HOPKINS COMPUTER RESEARCH REPORTS

REPORT #32

JULY 1974

A COMPLETE MODEL FOR REPRESENTING
THE COORDINATION OF ASYNCHRONOUS PROCESSES

BY

TILAK AGERWALA

MASTER

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE

COMPUTER SCIENCE PROGRAM

THE JOHNS HOPKINS UNIVERSITY

BALTIMORE, MARYLAND

DISCLAIMER

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

DISCLAIMER

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

A COMPLETE MODEL FOR REPRESENTING
THE COORDINATION OF ASYNCHRONOUS PROCESSES*

MASTER

Tilak Agerwala
Electrical Engineering Dept.
The Johns Hopkins University
Baltimore, Maryland 21218

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

NOTICE

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

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

84

1. INTRODUCTION

We are entirely concerned here with discrete systems. A digital computer system, for example, falls in this category; a system where one studies fluid motion does not. The systems we are interested in exhibit concurrency. They consist of a number of interacting processes. For our purpose, we consider a process to be a unified sequence of actions. In a computer system, a process would be a procedure in execution by a processor. The processes are asynchronous, i.e., we are not permitted to make any assumptions regarding the relative speeds with which actions in different processes are performed. The processes proceed at their own pace (which may be variable) and interact with one another to contribute to the total system behavior.

One can visualize two extreme forms of concurrency. The processes may be completely dependent in which case they must be executed sequentially. On the other hand, they may be completely independent in which case they can be allowed to proceed concurrently or in any order. Any given system will probably lie somewhere between these two extremes. It is the in between behavior that we are more interested in. In this case, the processes are said to interact in some manner. In large systems this interaction can be quite complex. Each process by itself may be easy to understand but when processes are placed in an environment where they have to interact with one another, the overall behavior can be quite complex.

In recent years much work has been done in the area of studying parallel systems. Various models have been proposed and studied in depth. [1, 4, 7, 8, 9, 13, 15, 17, 19, 20, 21, 23, 24, 27, 28, 30, 31, 33, 35].

In some models more emphasis has been placed on the computational aspects such as determinacy. In others, the emphasis is on the control part and the ability and ease of representing different kinds of interaction. In either case, however, to quote Miller [25], "one is aiming at modeling computational processes in such a way that they closely resemble how computers actually operate. This approach, with its general methods for representing and manipulating computational processes, has the potential of providing major tools for aiding in the design of future computers and their associated systems programs. Also it provides insight into new organizational structures for computers and their operation."

Various language structures have been proposed to implement desired interactions between processes. [3, 6, 10, 11, 12, 14, 16, 18, 29, 32, 34, 36]. For our purpose, we can consider them to be models for representation of parallelism. We will not describe any of the models mentioned so far but refer the reader to two excellent papers [5, 25].

Our main interest in this paper is the study of control flow and its representation. In all the models mentioned, there is a controlling agent either explicitly specified or implicit in the model. We will focus attention on this control part.

Basically, any model for studying parallel systems consists of two parts:

(1) A control C which decides whether a particular process is to be enabled or not.

(2) A set of asynchronous processes $PR = \{P_1, P_2, \dots, P_n\}$ that are to be "coordinated."

If C and PR are the only two entities specified, we have a completely uninterpreted model. In general, each process performs some computation. While studying the representation of coordinations, we are not concerned with the computation performed by the system as a whole or that performed by each individual process. If the computation is of interest, additional specification must be provided. Given an uninterpreted model, if

(1) A set of storage variables $M = \{m_i \mid 1 \leq i \leq n\}$ is defined where each m_i carries with it size and type information and

(2) For each process P_i , a set of input and output data sets is specified where each data set is a subset of M ,

then the model is said to be semi-interpreted. Given a semi-interpreted model, if one defines

(1) The initial values of variables in M

(2) A set of functions, one for each process P_i which takes the values of the variables in the input data set of P_i , performs some transformations on these values and modifies the values of the output data set of P_i ,

then a completely interpreted model results. Thus, an uninterpreted model is an abstraction of a real system that allows one to focus attention on the flow of control. From now on we will be concerned only with uninterpreted models. The three models that we will propose are uninterpreted. When we refer to an existing model (say X) we refer to the model that results when X is stripped of all interpretation.

We now come to the question of completeness. The notion of coordination is vague and an uninterpreted model is complete if it can represent all possible types of interactions between a group of processes, i.e., if the model provides a formal counterpart to the notion of coordination.

None of the models we have examined so far are complete. We feel that this is a fairly serious shortcoming. Apart from the fact that it would naturally be nice to have a complete model, psychological problems can arise because of the incompleteness. Since one is not certain initially whether a desired coordination can be represented or not, the task of modeling a particular system becomes all the more frustrating and often leads to false claims about capabilities. With the lack of a formal model, there is the lack of a uniform underlying theory.

Secondly, it is desirable that the model provide a natural way of representing a desired coordination. The human mind finds it difficult to "think parallel" and the model must make clear exactly what is going on. The interaction between the different parts should be made visible and obvious. Without this, it would be impossible to convince oneself that a particular exercise in modeling was correctly carried out and that a given representation did in fact reflect what it was supposed to. Some of the models mentioned do not meet this requirement satisfactorily.

In this paper we will present three models for representing interaction in a parallel system. Each model arises out of different motivations. We will show that the three models are equivalent and complete. One of these, the extended Petri net model was first studied in [2]. It is this model we are primarily interested in because it retains all the nice features of Petri nets such as ease of understanding and ease of representation. These three models should help greatly

in any systematic and formal study of parallel systems. In particular it will be possible to state in a precise and natural manner the coordination desired between a group of processes, and given a concurrent system, to study the interaction between the various parts. The next step would be the development of orderly means of implementing a desired coordination.

2. COORDINATION SYSTEM

Since we are talking about "complete" models, i.e., models which provide a formal counterpart to the notion of coordination, we must explain intuitively at least, what is meant by a coordination system. A coordination system consists of a group of processes placed in a certain environment. Each process can be completely independent of some processes and completely or partially dependent on others. Whenever a process desires to start, it checks certain conditions (or alternatively, requests a centralized controller to do so). If the conditions are satisfied the process is enabled (allowed to run) if not, it is disabled (not allowed to run). We assume that the conditions are such that there exists an effective way to determine whether the conditions are satisfied or not. In a finite though a priori unknown and unbounded time, an enabled process completes and stops. For any given system, whether a process is allowed to proceed or not depends in some way on the "past history" of the system and the initial state of the system.

One has to be careful about a certain characteristic of this system. It is possible that there are two processes P_1 and P_2 such that at a particular instant the conditions for enabling each one are satisfied but if P_1 is enabled P_2 should not be. This is because enabling P_1 changes the history of the system and the conditions for enabling P_2 may no longer be satisfied. This problem can be taken care of by ensuring that the condition checking takes place serially.

The behavior of a given coordination system can be described in two ways:

- (1) By naming the processes, the initial conditions and the conditions under which each process can be enabled.

- (2) By considering the sequences which the system can produce. In this approach, at every stage we allow only one process to run. Only when it signals completion, can another process be enabled. In this way, a temporal sequence of processes is constructed. We claim that a coordination system can be characterized completely by the set of all such sequences it can produce.

We feel that this constitutes a reasonable, intuitive description of a coordination system. All practical systems we can think of can be characterized in the above manner.

This completes the ground work. We have intuitively described a coordination system. We will now introduce the three uninterpreted models that formalize this intuitive notion.

3. THE TURING MACHINE MODEL

It is our belief that the notion of "algorithm" as formalized by Turing machines is very closely related to the notion of "coordination." If a model for representing interaction in a parallel system is not "computationally complete" it should be possible to specify a coordination problem where the model fails. Since a coordination system can be characterized by process sequences, any model can be viewed as a language recognizer. If the model does not have the power of a Turing machine we can specify languages it cannot recognize. Each of these languages describes a coordination problem where the model fails.

Secondly, the decision as to whether a particular process should be enabled is based on a set of conditions. If there is any effective way to determine whether these conditions hold, the Turing machine can make the appropriate decision. These two points motivate a model where a Turing machine serves as a central controller and decision maker. The model is shown in fig. 1.

