AN ABSTRACT OF THE THESIS OF

Mohammed Gazaly for the degree of Doctor of Philosophy in Computer Science presented on June 8, 1981

Title: Specification of Parallel Processing using Production Systems.

Abstract approved: ________________________

Theodore G. Lewis

This research work investigates the use of production systems as a model of parallel processing. The purpose of the model is to provide a suitable medium within which parallel processing systems can be systematically specified, analyzed, and designed. Furthermore, the model provides a suitable means for deriving implementations of synchronization policy specifications by production systems.

First, a review of some existing models of parallel processing is presented. Second, production systems are formally defined and their computation power demonstrated. Third, production system specification of synchronization policies of access to shared resources is systematically analyzed for compatibility, liveness and fairness. Furthermore, systematic design procedures for specifying systems of competing concurrent processes which are guaranteed to be compatible, live and fair are developed. Fourth, the
property of maximal compatibility is investigated and a systematic procedure to derive implementations of the synchronization policy specifications within production systems is presented. Finally, production systems are compared to the other parallel processing models. The interactions between the different properties of synchronization policies, namely, maximal compatibility, liveness, and fairness are also investigated. The thesis is concluded by posing problems for further research.
Specification of Parallel Processing
Using Production Systems

by

Mohammed Gazaly

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of
Doctor of Philosophy
Completed June 8, 1981
Commencement June 1982
APPROVED:

Redacted for privacy
Professor of Computer Science
in charge of major

Redacted for privacy
Chairperson of the Department of
Computer Science

Redacted for privacy
Dean of Graduate School

Date thesis is presented June 8th, 1981

Typed by Fadiah K. Al-Fahl for Mohammed Gazaly
ACKNOWLEDGEMENT

I am extremely grateful for the assistance and support given by the members of my doctoral committee. In particular, I would like to give special thanks to my major professor Dr. T.G. Lewis, Dr. M. Freiling and Dr. W. Bregar for their efforts and time invested in this research work. I am also thankful for the opportunity provided by the Computer Science Department and Oregon State University to make this research work possible.
# TABLE OF CONTENTS

## I. Introduction

1. Parallel Processing .............................................. 2
2. The Synchronization Problem ................................. 6
   2.1 The Model.................................................. 7
   2.2 Policies vs Mechanisms.................................. 7
3. Existing Models.................................................. 10
   3.1 Petri Nets .................................................. 11
   3.2 Computation Graphs ....................................... 17
   3.3 Parallel Program Schemata ................................ 25
   3.4 Other Models............................................... 33
4. Synchronization Mechanisms................................... 39

## II. Production Systems

1. History........................................................... 45
2. Production Systems and the Synchronization Problem ....... 55
   2.1 Production Systems ....................................... 55
   2.2 Operations of the PS Model.............................. 59
3. Computational Power of Production Systems................ 65
   3.1 Production Systems and Finite-State Machines .......... 65
   3.2 Production Systems and Turing Machines ................ 71
4. Advantages and Disadvantages of Production Systems ...... 75
III. Specification and Analysis of Synchronization Policies Using Production Systems
1. Introduction ...................................... 82
2. Production Systems and the
   Synchronization Problem ......................... 85
   2.1 Production Systems ......................... 85
   2.2 Production Systems as Policy Specifications .......... 92
   3.1 Definitions .................................. 100
   3.2 Determining Compatibility .................... 102
   3.3 Designing Compatible Production Systems .......... 104
4. Specification for Live Systems ................. 118
   4.1 Definitions .................................. 118
   4.2 An Algorithm for Determining Liveness .......... 120
   4.3 Designing Live Production Systems .......... 130
   4.3.1 The First Approach ..................... 130
   4.3.2 The Second Approach .................... 132
   4.3.3 The Third Approach ..................... 150
5. Specification of Fair Systems ................. 174
   5.1 Definitions .................................. 175
   5.2 Determining Fairness ....................... 177
   5.3 Designing Fair Systems .................... 188
6. Conclusion ....................................... 192
References ........................................ 195
IV. Maximal Compatibility and Implementation Derivation ............................................. 198
   1. The Compatibility Set and Maximal Compatibility ........................................... 199
      1.1 The Compatibility Set ................................................................. 199
      1.2 Maximal Compatibility ............................................................... 203
      1.3 Determining Maximal Compatibility ............................................... 205
      1.4 Designing Maximally Compatible Production Systems ........................... 208
   2. Implementation Derivation ................................................................. 225

V. Conclusion ................................................................. 236
   1. Production Systems and the Existing Models ........................................... 236
   2. Compatibility, Maximal Compatibility, Liveness, and Fairness .............. 243
   3. Further Research and Problem Areas ................................................. 253

Bibliography ................................................................. 256
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>98</td>
</tr>
<tr>
<td>7</td>
<td>103</td>
</tr>
<tr>
<td>8</td>
<td>103</td>
</tr>
<tr>
<td>9</td>
<td>107</td>
</tr>
<tr>
<td>10</td>
<td>108</td>
</tr>
<tr>
<td>11</td>
<td>109</td>
</tr>
<tr>
<td>12</td>
<td>121</td>
</tr>
<tr>
<td>13</td>
<td>122</td>
</tr>
<tr>
<td>14</td>
<td>123</td>
</tr>
<tr>
<td>15</td>
<td>136</td>
</tr>
<tr>
<td>16</td>
<td>141</td>
</tr>
<tr>
<td>17</td>
<td>152</td>
</tr>
<tr>
<td>18</td>
<td>156</td>
</tr>
<tr>
<td>19</td>
<td>157</td>
</tr>
<tr>
<td>20</td>
<td>164</td>
</tr>
<tr>
<td>21</td>
<td>179</td>
</tr>
<tr>
<td>22</td>
<td>180</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>23</td>
<td>181</td>
</tr>
<tr>
<td>24</td>
<td>181</td>
</tr>
<tr>
<td>25</td>
<td>201</td>
</tr>
<tr>
<td>26</td>
<td>206</td>
</tr>
<tr>
<td>27</td>
<td>214</td>
</tr>
<tr>
<td>28</td>
<td>217</td>
</tr>
<tr>
<td>29</td>
<td>225</td>
</tr>
</tbody>
</table>
Specification of Parallel Processing Using Production Systems

I. Introduction

This research work investigates the use of production systems as a model of parallel processing. The purpose of the model is to provide a suitable medium within which parallel processing systems can be systematically specified, analyzed, and designed, and to provide a suitable means for deriving implementations of synchronization policies.

First, a review of some existing models of parallel processing is presented. Second, production systems are formally defined and their computation power demonstrated. Third, production system specification of synchronization policies of access to shared resources is systematically analyzed for compatibility, liveness and fairness. Furthermore, systematic design procedures for specifying systems of competing concurrent processes which are guaranteed to be compatible, live and fair are developed. Fourth, the property of maximal compatibility is investigated and a systematic procedure to derive implementations of the synchronization policy specifications is presented. Finally, production systems are compared to the other parallel processing models and the interactions between the different properties of synchronization policies are also investigated.
The thesis is concluded by posing problems for further research.

Figure 1 shows the logical structure of this thesis.

1. **Parallel Processing**

Parallel Processing is the simultaneous execution of a number of computing processes by a number of processing units. At one extreme, the processes may be dispersed among several computing devices which may be geographically separated. Systems with this type of configuration are labelled as distributed processing systems. At the other extreme, there are multiprogrammed systems in which the processes run on a single processor.

There are two distinct reasons for the importance of parallel processing:

1. Parallel (concurrent) execution can speed up execution by performing operations concurrently that would otherwise have to be done sequentially. Thus multiplication of two $N \times N$ matrices, which takes $N^3$ multiplications when done by a normal sequential matrix multiplication algorithm, can be performed concurrently in one multiplication time and $\log N$ addition times if $N^3$ processors are available [150].
SYNCHRONIZATION POLICIES

PRODUCTION SYSTEM SPECIFICATION

COMPATIBILITY  LIVENESS  FAIRNESS  MAXIMAL COMPAT.

ANALYSIS  DESIGN

IMPLEMENT  INTERACTION

Figure 1
2. A second, more fundamental reason for parallel processing is that applications such as airline reservation systems, banks, ships, navies and cities are naturally modelled by systems of concurrently executing processes.

In the case of matrix multiplication concurrency is not inherent in the structure of the problem, and is introduced merely for the sake of efficiency. The real payoff of concurrent execution arises not from the fact that applications can be speeded up by artificially introducing concurrency, but from the fact that the real world functions by the execution of concurrent activities. Many applications are modelled more naturally by concurrent than by sequential activities and should be specified in terms of concurrently executing processes even if the program is to be executed on a machine with only a single processor. This avoids the need for programmers to specify the order of execution in cases where this is an "implementation detail".

In addition to the two reasons mentioned above, the advantages of parallel processing include improved reliability with the possibility of graceful degradation. Since there is generally more than one processor in the system, the malfunction of a fraction of those processors should not bring the system to a total breakdown, and the overhead in switching among processes may be decreased.
Finally, one has to note that, while in the early days of computing a large percentage of the dollar value of a system resided in the circuits of the arithmetic and logical units, this is no longer true with the advent of integrated circuits and VLSI. Furthermore, replication of CPU's does not increase the price of a system as significantly as before.
2. The Synchronization Problem

Concurrent execution can be very simply modelled if concurrently executing processes are totally independent of each other. However, in most applications the set of concurrently executing processes is working cooperatively on some larger process, and must perform both communication and synchronization at selected points of the computation, [150].

Processes can communicate by shared nonlocal variables or by message passing. Communication by shared variables may present problems both because careful synchronization is required to avoid access conflicts among processes which share a variable, and because shared variables presuppose a shared-memory implementation. However, in some instances shared variables may be both more natural and more efficient than message passing in modelling an application. Thus, shared memory is likely to remain an important resource at the hardware level, and shared variables are likely to remain an important mechanism for interprocess communication.
2.1 The Model

The ability to solve synchronization problems is critical in any study of parallel processing because although faster algorithms can be constructed for various computations by allowing parallelism, the overhead due to the necessity for process synchronization may well offset the gain in speed due to the parallelism. In formulating such problems, it is essential to develop models for the precise description of parallel computations. Such a model must include means for defining both the computation steps and the sequencing of the steps. In addition, the model should distinguish between the policy adopted to solve the problem and the mechanisms used to enforce the policy. Further elaboration on this distinction is in order.

2.2 Policies vs Mechanisms

In reference to the synchronization problem, a policy defines the order in which the competing processes are served and defines how conflicts among simultaneous requests for the same resources are resolved. As an example, consider the readers/writers problem [91]; many readers can simultaneously read, and a writer can only write if no reader or writer is accessing the resource. A policy to solve this problem is to allow all readers simultaneous access while a
writer must have exclusive access. Another policy is to allow certain subsets of readers simultaneous access while a writer must have exclusive access. Although the policies differ, they both meet the specification requirements of the problem.

Having defined a policy, we must provide a means for enforcing the policy, i.e., a mechanism. For example, one can use semaphores [37] to implement the two policies mentioned above. Other choices include the critical region concept and condition (event) queues defined by Hansen [62].

The separation of policy and mechanism makes a model potentially useful for design specification of systems. It provides the designer with a degree of freedom since in specifying a policy, no assumptions have to be made with respect to mechanisms. Furthermore, it makes it easier to modify an implementation without reconsideration of the policy. Without this separation, the designer may find it hopeless to modify an implementation because of the rigid assumptions in the basic design. The alternative - to replace the original design can be a serious, if not impossible matter, because the rest of the software is intimately bound to the conventions required by the original system. The following quotation from [155] stresses the significance of this separation:
"Such separation contributes to the flexibility of the system, for it leaves the complex decisions in the hands of the person who should make them—the higher level system designer."

Separation of policy and implementation opens the way for comparison of different implementations of the same policy, a fundamental subject of study for any theory of computation. The separation can also help determine if sources of shortcomings are within the policy itself or a flaw in a particular implementation.

A policy specification language should easily describe any policy in readable form, and make it easy to demonstrate properties of the policy.

In addition, the language provided for expressing policies should provide a model of the underlying processes which reflects their essential nature, and yet is comprehensible. A test for adequacy of such a model is that it be possible to systematically derive an implementation or mechanism for the policy from the model.
3. **Existing Models**

With these objectives in mind, we proceed to survey some existing models; namely, Petri nets, computational graphs and parallel program schemata. In addition, we briefly consider a set of other models which include finite state machines, the UCLA graph model, Rodriguez' model, Adam's model, Bochmann's model, Lynch & Fischer's model, flow expressions, and the rational design methodology.

For each of the three models, Petri nets, computational graphs and parallel program schemata, we present a formal definition and describe the operations of the model. The model is further illustrated by an example. Finally, we discuss the advantages and disadvantages of the model.
3.1 Petri Nets

Many definitions for Petri nets have been introduced [8], [94], [119], [130]. A Petri net (PN) 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 of a PN may hold zero or more tokens.

For a given transition, those places that have arcs directed into the transition are called input places and those places having arcs directed out of the 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 enabled. An enabled 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 PN with tokens is a marked PN. The distribution of tokens in a marked PN defines the state of the net and is called its marking. The marking may change as a result of the firing of transitions.
A simple example of a PN is shown in Figure 2.

Here tokens are shown as black dots. The initial state has a token only in place $p_1$. The activity of the net is then described by the successive firings of transitions. In this example, $t_1$ can fire followed by $t_2$ and $t_3$. Only after both $t_2$ and $t_3$ have fired are $t_4$ and $t_5$ enabled. Either $t_4$ or $t_5$ can fire but not both, i.e., firing either $t_4$ or $t_5$ disables the other, they are said to be in conflict. When either $t_4$ or $t_5$ fires it brings the net back to its initial state and the process is ready to repeat.

As can be seen from the above example, in addition to the static properties represented by the graph, a PN has dynamic properties that result from its execution.

One can see that transitions in a PN are of the AND-input AND-output form. Other classes of PN's have been devised [8], some of which include the following types of transitions in addition to the ordinary transitions of a PN:
1. transitions with OR-input logic are allowed.
2. transitions which fire iff some of the input places have no tokens.
3. transitions which fire iff the output place has no tokens.

We now consider the advantages and disadvantages of Petri nets as a model of asynchronous concurrent systems.

The simplicity and power of Petri nets make them suitable for working with asynchronous concurrent systems. The success of any model is due to two factors: its modeling power and its decision power [130]. These two factors generally work at cross purposes. PN models represent an attempt to compromise between these two factors. They have better modeling power than finite state machines and limited modeling power relative to Turing machines. Extended Petri nets have the modeling power of Turing machines.

Petri nets have the ability to model a system hierarchically. An entire net may be replaced by a single place or transition for modeling at a more abstract level (abstraction) or places and transitions may be replaced by subnets to provide more detail modeling (refinement).
Petri nets are uninterpreted models, thus one may deal with the abstract properties inherent in the structure of the net independently of the semantics associated with the net.

It is difficult to model some events or conditions in systems by Petri nets. It is difficult to model priority systems by Petri nets. Kosaraju describes a coordination problem and proves that it is not representable by Petri nets. The problem [2] is as follows. There are four cyclic processes, \( P_1, P_2, C_1 \) and \( C_2 \) and two buffers \( B_1 \) and \( B_2 \). \( P_1 \) and \( P_2 \) are producers which place one item on top of \( B_1 \) and \( B_2 \) respectively in every cycle. \( C_1 \) and \( C_2 \) consume one item each from the bottom of \( B_1 \) and \( B_2 \) respectively. However, \( C_1 \) has higher priority than \( C_2 \) so that \( C_2 \) can consume only if \( B_1 \) is empty. Intuitively, this problem is not representable by Petri nets because \( C_2 \) can consume only if \( B_1 \) contains no tokens. In a Petri net, a transition can fire if and only if all input places contain tokens.

Petri nets, especially those with conflicts, are difficult to analyze. Analysis techniques for Petri nets include the methods of computational induction and inductive assertions [2] and use of invariants [94]. Although there are several approaches to the analysis of Petri nets, almost all work in this area eventually use one basic
technique involving the concept of the reachability tree [130]. Analysis techniques based on computational induction and inductive assertions do not offer systematic procedures for proving properties of the model. Though many efforts have been made, the assertion approach can not yet be applied in every day usage. As it is mechanizable only to a small extent, the main difficulties for the programmer consist in guessing the correct assertions, and in deriving one assertion from another. The reachability problem is exponential time-hard and exponential space-hard. Although the reachability problem is decidable, however, the complexity may make the analysis, even for simple questions, unfeasible. In general, it may be very difficult to show that a "net is deadlock free."

Some PN problems are not solvable, e.g., the subset and equality problems. The subset problem is to determine if the reachability tree of one net is a "subset" of the reachability tree of another net. The equality problem is to determine if the reachability tree of one net is equal to that of another net [130]. The reason for this is that an arbitrary number of tokens can be generated, and the number of "states" (and hence the reachability tree of the Petri net) may be infinite.
Little attention has been given to developing modeling techniques specifically for Petri nets. Work on design with Petri nets and implementation of Petri nets has been limited in scope. Usually, the design is created in a traditional representation, then the design would be converted into a Petri net and the Petri net analyzed. If no design errors were discovered, the design could then be implemented in the traditional manner. If there were errors, however, it would be necessary to determine how the error which was found in the Petri net representation manifests itself in the original design, modify the design, and repeat the entire process of conversion to a Petri net and analysis.

While Petri nets allow nondeterminism, no two transitions can fire simultaneously. When two transitions are in conflict, it is assumed that only one will fire. This assumption implicates the existence of a resolving mechanism at that level "which is not modelled by the Petri net." As a consequence, Petri nets are very poor at modelling decision making.

Finally, it is noteworthy that Petri net models represent aspects of control with data left out altogether. This adds another degree of difficulty in the derivation of a Petri net model.
3.2 Computation Graphs

Computation graphs were devised by Karp and Miller [80]. The model can be defined as follows:

Definition: A computation graph is a directed graph consisting of:

1. nodes \( n_1, \ldots, n_k \);
2. branches \( d_1, \ldots, d_t \), where any given branch \( d_p \) is directed from a specified node \( n_i \) to a specified node \( n_j \);
3. four nonnegative integers, \( A_p, U_p, W_p, \) and \( T_p \), where \( T_p \geq W_p \) associated with each branch \( d_p \), and
   a. \( A_p \) gives the number of data words initially in a first-in first-out queue associated with \( d_p \);
   b. \( U_p \) gives the number of words added to the queue whenever the operation \( O_i \) associated with \( n_i \) terminates,
   c. \( W_p \) gives the number of words removed from the queue whenever the operation \( O_j \) is initiated; and
   d. \( T_p \) is a threshold giving the minimum queue length of \( d_p \) which permits the initiation of \( O_j \).

Upon initiation of \( O_j \) only the first \( W_p \) of the \( T_p \) operands for \( O_j \) are removed from the queue. The others, if any, remain available for later initiations of \( O_j \).
The operation \( O_j \) associated with a given node \( n_j \) is eligible for initiation if and only if, for each branch \( d_p \) directed into \( n_j \), the number of words in the queue associated with \( d_p \) is greater than or equal to \( T_p \). It is assumed that no two performances of \( O_j \) can be simultaneously initiated. After \( O_j \) becomes eligible for initiation, \( W_p \) words are removed from each branch \( d_p \) directed into \( n_j \). The operation \( O_j \) is then performed. When \( O_j \) terminates, \( U_q \) words are placed on each branch \( d_p \) directed out of \( n_j \). The times required to perform the steps mentioned above are left unspecified by the model. These times may differ for different initiations and performances of the same operation.

As an example of a numerical computation process, let us consider the numerical solution to the ordinary differential equation:

\[
\frac{d y}{d x} = f(x, y)
\]

with the initial or boundary condition \((y_0, x_0)\). An approximation formula based on Taylor series expansion[145] is:

\[
y_{i+1} = y_i + f(x_i, y_i)(x_{i+1} - x_i)
\]

The computation for \( i \) running from 0 to \( n-1 \) is given by the following computation graph. The values \( x_k \), \( 0 \leq k \leq n \), and \( y_0 \) are assumed to be available in some storage medium before the computation begins.
The quadruple \((A_p, U_p, W_p, T_p)\) is shown on each branch \(d_p\). See Figure 3. The nonzero \(A_p\) correspond to initial data as follows. On the branch \((n_1, n_1)\) are placed \(n\) arbitrary markers to indicate that \(i\) must be incremented \(n\) times. On branches \((n_1, n_2)\) and \((n_1, n_3)\) is placed the value \(x_0\). On branches \((n_5, n_5)\) and \((n_5, n_3)\) is placed the value \(y_0\). The operations are defined as follows.

\(O_1\) increments index \(i\) and fetches the corresponding \(x_i\) from storage. It also places the value \(x_i\) on branches \((n_1, n_2)\) and \((n_1, n_3)\); and the value \(i\) on branch \((n_1, n_6)\). It is assumed the initial value of \(i\) is zero.
$O_2$ takes operands $x_i$ and $x_{i+1}$ from $(n_1, n_2)$ forms the difference $(x_{i+1} - x_i)$ and places the result on $(n_2, n_4)$.

$O_3$ takes $x_i$ from $(n_1, n_3)$ and $y_i$ from $(n_5, n_3)$, (remember $(n_5, n_3)$ initially has $y_0$) computes $f(x_i, y_i)$ and places the result on $(n_3, n_4)$.

$O_4$ multiplies the result of $O_3$ by the result of $O_2$, i.e., $f(x_i, y_i)(x_{i+1} - x_i)$ and places the result on $(n_4, n_5)$.

$O_5$ takes the result of $O_4$ and the operand $y_i$ from $(n_5, n_5)$ (remember $(n_5, n_5)$ initially has $y_0$), forms their sum and places the result (i.e., the value of $y_{i+1}$) on branches $(n_5, n_6)$ and $(n_5, n_3)$.

$O_6$ stores the result of $O_5$ as $y_{i+1}$.

An execution of a computation graph is defined by the possible sequences of initiations. Since more than one operation can simultaneously be executed an execution sequence is described by a sequence of non-empty sets $S_1, S_2, \ldots, S_p, \ldots$ such that each set $S_p$ is a subset of $\{1, 2, \ldots, N\}$ where $N$ is the number of nodes in the graph. Each occurrence of $S_p$ denotes the simultaneous initiation of $O_j$ for all $j \in S_p$. There is no implication that successive initiations are equally spaced in time.
The number of possible executions is constrained by the initiation rules. Furthermore, an execution is termed "proper" whenever an operation which is eligible for initiation will actually be initiated after some finite number of initiations of other operations.

To illustrate, we interpret the sequence \( \{n_1, n_3\}, \{n_1, n_2\}, \{n_4\} \) denotes that operations 0₁ and 0₃ are initiated simultaneously. As a consequence of performing 0₁, branch \((n_1, n_1)\) has one item less in its queue, i.e., \((n-1)\) items (markers), branches \((n_1, n_2)\) and \((n_1, n_3)\) have two elements, i.e., \(x_0\) and \(x_1\) on each of their queues, and \((n_1, n_6)\) has one element, i.e., the value of \(i=1\), on its queue. When 0₃ is performed, an element, i.e., \(x_0\) is taken off the queue belonging to \((n_1, n_3)\), and \(y_0\) is taken off \((n_5, n_3)\) and the value \(f(x_0, y_0)\) is placed on the queue of \((n_3, n_4)\). After simultaneously performing 0₁ and 0₃, \((n_1, n_1)\) has \((n-1)\) markers on its queue, \((n_1, n_2)\) has \(x_0\) and \(x_1\), \((n_1, n_3)\) has \(x_1\), \((n_1, n_6)\) has the value "1" on its queue, \((n_5, n_3)\) has no elements in its queue and \((n_3, n_4)\) has the value of \(f(x_0, y_0)\) on its queue.

Next 0₁ and 0₂ are simultaneously initiated. 0₂ can be initiated since there are two elements \(x_0, x_1\) on branch \((n_1, n_2)\) and its threshold = 2. After performing 0₁ and 0₂, \((n_1, n_1)\) has \((n-2)\) markers, \((n_1, n_2)\) has \(x_1\) and \(x_2\).
(n₁, n₃) has x₁ and x₂, (n₁, n₆) has the values "1" and "2" and (n₂, n₄) has the value (x₁ - x₀). Note that (n₃, n₄) has the value f(x₀, y₀).

Since (n₂, n₄) and (n₃, n₄) each has one element, which is equal the threshold for each branch, 0₄ can be initiated and as a result, (n₂, n₄) and (n₃, n₄) both have zero elements and (n₄, n₅) has one element.

The above sequence of operations is not a proper execution since there are operations that can be initiated but they are not initiated in the sequence. Operations 0₁ and 0₅ are such operations. Furthermore, note that the above sequence is not the only possible execution.

Computation graphs are particularly convenient for representing many repetitive processes. The model is capable of describing the sequencing of many "inner loops" encountered in numerical computation, including some which depend on several indices. All computations described within the model are determinate, even when the speeds of the computation steps are variable and specified. Determinacy is defined as the condition that the model represents the same computation regardless of which proper executions occur.
Conditions for termination and for queues to remain bounded can be derived [80]. Also, there is a procedure for finding the number of performances of each computation step. The model does not include any assumptions about timing; there is no implication that successive initiations are equally spaced in time.

The model is completely interpreted, i.e., each operation is semantically specified. Thus, the model may be used as a programming language, although only as a representation language at the present time.

In contrast to the above advantages, the fixed queue discipline restricts the model. The fact that each data queue has a unique "source" and a unique "sink" raises severe restrictions on the type of sequence control which can be represented within the model. The lack of having a common memory for data (or random access storage) restricts the practical use of the model to represent a larger class of computations.

The lack of a facility for data dependent conditional transfers makes the model inappropriate for representing many computations. Branches do not carry control information. The control corresponds to an AND-input AND-output logic. Computation graphs are unable to model OR-input
OR-output logic.

Finally, it is notable that no two performances of an operation can be simultaneously initiated in this model. Furthermore, no specification of the particular operations allowed is devised yet, and no method of translating from conventional programming languages into the language of computation graphs is available yet.
3.3 Parallel Program Schemata

Parallel program schemata (PPS) have been developed by Karp and Miller [81]. A PPS consists of: a set $M$ of memory locations; a finite set $A$ of operations; and $\mathcal{J}$, a control for sequencing operations. Associated with operation $\alpha \in A$ is a set of domain locations $D(\alpha) \subseteq M$ and a set of range locations $R(\alpha) \subseteq M$.

The performance of operation $\alpha$ in a computation proceeds as follows. When $\alpha$ is initiated it obtains its operands from locations $D(\alpha)$. At some later time operation $\alpha$ completes its performance and upon its termination it places its results in the $R(\alpha)$ locations. Also, to represent conditional transfers, each operation $\alpha$ has a number of outcomes $\alpha_1, \alpha_2, \ldots, \alpha_{K(\alpha)}$, $K(\alpha)$ is a positive integer, such that upon termination one of these outcomes is chosen.

In this model the control is a very general sequencing structure that is formulated as a transition system $\mathcal{J} = (Q, q_0, \Sigma, \tau)$ where $Q$ is a set of states, $q_0$ is a designated initial state, $\Sigma$ is the alphabet consisting of, for each $\alpha \in A$, an initiation symbol $\alpha$ designating the initiation of operation $\alpha$ and the termination symbols $\alpha_1, \alpha_2, \ldots, \alpha_{K(\alpha)}$ designating the possible outcomes of a performance of $\alpha$, and finally $\tau$ is a partial transition
function from Qx to Q, which is total on termination symbols.

The control is designed to work as follows. If the schema control is in some state q then certain transitions from this state are possible. Any termination could occur. Also, certain operations could be initiated from the state. For any operation having an initiation defined then \( \tau(q, \alpha) \) would be defined and specify a next state. Parallel operation is achieved by having several initiations occur before their terminations occur.

Although the domain and range locations in memory for an operation are defined in a schema, the particular functions computed by a performance of an operation are left undefined. An interpretation of a schema specifies the functions. This is done by associating a set \( C(i) \) of possible values for each \( i \in M \), by specifying an initial memory contents \( c_0 \), and for each \( \alpha \in A \) specifying two functions

\[
F_\alpha : \{X \in D(\alpha) : C(i) \} \rightarrow \{X \in R(\alpha) : C(i) \}
\]

and

\[
G_\alpha : \{X \in D(\alpha) : C(i) \} \rightarrow (\alpha_1, \ldots, \alpha_k(\alpha)).
\]

In a computation under interpretation \( \delta \), \( F_\alpha \) determines the results to be stored in locations \( R(\alpha) \) by a performance of \( \alpha \), and \( G(\alpha) \) determines the conditional branch to be taken.
Given an interpretation the notion of a computation can be defined. A finite or infinite sequence z of initiation and termination symbols is called an $I$-computation for the schema if the following conditions hold:

1. The control of the schema must allow the initiations and terminations to occur, starting from the initial state $q_0$ and proceeding event by event,

2. For a finite computation all previously initiated performances must have terminated and that the state reached is one in which no operation can be initiated; and

3. Neither initiations nor terminations are "infinitely slow" as compared to the other initiations or terminations. For terminations this means that once an operation is initiated its termination must occur only after a finite number of other initiations and terminations. That is, the computation being performed by the operation does not require an infinite amount of time. For initiations, the condition means that an initiation of an operation must occur after a finite number of steps if from some state on the initiation is defined.

Karp and Miller [81] give a formal statement of the above three conditions.
In an $\mathcal{I}$-computation memory locations are read at the time of the initiation and changed at the time of terminations. Within this definition it is possible to have several performances of a given operation going on concurrently. This corresponds to several initiation symbols occurring in the sequence before the related termination symbols. Thus a queue of currently operating performances for each operation is kept, and when a termination occurs it is paired with the earliest outstanding initiation of the operation.

An example of the control structure of a parallel program schema is shown in Figure 4.
The schema of this example has the set of operations 
Q = \{a, b, c, d\}. Operation d has two possible outcomes d_1 and d_2, and all the other operations have a single outcome. All defined transitions for initiation symbols are shown, but only those transitions for termination symbols that can occur in computations are shown in Figure 4.

Starting at the initial state q_0 one can see that either operation a or operation b can be started. In fact, the diagram from state q_0 to q_8 shows that operations a and b can be performed in either order for initiations and terminations; that is, they can be going on concurrently. Also, only when both performances have been completed can operation c be initiated. This is a schema representation for a simple FORK-JOIN pair. The performance of operation c follows after both a and b are completed, then operation d is performed. Operation d has two outcomes. Outcome d_1 causes the process to be repeated whereas outcome d_2 causes the process to end.

Let M = \{1,2,...,8\} with domain and range locations:
D(a) = \{1,2\}, R(a) = \{3\}, D(b) = \{4,5\}, R(b) = \{6\},
D(c) = \{3,6\}, R(c) = \{7\}, D(d) = \{7\}, R(d) = \{8\}. One interpretation for the schema is the common iterative point relaxation process \(f = \frac{1}{4}(N+S+E+W)\) on a single point; where operations a and b are (N+S) and (E+W), respectively, operation c adds these two sums and operation d is the
multiplication by \( \frac{1}{i} \), and a test for ending.

For each \( i \in M \) an \( i \)-computation \( z \) defines a sequence of values in location \( i \); denote this sequence by \( S_{\ell_i}(z) \). A schema is called \textit{determinate} if and only if for all interpretations, all \( i \in M \) and each pair of \( i \)-computations \( y \) and \( z \), \( S_{\ell_i}(y) = S_{\ell_i}(z) \). Thus even though the order of completing concurrently operating operations is different, in a determinate schema, the values in memory are the same.

