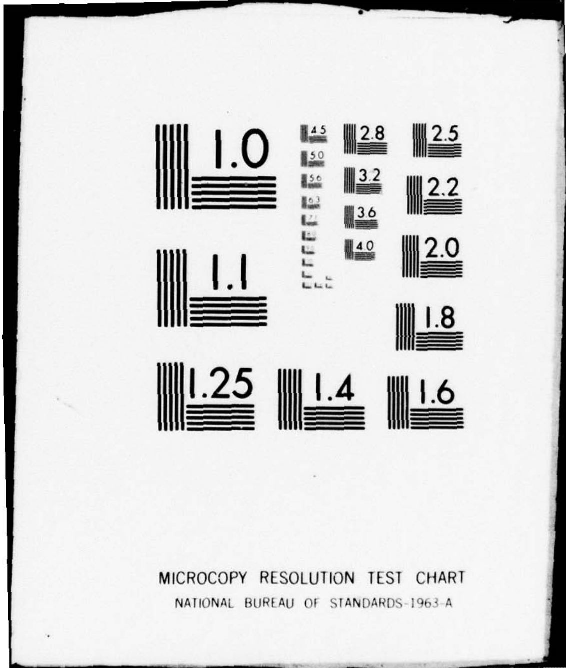
AD-A062 767    WHARTON SCHOOL  PHILADELPHIA PA DEPT OF DECISION SCIENCES    F/G 13/8
USE OF PRODUCTION SYSTEMS FOR MODELLING ASYNCHRONOUS, CONCURREN--ETC(U)
APR 77  M D ZISMAN    N00014-75-C-0440
UNCLASSIFIED    77-04-01    NL

| OF |
AD
A062767

END
DATE
FILMED
3-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A062767
DDC FILE COPY

049-278 360

# Use of Production Systems for Modelling Asynchronous, Concurrent Processes

Michael D. Zisman

Working Paper 77-04-01

JUN 16 1978

10 April 1977

DDC
RECEIVED
JAN 2 1979
F

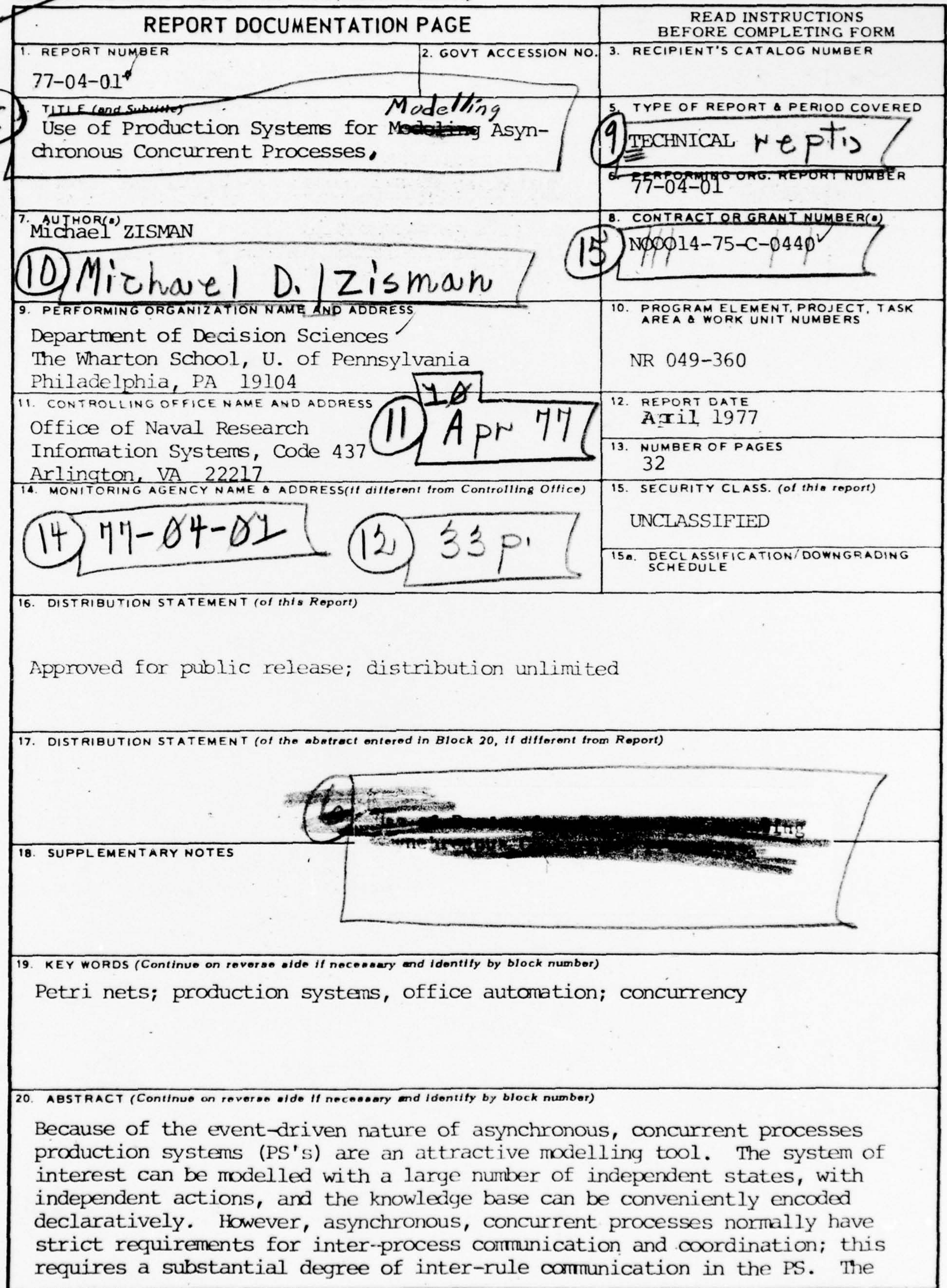
Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

This paper will be presented at the Workshop on Pattern Directed Inference Systems in Honolulu, Hawaii, May 23-27, 1977.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

