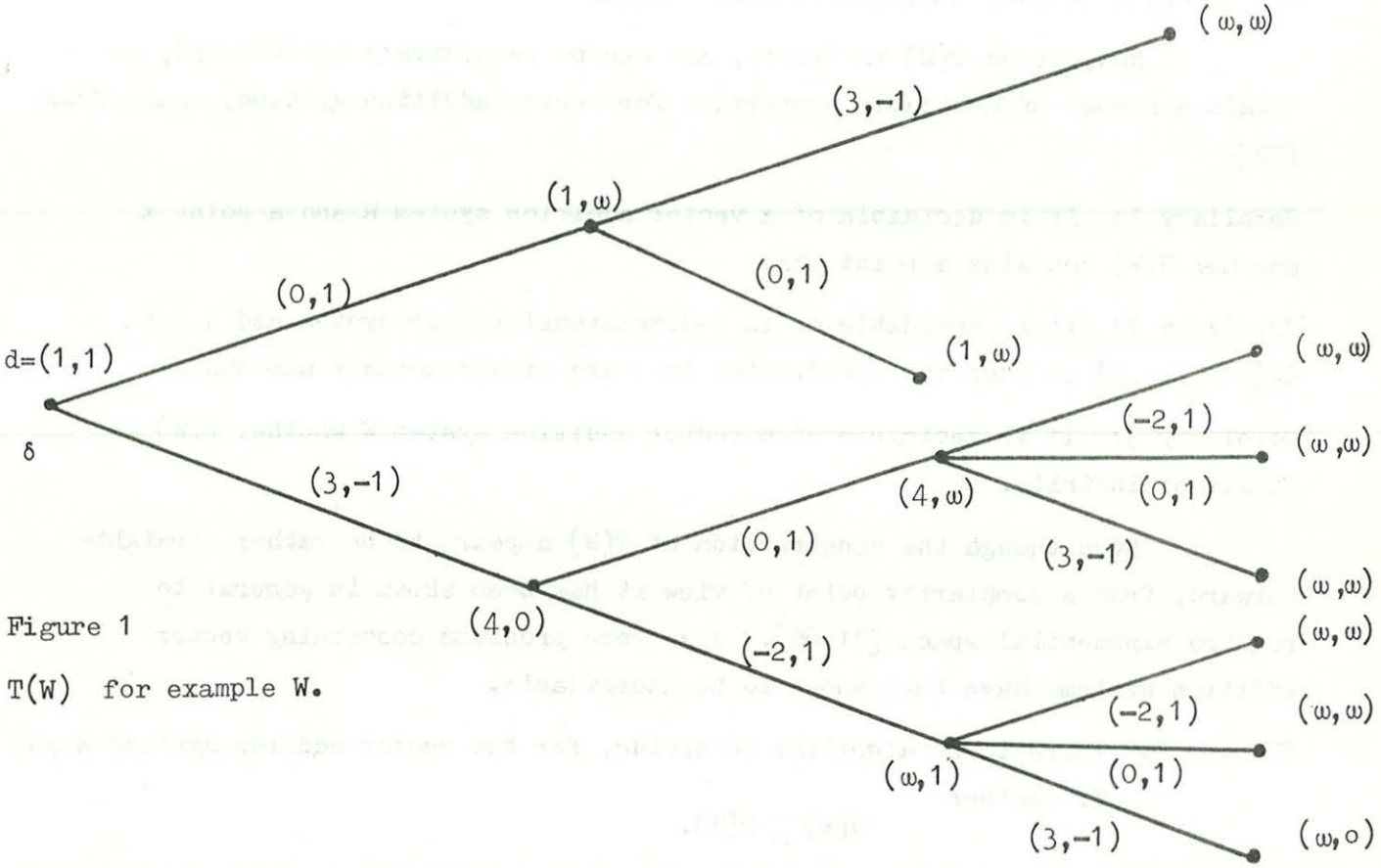


Figure 1

T(W) for example W.

The crucial fact about T(W) that makes it useful is stated in the following theorem.

Theorem 1: For any vector addition system W, T(W) is finite.

This is proved in [72, Theorem 4.1]

Before continuing we note that T(W) provides some information about the reachability set R(W). If T(W) contains a node ζ and $l(\zeta)$ is finite in all components then the path from δ to ζ shows how the vector $l(\zeta)$ can be reached from d by successively adding elements from W such that the path always remains in the first orthant. If some co-ordinates of a node ζ are ω, this in some sense means that by successive application of some subsequence of vectors this co-ordinate value can be made "as large as desired", or can be "pumped". Since several ω's in ζ can interact with each other, care must be taken in such pumping. With careful analysis (see proof in [72, Theorem 4.2]) we obtain the following theorem.

Theorem 2: Let x be an r-dimensional vector of nonnegative integers. Then the following statements are equivlant:

(1) there is a $y \in R(W)$ such that $x \leq y$;
(2) there is a node $\eta \in T(W)$ such that $x \leq l(\eta)$.

Now, since T(W) is finite, and can be recursively constructed, we obtain a number of decidable properties for vector addition systems, again from [72].

Corollary 1: It is decidable of a vector addition system W and a point x whether R(W) contains a point $y \geq x$.

Corollary 2: It is decidable of an r-dimensional vector system and a set $O \subseteq \{1,2,\ldots,r\}$ whether the coordinates in O are simultaneously unbounded.

Corollary 3: It is decidable of a vector addition system W whether R(W) is finite or infinite.

Even though the construction of T(W) appears to be rather straightforward, from a complexity point of view it has been shown in general to require exponential space [21,85]. Also some problems concerning vector addition systems have been shown to be undecidable.

Theorem 3: There is no algorithm to decide, for two vector additon systems W and V, whether

$$R(W) \subseteq R(V).$$

This is an unpublished result of M.O. Rabin. This result and the following by Hack are given in [51].

Theorem 4: There is no algorithm to decide, for two vector addition systems W and V, whether

$$R(W) = R(V)$$

A final problem for vector addition systems that has obtained considerable attention is called the reachability problem. This is the question:

Is there an algorithm to decide, given a vector addition system W and a nonnegative integer vector x, whether $x \in R(W)$.

Sacerdote and Tenny claim to have such an algorithm. Also see [55,107,142].

We now turn to Keller's VRS's [74].

Definition 3: A vector replacement system $V = (d,V)$ consists of:

(i) d, an r-dimensional vector of nonnegative integers, and

(ii) V, a finite set of ordered pairs of r-dimensional integer vectors $V = \{(u_1,v_1),(u_2,v_2),\ldots,(u_p,v_p)\}$ where $u_i \leq 0$ and $u_i \leq v_i$, $i = 1,2,\ldots,p$.

The u_i are called the test vectors and the v_i are called the replacement vectors, and for notational convenience we let $p(v_i) = u_i$. In [74] the components of u_i are allowed to be strictly positive as well as 0 or negative. However, since as Keller notes, only the nonpositive components matter mathematically, we restrict ourselves to u_i entries which are 0 or less.

The reachability set R(v) of a vector replacement system $V = (d,V)$ is the set of all vectors of the form

$d + v^{(1)} + v^{(2)} + \ldots + v^{(s)}$ such that $v^{(j)} \in V$ for all $j \in 1,2,\ldots,s$ and
$d + v^{(1)} + v^{(2)} + \ldots + v^{(i-1)} + p(v^{(i)}) \geq 0$ for all $i = 1,2,\ldots,s$.

Clearly, vector addition systems are simply a special type of vector replacement system for which $(u_i)_j = \min[0,(v_i)_j]$ for all i and j. The concepts of reachability sets for VAS and VRS are also obviously very similar. What Keller could show, in fact, was the notion of the T(W) tree immediately generalized to VRS's where the test $1(\eta) + w \geq 0$ was replaced with a $1(\eta) + u_i \geq 0$ test, but where new node labels were formed by using the replacement vectors as $(1(\eta) + v_i)_j$. Then he showed Theorems 1, 2, and Corollaries 1, 2, and 3 could be generalized to VRS with no essential changes in their proofs. Thus, the finiteness of the tree T(V) and its resulting meaning for R(V) carried over immediately to VRS. Of course, the undecidability results (Theorems 3 and 4) also trivially hold for VRS. Whether the decidability of the reachability problem carries over to VRS is not immediately clear (the algorithm and it's proof are complex) but I suspect that it does.

D: Encoding Parallelism and Asynchronism Problems.

In this section we discuss some of the encodings of parallel and asynchronous problems into vector addition and vector replacement systems. To do so we will find it necessary to briefly describe some of the structures being encoded. This will be done in the briefest way possible. Readers unfamiliar with these structures will undoubtedly find it necessary to obtain necessary details in the cited literature.

The first use of vector addition systems, and indeed their original introduction, was for showing that certain problems about counter schemata (a special form of parallel program schemata) were decidable [72]. A counter schema consists of a set M of memory locations, a finite set A of operations, and a control T. The control contains a finite set of states S, plus a finite number k of counters. Thus, a state of the control is an element of $S \times N^k$. Any given counter schema S is encoded into a vector addition system W_s as follows. W_s has

$$|S| + k + |A|$$

coordinates, where the $|S|$ coordinates represent the S state behaviour. The k coordinates represent counter values and the $|A|$ coordinates represent the μ list lengths of operations in A during computations for the schema. Each of these three encodings into the $|S|$, k, and $|A|$ coordinates are quite typical of VAS encodings. Since $W_s = (d,W)$ to define the VAS we have to say both how d and W are formed. The vector d is formed as follows:

$d(s_o) = 1$,i.e. the coordinate corresponding to the initial state $s_0 \in S$ is set equal to 1.

$d(s) = 0$ for $s \in S$, $s \neq s_0$

$d(i) = \pi_i$ $i = 1,2,\ldots,k$. Here d(i) represents the coordinate of the i^{th} counter and π_i is the initial value of this counter.

$d(a) = 0$ $a \in A$. Here d(a) represents the coordinate of operation a, and initially it has no performances in progress so it is set to 0.

Now, since $\pi_i \geq 0$ for each counter, the vector $d \geq 0$ as required. We will look now at how vectors in W are formed by describing separately how the $|S|$, k, and $|A|$ parts are encoded. We assume that the schema is undergoing a state transition from $s_j \in S$ to $s_i \in S$ due to some event σ which is an initiation or termination of an operation $a \in A$. The $|S|$ part of each $w \in W$ has +1 in one coordinate position (the i^{th}), −1 in one position (the j^{th}), and 0's elsewhere. That is, it is of the form:

0...0 1 0...0 −1 0...0

or 0...0 −1 0...0 1 0...0

This indicates the possibility of a state change from state s_j to s_i. Being in a state s_j will be indicated in arriving at a point in $R(W_s)$ which has +1 in the coordinate associated with s_j and 0's in all other coordinates associated with other elements of S. Thus applying this part of the vector corresponds to changing coordinate s_j from 1 to 0 and coordinate s_i from 0 to 1. The k part of the vector contains in each coordinate the change in value of the counter which is expected to occur in the transition from s_j to s_i under even σ. For

the $|A|$ part of the vector, if σ is an initiation of operation a then +1 is entered into the coordinate for a, with 0's elsewhere. If σ is a termination of operation a then -1 is entered into the coordinate for a and 0's placed elsewhere.