$P_1, P_2, \dots, P_n$ are processes which have to be coordinated. Whenever a process P_i desires to run, it sends a signal r_i to the finite state machine. On receiving r_i the fsm starts computing on the infinite tape according to its built-in program. The entire past history can be stored on the tape. If the fsm decides that P_i can be enabled it sends a signal e_i to P_i else it transmits d_i and disables P_i . All this information can be stored on the infinite tape. When P_i stops, it sends a signal f_i to the fsm and this is also recorded. When a process desires

to send a signal to the fsm we say it is waiting for service. After servicing each request the fsm goes into one of the states, q_f or q_s . (The reason for having two states will be explained later). Only when it is in one of these states is it ready to accept requests from the processes. At any stage, there may be more than one process awaiting service. According to our description of a coordination system, the important point was that condition checking occurred serially. The order is unimportant since all processes are to be treated equally with no built in priority. (If a priority scheme is desired, it can easily be built into another fsm which checks priorities and forwards the request with the highest priority.) Thus, we have introduced queues to ensure that condition checking occurs serially. The processes desiring service compete for the queue. The order in which requests are placed, however, is arbitrary and a priori unknown. The presence of the queues is not important, the fact that condition checking occurs serially, is. After its first enabling, a process can send an r signal only if it has previously received an e or d signal. Thus the queue lengths will not exceed n .¹

To describe a system modeled in this manner, one of two specifications has to be given:

- (1) The processes are named and the fsm is specified.
- (2) Only one process is allowed to run at any stage and the fsm services another request only after receiving an f_i signal. When it receives an f_i signal, it makes a decision to go into state q_f or q_s . Any sequence of process enablings that causes it to go into state q_f

¹We have specified only a control and a set of processes to be coordinated. The Turing machine model is thus uninterpreted.

is a sequence produced by this system. The set of all sequences that can be produced characterizes the behavior of the system.

The Turing machine model is a formal model for representing coordination between a group of processes. The description of a coordination system given earlier was intuitive and not well defined. For this reason we can never prove rigorously that the model provides a correct formal counterpart to the vague notion of a coordination system, i.e., that the model is complete. However, the close relationship between the notion of algorithm and coordination (pointed out earlier in this section), appears to justify the claim that the Turing machine model is complete.

4. THE EXTENDED PETRI NET MODEL

This model emerged during a study of the capabilities and limitations of Petri nets [2]. Kosaraju [22] presented certain problems and proved that the desired coordination in each case could not be represented using Petri nets. The approach in [2] was as follows: A number of modifications were made to ordinary Petri nets. One of these modifications yielded the extended Petri net model. All the other models obtained were less powerful and equivalent to ordinary Petri nets. This led us to believe that the extended model was complete. We applied the model to many coordination systems (including the problems proposed by Kosaraju) and in each case, the model was adequate. The extended model also retains all the good features of Petri nets. A net from this

model is easy to understand and provides a clear representation of the interaction between the processes involved. A formal study of the capabilities of this model was thus called for.

The Turing machine model was one that provided a centralized control. Whether a process should be enabled or not was decided by a central agency. In contrast to this, the extended Petri net model essentially provides for distributed control. Each process checks certain "local" conditions and decides for itself whether to proceed or not. As far as completeness is concerned, the type of control provided is not critical. We will show in this section that the extended Petri net model can be used to "build" a central controller as powerful as the Turing machine central controller. We will prove that extended Petri nets are computationally complete even though they are completely uninterpreted.

Definitions

1. The extended Petri net model is a directed graph defined as a six-tuple (T, P, A, M^0, M_F, f) where:

$T = \{t_i | 1 \leq i \leq n\}$ is a finite set of "transitions."

$P = \{p_i | 1 \leq i \leq m\}$ is a finite set of "places."

$A \subseteq \{(p_i, t_j) | p_i \in P, t_j \in T\} \cup \{(t_i, p_j) | t_i \in T, p_j \in P\} \ni$

$(p_i, t_j) \in A \Rightarrow (t_j, p_i) \notin A$ and

$(t_i, p_j) \in A \Rightarrow (p_j, t_i) \notin A$ is a finite set of directed arcs. Each arc either connects a transition to a place or a place to a transition.

$M^0: P \rightarrow N$ (N is the set of non-negative integers) is a function that gives the number of "markers" in each place initially.

$M_F = \{M_{Fi} \mid 1 \leq i \leq p\}$ is a finite set of final markings

where $M_{Fi}: P \rightarrow N \cup \{d\}$

$f: A \rightarrow \{T, F\}$ is a predicate on the arcs

where $T = \text{true}$ and $F = \text{false}$.

T, P form the nodes of the graph. The transitions represent processes and the places, conditions.

2. $I_i = \{j \mid (p_j, t_i) \in A\}$ i.e., the set of input places of t_i .

3. $O_i = \{j \mid (t_i, p_j) \in A\}$ i.e., the set of output places of t_i .

4. $I_i' = \{j \mid (p_j, t_i) \in A \text{ and } f((p_j, t_i)) = T\}$

5. $I_i'' = \{j \mid (p_j, t_i) \in A \text{ and } f((p_j, t_i)) = F\}$

6. $P_i =$ the number of markers in p_i at some instant.

Simulation rules

Transitions represent events or processes. The firing of a transition represents the occurrence of an event. Whenever a transition t_i desires to fire it checks its input places and is enabled if and only if $P_j > 0, \forall j \in I_i'$ and $P_j = 0, \forall j \in I_i''$. At this stage a marker is removed from each $p_j, j \in I_i'$, and the transition t_i starts firing. At some later finite, though a priori unknown and unbounded time, it completes firing and places a marker in each output place. Firing of a transition thus causes a change of state of the net and other transitions may be enabled or disabled as a result of this firing. The number of markers in any place is shown in the net by a corresponding number of dots. The movement of markers clearly shows which conditions cause a transition to fire and also which conditions come into being at the completion of firing.

Ordinary Petri nets are similar to extended Petri nets except that $f(a) = T, \forall a \in A$. This apparently small change, however, results in the tremendous power of the extended Petri net model.

Let us illustrate the model by means of an example. The problem is quite realistic and may arise in any operating system. It is one of the problems proposed by Kosaraju for which ordinary Petri nets are inadequate.

Example

1. There are four processes R_1, R_2, C_1 and C_2 . R_1 and R_2 are "producers." They produce "items" asynchronously and place them in buffers (FIFO queues) B_1 and B_2 respectively. The buffers are unbounded. The "consumers" C_1 and C_2 consume items from B_1 and B_2 respectively. These four asynchronous processes must interact in some manner so that a consumer does not try to consume when its buffer is empty. There is also an additional requirement. C_1 and C_2 share the channel through which they access the buffers. We give C_1 priority so that C_2 can consume only if B_1 is empty.

The net N in fig. 2 represents the desired coordination.

$$N = (T, P, A, M^0, M_F, f)$$

$$T = \{t_1, t_2, \dots, t_8\}$$

$$P = \{p_1, p_2, \dots, p_{10}\}$$

$$A = \{(p_1, t_1), (t_1, p_2), (p_2, t_2), (t_2, p_1), (t_2, p_9), \\ (p_3, t_3), (t_3, p_4), (p_4, t_4), (t_4, p_3), (t_4, p_{10}), \\ (p_5, t_5), (t_5, p_6), (p_6, t_6), (t_6, p_5), (p_9, t_5), \\ (p_7, t_7), (t_7, p_8), (p_8, t_8), (t_8, p_7), (p_9, t_7), (p_{10}, t_7)\}$$

$$M^o(p_i) = 1, i \in \{1, 3, 5, 7\}$$

$$M^o(p_i) = 0, i \in \{2, 4, 6, 8, 9, 10\}$$

$$M_F = \{M_{F1}\} \text{ here } M_{F1}(p_i) = M^o(p_i) \text{ } i \in \{1, 3, 5, 7\}$$

$$\text{and } M_{F1}(p_i) = d \text{ for } i \in \{2, 4, 6, 8, 9, 10\}$$

(Note: M_F has been introduced in the definition for the purpose of being able to describe the behavior of a net by specifying the sequences produced by the net. This will be discussed later and may be ignored at this stage.)

$f((p_9, t_7)) = F$, we have indicated this by putting a bar across arc (p_9, t_7) in the figure.

$$f(a) = T, \forall a \in A - \{(p_9, t_7)\}$$

t_1, t_3, t_5, t_7 represent processes R_1, R_2, C_1 and C_2 respectively. The other transitions are "dummy" transitions introduced because a place p_1 cannot be an input and output place of the same transition.

Thus, R_1 and R_2 proceed asynchronously. The number of markers in p_9 at any stage is the difference between the number of times t_2 and t_5 have fired. Therefore p_9 is greater than zero if and only if B_1 is not empty. Thus, C_1 can consume only if B_1 is not empty. Similarly, C_2 can consume only if B_1 is empty and B_2 is not empty.

Two points have to be stressed. Firstly, the net for this problem was produced with almost no effort. Most other models cannot even represent the desired coordination. Secondly, the net is very easy to understand.