78 12 27 091

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER 77-04-01 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle) Use of Production Systems for Modeling Asynchronous Concurrent Processes | | 5. TYPE OF REPORT & PERIOD COVERED TECHNICAL |
| | | 6. PERFORMING ORG. REPORT NUMBER 77-04-01 |
| 7. AUTHOR(s) Michael ZISMAN | | 8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0440 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Decision Sciences, The Wharton School, U. of Pennsylvania, Philadelphia, PA 19104 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-360 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Information Systems, Code 437, Arlington, VA 22217 | | 12. REPORT DATE April 1977 |
| | | 13. NUMBER OF PAGES 32 |
| 14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | | 15. SECURITY CLASS. (of this report) UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Petri nets; production systems, office automation; concurrency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Because of the event-driven nature of asynchronous, concurrent processes production systems (PS's) are an attractive modelling tool. The system of interest can be modelled with a large number of independent states, with independent actions, and the knowledge base can be conveniently encoded declaratively. However, asynchronous, concurrent processes normally have strict requirements for inter-process communication and coordination; this requires a substantial degree of inter-rule communication in the PS. The

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408 757

result of this is that a complex control structure is embedded in the short term memory (STM); this is generally considered unattractive for a number of reasons. This paper proposes a separate, explicit control stucture for modelling asynchronous, concurrent processes with PS's. Specifically, the use of a Petri net is addressed. A system of asynchronous, concurrent processes can be modelled using PS's to model the individual processes or events and using a Petri net to model the relationships between the processes. Furthermore, a hierarchy of such networks is proposed; an allowable productionrule action is the instantiation of another network. This is supported with a structured, hierarchial STM.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION per ltr. on file
BY
DISTRIBUTION/AVAILABILITY CODES
SPECIAL
A

# Use of Production Systems for Modelling Asynchronous, Concurrent Processes

Michael D. Zisman

ABSTRACT

Because of the event-driven nature of asynchronous, concurrent processes, production systems (PS's) are an attractive modelling tool. The system of interest can be modelled with a large number of independent states, with independent actions, and the knowledge base can be conveniently encoded declaratively. However, asynchronous, concurrent processes normally have strict requirements for inter-process communication and co-ordination; this requires a substantial degree of inter-rule communiation in the PS. The result of this is that a complex control structure is embedded in the short term memory (STM); this is generally considered unattractive for a number of reasons. This paper proposes a separate, explicit control structure for modelling asynchronous, concurrent processes with PS's. Specifically, the use of a Petri net is addressed. A system of asynchronous, concurrent processes can be modelled using PS's to model the individual processes or events and using a Petri net to model the relationships between the processes. Furthermore, a hierarchy of such networks is proposed; an allowable production rule action is the instantiation of another network. This is supported with a structured, hierarchial STM.

## 1.0 INTRODUCTION

Davis and King [2] suggest that production systems (PS) are most useful in problem domains which are generally modelled by a large number of independent states, with independent actions, and where the knowledge base is best encoded declaratively as opposed to procedurally. Furthermore, they suggest that a fundamental characteristic of PS's is their restriction on the interaction between rules. To produce a degree of interaction between rules requires the introduction of indirect communication through the short term memory (STM). This results in using the STM

both for data and for complex control mechanisms.

This paper investigates the possibility of introducing a separate explicit control structure for PS's where there is a need for substantial interaction between rules. Specificially, we are interested in developing a formalism for modelling a system which is composed of a collection of asynchronous concurrent events. The particular problem domain of interest to us is that of office procedures. We are interested in developing a formalism with which we can model procedures which exist in many office environments; we choose to view instances of these procedures as asynchronous concurrent processes. Modelling office procedures in this way is attractive because an office can be viewed as an environment in which a large number of independent tasks are in progress and these tasks tend to be primarily event driven (i.e., recognize/act logic).

PS's initially seem attractive for modelling in this domain since the domain has many of the positive characteristics outlined in [2]. There are generally a large number of independent (asynchronous and concurrent) states and the knowledge base can be conveniently encoded declaratively (because of its event driven nature). However, most systems of asynchronous, concurrent processes have strict requirements for inter-process communication and coordination. To handle this communication with standard PS's would require a large number of state variables for

inter-rule communication resulting in a complex control structure being embedded in the STM. We are faced with a situation, then, where certain characteristics of the problem domain make a PS representation very attractive and where other characteristics seriously discourage the use of PS's.

We suggest that by augmenting the PS representation to explicitly deal with complex inter-rule interaction we can mitigate some of the difficulties of PS's and still capitalize on the advantages of the PS representation. In this paper, we will suggest that Petri Nets be used to provide this explicit control structure. We will use PS's to model each individual process (event) and the Petri net to model the interaction and temporal relationships between these events. The Petri net will be used to dynamically extract from all of the available rules about the system, those rules which are relevant given the present system state (the present state being defined as the union of all enabled transitions). In this way, the Petri net adds structure to the PS formalism.

Before developing this model further, an example of a "typical" office procedure will be useful. Consider the procedure whereby a journal editor might process papers submitted for publication. When the paper is submtted to the journal secretary, an acknowledgement letter is sent to the author and a request is made of the editor for the names

of referees for the paper. If the editor does not respond within a certain time period a reminder message is sent to him. When he does respond with the names of referees, a message is sent to each referee asking if he might review the paper. Each returns a postcard indicating whether or not he can review the paper; if he cannot, the editor must choose another referee. If he can review the paper, he is given a reasonable amount of time in which to do so; if he does not submit the review in this amount of time, a reminder letter is sent to him. When all of the referees have submitted their reports, a decision is requested of the editor. When the editor makes a decision, the author is informed.

## 2.0 PROCESS REPRESENTATION

Let us assume for the moment the PS's are a useful formalism for modelling each individual process in a system of asynchronous, concurrent processes. In this section, we will look at various formalisms for modelling the relationships between these processes. We will call this the process representation, as opposed to the knowledge representation. Here, we will review formalisms for representing systems which are composed of a sequence of asynchronous and possibly concurrent events (e.g., an office procedure composed of several steps handled by different people). We will consider finite state machines, partial

orderings (PERT networks) and finally Petri nets.

## 2.1 State Machines

A state machine is a finite state automaton which can be represented either by a state diagram (a directed graph) or by a state transition table.

In graphical form, a state machine consists of a number of states, which are nodes in the graph, and a number of directed arcs connecting states. Some state is labelled as the initial state.

At any time, the system is in one and only one state. Based on input to the system, or more generally, some observation about the environment, the system takes an action and changes states. By definition, this is a sequential process. Although it is physically possible to represent concurrent processes with a state machine, the number of states will grow exponentially with the number of tasks to be performed concurrently.