Now, since there are only a finite number of states in S, and a finite number of events σ, we see that the construction forms a finite number of integer valued vectors, so W is as required and $W_s = (d,W)$ is indeed a VAS. It is also quite clear that starting at d and proceeding from reachable point to reachable point in $R(W_s)$ corresponds to a computation for the schema S. Thereby through this encoding we can study properties of computations via reachability questions, in particular using the tree construction $T(W_s)$, in W_s. In [72] this is done for such properties as repetition-freeness, commutativity, boundedness, determinacy and others, where sometimes coordinates are added to W_s to encode and test the property in question.

We illustrate how a "mutual-exclusion" question can be viewed via W_s. Suppose we have designed a counter schema in which operations a and b are used to represent two critical regions which are never to be performed concurrently. We will see that the question of whether the schema actually accomplishes this aim can be decided using W_s without adding any extra coordinates.

We proceed as follows:

Given schema S we form W_s and from that construct the tree $T(W_s)$, which by Theorem 1 is finite. Now, operations a and b can be performed concurrently if and only if at some point in some computation their μ lists are simultaneously greater than 0. But this condition holds is and only if some node label of $T(W_s)$ has both its a and b coordinates greater than zero. This is immediately checkable by inspecting each of the finitely many labels. Thus, this mutual exclusion problem is decidable directly through our encoding and $T(W_s)$ construction. Granted, the algorithm to decide may not be particularly elegant or efficient by this approach, but at least the decidability was obtained through a very direct and simple observation.

We next turn to a "maximum parallelism" question in terms of computation graphs. Computation graphs were introduced in [69] and have been widely studied. In [99] it was shown how to encode computation graphs into VRS. Basically a computation graph is a finite directed graph with nodes $n_1, n_2, \ldots n_l$ and edges $d_1, d_2, \ldots, d_t$, where each node represents an operation, each edge represents a first-in-first-out queue of data from the source to sink node of the edge, and each edge d_p has four control parameters A_p, U_p, W_p and T_p associated with it

with $T_p \geq W_p$. Letting $I(n_i)$ denote the set of indices of edges directed into n_i and $O(n_i)$ the set of indices of edges directed out of n_i, the vector replacement system $V(G) = (d,V)$ associated with a computation graph G is defined as follows. $d = (A_1, A_2, \ldots A_t)$, the vector of initial number of items in each queue. V consists of l pairs of vectors, one for each n_i of G. Letting (u_i, v_i) denote the pair for n_i we have

$$(u_i)_j = \begin{cases} -T_j \text{ if } j \in I(n_i) \\ \\ 0 \text{ otherwise} \end{cases}$$

and

$$(v_i)_j = \begin{cases} -W_j \text{if } j \in I(n_i) \cap \overline{O(n_i)} \\ U_j \text{ if } j \in O(n_i) \cap \overline{I(n_i)} \\ \\ U_j - W_j \text{ if } j \in I(n_i) \cap O(n_i) \\ 0 \text{ otherwise} \end{cases}$$

A vector in R(V(G)) corresponds to a set of mutually attainable queue lengths on the edges of G, and using the T(V(G)) construction one can readily determine for each queue whether it is bounded or not. The problem we are interested in, however, requires considerable addition to V(G) for solution.

In a computation graph a node can "fire" if it has sufficient data on all of its incoming queues (as defined by the T parameters). A computation then corresponds to a sequence of subsets of nodes, where the nodes in each such set are all envisaged as firing concurrently. Maximum parallelism for a computation graph thus corresponds to the maximum size subset of nodes that can occur in any computation for that graph. We wish to encode this maximum parallelism question into VRS form. To do so the V(G) construction requires substantial additions. In V(G) each (u_i, v_i) pair corresponded to a single node firing. Since we wish to model simultaneous node firings we add to this a (u,v) pair for each subset of nodes, and let the entries indicate the overall affect on the queue lengths of this simultaneous firing. (This is the vector sums of the single node (u_i, v_i) vectors for nodes in the subset). Now in addition we add coordinates to the modified V(G). We add a single "control" coordinate, plus one coordinate for each (u,v) pair. In the control coordinate both the u value and the v value are set to -1. The value in the (u,v) coordinate is set to k, where k is the

cardinality of the subset of nodes represented by the (u,v) vector pair. All other extra coordinate values are set to 0 to complete this part of the construction. To complete the construction we add a mate (u',v') vector pair to V for each (u,v) already constructed. Both u' and v' have a -k placed in the (u,v) coordinate; v' has a + 1 in the control coordinate, and both u' and v' are zero elswhere. The initial vector is that of V(G) with a 1 added in the control coordinate and 0's elsewhere. This completes the construction. We now have a VRS for G that we denote by V'(G). Applying any particular (u,v) pair (not the (u',v') mates) corresponds to the simultaneous firing of the nodes in the subset represented by (u,v). The control coordinate insures that after a (u,v) pair is applied then a (u',v') pair must be applied before another (u,v) pair is applied. The +k and -k entries insure that the (u',v') pair applied must be the mate of the (u,v) pair just previously applied. A reachable path in V'(G) now corresponds directly to a firing sequence in the computation graph. We next construct the tree T(V'(G)). Inspection of the node labels in the (u,v) coordinates with nonzero value determine exactly which subsets of nodes can fire simultaneously, and maximizing over this value gives the attainable maximum parallelism of G.

To solve this problem, then, we have introduced several more encoding tricks. The (u,v) pairs provided the desired values to measure the size of the sets, and the extra (u,v) pairs generalized single node firing to multiple node firing. Again, however, we make no claims for the efficiency of this decision procedure.

We now turn to the relationship between generalized Petri nets and vector addition systems.

Suppose P is an irreflexive generalized Petri net without equivalent transitions, where $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ and $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_t\}$. A system $W(P) = (d, W)$ corresponding to P is defined as follows

(1) d is an n-coordinate nonnegative integer vector:

$$d = (M_o(\pi_1), M_o(\pi_2), \ldots, M_o(\pi_n)).$$

We also use M_o to represent this marking vector.

(2) W is a set of t vectors, one for each transition of P. Let w_j denote the vector for transition σ_j and $(w_j)_k$ the k^{th} coordinate value of w_j, then define

$$(w_j)_k = \Delta_O(\sigma_j, \pi_k) - \Delta_I(\pi_k, \sigma_j).$$

It should be clear that W(P) is a vector addition system in which any reachable path of W(P) corresponds to a firing sequence of P. Without going into detail (see [99]) it should be clear that for any vector addition system one can also construct a corresponding irreflexive generalized Petri net without equivalent transitions. Thus, there is an isomorphism between these two structures, giving the result:

Theorem 5: There is an isomorphism between irreflexive generalized Petri nets without equivalent transitions and vector addition systems which provide an isomorphism between firing sequences and reachable paths.

The reader may wish to provide the details for these constructions and results which have been omitted here. Note that the irreflexive and equivalent transition restrictions are important to have the simple isomorphism results. If a Petri net had equivalent transitions σ_i and σ_j then the construction of W(P) would give the same vector for w_i and w_j. Since W is a set the information about equivalent transitions is lost in the mapping from the generalized Petri net P to W(P). Thus there would not longer be an isomorphism between firing sequences and reachable paths. The irreflexive property of P means that in transforming a vector addition system to a generalized Petri net that a nonzero entry $(w_j)_k$ in $w_j \in W$ immediately indicates the interconnection of place π_k with transitions a_j. If $(w_j)_k = 0$ there is no direct connection. If $(w_j)_k = a > 0$ then π_k is in the output set of places for σ_j and has $\Delta_O(\sigma_j, \pi_k) = a$. If $(w_j)_k = a < 0$ then π_k is in the input set of σ_j and has $\Delta_I(\pi_k, \sigma_j) = a$. Irreflexitivity insures that no confusion can exist from the general relation $\Delta_O(\sigma_j, \pi_k) - \Delta_I(\pi_k, \sigma_j)$.

From Theorem 5 relating reachable points in W(P) and reachable markings in P we immediately obtain:

Corollary 4: For any irreflexive generalized Petri net without equivalent transitions $R(W(P)) = R(P, M_o)$.

Thus many properties about generalized Petri nets can be studied via the corresponding vector addition system. For example, the coordinate values of reachable points in R(W(P)) determine safeness and k-safeness.

For the remainder of this lecture when we use P for a Petri net we will mean an irreflexive generalized Petri net without equivalent transitions. A simple restatement of safeness now is:

Corollary 5: P is safe if and only if each reachable point in R(W(P)) has coordinate values that lie in the set {0,1}, and P is k-safe if and only if the coordinate values lie in the set {0,1,...,k}.

Corollary 6: The properties safe and k-safe for P are decidable.

Proof: Inspect nodes of T(W(P)). For safeness labels on the tree must have coordinate values only from {0,1}, and for k-safeness from {0,1,...,k}. Naturally all of T(W(P)) may not have to be constructed to prove that a given P is not safe or not k-safe.

A much less immediate corollary, which was shown by Hack [55] through a complex series of constructions using Petri nets, is:

Corollary 7: The questions of liveness of a Petri net P and of whether $x \in R(W(P))$ in a vector addition system are recursively equivalent.

The corollaries stated for vector addition systems — using the T(W) tree — are also directly translated into results for Petri nets. Namely, for any marking M it is decidable whether there is an $M' \geq M$ in $R(P,M_o)$. It is decidable, for any subset of places, whether markings can be reached where the number of tokens in these places are simultaneously unbounded. It is decidable whether $R(P,M_o)$ is finite or infinite.

Consider now the property of whether a given transition σ is dead with respect to a particular marking M. A simple modification of W(P) allows one to decide this. Construct $W'(P) = (M',W')$ exactly like W(P) but add one extra coordinate to represent the firing of σ. Let M' be the initial marking which is equal to M, and with 0 the extra coordinate value. Now the $w \in W'$ representing σ let the extra coordinate value equal one, and for all other $w \in W'$ that coordinate value is set equal to zero. Now σ is dead with respect to M if and only if there is no $p \in W'(P))$ with a value in the extra coordinate greater than zero. This can be tested by inspection of T(W'(P)). This technique of adding coordinates to count or test certain properties is useful for testing other properties as well.

The generalization of vector addition systems made by Keller [74] to vector replacement systems allows one to develop a correspondence between vector replacement systems and generalized Petri nets without equivalent transitions giving analogous results to those we have just discussed, see Keller [74] and Miller [99] for details.

E: Conclusions

From all of the different uses for vector addition systems we have discussed, as well as from the inherently simple combinatorial structure of VAS it should be clear that vector addition systems form a basic mathematical idealization that is both useful and elegant for considering problems of parallelism and synchronization. I suspect that numerous other applications of VAS will continue to be discovered.

Lecture 3

On Formulations Relating to Loosely Connected Processes

In attempting to understand and design complex systems programs an extensive literature has arisen concerning such systems. A part of this work deals with resource sharing among loosely connected processes, where the system is thought of being composed of a number of semi-independent processes that run, more-or-less simultaneously, but that have to interact in a cooperative manner when using resources that are available to the various processes. This literature on system synchronisation includes three principal types of work, namely:

1) Synchronisation primitives
2) Programming solutions to particular problems
3) Mathematical formulations, e.g., using Petri nets, path expressions, graph models, systems of processes models, etc.

In this talk I will discuss, primarily, some relatively new results on the system of processes approach of these mathematical formulations. This work is in recent papers by Miller and Yap, but also has close ties to the work of Lipton and also Gilbert and Chandler. Some of the Petri net and other parallelism model applications have already been alluded to in the previous talks. The path expression approach as described in works of Campbell, Habermann and Lauer, I will not discuss here. Much of this work was done right here at Newcastle and people from here would be much more capable of discussing this than I.