Describing the coordination represented by a net in this model

As before there are two ways of doing this. For each of these methods we require the notion of labeling. In the example, we specified that certain processes were associated with certain transitions. Other transitions we said were dummy transitions. This naming of transitions is a "trans-labeling" of the net.

Definition

7. More formally we define a trans-labeling $I(T', E)$ of a net (T, P, A, M^0, M_F, f) as follows:

$T' = \{t_{a_1}, t_{a_2}, \dots, t_{a_m}\} \subseteq T$ is a set of transitions

$E = \{E_1, \dots, E_k\}$, $k \leq m$ is a set of event or process names.

$I: T' \rightarrow E$ is a function from T' onto E .

Thus, the same event or process name may be attached to different transitions and the same net may represent different coordinations depending on its trans-labeling.

The two ways of specifying the coordination represented by a net are:

(1) By defining T, P, A, M^0, f and giving a trans-labeling

(2) By considering the sequences produced by the net. For this

we require some more definitions.

Definitions

8. A marking M is covered by M_{F_i} , $1 \leq i \leq p$ if $M(p_k) = M_{F_i}(p_k) + \sum_{j=1}^p M_{F_j}(p_k) \in \mathbb{N}$, i.e., $M_{F_i}(p_k) \neq d$.

9. $\bar{t} = t_{b_1} t_{b_2} \dots t_{b_n} \in T^*$ is said to be a simulation sequence of a net $N = (T, P, A, M^0, M_F, f)$ if $\exists$ a sequence of

markings $M^0, M^1, \dots, M^n \ni t_{b_i}$ can be enabled in M^{i-1} , firing of t_{b_i} in M^{i-1} results in M^i , $1 \leq i \leq n$ and M^n is covered by M_{F_j} for some $M_{F_j} \in M_F$.

10. The set of all simulation sequences of N is called the simulation set of N or SIMSET_N .

11. Let $T' \subseteq T$. Then for each simulation sequence $\bar{t} = t_{b_1} t_{b_2} \dots t_{b_n}$ of N we define a reduced simulation sequence $\bar{t}' = t_{c_1} t_{c_2} \dots t_{c_p}$ with respect to T' where $\bar{t}'$ is obtained from $\bar{t}$ by deleting all $t_{b_i} \in T - T'$.

12. $\text{SIMSET}|T'$ is the set of all reduced simulation sequences of N with respect to T' .

13. For each reduced simulation sequence $t_{b_1} \dots t_{b_m}$ with respect to T' we define a process simulation sequence $E_{c_1} E_{c_2} \dots E_{c_m}$ with respect to $I(T', E)$ where $I(t_{b_i}) = E_{c_i}$, $1 \leq i \leq m$.

14. The set of all process simulation sequences of N with respect to $I(T', E)$ is called $I(\text{SIMSET}_N)$.

The coordination represented by a given net N and trans-labeling I is completely specified by $I(\text{SIMSET}_N)$.¹

Completeness of the extended Petri net model

The following theorems will establish the equivalence of the extended Petri net model and the Turing machine model with respect to their capability of representing coordinations.

Definition

15. A finite state machine is a six-tuple.

$$S = (Q, I, O, \delta, q_0, Q_F)$$

$Q = \{q_i | 0 \leq i \leq n\}$ is a finite set of states.

$I = \{a_i | 1 \leq i \leq m\}$ is a finite set of input symbols.

$O = \{b_i | 1 \leq i \leq p\}$ is a finite set of output symbols.

¹Note that a trans-labeled extended Petri net is uninterpreted.

$\delta : Q \times I \rightarrow Q \times O$ is a transition function

$q_0 \in Q$ is the initial state of the machine.

$Q_F \subseteq Q$ is a set of final states.

The machine is synchronised with its environment. If the machine is in state q_i , receives an input a_j then if $\delta((q_i, a_j)) = (q_k, b_\ell)$, it goes to state q_k and outputs b_ℓ . When the environment receives the output it presents the machine with its next input.

Theorem 1

Petri nets can simulate finite state machines.

Proof

For any finite state machine defined as above we construct a Petri net $N = (T, P, A, M^0, M_F, f)$ which simulates the behavior of S step by step. N is formally defined as follows:

$$T = \{t_{ij} \mid 0 \leq i \leq n, 1 \leq j \leq m\} \cup \{\bar{t}_{ij} \mid 0 \leq i \leq n, 1 \leq j \leq m\}$$

$$P = Q \cup I \cup \{p_{ij} \mid 0 \leq i \leq n, 1 \leq j \leq m\}$$

A is constructed according to the following rules:

$$\forall i \in \{0, 1, 2, \dots, n\} \text{ and } j \in \{1, 2, \dots, m\},$$

$$(q_i, t_{ij}) \in A$$

$$(a_j, t_{ij}) \in A$$

$$(t_{ij}, p_{ij}) \in A$$

$$(p_{ij}, \bar{t}_{ij}) \in A$$

$$((\bar{t}_{ij}, q) \in A \text{ and } (\bar{t}_{ij}, b) \in A) \text{ if and only if } \delta((q_i, a_j)) = (q, b)$$

M^0 and M_F are defined as follows;

$$M^0(x) = \begin{cases} 1 & \text{if } x \text{ is } q_0 \\ 0 & \text{otherwise} \end{cases}$$

Let $Q_F = \{q_{i1}, q_{i2}, \dots, q_{ik}\}$ then $M_F = \{M_{F1}, M_{F2}, \dots, M_{Fk}\}$ where

$$M_{Fj}(x) = \begin{cases} 1 & \text{if } x \text{ is } q_{ij} \\ 0 & \text{otherwise} \end{cases}, \quad 1 \leq j \leq k.$$

$$f(a) = \text{True } \forall a \in A$$

We also make the following assumptions about the environment in which this net is placed. At any stage, the environment signals an input a_i by placing a marker in place a_i . At some later stage, N signals an output b_j by placing a marker in place b_j . This is noted by the environment which removes the marker. Only when the marker is removed does the environment provide the next input to N.

Lemma 1

If

S is in state q_c , receives an input a_d , goes to state q_e and outputs b_f .

then

if N has marking M^i where

$$M^i(x) = \begin{cases} 1 & \text{if } x \text{ is } q_c \\ 0 & \text{otherwise} \end{cases}$$

and receives an input a_d

then N can only output b_f and when the output is consumed by the environment, marking M^{i+1} results where

$$M^{i+1}(x) = \begin{cases} 1 & \text{if } x \text{ is } q_e \\ 0 & \text{otherwise} \end{cases}$$

Proof

If S in state q_c receives an input a_d and goes to state q_e outputting b_f , then $\delta((q_c, a_d)) = (q_e, b_f)$. If N has marking M^i and receives an input a_d , then marking M' results where

$$M'(x) = \begin{cases} 1 & \text{if } x \text{ is } q_c \text{ or } a_d \\ 0 & \text{otherwise} \end{cases}$$

By construction, the input places of transition t_{cd} are places q_c and a_d .

Therefore t_{cd} can be enabled in M' . Assume that transition t_{ij} , $i \neq c$ or $j \neq d$ can be enabled in M' . Input places of t_{ij} are q_i and a_j . But q_c and a_d are the only places having markers. Therefore $i = c$ and $j = d$. Also, $M'(p_{ij}) = 0$, $\forall p_{ij}$ and since each $\bar{t}_{ij}$ has exactly one input place, i.e., p_{ij} , no $\bar{t}_{ij}$, $0 \leq i \leq n$, $1 \leq j \leq m$ can be enabled in M' . Therefore t_{cd} is the only transition that can be enabled. Firing of t_{cd} results in M'' , where

$$M''(x) = \begin{cases} 1 & \text{if } x \text{ is } p_{cd} \\ 0 & \text{otherwise} \end{cases}$$

$\bar{t}_{cd}$ is the only transition that can be enabled in M'' . When $\bar{t}_{cd}$ fires, it results in M''' where

$$M'''(x) = \begin{cases} 1 & \text{if } x \text{ is } q_e \text{ or } b_f \\ 0 & \text{otherwise} \end{cases}$$

This is because $\delta((q_c, a_d)) = (q_e, b_f)$ and by the rules for constructing A , $(\bar{t}_{cd}, q_e) \in A$ and $(\bar{t}_{cd}, b_f) \in A$. The appearance of a marker in b_f signals the output. When the environment removes the marker, M^{i+1} results where

$$M^{i+1}(x) = \begin{cases} 1 & \text{if } x \text{ is } q_e \\ 0 & \text{otherwise} \end{cases}$$

Q. E. D.

Initially, S is in state q_0 and N has marking M^0 , where

$$M^0(x) = \begin{cases} 1 & \text{if } x \text{ is } q_0 \\ 0 & \text{otherwise} \end{cases}$$

By repeated application of the lemma we see that for any input sequence $i_1 i_2 \dots, i_n$ if S outputs $o_1 o_2 \dots, o_n$ and ends in state q , then N can output only $o_1 o_2 \dots, o_n$ and when o_n is consumed by the

environment, the resulting marking is M^n where

$$M^n(x) = \begin{cases} 1 & \text{if } x \text{ is } q \\ 0 & \text{otherwise} \end{cases}$$

Thus N replicates the behavior of S on a step by step basis and this proves the theorem.