The difficulties with state machines in our problem domain are easily illustrated by considering three variations of the same modelling problem. First, assume that there are a number of tasks to be performed in a fixed order and only one task can be performed at a time. A state machine will easily represent this system and the number of states will grow linearly with the number of tasks. Next,

consider the case where the tasks still must be performed one at a time but where we remove the specific ordering constraint. This would be the case if the tasks were independent but all required some unique resource. In any state, we would like to know which tasks have been completed, and which task, if any, is now in progress. In this case, the number of states will grow exponentially with the number of tasks. Lastly, consider the case where the tasks can be performed in any order and where any degree of concurrency is allowed. Again, if we wish to know, for each state, which tasks have been completed and which are in progress, the number of states will grow exponentially with the number of tasks.

As can be seen, the representation of concurrency is possible in a state machine, but it becomes unwieldy for a large number of tasks.

## 2.2 Partial Orderings

A partial order for a set of events is represented by a directed graph where the arcs represent events to be completed and the nodes represent the termination and commencement of events. The events on a path of the graph represent events which must occur in a fixed order (i.e., the order of their occurrence in the path). Events not on the same path have no ordering relation. All arcs in the graph must be traversed.

Consider again the examples we used to analyze state machines. Representing n tasks which must be performed in a fixed order is simply a partial order graph with one path. One arc is required for each task, so we will need n arcs and n+1 nodes. The graph will grow linearly with the number of tasks.

If we wish to represent the case where the tasks can be performed in any order with any degree of concurrency, partial order graphs are still satisfactory. This is simply a graph with 2 nodes and n arcs. The graph will grow linearly with the number of tasks.

Partial order graphs cannot be used to represent the case where the tasks can be performed in any order but only one at a time. For each possible ordering, we would need a separate graph, or n! graphs. The difficulty here is that partial order graphs can represent only two extremes - fixed order and complete concurrency. Either two events have a precedence relation or they have no relation. There is no way to represent the case where two events can proceed in any order but one must wait for the completion of the other. This capability is obviously fundamental to resource allocation.

As we can see, the problem with partial order graphs is that they are a fixed representation for what is inherently a dynamic system. We must determine a priori all of the precedence relations in the system. For example, in PERT

networks, we first determine how long each event will take to complete; this allows the PERT system to determine all of the precedence relationships. Partial order graphs have no power for describing coordination between events that do not have a strict precedence relation.

This ability to represent coordination among concurrent processes is crucial to our representation of office processes. Resource allocation is an obvious example. Another example is the need to make a choice among alternative actions and to resolve conflicts among competing events. We have shown that state machines can represent choice and conflict but are very unwieldy for representing concurrency. Partial order graphs, on the other hand, can represent concurrency but cannot represent choice, conflict, and coordination among processes. We now turn our attention to a representation which will address these problems.

## 2.3 Petri Nets

In 1962, C.A. Petri devised a general graphical representation for systems [7]. His work was extended in the late 1960's by Anatol Holt [3,4]. Holt called his representation "Petri nets" since they were based on the work of C.A. Petri.

We will assume here that the reader has some acquaintance with Petri nets and only a brief description will be given here. An excellent tutorial can be found in Holt [3].

Like state machines and partial order graphs, Petri nets are represented as directed graphs. They have been used in the study of parallel computation, multi-processing, and computer systems modelling as well as the modelling of other physical processes and human activity processes. The following definition of Petri nets is from Miller [5].

> "A Petri net is a graphical representation with directed edges between two different types of nodes. A node represented as a circle is called a _place_ and a node represented as a bar is called a _transition_. The places in a Petri net have the capability of holding _tokens_. For a given transition, those places that have edges directed into the transition are called _input places_ and those places having edges directed out of this transition are called _output places_ for the transition. If all the input places for a transition contain a token, then the transition is said to be _active_. An active transition may _fire_. The firing removes a token from each input place and puts a token on each output place. Thus a token in a place can be used in the firing of only one transition. A simple example of a Petri net is shown in figure 1. Here tokens are shown as black dots. The starting condition has a token only in place P1. The activity of the net (or process) is then described by the successive firings of transitions. In this example, T1 can fire followed by T2 and T3. Only after both T2 and T3 have fired are T4 and T5 active. Either T4 or T5 can fire but not both. When either T4 or T5 fires it brings the net back to its starting condition and the process is ready to repeat."

We note here that state machines and partial order graphs are both restricted forms of Petri nets. A state machine is a Petri net where each transition can have exactly one input place and exactly one output place. Since each arc then has just one transition, we remove the bar from the diagram. A partial order graph is a Petri net where each place has exactly one input arc and exactly one output arc (in Petri net terms, this is called a marked graph). Since each arc then has exactly one place, we remove the circle from the graph and simply place a dot on each arc where a token resides.

We now turn our attention to the same problem we used to study state machines and partial order graphs. First we wish to represent the case where n tasks are to be performed in a fixed order, one at a time. Figure 2a shows the Petri net for this case. Two states are required for each task (since we want to represent the system at rest as well as busy), so the number of states will grow linearly with the number of tasks.

Next we consider the case where the tasks are to be performed in any order, but only one at a time. Figure 2b represents this case. Each letter indicates a distinct task. Letters before the "/" indicate tasks already completed and letters after the "/" indicate tasks in progress. Place P1 is the resource allocator and the token in P1 represents the unique resource. Once it is "assigned"

to one task, it is unavailable until that task completes. All other tasks will be held up until the active task completes. This scheme will work for any number of resources and competing tasks. We simply initialize the net with one token for each resource. The number of states required for this representation is three for each task and one for the resource allocator, so growth is linear with the number of tasks.

The last case we consider is where the tasks can be performed in any order and any degree of concurrency is allowed. Figure 2c shows the Petri net for this representation. In this case three states are required for each task (not started, in progress, and complete), hence the number of states still grows linearly with the number of tasks.

It is instructive to compare this representation to the state machine example. With the state machine, the machine could be in only one state at a time. This forced us to code all relevant information about the process into each state. With Petri nets, we effectively partition the state into a number of components. Each place, or state, represents a possible assertion about the total system state. If there is a token in a place, that component of the state is asserted. A token in P2 of figure 2b is an assertion that task A is in progress. Thus we can represent all 27 combinations of a three task problem with only 9

places. With a state machine, we need a state for each possible combination and hence 27 states. To represent 20 tasks requires 60 states in a Petri net and 3.5 billion states with a state machine for the same level of information content. When we place the state machine restriction on Petri nets, we allow each transition to have only one input and only on output place. This means that the total information about when a transition can fire must be stored in only one place and all of the information associated with a transition firing must flow to only one place. In the more general Petri net, we allow information from several places to influence the occurrence of events, thus partitioning the total system state into logical components.