After some informal descriptions of synchronisation problems I will describe two mathematical formulations for such problems: a system of process model and a synchronisation graph model. Using these formulations I will then show how various properties of synchronisation can be formalised, and also discuss a few theorems:

A. Typical Synchronisation Problems

There have been quite a few different synchronisation problems discussed in the literature. They have been given names such as: the mutual exclusion problem, the producer-consumer problem, the readers-writers problem, the dining philosophers problem, the cigarette smokers problem, etc. Each of these problems depicts a certain type of interaction between a number of concurrent processes and the utilisation of common resources. We will use mutual exclusion and the dining philosophers problems as examples in our formulations.

Mutual exclusion can be described as having n processes that access a common resource (say a file). To insure integrity of the file we restrict utilisation in such a way that at most one process can have access to the file at any time. Beyond that one wants the system to be designed so that all processes are treated "fairly" in being able to get access. No process should be locked out, nor should the system deadlock. In our formal treatment these requirements will be formally stated.

The dining philosophers problem considers n processes arranged in a circle with a resource between each process around the circle. For a process to enter a certain critical region it must gain access to two resources - one to its left and one to its right. Again, only one process can access a given resource at a time, and the problem is to design a system that allows such access, and again gives "fair" treatment to each process. Figures 1 and 2 depict mutual exclusion and dining philosophers, respectively.

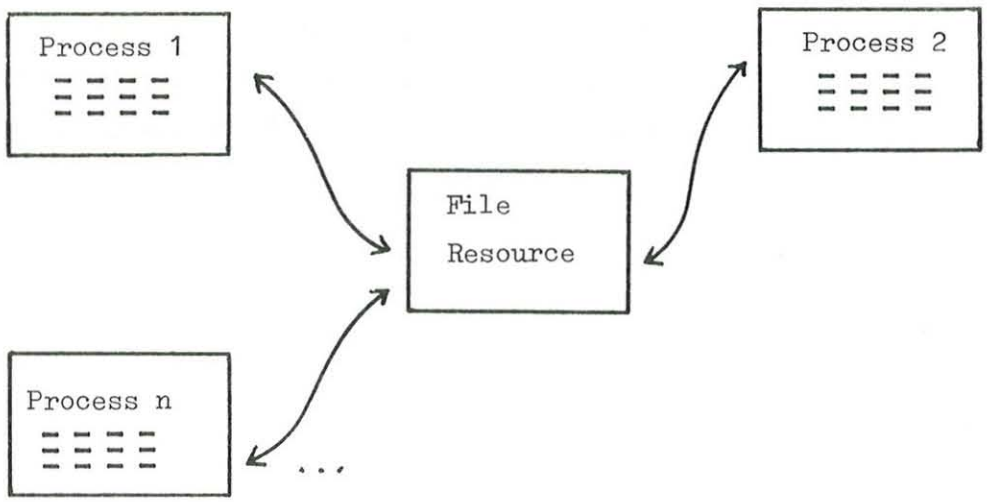


Figure 1 : Mutual Exclusion

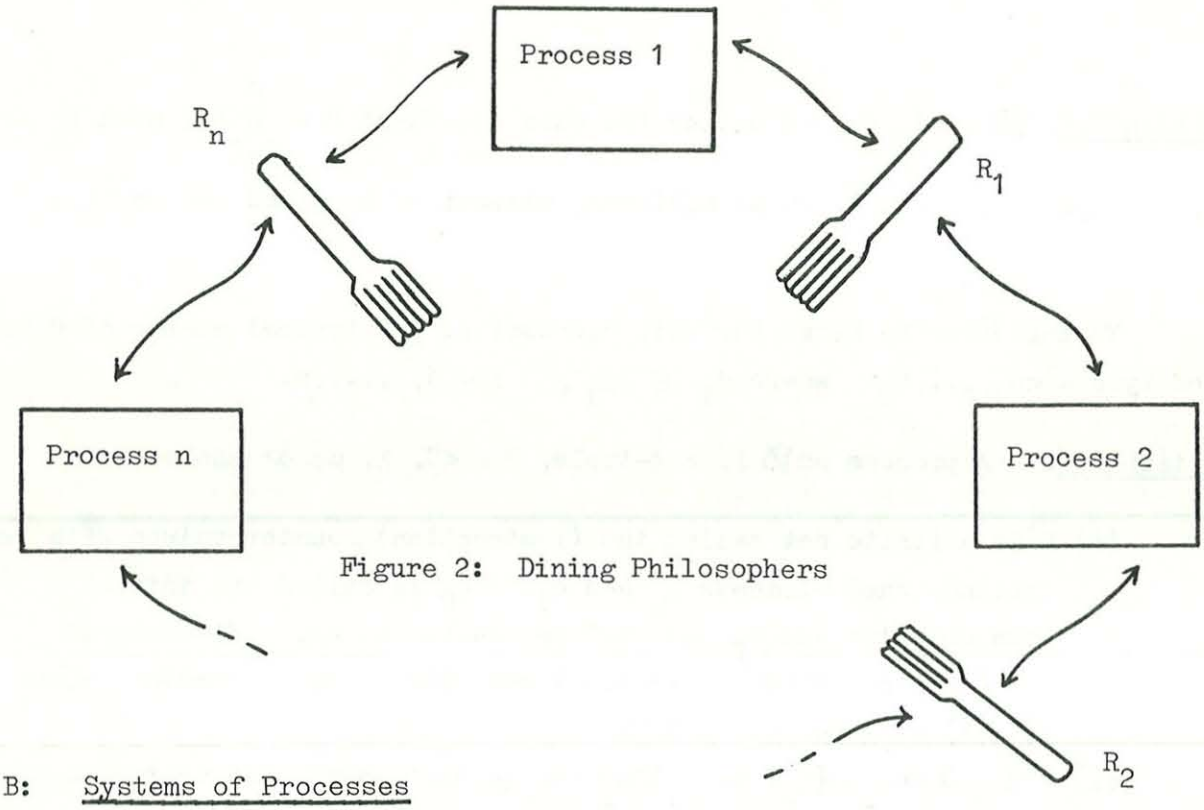


Figure 2: Dining Philosophers

B: Systems of Processes

We now define our system of processes model. In our formulations, we will see that resources are not directly formalised. This is an omission and not an oversight. We want to model the fact that synchronisation in processes is a function of the common data variables. It is these variables, rather than the resources themselves, that are visible to the processes. Even though these data variables are not the resources themselves, they are intended to reflect the state of the resources as seen by the processes (e.g. whether a resource is available).

A "process", as formulated here, consists of a finite set of instructions (i.e. the program) which begins its computation at a given initial instruction, with some initial data values. The process sequentially executes instructions, where each instruction determines two things: it computes new data values and it specifies the next instruction to be executed. We also include the concept of a process failing. This is done by specifying a failure function which determines how the data values are changed in the event of failure.

Definition 2.1. $\mathcal{D} = \langle \bar{d}_0, D \rangle$ is called the data set where $D = \overset{m}{\underset{i=1}{X}} D_i$, each D_i is a set, $i = 1,2,\ldots,m$, and $\bar{d}_0$ is an arbitrary element of D called the initial data.

We take $\mathcal{D}$ to be fixed for this discussion. A typical member of D is denoted by $\bar{d} = \langle d_1, \ldots, d_m \rangle$ where $d_i \in D_i$, $i = 1,2,\ldots,m$.

Definition 2.2. A process on $\mathcal{D}$ is a 4-tuple, $P = \langle C, \lambda, \mu, \phi \rangle$ where:

(i) C is a finite set called the (instruction) counter values with two distinguished elements c_0 and c_f. c_0 is called the initial counter value and c_f the failure counter value. Elements of $C - \{c_f\}$ is called the (normal) next instruction function, where $\lambda(c_f;\bar{d}) = c_0$ for all $\bar{d} \in D$.

(ii) $\lambda: C \times D \longrightarrow C - \{c_f\}$ is called the (normal) next instruction function, where $\lambda(c_f;\bar{d}) = c_0$ for all $\bar{d} \in D$.

(iii) $\mu: C \times D \longrightarrow D$ is called the failure transformation function.

Each instruction of the program is represented by a counter value of C. We will often refer to members of C as instructions, even though this is not strictly accurate. Beginning at the initial instruction, c_0, and with initial data, $\bar{d}_0$, the process progresses by using the λ function to specify the (normal) next instruction to be executed, and the μ function to specify the new data values. Note that the range of λ excludes c_f so that, normally, the failure counter value is not entered. Later we will show how a transition into the failure value is accomplished.

We are now ready to define how a collection of semi-independent processes act cooperatively through a common data set $\mathcal{D}$.

Definition 2.3. A system of n processes on $\mathcal{D}$ is a set $\Sigma = \{P^i\}_{i=1}^n$ where each $P^i = \langle C^i, \lambda^i, \mu^i, \phi^i \rangle$, $i = 1,2,\ldots,n$, is a process on $\mathcal{D}$, and $C^i \cap C^j = \emptyset$ for $i \neq j$.

We call P^i the i^{th} process. We will use superscripts to denote the process being referred to. For example, c_0^i and c_f^i are the initial and failure counter values of the i^{th} process. We often suppress reference to $\mathcal{D}$ and n when understood. Hence "process" and "system of processes" (or just "system") are usually used. Note that the definitions of process and system of processes imply that the only communication between processes occur through $\mathcal{D}$. Also, although $\mathcal{D}$ could include all the variables of interest in the computationa, control, and interaction aspects of the processes, it is often convenient to consider $\mathcal{D}$ to be only that part of the data used for process control and interaction. No process may modify or read another process's counter value. In particular, the use or intent to use a common resource by one process can only be indicated to other processes by some conventions on $\mathcal{D}$ values.

We next consider how system actions may be defined:

Definition 2.4. An instantaneous description (i.d.) of Σ is an (n+m)-tuple, $I = \langle c^1, \ldots, c^n, d_1, \ldots, d_m \rangle$ where $c^i \in C^i$, $i = 1, 2, \ldots, n$, and $\langle d_1, \ldots, d_m \rangle \in D$. The initial i.d. of Σ is $I_0 = \bar{c}_0; \bar{d}_0$ where $\bar{c}_0 = \langle c_0^1, c_0^2, \ldots, c_0^n \rangle$, and $\bar{d}_0$ is the initial data value.

Definition 2.5. Let $I = \bar{c}; \bar{d}$, $I' = \bar{c}'; \bar{d}'$ be i.d.'s of Σ, $i \in [n]$. The binary relation "$\xrightarrow[i,\Sigma]{}$" is said to hold between I and I', written

$I \xrightarrow[i,\Sigma]{} I'$, iff (i) $\Pi_j(\bar{c}') = \Pi_j(\bar{c})$, for $j = 1, \ldots, n$, $j \neq i$,

and (ii) either $\Pi_i(\bar{c}') = c_f^i$ and $\bar{d}' = \phi^i(\bar{d})$

or $\Pi_i(\bar{c}') = \lambda^i(\Pi_i(\bar{c}); \bar{d})$ and $\bar{d}' = \mu^i(\Pi_i(\bar{c}); \bar{d})$.

We write $I \xrightarrow[\Sigma]{} I'$ iff $\exists\, i \in [n]$ such that $I \xrightarrow[i,\Sigma]{} I'$.