Q. E. D.

While proving theorem 1, we have also proved the following: no simulation of N results in a marking at any stage that assigns more than one marker to any place. N is thus a "safe" net and we get the following stronger result.

Theorem 2

Safe Petri nets can simulate finite state machines.

Example

2. Let $S = \{Q, I, O, \delta, q_0, Q_F\}$ be a finite state machine where

$$Q = \{q_0, q_1\}$$

$$I = \{a_1, a_2\}$$

$$O = \{b_1, b_2\}$$

$$\delta((q_0, a_1)) = (q_1, b_2)$$

$$\delta((q_0, a_2)) = (q_0, b_1)$$

$$\delta((q_1, a_1)) = (q_0, b_1)$$

$$\delta((q_1, a_2)) = (q_1, b_2)$$

$$Q_F = \{q_1\}$$

The state diagram of the finite state machine is given in fig. 3. The quantity on the left of the "bar" is the input and on the right, the output. The Petri net that simulates the behavior of the above finite state machine on a step by step basis is given in fig. 4. The reason for introducing places p_{ij} and transitions $\bar{t}_{ij}$ is the fact that a place can-

not be an input place and an output place of the same transition. Thus $\delta((q_0, a_2)) = (q_0, b_1)$ could be represented as in fig. 5. This, however, violates the above requirement.

Theorem 3

Turing machines can be simulated using extended Petri nets.

Proof

Fig. 6 represents a Turing machine. A Turing machine consists of a finite state machine connected to an infinite tape by means of a read/write head. The tape is divided into squares and each square contains a symbol chosen from a finite alphabet. In any one step, the finite state machine does the following:

It reads the current square under the head. Depending on the symbol read and the current state it proceeds in one of three ways.

(1) It prints a symbol on the current square, moves the head left or right one square or leaves it where it is, and goes into a new state.

Or

(2) It decides to go into a special HALT state.

Or

(3) It assumes there is an "oracle" in its environment and asks the oracle whether the present contents of the tape belong to a given set A. It then goes into one of two states depending on the answer it receives.

A Turing machine with the first two options only, correctly captures the intuitive notion of "algorithm" or "effective computability". Thus we will disallow the third alternative and assume that the Turing machine

operates in a closed manner and does not interrogate its environment.

A well known result of automata theory is that a Turing machine with only two tape symbols can simulate a Turing machine with any finite number of tape symbols [26]. Without loss of generality, we will assume that the two symbols are 0 and 1. The contents of the left and right part of the tape can be viewed as two binary numbers. The number to the left of the read/write head is

$$L = \sum_{i=0}^N \ell_i 2^i$$

where ℓ_i is the bit in the $(i+1)$ st square to the left of the read/write head. The number to the right of the read/write head is

$$R = \sum_{j=0}^M r_j 2^j$$

where r_j is the bit in the $(j+1)$ st square to the right of the read/write head.

Let us concentrate for the moment on what happens to L and R on a move left. At some stage, let fig. 7 represent the tape configuration. Here, $L = 3$, $R = 2$, CS (= bit in current square) = 1. The configuration after a shift left is given in fig. 8. Here, $L = 1$, $R = 5$, $CS = 1$. It is easy to see that a shift left corresponds to the following transformations of L , R , CS .

$$L \leftarrow Q(L/2)$$

$$R \leftarrow 2R + CS$$

$$CS \leftarrow 2[L/2 - Q(L/2)]$$

where $Q(x)$ is the largest integer less than or equal to x . Similarly a shift right implies the following transformations:

$$R \leftarrow Q(R/2)$$

$$L \leftarrow 2L + CS$$

$$CS \leftarrow 2[R/2 - Q(R/2)]$$

We will now show how extended Petri nets can be used to perform these transformations. The left hand part of the tape, the right hand part and the current square correspond to places ℓ , r , and cs . L , R , and CS are the number of markers in places ℓ , r , and cs respectively. Let us consider some nets we will need to perform the transformations.

Fig. 9 is the block diagram of a "dividing" net. The dotted lines indicate some connection between the net and the places x , START DIV, DIV COMP, REM 0, and REM 1. The div net behaves as follows: Initially all places in the net are empty, $X \geq 0$, START DIV, DIV COMP, REM 0 and REM 1 are empty and no transition can be enabled. At some stage the environment places one marker in START DIV. After some finite time, markers appear in REM 0, REM 1 and DIV COMP. A marker in REM 0 or REM 1 indicates the remainder on dividing X by 2. (If a lower case letter denotes a place the corresponding capital denotes the number of markers in the place). When a marker appears in DIV COMP, it signifies completion of division: The number of markers in x is $Q[(\text{number of markers in } x \text{ initially})/2]$, the net has returned to its original state and there is no marker in START DIV. We assume that the environment does not disturb x once a marker is placed in START DIV and a marker is placed in START DIV only when there are no markers in START DIV, DIV COMP, REM 0 and REM 1.

Lemma 2

The net in fig. 10 has the desired behavior of a div net.

Proof

Initially all places in the net are empty and an examination reveals that no transition can be enabled. When a marker is placed in START DIV,

t_1 is the only transition that can fire. When it does, a marker is placed in p_1 and the marker in START DIV is removed. Since no transition in the net has START DIV as an output place, the net will not place a marker in START DIV. $P_1 = 1$ at this stage.

Case 1

$$X = 0$$

With $P_1 = 1$ and $X = 0$, the only transition that can be enabled is t_7 . When it fires, a marker is placed in REM 0 and one in p_4 . Now, the only transition that can be enabled is t_6 . When it fires a marker is placed in DIV COMP and the marker in p_4 is removed. x is left undisturbed. Therefore when a marker appears in DIV COMP, $X = 0$, the net is in its original state with all places empty, and there is a marker in REM 0 signaling a zero remainder.

Case 2

$$X = 2n, \quad n = 1, 2, 3, \dots$$

With $P_1 = 1$, t_2 is the only transition that can fire. When it does, a marker is removed from x and p_1 and a marker placed in p_2 . t_3 is the only transition that can now fire. When it does, another marker is removed from x , a marker is placed in p_3 and one in p_1 . The cycle of firing t_2 followed by t_3 takes place n times. Each time two markers are removed from x and one marker placed in p_3 . At the end, $X = 0$, $P_1 = 1$, $P_3 = n$, and all other places in the net are empty. The only transition that can now fire is t_7 . Firing of t_7 causes a marker to be placed in REM 0 and one in p_4 . All other places in the net are empty except p_3 which at this stage contains the quotient n . The contents of p_3 must be transferred to x . This is achieved by repeating the firing

of t_4 followed by t_5 n times. No other transition can fire during this time. In each cycle one marker is removed from p_3 and one marker placed in x . At the end, $X = n$ and $P_4 = 1$. t_6 can now fire signaling DIV COMP. When a marker appears in DIV COMP, $X = n$, $REM\ 0 = 1$ and the net is in its original state.

Case 3

$$X = 2n + 1, n = 0, 1, 2, 3, \dots$$

If $n = 0$, i.e., $X = 1$, t_2 is the only transition that can fire. Firing of t_2 causes a marker to be removed from x and p_1 so that $X = P_1 = 0$ and a marker is placed in p_2 . t_8 can now fire (no other transition can). It removes the marker from p_2 and places one in $REM\ 1$ and one in p_4 . All places in the net except for p_4 are empty. The only transition that can fire is t_6 and it places a marker in DIV COMP. At this stage $REM\ 1$ has a marker, $X = 0$ and the net is in its original state. For $n > 0$, $P_1 = 1$ the sequence t_2 followed by t_3 is repeated n times and this is the only activity possible. In each cycle, two markers are removed from x and one placed in p_3 . At the end, $X = 1$, $P_1 = 1$ and $P_3 = n$. t_2 is now the only transition that can fire. When it does, a marker is placed in p_2 and one marker removed from x and p_1 . At this stage $X = 0$. t_8 is enabled and when it fires a marker is removed from p_2 , one marker is placed in p_4 and one in $REM\ 1$. The appearance of a marker in p_4 indicates that the remainder has been signaled. The markers in p_3 ($P_3 = n$) now have to be transferred to x . This is achieved by the firing sequence t_4 followed by t_5 repeated n times. At the end, $X = n$, $P_4 = 1$ and all other places in the net are empty. t_6 can now

be enabled. Firing of t_6 returns the net to its original state and a marker is placed in DIV COMP. At this stage, REM 1 has a marker and $X = n$. This completes the proof of the lemma.