We now consider advantages of this model. The model is defined at a level of abstraction which leaves unspecified certain details pertinent to the operation of programs, and emphasis is placed on properties that hold true regardless of how these details are specified. This choice limits the types of questions that one can express within the model, but it permits the development of decision procedures for certain interesting properties.

The control of sequencing in this model is generalized to allow concurrent execution of several operations. The choice between outcomes of an operation can be nondeterministically made. A schema is uninterpreted; that is, it does not assign a particular "meaning" to its operations. The only information given about operations is their associated domain and range locations in the memory. The model
includes the notions of random access memory and data dependent conditional branching.

In investigating properties of parallel processing, two different approaches appear. First, one may wish to sufficiently restrict the basis entities of the model so that the properties desired can be shown to hold. A second approach is to have a free model and establish necessary and sufficient conditions for properties to hold. This is the approach used in PPS studies.

For certain classes of schemata, some questions are solvable and decision procedures for properties can be derived [81]. Special classes of PPS include counter schemata, repetition free-schemata, vector addition systems, flow-chart schemata, and finishing schemata [8], [28], [81].

We now consider the disadvantages of this model. The rule that performances of an operation terminate in the same order as they are initiated restricts the modeling power of the model. No information about the nature of the operations is specified within the model. The model also does not include notions such as indexing, data structure, the equality of the F and G functions for different operations, and time. When the time element is included,
the effects of priority queuing and interrupts can be modeled.

The number of states can exponentially grow as more "parallelism" is possible within the model. As a consequence, algorithms (within this model) for some solvable problems can be unfeasible. Furthermore, some problems; for instance, whether two finite-state schemata are equivalent, are undecidable [81]. Techniques for transforming schemata to equivalent schemata with desired properties are not developed. In general, this model is a "relatively complex" representation of parallel processing. The model has been introduced as a generalization of computation graphs.

3.4 Other Models

Other approaches to the representation of parallel processing exist. Finite state machines (FSM) [88] have been used to describe parallel processing. Since in a FSM the number of states is finite, it is possible to answer almost any question about a FSM model; hence the model has very high decision power. On the other hand, the class of systems which can be modeled is severely limited, which means that such a model has very low modeling power.
Another model known as the UCLA graph model was developed by Estrin [7], [8], [111], [112], [113]. It is also called a directed acyclic bilogic graph. In this model, the nodes of the graph represent operations as partial functions and, the edges represent flow control. Combinational logic controls the sequencing of operations. If the input logic of a node is AND, tokens are needed on each input edge to enable an operation. For OR logic, tokens are needed on any one input edge. Execution of the node removes the enabling tokens and places tokens on the output edges according to the output logic. For AND output logic tokens are placed on all output edges, while for OR logic, tokens are placed on any one output edge. A start and final edges are used to initiate and terminate an execution. An execution starts by placing a token on the start edge and terminates when a token appears on the final edge.

Rodriguez [8], has extended the ideas behind the bilogic graph model in terms of more specific and more interpretative operators and more elaborate control. His model is a directed graph composed of nodes, representing operations, and edges, representing data and control information. A status is associated with each edge. The status of an edge may be either ready, idle, disabled, or blocked. For each node, a table of transitions lists the combinations of status values which enable an operation and the new
status values which are assigned to the edges as a result of its execution. The state of the graph is defined by the status values of its edges.

Adams [1], [8] developed a model that is completely interpreted and is closer to a real programming language than any of the models mentioned in this section. It allows not only queues of data but also the possibility of their being structured, as well as the definition of recursive procedures. The model is a directed graph in which the sequencing control is governed by the flow of data. The nodes of the graph represent computation steps, and the edges represent transmission paths for data and control. The nodes are either primitive or procedure nodes. Primitive nodes may be either computational, that is, they map values on the incoming edges into values on the outgoing edges, or both computational and logical by also mapping edge status. Procedure nodes consist of a subgraph program itself composed of primitive/procedure nodes and a set of interface edges whose data types must be compatible with their respective counterparts in the calling graph. The edges represent values of FIFO queues of data being either a simple entity or an ordered set. The possible data types are separately defined by a context-free grammar. This model is determinate and has the universal computing capability (Turing machine).
Bochmann defines a model based on transition systems. The system is characterized by a (usually infinite) set of possible states. A transition from one state to another is effected by an operation. Only "quasi-parallel" state transitions are allowed [11].

Lynch and Fischer [109] developed a model for describing both the behavior and the implementation of distributed systems. A system's (input-output) behavior is modeled by a set of finite and infinite sequences of actions, each action involving access to a variable. The implementation model relies on the basic notions of process and variable assuming indivisibility of variable access. Time is not included in the model. Furthermore, the existence of a test-and-set operation is assumed.

The models considered so far use state transition diagrams to describe parallel processing. Programming languages, although not a good medium for analyzing a system's behavior have been used. Shaw [141] develops a notation: flow expressions (FE), which can be used in the modeling of concurrent programs and in the specification and solution of synchronization problems. Flow expressions are also related to path expressions [19], developed by Campbell. A (sequential) flow is represented by a, possibly infinite, string composed of: symbols that are names for atomic
(indivisible) entities, lock symbols, wait/signal symbols. Flow expressions are recursively defined in the following manner: if $S$, $S_1$ and $S_2$ are FE's, then so are $S_1 \cdot S_2$ (concatenation), $S_1 \cup S_2$ (union), $S^*$ (the set of all finite length strings), $S^\infty$ (infinite repetition), $S_1 \circ S_2$ (shuffle), and $S^\Box$ (closure of $\Box$). Formal properties of FE's can be used to analyze the system being represented. It has been conjectured [141] that the formal descriptive power of FE's (excluding the $\infty$ operator) lies somewhere below context-sensitive grammar and is incomparable with context-free grammars.

Another approach to the description of parallel processing starts out from the concept of abstract data types or modules, as developed for the structured design of computer software. Rational design methodology (RDM*) [13], [14], [152], [153] being developed by Honeywell embodies this approach. An abstract data module provides a certain set of interface operations which may be executed in interaction with the other modules of the system. Although abstract data types are used in RDM*, a system's behavior is described within a state-transition model. A state is defined by the set of values of data objects at any time. A static description of the system is given through the concept of an abstract machine. An abstract machine contains data types and objects, as well as program text. The static description
then consists of a collection of abstract machines which comprise the processing sites, a machine containing the flow paths, and a machine containing the relevant environment. Each abstract machine has a controller which is termed a *process* when in execution. The flow of control is described by sequences of *events* which have corresponding *invariants*. Thus proof of correctness of program texts in RDM is patterned after inductive assertion techniques.

RDM* was first developed to handle the design of sequential programs operating serially. Thus, a computational model is yet needed for specifying and designing systems of concurrent processes.
We earlier stated that a model should distinguish between the policy to be specified and the mechanisms used to enforce and implement the policy. In this section, we study some mechanisms which have been widely used for implementing synchronization policies. The mechanisms we discuss are busy waiting and memory interlock, the "test and set" instruction, semaphores, critical regions, monitors, and kernels.

It has been shown that with only memory interlock and busy waiting, mutual exclusion can be guaranteed [74]. As a result of the primitiveness of these two mechanisms, the logic of synchronization routines implemented using the two mechanisms can be extremely subtle. Furthermore, busy waiting may not be a desirable feature, and may also lead to the indefinite postponement of a process if not properly used. Memory interlock is actually a form of mutual exclusion; therefore, in effect mutual exclusion is guaranteed by mutual exclusion on certain words. The implementation of synchronization policies using these mechanisms is inefficient. In addition, the burden of writing correct solutions is borne by the programmer.

The test-and-set instruction, as the terminology
implies, tests and sets a word. The two steps involved must be carried out by a single, uninterruptable machine instruction. Although the use of this instruction may result in simpler implementations, the implementations may still involve busy waiting; and memory interlock is implicitly required. Furthermore, the programmer still has to bear the burden of writing correct solutions.

A semaphore is a special type of shared variable upon which several primitive synchronization operations can be performed. Typically, a semaphore consists of two components: (1) an integer counter, which defines the number of signals sent, but not yet received; its initial value defines the initial number of signals, and (2) a queue of processes waiting to receive signals not yet received. Initially, the queue is empty. The operations allowed on semaphores are WAIT and SIGNAL. WAIT allows a process to proceed if the counter is greater than zero and if so, decrements the counter, else the process is put on the queue. SIGNAL removes a process from the queue if any, else increments the counter.

Semaphores can be used to maintain mutual exclusion and control the order in which processes access resources. The semaphore construct is sufficient to solve a wide variety of process synchronization problems, although sometimes with great difficulty. One type of system, whose operations cannot
be fully expressed with a semaphore program, is a system in which the processes do not execute within a single, global environment. The main problems with the semaphore approach are:

1. Primitiveness of the semaphore operations.
2. Total lack of modularity in the programs.
3. Semaphores are shared data, the operations on semaphores must exclude each other in time.
4. Semaphores do not resolve deadlock (they may lead to a deadlock situation).
5. Semaphores force the programmer to be explicitly aware of scheduling details.

Critical regions are structured statements within which concurrent processes can refer to and change shared variables. The processes can only access the shared variable within critical regions. Critical regions are based on the following assumptions:

1. When a process wishes to enter a critical region, it will be enabled to do so within a finite time.
2. At most, one process at a time can be inside a critical region.
3. A process remains inside a critical region for a finite time only.

Critical regions, when nested, may lead to deadlock.
However, critical regions, being a more structured construct, somewhat relieve the programmer from the burden of the details involved. Critical regions, depending on the implementation, may cause a process switch (or context switch) which implicates extra overhead. The assumptions the critical region construct presumes are the essence of the safe access problem. The construct does not provide an efficient method for implementing and guaranteeing these assumptions.

Monitors essentially imply that the user's programs (processes) and devices (resources) do not communicate directly and that all their interactions are passed through an operating system that cannot be interrupted. All procedures which define operations on shared resources are included within the monitor. When a process wishes to access a shared resource, it must do so by executing one of the procedures of the monitor. A process retains exclusive control of the monitor while executing one of the monitor procedures until it surrenders control. All monitor procedures are uninterruptable; hence, mutual exclusion is guaranteed.

The advantage of this approach is its simplicity and its straightforward handling of asynchronism. In addition, monitors provide modularity by restricting the ways in which each process can access shared resources. However, monolithic monitors are not suitable for most operating systems
because of two fundamental problems. The first is that activity in any part of the operating system disables interrupts from all devices. Thus, devices are held up waiting for new commands. The second problem is that tables must be maintained to record the status of each and every device. The size and complexity of these tables can become excessive. In addition, monitors imply a process switch each time a request is made, thus resulting in undesired overhead and inefficiency.

A kernel [74] is a module that implements processes and provides them with a mechanism for interprocess communication. When a process makes a request for a resource, control is transferred to the kernel which in turn enables a resource manager to handle the request. A similar sequence of action takes place when an interprocess communication operation is invoked by a process. For example, if a process invokes a WAIT or SIGNAL operation, the corresponding procedure within the kernel is invoked. A number of procedures or resource managers can be active at one time. The procedures and resource managers are monitors in the sense that one process only can enter a monitor one at a time, however, they can be partitioned in subcomponents such that one component handles incoming requests while another is already executing a previous request.
Although kernels solve some of the problems encountered with monitors, they still do not allow the maximum concurrency (parallelism) possible since one process only is allowed in a monitor. In addition, a process switch is involved every time a procedure or resource manager is invoked. This is actually done for the purpose of protection, only the kernel is allowed direct access to these monitors.

The mechanisms discussed in this section represent a cross-section of those mechanisms introduced at different levels of abstraction. The mechanisms introduced at lower levels of abstraction burden the users with the details and the task of programming the synchronization routines. On the other hand, mechanisms at higher levels of abstraction incur more overhead due to process and context switching; and allow a lower degree of parallelism.
II Production Systems

1. History

Production systems (PS) were first introduced by Post [131] in 1943 as a general computation model. Post proposed a symbol manipulation system which both proved to be extremely powerful computationally and which has served as the basis for some recent psychological models of procedural knowledge.

Production systems as originally proposed by Post consisted of a set of rules, called productions, for rewriting strings of symbols and a specification of some initial strings called axioms. For instance, consider the following simple production system:

Axioms

\[ a, b, aa, bb \]

Productions

\[(P1) x \rightarrow axa\]
\[(P2) x \rightarrow bxb\]

This is a system for writing palindromes, or strings (not including the null string) that read the same forward and backward. The axioms consist of the shortest palindromes and the two productions allow these initial strings to be written into longer ones. The symbol "x" is a variable that may match any arbitrary (non-null) string. By taking
a as the initial string and applying productions \((P_1),(P_2)\) and then \((P_4)\), we obtain: \[a \rightarrow aaa \rightarrow baaab \rightarrow abaaaba.\]

As another example consider the production system which will generate parenthesized arithmetic expressions involving the variables \(a\), \(b\), and \(c\):

\[
\text{Axiom ( )}
\]

\[
\text{Productions (P1)} \quad x_1 ( ) x_2 \rightarrow x_1 (( ) + ( )) x_2
\]

\[
\text{(P2)} \quad x_1 ( ) x_2 \rightarrow x_1 (( ) - ( )) x_2
\]

\[
\text{(P3)} \quad x_1 ( ) x_2 \rightarrow x_1 (( ) / ( )) x_2
\]

\[
\text{(P4)} \quad x_1 ( ) x_2 \rightarrow x_1 (( ) * ( )) x_2
\]

\[
\text{(P5)} \quad x_1 ( ) x_2 \rightarrow x_1 a x_2
\]

\[
\text{(P6)} \quad x_1 ( ) x_2 \rightarrow x_1 b x_2
\]

\[
\text{(P7)} \quad x_1 ( ) x_2 \rightarrow x_1 c x_2
\]

The symbols "\(x_1\)" and "\(x_2\)" are called variables that may match any arbitrary strings. The following gives the derivation of \(((a+b)/c)\)

\[
\text{Axiom ( )}
\]

\[
\text{Production (P3)} \quad (( ) / ( ))
\]

\[
\text{Production (P1)} \quad ((( ) + ( )) / ( ))
\]

\[
\text{Production (P5)} \quad ((a + ( )) / ( ))
\]

\[
\text{Production (P6)} \quad ((a + b ) / ( ))
\]

\[
\text{Production (P7)} \quad ((a + b ) / c )
\]
Post's model has seen a great deal of development and has been applied to a diverse collection of problems. Zisman [156] used PS to model individual processes or events and a Petri net to model the relationship between the processes. The resultant model, called augmented Petri nets is used to represent asynchronous, concurrent processes. Most of the work done using PS has been in the study of coding in human information processing. Anderson [3], Davis [32], Klahr [87], and Newell [124], [125] have developed PS models which are very similar in structure. Newell [124] briefly summarized the model:

"Structurally, the subject is an information processing system (IPS) consisting of a processor containing a short-term memory (STM) which has access to a long-term memory (LTM). The processor also has access to the external environment which may be viewed as an external memory (EM)...

All action of the system takes place via the execution of elementary processes, which take their operands in STM. The only information available upon which to base behavior is that in STM, other information (either in LTM or EM) must be brought into STM before it can effect behavior. At this level the system is serial in nature: only one elementary information process is executed at a time and has available to it the contents of STM as produced by prior elementary processes... The program of the subject appears to be well
represented by a production system. This is a scheme of the form:

\[
C_1 \rightarrow A_1 \\
C_2 \rightarrow A_2 \\
\ldots \\
C_n \rightarrow A_n
\]

Each of the lines consists of a condition \((C_i)\) and an action \((A_i)\), and is called a production. The ordered list of productions is called a production system. The system operates by continually selecting for execution the first action from the top whose condition is satisfied... To provide a complete model for a subject's problem solving requires specifying the memory structures and the symbolic representation, which is implied indirectly in the first two items. On the other hand, strategies and methods of problem solving are to be represented by the contents of production systems, and are not given as separate desiderata.
An example [124] is shown in Figure 5.

Figure 5
The STM holds an ordered set of symbolic expressions. The ordering is important, as will be seen later, because expressions always enter STM at the front and the conditions examine the expression in order starting at the front.

The LTM consists entirely of an ordered set of productions. Each production is written with the condition on the left separated from the action on the right by an arrow. In this system there are four productions: PD1, PD2, PD3 and PD4. Some of the conditions (e.g., that of PD4) consist of only a single symbolic expression (e.g., PD4 has AA); others have a conjunction of two (e.g., PD1 has AA and BB). Some actions consist of a single symbolic expression (e.g., PD3 with BB), some have a sequence of expressions (e.g., PD4 with CC followed by DD), some have expressions that indicate operations to be performed (e.g., the SAY in PD2).

Initially, none of the productions of Figure 4 is satisfied by the contents of STM and nothing happens. However if an AA enters into STM from the external world, AA is shifted into STM and TT is removed. STM now holds

$$AA~QQ~(EE~FF)~RR~SS$$

As a consequence, the only condition of the four productions satisfied is that of PD4, the AA on the left side of
PD4 matching the AA in STM. This leads to the action of
PD4 being evoked, first the CC then the DD. STM now holds:

\[
\text{DD CC AA QQ (EE FF)}
\]

Notice that AA is still in STM but RR and SS have disappeared off the end.

A production (PD4) having been successfully evoked, the system starts the cycle over. PD4 is still satisfied since AA is still in STM. PD3 is also satisfied since the DD matches the DD in STM and (EE) also matches (EE FF) in STM. When two (or more) productions are simultaneously satisfied, the rule for resolving such conflicts is to take the first one in order—here PD3. The result of PD3's action is to put BB into STM as shown below:

\[
\text{BB DD (EE FF) CC AA}
\]

Notice that when PD3 was evoked the two items in its condition moved up to the front of STM in the same order as in the condition. Thus, attended items stay current in STM, while the others drift toward the end, ultimately to be lost. This mechanism provides a form of automatic rehearsal, though it does not preclude deliberate rehearsal.
At the next cycle PD1 is evoked, being the first of the productions satisfied, which includes PD2, PD3, and PD4. The action of PD1 introduces a basic encoding (i.e., construction) operation. (OLD **) is a new expression, which will go into STM like any other. But ** is a variable whose value is the front element in STM. In this case, the front element is AA which was moved up when the condition of PD1 was satisfied. Hence the new element is (OLD AA). This element replaces the front element, rather than simply pushing onto the front. STM now holds.

(OLD AA) BB DD (EE FF) CC

On the next cycle only PD2 is satisfied. Its action involves SAY, which is a primitive operation of the system that prints out the expression following it in the element, i.e., it prints HI.

Since STM is not modified as a result of executing the actions of PD2, the system continues to evoke PD2 and say HI. In general, a production system halts when no production is satisfied or when an action explicitly halts the system.
In Newell's model symbols are inserted at the front of STM. The reason for this restriction is that the purpose of the system is to model human behavior, i.e., as newer elements are inserted older elements in STM get lost. In general, in a PS one may insert symbols at any location in STM and STM may also retain its older contents. Note that STM is assumed finite.

Newell's model has a mixture of parallelism and seriality. While all the available productions can be tested against STM in an unlimited capacity and parallel manner, only one production can be executed in any unit of time. In another model proposed by Anderson [3] multiple productions are allowed to execute in parallel.

Newell's model also shows one way to resolve conflicts among the productions that can be executed, i.e., the first production in order is chosen to be executed. However, McDermott and Forgy [114] explore and evaluate other conflict resolution rules such as special case, recency, distinctiveness and arbitrary decision rules. One may also wish to consider a rule which nondeterministically chooses a production (or a set of productions) to be executed.
In addition, a PS is not limited to certain types of action. The actions may include the construction of new production rules and appending those to the ones already in LTM. This gives the system the ability to "learn" as it executes. The system can also be extended so one can actually tailor the system to his own needs. There are, of course, trade-offs to be considered. We will elaborate on this point in the next section in which we give a formal rigorous definition of production systems.
2. Production Systems and the Synchronization Problem

2.1 Production Systems

A production system is a quintuple \((\Sigma, \mathcal{P}, \mathcal{R}, \chi_0, \beta)\) where

1. \(\Sigma\) is a finite alphabet.
2. \(\mathcal{P}\) is a finite set of productions where each production \(P_i\) is of the form:
   \[ P_i : (C_i \rightarrow A_i) \],
   where
   - \(C_i\) is a set of conditions
   - \(C_{ij} : \Sigma^\beta \rightarrow \{\text{True, False}\}\), and
   - \(A_i\) is a set of actions \(A_{ij} : \Sigma^\beta \rightarrow \Sigma^\beta\).
3. \(\mathcal{R}\) is a partial mapping (possibly not a function) from \((\Sigma^\mathcal{P}) \times \Sigma^*\) into \((\Sigma^\mathcal{P})\) such that \(\mathcal{R}(u,v) = z\) where \(z \subseteq u\), and \(\emptyset = \text{the null set.}\)
4. \(\chi_0 \in \Sigma^\beta\).
5. \(\beta\) is a positive integer.

The elements of the alphabet \(\Sigma\) are to be used to represent the occurrence or absence of "events" and to represent values of "entities". Those elements that represent events are designated as "symbols", and those that represent values of entities are designated as "variables". Variables may take on any one of a finite set of values.
We will use the concept of a working memory (WM) to hold strings from $\Sigma^\beta$. Initially WM contains $\gamma_0 \in \Sigma^\beta$.

The definition of $\Sigma$ given above, provides a flexibility that allows a more compact (in relation to pure production systems in which for each pattern in WM, there is a corresponding production) PS. More importantly, the definition adds some degree of control structure which PSs, in general, lack. This lack of control structure in a PS is a major drawback to the use of PSs for modelling purposes. As will be later demonstrated, we take advantage of this added degree of control structure to the extent that makes the model more attractive and easier to understand.

In the definition of a production, a condition relates to the contents of WM. We place no restriction on the form of these conditions. This, in turn, provides more flexibility and structure. To illustrate, consider the case where for each of a number of conditions the same action is to be executed. Normally, this requires the introduction of a number of productions equal to the number of conditions under consideration. However, with our definition, we are able to group these conditions into one production.

An action in a production may modify WM by deleting, adding or modifying a data element. Since data elements
include variables, and depending on the values of the variables the appropriate action is to be taken, some control of the actions is necessary. This accounts for the introduction of procedural language constructs. We do not introduce any specific construct for the purpose of avoiding restriction and of allowing yet another degree of flexibility in the way the constructs are defined.

As a consequence, the designer is given the decision to determine where his production system should be within the range of production systems representable within our definition from the extreme of a pure PS to a "procedural language" PS. The choice is largely determined by the degree to which the PS is to be used for proving properties of the system being modelled and for deriving implementations. At one extreme, i.e., pure PS, one can make use of properties of PSs to prove properties of the system being modelled. At the other extreme, we have "pure" procedural (programming) language for which present proof techniques [4], [83], [128] are known to be tedious. Our contention is that for a particular system, there is a point between the two extremes which is most suitable.

the conflict resolution, is defined such that it can encompass many (if not all known) conflict resolution rules. Conflict resolution rules which rely on static
conditions, e.g., production order rules and special case rules, can be implemented without extra overhead. In this case the mapping is from \( P \) into \( P \) with the null string as an element of \( \Sigma^* \). Conflict resolution rules for which the conditions are dynamic, e.g., recency rules, use WM as their knowledge source. Noteworthy, the definition permits the simultaneous execution of productions.

\( \beta \) will be used to determine the size of WM. In general there is no restriction on the value of \( \beta \), i.e., it can be infinity. However, for certain classes of production systems, certain restrictions can be placed on the value of \( \beta \).

Our philosophy in developing PSs for the class of problems we consider is to avoid resorting to conflict resolution as much as possible. The productions themselves should be used to resolve conflicts. As a result, proofs, analyses of the PS and implementation derivation will be easier to conduct. As a consequence, in the PSs we develop, \( R \) will only contain the cases for which an element of the domain of \( R \) has more than one image. Identity mappings will be implicitly defined. More intuitively, in our production systems, \( R \) is used to resolve nondeterministic choices in the system being modeled, for instance, the fact that at some point in time a process may terminate or
continue access to a given resource.

For a particular element $\alpha$ in the domain of $R$, if $\alpha$ maps to all subsets of $\alpha$ except the null set, i.e., $\emptyset$, we then use the abbreviated notation $\alpha \rightarrow (2-\emptyset)$ instead of listing all subsets of $\alpha$.

2.2 Operations of the PS Model

Condition elements are templates; when each can be matched by an element in working memory, the production containing them is said to be enabled. We say a condition $C_i$ is satisfied if all $c_{ij}$ become true when applied to the contents of WM.

The production system operates within a control framework called the recognize-act cycle. The recognize phase finds the set of enabled productions, i.e., those productions whose condition elements are satisfied. The act phase executes the enabled productions, performing whatever actions occur in their action sides.

The recognize phase is further divided into match and conflict resolution. In matching, the PS finds the conflict sets, the sets of all enabled productions in the current cycle. In conflict resolution, one or more enabled productions are
selected according to the mapping specified by $\mathcal{R}$ to be executed.

**Definition:** A configuration $\gamma$ of a PS is a string of $(\Sigma \cup \lambda)^\beta$ where $\lambda$ is the "null character."

We say that a PS is in configuration $\gamma$ if the content of working memory is $\gamma$. We write $\gamma \rightarrow \gamma'$ where $\gamma, \gamma'$ are configurations such that $\gamma'$ is the result of executing the selected enabled productions of conflict sets with $\gamma$ in working memory. In general, we write $\gamma_0 \rightarrow^* \gamma_i$ if and only if there are configurations $\gamma_1, \ldots, \gamma_{i-1}$ such that $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \cdots \rightarrow \gamma_{i-1} \rightarrow \gamma_i$, $i \geq 0$. Furthermore, we say that configuration $\gamma_i$ is reachable from configuration $\gamma_0$. Clearly, a configuration is reachable from itself. We also write $\gamma_0 \rightarrow^+ \gamma_i$ to imply one or more transitions, whereas $\gamma_0 \rightarrow^* \gamma_i$ implies zero or more transitions.

We denote $(\gamma \rightarrow \gamma')$ as a transition. We will also label the arrows (→) by the productions that effect the transition wherever appropriate. For instance, $\gamma \xrightarrow{P_1, P_2} \gamma'$ implies that productions $P_1$ and $P_2$ effect the transition from $\gamma$ to $\gamma'$. A label may contain one or more productions.
Definition: Let

\[ \begin{align*}
\mathcal{C}_0 & \xrightarrow{P_0 \ldots P_k} \mathcal{C}_1 \ldots \mathcal{C}_{i-1} & \xrightarrow{P_{i-1} \ldots P_{k_{i-1}}} \mathcal{C}_i,
\end{align*} \]

then the trace in the transitions \( \mathcal{C}_0 \rightarrow^* \mathcal{C}_i \) is the (ordered) sequence:

\[ (P_0^0 \ldots P_k^0)(P_1^1 \ldots P_k^1) \ldots (P_{i-1}^{i-1} \ldots P_{k_{i-1}}^{i-1}) \]

Note that \( 0 \leq k_j \leq n \) for \( 0 \leq j \leq i-1 \), where \( n \) = the number of productions in the system. A trace may be empty (null) since a configuration is (trivially) reachable from itself.

In the next chapter, we will present a particular class of production systems for the purpose of specifying, analyzing and designing synchronization policies with desired properties such as compatibility, liveness, and fairness.

In these production systems, conflict resolution will only be dependent on the set of productions. The mapping \( \mathcal{R} \) will be from \( 2^\mathcal{P} \), i.e., the power set of \( \mathcal{P} \) into \( 2^\mathcal{P} \) itself. \( \mathcal{R} \) does not depend on working memory contents. In addition, the size of working memory will be finite and fixed.
Furthermore, the productions will be nondeterministically chosen to be executed. We will see that this class of production system is sufficient for the representation of the class of problems we are considering.

We now consider an example to illustrate the operation of a production system. The problem is Kosaraju's coordination problem presented in the first chapter as a sample of the class of priority problems not representable with Petri Nets. The problem is restated here for convenience. There are four cyclic processes, $D_1$, $D_2$, $C_1$ and $C_2$ and two buffers $B_1$ and $B_2$. $D_1$ and $D_2$ are producers which place one item each on top of $B_1$ and $B_2$ respectively. However, $C_1$ has higher priority than $C_2$ so that $C_2$ can consume only if $B_1$ is empty.

We will use WM to represent the contents of the buffers. Initially, the buffers are empty. We have to account for the top and bottom of each buffer. Thus, WM will initially contain the string $\$ _1 \not{\epsilon} _1 \$ _2 \not{\epsilon} _2$ of symbols, where $\not{\epsilon}_1$ and $\not{\epsilon}_2$ will serve as pointers to the top of buffers $B_1$ and $B_2$ respectively and $\$ _1$ and $\$ _2$ as pointers to the bottom of $B_1$ and $B_2$ respectively.
Producer D_1 can place an item x_1 on buffer B_1. This is represented by the following productions:

\[ P_{1a} : ((\emptyset) \rightarrow (\text{Replace}(\emptyset, x_1 \emptyset))) \]
\[ P_{1b} : ((\emptyset) \rightarrow (\text{Noop})) \]

Production \( P_{1b} \) allows producer D_1 to idle through a cycle of the production system. Noop stands for the null action. \( R \) includes the conflict set \( r_1 = \{P_{1a}, P_{1b}\} \), i.e., whenever both productions are enabled, either production but not both will be nondeterministically chosen to be executed. Note that \( R \) in this example does not depend on the contents of working memory.

The following productions for producer D_2 can be similarly interpreted.

\[ P_{2a} : ((\emptyset) \rightarrow (\text{Replace}(\emptyset, x_2 \emptyset))) \]
\[ P_{2b} : ((\emptyset) \rightarrow (\text{Noop})) \]

\( R \) includes the conflict set \( r_2 = \{P_{2a}, P_{2b}\} \).

For consumer C_1 to consume an item, B_1 has to contain at least one item. This can be represented as follows:

\[ P_{3a} : ((\$1x_1) \rightarrow (\text{Replace}(\$1x_1, \$1))) \]

Note that \( P_{2a} \) only allows the consumption of the item closest to \( \$1 \), i.e., the bottom-most item on B_1.
We also allow $C_1$ to idle by introducing the production:

$$P_{3b} : (\langle x_1 \rangle \rightarrow \text{(Noop)})$$

$\mathcal{R}$ contains $r_3 = \{P_{3a}, P_{3b}\}$.

In order for $C_2$ to consume, $B_1$ must be empty (no $x_1$ in WM), and $B_2$ must have at least one item. Thus, the productions:

$$P_{4a} : (\langle \text{fail } x_1 \rangle \langle x_2 \rangle \rightarrow \text{(Replace}(x_2, x_2))$$

$$P_{4b} : (\langle \text{fail } x_1 \rangle \langle x_2 \rangle \rightarrow \text{(Noop))}$$

$\mathcal{R}$ contains $r_4 = \{P_{4a}, P_{4b}\}$.

With $\mathcal{G} = \mathcal{G}_2$ in working memory, $P_{1a}, P_{1b}, P_{2a}, P_{2b}$ are enabled. For each conflict set of $\{P_{1a}, P_{1b}\}$ and $\{P_{2a}, P_{2b}\}$ one production is chosen to be executed. $\mathcal{R}$ non-deterministically chooses the productions to be executed. Let the productions chosen for execution be $P_{1a}$ and $P_{2a}$. As a consequence working memory now contains $\langle x_1 \rangle \langle x_2 \rangle$.

In the next cycle, the productions $P_{1a}, P_{1b}, P_{2a}, P_{2b}, P_{3a}$ and $P_{3b}$ are enabled. Let the productions chosen for execution be $P_{1a}, P_{2a}$ and $P_{3a}$. The content of working memory will be $\langle x_1 \rangle \langle x_2 \rangle$.