Definition 2.5 specifies how transitions from one i.d. into another may be effected. The either-or clause of (ii) indicates that a process may either fail (via ϕ) or take a normal transition (via λ and μ). We say that $I \xrightarrow[i,\Sigma]{} I'$ is a failure transition or normal transition according to which of the either-or clauses of (ii) is applicable. We also say that process i causes the transition $I \xrightarrow[i,\Sigma]{} I'$. As usual, references to Σ are omitted when convenient. The relations $\xrightarrow[i]{}^*$ and $\xrightarrow{}^*$ are the reflexive transitive closure of $\xrightarrow[i]{}$ and $\rightarrow$, respectively. If $I_0 \xrightarrow{}^* I_1$, where I_0 is the initial i.d., we say that I_1 is reachable.

Definition 2.6. A sequence of i.d.'s, $\bar{\mathcal{J}} = (I_1, I_2, \ldots)$ is called a transition sequence iff $i \geq 1$, $I_i \rightarrow I_{i+1}$. A transition sequence has the finite delay property for $J \subseteq [n]$ iff one of the following 2 conditions holds:

(i) $\bar{\mathcal{J}}$ is finite

(ii) $\forall j \in J$, either for infinitely many k's, $\Pi_j(I_k) = c_f^j$ or for infinitely many k's, $I_k \xrightarrow{j} I_{k+1}$ is a normal transition caused by process j.

Definition 2.7. A sequence of i.d.'s $\bar{\mathcal{J}} = (I_1, I_2, \ldots)$ is called a computation sequence on Σ^J, where $J \subseteq [n]$, iff

(i) $I_0 \xrightarrow{*} I_1$ i.e. I_1 is reachable.

(ii) $\forall i \in \{1,1,\ldots\}$ $I_i \longrightarrow I_{i+1}$ is a transition caused by some process j where $j \in J$.

(iii) $\bar{\mathcal{J}}$ has the finite delay property for J.

$\bar{\mathcal{J}}$ is called a nonfailing computation sequence on Σ^J is in addition to (i) - (iii), it satisfies

(iv) $\forall k \in \{1,2,\ldots\}$ $I_k \longrightarrow I_{k+1}$ is a normal transition.

Notation: If $J = [n]$, we say "computation sequence on Σ" or simply, "computation sequence," instead of "computation sequence on $\Sigma^{[n]}$". If $J = \{j\}$, then "$\Sigma^{\{j\}}$" is replaced by "Σ^j".

A computation sequence is thus seen to be a sequence of consecutive i.d.'s which occurs in some computation of the system Σ. The finite delay property for J implies that unless a process in J is failed, it must execute instructions at finite intervals in the computation sequence. That is, no nonfailing process in J can be "infinitely slower" than the other processes of the system. Note that the processes are independent in the sense that any process $i \in [n]$ may act at any time, that is, from any i.d. each process can cause either a normal or a failing transition.

C: Properties of Processes.

We now turn to stating certain properties on systems of processes which seem to be common properties in many of the synchronisation problems. Throughout this section, we assume Σ to be the system of n processes on $\mathcal{D}$, as already introduced.

Property P1: Instruction executability

$$\forall i \in [n],\ \forall c \in C^i,\ \exists \text{an i.d. } I \text{ which is reachable and } \Pi_i(I) = c.$$

This property simply states that we need only restrict our attention to those instructions that may be executed. Notice that the i.d. I may be reachable only via some process failing, for example, if $c = c^i_f$.

Property P2: Critical Region

$$\forall i \in [n],\ \underline{cr}_i \in C^i - \{c^i_f,\ c^i_0\}$$

This says that each process in Σ has a critical region in which the use of some resource is required. We have not yet specified how the resource is to be used. That will depend upon the nature of the particular synchronization problem.

Property P3: Trying Region

$$\forall i \in [n],\ T^i \subseteq C^i - \{\underline{cr}_i,\ c^i_f\} \text{ such that}$$

(i) $\forall c \in C^i,\ \forall \bar{d} \in D,\ \lambda^i(c;\bar{d}) = \underline{cr}_i \Rightarrow c \in T^i$

and (ii) $\forall c \in T^i,\ \forall \bar{d} \in D,\ \lambda^i(c;\bar{d}) = c' \Rightarrow c' \in T^i \cup \{\underline{cr}_i\}$.

This condition states that there is a subset of instructions T^i which precedes the critical region in the sense that before entering the critical region, the process has to execute instructions from T^i (condition (i)). T^i also has the property that the only normal (i.e. nonfailing) exit from T^i is through the critical region (condition (ii)). Combining P3 and P1 we see that there is a $c \in T^i$ and a $\bar{d} \in D$ such that $\lambda(c;\bar{d}) = \underline{cr}_i$. Thus the trying region (as T^i is called) is seen as a protocol that processes have to go through in order to be synchronised properly for entry into their critical region.

Property P4: Loose Connectedness

$\forall i \in [n]$ if $\bar{\mathcal{J}} = (I_1, \ldots, I_{k-1}, I_k)$ is a computation sequence of Σ for which $\Pi_i(I_j) \notin T^i \cup \{\underline{cr}_i\}$ for $j = 1, 2, \ldots, k-1$ and $\Pi_i(I_k) \in T^i$, and process i is nonfailing in $\bar{\mathcal{J}}$, then there exists a nonfailing computation sequence of Σ^i, $\bar{\mathcal{J}}' = (I'_1, \ldots, I'_{j-1}, I'_j)$ such that $I'_1 = I_1$, and $\Pi_i(I'_j) \in T^i$.

We view loose connectedness as the ability of a process, when outside of its trying region or critical region, to proceed independently to its trying region. Note that the existence of $\bar{\mathcal{J}}'$ is predicated upon the existence of $\bar{\mathcal{J}}$.

Property P5: Failure Indication

Let $\bar{c};\bar{d}$ and $\bar{c}';\bar{d}'$ be two reachable i.d 's and $i \in [n]$.
If $\Pi_i(\bar{c}) = c_f^i \neq \Pi_i(\bar{c}')$, then $\bar{d} \neq \bar{d}'$.

This property says that when a process is failed, this fact is indicated by the data values.

The next two properties specify the flow of instructions between the various regions of a process. Notice that they are conditions on single processes rather than cooperating processes in a system.

Property P6: Trying Region Reachability

$\forall i \in [n]$, if I_1 is any reachable i.d. of Σ where $\Pi_i(I_1) = c_0^i$, then there exists a nonfailing computation sequence $(I_1, I_2, \ldots, I_k)$ of Σ^i such that $\Pi_i(I_k) \in T^i$.

Property P7: Cyclic Processes

$\forall i \in [n]$, $\forall$i.d. I_1, where $\forall j \neq i$, $\Pi_j(I_1) \notin T^j \cup \{\underline{cr}_j\}$ then there exists a nonfailing computation sequence $(I_1, I_2, \ldots, I_k)$ of Σ^i such that $\Pi_i(I_k) = c_0^i$.

Property P6 says that process i (acting alone in Σ^i) can reach the trying region from its initial counter value via a nonfailing computation, independent of the data values. Property P7 says that process i (again acting alone in Σ^i) may always return to its initial instruction provided the other

processes are not trying or in their critical region, hence cyclic. So we see that the typical cycle of a cyclic process consists of going from c_0^i to T^i (by P6), from T^i to $\underline{cr}_i$ (by P3) and from $\underline{cr}_i$ back to c_0^i (by P7,) all in a nonfailing manner. Also, by the definition of λ^i, $\lambda^i(c_f;\overline{d}) = c_0^i$, so a failed process is restarted at c_0^i.

Property P8: Critical Region Reachability

$\forall i \in [n]$, a nonfailing computation sequence $(I_0,\ldots,I_k)$ such that $\Pi_i(I_k) = \underline{cr}_i$.

This property simply states that from the initial i.d. we should be able to reach each critical region in a nonfailing way. Of course, to do so may require processes other than process i to execute their instructions, possibly going through their own critical regions before process i reaches its critical region.

Property P9: Non-Trying Region

$\forall i \in [n]$ and $\forall \overline{d}$ $\lambda^i(\underline{cr}_i, \overline{d}) \notin T^i$.

This property states that each process upon leaving its critical region enters a region not in the trying region. This has often been termed "the rest of the program." It is useful to include this property. It allows one to test that the sequencing protocol is such that some process j which is trying to enter its critical region is not indefinitely delayed by some process i which is in a non-trying region.

D: Synchronisation Graphs.

The system of processes formulation seems rather straightforward (modulo a number of definitional decisions). It is intended to capture the basic notions of the synchronisation problem literature, and is somewhat related to a formulation of Lipton. Now we introduce a formalism we call synchronisation graphs. It provides a formal analysis technique which is close in spirit to that of transition graphs of Moore and Mealy for finite state machines. Gilbert and Chandler developed a similar graphical model for analysing synchronisation, but our use of synchronisation graphs differs from the results in their paper.

We first need some graph theoretic terminology.

Definition 3.1. A directed graph with multiloops, $G = \langle V, E, \gamma \rangle$ is a triple such that V and E are sets (of vertices and edges, respectively) where $\gamma: E \to V \times V$ such that $\forall e, f \in E$, $\gamma(e) = \gamma(f) = \langle u, v \rangle$ and $e \neq f \Rightarrow u = v$.

If $\gamma(e) = \langle u, v \rangle$, then the edge e is directed from u to v. If $u \neq v$, then there is at most one $e \in E$ such that $\gamma(e) = \langle u, v \rangle$. But if $u = v$, then more than one edge, say e and f, may exist such that $\gamma(e) = \gamma(f) = \langle v, v \rangle$. Thus, a directed graph with multiloops is a special case of directed multigraphs in which the only multiple edges are self-loops.

Definition 3.2. The outdegree of v, $v \in V$, is the cardinality of the set $\{e \in E \mid \exists u$ such that $\gamma(e) = \langle v, u \rangle\}$. The outdegree of G is the maximum over all outdegrees of $v \in V$.

We are now ready to define a synchronisation graph. Let $\mathcal{C} = \langle \bar{c}_0, C \rangle$, where $C = \overset{n}{\underset{i=1}{X}} C^i$, each C^i is a finite set with $\{c_0^i, c_f^i\} \subseteq C^i$, $c_0^i \neq c_f^i$, $C^i \cap C^j = \emptyset$ for $i \neq j$, and $\bar{c}_0 = \langle c_0^1, c_0^2, \ldots, c_0^n \rangle$.

Definition 3.3. A synchronisation graph on $\langle \mathcal{C}, \mathcal{D} \rangle$ is a triple $S = \langle G, \alpha, \beta \rangle$ such that

(i) $G = \langle V, E, \gamma \rangle$ is a directed graph with multiloops and each $v \in V$ has outdegree 2n.

(ii) $\beta: E \to \{0, 1\} \times [n]$ such that $\forall e, f \in E$, $\gamma(e) = \langle u, v \rangle$, $\gamma(f) = \langle u, w \rangle$ and $e \neq f \Rightarrow \beta(e) \neq \beta(f)$.

(iii) $\alpha: V \to C \times D$ such that

(a) α is an injection.

(b) $\exists v_0 \in V$, called the initial vertex and $\alpha(v_0) = I_0$ where $I_0 = \bar{c}_0; \bar{d}_0$.

(c) $\forall e \in E$, $\beta(e) = \langle b, i \rangle$ and $\gamma(e) = \langle u, v \rangle \Rightarrow \Pi_j(\alpha(u)) = \Pi_j(\alpha(v))$ for all $j \neq i$, $j \in [n]$. Furthermore, $b = 0 \Rightarrow \Pi_i(\alpha(v)) = c_f^i$.