Q.E.D.

Fig. 11 is the block diagram of a "multiplying net." The dotted lines denote connections to the net. The behavior of the mult net is as follows: Initially all places in the net are empty, $X \geq 0$ and MULT COMP is empty. The environment places a marker in START MULT. At some later time a marker appears in MULT COMP. This signals the following: The number of markers in x has been doubled, the net is restored to its original state and START MULT is empty. We assume again that the environment does not disturb x during the operation of the net and that a marker is placed in START MULT only when START MULT and MULT COMP are empty.

Lemma 3

The net in fig. 12 has the desired behavior of the mult net.

Proof

Initially all places in the net are empty and an examination reveals that no transition can fire. When a marker is placed in START MULT, t_1 is the only transition enabled and when it fires it removes the marker from START MULT and places one in p_1 . Since START MULT is not the output place of any transition in the net, the net will not place a marker in START MULT.

Case 1

$X = 0$.

With $P_1 = 1$, the only transition that can be enabled is t_8 . When it fires, a marker is placed in p_8 and a marker is removed from p_1 . At this stage $P_8 = 1$ and all other places in the net are empty. t_9 is now the only transition that can fire and when it does, the net is returned to its original state and a marker is placed in MULT COMP signaling completion. $X = 0$ initially and remains 0.

Case 2

$X = n$, $n = 1, 2, \dots$

With $P_1 = 1$, the only transition that can fire is t_2 . On completion of firing it places one marker in p_2 and one in p_3 removing one marker from x . The only enabled transitions are now t_3 and t_4 . They fire in some order. Each places a marker in p_4 . t_7 "waits" till each has completed firing. When t_7 fires, a marker is replaced in p_1 . The above sequence has the following effect: one marker is removed from x and two markers placed in p_4 . This sequence is repeated n times and is the only activity possible. At the end, $X = 0$, $P_4 = 2n$, $P_1 = 1$ and all other places are empty. $P_4 = 2n$ is signaled by the firing of t_8 which removes the marker from p_1 and places a marker in p_8 . The markers in p_4 now have to be transferred to x . This is achieved by repeating the sequence of firing t_5 followed by t_6 $2n$ times. Once again, this is the only activity possible. On completion of this activity $X = 2n$, $P_8 = 1$ and all other places are empty. t_9 is the only transition that can fire. When it does a marker is placed in MULT COMP signaling completion of multiplication. At this stage $X = 2n$ and the net has been restored to its initial state. This completes the proof of the lemma.

Q.E.D.

We have already proved that any finite state machine can be simulated using a safe Petri net. The finite state part of a Turing machine can be described in terms of quintuples of the form given below:

$$(1) (q_i, a_i, b_i, s, q_j)$$

$$(2) (q_i, a_i, H, -, -)$$

where q_i, q_j are states of the finite state machine, $a_i \in \{0, 1\}$, $b_i \in \{0, 1\}$, $s \in \{L, R, N\}$, H is the halt state. The quintuples have the following meaning:

(1) If the finite state machine is in state q_i , scanning a_i then it writes b_i , moves left if s is L , right if s is R , does not move if s is N and goes into state q_j .

(2) If the finite state machine is in state q_i , scanning a_i then it goes in the HALT state and stops.

Consider a particular quintuple of the first kind: $(q_1, 0, 1, L, q_2)$. Let fsm net denote the Petri net that simulates the behavior of the finite state machine. The part of the fsm net that simulates the above quintuple is given in fig. 13. A particular quintuple of the second kind is: $(q_1, 1, H, -, -)$. The part that simulates this is given in fig. 14. Given all the quintuples, the entire fsm net can be constructed. The outputs of this net are WRITE 1, WRITE 0, SHIFT L, SHIFT R, SHIFT N and READ cs. The inputs are 0 READ, 1 READ, WRITE COMP and SHIFT COMP. The outputs of the fsm net are the inputs of its environment and the inputs of the fsm net, the outputs of its environment. When the fsm net places a marker in WRITE 1, it waits till the environment signals completion by placing a marker in WRITE COMP. A

SHIFT L or SHIFT R is acknowledged by the environment placing a marker in SHIFT COMP and a READ cs is acknowledged by placing a marker in 0 READ or 1 READ depending on the value of CS (i.e., the bit in the current square being scanned). The fsm net is represented as shown in fig. 15.

The three nets, div, mult and fsm will be used to simulate the Turing machine. Recall that l , r , and cs represent the left portion, right portion and current square respectively. In what follows we will discuss different portions of the overall net, N . The net in fig. 16 represents the "read" part of N . When the fsm net desires to read the bit in cs it places a marker in READ cs and waits for an answer from the environment. If $CS = 1$, t_1 can fire causing a marker to be placed in p_1 and a marker is removed from cs and READ cs . No other transition can fire. To keep the readout non-destructive, a marker should be replaced in cs . This is achieved by the firing of t_2 which places a marker in cs and one in 1 READ. If cs was empty when a marker appeared in READ cs , t_3 would be enabled and firing of t_3 would cause a marker to be removed from READ cs and a marker would be placed in 0 READ.

When a marker appears in 0 READ or 1 READ, the fsm net removes this marker, places one in WRITE 0 or WRITE 1 and awaits completion of writing. The portion of N that achieves the writing is given in fig. 17.

Let us consider the four cases separately.

Case 1

One marker in WRITE 0 and $CS = 0$. The only transition that can fire

is t_1 . When it does, a marker is removed from WRITE 0 and a marker is placed in WRITE COMP. cs remains empty.

Case 2

One marker in WRITE 0 and CS = 1. The only transition that can fire is t_2 . When it does, the marker is removed from cs and "writing complete" is signaled by placing a marker in WRITE COMP. Thus CS is changed from 1 to 0.

Case 3

One marker in WRITE 1 and CS = 0. t_5 is the only transition that can fire. When it does, t_6 is enabled and firing of t_6 causes a marker to be placed in cs and one in WRITE COMP. Thus CS goes from 0 to 1.

Case 4

One marker in WRITE 1 and CS = 1. The only transition that can fire is t_3 . Firing of t_3 enables t_4 and removes the marker from cs. Firing of t_4 replaces a marker in cs and signals "writing complete." Thus CS remains 1.

Having received the "write" acknowledgment the finite state machine decides to move the head left or right or to keep it where it is. These are signaled respectively by placing a marker in SHIFT L, SHIFT R or SHIFT N. Fig. 18 gives the net that achieves a "left shift."

When a marker is placed in SHIFT L, t_1 is the only transition that can fire. When it does, it places a marker in START DIV and START MULT, the marker in SHIFT L being removed. This causes two activities to start. Recall that l , r and cs represent the left part of the tape, the right part and current square respectively. As we have

already shown, the action of the div net has the following results:

- (1) The marker in START DIV is removed.
- (2) $L \leftarrow Q(L/2)$.
- (3) A marker is placed in REM 0 if $2[L/2 - Q(L/2)] = 0$
A marker is placed in REM 1 if $2[L/2 - Q(L/2)] = 1$.
- (4) Completion of division is signaled by a marker placed in DIV COMP.

The action of mult net has the following result:

- (1) The marker in START MULT is removed.
- (2) $R \leftarrow 2R$
- (3) Completion of multiplication is signaled by a marker placed in MULT COMP.

The bit in cs has to be added to R before cs is updated. Thus when multiplication is completed if $CS = 1$, t_3 fires causing a marker to be placed in r and one in p_1 . The marker in cs is removed. If $CS = 0$, t_2 fires causing a marker to be placed in p_1 and leaving r undisturbed. Note that in either case, $CS = 0$ when a marker appears in p_1 . At this stage the transformation $R \leftarrow 2R + CS$ has been completed. cs can now be updated. When a marker appears in DIV COMP and $P_1 = 1$, cs is ready to be updated. If the remainder of the division was 1, t_5 fires causing a marker to be placed in cs so that $CS = 1$. If the remainder was 0, cs is left undisturbed and t_4 fires. In either case a marker is placed in SHIFT COMP. At this stage the following transformations have been completed.

- (1) $L \leftarrow Q(L/2)$
- (2) $R \leftarrow 2R + CS$

$$(3) \quad CS \leftarrow [L/2 - Q(L/2)]$$

and the net is restored to its original state.