The operation of this production system continues as illustrated above. As a result of the processes being cyclic, the operation of the production system never halts.
3. Computational Power of Production Systems

In this section we compare the computational power of PS to that of finite-state machines (FSM) and that of Turing machines (TM). We consider FSMs because the class of problems we address in this thesis are related to FSMs. The comparison with Turing machines is intended to illustrate the modelling power of PS. The proofs given below are similar to those of Anderson [3] and Minsky [120].

3.1 Production Systems and Finite-State Machines

Production systems for which:

(1) WM holds a finite number of $\Sigma$-elements.

(2) $\Sigma$ is finite, and

(3) The number of values that a variable in $\Sigma$ can take on is finite, are equivalent to FSMs. The number of configurations possible in such a PS is finite. For example, if the number of symbols and values taken on by variables is $k$ and the number of elements that WM can hold is $m$, then there are $k^m$ configurations. The next configuration of WM is determined (perhaps nondeterministically in some cases) by the current content of WM. The productions enabled and executed are entirely determined by the current configuration of WM.
Lemma For every PS with the three properties above, there is a (nondeterministic) FSM which "simulates" the PS.

Proof (By construction)

A nondeterministic FSM (NFSM) is defined \[88\] as a quintuple \((S, I, O, \Sigma, \lambda)\) where

- \(S\) is a finite, nonempty set of states.
- \(I\) is a finite input alphabet.
- \(O\) is a finite output alphabet.
- \(\Sigma: I \times S \rightarrow 2^S - \emptyset, \emptyset = \) the null set, is the state transition function.
- \(\lambda: S \rightarrow O\) is the output function.

The construction procedure is as follows:

1. Create a distinct state in \(S\) for each possible configuration of \(WM\). Since the number of configurations is finite, \(S\) is finite. Note also that there may be configurations that are not reachable from the initial configuration. Such configurations may be ignored.

2. Create a new element in \(I\) for each distinct input that might enter \(WM\). If multiple symbols enter \(WM\) at once, then the elements of \(I\) will be all strings up to length equal to the size of \(WM\). Elements that may enter \(WM\) are in \(\Sigma\) (or \(\Sigma^*\)).

3. The elements of \(O\) are the operations other than
those that modify WM associated with the production rules.

(4) The transition function specifies that if the NFSM is in a state $s_1$, and an input $i$ is entered, the machine will (nondeterministically) transit to a state of a subset of $S$. To construct $S$, for every distinct configuration $s$ that is reachable from the initial configuration $s_0$, consider the production rules that are enabled and executed with $s$ in WM. For all inserted elements $x_i$ (the input string), $1 \leq i \leq$ size of WM as a result of executing those production rules, form:

$$S(s, x_1 \ldots x_n) = s'$$

where $s,s'$ correspond to the configurations $s$, $s'$ such that $s \rightarrow s'$ in the production system; and $n = \#$ of inserted elements. If no element is inserted, then include:

$$S(s, \lambda) = s'$$

where $s \rightarrow s'$, and $\lambda$ = the null string.

(5) The output function can be constructed as follows:

$$\lambda(s) = y$$

where $s$ corresponds to the configuration $s$, $y$ corresponds to the set of operations $(0_1, \ldots, 0_k)$ such that these operations are executed as a result of executing the corresponding productions with $s$ in WM, and $0_i$, $1 \leq i \leq k$ do not modify WM.
As a result of these considerations:

(1) Synchronization problems only require NFSM's to generate them.

(2) The sequences of permitted operations are NFSM recognizable.

**Theorem**

The set of operations generated by a production system may be viewed as a type 3 (regular) grammar.

**Proof:**

Let the non-terminal symbols of the grammar be $\mathcal{S}_i$ for each reachable configurations of WM. The number of these configurations is finite.

For each configuration $\mathcal{S}_i$, find all $P_i$ enabled $\mathcal{P}$. For each subset $S \in \mathcal{R}(\mathcal{P})$ form the grammar production

$$\mathcal{S}_i \rightarrow o_{i_1} \ldots o_{i_n} \mathcal{S}_j$$

where $o_{i_1} \ldots o_{i_n}$ are operations performed by the productions in $S$ and $\mathcal{S}_j$ is the result of applying all the productions in $S$. All productions of this form suit the criteria for regular grammars, and thus the set of operation sequences is a regular set.
Thus, all sequences generates by a production system can be accepted by a FSM.

Lemma

For every FSM, there is a PS which "simulates" the FSM.

Proof: (By construction)

Let the FSM be \((S, I, 0, s_0, \lambda)\). We will construct a PS \((\Sigma, P, \mathcal{R}, \gamma_0, \beta)\) as follows:

1. Create a new element in \(\Sigma\) for each distinct input in \(I\), and each distinct state in \(S\). WM will contain the present state, and the input and output strings. Initially, WM will contain \((\$\gamma s_0 \ Z\$)\) where \(s_0\) is the initial state of the FSM and \(Z\) is the input string. \$ and \(\gamma\) are special characters. \$ is used to mark the left and right ends of working memory. The symbol to the right of \(\gamma\) is the present state of the FSM. Output will put in between the left \$ and \(\gamma\).

2. To construct \(P\), for each \(x \in I, s_i \in S, s_j \in S\) and \(o_i \in 0\) such that \(\xi(s_i, x) = s_j\) and \(\lambda(s_i) = o_i\), construct the productions:

\([(\xi s_i, x) \rightarrow (\text{Replace}(\xi s_i, x, o_i \gamma s_j))\)]
If $S$ is nondeterministic this will be reflected in the PS by including more than one production with the same conditions, however, with different actions.

(3) To construct $R$, we first note that if the FSM is deterministic, then $R$ is trivially the identity mapping. If the FSM is nondeterministic then $R$ should include the mappings $$\{P_1, \ldots, P_k\} \rightarrow \{\{P_1\}, \{P_2\}, \ldots, \{P_k\}\}$$ where each $P_i$, $1 \leq i \leq k$ corresponds to a state transition $S(s, x) = s_{j_i}$, i.e., there is a nondeterministic transition from $s$ to one of the states $s_{j_i}$, $1 \leq i \leq k$. Whenever $P_1, \ldots, P_k$ are enabled in the same cycle, one of the above mappings is nondeterministically chosen. Note that this is done for every nondeterministic transition.

(4) In general, $\beta$, the size of WM is not bounded.
3.2 Production Systems and Turing Machines

Production systems in which the size of working memory is unbounded are equivalent to Turing machines (TM).

Lemma

For every TM there is a PS which "simulates" the TM.

Proof: (By construction)

Formally a TM \([120]\) is denoted \(T = (S, \Sigma, \Gamma, \delta, \lambda, \theta)\)

where:

- \(S\) is the finite set of states.
- \(\Gamma\) is the finite set of allowable tape symbols. One of these, usually denoted \(B\), is the blank.
- \(\Sigma\), a subset of \(\Gamma\) not including \(B\), is the set of input symbols.
- \(\delta\), the next state function is a mapping from \(S \times \Gamma\) to \(S\).
- \(\lambda\), the output function is a mapping from \(S \times \Gamma\) to \((\Gamma - B)\). The output is to be printed on a scanned square of the tape. \(\theta\), the next move function is a mapping from \(S \times \Gamma\) to \(\{L, R\}\). A move is made either to the left (L) or right (R).
The construction procedure is as follows:

(1) To be able to simulate the TM, the PS needs to encode the initial configuration of the TM. That is, it must encode in WM all the nonblank symbols on the tape, the position of the TM and its state. The nonzero symbols to the left of the TM will be separated from those to the right by the special marker $\uparrow$ ($\uparrow$ is not in $\Gamma$). We assume that the TM is at the symbol (s) to the right of and nearest to $\uparrow$. WM will look like $\underline{x_1 \uparrow x_2 \ $s_i$}$ where $x_1$ and $x_2$ are the strings of symbols to the left and right of the TM respectively, $s_i$ is the state of the TM. The left most $\$ is used to denote the left end of WM. Its purpose will become clear is step 1. The second $\$ is used to separate the representations of the tape content and the machine state. The contents of each square of the tape are stored in one location in WM.

(2) To construct $P$, the set of productions, for each $x$ and $y$ in $\Gamma$ and $s_i$ and $s_j$ in $S$ such that $S(s_i, x) = s_j$, $\lambda(s_i, x) = y$ and $\theta(s_i, x) = R$, let the corresponding production be

$((\underline{Z_1 \ XZ_2 \ $s_i$}) \rightarrow$

(Replace($s_i$, $s_j$))

(Replace($\uparrow x$, $y\uparrow$)))
Similarly, for each $x$ and $y$ in $\Gamma$ and $s_i$ and $s_j$ in $S$ such that 
$\Sigma(s_i, x) = s_j$, $\lambda(s_i, x) = y$ and $\theta(s_i, x) = L$.

Let the corresponding production be 

\[ (\langle Z_1 \uparrow x Z_2 \rangle s_i) \rightarrow (\text{Replace}(s_i, s_j)) \]
\[ (\text{Replace}(\uparrow x, \uparrow y)) \]

(3) Since the right and left halves of the tape are infinite but have only finite nonblank portions, it is possible that the TM might read to the end of the finite nonblank portions. Then the left location (entry) in WM would be of the form " " with no elements and the right location in WM would be state $s_i$. It is necessary to insert another special symbol not in $\Gamma$, for instance, $\#$ so rules in step 2 can apply. Therefore, we include the following production rules:

\[ \langle Z \uparrow z \rangle s_i \rightarrow (\text{Replace}(\uparrow z, \#)) \]
\[ \langle z \uparrow s \rangle s_i \rightarrow (\text{Replace}(\uparrow s, \#)) \]

(4) To construct $R$, if the TM is deterministic, then $R$ is trivially the identity mapping. If the TM is non-deterministic, then for each non-deterministic transition:
\( \xi(s_i, x) = s_{j_k}^l, 1 \leq l \leq k \)

\( \lambda(s_i, x) = y_{j_k}^l \)

\( \theta(s_i, x) = D_{j_k}^l, D_{j_k}^l \in \{L,R\} \)

the corresponding productions \( \{P_1, \ldots, P_k\} \) are enabled. Then \( \mathcal{R} \) should include

\( \mathcal{R}: \{P_1, \ldots, P_k\} \rightarrow \{\{P_1\}, \ldots, \{P_k\}\} \)

(5) In general, we may assume that \( \beta \), the size of \( WM \), is not bounded.

Anderson [3] also shows how a PS with the capability of generating new production rules as actions can be constructed to simulate a TM.
4. Advantages and Disadvantages of Production Systems

We have just shown how to construct a PS system that would generate any input-output relations definable by a TM. We have also determined the class of PS (those with fixed size WM) that is "equivalent" to FSM. These two results demonstrate the computational and modelling power of PS, and the feasibility to relate PS to other models of computation. This, in turn, should help in the analysis of a particular PS. On the other hand, PS systems seem too flexible because some PS system could be proposed to account for almost any behavior.

PS restricts the interactions between rules. Each production makes reference to a data base (WM) common to all productions and no production makes reference directly to other productions.

Unlike most programming languages there are no special facilities for storing control information. All control information is stored in WM which also serves to store the input and output of the computation. The primary effect of the indirect limited interaction is to produce a system which is strongly modular. That is, if a particular production is added, or changed, the basic performance of the system tends to remain relatively unaffected.
The inherent modularity of PS eases the task of programming (using them in specification of a system) in them. This convenience is lessened as one drifts away from the basic "pure" PS to the more complex PS which uses more complex condition scans and conflict resolution strategies.

PS are data-oriented machines. Although this property greatly contributes to the modularity of the system, it also makes the control (behavior) flow of a PS more difficult to analyze. Even for simple tasks, overall behavior of a PS may not be at all evident from a simple review of its rules. While procedural languages are oriented toward the explicit handling of control flow and stress the importance of its influence on the fundamental organization of the program, PSs emphasize the statement of independent data from a domain, and make control flow a secondary issue.

In general, a production rule in a PS is written as a transformation from one string of symbols to another. At one extreme, the symbols may not represent variables, in which case, the number of productions may combinatorially grow. This has been a major deficiency of PS. To overcome this problem, the concept of a variable has been used, i.e., a symbol may represent a number of possibilities. In addition, the introduction of variables may add another degree of structure to the control flow of a PS. Furthermore, this
added degree of structure may facilitate the analysis of a
PS and the specification of the set of actions taken with
respect to a single entity in the system.

PS models are very difficult to read and understand.
This is mainly a consequence of the lack of structure in
the control flow, the way production rules are specified
and the fact that control and data elements are encoded
within a single memory. However, improvement on those two
shortcomings, which can be accomplished as discussed in the
preceeding paragraphs, should alleviate this difficulty.

An advantage of the PS model (the way we define PS) is
the fact that they are self-contained. Input and output
are generated by the system itself. As a consequence, an
analyst does not have to worry about generating all possible
sequences of input since they should be produced by the
system. For a particular sequence of input, it is assumed
that the system nondeterministically chooses the right pro-
duction rules to effect that sequence of input. This pro-
cess is performed within conflict resolution.

PS are well-suited for modelling systems which involve
the existence of multiple, non-trivially different, independ-
ent states; and in which only limited communication between
the action of the system. Of particular interest to us, are
parallel processing systems. In such systems processes are independent entities with limited communication. Processes communicate to exchange information and share resources.
Specification and Analysis of Synchronization Policies
Using Production systems

In this chapter, production systems are used to specify policies of access to shared resources. The production system specifications are analyzed for the following properties: 1) compatibility, 2) liveness and 3) fairness. Furthermore, policies in production system form can be restricted to specify systems of competing concurrent processes which are guaranteed to be compatible, live, and fair using design procedures presented in this chapter.

This chapter is a replica of a paper to be submitted for publication.
Specification and Analysis of Synchronization Policies
Using Production Systems

M. Gazaly
W. Bregar
T. Lewis
M. Freiling

Computer Science Dept.
Oregon State Univ.
Corvallis, OR. 97331
Ph (503) 754-3273

Keywords and phrases: concurrent programming, synchronization, interprocess communication, specification of concurrent processes, reader-writer problem, dining philosopher's problem, deadlock, production system.
Abstract

Production system specification of policies of access to shared resources can be analyzed for the following properties (1) compatibility, (2) liveness (free of deadlock), and (3) fairness (no starvation of one process by another). Furthermore, policies in production system form can be restricted to specify systems of competing concurrent processes which are guaranteed to be compatible, live, and fair using design procedures presented in this paper.
Specification and Analysis of Synchronization Policies Using Production Systems

1. Introduction

Production systems (PS) were first introduced by Post [POST 43]. They have been applied to a diverse collection of problems [ANDE 76], [BREG 80], [DAVI 75], [KLAI 73]. Most of the work done using PS's has been in the study of coding in human information processing [NEWE 72], [NEWE 73].

Zisman [ZISM 78] proposed PS's to model individual processes or events and a Petri net to model the relationship between the processes. The resultant model, called augmented Petri nets is used to represent asynchronous, concurrent processes.

This paper is concerned with the use of PS's as a formal model of policies for synchronizing concurrent systems. The model derived is equivalent to a nondeterministic finite state machine (NFSM), [GAZA 81]. We show that a PS can be used to construct a high level specification for synchronization of accesses to shared resources by concurrent processes.
The purpose of using production systems for specifying synchronization policies is threefold [GAZA 81]:

1. To provide a suitable tool for specifying synchronization policies.

2. To provide a model within which policies can be systematically designed and analyzed.

3. To enable the designer to derive implementations of the policies being specified.

We believe production systems to be easier to use and more suitable than Petri nets [LAUT 74], [PETE 77], parallel program schemata [KARP 69], and others [BAER 73], [CAMP 74], [KARP 66], [KELL 76], [LEWI 80], [LIPT 75], [MILL 73], [SHAW 78]. In this paper we present design procedures which systematically construct production systems with certain desired properties. Production systems, as an analysis medium, are comparable to Petri nets. The analysis procedures are based on concepts similar to the reachability tree [PETE 77] of Petri nets. Finally, systematic procedures for deriving implementations of policies can be constructed [GAZA 81]. The basic idea here is to generate synchronization procedures from the actions of a production system. The procedures can then be executed by competing processes without failures due to incompatibility starvation, or deadlock.
We define a policy as a plan to be carried out by a system. A specification is a written description of a policy. We will adopt PS notation as a language for specifying policies.

In this paper, we present procedures for analyzing and designing policy specifications with respect to three major goals of synchronization policies: (1) compatibility, (2) liveness, and (3) fairness.

In section 2, we define production systems and show how they can be used to model synchronization policies. In section 3, we present an algorithm for determining whether or not a policy specification is compatible. An algorithm for constructing compatible policies is also given. In sections 4 and 5, we investigate liveness and fairness and present algorithms for determining whether or not these properties hold for a given policy specification. Algorithms for constructing live and fair policies are also presented. In section 6, we briefly discuss the use of production systems as a model of concurrent processing, evaluate the analysis and design algorithms derived, and consider some problems for further research.
2. Production Systems and the Synchronization Problem

2.1 Production Systems

A PS is a quintuple \((\Sigma, \mathcal{P}, \mathcal{R}, \mathcal{C}_0, \beta)\) where

1. \(\Sigma\) is a finite alphabet.
2. \(\mathcal{P}\) is a finite set of productions where each production \(P_i\) is of the form:
   \[ P_i : (C_i \rightarrow A_i), \]
   where \(C_i\) is a set of conditions
   \[ c_{ij} : \Sigma^\beta \rightarrow \{\text{True, False}\}, \]
   and \(A_i\) is a set of actions \(a_{ij} : \Sigma^\beta \rightarrow \Sigma^\beta\).
3. \(\mathcal{R}\) is a set of conflict sets \(r_i\) where each \(r_i\) is a subset of \(\mathcal{P}\), the set of productions.
4. \(\mathcal{C}_0 \in \Sigma^\beta\).
5. \(\beta\) is a positive integer.

The elements of the alphabet \(\Sigma\) are to be used to represent the occurrence or absence of "events" and to represent values of "entities". Those elements that represent events are designated as "symbols", and those that represent values of entities are designated as "variables". Variables may take on any one of a finite set of values.

We will use the concept of a working memory (WM) to hold strings from \(\Sigma^\beta\). Initially WM contains \(\mathcal{C}_0 \in \Sigma^\beta\).
We say a condition $C_i$ is satisfied if all $c_{ij}$ become true when applied to the contents of WM. Furthermore, we say a production $P_i: (C_i \rightarrow A_i)$ is enabled when $C_i$ is satisfied.

The production system operates within a control framework called the recognize-act cycle. The recognize phase is further divided into match and conflict resolution. In matching, the PS finds the conflict sets, the set of all enabled productions in a recognize-act cycle. In conflict resolution, one or more enabled productions are selected according to $R$ as explained in the following paragraph. The act phase executes the selected enabled productions, by performing their associated actions, i.e., applying each $a_{ij}$ to WM.

Each $r_i$ in $R$ represents a conflict set. For each $r_i$, one production is nondeterministically chosen from the enabled productions in $r_i$. The chosen productions are executed simultaneously. In the systems we consider, the order of the execution of productions will not affect the contents of WM.

We say that a PS is in configuration $\mathcal{C}$ if the contents of working memory is $\mathcal{C}$. We denote $\mathcal{C} \xrightarrow{\mathcal{R}} \mathcal{C}'$ as a transition where $\mathcal{C}, \mathcal{C}'$ are configurations such that $\mathcal{C}'$ is the result of executing the selected enabled productions with $\mathcal{C}$ in working memory. In general, we write $\mathcal{C}_0 \xrightarrow{\mathcal{R}} \mathcal{C}_i$ iff
there are configurations $\mathcal{C}_1, \ldots, \mathcal{C}_{i-1}$ such that

$\mathcal{C}_0 \rightarrow \mathcal{C}_1 \rightarrow \mathcal{C}_2 \cdots \rightarrow \mathcal{C}_{i-1} \rightarrow \mathcal{C}_i$, $i \geq 0$.

Furthermore, we say that configuration $\mathcal{C}_i$ is reachable from configuration $\mathcal{C}_0$. Clearly, a configuration is reachable from itself. We also write $\mathcal{C}_0 \rightarrow^+ \mathcal{C}_i$ to imply one or more transitions.

We also label the arrows ($\rightarrow$) with the productions that effect the transition wherever appropriate. For instance, $\mathcal{C} \xrightarrow{P_1, P_2} \mathcal{C}'$ implies that productions $P_1$ and $P_2$ effect the transition from $\mathcal{C}$ to $\mathcal{C}'$. A label may contain one or more productions.

Let $\mathcal{C}_0 \xrightarrow{P_0^1 \cdots P_0^k} \mathcal{C}_1 \cdots \mathcal{C}_{i-1} \xrightarrow{P_{i-1}^1 \cdots P_{i-1}^k} \mathcal{C}_i$.

then the trace in the transitions $\mathcal{C}_0 \rightarrow^* \mathcal{C}_i$ is the ordered sequence $(P_0^1 \cdots P_0^k)(P_1^1 \cdots P_1^k) \cdots (P_{i-1}^1 \cdots P_{i-1}^k)$. Note that $0 \leq k \leq n$ for $0 \leq j \leq i-1$, where $n$ is the number of productions in the system. A trace may be empty (null) since a configuration is (trivially) reachable from itself.
In this paper, a condition of the form \((x)\) will be interpreted as follows. If working memory contains \(x\) the condition will be true, otherwise false. A condition of the form \((\neg x)\) will be true if working memory does not contain \(x\), otherwise the condition will be false.

In addition, the actions permitted will be Insert, Delete, Replace, Assign and Noop. Insert\((x)\) inserts \(x\) into working memory, i.e., \(x\) will be present in working memory and the condition \((x)\) will be true whereas \((\neg x)\) will be false. Analogously, Delete\((x)\) deletes \(x\) from working memory. Replace\((x, y)\) deletes \(x\) from working memory and inserts \(y\) into working memory. The actions Insert, Delete and Replace operate on symbols.

The action Assign operates on variables. Assign\((x)\) assigns a value to \(x\). Assign can be viewed as a function from \(\Sigma^\beta\) to \(\Sigma^\beta\). We will use Assign to define the priorities of requests and accesses. If Assign uses "counters", e.g., to represent the last request assigned a priority in a FCFS priority scheme, we assume that such counters will be encoded in working memory.

Finally, Noop leaves the content of working memory unchanged, i.e., it may be interpreted as the identity mapping from \(\Sigma^\beta\) to \(\Sigma^\beta\).
Example 1

Let PS = (\(\Sigma, \mathcal{P}, \mathcal{R}, \gamma_0, \beta\)) where 
\(\Sigma = \{I_1, I_2, I_3, R_1, R_2, R_3, A_1, A_2, A_3\}\). In this example, all elements of \(\Sigma\) are symbols.

\(\mathcal{P}\) contains the following productions.

\(P_{1a} : (\text{(I}_1\text{)(fail R}_1\text{)(fail A}_1\text{)} \rightarrow \text{(Insert R}_1\text{))}

\(P_{1b} : (\text{(I}_1\text{)(fail R}_1\text{)(fail A}_1\text{} \rightarrow \text{(Noop))}

In the above productions the condition (I\(_1\)) will be true if WM contains I\(_1\). The conditions (fail R\(_1\)) and (fail A\(_1\)) will be true if WM does not contain R\(_1\) and A\(_1\) respectively. The action (Insert(R\(_1\))) inserts R\(_1\) into working memory, i.e., R\(_1\) is made. The action Noop stands for the null action.

The following productions can be similarly interpreted.

\(P_{2a} : (\text{(I}_2\text{)(fail R}_2\text{)(fail A}_2\text{)} \rightarrow \text{(Insert R}_2\text{))}

\(P_{2b} : (\text{(I}_2\text{)(fail R}_2\text{)(fail A}_2\text{} \rightarrow \text{(Noop))}

\(P_{3a} : (\text{(I}_3\text{)(fail R}_3\text{)(fail A}_3\text{)} \rightarrow \text{(Insert R}_3\text{))}

\(P_{3b} : (\text{(I}_3\text{)(fail R}_3\text{)(fail A}_3\text{} \rightarrow \text{(Noop))}

\(P_{4a} : (\text{(R}_1\text{)} \rightarrow \text{(Replace (R}_1\text{, A}_1\text{))})

\(P_{4b} : (\text{(R}_1\text{)} \rightarrow \text{(Noop))}

\(P_{5a} : (\text{(R}_2\text{)} \rightarrow \text{(Replace (R}_2\text{, A}_2\text{))})

\(P_{5b} : (\text{(R}_2\text{)} \rightarrow \text{(Noop))}
The action (Delete $A_i$) deletes $A_i$ from WM.

$\mathcal{R}$, the set of conflict sets, contains the following sets: $r_1 = \{P_{1a}, P_{1b}\}$, $r_2 = \{P_{2a}, P_{2b}\}$, $r_3 = \{P_{3a}, P_{3b}\}$, $r_4 = \{P_{4a}, P_{4b}\}$, $r_5 = \{P_{5a}, P_{5b}\}$, $r_6 = \{P_{6a}, P_{6b}\}$, $r_7 = \{P_{7a}, P_{7b}\}$, $r_8 = \{P_{8a}, P_{8b}\}$, $r_9 = \{P_{9a}, P_{9b}\}$.

$\chi_0$, the initial configuration, is $I_1 I_2 I_3$, and $\beta = 6$, i.e., the maximum number of elements that WM holds.

Since WM initially contains $I_1 I_2 I_3$, the conditions for $P_{1a}, P_{1b}, P_{2a}, P_{2b}, P_{3a}$ and $P_{3b}$ are satisfied. Thus, with $\chi_0$ in WM the productions $P_{1a}, P_{1b}, P_{2a}, P_{2b}, P_{3a}$ and $P_{3b}$ are enabled.

For each of the conflict sets $r_1 = \{P_{1a}, P_{1b}\}$.
\( r_2 = \{ p_{2a}, p_{2b} \}, \text{ and } r_3 = \{ p_{3a}, p_{3b} \} \), \( R \) nondeterministically selects one production from each conflict set to be executed. Let the productions selected for execution be \( p_{1a}, p_{2b} \) and \( p_{3a} \). As a result the transition:
\[
\chi_0 = I_1 I_2 I_3 \xrightarrow{p_{1a}, p_{2b}, p_{3a}} R_1 I_1 I_2 R_3 I_3 = \chi_1 \text{ occurs.}
\]

In the next cycle, with \( \chi_1 \) in WM, the productions enabled are \( p_{4a}, p_{4b}, p_{2a}, p_{2b}, p_{6a} \) and \( p_{6b} \). Let the productions selected for execution be \( p_{4a}, p_{2a} \) and \( p_{6a} \). The transition:
\[
\chi_1 = R_1 I_1 I_2 R_3 I_3 \xrightarrow{p_{4a}, p_{2a}, p_{6a}} A_1 I_1 R_2 I_2 A_3 I_3 = \chi_2 \text{ is effected.}
\]

In the following cycle, the productions enabled are \( p_{7a}, p_{7b}, p_{5a}, p_{5b}, p_{9a} \) and \( p_{9b} \). Let \( p_{7a}, p_{5b} \) and \( p_{9a} \) be selected for execution. The transition:
\[
\chi_2 = A_1 I_1 R_1 I_2 A_3 I_3 \xrightarrow{p_{7a}, p_{5b}, p_{9a}} I_1 R_2 I_2 I_3 = \chi_3 .
\]

The operation of this PS continues as demonstrated above.
2.2 Production Systems as Policy Specifications

In general, a synchronization policy involves a finite number of requests of different types made to different resources by different processes. A process is a sequence of operations performed one at a time. Two processes are said to be concurrent if their operations can either overlap or interleave arbitrarily in time [TAUS 77]. When a request is "honored", it is referred to as an access. A priority may be associated with a request or access. The occurrence and honoring of requests is governed by a set of specification requirements involving the requests and accesses. The problem is to specify the policy so that the specification requirements are enforced and not violated.

A production system is to specify the behavior of a system of processes running in a common environment which includes shared resources. The production system should specify the behavior even when there are "illegal" operations performed within the system. From this specification of behavior, a controller, centralized or distributed, may be generated to supervise actual processes in such a way as to ensure that the specified behavior actually occurs [GAZA 81].
We will use a PS to specify a policy for synchronization of processes in the following manner.

(1) The working memory represents an instantaneous description of the state of the resources and processes with respect to (pending) requests, accesses being made and other related information such as priorities.

(2) The productions will represent the occurrence of process requests and accesses under the proper conditions.

(3) Conflict resolution models the nondeterminacy of the occurrence of requests, granting accesses and termination of accesses.

Without loss of generality, we also restrict the representation of the system being modeled as follows. Requests and accesses will be represented by symbols. Priorities of requests and accesses will be represented as variables. As a consequence, the operations permitted on request and access symbols are Replace, Delete, and Insert. Furthermore, a condition representing the presence or absence of a symbol x are (x) or (fail x) respectively. Variables can be assigned values. Conditions involving variables are represented by \((v_1 \text{ RE } v_2)\) where \(v_1\) and \(v_2\) are variables or constants. "RE" is a relational operator, e.g., greater than, equal to, ... etc. A condition may involve both symbols and variables. For example, the condition \((x_1 \wedge x_2 \wedge (p_1 > p_2))\) where \(x_1\) and \(x_2\) are symbols; and \(p_1\) and \(p_2\) are variables is true when WM contains \(x_1, x_2, p_1,\) and \(p_2\); and the value of \(p_1\) is greater than that of \(p_2\).
As an example, consider the production system presented in example 1. The symbols $R_1$, $R_2$ and $R_3$ can be interpreted as requests to access the same resource by processes 1, 2, and 3. $A_1$, $A_2$ and $A_3$ can be interpreted as the corresponding accesses. The productions can be interpreted as the manipulation of the requests and accesses in the system. Productions $P_{ia}$, $P_{ib}$, $1 \leq i \leq 3$ represent the occurrence of requests $R_1$, $R_2$ and $R_3$ respectively. A request $R_i$ can occur if $R_i$ and $A_i$ are not already in WM. Productions $P_{ia}$ and $P_{ib}$, $4 \leq i \leq 6$ allow requests $R_1$, $R_2$, and $R_3$ to be satisfied. A request $R_i$ can be satisfied if WM contains $R_i$. Similarly productions $P_{ia}$ and $P_{ib}$, $7 \leq i \leq 9$ allow requests $R_1$, $R_2$ and $R_3$ to terminate or continue in progress. $R_i$, the set of conflict sets allows each process to proceed at its own rate.

When we think of a production system as a model for a resource manager, it is important to remember that the production system represents a specification of the behavior of the manager. If we consider a transition sequence: $\gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_n$ where some productions

$P_a: ((R_i) c_1 c_2 \cdots c_k \rightarrow (Replace(R_i, A_i)) a_1 \cdots a_m)$

$P_b: ((R_i) c_1 c_2 \cdots c_k \rightarrow (Noop))$

are enabled in $\gamma_1$, $P_a$ is executed in the transition
\( \mathcal{G}_{n-1} \rightarrow \mathcal{G}_n \), and \( P_b \) being executed in the transitions \( \mathcal{G}_1 \rightarrow \mathcal{G}_{n-1} \). If the manager is distributed as a set of processes, then in \( \mathcal{G}_{n-1} \) some process has decided to grant access to process \( i \). This does not mean, however, that the granting process started in \( \mathcal{G}_1 \). It could have started much later, i.e., all that is implied is that \( \mathcal{G}_1 \) is the earliest stage at which the granting process makes a decision.

The above consideration will allow us to design policies that are independent of speed variations among the processes. This independence is not only achieved for the grant-access operation but also for the operations of making a request and terminating an access.