Comparing Petri nets to partial orderings, the key addition is a coordination mechanism. Although implicit in coordination, it is worth noting that Petri nets also provide for communication among processes. This coordination is achieved by allowing a place to be input to more than one transition. Although a very simple construct, it proves very powerful in describing required coordination. Note that this construct is specifically not allowed in the partial order graph restriction of Petri nets.

## 2.4 Choice Of Process Representation

As can be seen, state machines and partial order graphs are restriced forms of Petri nets. Partial order graphs do not have the capbility to model coordination among concurrent processes and choice among alternative actions which we require for our representation of asynchronous, concurrent processes. While state machines can represent choice and conflict easily, they are very unwieldy for representing concurrency because the number of states grows exponentially. Petri nets can represent asynchronous, conncurrent processes, process coordination, and choice and conflict among events. They can also be used to model resource a`locaion. Since Petri nets subsume the other formalisms which we have discussed, it is clear that we should use Petri nets for our problem.

To gain an appreciation for the descriptive power of Petri nets as they apply to office processes, we show a simple example. In the journal editing case, there are several instances when a message is sent from the system to the environment for which a reply is expected within a certain period. When the response arrives, the activity continues. However, if the response does not arrive within the specified period, some other action must be taken (such as generating another document). Figure 3 shows the Petri net representation for this. When we are in state P1 we are waiting for a reply within a certain period. Note that both

T1 and T2 are enabled, hence we have a conflict. This Petri net is non-deterministic since we cannot state with certainty whether T1 or T2 will fire. If the reply arrives within the specified period, T1 fires. If it does not arrive, however, T2 fires. The actions specified at T2 are taken and a token is placed back in P1, once again enabling T1 and T2. This will let us wait another time period before taking remedial (T2) action. We have not stated how we will resolve the conflict between firing T1 and T2. Clearly, conflict resolution requires that we have some knowledge about the conflict. We will directly address this issue in the next section when we take up knowledge representation.

## 3.0 KNOWLEDGE REPRESENTATION

One of the major research efforts in the field of artificial intelligence concerns the representation of knowledge. For a program to exhibit intelligence, it must possess knowledge. The difficulty lies in determining the proper structure for representing the knowledge applicable to the problem domain. Obviously, this difficulty increases as the scope of the problem domain broadens.

In representing asynchronous, concurrent processes, we clearly would like to encode knowledge about each process into our representation. This allows the system interpreting the representation to exhibit at least limited intelligence about the problem domain.

A number of formalisms for knowledge representation could be considered for this problem. Planner-like formalisms could be employed (e.g., in our example, we could have a goal of judging a paper which could be decomposed into a number of subgoals) or we could use a form of frame system formalism. However, given the event driven nature of these problems, production systems seem ideally suited.

## 3.1 Production Systems

A PS consists of a set of rules, or "productions", which are of the form (condition)->(action), a database or "context" which maintains state data, and a rule interpreter. The condition portion of each rule (left hand side or LHS) is tested. If the condition is true, the consequent action (right hand side or RHS) is executed. In a "pure" PS [2], the rules are in a sequential list, and rules are evaluated one at a time according to their order in the list. When a rule is found which is true in the current context, the RHS is executed and rule testing begins again at the top of the list of rules. When no rules have true LHS's or a "halt" RHS is executed, the system terminates processing.

In some uses of PS's, the methods for choosing which rules to evaluate and in what order, varies. However, the reader should note that the method chosen for determining "rule priority" is crucial to the efficiency and correctness

of the production system.

The use of production systems varies from simple string rewriting rules to the modeling of human cognitive processes [6], a system for assisting in medical diagnosis [8], and the development of an intelligent terminal agent [1]. The rules can be as simple as testing and replacing substrings in an input string or as general as predicate conditions which perform subtle pattern matching on the database and invoke arbitrary procedures which modify the database and produce side effects.

A simple example of production systems provides some interesting insights. This example deals with rewriting rules. These are used to examine an input string which is normally a sentence from some grammar and to reduce it according to the production rules.

Figure 4a is a set of production rules for reversing a string from an alphabet which does not contain the symbols $ and * (these are used as "marker" symbols). This is what we have called a "pure" production system so rules are chosen for testing according to their order in the rule list. For each rule, the input string is examined from left to right with a moving window looking for a match with the LHS of the rule being tested. If a match is found, the input string is modified by replacing the matched substring in the input string with the RHS of the production rule. Rule testing then resumes with P1. If any rule does not have a match in

the input string, then the next rule in the list is tested. If no rules match, then the system halts. Figure 4b shows how the input string "ABC" is modified during this process.

Upon first examining the production system in figure 4a, it may appear that the relatively simple task of reversing a string should not require six rules for its proper specification. The reason six rules are required here is because of the lack of control structure; we must be very careful that the actions of rules do not interfere with each other. In fact, this is precisely why two marker variables are required instead of just one. Furthermore, the ordering of rules in this example is crucial. If the relative ordering of certain rules is changed, the system will not operate as desired. For example, the reader should consider the effect of switching P1 and P6, or of switching P2 and P4.

It is, of course, possible to add control structure to the production system model and thereby increase its efficiency and/or decrease the number of rules required. One possibility is to add some information to each rule which we will call a branch label. This will be the label of some rule in the rule set. When a rule LHS is tested and found to be false, the next rule in the list is tested. However, if the rule LHS is found to be true, the RHS is executed and the next rule selected for testing is the one specified in the branch label.

It seems intuitive that once a certain rule is tested and found to be true, that this should give us some clue as to which other rules are now likely to be true. Pure production systems do not use this information, but blindly resume their search at the top of the rule list at each cycle. As we implied above, one way to make use of this information is to append to each rule a rule number which should be tried next if this rule is true. This branch instruction gives the system advice on how to proceed.

Figure 4c shows the production system for reversing a string with this added control structure. This added structure allows us to reduce the number of rules by one third and the number of marker variables by one half in this particular example. We argue that this makes the entire production system easier to understand. As far as efficiency is concerned, reversing the three character string required 57 rule tests in the production system without branching and only 18 rule tests in the production system with branch labels.

Adding control structure to the production system by the "branch if successful" label allows us to neatly represent the system as a state machine. (The original production system can also be represented by a state machine, but this is of little use.) Each rule is a state and each state has two outgoing arcs, marked true and false. When the system enters a state, it tests the rule in that

state. If the LHS is true, the RHS is executed and the system follows the arc marked true to its next state. If the LHS is not true, the RHS is not executed and the system follows the arc marked false to its next state. The process terminates when the "eureka" state is reached. Figure 5 shows the state machine corresponding to the production system in figure 4c.