Fig. 19 gives the net that achieves a "right shift." The shift net in fig. 19 is identical to the shift net in fig. 18 with l and r interchanged. Thus when a marker is placed in SHIFT R, the shift net starts its operation, the marker in SHIFT R is removed and after a finite time a marker appears in SHIFT COMP and the shift net is returned to its original state. This signals the completion of the following transformations:

$$(1) \quad R \leftarrow Q(R/2)$$

$$(2) \quad L \leftarrow 2L + CS$$

$$(3) \quad CS \leftarrow 2[R/2 - Q(R/2)].$$

Fig. 20 gives the net that achieves a "no shift."

We have shown how the finite state part can be simulated using a safe Petri net and have given the extended Petri nets that simulate the various operations such as read, write and shift. In each case, we have shown that the nets operate as desired. The fsm net and its environment operate in "synchronization". When the fsm desires to read, write or shift, a signal is given to the environment, the appropriate sub net performs the desired operation and on completion gives a signal to the fsm net. Each sub net is designed so that whenever a completion signal is given, the sub net is in its original state and ready to service another request. Only one sub net is in operation at any time and is not disturbed by anything external to it while in operation. Thus when the nets in figs. 14 through 20 are put together to form one big extended Petri net N , we are guaranteed that N will

simulate the behavior of the given Turing machine. This completes the proof of Theorem 3.

Q.E.D.

Theorem 4

Any coordination which can be represented by the Turing machine model can also be represented by an extended Petri net with a suitable trans-labeling.

Proof

We have shown how extended Petri nets can simulate Turing machines. We have only to show how the processes $P_1, \dots, P_n$ communicate with the central controller. Let the extended Petri net that simulates a given Turing machine central controller be C . In the Turing machine model there were two states q_f and q_s . The central controller serviced requests only when in one of these states. The finite state machine started in state q_f or q_s , serviced a particular request and reentered q_f or q_s only when the condition checking was complete. q_f and q_s will be represented in the extended Petri net as two places labeled q_f and q_s respectively. In the Turing machine model we have FIFO queues for the requests for service. However, as pointed out earlier, the important point is that the condition checking occur serially. Fig. 21 indicates how the serial checking is enforced. The connections are given for any process P_i . Initially $P_i^1 = 1$. If P_i desires to proceed, this is signaled by the firing of t_i^5 which places a marker in r_i . Whenever C is ready to accept requests, either $Q_f = 1$ or $Q_s = 1$. If $Q_f = 1$, then t_i^3 can fire, placing a marker in x_i . This is a signal

for C to begin checking if P_i can be enabled. When a decision is made a marker is placed in e_i or d_i depending on whether P_i can be enabled or not. At the same time, a marker is placed in q_s or q_f . If a marker is placed in e_i , then transition t_i^0 (which represents process P_i) can fire. When it completes firing a marker is placed in f_i signaling completion and another in p_i^1 . If a marker is placed in d_i , then t_i^0 can fire placing a marker in p_i^1 . In this case, t_i^0 is not enabled. With $P_i^1 = 1$, the entire process can be repeated again. A similar sequence takes place if $Q_s = 1$ instead of Q_f . To see that condition checking occurs serially note the following:

- (1) Q_s and Q_f are not 1 at the same time (by assumption).
- (2) Q_s or Q_f are 1 only at the end of condition checking (by assumption).
- (3) Either $R_i = 1$ or $F_i = 1$, $i \in \{1, 2, \dots, n\}$ where n is the number of processes.
- (4) Let $R_i = 1$ for $1 \leq i \leq a$, $F_i = 1$ for $a + 1 \leq i \leq b$, and $R_i = F_i = 0$ for $b < i \leq n$ at some stage
 - (i) If $Q_s = Q_f = 0$, none of the transitions t_i^j , $1 \leq i \leq n$, $1 \leq j \leq 4$ can fire. Therefore a marker is not placed in x_i or y_i , $1 \leq i \leq n$.
 - (ii) If $Q_s = 1$ and $Q_f = 0$, then the transitions t_i^2 , $1 \leq i \leq a$ and the transitions t_i^1 , $a < i \leq b$ will be in conflict and only one can fire. Firing of this transition will disable all other t_i^j 's $1 \leq j \leq 2$. Thus only one of the x_i 's or y_i 's would receive a marker, i.e., only one of the requests for service reaches C.
 - (iii) If $Q_s = 0$ and $Q_f = 1$ then transitions t_i^3 , $1 \leq i \leq a$ and transitions t_i^4 , $a + 1 \leq i \leq b$ would be in conflict, only one would fire and

again only one request for service would reach C.

If the complete extended Petri net which simulates a given Turing machine controller is $N = (T, P, A, M^0, M_F, f)$, then T, P, A, M^0, f have already implicitly been defined in theorems 3 and 4. We must still define M_F and the trans-labeling I .

$M_F = \{M_{F1}\}$ where

$$M_{F1}(x) = \begin{cases} 1 & \text{if } x \text{ is } q_f \\ 1 & \text{if } x \text{ is } p_i^1, 1 \leq i \leq n \\ d & \text{otherwise} \end{cases}$$

To the external observer only the firings of $t_i^0, 1 \leq i \leq \ell$ are of importance, since these are the transitions that represent the processes coordinated by the Turing machine central controller. Therefore,

$$I(x) = \begin{cases} P_i & \text{if } x \text{ is } t_i^0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This completes the construction of the extended Petri net which represents a coordination specified by some Turing machine central controller. Thus for every coordination system with a Turing machine central controller, an extended Petri net with a suitable trans-labeling can be constructed which represents the same coordination.

Q.E.D.

Theorem 5

For every coordination that can be represented by a trans-labeled extended Petri net, a Turing machine central controller can be constructed that provides the same coordination.

Proof

Let $N = (T, P, A, M^0, M_F, f)$ be any extended Petri net and let I be some trans-labeling of N . In the description of the simulation, we said that each time a transition desires to fire it checks the input places. With a central controller we assume as before that each time a transition desires to fire it places a request in a queue. It is easy to keep the description of N on the infinite tape in some coded form. The number of markers in each place at any instant is also stored. The simulation rules can be built into the finite state machine. Whenever the fsm receives a request from t_i to be enabled it checks the variables representing the input places of t_i to determine whether t_i can be enabled. If it can, an e_i signal is sent to t_i and the appropriate variables decremented. If t_i cannot be enabled it sends a signal d_i and leaves the variables unchanged. It is now ready to service the next request. When t_i receives an e_i signal, if t_i has an associated process P_j specified by the trans-labeling, this process is activated. When P_j completes, t_i sends an f_i signal to the fsm machine. On receiving this signal the fsm increments the variables representing the output places of t_i and readies itself for the next request. The finite state machine can also be programmed to select a request from the queue at random if that is desired.

Q.E.D.

5. THE SYNCHRONIZING SYSTEM (SS) MODEL

Motivations

We have described a coordination system and have presented two models, the Turing machine model and the extended Petri net model both of which are uninterpreted and equivalent with respect to their capabilities for representing coordinations. Even so, a model more amenable to mathematical analysis would be desirable.

As we mentioned in the introduction, a number of synchronizing primitives have been proposed to coordinate processes. Each of these primitives viewed as a model for synchronization is a special case of the SS model. The SS model is so general that it appears to be complete, i.e. appears to provide a formal counterpart to the notion of coordination. All existing models can now be viewed as special cases of one of the three models presented in this report.

In a subsequent report we will develop necessary and sufficient conditions to determine whether a model obtained from the SS model as a special case, is complete or not.

In this section, we will formally define the SS model and prove that it is equivalent to the other two models. Since further results on this model will not be presented here, the motivation for this model may not be fully apparent to the reader. However, for the sake of completeness, we will discuss the model in this report.

We described a coordination system as a group of processes placed in a certain environment. Whether a process was allowed to run or not was based on certain specified conditions being satisfied. Let us assume that the conditions can be specified as relations between integer

valued variables. Since the model is purely a mathematical tool and any coordination system can be characterized by the set of process sequences it produces, we assume that only one process at a time is allowed to run. The system makes a note of the enabling, running and completion of a process by changing the values of the variables in some specified manner. More formally,

Definitions

16. A synchronizing system, SS, is a sextuple $(X, C, T, J^{\circ}, J_F, Q)$

where:

$X = \{x_i | 1 \leq i \leq n\}$ is a finite set of integer valued variables.

$C = (C_1, C_2, \dots, C_p)$ is a finite sequence of conditions. Each condition is a 0/1 valued computable function on X . A condition C_i holds if and only if the result of applying C_i to X is 1.

Example: $C_1 = \{x_1 + x_3 \geq 3, x_2 = 2\}$. (Specified in this manner, the 0/1 valued computable function $C_i(X)$ is to be understood as: $C_i(X) = 1$ if $x_1 + x_3 \geq 3$ and $x_2 = 2$, otherwise $C_i(X) = 0$).