The configurations of a production system are not explicitly specified within the definition of production systems. The analysis procedures, to be presented in the following sections, test configurations and sequences of transitions to determine whether or not a production system satisfies a particular property. Thus, determining the reachable configurations and the possible sequences of transitions is necessary. We will perform this task by constructing a transition graph for the production system under consideration. The nodes of the graph will represent the reachable configurations, and the arcs the transitions
between those configurations.

The transition graph can be constructed by implementing a production system monitor (ROOD 80) which will simulate all possible transitions and produce the reachable configurations. The transition graph can be constructed from the transitions and the reachable configurations produced. The basic steps involved in this implementation are as follows:

1. Starting with the initial configuration, for every reachable configuration \( \gamma \) determine all enabled productions with \( \delta \) in WM.

2. Construct all possible combinations of the enabled productions choosing one production from each conflict set.

3. For every combination constructed in step 2 execute the corresponding actions, and include the transition and the resulting configuration in the transition graph.

To illustrate this approach, consider the production system \( \mathcal{PS} = (\Sigma, \mathcal{P}, \mathcal{R}, \mathcal{E}_0, \beta) \) where \( \Sigma = \{I_1, I_2, R_1, R_2\} \) and \( \mathcal{P} \) contains the following productions:

\[
P_{1a}: (\langle I_1 \rangle)(\text{fail } R_1)(\text{fail } A_1) \rightarrow (\text{Insert } R_1))
\]

\[
P_{1b}: (\langle I_1 \rangle)(\text{fail } R_1)(\text{fail } A_1) \rightarrow (\text{Noop})
\]

\[
P_{2a}: (\langle I_2 \rangle)(\text{fail } R_2)(\text{fail } A_2) \rightarrow (\text{Insert } R_2))
\]

\[
P_{2b}: (\langle I_2 \rangle)(\text{fail } R_2)(\text{fail } A_2) \rightarrow (\text{Noop})
\]

\[
P_3 : (\langle R_1 \rangle)(\text{fail } A_2) \rightarrow (\text{Replace}(R_1, A_1)))
\]
\[ P_4: \quad ((R_2)\text{fail } R_1)\text{fail } A_1) \rightarrow (\text{Replace}(R_2, A_2)) \]

\[ P_5: \quad ((A_1) \rightarrow (\text{Delete } A_1)) \]

\[ P_6: \quad ((A_2) \rightarrow (\text{Delete } A_2)) \]

\( R \) contains the conflict sets \( r_1 = \{P_{1a}, P_{1b}\}, \)
\( r_2 = \{P_{2a}, P_{2b}\} \) and \( r_1 = \{P_i\} \ 3 \leq i \leq 6, \)
\( \gamma_0 = I_1 I_2 \) and \( \beta = 4. \)

Starting with the initial configuration \( \gamma_0 = I_1 I_2, \)
the reachable configurations are \( \gamma_1 = R_1 I_1 I_2, \)
\( \gamma_2 = I_1 R_2 I_2, \) and \( \gamma_3 = R_1 I_1 R_2 I_2. \) We construct the corresponding nodes and transitions in the transition graph. The reachable configurations from \( \gamma_1 \) are
\( \gamma_4 = A_1 I_1 I_2 \) and \( \gamma_5 = A_1 I_1 R_2 I_2, \) and we include the nodes \( \gamma_4 \) and \( \gamma_5 \) and the appropriate transitions in the graph. Continuing in this fashion, we end up with the transition graph shown in Figure 6. The complexity of constructing the transition graph is, in general, of combinatorial complexity in terms of the number of accesses and requests in the system.
Figure 6
3. Specification of Compatible Policies

Throughout this paper we will use the following notation. The symbol $R_{i,j,k}$ denotes a request of type $i$ to resource $j$ by process $k$. The symbol $A_{i,j,k}$ denotes an access of type $i$ to resource $j$ exercised by process $k$. The variable $p_{i,j,k}$ denotes the priority associated with the request $R_{i,j,k}$ or access $A_{i,j,k}$. We will show all subscripts on the $R$'s, $A$'s and $p$'s only when it is appropriate.

The priorities of requests and accesses can be represented as a precedence matrix $PM$ which is encoded in $WM$. The rows and columns of $PM$ are the requests and accesses and the entries are the operations $>$, $=$, $<$, i.e., higher than, equal to, and less than respectively. Each row is represented with one location of $WM$. This representation permits the priorities of a set of requests and accesses $S$ to be cyclic. For example, assuming that the priorities of requests $R_1$, $R_2$, ..., and $R_N$ are $p_1$, ..., and $p_N$ respectively, the priorities are cyclic if $p_1 > p_2$, $p_2 > p_3$, ..., and $p_N > p_1$.

The purpose of priorities is to resolve contention of an incompatible set of accesses and requests. Priorities, in general, can be dynamically assigned. In the production systems we develop, we will use the action "Assign" to
assign the priority of a request with respect to other requests and accesses in the system. Whenever no restriction is placed on the priorities, we assume that the action "Assign" is defined by the user. When we place restrictions on priorities in the production system designed to satisfy a certain property, we also assume that the user is to specify the action Assign with the proper restrictions.

3.1 Definitions

A set $S$ of accesses is incompatible if it contains accesses which are not permitted to occur simultaneously; it is compatible otherwise. An incompatible set of accesses $S$ is minimal if no proper subset of $S$ is incompatible.

A set $Q = \{R_{i_1,j_1,k_1}, \ldots, R_{i_n,j_n,k_n}\}$ of $n$ requests is incompatible if its corresponding set of accesses $\{A_{i_1,j_1,k_1}, \ldots, A_{i_n,j_n,k_n}\}$ is incompatible; otherwise $Q$ is compatible.

A PS configuration $\mathcal{C}$ is incompatible if it contains an incompatible set of accesses; otherwise it is compatible. A production system itself is incompatible if an incompatible configuration is reachable from its initial configuration; otherwise it is compatible.
We assume that a synchronization problem is stated in terms of the minimal incompatible sets of requests (or accesses). That is, given the minimal incompatible sets, we are to analyze or design production systems which model the policy under consideration.

For many synchronization problems, it is sufficient to merely state the minimal incompatible sets in order to fully specify the problem. Some problems, however, the processes involved require access to several resources simultaneously. As a result, it is not possible to assume the problem solved if "fair" access is granted to each resource independent of the others. We will discuss this problem further when we consider solutions to the dining philosophers problem and when we consider fairness in section 5.
3.2 Determining Compatibility

We assume that the compatibility problem is posed by stating the minimal incompatible sets. Algorithm COMPAT, Figure 7, determines the compatibility of a production system which specifies a synchronization policy. The algorithm utilizes a function COMPCON, Figure 8, to check for the compatibility of a configuration. It is assumed that the reachable configurations are obtained from the transition graph.

Example 2

Consider the PS presented in example 1. Let the minimal incompatible sets be \( \{A_1, A_3\} \) and \( \{A_2, A_3\} \). We will show that PS is incompatible.

Consider the configuration \( \mathcal{S} = A_1 I_1 I_2 A_3 I_3 \). \( \mathcal{S} \) is reachable from \( \mathcal{S}_0 \) through the sequence of transitions

\[
\mathcal{S}_0 = I_1 I_2 I_3 \rightarrow R_1 I_1 I_2 R_3 I_3 \rightarrow A_1 I_1 I_2 A_3 I_3 = \mathcal{S}.
\]

Using algorithm COMPAT, testing the compatibility of \( \mathcal{S} \), the function COMPCON returns a "false" value, and the PS is incompatible.
procedure COMPAT (PS: a production system, S: minimal incompatible sets of accesses)

/* Algorithm COMPAT determines if a PS is compatible. */
/* Input: PS and S. */
/* Output: indicates whether the PS is compatible or not. */

begin
    for every reachable configuration
        if not COMPCON (Ø, S)
            then PS is incompatible, stop.
        endif
    endfor
endbegin

Figure 7

function COMPCON(Ø: a configuration, S: minimal incompatible sets of accesses)

/* Algorithm COMPCON determines if a configuration is compatible. */
/* Input: Ø and S. */
/* Output: COMPCON is set to "true" if Ø is compatible, otherwise "false". */

begin
    COMPCON = true
    for every s in S
        if s is in Ø
            then
                COMPCON = false
                return
        endif
    endfor
endbegin

Figure 8
3.3 Designing Compatible Production Systems

We first introduce the concept of the B-set and C-set for a particular request, in order to determine the conditions under which a request can be satisfied.

Definitions

The blocking set (B-set) for a particular request $R_{i,j,k}$ is the set of all sets of accesses $S_m$ such that $S_m \cup \{A_{i,j,k}\}$ is a minimal incompatible set. For example, if WM contains the request $R_{i,j,k}$ and a set of accesses $S_m$, then granting the access $A_{i,j,k}$ will result in a minimal incompatible set with $S_m$ in WM.

The contention set (C-set) for a particular request $R_{i,j,k}$ is the set of all sets of requests and accesses $S_m$ such that:

1. Each set $S_m$ contains at least one request.
2. For each $S_m'$, $S_m' \cup \{A_{i,j,k}\}$ is a minimal compatible set where $S_m'$ is formed from $S_m$ by replacing each request $R_{r,s,t}$ in $S_m$ with its corresponding access $A_{r,s,t}$.

The intuitive interpretation of the C-set is that satisfying $R_{i,j,k}$ and all requests in an element of C-set ($R_{i,j,k}$)
will result in an incompatible set of accesses.

If $\mathcal{G}$ is a configuration containing a request $R_{i,j,k}$ and $\mathcal{G}_0 \to^* \mathcal{G}$ and $\mathcal{G} \to^* \mathcal{G'}$ where $\mathcal{G}_0$ is the initial configuration, and $\mathcal{G'}$ contains $A_{i,j,k}$, we then say that $R_{i,j,k}$ is satisfied in the transitions $\mathcal{G} \to^* \mathcal{G'}$.

Algorithms BSETS and CSETS, shown in Figure 9 and Figure 10, construct the B-sets and C-sets, respectively, for a set of requests.

Algorithm COMPS, shown in Figure 11 constructs a compatible production system. The algorithm permits a process $k$ to make a request if $k$ has no other pending requests. A request $R$ is satisfied if both the following two conditions hold:

1. There is no element of B-set (R) in WM, thus avoiding incompatibility, and

2. There is no element $x$ of C-set (R) in WM such that all requests in $x$ have higher priority than $R$. In this case, compatibility is guaranteed by blocking $R$ if it has the lowest priority of a set of contending requests.

Once a request is satisfied, process $k$ can terminate the corresponding access if it has no pending requests.
A process can make a request or terminate an access at its own pace. The productions with the Noop action reflect this consideration.
procedure BSETS (Q: set of requests, M: minimal incompatible sets of accesses, B: B-sets for requests in Q)
begin
/* Algorithm BSETS calculates the B-sets for a set of requests. */
/* Input: the set of requests, and M, the minimal incompatible sets of accesses (or requests). */
/* Output: B, the B-sets for all requests */
for every request Ri,j,k
  for every minimal incompatible set S of accesses.
    if Ai,j,k is in S
      then include the set S' = S - {Ai,j,k} in the B-set of Ri,j,k.
  endif
endfor
endfor
endbegin

Figure 9
procedure CSETS (Q: set of requests, M: minimal incompatible sets of accesses, C: C-sets for requests in Q)

begin

/* Algorithm CSETS calculates the C-sets for a set of requests. */
/* Input: Q, the set of requests, and M, the minimal incompatible sets of accesses */
/* (or requests). */
/* Output: C, the C-sets for all requests. */

for every request R_{i,j,k}
  for every minimal incompatible set S of accesses.
    if A_{i,j,k} is in S
      then
        Let S' = S - \{A_{i,j,k}\}.
        Generate all sets S_l, 1 ≤ l ≤ 2^{S'} - 1 such that:
        (1) For every A_{l,m,n} in S', each S_l contains either A_{l,m,n} or R_{l,m,n}.
        (2) Each S_l contains at least one request.
        Include each S_l in the C-set of R_{i,j,k}.
      endif
  endfor
endfor
endbegin

Figure 10
procedure COMPS (Q: set of requests, S: minimal incompatible sets of accesses, PS: production system)

/* Algorithm COMPS constructs a compatible PS. */
/* Processing of requests is governed by the corresponding B-sets and C-sets. */
/* Input: Q and S. */
/* Output: PS, a compatible production system. */

begin

(1) Include the symbols $I_k$, $R_{i,j,k}$, $A_{i,j,k}$ and variables $p_{i,j,k}$ for all relevant values of $i,j$ and $k$ in the WM alphabet $\Sigma$.
Let the initial contents of WM be $I_k \ I_k \ I_k \ldots \ I_k \ I_k$, where $n$ = the number of processes. This represents the fact that no requests or accesses exist yet. In general, the initial configuration may contain information pertinent to the resources.

(2) for every process $k$

Let $R_{i_1,j_1,k}, \ldots, R_{i_m,j_m,k}$ be all the requests of process $k$.

a. for every request $R_{i,j,k}$

Include the following productions in $\mathcal{P}$, the set of productions, to allow for the occurrence of $R_{i,j,k}$.

$$
((I_k)(\text{fail } R_{i_1,j_1,k})(\text{fail } R_{i_2,j_2,k})\ldots
(fail R_{i_m,j_m,k})(\text{fail } A_{i,j,k})
\rightarrow (\text{Insert } R_{i,j,k}) \ (\text{Assign } p_{i,j,k})
\rightarrow (\text{Assign } p_{i,j,k})
$$

Figure 11
The first of the above two productions allows a process k to make a request \( R_{i,j,k} \) if the process has no outstanding requests and the process is not already exercising access \( A_{i,j,k} \). The action Assign is to be supplied by the user in accordance to the restrictions in the lemma which will follow shortly. The second production allows the process to idle if it is chosen to be executed.

endfor

Include all above productions as one conflict set in \( R \), the set of conflict sets.

b. for every request \( R_{i,j,k} \)

Construct the B-set and C-set for \( R_{i,j,k} \) using algorithms BSETS and CSETS.

for every element \( x \) in B-set \( (R_{i,j,k}) \)

Let \( A_{i_1,j_1,k_1}, \ldots, A_{i_l,j_l,k_l} \) be the accesses in \( x \).

Construct a condition \( c \) of the form

\[
\text{fail}(A_{i_1,j_1,k_1} \land \cdots \land A_{i_l,j_l,k_l})
\]

endfor

Let \( C_1 = (c_1, c_2, \ldots, c_L) \) be the set of such conditions.

for every element \( x \) in C-set \( (R_{i,j,k}) \)

Let \( R_{i_1,j_1,k_1}, \ldots, R_{i_l,j_l,k_l} \) be the requests in \( x \); and

\( A_{i_1,j_1,k_1}, \ldots, A_{i_l,j_l,k_l} \) the accesses in \( x \).

Figure 11 continued
Construct a condition $c$ of the form
\[(\text{fail} (A_{i_1,j_1,k_1} \land \cdots \land A_{i_l,j_l,k_l}) \land R_{i_1,j_1,k_1} \land \cdots \land R_{i_r,j_r,k_r} \land (p_{i_1,j_1,k_1} > p_{i,j,k}) \land \cdots \land (p_{i_l,j_l,k_l} > p_{i,j,k}))\]
endfor

Let $C_2 = \{c_1, c_2, \ldots, c_M\}$ be the set of all such conditions.

Include the following productions in $\mathcal{P}$ to satisfy $R_{i,j,k}$:

\[
((R_{i,j,k}) \ c_1 \ c_2 \ \cdots \ c_L \ c_1 \ \cdots \ c_M \rightarrow (\text{Replace}(R_{i,j,k}, A_{i,j,k}))
 \ 
 (\text{Assign} \ p_{i,j,k}))
 (\text{Noop}))
\]

In the first of the above productions, a request is satisfied if no elements of its B-set or C-set with higher priorities are in WM. Thus, no incompatible sets of accesses will occur. The purpose of the action Assign here is to associate a priority with access $A_{i,j,k}$, which may be different from that of request $R_{i,j,k}$.

The second production allows granting the access to occur at any time (cycle) whenever the conditions of the production hold.

Include the above two productions as one conflict set in $\mathcal{R}$.
endfor

Figure 11 continued
c. for every access $A_{i,j,k}$
   Include the following productions in $P$
   to allow an access to terminate or to
   continue in progress.

   $$(A_{i,j,k})(\text{fail } R_{i_1,j_1,k})...$$
   $$(\text{fail } R_{i_m,j_m,k}) \rightarrow \text{(Delete } A_{i,j,k})$$

   $$(A_{i,j,k})(\text{fail } R_{i_1,j_1,k})...$$
   $$(\text{fail } R_{i_m,j_m,k}) \rightarrow \text{(Noop)}$$

   The first production permits access
   $A_{i,j,k}$ to terminate if process $k$ has no
   outstanding requests. The second produc-
   tion permits the process to nondetermin-
   stically terminate the access.

   endfor

   Include all above productions which pro-
   cess accesses as one conflict set in $R$.

   endfor

(3) Let $\beta > (2 *$ the number of requests
+ the number of processes)

Since for each request there can be at most two
elements of the alphabet $\Sigma$, namely, $p_{i,j,k}$ and
either of $R_{i,j,k}$ and $A_{i,j,k}$ at any time and
there is an $I_k$ for each process, then $\beta$ must
be at least as specified above. Note that it is
assumed that each row of the priority matrix
 corresponding to a $p_{i,j,k}$ occupies one entry in
WM.

endbegin

Figure 11 continued
Example 3

We will construct a production system for example 2 using algorithm COMPS. The elements of $\Sigma$ are already defined. The minimal incompatible sets are

$\{A_1, A_3\}$ and $\{A_2, A_3\}$.

B-set ($R_1$) = B-set ($R_2$) = $\{\{A_3\}\}$.

B-set ($R_3$) = $\{\{A_1\}, \{A_2\}\}$.

C-set ($R_1$) = C-set ($R_2$) = $\{\{R_3\}\}$.

C-set ($R_3$) = $\{\{R_1\}, \{R_2\}\}$.

Step 2a in the algorithm produces the following productions to represent the occurrence of requests.

$$P_{1a}': ((I_1) (\text{fail } R_1) (\text{fail } A_1) \rightarrow (\text{Insert } R_1)(\text{Assign } p_1))$$

$$P_{1b}': ((I_1) (\text{fail } R_1) (\text{fail } A_1) \rightarrow (\text{Noop}))$$

Let $r_1 = \{P_{1a}', P_{1b}'\}$ be a conflict set.

$$P_{2a}': ((I_2) (\text{fail } R_2) (\text{fail } A_2) \rightarrow (\text{Insert } R_2)(\text{Assign } p_2))$$

$$P_{2b}': ((I_2) (\text{fail } R_2) (\text{fail } A_2) \rightarrow (\text{Noop}))$$

Let $r_2 = \{P_{2a}', P_{2b}'\}$ be a conflict set.

$$P_{3a}': ((I_3) (\text{fail } R_3) (\text{fail } A_3) \rightarrow (\text{Insert } R_3)(\text{Assign } p_3))$$

$$P_{7b}': ((\text{Noop})$$
Let \( r_3 = \{ P_{3a}, P_{3b} \} \) be a conflict set.

Applying step 2b, we get the following productions for satisfying requests.

\[
P_{4a} \equiv (R_1)(\text{fail } A_3)(\text{fail } (R_3 \land (p_3 > p_1))) \\
\rightarrow (\text{Replace}(R_1, A_1))(\text{Assign } p_1)
\]

\[
P_{4b} \equiv (R_1)(\text{fail } A_3)(\text{fail } (R_3 \land (p_3 > p_1))) \\
\rightarrow (\text{Noop})
\]

Let \( r_4 = \{ P_{4a}, P_{4b} \} \).

\[
P_{5a} \equiv (R_2)(\text{fail } A_3)(\text{fail } (R_3 \land (p_3 > p_2))) \\
\rightarrow (\text{Replace}(R_2, A_2))(\text{Assign } p_2)
\]

\[
P_{5b} \equiv (R_2)(\text{fail } A_3)(\text{fail } (R_3 \land (p_3 > p_2))) \\
\rightarrow (\text{Noop})
\]

Let \( r_5 = \{ P_{5a}, P_{5b} \} \).

\[
P_{6a} \equiv (R_3)(\text{fail } A_1)(\text{fail } A_2)(\text{fail } (R_1 \land (p_1 > p_3))) \\
\rightarrow (\text{Replace}(R_3, A_3))(\text{Assign } p_3)
\]

\[
P_{6b} \equiv (R_3)(\text{fail } A_1)(\text{fail } A_2)(\text{fail } (R_1 \land (p_1 > p_3))) \\
\rightarrow (\text{Noop})
\]

Let \( r_6 = \{ P_{6a}, P_{6b} \} \).

Finally, step 2c provides the following productions to allow accesses to terminate or continue in progress.

\[
P_{7a} \equiv (A_1) \rightarrow (\text{Delete } A_1)
\]

\[
P_{7b} \equiv (A_1) \rightarrow (\text{Noop})
\]
Let \( r_7 = \{P_{7a}, P_{7b}\} \).

\[ P_{8a}: \ (A_2) \rightarrow \ (\text{Delete } A_2) \]

\[ P_{8b}: \ (A_2) \rightarrow \ (\text{Noop}) \]

Let \( r_8 = \{P_{8a}, P_{8b}\} \).

\[ P_{9a}: \ (A_3) \rightarrow \ (\text{Delete } A_3) \]

\[ P_{9b}: \ (A_3) \rightarrow \ (\text{Noop}) \]

Let \( r_9 = \{P_{9a}, P_{9b}\} \).

Applying step 3, the size of \( WM = \beta = 2 \times 3 + 3 = 9 \).

The production system specified above is compatible if and only if for every contending set of requests one request has the lowest priority. This is proven in the following lemma.
Lemma

Algorithm COMPS constructs a compatible production system if and only if for every contending set of requests, one request has the lowest priority.

Proof:

We will first prove that the production system is compatible if for every contending set of requests, one request has the lowest priority. There are two ways that an incompatible set of accesses can occur. The first is when a configuration contains a set of accesses $S$ such that satisfying a request $R$ will result in a minimal incompatible set of accesses. In this case, $R$ will be blocked by a $B$-set condition. The second possibility is when a configuration contains a set of requests $Q$ such that satisfying requests in $Q$ will result in an incompatible set of accesses. Clearly $Q$ must contain a contending set of requests $Q'$. Then, by assumption, there is a request $R$ in $Q'$ that has lower priority than all other requests in $Q'$. On considering the $C$-set conditions of the productions in which requests are satisfied in algorithm COMPS, a request will not be satisfied if it has lower priority than all requests in an element of its $C$-set. Thus, request $R$ will be blocked and an incompatible set of accesses will not occur.
We now prove that if algorithm COMPS constructs a compatible production system then for every contending set of requests, one request has the lowest priority. Let more than one request in a set of contending requests \( Q \) have the lowest priority. Then, the productions in which requests are satisfied will allow the requests with the lowest priority to be satisfied since for each of those requests \( R \), there will be at least one other request \( R' \) whose priority is not greater than \( R \). As a consequence an incompatible set of accesses will occur and the production system is incompatible, a contradiction.

It should be clear from the above proof that when a configuration only contains a minimal incompatible set of requests, algorithm COMPS allows the maximum number of requests to be satisfied without destroying the compatibility of the production system. However, the algorithm may result in a transition \( \mathcal{S} \xrightarrow{\cdot} \mathcal{S}' \) in which the maximum number of requests are satisfied but \( \mathcal{S}' \) still contains requests and there is no transition possible from \( \mathcal{S}' \). Thus the requests in \( \mathcal{S}' \) will be indefinitely blocked. We address this problem in the following section.
4. Specification for Live Systems

4.1 Definitions:

If WM contains a configuration \( \mathcal{C} \) which contains a request or access \( x \) such that no production whose actions change \( x \) is enabled with \( \mathcal{C} \) in WM, we then say that \( x \) is blocked in \( \mathcal{C} \) by the unsatisfied conditions of the productions whose actions change \( x \). If there is an unsatisfied condition of the form \( \text{fail}(x_1 \land \cdots \land x_n \land c) \) where \( x_i, \ 1 \leq i \leq n \) are requests and/or accesses and \( c \) is a predicate, we then say that \( x \) is blocked by each \( x_i, \ 1 \leq i \leq n \). Note that a request can be blocked by elements of its B-set or its C-set.

A request or access \( x \) in a reachable configuration is indefinitely blocked in \( \mathcal{C} \) if \( x \) is blocked in every configuration \( \mathcal{C}' \) such that \( \mathcal{C} \xrightarrow{*} \mathcal{C}' \).

Corollary:

A request or access \( x \) is indefinitely blocked in a configuration \( \mathcal{C} \) if every configuration \( \mathcal{C}' \) such that \( \mathcal{C} \xrightarrow{*} \mathcal{C}' \) contains \( x \).
A configuration $\mathcal{X}$ is said to be a deadlock configuration (or not live) if it contains a set $X = \{x_0, \ldots, x_{n-1}\}$ of accesses and requests such that $x_{(i+1) \mod n}$ is indefinitely blocked by a set of requests and accesses which contains $x_{i \mod n}$. The set of accesses and requests $X$ is said to be deadlocked, i.e., $x_1$ is blocked by $x_0$, $x_2$ is blocked by $x_1$, ..., and $x_0$ is blocked by $x_{n-1}$.

A PS itself is not live if at least one deadlock configuration is reachable from the initial configuration, otherwise it is live. A PS is inherently live if none of its configurations is a deadlock configuration, whether or not it is reachable.
4.2 An Algorithm for Determining Liveness

Algorithm DLCK, shown in Figure 12, determines if a PS which specifies a synchronization policy is live. The basic idea is to determine if any reachable configuration contains a deadlock set of requests and accesses by first determining the set of requests and accesses $S$ which appear in a particular configuration $\mathcal{C}$ and also appear in all reachable configurations from $\mathcal{C}$; and second determining if $S$ contains a set of requests and accesses in which the requests and accesses block each other. The second step is performed by algorithm CYCSET, shown in Figure 13, which is invoked in DLCK.
procedure DLCK (PS: a production system)

/* Algorithm DLCK determines if PS is live.
/* Input: PS.
/* Output: Whether PS is live or not.

begin

for every reachable configuration $\mathcal{C}_{i_0}$

Let $\mathcal{Y} = \{\mathcal{C}_{i_0}, \mathcal{C}_{i_1}, \ldots, \mathcal{C}_{i_n}\}$ be the set of reachable configurations from $\mathcal{C}_{i_0}$.

Let $x_{i,j}$ be the set of accesses and requests in $\mathcal{C}_{i,j}$, $0 \leq j \leq n$ such that each request or access in $x_{i,j}$ is blocked by at least one particular condition in every $\mathcal{C}_{i,j}$, $0 \leq i \leq n$.

Let $X = \bigcap_{0 \leq j \leq n} x_{i,j}$

if $X \not\subset 2$

then

if CYCSET($X$, $\n$, PS)

then $\mathcal{C}_{i_0}$ is a deadlock configuration

and PS is not live, stop.

endif
endif
endfor

PS is live.
endbegin

Figure 12
function CYCSET(X: a set of requests and accesses, 
  Β: a set of configurations, PS: a production system)

/* Input: X, Β and PS. */
/* Output: CYCSET is set to "true" if X contains */
/* requests and accesses which block each */
/* other, otherwise CYCSET is set to "false". */

begin
  FLAG = true
  while (FLAG) do
    FLAG = false
    for every x_k in X
      if no x ≠ x_k in X blocks x_k in Β
        then
          Delete x_k from X
          FLAG = true
      endif
  endfor
  endwhile
  CYCSET = (|X| > 2) /* true or false */
endbegin

Figure 13
Example 4  The Dining Philosophers Problem.

There are N philosophers who spend their lives either eating or thinking. Each philosopher has his own place at a circular table, in the center of which is a large bowl of spaghetti. To eat spaghetti requires two forks, but only N forks are provided, one between each pair of philosophers. The only forks a philosopher can pick up are those on his immediate right and left. Each philosopher is identical in structure, alternatively eating then thinking. The problem is to simulate the behavior of the philosophers while avoiding deadlock, [DIJK 68], [HOLT 78].

Figure 14 shows a picture of a table setting with five plates and forks.
Let $R_{i,j}$ be a request made to access fork $i$ by process $j$. Similarly, let $A_{i,j}$ be an access made to fork $i$ by process $j$. The $i$th and $(i+1)$st philosophers share the $i$th fork. In the following notation addition on the subscripts is performed mod $N$. The minimal incompatible sets of accesses are $\{A_{i,i}, A_{i,i+1}\}, 0 \leq i \leq N-1$. Thus,

- $B$-set $\ (R_{i,i}) = \{\{A_{i,i+1}\}\}, 0 \leq i \leq N-1$, 
- $C$-set $\ (R_{i,i}) = \{\{R_{i,i+1}\}\}, 0 \leq i \leq N-1$, 
- $B$-set $\ (R_{i,i+1}) = \{\{A_{i,i}\}\}, 0 \leq i \leq N-1$, and 
- $C$-set $\ (R_{i,i+1}) = \{\{R_{i,i}\}\}, 0 \leq i \leq N-1$.

We will now construct a production system $PS$ for this problem using algorithm COMPS. We will show that $PS$ is not live. Let the initial contents of $WM$ be $\emptyset_0 = I_0 I_1 \ldots I_{N-1}$.

$\rho$ contains the following productions to allow requests to occur.

- $P_{ia}: \ (I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i})$ 
  $\quad \rightarrow \ (\text{Insert } R_{i,i})(\text{Assign } p_{i,i})$

- $P_{ib}: \ (I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i})$ 
  $\quad \rightarrow \ (\text{Noop})$

- $P_{ic}: \ (I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i-1,i})$ 
  $\quad \rightarrow \ (\text{Insert } R_{i-1,i})(\text{Assign } p_{i-1,i})$
Note that a philosopher must get the first fork before requesting the second one. This may lead to deadlock as will be shortly shown. The above productions are defined for $0 \leq i \leq N-1$, i.e., there are a total of $4N$ productions. $R$ contains the conflict sets $\{P_{ia}, P_{ib}, P_{ic}, P_{id}\}$, $0 \leq i \leq N-1$.

The following productions grant requests $R_{i,i}$:

$0 \leq i \leq N-1$.

$$P_{(N+i)a'}: (R_{i,i})(\text{fail } A_{i,i+1})$$
$$(\text{fail}(R_{i,i+1} \land (p_{i,i+1} > p_{i,i})))$$
$$\rightarrow (\text{Replace}(R_{i,i}, A_{i,i}))(\text{Assign } p_{i,i})$$

$$P_{(N+i)b'}: (R_{i,i})(\text{fail } A_{i,i+1})$$
$$(\text{fail}(R_{i,i+1} \land (p_{i,i+1} > p_{i,i})))$$
$$\rightarrow (\text{Noop})$$

In the above productions a request $R_{i,i}$ is satisfied if $A_{i,i+1}$ is not in WM, i.e., if the $(i+1)$st philosopher is not already using fork $i$ and the $(i+1)$st philosopher does not have a request $R_{i,i+1}$ for fork $i$ with higher priority $p_{i,i+1}$ than $p_{i,i}$. If a request is satisfied, the action Assign may change the priority $p_{i,i}$ which is now associated
with $A_{i,i}$.

The above productions are also defined for $0 \leq i \leq N-1$. $R$ contains the conflict sets $\{P_{(N+i)a}, P_{(N+i)b}\}$, $0 \leq i \leq N-1$.