Since a production system of this sort can be represented by a state machine, and a state machine is a restricted form of Petri net, it is certainly possible to represent a production system in a Petri net.

## 4.0 A REPRESENTATION FOR ASYNCHRONOUS, CONCURRENT PROCESSES

In the last two sections, we have discussed formalisms for process representation and formalisms for knowledge representation. Now we wish to derive a formalism suitable for the representation of asynchronous, concurrent processes. It is our contention that there is a real advantage, at least initially, in viewing process representation and knowledge representation separately. In those problems which exhibit some sort of flow characteristic, the current state of the system can often be used to determine which "chunks" of knowledge are likely to be useful in determining what to do next. Stated in a slightly different way, knowledge about the process flow can be a very useful aid in partitioning the total knowledge set

into useful subsets (not necessarily disjoint).

We suggest that each process in a system of asynchronous, concurrent processes be modelled as a set of productions. We can then develop a Petri net structure for this system of processes. Each transition in the net represents a process; since these processes are described by productions, the transition is really a "home" for the rules describing the process. When a transition is enabled (all input places have at least one token), its firing will be determined by the rules "residing" at the transition. All rules at all enabled transitions (the fact that there can be more than one enabled transition is key) will constitute what we will call the _active rule set_. Obviously, membership in this active rule set is transitory. The PS interpreter will continually cycle through rules in the active rule set; when a rule LHS becomes true, the rule actions will be executed and the transition from which this rule came will be fired. This will enable some transitions and disable others, thus modifying the active rule set. The Petri net will be used to dynamically extract from all of the available rules about the system those rules which are relevant given the current state of the system (the current state being defined as the union of all enabled transitions).

In this way, the Petri net adds structure to the PS formalism. The firing of a rule provides direct information (through the Petri net) to the rule interpreter about which other rules are now applicable. This provides, in essence, for direct rule interaction.

We will call this combined formalism an _augmented Petri net_. It is possible, and oftentimes advisable, to describe a system of asynchronous, concurrent processes as a hierarchy of augmented Petri nets. For example, we may want to describe the journal editing system as two augmented Petri nets, one from the viewpoint of the editor and one from the viewpoint of the referee. One of the allowed production rule actions is the (concurrent) instantiation of another augmented Petri net. Attached to each Petri net is its own STM. Lower level networks inherit the STM's of all parent nodes; the result is a structured, hierarchial STM.

We note the similarity between augmented Petri nets and augmented transition networks (ATN) [9]. Just as the ATN is a generalization of state machines, the augmented Petri net is a generalization of Petri nets. The registers in an ATN are analogous to our STM's and the conditions and actions on ATN arcs are equivalent to the production rules which reside at transitions. The main difference, of course, is that in the augmented Petri net, a number of transitions can be enabled simultaneously.

## 5.0 AN EXAMPLE

We will now describe the journal editing process in the augmented Petri net representation. First we list the states in which the system can exist:

1. waiting for a paper to arrive;
2. waiting for the editor to designate referees;
3. waiting for the referee to respond;
4. waiting for the referee to submit his report;
5. waiting for all reports to arrive;
6. waiting for the editor to make a decision.

Next we describe what can happen in each of these states:

1. When a paper arrives, generate an acknowledgement letter to the author and request that the editor designate referees.

2. If the editor does not respond within two weeks, generate a reminder letter to him and continue to do so every two weeks. When he does respond, generate letters to each of the referees requesting their services.

3. If the referee does not respond within two weeks, generate a reminder letter to him. If he does respond and states that he cannot review the paper, inform the editor and request that another referee be designated. If the referee agrees to review the paper allow one month for him to submit his report.

4. If the referee does not respond within one month, generate a reminder letter. When the report is received, inform the editor of its arrival.

5. When all of the reports are in, request that the editor make a decision.

6. If the editor does not make a decision within two weeks after all reports are in, generate a reminder letter. When he does make a decision, inform the author and executive editor.

At each step in the process, we wish to generate the proper documentation and to "file" it appropriately. Figure 6 is an augmented Petri net description of this system. Note that the system is described by two nets, one for the editor and one for the referee. The rule in T02 of the editor net instantiates the referee process for each referee selected by the editor. The two nets communicate through the hierarchial STM.

## 6.0 CONCLUSION

In this paper, we have developed a representation which we feel is useful for describing asynchronous, concurrent processes. This was done by separately studying the formalisms available for process representation and those available for knowledge representation. For reasons discussed throughout this paper, we have chosen to integrate PS's, a formalism for knowledge representation, with Petri nets, a formalism for process representation. We call this an augmented Petri net. By adding the Petri net as an explicit control structure for PS's, we have provided a formalism wherein a system of asynchronous, concurrent processes can be modelled with a collection of inter-related PS's sharing a common, structured STM. This extends the use of PS's to new problem domains.

A computer system for processing the augmented Petri nets in an office automation environment has been implemented on the Wharton School DEC-10 [10,11]; and is currently being used to manage the workflow for an associate editor of CACM. This is a particular use of the system; it accepts as input a general augmented Petri net and then interfaces with a number of office systems (e.g., mail system, document generators, filing systems, etc.) to carry out the work specified in the net.

## 7.0 ACKNOWLEDGEMENT

The author wishes to acknowledge the assistance and encouragement of Dr. Howard L. Morgan during the course of this work.

## 8.0 REFERENCES


1. Anderson, R.A. and J.J. Gillogly, "Rand Intelligent Terminal Agent (RITA):Design Philosophy", Rand Report R-1809-ARPA, Rand Corporation, Santa Monica, Cal., 1976.

2. Davis, R. and J. King, "An Overview of Production Systems", Stanford Artificial Intelligence Laboratory Memo AIM-271, October, 1975.

3. Holt, A. W.,"Introduction to Occurrence Systems", in Jacks, Edwin L. (ed.),<u>Associative Information Techniques</u>,New York:American Ellsevier, 1971.

4. Holt, A.W. and F. Commoner,"Events and Conditions",<u>Record of Project MAC Conference on Concurrent Systems and Parallel Computation</u>,ACM,New York,1970.

5. Miller, R.E., "A Comparison of some Theoretical Models of Parallel Computation", <u>IEEE Transactions on Computers</u>, vol. C-22, no. 8, August, 1973.