$T = (t_1, t_2, \dots, t_p)$ is a finite sequence of transformations. Each transformation is a set of assignments and gives new values to some members of X . The assignments could be arbitrary computable functions and are performed simultaneously. Example: $\{t_1 = x_1 + x_2 + x_3, x_3 \leftarrow x_1\}$.

$J^{\circ} : X \rightarrow \{\dots, -2, -1, 0, 1, 2, \dots\}$ is an initial assignment of values to elements in X .

$J_F = \{J_{F_i} | 1 \leq i \leq m\}$ where $J_{F_1} : \{\dots, -1, 0, 1, \dots\} \cup \{d\}$.

$Q : T' \rightarrow E$ where $T' = \{t_{a_1}, t_{a_2}, \dots, t_{a_r}\} \subseteq T$ and $E = \{E_1, E_2, \dots, E_k\}$ is a finite set of process names, $k \geq 1$. If $Q(t_i) = E_j$, we say that E_j is associated with t_i .

17. The values of elements in X can be specified at any stage i by a function J^i where the value of x_j is $J^i(x_j)$, $1 \leq j \leq n$. J^i is called the state of SS at stage i .

18. The state of SS at stage j is said to be covered by J_{F_i} if $J^i(x_k) = J_{F_i}(x_k)$, $k \in \{1, 2, \dots, n\} \Rightarrow J_{F_i}(x_k) \neq d$.

A simulation starts at stage 0 with SS in state J^0 and proceeds as follows: At any stage i , suppose conditions $C_{a_1}, C_{a_2}, \dots, C_{a_s}$ hold. Select one transformation t_j , $j \in \{a_1, a_2, \dots, a_s\}$ arbitrarily and activate the associated process P . (Note that P does not affect X in any way). On completion of P , change the values of elements in X in accordance with the assignments specified by t_j . Start stage $i + 1$. If there is no associated process, only assign new values and start stage $i + 1$.

Note: SS only specifies a control C and a set of processes to be controlled. The action of each process is unspecified and no I/O data sets are defined for the processes. Thus, this is also an uninterpreted model.

Definitions

19. A sequence of transformations $\bar{t} = (t_{a_1}, t_{a_2}, \dots, t_{a_k})$ is called a simulation sequence of SS if some simulation t_{a_ℓ} was selected at stage $\ell - 1$ of the simulation, $1 \leq \ell \leq k$ and the state of SS at stage k is J^k where J^k is covered by some $J_{F_i} \in J_F$.

20. The set of all simulation sequences of SS is called $\text{SIMSET}_{\text{SS}}$.

21. Corresponding to each simulation sequence $\bar{t} = (t_{a_1}, \dots, t_{a_k})$ of SS, we define a reduced simulation sequence $\bar{t}' = (t_{b_1}, t_{b_2}, \dots, t_{b_\ell})$ where $\bar{t}'$ is obtained from $\bar{t}$ by deleting all those transformations with

no associated processes.

22. Corresponding to each reduced simulation sequence $\bar{t}' = (t_{b_1}, \dots, t_{b_\ell})$ of SS we define a process simulation sequence $\bar{P} = (P_{c_1}, \dots, P_{c_\ell})$ where P_{c_i} is the process associated with t_{b_i} , $1 \leq i \leq \ell$.

23. The set of all process simulation sequences of SS is called $P(\text{SIMSET}_{SS})$.

The coordination represented by an SS is described by specifying $P(\text{SIMSET}_{SS})$ or by defining X, C, T, J°, J_F, Q .

Example

3. We will represent the coordination desired in the two producer, two consumer, two buffer problem of example 1 using this model.

Let $SS_1 = (X, C, T, J^\circ, J_F, Q)$ be the desired system.

$$X = \{x_1, x_2\}$$

$$C = (\{\text{True}\}, \{\text{True}\}, \{x_1 > 0\}, \{x_2 > 0, x_1 = 0\})$$

$$T = (\{x_1 \leftarrow x_1 + 1\}, \{x_2 \leftarrow x_2 + 1\}, \{x_1 \leftarrow x_1 - 1\}, \{x_2 \leftarrow x_2 - 1\})$$

$$J^\circ = \{(x_1, 0), (x_2, 0)\}$$

$$J_F = \{(x_1, d), (x_2, d)\}$$

$$Q(t_1) = R_1, Q(t_2) = R_2, Q(t_3) = C_1, Q(t_4) = C_2$$

$C_1 = C_2 = \{\text{True}\}$. Thus t_1 and t_2 can be selected at any stage in any simulation. Selection of t_1 corresponds to the activation of R_1 . When R_1 completes, x_1 is increased by 1. t_3 can be selected only if $x_1 > 0$. Selection of t_3 corresponds to the activation of C_1 . When C_1 finishes consuming, x_1 is

decreased by 1. Thus x_1 gives the number of items in B_1 , and C_1 can consume only when B_1 is not empty. Similarly, x_2 gives the number of items in B_2 , but C_2 can consume only if B_1 is empty and B_2 is not empty. This is because C_2 is associated with t_4 and t_4 can be selected only when $\{x_2 > 0, x_1 = 0\}$.

Theorem 6

The coordination represented by any SS can be represented by an extended Petri net with a suitable trans-labeling.

Proof

By itself, this theorem may be difficult to prove. However, most of the work has already been done in the proof of theorem 3. It is easy to see how a Turing machine central controller could be constructed to provide the coordination represented by any SS. The description of the SS could be stored on the infinite tape together with the current values of the variables x_i . A nondeterministic finite state machine could then be programmed to behave as follows: At each stage i the fsm examines the conditions $C_1, C_2, \dots, C_p$ and determines which ones hold. Of these, it arbitrarily selects one, say C_j .

If process P is associated with t_j , P is enabled. When P completes, the fsm performs the transformation t_j on X . It then repeats the above procedure. If no process is associated with t_j the fsm does not enable any process, performs the transformation t_j and proceeds to the next stage.

For every nondeterministic finite state machine an equivalent deterministic finite state machine can be constructed. By theorem 3,

the Turing machine central controller can be simulated by an extended Petri net with a suitable trans-labeling. This trans-labeled net represents the same coordination as the original SS.

Q.E.D.

Theorem 7

The coordination represented by any trans-labeled extended Petri net can be represented by an SS.

Proof

Let $N = (T, P, A, M^0, M_F, f)$ be an extended Petri net where

$$T = \{t_i \mid 1 \leq i \leq n\}$$

$$P = \{p_i \mid 1 \leq i \leq m\}$$

$$M_F = \{M_{F_i} \mid 1 \leq i \leq p\}$$

Let $I[T', E]$ be a trans-labeling of N where

$$T' = \{t_{b_1}, t_{b_2}, \dots, t_{b_k}\} \subseteq T$$

$$E = \{P_1, P_2, \dots, P_\ell\}, \ell \leq k.$$

We construct an SS = (X, C, T, J^0, J_F, Q) as follows:

$$X = \{x_i \mid 1 \leq i \leq m\}$$

$$C = (C_1, C_2, \dots, C_n) \text{ where for } 1 \leq j \leq n, 1 \leq i \leq m$$

$$x_i = 0 \in C_j \text{ if and only if } p_i \in I_j'' \text{ (see definition 5)}$$

$$x_i > 0 \in C_j \text{ if and only if } p_i \in I_j' \text{ (see definition 4)}$$

$$T = (\ell_1, \ell_2, \dots, \ell_n) \text{ where for } 1 \leq j \leq n, 1 \leq i \leq m$$

$$x_i + x_i + 1 \in \ell_j \text{ if and only if } p_i \in O_j \text{ (see definition 3)}$$

$$x_i + x_i - 1 \in \ell_j \text{ if and only if } p_i \in I_j'.$$

$$J^0(x_i) = M^0(p_i), 1 \leq i \leq m$$

$J_F = \{J_{F_i} | 1 \leq i \leq p\}$ where for $1 \leq i \leq p$

$J_{F_i}(x_j) = M_{F_i}(p_j), 1 \leq j \leq m.$

$Q(\ell_j) = P_i$ if and only if $I(t_j) = P_i$

We state without proof the following lemma:

Lemma

4. A sequence $(t_{a_1}, t_{a_2}, \dots, t_{a_n}) \in \text{SIMSET}_N$ if and only if the sequence $(\ell_{a_1}, \ell_{a_2}, \dots, \ell_{a_n}) \in \text{SIMSET}_{SS}.$

The association of process names $P_i, 1 \leq i \leq \ell$ to the transformations ℓ_j allows us to conclude that

$I(\text{SIMSET}_N) = P(\text{SIMSET}_{SS})$

Thus the coordination represented by SS is the same as that represented by N.

Q.E.D.