The following productions grant requests $R_{i-1,i}$, $0 \leq i \leq N-1$.

\[ P_{(2N+i)a} : ((R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \quad (\text{fail}(R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i}))) \]
\[ \quad \rightarrow (\text{Replace}(R_{i-1,i}, A_{i-1,i})) \]
\[ \quad (\text{Assign } p_{i-1,i})) \]

\[ P_{(2N+i)b} : ((R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \quad (\text{fail}(R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i}))) \]
\[ \quad \rightarrow (\text{Noop})) \]

The above productions are also defined for $0 \leq i \leq N-1$. $R$ contains the conflict sets $\{P_{(2N+i)a}, P_{(2N+i)b}\}$, $0 \leq i \leq N-1$. The interpretation of those productions is analogous to that of the preceding productions.

The following productions represent a philosopher putting down a fork. A philosopher can put down a fork if he does not have a pending request for another fork.
In the above productions, \(0 \leq i \leq N-1\). \(R\) contains the conflict sets \(\{P(3N+i)a', P(3N+i)b', P(3N+i)c', P(3N+i)d\}\), \(0 \leq i \leq N-1\).

We now proceed to apply algorithm DLCK to this production system. Consider the initial configuration \(\gamma_0 = I_0 I_1 \cdots I_{N-1}\). Since \(\gamma_0\) does not contain any requests or accesses, the intersection of the sets of requests and accesses in the configurations reachable from \(\gamma_0\) will be empty. As a consequence, the set \(X\) in the algorithm will be empty. Thus, \(\gamma_0\) is not a deadlock configuration.

Consider the configuration

\[
\gamma = A_{0,0} R_{N-1,0} I_0 A_{1,1} R_{0,1} I_1 \cdots A_{N-1,N-1} R_{N-2,N-1} I_{N-1}.
\]

\(\gamma\) is reachable from \(\gamma_0\) through the following transitions:

\[
\begin{align*}
\gamma_0 & \xrightarrow{P_0 a} P_1 a \cdots P(N-1)a \xrightarrow{R_0,0} R_1,1 I_1 \cdots R_{N-1,N-1} I_{N-1} \\
& \xrightarrow{P_0 a} P_1 a \cdots P(N-1)a \xrightarrow{R_0,0} R_1,1 I_1 \cdots R_{N-1,N-1} I_{N-1} \\
& \xrightarrow{P_0 a} P_1 a \cdots P(N-1)a \xrightarrow{A_{0,0} R_{N-1,0} I_0} \cdots A_{N-1,N-1} R_{N-2,N-1} I_{N-1}.
\end{align*}
\]

The only reachable configuration from \(\gamma\) is \(\gamma\) itself. The set of requests and accesses \(X\) specified in algorithm DLCK
Applying function CYCSET, we find that deletion of each $A_{i,i}$ is prevented by $R_{i-1,i}$ as can be deduced from the productions $P_{(3N+i)a}$ for $0 \leq i \leq N-1$. None of the philosophers can put down their forks. Each $R_{i-1,i}$ is blocked by $A_{i-1,i-1}$ as can be deduced from the productions $P_{(2N+i)a}$, $0 \leq i \leq N-1$. That is, the $i$th philosopher can not access fork $(i-1)$ because it is being used by the $(i-1)$st philosopher. As a consequence, no element of $X$ can be deleted in CYCSET.

Thus, for $N > 1$, $|X| > 2$, $\emptyset$ is a deadlock configuration, and PS is not live.

In the solution presented above for the dining philosophers problem, all properties of the policy specified by the production system, e.g., compatibility, liveness and fairness will relate to accesses and requests to the forks. Compatibility and liveness of the policy can be fully represented by compatibility and liveness in terms accesses and requests to the forks. However, for fairness the issue is a little different. Consider the situation in which a philosopher indefinitely alternates picking up the two forks on his right and left but is not able to get both forks at the same time. Evidently, since he never has both forks at any one time he can never eat the spaghetti and can
be "starved". This type of starvation cannot be represented as long as accesses to the forks are considered independent of each other. In other words, the fact that a philosopher can get access to each fork alone guarantees "fairness" in terms of access to the forks but not in terms of access to the spaghetti.

Consider another solution to the dining philosophers problem in which each philosopher requests and puts down both forks simultaneously. That is, both forks can be considered as one resource. In this case, access to the forks is "equivalent" to eating the spaghetti, and our analysis and design algorithm will fully represent this solution. As can be expected, this is possible because the problem in this case can be fully stated in terms of minimal incompatible sets.
4.3 Designing Live Production Systems

The approaches to the design of a live PS can be classified into three categories. In the first approach, the system is to be designed such that it is live without restricting the occurrence of requests and termination of accesses. In the second approach, the occurrence of requests and termination of accesses are only permitted to occur in a prespecified order such that deadlock can not occur. In the third approach, accesses are forced to terminate in order to satisfy other requests, thus preventing deadlock.

4.3.1 The First Approach

Lemma

A compatible production system constructed by algorithm COMPS is live if and only if there is no set X = \{x_0, \ldots, x_{n-1}\} of requests and/or accesses such that:

1. Not all elements of X belong to the same minimal incompatible set, and
2. Every \(x_i \mod n\) appears in a "fail" condition of all productions whose actions insert \(x_{(i+1) \mod n}\) in working memory.
The importance of this lemma is that a production system produced by COMPS may be analyzed for liveness by only looking at the form of the productions produced. The analysis procedure does not need to examine all reachable configurations of the system.

Proof:

We first prove the "if clause". A deadlock set \( X \) cannot consist of elements that belong to one minimal incompatible set. Since the production system is compatible, then by the lemma of section 3.3, for every set of contending requests one request \( R \) has the lowest priority. Thus, \( R \) can not block any other request of \( X \), and \( X \) can not be a deadlock set.

If elements of \( X \) do not all belong to the same minimal incompatible set, then the condition in the lemma guarantees that deadlock does not occur in this case.

The "only if" clause follows directly from the definitions of a live production system and a deadlock configuration.
4.3.2 The Second Approach

In this approach, requests and the corresponding accesses are sequentially ordered into different levels such that no minimal incompatible set spans two or more levels. As a consequence, accesses at any one level can not block requests at any different level. Requests by a process from a given level must not only be made simultaneously, but also granted simultaneously. Deadlock can not occur as a result of each process having a pending request that is blocked by an access of another process at the same level. When a process has been granted accesses at a level $L_j$, it can only make requests at a higher level $L_k$. Accesses can only be blocked by requests at a higher level. This approach is a generalization of the hierarchical resource allocation method \[HAVE\,68\], \[HANS\,73\].

We now show how to partition the requests such that no minimal incompatible set spans two or more levels.

A partition $\Pi$ on a set $S$ is a collection of disjoint subsets whose set union is $S$. 
Let $Q_0, Q_1, \ldots, Q_{n-1}$ be minimal incompatible sets of requests. Let $Q'_0, \ldots, Q'_{m-1}, 1 \leq m \leq n$, be the sets of requests such that:

1. $Q'_i \cap Q'_j = \emptyset$, the null set, and
2. For every $Q_i$, $0 \leq i \leq n-1$, $Q_i \subseteq Q'_j$ for some $0 \leq j \leq m-1$.
3. Every element of $Q'_i$, $0 \leq i \leq m-1$ is also an element of some $Q_j$, $0 \leq j \leq n-1$.

Clearly, \{Q'_0, \ldots, Q'_{m-1}\} forms a partition on $Q_0 \cup \ldots \cup Q_{n-1}$. Let $\Pi = \{Q'_0, \ldots, Q'_{m-1}\}$. Let $Q'_0$ be at the highest level, $Q'_1$ at the next lower level, \ldots, and $Q'_{m-1}$ at the lowest level. Note that accesses at different levels are compatible, since all the elements of any minimal incompatible set are at the same level. Requests that do not belong to any minimal incompatible sets can be arbitrarily placed at any level.

Let the requests be represented as nodes of a graph in which the edges connect the requests of minimal incompatible sets. For instance, a minimal incompatible set $X = \{x_0, \ldots, x_{n-1}\}$ can be represented by a subgraph whose nodes are $x_0, \ldots, x_{n-1}$, and whose edges are $(x_0, x_1), \ldots, (x_{n-2}, x_{n-1})$. Then, partitioning the incompatible sets as prescribed above is equivalent to finding the components of a graph [DEO 74], i.e., each component represents an
element of the partition. The complexity of finding the components of a graph is proportional to the number of edges in the graph or the square of the number of nodes [DEO 74].

Having obtained the sets of requests for each level, we have the choice of placing more than one set at the same level. At one extreme, all the sets are placed in one level, and at the other, each set \( Q'_i \) derived in the procedure is placed at a level by itself. The first choice corresponds to the complete allocation of all resources needed by a process.

Algorithm LVCMPS, shown in Figure 15, constructs a live production system using this approach. While algorithm COMPS presented earlier allows a process to make requests in any order, algorithm LVCMPS does not allow a process to make a request at a level \( l \) if the process has outstanding accesses at levels higher than level \( l \). That is, a process must make requests at lower levels before those at higher levels if the process is to exercise simultaneous accesses at different levels. In algorithm LVCMPS, a set of requests by a process at any one level are simultaneously granted, while in COMPS each request is separately granted. In LVCMPS, a process can terminate accesses at level \( l \) if it has no pending request at a higher level than level \( l \).
In COMPS, a process can terminate an access if it has no pending request. In LVCMPS, accesses by a process at a level \( l \) are simultaneously terminated, while accesses by a process in COMPS are terminated one at a time.

In general, LVCMPS restricts a process to simultaneously make those requests at the same level, and to simultaneously terminate the corresponding accesses. The requests of a process at a level \( l \) are also granted simultaneously. COMPS places no such restrictions on the occurrence and granting of requests, and termination of accesses.
procedure LVCMPs(S: minimal incompatible sets of accesses, PS: a production system)

/* Algorithm LVCMPs constructs a live production system by the hierarchical ordering of requests method. */
/* Input: S. */
/* Output: PS. */

begin

(1) Include the symbols $I_k, R_{i,j,k}$ and $A_{i,j,k}$ and variables $p_{i,j,k}$ for all relevant values of $i,j,k$ in the working memory alphabet $\Sigma$. Let the initial configuration be $\sigma_0 = I_{0..n-1}$, $n =$ number of processes.

(2) Partition the requests (and accesses) into different levels such that no minimal incompatible set spans two or more levels.

(3) for every process $k$

a. for every level $\ell$ of the partition constructed in step (2).

Let $R_{i_1,j_1,k}$, ..., $R_{i_m,j_m,k}$ be the requests by process $k$ at level $\ell$.

Let $R_{i_1,j_1,k}$, ..., $R_{i_m,j_m,k}$, ..., $R_{s,j,s,k}$ be all the requests of process $k$.

Let the accesses by process $k$ at higher levels than level $\ell$ be $A_{i_1',j_1',k}$, ..., $A_{i_m',j_m',k}$.

Include the following productions in $\mathcal{P}$, the set of productions, to allow for the occurrence of requests by process $k$ at level $\ell$. In these productions process $k$ can make its requests at level $\ell$ if it has no pending requests, it is not already exercising the accesses corresponding to the requests being made and it is

Figure 15
not exercising accesses at levels higher than level \( \ell \). Requests by a process at a level \( \ell \) are simultaneously made (and simultaneously satisfied as will be shortly seen) to avoid deadlock within level \( \ell \).

\[
(I_k(\text{fail } R_{i_1,j_1,k}, \ldots, \text{fail } R_{i_s,j_s,k})
\text{fail } A_{i_1,j_1,k}, \ldots, \text{fail } A_{i_m,j_m,k})
\text{fail } A'_{i_1,j_1,k}, \ldots, \text{fail } A'_{i_m,j_m,k})
\rightarrow (\text{Insert } R_{i_1,j_1,k}, \ldots, R_{i_m,j_m,k})
\text{Assign } p_{i_1,j_1,k}, \ldots, \text{Assign } p_{i_m,j_m,k})
\]

In the above productions the actions Assign is to be supplied by the designer in accordance with the conditions placed by the lemma which will follow shortly.

endfor

Include all above productions as one conflict set in \( R \), the set of conflict sets.

b. for every level \( \ell \)

Let \( Q = \{ R_{i_1,j_1,k}, \ldots, R_{i_m,j_m,k} \} \) be the requests of process \( k \) at level \( \ell \).

Determine the B-sets and C-sets for \( Q \) using algorithms BSETS and CSETS.

Let the B-set conditions for requests in \( Q \) as in algorithm COMPS be \( \{ c_{i1}, c_{i2}, \ldots, c_{iM} \} \), \ldots, and

\( \{ c_{m1}, c_{m2}, \ldots, c_{MM} \} \) respectively.

Figure 15 continued
Similarly, let the C-set conditions for requests in Q be \( \{c_{11}, c_{12}, \ldots, c_{1M_1} \}, \ldots, \) and \( \{c_{m1}, c_{m2}, \ldots, c_{mM_m} \} \).

Include the following productions in \( \mathcal{P} \).

\[
((R_{i1}, j_1, k) \ldots (R_{i_m}, j_m, k) \\
\begin{array}{cccc}
c_{11} & c_{12} & \cdots & c_{1M_1} \\
c_{21} & c_{22} & \cdots & c_{2M_2} \\
\vdots & & & \\
c_{m1} & c_{m2} & \cdots & c_{mM_m} \\
c_{i1}' & c_{i2}' & \cdots & c_{iM_1}' \\
c_{21}' & c_{22}' & \cdots & c_{2M_2}' \\
\vdots & & & \\
c_{m1}' & c_{m2}' & \cdots & c_{mM_m}'
\end{array}
\]

\[\rightarrow (\text{Replace}(R_{i1}, j_1, k, A_{i1}, j_1, k)) \ldots \]

\[\begin{array}{c}
(\text{Replace}(R_{i_m}, j_m, k, A_{i_m}, j_m, k)) \\
(\text{Assign} \ p_{i1}, j_1, k) \ldots (\text{Assign} \ p_{i_m}, j_m, k)
\end{array}\]

Figure 15 continued
Include the above two productions as one conflict set in $\mathcal{R}$.

In the above productions, the requests by process $k$ at level $\ell$ are satisfied simultaneously. The requests are satisfied if none of their B-set or C-set conditions is true when applied to working memory. As mentioned earlier, the requests are satisfied simultaneously to avoid deadlock within any one level. The action Assign, to be supplied by the designer, associates a priority $p_{i,j,k}$ with the corresponding access $A_{i,j,k}$.

endfor

c.for every level $\ell$

Let $A_{i_1,j_1,k}, \ldots, A_{i_m,j_m,k}$ be the accesses of process $k$ at level $\ell$.

Let the requests by process $k$ at levels higher than $\ell$ be $A'_{i_1,j_1,k}, \ldots, A'_{i_m,j_m,k}$.

The set of accesses by process $k$ at level $\ell$ can be terminated if process $k$ has no pending requests at levels higher than $\ell$.

Figure 15 continued
There can be no request at levels lower than \( l \) while process \( k \) is exercising accesses at level \( l \).

\[
((A_{i_1,j_1,k})\ldots(A_{i_m,j_m,k})
\text{ (fail } R_{i_1,j_1,k})\ldots(\text{fail } R_{i_m,j_m,k})
\longrightarrow (\text{Delete } A_{i_1,j_1,k})\ldots
(\text{Delete } A_{i_m,j_m,k}))
\]

\[
((A_{i_1,j_1,k})\ldots(A_{i_m,j_m,k})
\text{ (fail } R_{i_1,j_1,k})\ldots(\text{fail } R_{i_m,j_m,k})
\longrightarrow (\text{Noop}))
\]

\text{endfor}

Include all above productions as one conflict set in \( \mathcal{R} \).

\text{endfor}

(4) Let \( \beta \), the size of working memory, =
\( (2 \times \text{ the number of requests} \times \text{ the number of processes}) \), i.e., for each request there can only be two elements \( p_{i,j,k} \) and either \( R_{i,j,k} \) or \( A_{i,j,k} \) at any one time, and \( I_k \) for each process \( k \).

\text{endbegin}

Figure 15 continued
Example 5

Consider the dining philosophers problem presented earlier. We will use algorithm LVCMPS to construct a live production system for this problem. Partitioning the requests such that no minimal incompatible set spans two or more levels, we have the partition:

\{ \{R_{i,i}, R_{i,i+1}\} \mid 0 \leq i \leq N-1 \}. Figure 16 shows the different levels and the corresponding requests. Note that the minimal incompatible sets are disjoint, and we choose to place each one at a level by itself. The initial configuration is:

I_0 \ I_1 \ \cdots \ \ I_{N-1}.

\begin{array}{c|cc}
\text{Level 0} & R_{0,0} & \cdots & R_{0,1} \\
\text{Level 1} & R_{1,1} & \cdots & R_{1,2} \\
\vdots & \vdots & \ddots & \vdots \\
\text{Level N-2} & R_{N-2,N-2} & \cdots & R_{N-2,N-1} \\
\text{Level N-1} & R_{N-1,N-1} & \cdots & R_{N-1,0} \\
\end{array}

Figure 16
Following step 3a of LVCMPS, we construct the productions representing a philosopher making a request. For the zeroth philosopher, \( R_{0,0} \) is at a higher level than \( R_{N-1,0} \). Thus, he must request the \((N-1)st\) before the zeroth fork. The philosopher can make the request \( R_{0,0} \) if he has no pending requests and if he has not already picked the zeroth fork. He can make the request \( R_{N-1,0} \) if he has no pending requests, if he has not already picked up the \((N-1)st\) fork or the zeroth fork.

\[
P_{0a}': ((I_0)(\text{fail } R_{0,0})(\text{fail } R_{N-1,0})(\text{fail } A_{0,0})
\rightarrow (\text{Insert } R_{0,0})(\text{Assign } p_{0,0}))
\]

\[
P_{0b}': ((I_0)(\text{fail } R_{0,0})(\text{fail } R_{N-1,0})(\text{fail } A_{0,0})
\rightarrow (\text{Noop}))
\]

\[
P_{0c}': ((I_0)(\text{fail } R_{0,0})(\text{fail } R_{N-1,0})
(\text{fail } A_{0,0})(\text{fail } A_{N-1,0})
\rightarrow (\text{Insert } R_{N-1,0})(\text{Assign } p_{N-1,0}))
\]

\[
P_{0d}': ((I_0)(\text{fail } R_{0,0})(\text{fail } R_{N-1,0})
(\text{fail } A_{0,0})(\text{fail } A_{N-1,0})
\rightarrow (\text{Noop}))
\]

For the ith philosopher, \( 1 \leq i \leq N-1 \), \( R_{i-1,i} \) is at a higher level than \( R_{i,i} \). Thus, he can request the ith fork if he has no pending requests, and if he has not already
picked up the ith or (i-1)st fork. The philosopher can request the (i-1)st fork if he has no pending requests and if he has not already picked it up. The ith philosopher must request the ith fork before the (i-1)st fork.

\[ P_{ia}: \quad ((I_i)(\text{fail } R_{i-1,i})(\text{fail } R_{i,i})(\text{fail } A_{i,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Insert } R_{i,i})(\text{Assign } p_i,i)) \]

\[ P_{ib}: \quad ((I_i)(\text{fail } R_{i-1,i})(\text{fail } R_{i,i})(\text{fail } A_{i,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Noop})) \]

\[ P_{ic}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Insert } R_{i-1,i})(\text{Assign } p_{i-1},i)) \]

\[ P_{id}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Noop})) \]

The action Assign in the above productions, as in all productions to follow, assigns a value to the priority of the request or access being made. It is to be supplied by the user in accordance with the restrictions placed by the lemma which follows this example. Include \( \{P_{ia}, P_{ib}, P_{ic}, P_{id}\}, 0 \leq i \leq N-1 \) in \( \mathcal{R} \).
Following step 3b of LVCMPS, we get the productions to satisfy requests. Since for each philosopher there is at most one request per level, the B-set and C-set conditions for a particular production will be those of a single request. The B-set and C-set conditions were developed in example 4, and the productions for satisfying requests will be identical to those of example 4. The productions are as follows:

\[ P(N+i)a: \quad ((R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \quad (\text{fail } (R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i}))) \]
\[ \quad \rightarrow (\text{Replace}(R_{i-1,i}, A_{i-1,i})) \]
\[ \quad (\text{Assign } p_{i-1,i}) \]

\[ P(N+i)b: \quad ((R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \quad (\text{fail } (R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i}))) \]
\[ \quad \rightarrow (\text{Noop}) \]

In the above productions, the ith philosopher can pick up the (i-1)st fork if the (i-1)st philosopher has not already picked it up and if the (i-1)st philosopher is not requesting the same fork with higher priority. Include \( \{P(N+i)a, P(N+i)b\} \), \( 0 \leq i \leq N-1 \) in \( \mathcal{R} \).

\[ P(2N+i)a: \quad ((R_i,i)(\text{fail } A_{i,i+1}) \]
\[ \quad (\text{fail } (R_{i,i+1} \land (p_{i,i+1} > p_{i,i}))) \]
\[ \quad \rightarrow (\text{Replace}(R_i,i, A_{i,i}))(\text{Assign } p_{i,i}) \]
\[ P(2N+i)b : (R_i, i)(\text{fail } A_{i, i+1}) \]
\[ \quad (\text{fail } (R_i, i+1 \land (p_{i, i+1} > p_{i, i}))) \]
\[ \rightarrow (\text{Noop}) \]

In the above productions the \( i \)th philosopher can pick up the \( i \)th fork if the \((i+1)\)st philosopher has not already picked it up and if the \((i+1)\)st philosopher is not requesting the same fork with higher priority. Include \( \{P(2N+i)a', P(2N+i)b\} \), \( 0 \leq i \leq N-1 \), in \( R \).

Following step 3c in LVCMPS, a philosopher can terminate an access if he has no pending requests at higher levels. There can be no pending requests at a lower level than an access that is to be terminated. This is a result of the restriction that requests at lower levels are to be made before those at higher levels for a particular philosopher. The productions for the zeroth philosopher are:

\[ P_{3Na} : ((A_0, 0) \rightarrow (\text{Delete } A_{0, 0})) \]

\[ P_{3Nb} : ((A_0, 0) \rightarrow (\text{Noop})) \]

\[ P_{3Nc} : ((A_{N-1, 0})(\text{fail } R_{0, 0}) \rightarrow (\text{Delete } A_{N-1, 0})) \]

\[ P_{3Nd} : ((A_{N-1, 0})(\text{fail } R_{0, 0}) \rightarrow (\text{Noop})) \]
The productions for the $i$th philosopher, $1 \leq i \leq N$, are:

- $P(3N+i)a$: \(((A_i,i) (\text{fail } R_{i-1,i}) \rightarrow (\text{Delete } A_{i,i}))

- $P(3N+i)b$: \(((A_i,i) (\text{fail } R_{i-1,i}) \rightarrow (\text{Noop}))

- $P(3N+i)c$: \(((A_{i-1,i}) \rightarrow (\text{Delete } A_{i-1,i}))

- $P(3N+i)d$: \(((A_{i-1,i}) \rightarrow (\text{Delete } A_{i-1,i}))

Include \(\{P(3N+i)a', P(3N+i)b', P(3N+i)c', P(3N+i)d'\}, 0 \leq i \leq N-1\) in $\mathcal{R}$.

Thus, the PS above specifies a live solution to the dining philosophers problem. Note that it is assumed that the priorities of requests at any one level are non-cyclic as the following lemma states. It is also assumed that requests of a particular process $k$ are only made in the order implied by the productions which allow for the occurrence of requests. That is, requests at lower levels are made before requests at higher levels if the corresponding accesses are to be exercised simultaneously.
Lemma

If requests occur as prescribed in algorithm LVCMPS and if the priorities of requests at any one level are non-cyclic, then algorithm LVCMPS constructs a live compatible production system PS.

Proof:

We will first prove that PS is compatible. Since requests at different levels are compatible, we only have to consider the requests at any one level. Since the priorities of requests at any one level are non-cyclic, then for each contending set of requests Q, there is only one request with the lowest priority. If this were not true, then there is more than one request with the lowest priority, and either of two cases is possible. The first is that more than one request have the same priority in which case their priorities are clearly cyclic. The second possibility is that for every request R in Q there is another request R' in Q such that R' has lower priority than R'. That is, the priorities are cyclic. Thus, for every contending set of requests there is only one request with the lowest priority. From the productions developed in step 3b of the algorithm, no request R is satisfied if working memory contains accesses that form elements of B-sets(R) and if working memory
contains contending requests with higher priority than $R$. Thus, the same proof for the compatibility of the production system produced by algorithm COMPS applies here.

We now prove that PS is live. Assume PS is not live. Then, there is a set $X = \{x_0, \ldots, x_{n-1}\}$ of requests and accesses that can be deadlocked.

Consider the case where the $x_i$'s are all requests. Clearly, the requests can not be at different levels. If they did, a minimal incompatible set would span at least two levels, thus, contradicting the basis on which the requests are partitioned, i.e., no minimal incompatible set spans two or more levels. If the requests are all at one level, then the condition that the priorities at any one level are non-cyclic guarantees that at least one request (one with highest priority) can be satisfied and $X$ can not be deadlocked.

Now consider the case when $X$ contains requests and accesses. The accesses and requests can not be at the same level. If they were, then the accesses can not be prevented from terminating because requests by a particular process that are at the same level are satisfied simultaneously. Accesses can not be prevented from terminating by other accesses. Thus $X$ can not be deadlocked.
If the accesses and requests are at different levels, then consider the accesses and requests that are at the highest level. The accesses can not be prevented from terminating since they can only be prevented by requests at higher levels; and there are none because they are at the highest level. If there are only requests at the highest level, then the condition that the priorities of requests at the same level are non-cyclic guarantees that at least one request can be satisfied. Thus X can not be deadlocked, and the proof is complete.
4.3.3 The Third Approach

In this approach, deadlock sets are allowed to occur, however, deadlock is prevented by allowing one of the requests or accesses of the deadlock set to be satisfied or be terminated respectively. This in turn may require the deliberate termination of an access in order to guarantee compatibility. In this case, the access terminated is replaced by the corresponding request.

Whenever a deadlock set containing accesses occurs, it is only necessary to terminate at most one access. To accomplish this goal, the requests are partitioned into compatible sets such that:

(1) Each compatible set of the partition is assigned a level. Requests at level $i$ have higher priority than requests at levels $j$, $j > i$, and have lower priority than requests at levels $k$, $0 \leq k \leq i-1$. The priorities of the levels are implicit. We also say that level $(i-1)$ is higher than level $i$.

(2) Of each minimal incompatible set $Q$, the lowest level of any request $R$ in $Q$ contains only that element of $Q$. Thus, whenever preemption is required, the access at the lowest level, i.e., only the access with lowest priority is terminated.
(3) For each process, the requests by the process may be at one level only if they are compatible, otherwise, they must be at consecutive levels.

The third condition above is required so that a situation as shown in Figure 17 does not occur. \( A_{2,2} \) and \( R_{1,2} \) are an access to resource 2 and a request for resource 1, respectively, by process 2. \( A_{1,1} \) and \( R_{2,1} \) are an access to resource 1 and a request for resource 2, respectively, by process 1. \( A_{1,1} \) is prevented from termination by \( R_{2,1} \), so is \( A_{2,2} \) by \( R_{1,2} \). \( R_{2,1} \) is blocked by \( A_{2,2} \), so is \( R_{1,2} \) by \( A_{1,1} \). The requests and accesses form a deadlock set. Neither of the two accesses is at a lower level than the respective requests they block, thus they can not be preempted. We thus place the third condition to avoid such a situation. If the requests are placed as prescribed by condition (3), either of the accesses would have been lower than the request it is blocking, and preemption could be possible.

As a consequence of partitioning the requests as prescribed above, in order to satisfy a request \( R \) at level \( \ell \), only those elements of the \( C\text{-set}(R) \) in which all requests are at level \( \ell \) or higher need be checked. However, all elements of \( B\text{-set}(R) \) must be checked.
Figure 17
The production system we construct for modelling a synchronization policy as specified above will only reflect the effects of removing accesses on the state of the resources with respect to the requests and accesses made to the resources. The effects of preemption on the behavior of processes, e.g., roll back and restart, are not modelled here since we are only using production systems as a specification of synchronization policies.

Algorithm COM, shown in Figure 18, constructs a partition of a set of requests as specified above. A request \( R \) is placed at level \( L \) if no minimal incompatible set consists of \( R \), at least one other request at level \( L \) and requests at levels \( l < L \). Otherwise, \( R \) is placed at the next level. As a consequence, for each minimal incompatible set, there is only one request at a lowest level, i.e., with a lowest priority.

Algorithm PLVCMPS, shown in Figure 19, constructs a compatible live production system. The requests are partitioned into compatible sets by algorithm COM. No explicit priorities are associated with the requests or accesses because the priorities are implicitly set by ordering the requests into different levels. Here, the levels contain compatible requests whereas the levels in algorithm LVCMPS contain incompatible requests. The size of working memory is
also reduced since priorities do not have to be represented within working memory.

As in algorithm COMPS, a process can make a request if it has no pending requests and if it is not already exercising the corresponding access. A request $R$ is satisfied if working memory does not contain any elements of $C\text{-set}(R)$ or elements of $C\text{-set}(R)$ in which requests have priorities equal or higher than $R$. While requests of a process at any level in algorithm LVCMPS are satisfied simultaneously requests of a process in PLVCMPS are satisfied one at a time as in algorithm COMPS. The same situation applies for accesses. Since the priorities of requests are determined by the levels in which they are placed in algorithm COM, in contrast to algorithm COMPS, not all elements of $C\text{-set}(R)$ (depending on what level $R$ is at) are checked as mentioned earlier.

A process can normally terminate an access if it is not blocking a request of higher priority. When an access blocks a request of higher priority it is preempted, i.e., the access is replaced by the corresponding request, preventing deadlock from occurring, even if working memory contains a deadlock set containing requests and accesses. A deadlock set can not consist of requests only since requests at any one level are compatible and requests at higher levels have higher priority and can be satisfied, thus deadlock can
not occur. For each minimal incompatible set, one and only one access is preemptable, i.e., the one with lowest priority. Thus, the number of accesses for which there will be preemption productions will be at most as large as the number of minimal incompatible sets.
procedure COM (Q: a set of requests, I: minimal incompatible sets of requests, Q': an ordered partition or Q)

/* Input: Q and I. */
/* Output: Q'. */

begin
  L = 0
  for every process k
    for every request R_i,j,k
      if R_i,j,k and at least one request at level L and any requests at higher levels than L form a minimal incompatible set
      then
        L = L + 1
      endif
      Place R_i,j,k at level L.
    endfor
  endfor

  Q' = \{ Q'_i \mid Q'_i is the set of requests at level i, 0 \leq i \leq L \}
endbegin

Figure 18
procedure PLVCMPs(S: set of requests, Q: minimal incompatible sets of requests, PS: a production system)

/* Algorithm PLVCMPs constructs a compatible live PS using preemption. */ /* Input: S and Q. */ /* Output: PS. */

begin

(1) Let the WM alphabet contain the symbols I_k, R_i,j,k and A_i,j,k for all relevant values of i,j,k.
Let the initial configuration be \( \gamma_0 = I_0 \cdots I_{N-1} \)
N = the number of processes.

(2) Construct an ordered partition on S using algorithm COM. Each element of the partition contains a compatible set of requests.

(3) for every process k
Let \( R_{i_1,j_1,k}, \ldots, R_{i_m,j_m,k} \) be the requests of process k.

a. for every request \( R_{i,j,k} \)
Include the following productions in \( P \), the set of productions, to allow for the occurrence of \( R_{i,j,k} \). A request can be made if the process has no pending requests and if the process is not already exercising the corresponding access.

\[
((I_i)(\text{fail } R_{i_1,j_1,k}) \cdots \\
(\text{fail } R_{i_m,j_m,k})(\text{fail } A_{i,j,k})
\rightarrow (\text{Insert } R_{i,j,k}))
\]

Figure 19
\[(I_i)(\text{fail } R_{i_1}, j_1, k_1) \ldots \]
\[(\text{fail } R_{i_m}, j_m, k_m)(\text{fail } A_{i_1}, j_1, k_1) \rightarrow (\text{Noop})\]

Note that no priorities are explicitly associated with requests.