6. Newell, A. and H. Simon, <u>Human Problem Solving</u>, Prentice-Hall, 1972.

7. Petri, C.A., "Communication with Automata", Suppl. 1 to Tech. Rep. RAD C-TR-65-337,vol. 1, Griffiss Air Force Base, New York, New York, 1966 (translated from <u>Kommunikation mit Automaton</u>,Univ. Bonn, Bonn, Germany, 1962).

8. Shortliffe, E.H., "MYCIN: A Rule Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection", Computer Science Department, Stanford University, Stanford, Cal., STAN-CS-74-465, October, 1974.

9. Woods, W.,"Transition Network Grammars for Natural Language Analysis", <u>Communications of the ACM</u>, vol. 13, no. 10, October, 1970.

10. Zisman, M.D.,"A Representation for Office Processes", Working Paper 76-10-03, Department of Decision Sciences, The Wharton School, University of Pennsylvania, October, 1976.

11. Zisman, M.D., "SCOOP: System for Computerization of Office Processes", Working Paper 76-06-03, Department of Decision Sciences, The Wharton School, University of Pennsylvania, June, 1976.

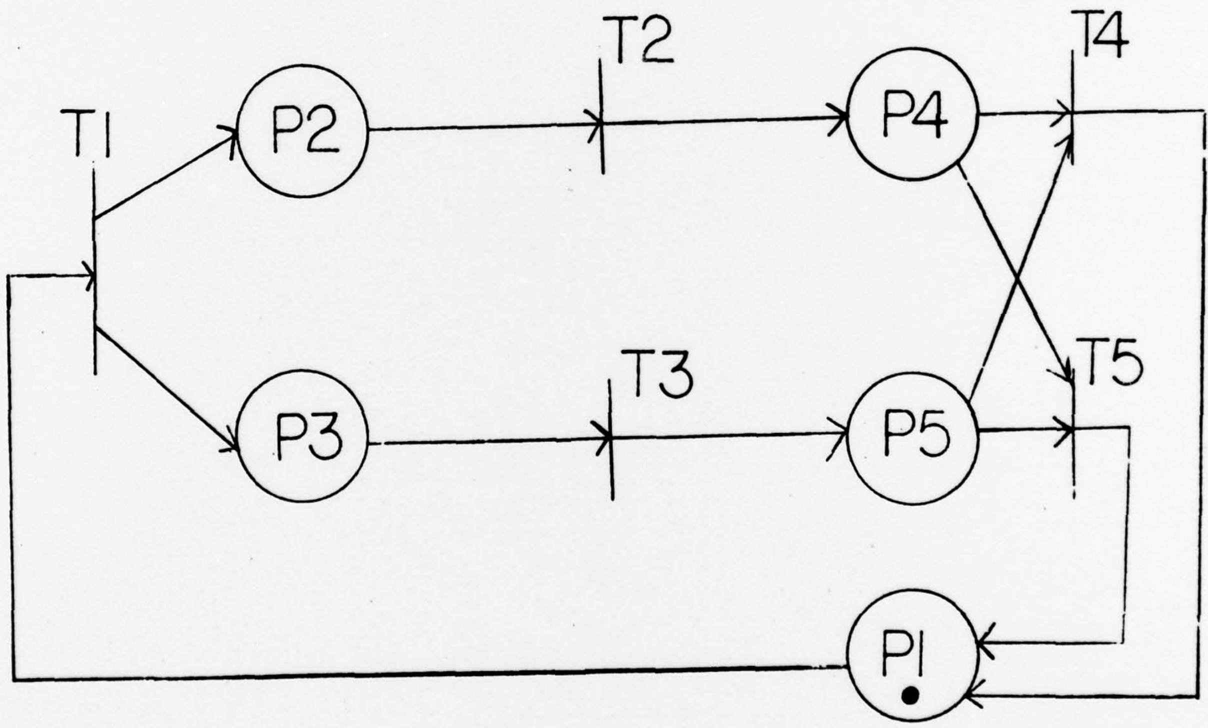



Figure 1

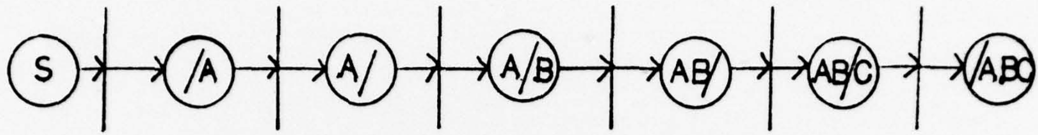



Figure 2a

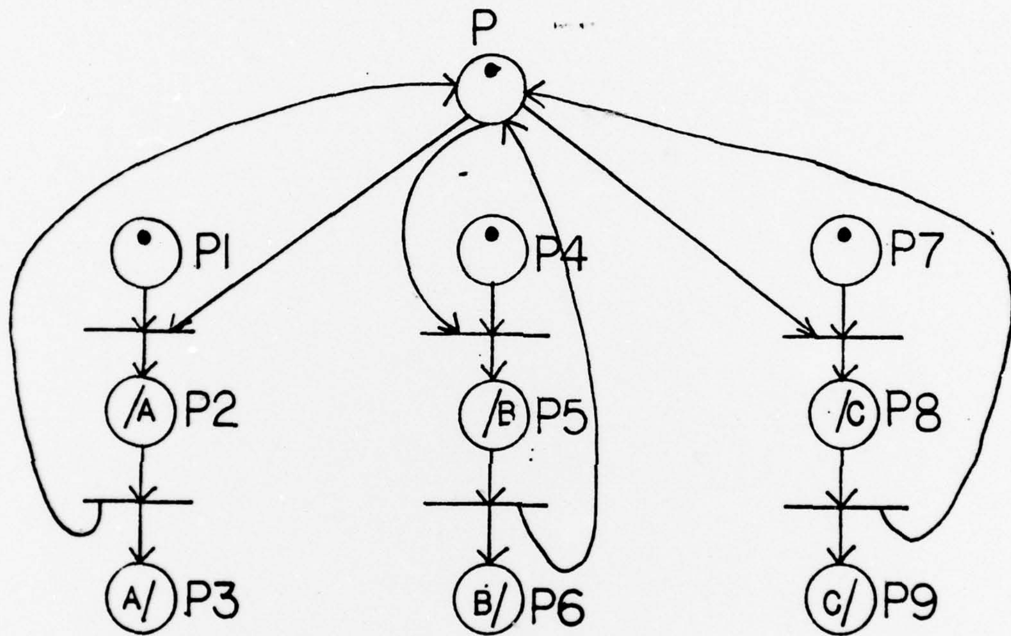



Figure 2b

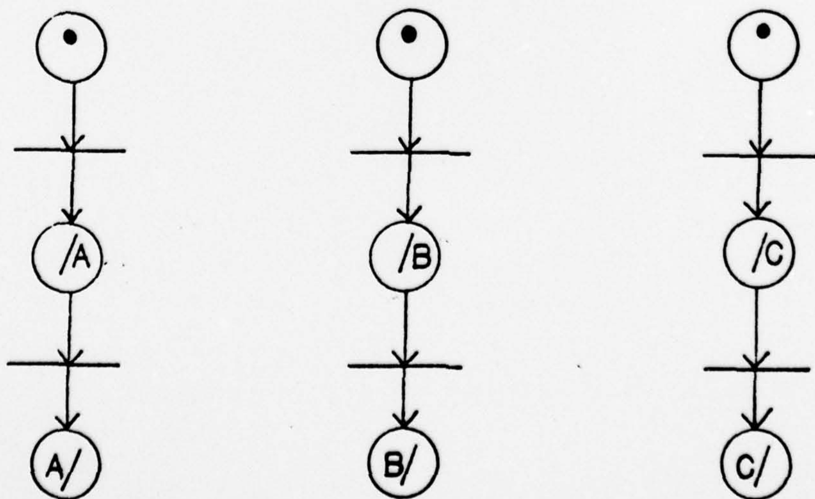



Figure 2c

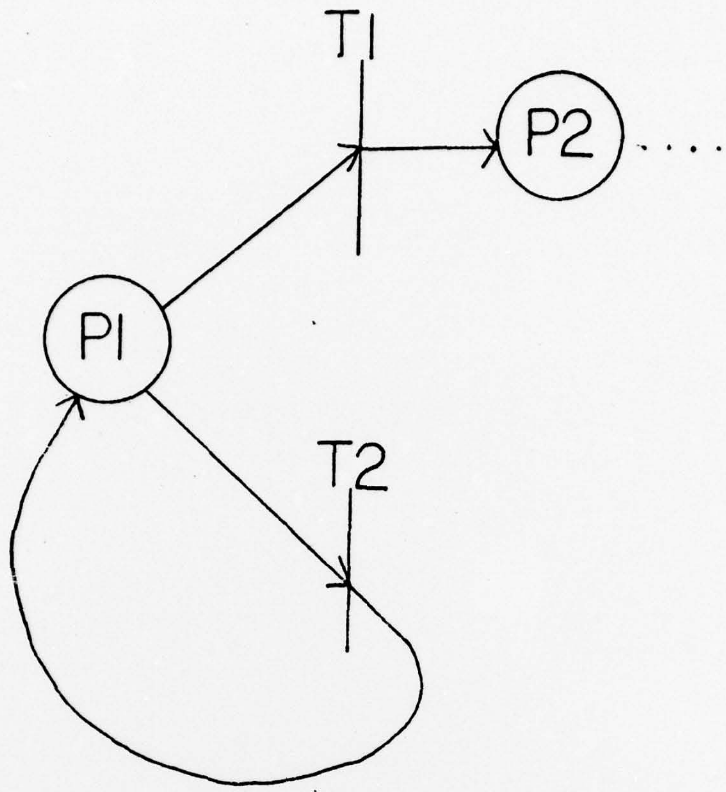



Figure 3

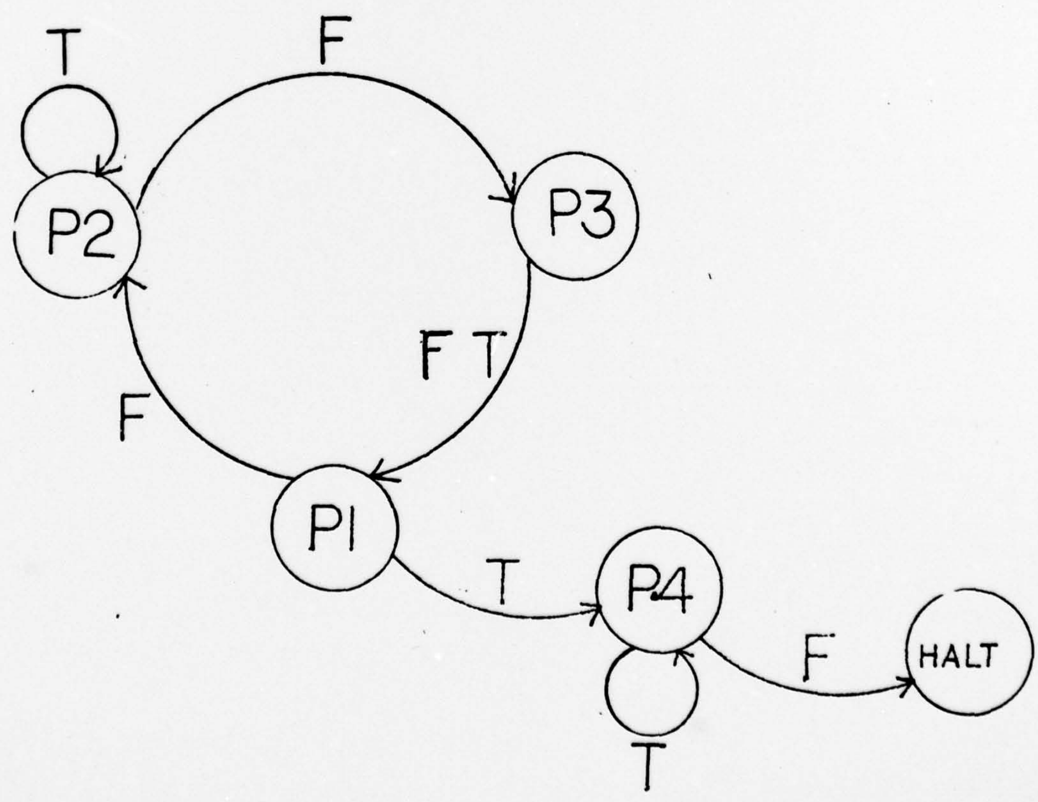



Figure 5

Production Rules for String Reversal
Pure Production System

```
P1: $$ -> *
P2: *$ -> *
P3: *x -> x*
P4: * -> null.
P5: $xy -> y$x
P6: null ->$
```