Theorems 4, 5, 6 and 7 lead to the following important result.

Theorem 8

The Turing machine model, the extended Petri net model and the SS model are equivalent with respect to their capabilities for representing the coordination of asynchronous processes.

6. CONCLUSIONS

We have tried to formalize the notion of a coordination system in this report. A model for representing coordinations is complete if it can represent all possible types of interaction between a group of processes, i.e. if it provides a formal counterpart to the notion of coordination. We have presented three models, each arising out of different motivations and each apparently complete. The Turing machine model appears to be complete because of the close relation between the notions of "algorithm" and "coordination". A separate study performed earlier indicated that the

extended Petri net model may be complete. Starting from the intuitive description of a coordination system we defined the SS model. This model also provides a generalization of all existing synchronizing primitives. Its generality indicated that it might also be complete.

Theorem 8 establishes the equivalence of these three independently motivated, apparently complete, models.

We now have two alternatives open to us. On the basis of the above argument, we can claim that there is sufficient reason to believe that each of the three models proposed is complete. Or, we can formally define a coordination system to be one composed of a set of processes controlled by a Turing machine central controller. The Turing machine model then becomes complete by definition, and the extended Petri net and SS models, by equivalence.

The difference between the two alternatives is not very significant. We tend to favor the first one and will conclude that each of the three models presented in this report is complete and provides a formal counterpart to the vague notion of coordination.

The model referred to in the title of this paper is the extended Petri net model and is the one we are most interested in. This model provides a convenient tool to study concurrent systems. Coordinations can easily be represented and the nets are easy to understand. In a subsequent report, we will discuss the following:

- (1) Incompleteness of existing models.
- (2) Usefulness of the extended Petri nets in modeling operating systems and program constructs.
- (3) New complete high level language primitives derived from the model and their implementation.
- (4) Impact of the model on multiprocessor system design.

ACKNOWLEDGEMENT

I would like to thank Professor M. J. Flynn for his continued guidance, encouragement and support and for numerous suggestions and discussions during the course of this research.

Figure 1

Figure 2

Figure 3

Figure 4

Figure 5

Figure 6

Figure 7

Figure 8

Figure 9

Figure 10

Figure 11

Figure 12

Figure 13

Figure 14

Figure 15

Figure 16

Figure 17

Figure 18

Figure 19

Figure 20

x_1 : check if P_1 can be enabled

y_1 : record completion of P_1

Figure 21

REFERENCES

- [1] Adams, D. A. "A model for parallel computations". In Parallel processor systems, technologies and applications, L. C. Hobbs, et. al. (Eds.), Spartan Books, New York, 1970, 311-334.
- [2] Agerwala, T. and M. J. Flynn. "Comments on the capabilities, limitations & 'correctness' of Petri nets". Proceedings of the First Annual Symposium on Computer Architecture, Univ. of Florida, Dec. 9-11, 1973
- [3] Anderson, J.P. "Program structures for parallel processing". Communications ACM 8, 12 (Dec. 1965, 786-788.
- [4] Baer, J. L. "Graph models of computations in computer systems". Ph.D. Dissertation, Dept. of Electrical Engineering, Univ. of California, Los Angeles, California, 1968.
- [5] Baer, J. L. "A survey of some theoretical aspects of multiprocessing". ACM Computing Surveys, Vol. 5, No. 1, March 1973.
- [6] Betourne, C., et. al. "Process management and resource sharing in the multiaccess system ESOPE". CACM, Vol. 13, No. 12, Dec. 1970, pp. 727-733.
- [7] Bredt, T. H. and E. J. McClusky. "Analysis and synthesis of control mechanisms for parallel processes". In Parallel processor systems, technologies and applications, L. C. Hobbs, et. al. (Eds.), Spartan Books, New York, 1970, 287-296.
- [8] Cerf, V. G. "Multiprocessors, semaphores and a graph model of computations". Ph.D. Dissertation, Computer Science Dept., University of California, Los Angeles, California, 1972.
- [9] Commoner, R.; A. W. Holt; S. Even and A. Pnueli. "Marked directed graphs". J. Computer and Systems Sciences 5 (1971), 511-523.
- [10] Conway, M. "A multiprocessor system design". In Proc. AFIPS 1963 Fall Joint Computer Conference, Spartan Books, New York, 139-146.
- [11] Dennis, J. B. and E. C. Van Horn. "Programming semantics for multi-programmed computations". Comm. ACM 9, 3 (March 1966, 143-155.
- [12] Dijkstra, E. W. "Co-operating sequential processes". In Programming languages, F. Genuys (Ed.), Academic Press, New York, 1968, 43-112.
- [13] Genrich, H. "Einfache Nicht-sequentielle Prozesse". doctoral dissertation, Gesellschaft fur Mathematik und Datenverarbeitung, 5201 Burlinghoven.
- [14] Gosden, J. A. "Explicit parallel processing descriptions and control in programs for multi and uniprocessor computers". In Proc. AFIPS 1966 Fall Joint Computer Conference, Spartan Books, New York, 651-660.
- [15] Gostelow, K. P. "Flow of control, resource allocation and the proper termination of programs". Ph.D. Dissertation, Computer Science Dept.,

University of California, Los Angeles, Calif., 1971.

- [16] Hansen, P. B. "The nucleus of a multiprogramming system". CACM, Vol. 13, No. 4, April 1970, pp. 238-241, 250.
- [17] Holt, A. W. and Commoner, F. "Events and Conditions". Record of the Project MAC Conference on concurrent systems and parallel computation, Association for Computing Machinery, New York, 1970, pp 3-52.
- [18] IBM System/360 operating system: Supervisor and data management services. IBF Form No. (28-6646-2, IBM Corp., Nov. 1968.
- [19] Jump, J. Robert and Thiagarajan P. S. "On the interconnection of asynchronous control structures". Laboratory for Computer Science and Engineering, Dept. of Elec. Eng., Rice University, Houston, Texas, Sept. 1972.
- [20] Karp, R. M. and R. E. Miller. "Properties of a model for parallel computations: determinacy, termination, queuing". SIAM J. Applied Math. 14 (Nov. 1966), 1390-1411.
- [21] Karp, R. M. and R. E. Miller. "Parallel program schemata". J. of Computer and System Sciences 3 (May 1969), 147-195.
- [22] Kosaraju, S. Rao. "Limitations of Dijkstra's Semaphore Primitives and Petri Nets". Hopkins Computer Research Report #25, Research Program in Computer Systems Architecture, The Johns Hopkins University, Baltimore, Md., May 1973.
- [23] Luconi, F. L. "Asynchronous computational structures". Ph.D. Thesis, Massachusetts Institute of Technology, 1968.
- [24] Martin, D. F. "The automatic assignment and sequencing of computations on parallel processor systems". Ph.D. Dissertation, Dept. of Electrical Engineering, Univ. of California, Los Angeles, California, 1966.
- [25] Miller, R. E. "A comparison of some theoretical models of parallel computation". IEEE TC, Vol. C-22, No. 8, August 1973.
- [26] Minsky, Marvin. "Computation: finite and Infinite machines". Prentice Hall, Inc., 1967.
- [27] Noe, J. D. "A Petri net model of the CDC 6400". Proceedings of the ACM/SIGOPS Workshop on Systems Performance Evaluation, April 1971, 362-378.
- [28] Patil, S. S. "Coordination of asynchronous events". Ph.D. Dissertation, Massachusetts Institute of Technology, 1970.
- [29] Patil, S. S. "Limitations and capabilities of Dijkstra's Semaphore primitives for coordination among processes". Project MAC, Computational Structures Group Memo, 57, Feb. 1971, pp. 1-18.
- [30] Petri, C. A. "Kommunikation mit automaten". Translated in Project

MAC-M-212 Report, Originally published in 1962.

- [31] Rodriguez, J. E. "A graph model for parallel computations". Ph.D. Thesis, Massachusetts Institute of Technology, 1967.
- [32] Saltzer, J. H. "Traffic control in a multiplexed computer system". Ph.D. Dissertation, Massachusetts Institute of Technology, 1966.
- [33] Slutz, D. R. "The Flow graph schemata model of parallel computation". Ph.D. Dissertation, Massachusetts Institute of Technology 1968.
- [34] Vantilborgh and van Lamsweerde, A. "On an extension of Dijkstras Semaphore Primitives". Information Processing Letters 1 (1972) pp. 181-186.
- [35] Volansky, S. A. "Graph model analysis and implementation of computational sequences". Ph.D. Dissertation, Dept. of Electrical Engineering, Univ. of California, Los Angeles, Calif., 1970.
- [36] Wodon, P. "Still another tool for controlling cooperating algorithms". Carnegie-Mellon University Report, (1972).