**b.** for every request \(R_{i,j,k}\)

- Construct the B-set\((R_{i,j,k})\) and C-set\((R_{i,j,k})\) using algorithms BSETS and CSETS.

- for every element \(X\) in B-set\((R_{i,j,k})\)
  - Let \(A_{i_1}, j_1, k_1, \ldots, A_{i_L}, j_L, k_L\) be the accesses in \(X\).
  - Construct a condition \(c\) of the form
    \[(\text{fail } (A_{i_1}, j_1, k_1 \wedge \ldots \wedge A_{i_L}, j_L, k_L))\]

- endfor

- Let \(c_1 = (c_1, c_2, \ldots, c_L)\) be the sets of such conditions.

- for every element \(x\) in C-set\((R_{i,j,k})\)
  - such that all requests in \(x\) are at the same level as \(R_{i,j,k}\) or higher
    - Let \(R_{i_1}, j_1, k_1, \ldots, R_{i_L}, j_L, k_L\) be the requests in \(x\); and \(A_{i_1}, j_1, k_1, \ldots, R_{i_L}, j_L, k_L\) the accesses in \(x\).
    - Construct a condition \(c'\) of the form:

**Figure 19 continued**
Let $c_2 = (c'_1, c'_2, \ldots, c'_M)$ be the set of all such conditions.

Note that the $C$-set conditions here do not include any terms involving priorities as in algorithm COMPS. This is a result of the fact that the priorities are preset and implicitly defined by the levels.

Include the following productions in $P$ to satisfy $R_{i,j,k}$. A request can be satisfied if none of its $B$-set elements are in working memory and if none of its $C$-set elements in which all requests have priorities equal to or higher than the request itself are in working memory.

$$\((R_{i,j,k}) c_1 c_2 \ldots c_L c_i c'_2 \ldots c'_M \rightarrow (\text{Replace}(R_{i,j,k}, A_{i,j,k}))\)$$

$$\((R_{i,j,k}) c_1 c_2 \ldots c_L c_i c'_2 \ldots c'_M \rightarrow (\text{Noop})\)$$

Include the above two productions as one conflict set in $R$.

Figure 19 continued
c. for every access $A_{i,j,k}$

if $A_{i,j,k}$ is not at a lowest level of a minimal incompatible set

then

There are no preemption productions for $A_{i,j,k}$.

Include the following productions in $\mathcal{P}$ to allow an access to normally terminate or continue in progress.

A process can terminate an access if it has no pending requests.

\[ ((A_{i,j,k})(\text{fail } R_{i_1,j_1,k}) ... \\
(fail R_{i_m,j_m,k}) \rightarrow (\text{Delete } A_{i,j,k})) \]

\[ ((A_{i,j,k})(\text{fail } R_{i_1,j_1,k}) ... \\
(fail R_{i_m,j_m,k}) \rightarrow (\text{Noop})) \]

else

There are preemption productions for $A_{i,j,k}$. Thus, a process can normally terminate an access only if conditions for preemption do not hold.

Let the elements of C-set($A_{i,j,k}$) in which all requests and accesses have higher priority than $A_{i,j,k}$ be $x_1, ..., x_2$. Let the corresponding C-set conditions as constructed in step b be $\{c_1, c_2, ..., c_L\}$.

Include the following productions in $\mathcal{P}$ to allow normal termination of $A_{i,j,k}$ or its continuation.

Figure 19 continued
Thus, process $k$ can normally terminate $A_{i,j,k}$ if it has no pending requests and if the conditions for preemption do not hold.

Include the following productions to preempt $A_{i,j,k}$. An access is preempted if any of the conditions for preemption hold when applied to working memory, i.e., when there are contending requests with higher priority.

\[
((A_{i,j,k})(\text{fail } R_{i_1,j_1,k}) \ldots \text{ fail } R_{i_m,j_m,k}) \ c_1 \ c_2 \ldots \ c_L, \\
\longrightarrow (\text{Delete } A_{i,j,k})
\]

\[
((A_{i,j,k})(\text{fail } R_{i_1,j_1,k}) \ldots \text{ fail } R_{i_m,j_m,k}) \ c_1 \ c_2 \ldots \ c_L, \\
\longrightarrow (\text{Noop})
\]

Figure 19 continued
(4) Let $\beta$, the size of working memory $= (\text{the number of requests} + \text{the number of processes})$. There is an $I_k$ for each process $k$ and either $A_{i,j,k}$ or $R_{i,j,k}$ for each request. As mentioned earlier, the priorities are implicitly defined by the levels of the partition of requests.
Example 6

In this example, we present another solution to the dining philosophers problem using algorithm PLVCMPS. We will use the same symbols and variables used earlier.

Using algorithm COM, and considering the requests in the order of \( R_{N-1,0}, R_{0,0}, R_{0,1}, R_{1,1}, \ldots, R_{N-1,N-1} \), we get the following ordered partition:

\[
Q'_0 = \{R_{N-1,0}, R_{0,0}\}, \quad Q'_1 = \{R_{0,1}, R_{1,1}\}, \ldots, \quad Q'_{N-1} = \{R_{N-2,N-1}, R_{N-1,N-1}\}.
\]

Requests in \( Q'_0 \) have the highest priority and requests in \( Q'_{N-1} \) have the lowest priority. Figure 20 shows the different levels and the corresponding requests.

The initial configuration is \( \chi_0 = I_0 \ldots I_{N-1} \), \( N \) = the number of philosophers.

Following step 3a of PLVCMPS, we get the following productions representing the occurrence of requests. In all of these productions a philosopher can request a fork if he does not have a pending request for either fork and if he has not already picked up the fork being requested.

\[
P_{ia'} \quad ((I_i)(\text{fail } R_{i-1,i})(\text{fail } R_{i,i})(\text{fail } A_{i-1,i}) \quad \rightarrow \quad (\text{Insert } R_{i-1,i}))
\]
<table>
<thead>
<tr>
<th>Level 0</th>
<th>( R_{N-1,0} )</th>
<th>( R_{0,0} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>( R_{0,1} )</td>
<td>( R_{1,1} )</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Level N-2</td>
<td>( R_{N-3,N-2} )</td>
<td>( R_{N-2,N-2} )</td>
</tr>
<tr>
<td>Level N-1</td>
<td>( R_{N-2,N-1} )</td>
<td>( R_{N-1,N-1} )</td>
</tr>
</tbody>
</table>

**Figure 20**
\[ P_{ib} : ((I_i)(\text{fail } R_{i-1,i})(\text{fail } R_{i,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Noop})) \]

\[ P_{ic} : ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i}) \rightarrow (\text{Insert } R_{i,i})) \]

\[ P_{id} : ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i}) \rightarrow (\text{Noop})) \]

Include \{P_{ia}, P_{ib}, P_{ic}, P_{id}\}, 0 \leq i \leq N-1 in \mathcal{R}.

Step 3b of PLVCMPs provides the productions for satisfying requests. Since \( R_{i,i}, 0 \leq i \leq N-2 \), has higher priority than \( R_{i,i+1} \) which is the only element of \( C\text{-set}(R_{i,i}) \), the \( i \)th philosopher has only to check if the \((i+1)\)st philosopher has not already picked up the \( i \)th fork. Thus the productions for requests \( R_{i,i}, 0 \leq i \leq N-2 \) are of the form:

\[ P_{(N+i)a} : ((R_{i,i})(\text{fail } A_{i,i+1}) \rightarrow (\text{Replace}(R_{i,i}, A_{i,i}))) \]

\[ P_{(N+i)b} : ((R_{i,i})(\text{fail } A_{i,i+1}) \rightarrow (\text{Noop})) \]

Include \{\( P_{(N+i)a}, P_{(N+i)b} \}, 0 \leq i \leq N-2 in \mathcal{R}.

For \( R_{N-1,N-1} \), the request \( R_{N-1,0} \) is in \( C\text{-set}(R_{N-1,N-1}) \) and has higher priority. Thus, the \((N-1)\)st philosopher must check if the zeroth philosopher has not already picked up
the (N-1)st fork and if the zeroth philosopher does not have a request for the same fork. The productions for $R_{N-1,N-1}$ are as follows:

\[ P(2N-1)a' \rightarrow ((R_{N-1,N-1})(\text{fail } A_{N-1,0})(\text{fail } R_{N-1,0}) \rightarrow (\text{Replace}(R_{N-1,N-1}, A_{N-1,N-1}))) \]

\[ P(2N-1)b' \rightarrow ((R_{N-1,N-1})(\text{fail } A_{N-1,0})(\text{fail } R_{N-1,0}) \rightarrow (\text{Noop})) \]

Include \{P(2N-1)a', P(2N-1)b\} in $\mathcal{R}$.

For request $R_{N-1,0}$, there are no elements of $\text{C-set}(R_{N-1,0})$ that have higher priority. Thus the zeroth philosopher has only to check if the (N-1)st has not already picked up the zeroth fork. The productions for $R_{N-1,0}$ are as follows:

\[ P_{2Na} : ((R_{N-1,0})(\text{fail } A_{N-1,N-1}) \rightarrow (\text{Replace}(R_{N-1,0}, A_{N-1,0}))) \]

\[ P_{2Nb} : ((R_{N-1,0})(\text{fail } A_{N-1,N-1}) \rightarrow (\text{Noop})) \]

Include \{P_{2Na}, P_{2Nb}\} in $\mathcal{R}$.

For requests $R_{i-1,i}$, $1 \leq i \leq N-1$, the requests $R_{i-1,i-1}$ form the respective elements of $\text{C-set}(R_{i-1,i})$ and have higher priority than $R_{i-1,i}$. Thus the ith philosopher must not only check if the (i-1)st philosopher has already picked up
the (i-1)st fork but also if the (i-1)st philosopher does not have a request for the same fork. The productions for $R_{i-1,i}, 1 \leq i \leq N-1$ are as follows:

\[ P(2N+i)a' \equiv ((R_{i-1,i})(fail A_{i-1,i-1})(fail R_{i-1,i-1}) \rightarrow (Replace(R_{i-1,i}, A_{i-1,i-1}))) \]

\[ P(2N+i)b' \equiv ((R_{i-1,i})(fail A_{i-1,i-1})(fail R_{i-1,i-1}) \rightarrow (Noop)) \]

Include \{P(2N+i)a', P(2N+i)b'\}, $1 \leq i \leq N-1$ in $\mathcal{R}$.

Applying step 3c of algorithm PLVCMS, we get the following productions to allow the zeroth philosopher to normally terminate access to either fork on his right and left. Since $R_{N-1,0}$ is not at a lowest level of the requests of the minimal incompatible set \{RN-1,0, RN-1,N-1\}, there are no preemption productions for AN-1,0. Furthermore the zeroth philosopher can terminate AN-1,0 if he has no pending request. The same situation applies for A0,0. The productions allowing the normal termination of AN-1,0 and A0,0 are as follows:

\[ P_{3Na} \equiv ((A_{N-1,0})(fail R_{0,0}) \rightarrow (Delete A_{N-1,0})) \]

\[ P_{3Nb} \equiv ((A_{N-1,0})(fail R_{0,0}) \rightarrow (Noop)) \]

\[ P_{3Nc} \equiv ((A_{0,0})(fail R_{N-1,0}) \rightarrow (Delete A_{0,0})) \]
For accesses $A_{i,i}$, $1 \leq i \leq N-2$ there are no elements of $C$-set($A_{i,i}$) with higher priorities, thus the productions representing normal termination of those accesses will be analogous to those of $A_{0,0}$ and $A_{N-1,0}$. In addition, there are no preemption productions for these accesses.

Accesses $A_{i-1,i}$, $1 \leq i \leq N-2$, are at a lowest level of the respective minimal incompatible sets $\{A_{i-1,i-1}, A_{i-1,i}\}$. Therefore, the $i$th philosopher can normally terminate accessing the $(i-1)$st fork if he has no requests and if the $(i-1)$st philosopher does not have a request for the same fork.

The following productions represent normal termination of accesses $A_{i-1,i}$ and $A_{i,i}$, $1 \leq i \leq N-2$.

$$P_{(3N+i)a} (A_{i-1,i})(\text{fail } R_{i,i})(\text{fail } R_{i-1,i-1}) \rightarrow (\text{Delete } A_{i-1,i})$$

$$P_{(3N+i)b} (A_{i-1,i})(\text{fail } R_{i,i})(\text{fail } R_{i-1,i-1}) \rightarrow (\text{Noop})$$

$$P_{(3N+i)c} (A_{i,i})(\text{fail } R_{i-1,i}) \rightarrow (\text{Delete } A_{i,i})$$
\[ P(3N+i)d: ((A_i,i)(\text{fail } R_{i-1,i}) \rightarrow (\text{Noop})) \]

Include \( \{P(3N+i)a, P(3N+i)b, P(3N+i)c, P(3N+i)d\} \), \( 1 \leq i \leq N-2 \) in \( R \).

The preemption productions for \( A_{i-1,i} \) will be presented later along with the other preemption productions for the other requests. The preemption productions are of the same form, thus we are avoiding unnecessary redundancy.

Accesses \( A_{N-2,N-1} \) and \( A_{N-1,N-1} \) are both at a lowest level of those requests of the respective minimal incompatible sets \( \{A_{N-2,N-2}, A_{N-2,N-1}\} \) and \( \{A_{N-1,N-1}, A_{N-1,0}\} \). Thus the \( (N-1) \text{st} \) philosopher can only terminate access to either fork if none of the respective philosophers on his right and left have pending requests for those forks and if he does not have a pending request. In addition, there are preemption productions for both accesses. Thus, we have the following productions

\[ P(4N-1)a: ((A_{N-2,N-1})(\text{fail } R_{N-1,N-1})(\text{fail } R_{N-2,N-2}) \rightarrow (\text{Delete } A_{N-2,N-1})) \]

\[ P(4N-1)b: ((A_{N-2,N-1})(\text{fail } R_{N-1,N-1})(\text{fail } R_{N-2,N-2}) \rightarrow (\text{Noop})) \]

\[ P(4N-1)c: ((A_{N-1,N-1})(\text{fail } R_{N-2,N-1})(\text{fail } R_{N-1,0}) \rightarrow (\text{Delete } A_{N-1,N-1})) \]
The preemption productions for $A_{N-1,N-1}$ are as follows:

$$P_{4Na} : ((A_{N-1,N-1})(R_{N-1,0}) \rightarrow (\text{Replace}(A_{N-1,N-1}, R_{N-1,N-1})))$$

Include $\{P_{4Na}, P_{4Nb}\}$ in $\mathcal{R}$. 

The preemption productions for $A_{i-1,i} \leq i \leq N-1$, are as follows:

$$P_{4Na} : ((A_{i-1,i})(R_{i-1,i-1}) \rightarrow (\text{Replace}(A_{i-1,i}, R_{i-1,i})))$$

$$P_{4Nb} : ((A_{i-1,i})(R_{i-1,i-1}) \rightarrow (\text{Noop}))$$

Include $\{P_{4Na}, P_{4Nb}\}$ in $\mathcal{R}$. 

In the above preemption productions, an access is preempted if there is a contending request for the same fork with higher priority (at higher levels).

The size of $WM$, $\beta = (2*N+1) = 3N$. 

Lemma

Algorithm PLVCMPS constructs a compatible live production system.

Proof:

We first prove that the production system is compatible. Let $\mathcal{C}$ be a configuration containing a set of contending requests $Q = \{R_1, \ldots, R_N\}$, i.e., if the requests in $Q$ are satisfied, this will result in a transition $\mathcal{C} \rightarrow \mathcal{C}'$ such that $\mathcal{C}'$ contains a minimal incompatible set of accesses. If $Q$ consists of one request $R$, then, for a minimal incompatible set of accesses to occur, $\mathcal{C}$ must contain an element of the $B\text{-set}(R)$ in which case $R$ will not be satisfied. If $Q$ consists of more than one request, then those requests of $Q$ at the lowest level will be blocked by other requests of $Q$ at the higher levels, and $\mathcal{C}'$ will not contain an incompatible set of accesses. If all requests of $Q$ are at the same level, they will block each other until an access is preempted. Thus no incompatible set of accesses will occur in this case either. The above two situations are enforced by the C-set conditions developed in the algorithm.
To prove liveness, assume that the production system is not live. That is, there is a configuration \( \mathcal{C} \) which contains a set of requests and/or accesses \( X = \{x_0, \ldots, x_{n-1}\} \) such that each \( x_i \mod n \) is indefinitely blocked by \( x_{(i+1)\mod n} \). \( X \) does not consist of requests only. The requests of \( X \) can not be at one level since requests at any one level are compatible. If the requests are at different levels, then requests at the highest level can not be blocked by requests at lower levels and \( X \) can not be deadlocked.

If \( X \) consists of requests and accesses at any one level then \( X \) can not be deadlocked since requests and accesses at any one level are compatible. If the requests and accesses of \( X \) are at different levels and a request is blocked by an access at a lower level (lower priority), then the access will be preempted, \( X \) will not be deadlocked, and the process with the highest priority will be allowed to terminate normally. Requests in \( X \) can not all be blocked by accesses at higher levels (higher priority). This corresponds to the situation illustrated by Figure 9. We have demonstrated earlier that such a situation is not possible. If it is possible this would imply that for a process, all requests are not at one level or at consecutive levels, thus, contradicting the construction procedure by algorithm COM. If \( X \) only consists of accesses, then \( X \) can not be deadlocked and
the proof is complete.

In algorithm PLVCMPs, although deadlock can not occur, requests and accesses with higher priorities can indefinitely block requests with lower priorities. This is the starvation problem which we consider in the following section.
5. Specification of Fair Systems

Consider a sequence of transitions $\gamma \xrightarrow{+} \gamma'$ in which one of two situations is possible. The first is that a request $R$ appears in every configuration as a result of executing the null action production in every transition. Although $R$ appears in every configuration, $R$ is not blocked because of a policy decision and may be satisfied in every transition of $\gamma \xrightarrow{+} \gamma'$. The second possibility is that a request $R$ is blocked by another request or access for which the null action production is chosen to be executed in every transition of $\gamma \xrightarrow{+} \gamma'$. In this case $R$ is blocked by another request or access that is not blocked because of a policy decision. In both situations, since $R$ is not blocked (directly or indirectly) by a policy decision, we will exclude such situations from our definition of starvation. The following definition is a formalization of the situations described above.
5.1 Definitions

A request or access $x$ is **statically-blocked** in a sequence of transitions $\mathcal{G} \xrightarrow{\tau} \mathcal{G}'$ if:

1. In every configuration in the transitions $\mathcal{G} \xrightarrow{\tau} \mathcal{G}'$, either the productions whose actions modify $x$ are enabled but the corresponding null action (Noop) productions are chosen to execute; or $x$ is blocked by statically blocked requests or accesses, and

2. $x$ is not an element of a deadlocked set in any configuration in the transitions $\mathcal{G} \xrightarrow{\tau} \mathcal{G}'$.

In condition (1) in the above definition, if $x$ is blocked by a statically-blocked request or access which is in turn blocked by a statically-blocked request or access and so on, the last request or access in the sequence of statically-blocked requests and/or accesses will be statically-blocked as a result of the null action (Noop) productions being chosen to execute in $\mathcal{G} \xrightarrow{\tau} \mathcal{G}'$.

A request or access $x$ can be **starved** if there is a reachable configuration $\mathcal{G}$ such that:
(1) \( \mathcal{C} \) contains \( x \), and \( x \) is not an element of a deadlock set in \( \mathcal{C} \), and

(2) If there is any transition from \( \mathcal{C} \), then there is a sequence of transitions \( \mathcal{C} \rightarrow \mathcal{C} \) in which every configuration contains \( x \), and \( x \) is not statically-blocked in \( \mathcal{C} \rightarrow \mathcal{C} \).

Note that if a configuration contains an access or request \( x \) and there is no transitions from \( \mathcal{C} \), then \( x \) can be starved.

**Corollary**

If a request or access \( x \) is starved in \( \mathcal{C} \rightarrow \mathcal{C} \), then \( x \) is blocked (but not statically-blocked) in at least one configuration in \( \mathcal{C} \rightarrow \mathcal{C} \).

**Proof:** obvious.

A production system is **fair** if no request or access can be starved, otherwise the production system is unfair.

Algorithm COMPS, and LVCMPS presented in the previous sections do not construct fair production systems if the priorities of the requests of a minimal incompatible set
Q = \{R_1, \ldots, R_N\} are fixed and non-cyclic. Since the priorities of requests in Q are fixed and non-cyclic, then, without loss of generality, we may assume \( p_1 > \cdots > p_N \).

Consider the sequence of transitions

\[ \mathcal{G}_1 = x_0 R_1 \cdots R_N x_N \rightarrow x_0 A_1 x_1 A_2 \cdots A_{N-1} x_{N-1} R_N x_N = \mathcal{G}_2, \]

\[ \mathcal{G}_2 = x_0 A_1 x_1 A_2 \cdots A_{N-1} x_{N-1} R_N x_N \rightarrow x_0 x_1 \cdots x_{N-1} R_N x_N, \]

\[ \mathcal{G}_3 = x_0 x_1 \cdots x_{N-1} R_N x_N \rightarrow x_0 R_1 x_1 R_2 \cdots R_N x_N = \mathcal{G}_1. \]

In the first two transitions \( R_N \) is blocked by \( \{R_1, \ldots, R_{N-1}\} \) and \( \{A_1, \ldots, A_{N-1}\} \) respectively. In the last transition the null action production for \( R_N \) is executed. \( R_N \) is not statically-blocked in \( \mathcal{G} \rightarrow \mathcal{G} \), thus can starved, because the policy represented might grant accesses to \( R_1, \ldots, R_{N-1} \), before it gets around to \( R_N \) again.

In algorithm PLVCMP, priorities are fixed and non-cyclic, and requests with higher priorities can block requests with lower priorities such that the latter can be starved.

### 5.2 Determining Fairness

Algorithm FAIR, shown in Figure 21, determines if a production system is fair. Every configuration \( \mathcal{G} \) is tested for starvation. If there are no transitions possible from \( \mathcal{G} \) and \( \mathcal{G} \) contains requests and/or accesses which are not deadlocked, then the production system is not fair. That is,
requests and/or accesses can be starved.

For the transitions $\emptyset \rightarrow \emptyset$, if a non-empty set $X$ of requests and accesses appears in every configuration, then function STRVSET is called upon to determine if $X$ contains any element that can be starved.

STRVSET, shown in Figure 22, is analogous to algorithm CYCSET presented earlier. The two functions can be merged to return two values, each indicating whether the set of requests and accesses contains any deadlock set and whether any request or access can be starved.

Algorithm STRVSET determines if there are any elements of $X$ that can be starved in two steps:

1. Statically-blocked elements are deleted. An element of $X$ is deleted if it is not blocked in any of the configurations of $\Gamma$ or if it is only blocked by elements which have been deleted. This step is repeated until no more elements can be deleted.

2. Elements that are only blocked and block other elements of $X$ (after statically-blocked elements are deleted) are also deleted. As a result, elements that form a deadlock set and are only blocked by the deadlock set are deleted. Figure 23
procedure FAIR(PS: a production system, S: minimal incompatible sets of accesses)

/* Input: PS, and S. */
/* Output: whether PS is fair or unfair. */

begin
  for every reachable configuration \( S \) 
  if there are no transitions possible from \( S \) 
  then if \( S \) contain at least one request or access 
  then 
    Let \( \bar{S} = \{S\} \) and \( X \) be the set of accesses and requests in \( S \) 
  endif 
  else 
    for every sequence of transitions \( S \xrightarrow{\text{ transitions }} S' \) 
    Let \( X \) be the set of requests and accesses such that each element of \( X \) appears in every configuration in \( S \xrightarrow{\text{ transitions }} S' \). 
    Let \( \bar{X} = \) the set of configurations in \( S \xrightarrow{\text{ transitions }} S' \). 
  endfor 
  endif 

  if STRVSET(\( X, \bar{X}, PS \)) 
  then PS is unfair 
    return unfair 
  endif 
endfor 

PS is fair 

endbegin 

Figure 21
function STRVSET(X: a set of accesses and requests, V: a set of configurations, PS: a production system)

/* Input: X, V, and PS. */
/* Output: STRVSET is set to "true" if X contains any request or access that can be starved, otherwise it is set to "false". */

begin
X' = X
/* delete statically-blocked elements from X' */
FLAG = true
while (FLAG) do
  FLAG = false
  for every x in X'
    if x is not blocked in any configuration of V or x is blocked only by elements which have already been removed from X'
      then delete x from X'
      FLAG = true
    endif
  endfor
endwhile
/* delete elements that are blocked and block other requests. */
X" = X'
for every x" in X"
  if (x" is only blocked by elements of X' in V and x" blocks an element of X' in V')
    then delete x" from X"
  endif
endfor
if |X"| > 1
  then /* elements in X" can be starved */
    STRVSET = true
  else
    STRVSET = false
  endif
endbegin

Figure 22
Figure 23

Figure 24
shows a deadlock set \( \{x_0, x_1, x_2\} \) and an element \( x_3 \) which is blocked by element \( x_2 \) of the deadlock set. In the Figure an arc from node \( x_i \) to \( x_j \) means \( x_i \) is blocked by \( x_j \). Applying this second step to the set \( \{x_0, x_1, x_2, x_3\} \), \( x_0, x_1, x_2 \) are deleted, and \( x_3 \) is not deleted. That is, \( x_3 \) can be starved.

In addition, in this step, the elements of deadlock sets are not the only elements deleted. For instance, consider Figure 24. Each \( x_i, 0 \leq i \leq 2 \) is blocked by \( x_{i+1} \). However, \( x_1 \) is blocked by \( x_2 \) and blocks \( x_0 \), and is deleted. Similarly \( x_2 \) is also deleted. \( x_0 \) and \( x_1 \) are not deleted and thus, can be starved. Note that we do not necessarily end up with all elements that can be starved.

However, if there is any, we will end up with at least one element, which is sufficient to determine if the production system is fair or unfair.

Note that we define starvation for a single request.

As a consequence, as mentioned earlier, starvation in terms of a simultaneous set of requests or accesses \( Q \) is not representable within this definition. To overcome this shortcoming, one can associate an artificial request representing the simultaneous requests. The request is introduced into working memory whenever a first request of \( Q \) occurs and is
satisfied whenever a last request of Q is satisfied.
Example 7 A Readers/Writers Problem.

Two kinds of processes, called readers and writers, share a single resource. The readers can use the resource simultaneously, but each writer must have exclusive access [DIJK 68], [HANS 73].

Let \( N \) be the number of readers and \( M \) the number of writers. We will denote the readers as processes 0 through \( N-1 \), and the writers as processes \( (N) \) through \( (N+M-1) \). Let \( R_i, 0 \leq i \leq N-1 \) be read requests, and \( R_i, N \leq i \leq N+M-1 \) be write requests. Let \( L = N+M \).

Let the productions allowing the occurrence of requests be:

\[
P_{ia} : \quad ((I_i)(fail R_i)(fail A_i) \quad \quad \quad \rightarrow \quad \text{(Insert } R_i)(\text{Assign } p_i))
\]

\[
P_{ib} : \quad ((I_i)(fail R_i)(fail A_i) \quad \quad \quad \rightarrow \quad \text{(Noop)})
\]

The above productions are defined for read and write requests, i.e., \( 0 \leq i \leq L-1 \). Include each \( \{P_{ia}, P_{ib}\}, 0 \leq i \leq L-1 \) in \( \mathcal{R} \). Each reader and writer only has one request. Thus, there is no need to check for requests by the same process because there can not be any. Each process must check if it has not already had access to the resource.
Let the productions in which a read request can be satisfied be:

\[ P_{(L+i)a} : ((R_i)(\text{fail } A_N)\ldots(\text{fail } A_{L-1}) \]
\[ \quad (\text{fail } R_N)\ldots(\text{fail } R_{L-1}) \]
\[ \quad \rightarrow (\text{Replace}(R_i, A_i)) \]

\[ P_{(L+i)b} : ((R_i)(\text{fail } A_N)\ldots(\text{fail } A_{L-1}) \]
\[ \quad (\text{fail } R_N)\ldots(\text{fail } R_{L-1}) \]
\[ \quad \rightarrow (\text{Noop}) \]

That is, a reader can get access if no writer is writing or making a write request. Include \( P_{(L+i)a}, P_{(L+i)b} \mid 0 \leq i \leq N-1 \) in \( \mathcal{R} \).

Let the analogous productions for write requests be:

\[ P_{(L+i)a} : ((R_i)(\text{fail } A_0)\ldots(\text{fail } A_{N-1}) \]
\[ \quad (\text{fail } A_N)\ldots(\text{fail } A_{i-1}) \]
\[ \quad (\text{fail } A_{i+1})\ldots(\text{fail } A_{L-1}) \]
\[ \quad (\text{fail } (R_N \land (p_N \geq p_i)))\ldots \]
\[ \quad (\text{fail } (R_{i-1} \land (p_{i-1} \geq p_i))) \]
\[ \quad (\text{fail } (R_{i+1} \land (p_{i+1} \geq p_i)))\ldots \]
\[ \quad (\text{fail } (R_{L-1} \land (p_{L-1} \geq p_i))) \]
\[ \quad \rightarrow (\text{Replace}(R_i, A_i)) \]
\[ P_{(L+i)b} :\]
\[ ((R_i) (\text{fail } A_0) \ldots (\text{fail } A_{N-1})\]
\[ (\text{fail } A_N) \ldots (\text{fail } A_{i-1})\]
\[ (\text{fail } A_{i+1}) \ldots (\text{fail } A_{L-1})\]
\[ (\text{fail } (R_N \land (p_N \geq p_i))) \ldots\]
\[ (\text{fail } (R_{i-1} \land (p_{i-1} \geq p_i)))\]
\[ (\text{fail } (R_{i+1} \land (p_{i+1} \geq p_i))) \ldots\]
\[ (\text{fail } (R_{L-1} \land (p_{L-1} \geq p_i)))\]
\[ \rightarrow (\text{Noop})\]

A writer can write if no reader is reading and no other writer is writing or making a write request with higher priority. Include \( \{P_{(L+i)a}, P_{(L+i)b}\}, \ N \leq i \leq L-1, \) in \( \mathcal{R} \).

Let the productions allowing accesses to terminate or continue in progress be:

\[ P_{(2L+i)a} :\]
\[ ((A_i) \rightarrow (\text{Delete } A_i))\]

\[ P_{(2L+i)b} :\]
\[ ((A_i) \rightarrow (\text{Noop}))\]

Since no reader or writer can have a pending request while accessing the resource, accesses can be terminated without checking for pending requests. Include \( \{P_{(2L+i)a}, P_{(2L+i)b}\}, 0 \leq i \leq L-1 \) in \( \mathcal{R} \). The above productions are defined for read and write requests, i.e., \( 0 \leq i \leq L-1 \).
Using algorithm FAIR, consider the sequence of transitions \( \mathcal{C}_1 = R_1 I_1 I_2 \cdots I_N R_N I_{N+1} \cdots I_{L-1} \rightarrow \)
\( R_1 I_1 I_2 \cdots A_N I_N I_{N+1} \cdots I_{L-1} = \mathcal{C}_2 \rightarrow \)
\( R_1 I_1 I_2 \cdots I_N I_{N+1} \cdots I_{L-1} = \mathcal{C}_3 \rightarrow \)
\( R_1 I_1 I_2 \cdots R_N I_N I_{N+1} \cdots I_{L-1} = \mathcal{C}_1. \) \( \mathcal{C}_1 \) is reachable from the initial configuration \( \mathcal{C}_0. \) In the first and second transitions \( R_1 \) is blocked by \( R_N \) and \( A_N, \) respectively. In the third transition \( R_1 \) is not blocked and the null action production for \( R_1 \) is executed. \( R_1 \) is the only request that appears in all configurations. Applying function \texttt{STRESET}, we find that \( R_1 \) is not statically-blocked since \( R_1 \) is blocked in at least one configuration, i.e., in \( \mathcal{C}_1 \) and \( \mathcal{C}_2, \) and \( R_1 \) is blocked by \( R_N \) and \( A_N \) which do not appear in every configuration. \( R_1 \) is not an element of a deadlocked set since it is not blocked by the same element in all configurations. We thus end up with \( R_1 \) being starved, and consequently the PS is not fair.
5.3 Designing Fair Systems

It turns out that we do not need to develop any additional machinery to produce fair systems. The theorem below shows that the design procedures already studied are sufficient if we make certain assumptions about the priorities, and the incompatible sets.