We call α the vertex label function and β the edge label function of S. The motivation for synchronisation graphs is that, given any system of processes Σ, we can define a synchronisation graph S such that each vertex of S (i.e. vertex

of G where $S = \langle G, \alpha, \beta \rangle$) represents an i.d. of Σ where $\alpha(v)$ is the i.d. represented by vertex v. A directed edge of S, $e \in E$ such that $\gamma(e) = \langle u, v \rangle$, then represents a transition of Σ, $\alpha(u) \longrightarrow \alpha(v)$. If $\beta(e) = \langle b, i \rangle$, then $\alpha(u) \longrightarrow \alpha(v)$ is caused by process i, where b = 0 indicates a failure transition and b = 1 indicates a normal transition. Thus, the edge label function tells us which process caused the transition and whether the transition is a normal transition or a failure transition.

Definition 3.4. Let $J \subseteq [n]$. A sequence of edges, $\bar{p} = (e_1, e_2, \ldots)$ is a path of S^J iff $\Pi_2(\gamma(e_k)) = \Pi_1(\gamma(e_{k+1}))$ and $\Pi_2(\beta(e_k)) \in J$ for $k = 1, 2, \ldots$. If $J = [n]$, then we say "path of S" in place of "path of $S^{[n]}$". Similarly, if $J = \{i\}$, "path of S^i" will do.

We adopt various notations for paths. So $e_1 \xrightarrow{*} e_k$ denotes that there is a path beginning at e_1 and ending at e_k. Note that we can uniquely determine the sequence of vertices on a given path, but a sequence of vertices does not uniquely determine a path (because of self-loops). When both the vertices and edges of a path $\bar{p} = (e_1, e_2, \ldots)$ are of interest, we write $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_k} v_{k+1} \longrightarrow \ldots$ where $(v_1, v_2, \ldots)$ is the sequence of vertices uniquely defined by $\bar{p}$. Similarly we write $\bar{p} = v_1 \longrightarrow v_2 \ldots$ when only the vertices of $\bar{p}$ are of interest. We say a vertex v_{k+1} is reachable iff there is a path $v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots \xrightarrow{e_k} v_{k+1}$ starting from the initial vertex. Note that this coincides with our definition of a reachable i.d. in Section 2.1 in the sense that v_1 is reachable implies that $\alpha(v_1)$ is reachable (as an i.d.). From now on, we restrict our attention to only reachable vertices so that the terminology "$\forall v \in V$" should be read "$\forall v \in V$ and v is reachable". Hopefully this will cause no confusion, and for emphasis we sometimes still say "reachable".

Definition 3.5. Let $J \subseteq [n]$ and $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_k} v_{k+1} \xrightarrow{e_{k+1}} \ldots$ be a path of S^J with v_1 reachable. Then $\bar{p}$ is called a computation path of S^J iff one of the following two conditions holds:

(i) $\bar{p}$ if finite.

(ii) $\forall i \in J$, either $\Pi_i(\alpha(v_k)) = c_f^i$ for infinitely many k's or $\beta(e_k) = \langle 1, i \rangle$ for infinitely many k's.

Again, we say "computation path of S" or "computation path of S^i" when $J = [n]$ or $J = \{i\}$, respectively.

Definition 3.6. A computation path of S^J, $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots$ is called nonfailing iff $\forall k \in \{1, 2, \ldots, \}$ $(\beta(e_k)) = 1$.

The following theorem shows how the system of processes and synchronisation graphs are related formally:

Theorem 3.1. Let Σ be a system of n processes on $\mathcal{D}$ such that $C = \overset{n}{\underset{i=1}{X}} C^i$ where C^i are the counter values for the i^{th} process and $\mathcal{G} = \langle \bar{c}_0, C \rangle$. Then there eixsts an effectively constructed canonical synchronisation graph, $S = \langle G, \alpha, \beta \rangle$ on $\langle \mathcal{G}, \mathcal{D} \rangle$ satisfying the following:

(i) There is a bijection between the reachable i.d.'s of Σ and the vertices of S, as given by $\alpha: V \rightarrow C \times D$. Also, $\alpha(v_0) = \bar{c}_0; \bar{d}_0$ where v_0 is the initial vertex.

(ii) Each $v \in V$ has outdegree 2n.

(iii) There is a bijection between computation sequences of Σ, $\bar{\mathcal{I}} = (I_1, I_2, \ldots)$, and computation paths of S, $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots$, such that $\forall k = 1, 2, \ldots, \alpha(v_k) = I_k$.

(iv) $\forall e \in E$, $\beta(e) = \langle b, i \rangle$ and $\gamma(e) = \langle u, v \rangle$ implies that the transition $\alpha(u) \rightarrow \alpha(v)$ of Σ is caused by process i and the transition is a failure or a normal transition according to whether b = 0 or 1, respectively.

Following Theorem 3.1 we associate with each system of processes Σ its canonical synchronisation graph $S(\Sigma)$. We also write $\alpha(S) = \{\alpha(v) \mid v \in V\}$, and it is easy to see that $\alpha(S(\Sigma))$ is the set of all reachable i.d.'s of Σ.

The synchronisation graphs are built on the idea of the "global states" of the systems of processes, and this notion is not new. Clearly, a synchronisation graph for a particular synchronisation problem could have a very large number, or even an infinity, of vertices, and this makes it impractical as a detailed analysis tool for the problem. The novelty of the formulation, however, seems to be its use to state problem requirements and to prove general theorems like those for data requirements as stated later.

E: The Mutual Exclusion Problem.

We now begin to consider synchronisation problems and formulate Requirements for their solution in terms of synchronisation graphs. We choose the mutual exclusion problem to start our discussion, first, because it is one of the earliest and best known synchronisation problems, and second, because it has been studied extensively. It seems that all the analyses that have appeared in the literature concentrate on particular programming solutions. Our emphasis here, however, is the inherent properties of synchronisation problems, in the sense that all solutions (relative to our model of processes) should satisfy these properties.

Our strategy is as follows: We formalise the informal requirements on solutions to a particular synchronisation problem by placing restrictions on synchronisation graphs. Our definition of system of processes seems to include almost all conventional systems, in particular all alleged solutions to synchronisation problems in the literature. But, for any system of processes there is a synchronisation graph (Theorem 3.1). Thus, any alleged solution to a synchronisation problem leads to an associated synchronisation graph, which can then be analysed to see if it satisfies the Requirements we will give. For this approach to hold, of course, we must assume that the properties we state actually capture the informal problem requirements. This assumption cannot be formally guaranteed, but we hope the stated Requirements capture the intuitive ideas. Nevertheless, any argument with the stated properties can now be based on precise statements. Also, anyone who disputes our Requirements or Properties should consider providing alternative but precise formulations. The advantage of this (compared with previous approaches of supplying a particular programming solution along with an argument that the solution is correct) is that we now have a uniform framework (synchronisation graphs) to discuss Requirements on solutions without assuming specifics about a particular program solution.

The notation of Section D, associated with a synchronisation graph $S(\Sigma)$ is assumed for our statement of Requirements.

Requirement R1: Mutual Exclusion

$$\forall I \in \alpha(S),\ \forall i,j \in [n],\ i \neq j \Rightarrow \Pi_i(I) \neq \underline{cr}_i \text{ or } \Pi_j(I) \neq \underline{cr}_j.$$

This simply states that no (reachable) i.d.'s allow two processes to be in the critical region simultaneously.

Requirement R2: Trying Region Competition

$\forall i \in [n]$, if $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{k-1}} v_k$ is a computation path such that $\Pi_i(\alpha(v_j)) \in T^i$, $j = 1, 2, \ldots, k-2$ and $\Pi_i(\alpha(v_k)) = \underline{cr}_i$, then there exists a nonfailing computation path of S^J,

$\bar{p}' = v'_1 \xrightarrow{e'_1} v'_2 \longrightarrow \ldots \xrightarrow{e'_j} v'_j$ such that $v'_1 = v_1$, $\Pi_i(\alpha(v'_j)) = \underline{cr}_i$.

This Requirement states that if a process in the trying region can eventually enter its critical region from an i.d. I, then it should be able to reach its critical region without competing with other processes not already in the trying region. In particular, this formalises Dijkstra's requirement that a process stopping way outside its critical region cannot block another process. Note that this Requirement is predicated upon the existence of a computation path that enters the critical region. Otherwise, it would presuppose no lockout, an undesirable "logical defect."

Requirement R3: No Deadlock

There does not exist an infinite computation path $\bar{p} = v_1 \longrightarrow v_2 \longrightarrow \ldots$ such that for some $i \in [n]$, and for all $\ell = 1, 2, \ldots$, $\Pi_i(\alpha(v_\ell)) \in T^i$ and $\forall j \in [n]$, $\Pi_j(\alpha(v_\ell)) \neq \underline{cr}_j$.

That is to say, no deadlock implies that it is impossible for a process to be continually trying to enter its critical region but still have no process ever enter its critical region.

Requirement R4: No Lockout

There does not exist an infinite computation path $\bar{p} = v_1 \longrightarrow v_2 \longrightarrow \ldots$ such that for some $i \in [n]$, and for all $\ell = 1, 2, \ldots$, $\Pi_i(\alpha(v_\ell)) \in T^i$.

We note that R4 implies R3: Assuming R4, then using the properties of trying regions (P3) it is easy to see that if process i tries "long enough" without failing, then eventually $\underline{cr}_i$ will be entered. However, R3 does not imply R4.

Requirements R1, R2, and either R3 or R4 (along with suitable Properties) appear as the "minimal" Requirements on any solution to the mutual exclusion problem. However, it seems that Requirement R2, and some of the Properties are formulated explicitly here for the first time.

Additional requirements for "refined" solutions were subsequently added by informal statements in the mutual exclusion problem literature. We now attempt to give precise statements of these requirements.

Requirement R5: No Global Variables

$\forall i \in [m]$, $\exists j_0 \in [n]$ such that $\forall e \in E\ \forall u,v \in S$, $u \xrightarrow{e} v$ and $\Pi_{n+i}(\alpha(u)) \neq \Pi_{n+i}(\alpha(v)) \Rightarrow \Pi_2(\beta(e)) = j_0$.

This condition states that each variable of D is changed by actions from exactly one process. That process may be viewed as the owner of the variable. Other processes may read but not modify the variable.

Requirement R6: Finite Range

$\forall i \in [m]$, $|D_i|$ is finite.

Hence each variable may assume only finitely many values. Note that this implies that the corresponding synchronisation graph is finite i.e. has finitely many vertices.

Requirement R7: Linear Wait

$\forall i \in [n]$ and $\forall$computation paths $\bar{p} = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \cdots \xrightarrow{e_{k-1}} v_k$, $k \geq 4$, where $\Pi_i(\alpha(v_\ell)) \in T^i$ for $\ell = 1,2,\ldots,k$, there does not exist $j \in [n]$, $j \neq i$, such that $\Pi_j(\alpha(v_2)) = \Pi_j(\alpha(v_k)) = \underline{cr}_j$, $\Pi_j(\alpha(v_1)) \neq \underline{cr}_j$ and for some $1 \in \{3,4,\ldots,k-1\}$ $\Pi_j(\alpha(v_\ell)) \neq \underline{cr}_j$.

This condition states that if a process is in its trying region throughout some computation sequence, then in that sequence, no other process may enter its critical region more than once.