Figure 4a

---

Application of Above Rules
to String "ABC"

```
ABC  => $ABC          P6
     => B$AC          P5
     => BC$A          P5
     => $BC$A         P6
     => C$B$A         P5
     => $C$B$A        P6
     => $$C$B$A       P6
     => *C$B$A        P1
     => C*$B$A        P3
     => C*B$A         P2
     => CB*$A         P3
     => CB*A          P2
     => CBA*          P3
     => CBA           P4
```

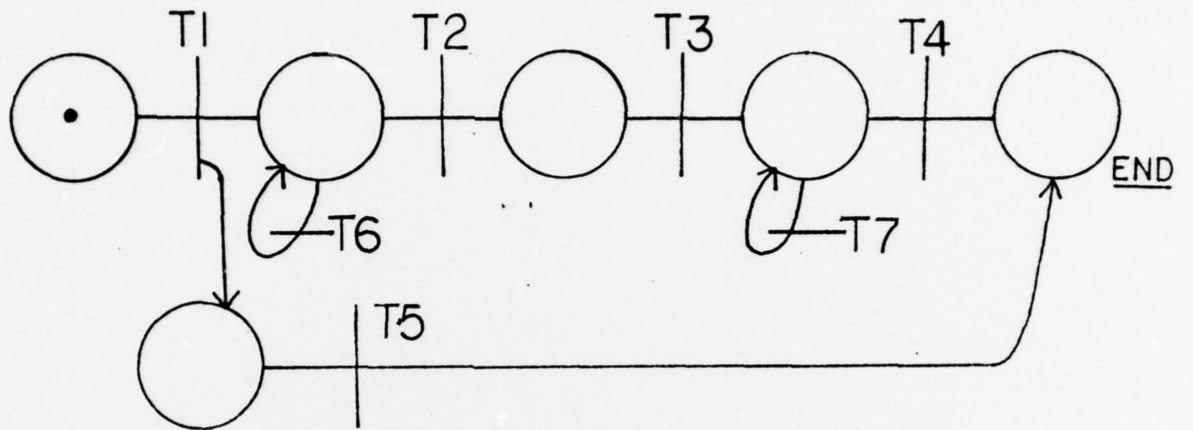
Figure 4b

---

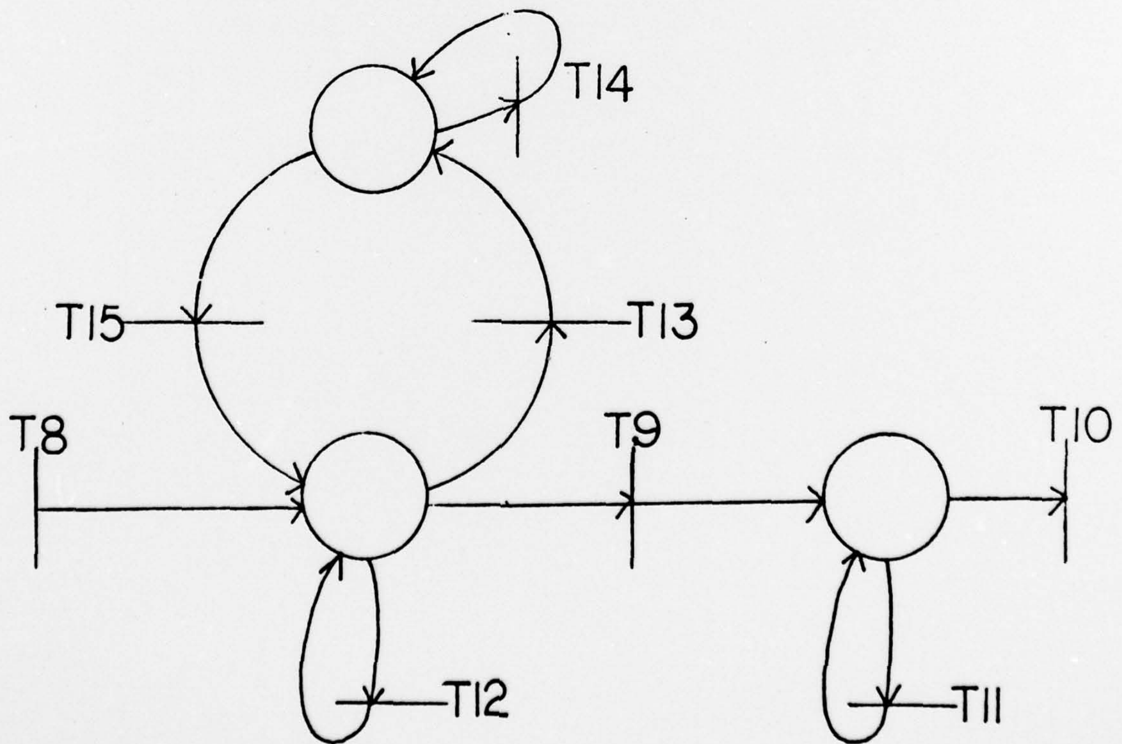
Modified Production Rules
for String Reversal

```
P1: $$ -> null        (P4)
P2: $xy ->y$x         (P2)
P3: null -> $         (P1)
P4: $ -> null         (P4)
```

Figure 4c




EDITOR




REFEREE
Figure 6a

Production Rules for Figure 6a

T01: If a paper is received => send acknowledgement letter to author and request names of referees (any number) from editor.

T02: If the editor supplies the names of referees => instantiate the referee process for each referee.

T03: If all of the referee activites terminate (i.e., fire T08) => request that the editor make a decision.

T04: If the editor supplies a decision on the paper => generate final documentation to author and editor-in-chief.

T05: If the author withdraws the paper => instantiate termination procedure.

T06: If the editor does not respond within two weeks of T06 enabling => send reminder letter to editor.

T07: If the editor does not make a decision within two weeks from T07 enabling => send reminder letter to editor.

T08: If (null condition, fires upon instantiation) => send letter to referee requesting services.

T09:If the referee returns postcard and can review the paper, then allow one month for report.

T10: If report is received => send thank-you letter to referee.

T11: If referee does not send report within one month from enabling of T11 => send reminder to referee.

T12: If referee does not return postcard within two weeks from enabling of T12 => send reminder letter to referee.

T13: If referee returns postcard and cannot review paper => request that editor supply another referee.

T14: If editor does not respond with two weeks from enabling of T14 => send reminder letter to editor.

T15: If editor does supply referee name => send letter to referee requesting services.