Definitions

A policy (a production system) is an alternation policy if for every request or access $x$ that occurs, only a finite number of occurrences of requests and accesses that can block $x$ have higher priority than $x$ while $x$ is in the system (WM).

A special class of alternation policies are first-come-first-serve policies in which for every request or access $x$ that occurs after (in a later cycle of the production system) a request $y$, $x$ has lower priority than $y$.

An alternation policy may assign two requests the same priority. Thus, in general, deadlock may occur within such a policy.
The Fairness Theorem

Algorithm COMPS constructs a fair production system if and only if the following two conditions hold:

1. Request priorities are assigned by an alternation policy.

2. For every set of requests and accesses
   \[ X = \{x_0, \ldots, x_{n-1}\} \] such that \( x_i \mod n \) can block
   \( x_{(i+1)\mod n} \), no request or access \( x_i \in X \) blocks
   any element which is not in \( X \).

The second condition is necessary because requests and accesses of a deadlock set can block other requests and accesses forever, thus starving them.

Proof:

We will prove the "if" clause first. Assume, that the
PS is unfair, i.e., a request or access \( x \) can be starved;
and there is a sequence of transitions \( \emptyset \rightarrow \emptyset \) such that
\( x \) is not statically-blocked in \( \emptyset \rightarrow \emptyset \), \( x \) is not an ele-
ment of a deadlock set in \( \emptyset \rightarrow \emptyset \) and \( x \) is blocked by a set
of requests and accesses in at least one configuration in
\( \emptyset \rightarrow \emptyset \). Let \( S \) be such a set. \( S \) must appear in every
configuration in \( \emptyset \rightarrow \emptyset \). If \( S \) does not appear in every
configuration in $\mathcal{X} \xrightarrow{t} \mathcal{X}$, then this implies that some requests in $S$ are satisfied and the corresponding accesses are terminated. Since the same sequence of transitions $\mathcal{X} \xrightarrow{t} \mathcal{X}$ can occur over and over again, then an infinite number of requests that block $x$ can be satisfied before $x$ is satisfied. Examining the productions for satisfying requests in algorithm COMPS, this implies that an infinite number of requests which can block $x$ have higher priority than $x$, contradicting our assumption of an alternation policy. If $S$ appears in every configuration in $\mathcal{X} \xrightarrow{t} \mathcal{X}$, then $S$ must contain a deadlocked set which blocks other requests not in $S$, thus contradicting the second condition.

We now prove the "only if" clause. Let the PS be fair. If the priorities of requests are not assigned on an alternation basis then there is a request $x$ such that an infinite number of requests can block $x$, and those requests have higher priorities than $x$ while $x$ is in $WM$. As a consequence, $x$ will never be satisfied, and thus starved, a contradiction.

If condition (2) does not hold, then clearly, a request which is not an element of a deadlock set $X$ can be indefinitely blocked by an element $x_i$ in $X$, and thus starved, a contradiction.
As a special case derivable from the above proof, algorithm COMPS constructs a fair production system when the production system is live and the priorities are assigned on an alternation basis.

In addition, when algorithm LVCMPS constructs a live production system, the production system is also fair if the priorities are assigned on an alternation basis. Algorithm PLVCMPS does not construct a fair production system since the priorities of request are fixed and non-cyclic.
6 Conclusion

In this paper, we have demonstrated the suitability of using productions systems as a model for concurrent processing, and in particular, for specifying synchronization problems. The resulting model is concise since the specification does not require the explicit representation of the various states of the system. The states of the system are implicitly defined through the set of productions, conflict resolution and the initial configuration. In addition, the use of variables eliminates the problem of combinatorial explosion in the number of productions, which is encountered in "pure" production systems [DAVI 75].

As demonstrated in this paper, we have developed formal definitions and systematic analysis procedures for properties that are of significant importance namely, compatibility, liveness and fairness. Furthermore, we have developed design procedures for policies with the desired properties.

Upon developing the analysis and design procedures, we have effectively shown that compatibility, liveness and fairness are computable. However, the complexity of the analysis algorithms, in general, is proportional to the number of configurations in the production system being analyzed. Although the number of configurations is combinatorial in
terms of the number requests, the number of reachable configurations is limited by the incompatible sets. For instance, consider a set of four requests \( Q = \{R_1, R_2, R_3, R_4\} \) made by four different processes. Then there are \( 3^4 = 81 \) configurations. Let every pair of requests in \( Q \) be a minimal incompatible set. Then if the production system is compatible, then there are \( 2^4 + 4(2^3) = 48 \) reachable configurations.

Furthermore, since in most problems minimal incompatible sets consist of two requests, the number of elements in each of the B-set and C-set for a particular request is reduced to the number of minimal incompatible set of which the request is an element.

We have not addressed an important property of concurrent processes, i.e., "the degree of parallelism", and the interactions between this property, liveness and fairness. For an analysis of this aspect, and the implementation derivation issue mentioned earlier, refer to [GAZA 81].

The results (lemmas) derived in this paper apply to algorithms which are prototypes of certain classes of solutions. For example, algorithm COMPS does not place any restrictions on the occurrence of requests, i.e., requests by a particular process can occur at different times, while in
algorithm LVCOMPS requests are grouped into different sets and all requests in any one set can only occur simultaneously.

Further research is needed to determine equivalent classes of production systems, so that the results obtained in this paper can be applied to a wider range of algorithms. Another area of interest is the analysis of the production system models with respect to "nondeterminancy", and "consistency" problems encountered in distributed databases.
References


IV Maximal Compatibility and Implementation Derivation

This chapter consists of two sections. The first section is concerned with maximal compatibility. We present systematic analysis and design procedures for maximally compatible productions systems. In the second section, we present a systematic implementation derivation procedure. Given a production system specification, the implementation procedure generates synchronization procedures from the actions of a production system. The procedures can then be executed by competing processes without violating any property of the policy.
1. The Compatibility Set and Maximal Compatibility

In this section, we develop concepts which will lead to a formal definition of "the degree of parallelism" attained by a production system specification of a synchronization policy.

1.1 The Compatibility Set

Definitions

The compatibility set (CS) of a compatible production system consists of the non-empty compatible sets of accesses that the production system allows to occur.

A compatibility set is maximal if it contains every non-empty compatible set of accesses.

A production system $PS_1$ has a higher degree of compatibility than a production system $PS_2$ if $CS(PS_1) \supset CS(PS_2)$.

The above relation implied by the compatibility set forms a partial order on production systems for a particular problem. To illustrate, consider the following example.
Example 8

Let $PS_0$, $PS_1$ and $PS_2$ be three compatible production systems. Let $CS(PS_0) = \{ \{ A_0, A_1 \}, \{ A_1, A_2 \} \}$, $CS(PS_1) = \{ \{ A_0, A_1, A_2 \} \}$, and $CS(PS_2) = \{ \{ A_0 \}, \{ A_1 \}, \{ A_2 \}, \{ A_0, A_1 \}, \{ A_0, A_2 \}, \{ A_1, A_2 \}, \{ A_0, A_1, A_2 \} \}$. Then since $CS(PS_0) \subseteq CS(PS_2)$ and $CS(PS_1) \subseteq CS(PS_2)$ then $PS_2$ has a higher degree of compatibility than both $PS_0$ and $PS_1$. However, $PS_0$ and $PS_1$ are not comparable since $PS_0 \not\subseteq PS_1$ and $PS_1 \not\subseteq PS_0$.

When determining the compatibility set of a production system, it may be useful to first determine the maximal compatibility set (MCS). Let the set of all accesses be $S$. Let $2^S$ be the power set of $S$. Let $S'$ be the set of all minimal, and non-minimal incompatible sets of accesses. Then $MCS = 2^S - S'$. Algorithm CMPSET, shown in Figure 25, constructs the compatibility set of a production system by examining each reachable configuration $\gamma$ and including the set of accesses that appear in $\gamma$ as one element in the compatibility set.

Algorithm CMPSET and algorithm COMPAT presented earlier for determining the compatibility of a production system can be merged into one procedure. The compatibility set, then,
would only be meaningful if the production system is found to be compatible.

**procedure** CMPSET(PS: a compatible production system, CS: the compatibility set of PS)

/* Input: PS. */ Output: CS.

begin
  Let CS be initially empty.
  for every reachable configuration \( \mathcal{Q} \) 
    Let \( s \) be the set of accesses in \( \mathcal{Q} \)
    if not ((s is in CS) or (s is empty))
      then include \( s \) in CS
    endif
  endfor
endbegin

Figure 25

**Example 9**

Consider a set of three requests \( R_0, R_1, R_2 \). Let the minimal incompatible sets of accesses be \( s_0 = \{A_0, A_1\} \) and \( s_1 = \{A_1, A_2\} \). Then \( \text{MCS} = \{\{A_0\}, \{A_1\}, \{A_2\}, \{A_0, A_2\}\} \).

Consider a production system \( \text{PS} = (\Sigma, \mathcal{P}, \mathcal{R}, \mathcal{Q}, \beta) \) where \( \Sigma = \{I_0, I_1, I_2, R_0, R_1, R_2, A_0, A_1, A_2\} \). \( \mathcal{P} \) consists of the following productions:

\[ P_{ia}: ((I_i) (\text{fail } R_i) (\text{fail } A_i)) \]

\( \rightarrow \) (Insert \( R_i \))
$P_{ib}': ((I_1)(\text{fail } R_1)(\text{fail } A_1) \rightarrow (\text{Noop}))$

$\mathcal{R}$ contains each set $\{P_{ia}, P_{ib}'\}$, $0 \leq i \leq 2$.

$P_{3a}': ((R_0)(\text{fail } R_1)(\text{fail } A_1) \rightarrow (\text{Replace}(R_0, A_0)))$

$P_{3b}': ((R_0)(\text{fail } R_1)(\text{fail } A_1) \rightarrow (\text{Noop}))$

$\mathcal{R}$ contains $\{P_{3a}', P_{3b}'\}$ as one conflict set.

$P_{4a}': ((R_1)(\text{fail } R_2)(\text{fail } A_0)(\text{fail } A_2) \rightarrow (\text{Replace}(R_1, A_1)))$

$P_{4b}': ((R_1)(\text{fail } R_2)(\text{fail } A_0)(\text{fail } A_2) \rightarrow (\text{Noop}))$

$\mathcal{R}$ contains $\{P_{4a}', P_{4b}'\}$ as a conflict set.

$P_{5a}': ((R_2)(\text{fail } A_1) \rightarrow (\text{Replace}(R_2, A_2)))$

$P_{5b}': ((R_2)(\text{fail } A_1) \rightarrow (\text{Noop}))$

$\mathcal{R}$ contains $\{P_{5a}', P_{5b}'\}$ as a conflict set.

$P_{(6+i)a}': ((A_i) \rightarrow (\text{Delete } A_i))$

$P_{(6+i)b}': ((A_i) \rightarrow (\text{Noop}))$

$\mathcal{R}$ contains each $\{P_{(6+i)a}', P_{(6+i)b}'\}$, $0 \leq i \leq 2$ as a conflict set. $\gamma_0 = I_0 I_1 I_2$ and $\beta = 6$.

The reachable configurations are

$\begin{align*}
& (I_0 I_1 I_2), (R_0 I_0 I_1 I_2), (I_0 R_1 I_1 I_2), (I_0 I_1 R_2 I_2), \\
& (R_0 I_0 R_1 I_1 I_2), (R_0 I_0 I_1 R_2 I_2), (I_0 R_1 I_1 R_2 I_2), \\
& (R_0 I_0 R_1 I_1 R_2 I_2), (A_0 I_0 I_1 I_2), (A_0 I_0 R_1 I_1 I_2), \\
& (A_0 I_0 I_1 R_2 I_2), (A_0 I_0 R_1 I_1 R_2 I_2), (I_0 A_1 I_1 I_2), \\
& (R_0 I_0 A_1 I_1 I_2), (I_0 A_1 I_1 R_2 I_2), (R_0 I_0 A_1 I_1 R_2 I_2), \\
& (I_0 I_1 A_2 I_2), (R_0 I_0 I_1 A_2 I_2), (I_0 R_1 I_1 A_2 I_2).
\end{align*}$
Using algorithm CMPSET, we find that
\[ CS(PS) = \{\{A_0\}, \{A_1\}, \{A_2\}, \{A_0, A_2\}\}, \]
Thus \( CS(PS) = MCS \) and is maximal.

1.2 Maximal Compatibility

In this thesis, we consider production systems in which processing a request or an access is represented within one production rule. With this assumption in mind, we proceed to introduce the following definitions.

A transition \( \gamma \rightarrow \gamma' \) is maximally compatible if for every request \( R_{i,j,k} \) in \( \gamma \), \( R_{i,j,k} \) appears in \( \gamma' \) if and only if \( R_{i,j,k} \) is blocked by an element of B-set \( (R_{i,j,k}) \) in either \( \gamma \) or \( \gamma' \).

Intuitively, a maximally compatible transition is a transition in which the maximum number of requests are satisfied without destroying the compatibility of the production system.
Consider a configuration $\mathcal{C}$ that consists of a request $R_1$ and an access $A_2$ such that the set \{A_1, A_2\} is minimal incompatible. Then, according to the above definition, a transition $\mathcal{C} \rightarrow \mathcal{C}'$ such that $\mathcal{C}'$ contains $R_1$ but not $A_2$ is maximally compatible. $R_1$ is blocked by an element of its B-set, i.e., $A_2$ in $\mathcal{C}$. Furthermore, the transition $\mathcal{C} \rightarrow \mathcal{C}'$ where the productions with the Noop action are executed is maximally compatible. $R_1$ is blocked by $A_2$ in both configurations $\mathcal{C}$ and $\mathcal{C}'$.

If a configuration consists of a minimal incompatible set \{R_1, R_2\}, then a transition $\mathcal{C} \rightarrow \mathcal{C}'$ in which either $R_1$ or $R_2$ is satisfied is maximally compatible. That is, either $R_1$ or $R_2$ is blocked by an element of its B-set, namely, $A_2$ or $A_1$, respectively. However, the transition $\mathcal{C} \rightarrow \mathcal{C}'$ is not maximally compatible because both $R_1$ and $R_2$ are not blocked by any elements of their respective B-sets in both configurations $\mathcal{C}$ and $\mathcal{C}'$. 
A production system PS is maximally compatible if both the following conditions hold:

1. The compatibility set of PS is maximal, and
2. For every configuration $\mathcal{C}$ containing requests, there is a maximally compatible transition (possibly trivial) $\mathcal{C} \rightarrow \mathcal{C}'$.

In condition (2) in the above definition, we consider those configurations containing requests only as well as those containing requests and accesses. In addition, the compatibility set of a production system may be maximal but the production system may not be maximally compatible.

1.3 Determining Maximal Compatibility

The analysis procedure for maximal compatibility consists of two main steps. The first is to test if the compatibility set is maximal. The second verifies that for every reachable configuration $\mathcal{C}$ containing requests, there is a maximally-compatible transition even if the trivial transition is the only transition possible from $\mathcal{C}$.

Algorithm MAXCOM, shown in Figure 26, determines whether a production system is maximally compatible. The algorithm makes use of algorithm CMPSET presented earlier to determine the compatibility set of the production system under
procedure MAXCOM(PS: a compatible production system)

/* Input:   PS. */
/* Output: whether PS is maximally compatible. */

begin
  call CMPSET (PS, CS)
  if ( CS(PS) not equal MCS)
    then PS is not maximally compatible, stop.
  else
    for every reachable configuration $\xi$ such that $\xi$ contains requests
      if there is no transition $\xi \rightarrow \xi'$ such that every request $R$ that appears in both configurations $\xi$ and $\xi'$ is blocked by an element of $B$-set($R$) in either $\xi$ or $\xi'$
        then PS is not maximally compatible, stop.
      endif
    endfor
  endif
  PS is maximally compatible.
endbegin

Figure 26
Example 10

Consider the production system PS presented in example 9. We have already demonstrated that CS(PS) is maximal. We now check if the second condition for maximal compatibility is satisfied, using algorithm MAXCOM. Consider the configuration $\gamma = I_0 R_0 I_1 R_1 I_2 R_2$. The transitions possible from $\gamma$ are $\gamma \rightarrow \gamma$, and $\gamma \rightarrow I_0 R_0 I_0 R_1 I_2 A_2 = \gamma'$. It is clear that $\gamma \rightarrow \gamma$ is not maximally compatible. In the transition $\gamma \rightarrow \gamma'$, the set $Q \cap Q'$ in the algorithm is $\{R_0, R_1\}$. While $R_1$ is blocked by an element of $B$-set($R_1$) in $\gamma'$, i.e., $A_2$, $R_0$ is not blocked by any element of $B$-set($R_0$) in either $\gamma$ or $\gamma'$. Then $\gamma \rightarrow \gamma'$ is not maximally compatible either, and PS is not maximally compatible although CS(PS) is maximal.

If one wishes to compare two productions systems with respect to maximal compatibility, the conditions for the occurrence of a request R should be the same in both production systems.
1.4 Designing Maximally Compatible Production Systems

Lemma

A compatible production system constructed by algorithm COMPS is maximally compatible if and only if in every reachable configuration $\mathcal{C}$ no request $R$ that is blocked, blocks another request.

Proof:

We will first prove that the production system is maximally compatible. We will show, by contradiction, that the compatibility set is maximal. Assume that the compatibility set is not maximal. That is, there is a compatible set of accesses $S = \{A_1, \ldots, A_N\}$ such that $S$ is not in $\text{CS}(\text{PS})$.

Considering the productions, in algorithm COMPS, which allow requests to occur, there is a reachable configuration $\mathcal{C}$ which only contains requests $R_1, \ldots$ and $R_N$. None of the requests in $\mathcal{C}$ is blocked, and the processing productions for requests will allow those requests to be satisfied. Thus, there is a reachable configuration that contains $S$. Therefore, the compatibility set is maximal.

We now show, by contradiction, that for every configuration $\mathcal{C}$ containing requests, even if $\mathcal{C} \rightarrow \mathcal{C}$ is the only
transition from $\mathfrak{S}$, there is a maximally compatible transition $\mathfrak{S} \rightarrow \mathfrak{S}'$. Assume there is no maximal transition from a configuration $\mathfrak{S}$. Then, there is a transition $\mathfrak{S} \rightarrow \mathfrak{S}'$ such that the maximum number of requests is not satisfied in $\mathfrak{S} \rightarrow \mathfrak{S}'$. The unsatisfied requests $Q = \{R_1, \ldots, R_N\}$ must be blocked in $\mathfrak{S}$ by elements of their C-sets, otherwise, there is a maximally compatible transition from $\mathfrak{S}$. The requests in the C-sets themselves must be blocked, otherwise, they can be satisfied in a transition $\mathfrak{S} \rightarrow \mathfrak{S}''$ and requests in $Q$ will be blocked by elements of their B-sets in $\mathfrak{S}''$, and $\mathfrak{S} \rightarrow \mathfrak{S}''$ is maximally compatible. Thus, requests in $Q$ are blocked by requests $Q'$ in their C-sets which are in turn blocked. Thus, the assumption that no request that is blocked blocks another is contradicted, and the production system is maximally compatible.

We now prove that if the production system is maximally compatible, then in every reachable configuration $\mathfrak{S}$ no request $R$ that is blocked blocks another request. Assume the opposite, that is, there is a request $R$ that is blocked and blocks another request in a reachable configuration $\mathfrak{S}$. As a consequence, there will be no transition $\mathfrak{S} \rightarrow \mathfrak{S}'$ such that $R$ will be satisfied. Thus, the requests being blocked by $R$ will not be satisfied while they can be without destroying the compatibility of the production system. Thus the production system is not maximally compatible, a contradiction.
The significance of the above lemma is reflected in the way the priorities of requests can be assigned so that the conditions for maximal compatibility are ensured. In algorithm COMPS, the condition in the above lemma is satisfied if the priorities are assigned such that:

1. Every request \( R \) that is blocked by an element of its B-set, has lower priority than requests that it can block, and

2. If a request \( R \) is an element of more than one incompatible set \( Q_0, \ldots, Q_{N-1} \), and it has the lowest priority in any one incompatible set \( Q_i \), then it also has the lowest priority in every incompatible set \( Q_j \), \( 0 \leq j \leq N-1 \).

Next, we investigate the classes of systems in which the priorities can be assigned as prescribed above.

**Lemma**

For a set of request \( Q = \{R_0, \ldots, R_{N-1}\}, N \geq 3 \), such that \( R_i \mod N \) and \( R_{(i+1)\mod N} \) belong to a minimal incompatible set \( Q_i \) where \( Q_j = Q_k \) if and only if \( k=j \), the priorities can be assigned as prescribed above if and only if \( N \) is even.
Proof:

We first show that if the priorities can be assigned as prescribed, then $N$ is even. Assume $N$ is odd. Let $R_0$ have higher priority than $R_1$ and $R_{N-1}$. Since $R_1$ has lower priority than $R_0$, then it should have lower priority than $R_2$. Continuing in this fashion, we see that the even-numbered requests have higher priorities than their adjacent odd-numbered requests. Thus $R_{N-2}$, which is odd-numbered, has lower priority than $R_{N-1}$, but $R_{N-1}$ has lower priority than $R_0$, thus $R_{N-1}$ violates the prescription of the priorities. Thus, $N$ must be even.

We now show that if $N$ is even, then the priorities can be assigned as prescribed. Let $R_0$ have higher priority than $R_1$ and $R_{N-1}$. Since $R_1$ has lower priority than $R_0$, then it should have lower priority than $R_2$. Continuing in this fashion, we see that the even-numbered requests have higher priorities than odd-numbered requests. Thus, $R_{N-2}$ which is even-numbered will have higher priority than $R_{N-1}$ and $R_{N-3}$; and $R_{N-1}$ has lower priority than $R_{N-2}$ and $R_0$, and no contradictions arise. Thus, the priorities can be assigned as prescribed if $N$ is even.
As a special case of the above lemmas, a compatible production system constructed by algorithm COMPS is maximally compatible when the minimal incompatible sets are disjoint and for each minimal incompatible set $Q = \{R_1, \ldots, R_N\}$, only one request has the lowest priority.

We now present a more general algorithm, MXCMPS, to construct a maximally compatible PS, see Figure 27.

The basic idea is to allow a request to be satisfied, not only when elements of its B-set and C-set are not in the system (in WM), but also when requests in its C-set can not be satisfied, consequently, permitting more concurrency. Thus, the conditions under which a request can be satisfied now include, besides the B-set conditions, elements of the C-set with the extra check that those requests in the C-set can not be satisfied. However, in order to check if those requests in the C-set can not be satisfied, we must check the conditions necessary to satisfy those requests. Since these again may involve requests, we may end up with a recursive problem, i.e., when a request reappears as an element of the C-set of another request.

To avoid this problem, we inject the requirement that the priorities of the requests be non-cyclic. We are thus guaranteed that a request does not reappear, and as a
consequence we do not have to continue checking for conditions since it is guaranteed by the above requirement that the condition under question holds, and there is no need to check its presence.

Algorithm MXCMPS utilizes a recursive procedure TREE, Figure 28, in order to generate the different conditions of the C-set for every request. Algorithm TREE terminates because of the non-cyclic priorities and the finite number of requests.

Initially algorithm TREE is called with a request $R_{i,j,k}$ as its root node. The nodes for which there is an arc from the root node represent the B-set and C-set conditions for $R_{i,j,k}$. The above procedure is repeated for every new node appended when two conditions apply:

1. No request appears more than once on any path from the root node, and
2. The B-set and C-set conditions for requests at any node are not empty.
procedure MXCMPS (S: sets of minimal incompatible accesses,
   PS: a maximally compatible production system)

/* Input: S. */
/* Output: PS. */

begin
   (1) Let the WM alphabet \( \Sigma \) contain the symbols
       \( I_k \), \( R_{i,j,k} \) and \( A_{i,j,k} \); and variables \( p_{i,j,k} \)
       for all relevant values of \( i,j,k \).
       Let the initial configuration \( \xi_0 \) be
       \( I_1 \ldots I_n \), \( n \) = the number of processes.

   (2) for every process \( k \) Let the requests made by \( k \) be
       \( R_{1_1,j_1,k} \ldots R_{i_m,j_m,k} \)
       for every request \( R_{i,j,k} \)
       Include the following productions in \( \mathcal{P} \),
       the set of productions

       \[
       (I_k)(\text{fail } R_{i_1,j_1,k}) \ldots (\text{fail } R_{i_m,j_m,k})
       \]

       (fail \( A_{i,j,k} \) \( \rightarrow \) (Assign \( p_{i,j,k} \))

       \[
       (I_k)(\text{fail } R_{i_1,j_1,k}) \ldots (\text{fail } R_{i_m,j_m,k})
       \]

       (fail \( A_{i,j,k} \) \( \rightarrow \) (Noop))

   endfor
   Include the above productions in \( \mathcal{R} \), the
   conflict resolution, as one conflict set.
   for every request \( R_{i,j,k} \)
   Let the root node of \( T \) be \( R_{i,j,k} \)
   \( V = R_{i,j,k} \)
   call TREE(S, T, V)

Figure 27
for every path from the root node of T to a pendant node (leaf)
Let the path be $V_0, V_1, \ldots, V_{k-1}$, where $V_0 = R_{i,j,k}$, i.e., the root node.
Construct the following condition
$$\text{fail (} V_1 \land \text{fail (} V_2 \land \text{fail (} V_3 \land \ldots \land \text{fail (} V_{i-2} \land \text{fail (} V_{i-1} \ldots)\ldots)))$$
where $V_i$ = the contents of node $V_i$, i.e., a conjunction of conditions.
Let $c_i$ be the condition, as constructed above, for the $i$th path.
endfor

Include the following productions in $P$.

$$((R_{i,j,k})c_1 \cdots c_m \rightarrow \begin{cases} \text{(Replace (} R_{i,j,k} : A_{i,j,k}) \\ \text{(Assign (} p_{i,j,k}) \end{cases})$$

$$((R_{i,j,k})c_1 \cdots c_m \rightarrow \text{(Noop)})$$
where $c_i$ = the condition constructed for the $i$th path as mentioned earlier, and $m$ = the number of paths.
Include the above two productions in $R$ as one conflict set.
endfor

for every access $A_{i,j,k}$

Include the following productions in $P$.

$$((A_{i,j,k})\text{fail } R_{i_1,j_1,k_1} \ldots \text{fail } R_{i_m,j_m,k_m} \rightarrow \begin{cases} \text{(Delete } A_{i,j,k}) \end{cases})$$

$$((A_{i,j,k})\text{fail } R_{i_1,j_1,k_1} \ldots \text{fail } R_{i_m,j_m,k_m} \rightarrow \text{(Noop)})$$
endfor

Include the above productions in $R$ as one conflict set.
endfor

Figure 27 continued
(3) Let $\beta$, the size of WM be $= (\text{the number of processes} + 2 \times \text{the number of requests})$
procedure TREE (S: minimal incompatible sets of accesses,  
T: a directed tree, 
V: a node of T) 

/* Input: S and V. 
/* Output: T. 

begin 
for every request $R_{i,j,k}$ in V 
    Let $S'$ be the minimal incompatible sets of which  
    $R_{i,j,k}$ is an element. 
    for every $s'$ in $S'$ 
        if a request in $s'$ already appears in a node  
           on the path from the root of $T$ to and in-  
           cluding node $V$ 
            then delete $s'$ from $S'$. 
        endif 
    endfor 
    if $S'$ is empty 
        then return 
    endif 
    Construct the B-set $(R_{i,j,k})$ and C-set $(R_{i,j,k})$  
    using algorithms BSETS and CSETS with $S'$ as the  
    minimal incompatible sets. 
endfor 

Let the number of requests in V be $n$. 
Let $c_{1i}, \ldots, c_{li}$ be the B-set and C-set conditions  
for the $i$th request. 
for every expression $V' = c_{k1} \wedge c_{k2} \wedge \cdots \wedge c_{kn}$  
    where $c_{ki}$, $1 \leq i \leq n$ is a C-set or B-set con-  
    dition for the $i$th request in V. 
    Include an arc from $V$ to a new node $V'$. 
    call TREE (S, T, V') 
endfor 
endbegin 

Figure 28
Example 11

We will construct a maximally compatible PS for the system of requests presented in example 9 using algorithm MXCMPS. There are three requests $R_0$, $R_1$, and $R_2$ made by three processes. The minimal incompatible sets are $\{A_0, A_1\}$, $\{A_1, A_2\}$.

1. Let $\Sigma = \{I_0, I_1, I_2, R_0, R_1, R_2, A_0, A_1, A_2, p_0, p_1, p_2\}$, where $I_i$, $R_i$, and $A_i$, $0 \leq i \leq 2$ are symbols, $p_i$, $0 \leq i \leq 2$ are variables.

2. The initial configuration $= \chi_0 = I_0 I_1 I_2$.

3. $P$ includes the following productions:

   $P_{ia}$: $((I_i)(\text{fail } R_i)(\text{fail } A_i) \rightarrow (\text{Insert } R_i)(\text{Assign } p_i))$

   $P_{ib}$: $((I_i)(\text{fail } R_i)(\text{fail } A_i) \rightarrow (\text{Noop}))$

   $R$, the conflict resolution, contains $\{P_{ia}, P_{ib}\}$, $0 \leq i \leq 2$.

4. Executing algorithm TREE for $R_0$ we have the following tree.
We thus have the following productions:

\[ P_{3a} : (R_0)(\text{fail } (R_1 \land (p_1 > p_0)) \land \text{fail } (R_2 \land (p_2 > p_1))) \]
\[ \quad \text{(fail } (R_1 \land (p_1 > p_0)) \land (\text{fail } A_2)))(\text{fail } A_1) \]
\[ \quad \quad \quad \rightarrow (\text{Replace } (R_0, A_0))(\text{Assign } p_0) \]

\[ P_{3b} : (R_0)(\text{fail } (R_1 \land (p_1 > p_0)) \land \text{fail } (R_2 \land (p_2 > p_1))) \]
\[ \quad \text{(fail } (R_1 \land (p_1 > p_0)) \land (\text{fail } A_2)))(\text{fail } A_1) \]
\[ \quad \quad \quad \rightarrow (\text{Noop}) \]

\( R \) contains \( \{P_{3a}, P_{3b}\} \).

For \( R_1 \) we have the following tree,
and the following productions:

\[
P_{4a}: \quad (R_1)(\text{fail } (R_0 \land (p_0 > p_1)))(\text{fail } (R_2 \land (p_2 > p_1)))(\text{fail } A_0)(\text{fail } A_2) \\
\quad \rightarrow (\text{Replace } (R_1, A_1))(\text{Assign } p_1)
\]

\[
P_{4b}: \quad (R_1)(\text{fail } (R_0 \land (p_0 > p_1)))(\text{fail } (R_2 \land (p_2 > p_1)))(\text{fail } A_0)(\text{fail } A_2) \rightarrow \text{(Noop)}
\]

\(R\) contains \(\{P_{4a}, P_{4b}\}\).