Requirement R8: FIFO

$\forall i, j \in [n]$, computation paths $\bar{p} = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$, $\Pi_i(\alpha(v_1)) \in T^i$, $\Pi_j(\alpha(v_1)) \notin T^j \cup \{\underline{cr}_j\}$ and $\Pi_j(\alpha(v_k)) = \underline{cr}_j$ implies that $1 \in \{1,\ldots,k-1\}$ such that $\Pi_i(\alpha(v_\ell)) = \underline{cr}_i$ or c_f^i.

This Requirement imposes a FIFO discipline on processes entering the critical region. The trying region acts as the FIFO queue (compare Peterson and Fischer's notion of "gateway"). Any process may default its position in the FIFO queue by failing. Naturally, this property implies R7 (Linear wait). We note that R8 may be relaxed somewhat by assuming that one distinguished instruction in T^i is the "door" and the priority of processes depend on which process enters the door first. We do not consider this more complicated Requirement.

Requirement R9: Branch or Write

(i) $\forall i \in [n]$, $\forall c \in C^i$, if for some $\bar{d}' \in D$, $\mu^i(c;\bar{d}') \neq \bar{d}'$, then there exist $c' \in C^i$ such that $\forall \bar{d} \in D$, $\lambda^i(c;\bar{d}) = c'$.

(ii) $\forall i \in [n]$, $\forall c \in C^i$, if for some $\bar{d}', \bar{d}'' \in D$, $\lambda^i(c;\bar{d}') \neq \lambda^i(c;\bar{d}'')$,

then $\forall \bar{d} \in D$, $\mu^i(c;\bar{d}) = \bar{d}$.

This condition forces all instructions to be of one of two types: Condition (i) says that if the instruction "writes" into some variable, then it may never cause a branch (i.e. $\lambda^i(c;\bar{d})$ is independent of $\bar{d}$). Condition (ii) says that if the instruction "branches", then it may never write into a variable (i.e. $\mu^i(c;\bar{d}) = \bar{d}$ always).

One may see Requirement R9 as an attempt to restrict the power of synchronisation primitives. An example of a "trivial" solution that is excluded by Requirement R9 is the following: Each process has its critical region preceded by a P(S) and followed by a V(S)

```
    ⋮
  P(S);
  cr_i
  V(S);
    ⋮
```

Note that this solution satisfies what we had called the "minimum requirements" of the mutual exclusion problem, i.e. R1, R2 and R3 (or R4, depending on the different interpretations of the P(S) instruction). Also, the semantics of P(S) is the "busy wait" interpretation, since our definition of processes cannot model the "queue" interpretation.

Requirement R10: Monadic Instructions

$\forall i \in [n]$, $\forall c \in C^i$, $\exists j_0 \in [m]$ such that if $\bar{d}' = \mu^i(c;\bar{d})$, then $\bar{d}$ and $\bar{d}'$ are identical except on j_0.

This condition says that at most one variable may be modified by an instruction. This is a feature satisfied by many programming languages where there is only single variable (as contrasted with array) assignment statements. Like R9, this condition seeks to restrict the power of instructions. Notice that this condition does not exclude an instruction depending upon more than one variable.

F. The Requirements for Dining Philosophers

It is interesting to note that almost no change is required in stating the requirements for the dining philosophers problem from what we have already done for mutual exclusion. The only change is that Requirement R1 (Mutual Exclusion) is replaced with a new neighbour exclusion requirement as stated now:

Requirement R1':

$$\forall I \in \alpha(S),\ \forall i \in [n-1]$$
$$(\Pi_i(I) \neq \underline{cr}_i \text{ or } \Pi_{i+1}(I) \neq \underline{cr}_{i+1})$$
$$\text{and } (\Pi_n(I) \neq \underline{cr}_n \text{ or } \Pi_1(I) \neq \underline{cr}_1).$$

G. Some Theorems on Synchronisation

We now discuss three theorems concerning synchronisation. The first two theorems are lower bounds on the size of D, and the third has to do with the question of representing simultaneity in terms of sequences.

Theorem 4.1. Let Σ be a system of processes that satisfies Properties P2 (Critical Regions) and P8 (Critical Region Reachability). If $S(\Sigma)$ satisfies Requirement R1 (Mutual Exclusion) then $m \geq 1$. That is, D has at least one variable.

This theorem is tight, since the P, V solution uses only one variable and satisfies the stated requirements.

Theorem 4.2. Let Σ be a system of processes that satisfies Properties P2 and P8. If $S(\Sigma)$ satisfies Requirement R1 (Mutual Exclusion) and Requirement R5 (No Global Variables) then $m \geq n$. In particular, D has at least one local variable per process.

This theorem is also tight. A solution exists with only one local variable per process where each such variable takes on only three values.

Theorem 4.3. Under the conditions that (i) instructions are dichotomised, and (ii) for all instructions c, at most one variable in $\delta(c) \cup p(c)$ is public, then "simultaneity $\equiv$ commutativity".

Without going into the details necessary to describe all the terms used here, this theorem means that under some fairly stringent conditions on the complexity of instructions the behaviour of the system as seen by sequences of actions is identical to the behaviour (i.e., includes all possible behaviours) when simultaneous actions actually can occur. Also, by examples, it is known that deleting either of these conditions causes simultaneous behaviour which does not occur when viewed as instruction sequences.

This result will appear in a forthcoming paper, "On Formulating Simultaneity for Studying Parallelism and Synchronisation" by Miller and Yap. A preliminary version of this result appears in the 1978 ACM Theory of Computing Proceedings.

Selected Bibliography

[1] Adams, D.A., "A model for parallel computations," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., Ed. Washington, D.C. :Spartan, 1970, pp. 311-333.

[2] Agerwala, T., "A complete model for representing the coordination of asynchronous processes," Hopkins Computer Research Report #32, Computer Science Program, the Johns Hopkins University, (July 1974).

[3] Agerwala, T., "An analysis of controlling agents for asynchronous processes," Hopkins Computer Research Report #35, Computer Science Program, The Johns Hopkins University, (August 1974).

[4] Agerwala, T. and Flynn, M., "Comments on capabilities, limitations, and 'correctness' of Petri nets," in Proceedings of the 1st Annual Symposium on Computer Architecture, Lipovski, G.J. and Szygenda, S.A. (Eds.) University of Florida, (December 1973), pp. 81-86.

[5] Anderson, D.W., F.J. Sparacio, and R.M. Tomasulo, "Machine philosophy and instruction handling," IBM J. Res. Develop., Vol. 11, Jan. 1967, pp. 8-24.

[6] Aschenbrennner, R.A.: Flynn, M.J.; and Robinson, G.A., "Intrinsic multiprocessing," Proc. AFIPS, 1967 Spring Jt. Computer Conf., 30, AFIPS Press, Motvale, N.J., 1967, pp. 81-86.

[7] Baer, J.L. and E.C. Russel, "Preparation and evaluation of computer programs for parallel processing systems," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., Ed. Washington, D.C.: Spartan, 1970, pp. 375-415.

[8] Baer, J.L., D.P. Bovet, and G. Estring, "Legality and other properties of graph models of computations," J.Assoc.Comput. Mach., 17, July 1970, pp. 543-552.

[9] Baer, J.L., "A survey of some theoretical aspects of multiprocessing," ACM Computing Surveys, Vol. 5, No. 1, March 1973, pp. 31-80.

[10] Bahrs, A.A., "Operation patterns (an extensible model of an extensible language)," Int'l Symp. Theoretical Programming, Novosibirsk, USSR, Aug. 7-11, 1972, Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, 1974, pp. 217-246.

[11] Baker, H.G., "Petri nets and languages," Computation Structures Group Memo 68, Project MAC, M.I.T. (May 1972).

[12] Baker, H.G., "Equivalence problems of Petri nets," S.M. Thesis, Department of Electrical Engineering, M.I.T., (June 1973).

programming control," Comm. Assoc. Comput. Mach., 8, pp. 569-570, September 1965.

[40] Dill, F.H., "Alternative computer archiectures using LSI." IBM Research Report RC 5555, June 1976.

[41] Estrin, G., B. Bussell, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," IEEE Trans. Electron. Comput., EC-12, pp. 747-755, December 1963.

[42] Flynn, M.J., A. Podvin, and K. Shimizu, "A multiple instruction stream processor with shared resources," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs, et al., Ed. Washington, D.C.: Spartan, 1970, pp. 251-286.

[43] Gill, S., "Parallel programming," Comput. J., pp- 2-10, April 1958.

[44] Goldstine, H.H., L.P. Horwitz, R.M. Karp, and R.E. Miller, "On the parallel execution of macroinstructions," IBM Research Report RC-1262, August 17, 1964.

[45] Gonzales, M.J. and C.V. Ramamoorthy, "Recognition and representation of parallel processable streams in computer programs," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs et al., Ed. Washington, D.C.,: Spartan 1970, pp. 335-371.

[46] Gonzales, M.J. and C.V. Ramamoorthy, "Program suitability for parallel processing," IEEE Trans. Comput., C-20, pp. 647-654, June 1971.

[47] Gosden, J.A., "Explicit parallel processing description and control in programs for multi- and uni-processor computers," in 1966 Fall Joint Comput. Conf., AFIPS Conf. Proc., 29,. Washington, D.C.: Spartan, 1966, pp. 651-660.

[48] Graham, W.R., "The parallel and the pipeline computers," Datamation, pp. 68-71, April 1970.

[49] Graham, W.R., "The impact of future developments in computer technology," presented at the Joint Air Force and Lockheed Aircraft Conf. Comput. Oriented Analysis of Shell Structures, August 13, 1970.

[50] Gregory, J. and R. McReynolds, "The SOLOMON computer," IEEE Trans. Electron. Comput., EC-12, pp. 774-781, December 1963.

[51] Hack, M., "The equality problem for vector addition systems is undecidable," Computation Structures Group Memo 121, Project MAC, M.I.T., 1975, pp. 1-32.

[52] Hack, M., "Analysis of production schemata by Petri nets," S.M. Thesis, Department of El. Eng., MIT; also MAC tr-94,

Project MAC, MIT, (February 1972), Errata Hack, M., "Corrections to 'Analysis of production schemata by Petri nets'," Computation Structures Note 17, Project MAC, MIT, (June 1974).

[53] Hack, M., "A Petri net version of Rabin's undecidability proof for vector addition systems," Computation Structures Group Memo 94, Project MAC, MIT, (December 1973).

[54] Hack, M., "Decision problems for Petri nets and vector addition systems," Computation Structures Group Memo 95-1, Project MAC, MIT, (August 1974).

[55] Hack, M., "The recursive equivalence of the reachability problem and the liveness problem for Petri nets and vector addition systems," Computation Structures Group Memo 107, Project MAC, MIT, (August 1974), 9 pp; also in 15th Symposium on Switching and Automata Theory, IEEE, New York.

[56] Hack, M., "Petri net languages," Computation Structures Group Memo 124, Project MAC, MIT, (June 1975).

[57] Hansal, A. and G.M. Schwab, "On marked graphs III," Report LN 25.6.038, IBM Vienna Labs, Vienna, Austrai, (September 1972).

[58] Henhapl, W., "Firing sequences of marked graphs," Report LN 25.6.023, IBM Vienna Labs, Vienna, Austria, (February 1972).

[59] Henhapl, W., "Firing sequences of marked graphs II," Report LN 25.6.036, IBM Vienna Labs, Vienna, Austria, (June 1972).

[60] Harper, S.D., "Automatic parallel processing," Proc. Computing and Data Processing Society of Canada, Second Conference, (June 1960, 321-331.

[61] Holt, A.W., et al., "Applied Data Res. Inc., Rep. AD676972, Inform. Syst. Theory Project Final Rep.,Rome Air Devel. Cen., Contract AF30(602)-4211, September 1968.

[62] Holt, A.W. and F. Commoner, "Events and conditions," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York: Assoc. Comput. Mach., 1970, pp. 3-52.

[63] Holt, R.C., "On deadlocks in computer systems," Ph.D. dissertation, Cornell University, Ithaca, January 1971; also Department Comput. Sci. Tech. Rep. 71-91.

[64] Horwitz, L.P., R.M. Karp, R.E. Miller and S. Winograd, "Index Register Allocation," IBM Research Report RC-1264, August 20, 1964. ACM Journal, Vol. 13, No. 1, pp. 43-61, January 1966.

[65] Irani, K.B. and C.R. Sonnenburg, "Exploitation of Implicit Parallelism in Arithemetic Expressions for an Asynchronous Environment," Department of Elec. and Computer Engineering, University of Michigan Report, Ann Arbor Michigan, 1975.

[66] Izbicki, H., "On marked graphs," IBM Lab., Vienna, Austria, Rep. LR 25.6.023, September 1971.

[67] Izbicki, H., "On marked graphs II," Report LN 25.6.029, IBM Vienna Labs, Vienna, Austria, January 1972.

[68] Jones, N.D., L.H. Landweber, and Y.E. Lien, "Complexity of some problems in Petri nets," 1976.

[69] Karp, R.M. and R.E. Miller, "Properties of a model for parallel computations; determinacy, termination, queueing," IBM Research Report RC-1285, September 1964. Also, SIAM J, Vol. 14, No. 6, pp. 1390-1411, November 1966.

[70] Karp, R., R. Miller, and S. Winograd, "The organisation of computations for uniform recurrence equations," IBM Research Report RC-1667, 1966. Also JACM, Vol. 14, No. 3, July 1967, pp. 563-590.

[71] Karp, R. and R. Miller, "Parallel program schemata: A mathematical model for parallel computation," IEEE Conf. Record 8th Annual Symposium on Switching and Automata Theory, pp. 55-61, October, 1967.

[72] Karp, R.M. and R.E. Miller, "Parallel program schemata," IBM Research Report RC 2053, 1968. JCSS 3, pp. 147-195, May, 1969.

[73] Keller, R.M., "Look-ahead processors," ACM Computing Surveys, 7, No. 4, December 1975, pp. 177-195.

[74] Keller, R.M., "Vector replacement systems: A formalism for modelling asynchronous systems," Princetone University, E.E. Technical Report No. 117, December, 1972. Revised January 1974.

[75] Keller, R.M., "Parallel program schemata and maximal parallelism," J.ACM 20, 3 (July 1973) 514-537; and J.ACM 20, 4 (October 1973), 696-710.

[76] Keller, R.M., "On maximally parallel schemata," in Conf. Rec., 1970 IEEE 11th Annu. Symp. Switching and Automata Theory, pp. 32-50.

[77] Keller, R.M., "On the decomposition of asynchronous systems," in Conf. Rec., 1972 IEEE 13th Annu. Symp. Switching and Automata Theory pp. 78-89.

[78] Knuth, D., "Additional comments on a problem in concurrent programming control," Comm. Assoc. Comput. Mach., Vol. 9, pp. 321-322, May 1966.

[79] Kosaraju, S.R., "Limitations of Dijkstra's semaphore primitives and Petri nets," Technical Report 25, The Johns Hopkins University, (May 1973), also in Operating Systems Review, Vol. 7, No. 4, (October 1973), pp. 122-126.

[80] Dosinski, P.R., "A data flow programming language," IBM T.J. Watson Research Centre Report RC-4264, Yorktown Heights, N.Y., March 1973.

[81] Kotov, V.E., and A.S. Maringani, "On transformation of sequential programs into asynchronous parallel programs" in Proc. IFIPS Congress, 1968, pp. J37-J45.

[82] Kotov, V.E., "Towards automatic construction of parallel programs," Int'l Symp. on Theoretical Programming, Novosibirsk, USSR, August 7-11, 1972. In Lecture Notes in Computer Science, Vol. 5, Springer-Verlag 1974, pp. 309-331.

[83] Kuck, D.J.; Muraoka, Y.; and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up." IEEE Trans. Computers, C-21, 12 December 1972, 1293-1409.

[84] Lehman, "A survey of problems and preliminary results concerning parallel processing and parallel processors," Proc. IEEE, Vol. 54, December 1966, pp. 1889-1901.

[85] Lipton, R.J., "The reachability problem requires exponential space," Yale University, Computer Science Department, Research Report #62, January 1976 (to appear in Theoretical Computer Science J.).

[86] Lipton, R.J., L. Snyder, and Y. Zalcstein, "A comparative study of parallel computation," Proceedings, 15th Annual IEEE Symposium on Switching and Automata Theory, October, 1974.

[87] Lipton, R.J., R.E. Miller, and L. Snyder, "Introduction to linear asynchronous structures," to appear in Proc. of Symposium on Petri Nets and Related Methods, M.I.T., Cambridge, Mass., July 1-3, 1975.

[88] Lipton, R.J., R.E. Miller, and L. Snyder, "Synchronisation and computing capabilities of linear synchronous structures," in Proceedings of the Sixteenth Annual Symposium on Foundations of Computer Science, Berkeley, Cal., October 13-15, 1975, pp. 19-28. Also full version to appear in JCSS.

[89] Logrippo, L., "Renamings in program schemas," in Conf. Rec. 1972 IEEE 13th Ann. Symp. Switching and Automata Theory, pp. 67-70.

[90] Logrippo, L., "Renamings in parallel program schemas," Ph.D. dissertation, University of Waterloo, Waterloo, Canada, February 1974.

[91] Luconi, F.L., "Output functional computational structures," in Conf. Rec., 1968 IEEE 9th Ann. Symp. Switching and Automata Theory, pp. 76-84.

[92] Martin, D.F., and G. Estrin, "Models of computations and systems -- Evaluation of vertex probabilities in graph models of computations," J. Assoc. Comput. Mach., Vol. 14, pp. 281-299, April 1967.

[93] Merlin, P.M., "A Methodology for the Design and Implementation of Communication Protocols," IEEE Trans. on Communications, Vol. COM-24, No. 6, June 1976, pp. 614-621.

[94] Miller, R.E., "Some undecidability results for parallel program schemata," IBM Research Report RC 3371, May, 1971. Also, SIAM Computing Journal, Vol. 1, No. 1, pp. 119-129, March, 1972.

[95] Miller, R.E., and J. Cocke, "Configurable computers: A new class of general purpose machines," IBM Research Report RC 3897. Invited paper presented at the Symposium on Theoretical Programming, Novosibirsk, USSR, August, 1972. In Lecture Notes in Computer Science, Vol. 5, "International Symposium on Theoretical Programming," Springer-Verlag, New York, 1974, pp. 285-298.

[96] Miller, R.E., "A comparison of some theoretical models of parallel computation," IBM Research Report RC-4230. Also IEEE Transactions on Computers, Vol. C-22, No. 8, pp. 710-717, August, 1973.

[97] Miller, R.E., and W.A. Brinsfield, "Insertion of parallel program schemata," Proc. of the 7th Annual Princeton Conference on Information Sciences and Systems, March, 1973.

[98] Miller, R.E., "Eight Lectures on Parallelism: I, Configurable Computers and the Data Flow Model Transformation; II, Computation Graphs and Petri Nets; III-VII, Parallel Program Schemata; VIII, Relationships between Various Models of Parallelism and Synchronisation." Presented at CIME International Mathematical Summer Centre on "Theoretical Computer Science," June 9-14, 1975, Bressanone, Italy. In Proceedings. pp. 5-63.

[99] Miller, R.E., "Relationships among models of parallelism and synchronisation," (Revision of RC-5074) to appear in Proceedings of Symposium on Petri Nets and Related Methods, M.I.T., Cambridge, Mass, July 1-3, 1975.

[100] Miller, R.E., and J.D. Rutledge, "Generating a data flow model of a program," IBM Tech. Disclosure Bull., Vol. 8, pp. 1550-1553, 1966.

[101] Miranker, W.L., "A survey of parallelism in numerical analysis," SIAM Rev., Vol. 13, pp. 524-547, October 1971.

[102] Misunas, D., "Petri nets and speed independent design," CACM, 16, No. 8, August 1973, pp. 474-481.

[103] Morris, D.; and Treleaven, P.C., "A stream processing network," Sigplan Notices 10, 3, (March 1975), 107-112.

[104] Munro, I. and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in Conf. Rec., 1971 IEEE 12th Ann. Symp. Switching and Automata Theory, pp. 132-139. Also JCSS, Vol. 7, No. 2, pp. 189-198.

[105] Mrutha, J.C., "Highly parallel information processing systems," Advances in Computers, pp. 1-116, 1966.

[106] Narinyani, A.S., "Looking for an approach to a theory of models for parallel computation," Int'l Symp. on Theoretical Programming, Novosibirsk, USSR, August 7-11, 1972. IN Lecture Notes in Computer Science, Vol. 5, Springer-Verlag, pp. 247-284.

[107] Nash, B.O., "Reachability problems in vector addition systems," The American Mathematical Monthly, 80, 3, 292-295, March, 1973.

[108] Noe, J.D., "A Petri net model of the CDC 6400," Report 71-04-03, Computer Science Department, University of Washington, (1971); also in Proc. of the ACM SIGOPS Workshop on System Performance Evaluation, ACM, New York, (1971), pp. 362-368.

[109] Noe, J.D. and G.J. Nutt, "Macro E-nets for representation of parallel systems," in IEEE Trans. on Computers, Vol. C-22, No. 8, (August 1973) pp. 718-727.

[110] Patil, S.S. and Dennis, J.B., "The description and realisation of digital systems," Computation Structures Group Memo 71, Project MAC, M.I.T. (October 1972); also in Sixth Annual IEEE Computer Society Int'l Conference Digest of Papers, IEEE, (1972).

[111] Patil, S.S., "Coordination of asynchronous events," Ph.D. dissertation, M.I.T., Cambridge. (Project MAC Rep. TR-72, September 1967).

[112] Patil, S.S., "Closure properties of interconnections of determinate systems," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Comput., New York: Assoc. Comput. Mach., 1970, pp. 10-116.

[113] Peterson, J.L., "Petri Nets," U. of Texas msc., July 1976.

[114] Peterson, J.L., "Modelling of Parallel Systems," Ph.D. Thesis, Elec. Eng. Department, Stanford University, January, 1974.

[115] Peterson, J.L., and T.H. Bredt, "A comparison of models of parallel computation," Inform. Processing 74, Proceedings IFIP Congress 1974, 466-470, August, 1974.

[116] Petri, C.A., "Communication with automata," Suppl. 1 to Tech. Rep. RAD C-TR-65-337, Vol. 1, Griffiss Air Force Base, New York, 1966. (Translated from Kommunikation mit Automaten, University of Bonn, Bonn, Germany, 1962.)

[117] Ramamoorthy, C.V. and M.J. Gonzalez, "A survey of techniques for recognisable parallel processable streams in computer programs," in 1969 Fall Joint Comput. Conf., AFIPS Con. Proc., Vol. 35, Montvale, N.J.: AFIPS Press, 1969, pp. 1-15.

[118] Reddi, S.S. and E.A. Feustel, "A restructurable computer system," Report, Laboratory for Computer Science and Engineering, Rice University, Houston, Texas, March 1975.

[119] Reigel, E.W., "Parallelism exposure and exploitation," in Parallel Processor Systems, Technologies, and Applications, L.C. Hobbs et al., Ed. Washington, D.C.: Spartan, 1970, pp. 417-438.

[120] Reiter, R., "Scheduling parallel computations," J. Assoc. Comput. Mach., Vol. 15, pp. 590-599, 1968.

[121] Riddle, W.E., "The modelling and analysis of supervisory systems," Ph.D. Thesis, Computer Science Department, Stanford University, (March 1972).

[122] Riddle, W.E., "The equivalence of Petri nets and message transmission models," SRM 97, The University of Newcastle upon Tyne, (August 1974).

[123] Rodriguez, J.E., "A graph model for parallel computation," Ph.D. dissertation, M.I.T., Cambridge, September 1967. (Also M.I.T., ESL, and Project MAC Rep. ESL-R-398, MAC-TR-64, September 1969.)

[124] Rohrbacher, D.L., "Advanced computer organisation study," Rome Air Devel. Corp. Tech. Report RADC-TR-66, 7 (2 vols.) AD 631 870, and 631 871 (April 1966).

[125] Rose, C.W., "A system of representation for general purpose digital computer systems," Ph.D. dissertation, Case Western Reserve University, Cleveland, Ohio, September 1970.

[126] Rose, C.W., "LOGOS and the software engineer," in 1972 Fall Joint Comput. Con., IFIPS Conf. Proc, 41. Montvale, N.J.: AFIPS Press, 1972, pp. 311-323.

[127] Rose, C.W., and F.T. Bradshaw, "The LOGOS representation system," Case Western Reserve University, Cleveland, Ohio, Rep., October 1971.

[128] Russell, E.C., "Automatic program analysis," Ph.D. dissertation, University of California, Los Angeles, 1969.

[129] Rutledge, J.D., "Parallel processes, schemata and

transformations," IBM Res. Rep. RC 2912, June 1970.

[130] Rutledge, J.D., "Program schemata as automata, part I," in Conf. Rec. 1970 IEEE 11th Annu. Symp. Switching and Automata Theory, pp. 7-24.

[131] Schwartz, J., "Large parallel computer," J. Assoc. Comput. Machine., pp. 25-32, January 1966.

[132] Senzig, D.N. and R.V. Smith, "Computer organisation for array processing," in 1965 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 27. Montvale, N.J.: AFIPS Press, 1965, pp. 117-128.

[133] Shapiro, R.M. and H. Saint, "The representation of algorithms,' Applied Data Res., Inc., Rome Air Develop. Cen., Tech. Rep. TR-69-313. Vol. 2, September 1969.

[134] Shapiro, R.M. and H. Saint, "The representation of algorithms as cyclic partial orderings," Meta Information Applications, Inc., NASA Final Rep., Contract NASW-2097, July 1971.

[135] Slutz, D.R., "Flow graph schemata," in Rec. Project MAC Conf. Concurrent Syst. and Parallel Computation. New York: Assoc. Comput. Mach., 1970, pp. 129-141.

[136] Slutz, D.R. "The flowgraph schemata model of parallel computation," Ph.D. dissertation, M.I.T., Cambridge, September 1968.

[137] Sonnenburg, C.R., "A configurable parallel computing system," Ph.D. Dissertation, University of Michigan, Ann Arbor, October 1974.

[138] Stone, H.S., "A pipeline push-down stack computer," in Parallel Processor Systems, Technologies, and Applications. Spartan Books, Washington, D.C., 1970, pp. 235-249.

[139] Syre, J.C., "From the single assignment software concept to a new class of multiprocessor architectures," Report, 1975 Department d'Informatique, C.E.R.T. BP4025, 31055 Toulouse Cedex, France.

[140] Tjaden, G.S. and M.J. Flynn, "Detection and parallel execution of independent instruction," IEEE Trans. Comput., Vol. C-19, pp. 889-895, October 1970.

[141] Thurber, K.J., "Associative and Parallel Processors," Computing Surveys, Vol. 7, No. 4, December 1975.

[142] van Leeuwen, J., "A partial solution to the reachability-problem for vector-addition systems," Proceedings, 6th Annual ACM Symposium on Theory of Computing, 303-309, May, 1974.

[143] Vantilborgh, H. and A. van Lansweerde, "On an extension of Dijkstra's semaphore primitives," Information Processing Letters, 1, 181-186, October 1972.

[144] Winograd, S., "Parallel interactive methods," in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Ed. New York: Plenum, 1972.

[145] Yoeli, M., "Petri nets and asynchronous control networks," Applied Analysis and Computer Science Research Report CS-73--07, University of Waterloo, Waterloo, Ontario, Canada, April, 1973.

ADDITIONAL REFERENCES

[1] Anshel, M., "Decision problems for HNN groups and vector addition systems," Mathematics of Computation 30 No. 133 (January 1976), pp. 154-156.

[2] Brinch Hansen, P., "A Comparison of Two Synchronisation Concepts," ACTA Informatica 1 (1972), pp. 190-199.

[3] Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering SE-1 (2), June 1975, pp. 199-207.

[4] Campbell, A.H.,and A.N. Habermann, "The Specification of Process Synchronisation by Path Expressions," Lecture Notes in Computer Science 16, Springer Verlag, Heidelberg (1974), pp. 89-102.

[5] Courtois, P.J., F. Heymans, D.L. Parnas. Concurrent control with "readers" and "writers". CACM 14(10):667-668.

[6] Cremers, Armin and T.N. Hibbard, "An Algebraic Approach to Concurrent Programming Control and Related Complexity Problems," Report, USC Computer Science Program, November, 1975.

[7] Crespi-Reghizzi, S. and D. Mandrioli, "A decidability theorem for a class of vector-addition systems," Information Processing Letters 3 No. 3 (January 1975) pp. 78-80.

[8] Ellis, C.A., "The Validation of Parallel Co-operating Processes," University of Colorado, Computer Science Report CU-CS-065-75, April, 1975.

[9] Feldman, J.A., "Synchronising Distant Cooperating Processes," Department of Computer Science, University of Rochester, Report TR26, October 1977.

[10] Gilbert, Philip and W.J. Chandler, "Interference Between Communicating Parallel Processes," CACM 15, No. 6 (June, 1972) 427-437.

[11] Habermann, A.N. Synchronisation of Communicating Processes. CACM 15(3): (March 1972) 171-176.

[12] Hoare, C.A.R., "Towards a Theory of Parallel Programming," in Operating Systems Techniques, ed. R.H. Perrot, Academic Press, London (1971), pp. 61-71.

[13] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Comm. of the ACM 17 (10), October 1974, pp. 549-557.

[14] Horning, J.J. and B. Randell, "Process Structuring" ACM Computing Surveys 5 (March 1973), pp. 5-30.

[15] Howard, J.H., "Signalling in Monitors," Proc. of the Second International Conference on Software Engineering, San Francisco, October 1976.

[16] Karp, R.A. and D.C. Luckham, "Verification of Fairness in an Implementation of Monitors," ibid.

[17] Keller, R.M., "Formal Verification of Parallel Programs," CACM 19, 7 (July 1976), pp. 371-384.

[18] Lamport, L., "On Concurrent Reading and Writing," Report CA-7409-0511, Massachusetts Computer Associates, Inc., September 1974, revised March 1976.

[19] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System," Report CA-7603-2911, Massachusetts Computer Associates, Inc., March 1976. Also CACM, 21, (July 1978) pp. 558-565.

[20] Landweber, L.J. and E.L. Robertson, "Properties of conflict free and persistent Petri nets", JACM 25, (July 1978) 352-364.

[21] Lauer, P.E. and R.H. Campbell, "Formal Semantics of High-level Primitives for Co-ordinating Concurrent Processes," ACTA Informatica 5 (1975), pp. 247-332.

[22] Lipton, R.J., "Schedulars as enforcers in Synchronisation Processes," Operating Systems Proceedings of an Int'l Symposium held at Rocquencourt, Lecture Notes in Computer Science Vol. 16, Springer-Verlag, Heidelberg 1974, 237-249.

[23] Lipton, Richard J., "On Synchronisation Primitive Systems," Ph.D. Thesis, Carnegie-Mellon University, 19733 and Research Report #22, Yale University, Department of Computer Science, October 1973.

[24] Logrippo, L., "Renamings and Economy of Memory in Program Schemata," JACM 25 (January 1978) pp. 10-22.

[25] Meyer, S.C., "An analysis of two models for parallel computations," Ph.D. Thesis, Department of Electrical Engineering, Rice University, Houston, Texas (1974).

[26] Miller, R.E., "Theoretical Studies of Synchronous and Parallel Processing," Proceedings of the 1977 Conference on Information Sciences and Systems, John Hopkins University, March 1977, pp. 333-339.

[27] Miller, R.E., "Mathematical studies of parallel computation," Proceedings 1st IBM Symp. on Mathematical Foundations of Computer Science, October 20-22, 1976, IBM Amagi Homestead, Japan, 23 pp.

[28] Miller, R.E. and C.K. Yap, "On Formulating Simultaneity for Studying Parallelism and Synchronisation," Proceedings of the Tenth Annual ACM Symposium on Theory of Computing,

May, 1978, pp. 105-113.

[29] Miller, R.E. and C.K. Yap, "Formal Specification and Analysis of Loosely Connected Processes," IBM Research Report RC-6716, September, 1977.

[30] Osterweil, J.P. and G.J. Nutt, "Modelling Process-Resource Activity," University of Colorado Computer Science Report CU-CS-084-75, November 1975.

[31] Parent, Michel, "Presentation of the Control Graph Models," Operating Systems Proceedings of an Int'l Symposium held at Rocquencourt, Lecture Notes in Computer Science Vol. 16, Springer-Verlag, Heidelberg 1974, pp. 279-292.

[31] Peterson, J.L., "Petri nets," ACM Computing Surveys 9(September 1977) 223-252.

[33] Peterson, G.L. and M.J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System, extended abstract." Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, May 1977, 91-97.

[34] Rackoff, C., "The covering and boundedness problems of vector addition systems," to appear in Theoretical Computer Science J., 1977.

[35] Rivest, R.L. and V.R. Pratt, "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report." Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Sceince, October, 1976, 1-8.

[36] Sacerdote, G.S. and R.L. Tenney, "The decidability of the reachability problem for vector addition systems," to appear Proceedings ACM Theory of Computing Symposium, May 1977.

[37] Sayward, F.G., "Research Issues in Synchronisation Primitives for Operating Systems Languages," in Research Divisions in Software Technology P. Wagner (ed) MIT Press, 1977.

[38] Schmid, H.A., "On the Efficient Implementation of Conditional Critical Regions and the Construction of Monitors," ACTA Informatica 6 (1976), pp. 227-249.

[39] Schneider, E.A., "Synchronisation of Finite State Shared Resources," Department of Computer Science, Carnegie-Mellon University, Ph.D. Thesis, March 1976.

[40] Yap, C.K., "On Abstract Synchronisation Problems and Synchronisation Systems." Unpublished manuscript, 1976.

[41] Zave, Pamela, "On the Formal Definition of Processes." Proceedings of International Conference on Parallel Processing, 1976.

[42] Zave, Pamela and D.R. Fitzwater, "Specification of Asynchronous Interactions Using Primitive Functions." Technical Report, Department of Computer Science, University of Maryland, 1977.