For \(R_2\), we have the following tree,
**P**$_{5a}$: $((R_2)(\text{fail } (R_1 \land (p_1 > p_2) \land \text{fail } (R_0 \land (p_0 > p_1))))
(fail (R_1 \land (p_1 > p_2) \land \text{fail } A_0))($ fail $A_1$)
$\rightarrow$ (Replace($R_2$, $A_2$))(Assign $p_2$))

**P**$_{5b}$: $((R_2)(\text{fail } (R_1 \land (p_1 > p_2) \land \text{fail } (R_0 \land (p_0 > p_1))))
(fail (R_1 \land (p_1 > p_2) \land \text{fail } A_0))($ fail $A_1$)
$\rightarrow$ (Noop))

$\mathcal{R}$ contains \{P$_{5a}$, P$_{5b}$\}.

(5) Add the following productions to $\mathcal{P}$:

P$_{(6+i)a}$: ((A$_i$) $\rightarrow$ (Delete A$_i$))

P$_{(6+i)b}$: ((A$_i$) $\rightarrow$ (Noop))

$\mathcal{R}$ contains \{P$_{(6+i)a}$, P$_{(6+i)b}$\}, $0 \leq i \leq 2$.

Note that it is assumed that the priorities of the requests are non-cyclic as the following lemma states.
Lemma

Algorithm MXCMPS constructs a maximally compatible production system if and only if the priorities of requests are non-cyclic.

Proof

We will first prove that the production system is maximally compatible. We will show that the production system is compatible, the compatibility set is maximal, and for every configuration $\mathcal{C}$ containing requests there is a maximally compatible transition from $\mathcal{C}$.

Assume the production system is incompatible. Then, there is a reachable configuration $\mathcal{C}$ containing a minimal incompatible set of accesses $\{A_0, \ldots, A_{k-1}\}$. None of the requests $\{R_0, \ldots, R_{k-1}\}$ could have been blocked by the B-set condition, otherwise, at least one request of $\{R_0, \ldots, R_{k-1}\}$ would not have been satisfied and the incompatible set of accesses would not have occurred. If the C-set conditions for all contending requests in the set $\{R_0, \ldots, R_{k-1}\}$ were satisfied, that would imply that for every contending request, at least one other contending request was blocked unless the priorities are cyclic. If the priorities are cyclic then no request is blocked. However, it is assumed
that the priorities are non-cyclic, thus a contradiction, and the production system is compatible.

We now show that the compatibility set is maximal. Let \( S = \{A_0, \ldots, A_{N-1}\} \) be a compatible set of accesses. Consider a configuration \( \mathcal{C} \) only containing the corresponding requests of \( S \). \( \mathcal{C} \) is reachable from the initial configuration. Since the requests are compatible, the conditions for processing the requests are satisfied, and there is a transition \( \mathcal{C} \rightarrow \mathcal{C}' \) such that \( \mathcal{C}' \) contains \( S \). Thus, the compatibility set is maximal.

We will now show that for every reachable configuration \( \mathcal{C} \), there is a transition \( \mathcal{C} \rightarrow \mathcal{C}' \) such that the maximum number of requests is satisfied without destroying the compatibility of the PS. Since we have already shown this property for configurations in which the requests form a compatible set, we will only consider configurations which contain incompatible sets of requests. Requests that are blocked by elements of their B-sets cannot be satisfied. In the algorithm MXCMPS, this is reflected in the request processing productions by the conditions developed in algorithm TREE. For those requests that are blocked by conditions involving elements of their C-sets, assume there is no transition such that, without destroying the compatibility of the PS, the maximum number of those requests is
satisfied. Consider a request \( R \) that has not been satisfied while it could have been. One of the conditions involving one of the elements of its \( C \)-set must have failed, i.e., the condition was not true. Such a condition is not true when the requests in the element of the \( C \)-set are present and have priorities higher than that of \( R \). If the requests have higher priority than \( R \), then they can be satisfied unless they are also blocked, in which case, the conditions in the processing productions for \( R \) guarantee that \( R \) can be satisfied, thus a contradiction. If the requests can be satisfied then in the resulting configuration, \( R \) is blocked by an element of its \( B \)-set, namely, those satisfied requests, thus a contradiction.

We now prove if algorithm MXCMPS is maximally compatible then the priorities of requests are non-cyclic. Consider a configuration \( \mathcal{C} \) containing a set of requests \( Q = \{ R_1, \ldots, R_N \} \) such that satisfying the requests in \( Q \) would result in an incompatible set of accesses. Let requests in \( Q \) have cyclic priorities. Thus, for every request in \( Q \) there is another request with lower priority. As a consequence, all requests in \( Q \) can be satisfied and the production system is incompatible; therefore it can not be maximally compatible.
2. Implementation Derivation

In this section, we show how to construct an implementation of a policy from its specification using PS. Algorithm IMPLEMENT, shown in Figure 28, produces the synchronization procedures by translating the production rules into procedural language constructs that will comprise the synchronization procedures.

procedure IMPLEMENT(PS: a production system, CODE: a set of procedures to be executed for synchronizing processes)

/* Input: PS. */
/* Output: CODE. */

begin
(1) Define a mapping \( g: \Sigma \rightarrow \Theta \), a new alphabet used to define variables in the synchronization procedures, where
   a. \( \Sigma \cap \emptyset = \emptyset \), the null set.
   b. \( \Theta \) contains variables of different types, e.g., boolean, integer, ... etc.
   c. For a symbol \( x \) in \( \Sigma \), \( g(x) = x' \) where \( x' \) is a boolean variable in \( \emptyset \). For a variable \( y \) in \( \Sigma \), \( g(x) = y' \) where \( y' \) is a variable in \( \emptyset \) of the same type as \( y \).
   d. \( g \) is a one-to-one mapping (a bijection).

(2) Disregard all no-action productions. Also, disregard all idling symbols (I's) in the other productions.

Figure 29
(3) Consider the initial configuration $\mathcal{X}_0$.

\begin{verbatim}
for every symbol $x$ except $I_i$'s in $\Sigma$
   Let $x' = g(x)$
   if (x appears in the initial configuration)
      then
         Let $x'$ be initially true or construct
         the assignment statement $x' = true$
   else
      Let $x'$ be initially false or construct
      the assignment statement $x' = false$
endif
endfor
for every variable $y$ in $\Sigma$
   Let $y' = g(y)$
   Let $v_0$ be the initial value of $y$ in the initial configuration.
   Let $y' = v_0$ initially, or construct the assignment statement $y' = v_0$.
endfor
Group the assignment statements constructed above for variables and symbols to form the "initialization code".
\end{verbatim}

(4) for every request $R_{i,j,k}$ (or set of requests $Q$)

\begin{verbatim}
Let the production which allows
$R_{i,j,k}$ (or $Q$) to occur be
$P_i : \ (C_1 C_2 \ldots C_n \rightarrow A_1 A_2 \ldots A_m)$
construct the following "Make-Request code":
begin
   Wait ($g(C_1) \land g(C_2) \land \ldots \land g(C_n)$)
   $g(A_1)$
   $g(A_2)$
   \vdots
   $g(A_m)$
   Check($A_1$)  /* Check is used to
   Check($A_1$)  /* wake up any pro-
   \vdots
   Check($A_m$)  /* cesses waiting on
   Check($A_m$)  /* a condition.
endbegin
\end{verbatim}

Figure 29 continued
In the above procedure:

A. \( g(C_i) \) is the same as \( C_i \), \( 1 \leq i \leq n \) with the following changes:
   (a) Every symbol in \( C_i \) is replaced by the relational expression \( g(x) = \text{true} \).
   (b) "fail" is replaced by the logical operator "not" (\( \neg \)).
   (c) Every variable \( y \) in \( C_i \) is replaced by \( g(y) \).

B. \( g(A_i) \) is the same as \( A_i \), \( 1 \leq i \leq m \) with the following modifications:
   (a) The insertion of a symbol \( x \) is replaced by the assignment: \( g(x) = \text{true} \).
   (b) The deletion of a symbol \( x \) is replaced by \( g(x) = \text{false} \).
   (c) Every variable \( y \) is replaced by \( g(y) \).

C. "Wait" can be implemented in many ways:
   (a) busy wait: in this case, the conditions are constantly evaluated until they are true, then the process may proceed to the next statement. For this implementation, \( \text{Wait}(x) \) can be replaced as:

   ```
   begin
   repeat
   until (x = \text{true})
   endbegin
   ```

Figure 29 continued
(b) **condition-queue wait:** In this case, if the condition holds, the process proceeds, otherwise it is put on a queue corresponding to the condition. The process is removed from the queue when the condition evaluates to true. The condition is reevaluated whenever one of its terms changes value. This is performed in the Check statement as will be shortly discussed. In this case, the Wait statement can be interpreted as:

```plaintext
begin
    if (x = false)
    then Put (xqueue, executing process)
endif
endbegin
```

(c) This third choice is the same as the second one (b) except, when a term changes value, the processes on the queues of the conditions of which the term is a part are removed to reevaluate the conditions. This is a form of busy-waiting known as a spin lock. The wait statement in this case is the same as in case b.

Note that these are not the only possible implementations of the Wait statement.

D. The implementation of the Check statement is tied to that of the Wait as mentioned earlier. Thus,

(a) For the busy-wait implementation, the Check statement is empty, since the process itself keeps checking the conditions.

Figure 29 continued
(b) For the condition-queue implementation, Check(x) is as follows:

\[
\begin{align*}
\text{begin} \\
\quad \text{for } \text{every condition } z \\
\quad \text{that contains } x \\
\quad \text{if } (\text{condition} = \text{true}) \\
\quad \quad \text{wake-up (zqueue)} \\
\quad \text{endif} \\
\end{align*}
\]

\[
\text{endbegin}
\]

The processes awakened proceed to execute the statement following the Wait statement.

(c) For the distributed-queue implementation the Check statement is as follows:

\[
\begin{align*}
\text{begin} \\
\quad \text{for } \text{every condition } z \\
\quad \text{that contains } x \\
\quad \quad \text{wake-up (zqueue)} \\
\quad \text{endfor} \\
\end{align*}
\]

\[
\text{endbegin}
\]

In this case, the processes are awakened to reevaluate the conditions they are waiting on.

(5) To determine the sequence of procedures that a process must execute in order to access a resource, the sequence can be broken down in the following fashion:

(a) Code invocations leading to placement of the request (s).

(b) Code invocations leading to permission of access.

(c) Make access (critical section)

Figure 29 continued
(d) Code invocation leading to termination of access.

For step (a), the code constructed in step 4 for the production allowing a request or a set of requests to occur is invoked. For step (b), the code constructed in step 4 for the production allowing a request or a set of requests to be satisfied is invoked. For step (d), the code constructed in step 4 for the production allowing normal termination of the access or set of accesses is invoked.

(6) In this step, we consider productions other than those which allow requests to occur and to be satisfied, and those which allow accesses to normally terminate. The preemption productions derived earlier to produce a live production system fall into this category. The code corresponding to these productions is invoked whenever the conditions for a request to be satisfied do not hold. The effect of executing the preemption code is to preempt any blocking access, thus changing the conditions governing the processing of other requests. The change in conditions is transmitted through busy wait testing or the execution of the "Check" code, after the conditions are changed, to wake up any waiting requests if any.

"endbegin"

Figure 29 continued
Consider the production system developed for the dining philosophers problem in chapter III, example 4. We will apply the implementation procedure IMPLEMENT to this production system.

Let \( g(R_{i,j}) = \text{REQUEST}(I,J) \),
\( g(A_{i,j}) = \text{ACCESS}(I,J) \), and
\( g(p_{i,j}) = \text{PRIORITY}(I,J) \)
where REQUEST and ACCESS are Boolean arrays with two dimensions, the first refers to the fork being requested, and the second refers to the philosopher making the request. The values of I and J range from zero to \((N-1)\), N being the number of forks or philosophers.

On examining the initial configuration
\( \chi_0 = I_0 I_1 \ldots I_{N-1} \), we see that \( \chi_0 \) does not contain any symbols except \( I_i \)'s which we disregard. \( \chi_0 \) also does not initialize any variables. Thus, we can assume that the Boolean arrays REQUEST and ACCESS are initially false, or develop the following initialization code.

\[
\begin{align*}
\text{begin} & \\
\text{for} & \quad I : = 0 \text{ To } N-1 \\
\text{REQUEST}(I-1,I) & = \text{REQUEST}(I,I) = \text{false} \\
\text{ACCESS} (I-1,I) & = \text{ACCESS} (I,I) = \text{false} \\
\text{endfor} & \\
\text{endbegin} & 
\end{align*}
\]
It is assumed that arithmetic on the subscripts is performed mod N.

We will derive procedures from the productions constructed for the problem. The following productions allow requests to occur.

\[ P_{ia}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i}) \rightarrow (\text{Insert } R_{i,i})(\text{Assign } p_{i,i})) \]

\[ P_{ib}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i,i}) \rightarrow (\text{Noop})) \]

\[ P_{ic}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Insert } R_{i-1,i})(\text{Assign } p_{i-1,i})) \]

\[ P_{id}: \quad ((I_i)(\text{fail } R_{i,i})(\text{fail } R_{i-1,i})(\text{fail } A_{i-1,i}) \rightarrow (\text{Noop})) \]

Productions \( P_{ib} \) and \( P_{id} \), \( 0 \leq i \leq N-1 \), will be disregarded since their actions are null. For production \( P_{ia} \) we have the following procedure:

\begin{verbatim}
begin
Wait ( \neg \text{REQUEST}(I,I) \wedge \neg \text{REQUEST}(I-1,I) \\
\wedge \neg \text{ACCESS}(I,I))
Assign-Priority(PRIORITY(I,I))
\text{REQUEST}(I,I) = \text{true}
\text{Check}(\text{REQUEST}(I,I))
end
\end{verbatim}
Similarly, the code for \(P_{ic}\) is as follows:

\[
\text{begin} \\
\quad \text{Wait (} \neg \text{REQUEST}(I,I) \land \neg \text{REQUEST}(I-1,I) \land \neg \text{ACCESS}(I-1,I)\) \text{)} \\
\quad \text{Assign-Priority(PRIORITY}(I-1,I)\)) \\
\quad \text{REQUEST}(I-1,I) = \text{true} \\
\quad \text{Check(REQUEST}(I-1,I)) \\
\text{endbegin}
\]

The productions which process requests \(R_{i,i}, 0 \leq i \leq N-1\) are

\[
P_{(N+i)a}: ((R_{i,i})(\text{fail } A_{i,i+1}) \\
\quad \text{(fail } (R_{i,i+1} \land (p_{i,i+1} > p_{i,i}))) \\
\quad \text{→ (Replace } (R_{i,i}, A_{i,i}))(\text{Assign } p_{i,i}))
\]

\[
P_{(N+i)b}: ((R_{i,i})(\text{fail } A_{i,i+1}) \\
\quad \text{(fail } (R_{i,i+1} \land (p_{i,i+1} > p_{i,i}))) \\
\quad \text{→ (Noop)})
\]

We ignore the "Noop" production \(P_{(N+i)b}\). For \(P_{(N+i)a}\), the corresponding code is

\[
\text{begin} \\
\quad \text{Wait (REQUEST}(I,I) \land \neg \text{ACCESS}(I,I+1) \\
\quad \land (\neg (\text{REQUEST}(I,I+1) \land \\
\quad \quad \text{(PRIORITY}(I,I+1) > \text{PRIORITY}(I,I))))\) \text{)} \\
\quad \text{ACCESS}(I,I) = \text{true} \\
\quad \text{REQUEST}(I,I) = \text{false} \\
\quad \text{Check(ACCESS}(I,I)) \\
\quad \text{Check(REQUEST}(I,I)) \\
\text{endbegin}
\]
The productions which process requests $R_{i-1,i}$, $0 \leq i \leq N-1$ are:

$P_{(2N+i)a}$  
\[ (R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \rightarrow \text{(fail } R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i})) \]
\[ \rightarrow \text{(Replace } (R_{i-1,i}, A_{i-1,i})) \]
\[ \rightarrow \text{(Assign } p_{i-1,i}) \]

$P_{(2N+i)b}$  
\[ (R_{i-1,i})(\text{fail } A_{i-1,i-1}) \]
\[ \rightarrow \text{(fail } R_{i-1,i-1} \land (p_{i-1,i-1} > p_{i-1,i})) \]
\[ \rightarrow \text{(Noop}) \]

We ignore $P_{(2N+i)b}$, and provide the code for $P_{(2N+i)b}$ as follows:

begin
Wait (REQUEST(I-1,I) \land \neg ACCESS(I-1,I-1)) \land
\neg (REQUEST(I-1,I-1) \land
(PRIORITY(I-1,I-1) > PRIORITY(I-1,I)))
ACCESS(I-1,I) = true
REQUEST(I-1,I) = false
Check(ACCESS(I-1,I))
Check(REQUEST(I-1,I))
end

The productions which process accesses are:

$P_{(3N+i)a}$  
\[ (A_{i,i})(\text{fail } R_{i-1,i}) \rightarrow \text{ (Delete } A_{i,i}) \]

$P_{(3N+i)b}$  
\[ (A_{i,i})(\text{fail } R_{i-1,i}) \rightarrow \text{ (Noop}) \]

$P_{(3N+i)c}$  
\[ (A_{i-1,i})(\text{fail } R_{i,i}) \rightarrow \text{ (Delete } A_{i-1,i}) \]

$P_{(3N+i)d}$  
\[ (A_{i-1,i})(\text{fail } R_{i,i}) \rightarrow \text{ (Noop}) \]
We disregard productions $P_{(3N+i)b}$ and $P_{(3N+i)d}$. The code for $P_{(3N+i)a}$ is as follows:

```
begin
  Wait (ACCESS(I,I) \land \neg REQUEST(I-1,I))
  ACCESS(I,I) = false
  Check(ACCESS(I,I))
endbegin
```

Similarly, the code for $P_{(3N+i)c}$ is:

```
begin
  Wait (ACCESS(I-1,I) \land \neg REQUEST(I,I))
  ACCESS(I-1,I) = false
  Check(ACCESS(I-1,I))
endbegin
```
V Conclusion

1. Production Systems and the Existing Models

The significance of parallel processing makes its representation within a formal model an essential step in the study, analysis and design of systems with such characteristics. In the first chapter of this thesis we have studied some existing models which included Petri nets, computation graphs and parallel program schemata besides others. The models were individually evaluated with respect to the extent to which they fulfilled the purposes and objectives of a formal model of parallel processing. In this section, we sum up our evaluation of those models. In the evaluation process, we contrast the models with production systems. We base our discrimination evaluation on three criteria.

The first criterion of our evaluation of the models is the ease of use of a model. The simpler the structure and representation of operations of the model, the easier it is to use and understand.

Some of the models considered in the first chapter of this thesis are more control oriented than data oriented. For example, Petri nets represent aspects of control with data leftout altogether, while parallel program schemata
emphasize control aspects through the explicit specification of states. As a consequence of this heavy emphasis on control aspects, it is difficult to derive a model since the flow of control must be monitored and explicitly dealt with. This problem is even more magnified when the number of places and transitions in Petri nets and the number of states in parallel program schemata is large. In Petri nets and computation graphs, the state of a system is not explicitly specified, however, control is defined through the explicit interconnections between places in Petri nets, and between operation nodes in computation graphs. These interconnections may obscure the basic concepts involved in the flow of control. In parallel program schemata, the states are explicitly specified and the explicit specification of control flow between these states can be a cumbersome task. However, the concept of a state abstracts and hides information, and consequently adds a certain degree of structure to the model.

On the other hand, production systems are data oriented. The flow of control is implicitly defined. Production systems restrict the interactions between rules, and as a consequence the model is strongly modular. This inherent modularity and the compactness of the representation of production system models make production systems extremely easy to use as a specification language. However, the
lack of structure in the control flow in production systems makes a model difficult to understand. However, this lack of structure in the control flow is lessened by allowing the use of variables and by allowing a certain degree of flexibility in the specification of the actions of the production system.

The second criterion is the modeling and decision powers of the models. As mentioned earlier these two factors work at cross purposes.

Petri nets have better modeling power than finite state machines and limited modeling power relative to Turing machines. It is difficult to model priority systems by Petri nets. The modeling power of computation graphs is restricted by the fixed queue discipline, the lack of a facility for data dependent conditional transfers, and the lack of having a common memory for data. The control in both Petri nets and computation graphs corresponds to an AND-input AND-output logic. The modeling power of parallel program schemata is restricted by the rule that performances of an operation terminate in the same order they are initiated. In general, these models may not be powerful enough to represent all classes of parallel processing systems.
In the second chapter, we have demonstrated the ability of production systems to simulate two types of machines; namely, finite state machines and Turing machines. In the third chapter, we have shown that Kosaraju's coordination problem, which is not representable within Petri nets, can be modelled by production systems with no difficulty or obscurity.

The decision power of any model is determined by the solvability of the problems within the model and the complexity of the algorithms used for solving the problems.

In the first chapter we have mentioned that the subset and equality problems with Petri nets are not solvable; and that some analytic questions about Petri nets are difficult. Nets with conflicts are difficult to analyze. It may be very difficult to prove liveness of a Petri net. Solution techniques developed for Petri nets either are of high complexity or lack structure and are not mechanizable. The decision power of computation graphs is influenced by its severely restricted modeling power. Methods of determining whether a computation terminates and of finding the number of performances of each computation have been developed [80]. Results involving the queue length for every branch have also been established. Whereas computation graphs model a restricted class of problems, parallel program
schemata model a larger class of problems. However, some problems of parallel program schemata are undecidable. For instance, the equivalence problem for some classes of parallel program schemata is undecidable [81]. Different classes of this model have been analyzed with respect to certain properties. The algorithms for the known solvable problems within this model can be unfeasible.

Much of the work required to determine the decision power of production system is yet to be done. In this research, we have shown that production systems with finite working memory are "equivalent" to nondeterministic finite state machines. Thus, the decision power of such production systems is as great as that of finite state machines. Furthermore, we have provided systematic analysis and design procedures for compatible, live and fair production systems. However, the analysis algorithms for some of those properties are of complexity comparable to that of the analysis techniques of Petri nets, and the other models.

The third criterion of the evaluation is the useability of a model as a specification language and as a design tool for policies and their implementations.

Petri nets can not yet be used as a specification language or a design tool. Little attention has been given to
developing modeling techniques specifically for Petri nets. Although computation graphs may be used as a programming language, their use as a specification language and a design tool is hindered by the fact that the operations allowed within the model can not be specified yet. Parallel program schemata are uninterpreted, i.e., certain details pertinent to the operation of programs are left unspecified. Furthermore, the explicit specification of states within the control element of the model makes it too abstract to use as a programming language and as a design tool.

In contrast, we have demonstrated in this thesis the suitability of production systems as a design tool. We were able to systematically use production system as a specification language for synchronization policies. Furthermore, we were able to systematically derive different implementations from the production system specifications of synchronization policies. This ease with which production systems can be used for the purposes mentioned above is a direct consequence of their modularity.

However, in specifying a synchronization policy in terms of a production system, there are certain cautions the designer should be aware of. The designer has to make sure that no conflicts occur with respect to modifying a certain item within working memory. Otherwise, the order of those
modifications in any one cycle, may affect the outcome of the actions performed.

In summary, production systems have higher modelling power than the existing models considered in this thesis. As a consequence its decision power may be limited. However, the decision power of certain classes of production systems, e.g., those with finite working memory, is comparable to that of the other models with high decision power. Production systems are more suitable than the existing models as a specification language and as a design tool for deriving implementations of the policies being specified.
2. **Compatibility, Maximal Compatibility, Liveness, and Fairness**

In this thesis we have developed analysis and design procedures for compatibility, maximal compatibility, liveness and fairness. Upon developing the analysis procedures, we have effectively demonstrated the computability of compatibility, maximal compatibility, liveness and fairness. However, the complexity of the analysis algorithms, in general, is proportional to the number of configurations in the production system being analyzed. Although the number of configurations is combinatorial in terms of the number of requests, the number of reachable configurations is limited by the minimal incompatible sets. For instance, consider a set of four requests made by four different processes. Then there are $((3)^4) = 81$ possible configurations. Let every pair of requests be a minimal incompatible set. If the production system is compatible, then there are at most $(2)^4 + 4(2)^3 = 48$ reachable configurations.

Furthermore, since in most problems minimal incompatible sets consist of two requests, the number of elements in the C-set for a particular request is bounded by the number of minimal incompatible sets of which the request is an element.
The design algorithms developed in this thesis reflect the nature of the properties to be achieved. For instance, algorithms COMPS, which was developed to produce a compatible production system, does not place any restrictions on the manner requests are made and accesses are terminated. However, algorithms LVCMPS and PLVCMPS place restrictions on the occurrence of requests and termination of accesses. In algorithm LVCMPS a process can only make a set of requests, i.e., those at the same level, simultaneously. Similarly, a process can only simultaneously terminate accesses at the same level. In algorithm PLVCMPS accesses are forced to terminate, thus restricting the degree of freedom with which a process can terminate accesses. Algorithm MXCMPS requires more complex conditions on the processing productions for requests than algorithm COMPS. Algorithm MXCMPS, however, permits more concurrency than algorithm COMPS.

Some of the trade-offs mentioned above can be formally derived within the properties of synchronization policies that we have studied. For example, the compatibility set of a production system constructed by algorithm COMPS is maximal, while that of a system constructed by algorithms LVCMPS or PLVCMPS is not. Consider a set of requests of a process $Q = \{R_1, \ldots, R_n\}$. Let $Q$ be at one level in algorithm LVCMPS. Then, clearly, the compatibility set does
not include the sets \( \{A_i\} \), \( 1 \leq i \leq n \). Thus, in general, algorithm LVCMPS does not construct a maximally compatible production system. In particular, algorithm LVCMPS constructs a production system which is not maximally compatible if there is more than one request of any process at any one level.

In algorithms LVCMPS and PLVCMPS, the conditions in the processing productions for requests are the B-set and C-set conditions. The C-set conditions can be extended to those C-set conditions in algorithm MXCMPS, thus increasing the compatibility of the production system, however the production system may not be maximally compatible yet. However, assuming all requests of every process are compatible, consider a new algorithm MXLVPS which is the same as algorithm LVCMPS with the modifications suggested above. Then the production system is maximally compatible if the following two conditions hold:

1. For each process \( k \), there is at most one request \( R_{i,j,k} \) at any one level.
2. For every minimal incompatible set of requests there is only one request with the lowest priority.

To show maximal compatibility under the two conditions mentioned above, assume that the production system is not
maximally compatible. Then, the compatibility set for the production system is not maximal or there is a configuration \( \mathcal{C} \) containing requests such that there is no transition from \( \mathcal{C} \) in which the maximum number of requests are satisfied without destroying the compatibility of the production system.

Since a process \( k \) has at most one request per level, and since the production which introduces requests of process \( k \) into \( WM \) only places an order in which the requests of process \( k \) are made, then any subset of compatible accesses made by process \( k \) is in the compatibility set of the production system. It is clear that any compatible set of accesses \( S = \{ A_1, \ldots, A_n \} \) where accesses \( A_i, 1 \leq i \leq n \) are made by different processes can occur in the production system. Since \( S \) is compatible, then the processing productions for requests guarantee that a configuration containing the set of accesses \( S \) is reachable. Thus, the compatibility set is maximal.

To show that for every configuration containing requests, there is a transition in which the maximum number of requests are satisfied without destroying the compatibility of the production system, the same proof used for algorithm MXCMPS applies here.
We now characterize the production systems in which requests can be partitioned as required by algorithms LVCMPS and MXLVPS such that for each process, at most one request is at any one level.

Lemma

A set of requests can be partitioned as prescribed in algorithms LVCMPS and MXLVPS such that each level contains at most one request per process if and only if for every process \( k \), there is no sets of minimal incompatible requests \( Q_1, \ldots, Q_n, n \geq 1 \), such that \( Q_i \cap Q_{i+1} \), \( 1 \leq i \leq n-1 \), is not empty and \( Q_1 \cup \ldots \cup Q_n \) contains at least two distinct requests \( R_{i_1, j_1, k} \) and \( R_{i_2, j_2, k'} \).

Proof:

First we prove the "if" clause by construction. Let the minimal incompatible sets of requests be \( Q_0, \ldots, Q_m \). Starting with \( Q'_0 = Q_0 \), let \( Q'_0 = Q'_0 \cup Q_j \) such that \( Q_j \cap Q'_0 \) is not empty. Continue in this fashion until no more \( Q_i \) can be added to \( Q'_0 \). Repeat the procedure for the remaining minimal incompatible sets that are not added to a \( Q'_i \), yet. The process terminates since there is a finite number of \( Q_i \)'s. Consider each \( Q_i \) constructed.
Each $Q'_i = Q'_0 \cup \ldots \cup Q'_j$ does not contain any sets $Q_1, \ldots, Q_n$ such that $Q_i \cap Q_{i+1}$ is not empty and $Q_1 \cup \ldots \cup Q_n$ contains at least two distinct requests $R_{i_1,j_1,k}$ and $R_{i_2,j_2,k}$ for every process $k$ by assumption. Thus $Q_i$'s form the desired partition.

Now, we prove the "only if" clause by contradiction. Assume there is a set of minimal incompatible requests $Q_1, \ldots, Q_n$, $n \geq 1$, such that $Q_i \cap Q_{i+1}$, $1 \leq i \leq n-1$ is not empty and $Q_1 \cup \ldots \cup Q_n$ contains at least two requests $R_{i_1,j_1,k}$ and $R_{i_2,j_2,k}$ for some process $k$. Then, the construction procedure will place $Q_1 \cup \ldots \cup Q_n$ at the same level, and the partition will be such that $R_{i_1,j_1,k}$ and $R_{i_2,j_2,k}$ are at the same level, a contradiction.

As an example of this characterization lemma, consider the production system constructed by algorithm LVCMPS for the dining philosophers problem. All minimal incompatible sets of requests are pairwise disjoint, thus there is no minimal incompatible sets with the properties mentioned in the lemma. Consequently, the requests can be partitioned in the manner prescribed by the lemma. We have already demonstrated this possibility when we constructed the partition in the example.
While some of the trade-offs involved in algorithm LVCMPS can be represented within the properties we consider in this thesis, the trade-offs in algorithm PLVCMPS are not. The trade-offs in algorithm PLVCMPS can be formally uncovered when performance issues such as throughput, efficiency...etc. are investigated. In this thesis we do not address these issues.

We now turn to discuss the relationship between maximal compatibility and fairness in view of the algorithms we have developed. In the lemma we state below, we will use one index on requests and accesses to simplify the expressions involved without loss of generality.

**Lemma**

If algorithm COMPS constructs a fair production system PS then PS is not maximally compatible if there are two minimal incompatible sets of requests $Q_1$ and $Q_2$ such that $|Q_1 U Q_2| > 3$, $Q_1 \cap Q_2$ is not empty and there is a request $R_k$ in $Q_1 \cap Q_2$ such that $(Q_1 U Q_2) - \{R_k\}$ is compatible.

**Proof:**

Let $S_1$ and $S_2$ be the accesses corresponding to $Q_1$ and $Q_2$ respectively. Let $\mathcal{C}$ be a configuration only containing
the requests in $Q_1 \cup Q_2$ and in which $R_k$ is blocked by requests in $(Q_1 \cup Q_2) - \{R_k\}$. Such a configuration is reachable since requests of a particular process occur independently of requests of other processes and since PS is fair, otherwise $R_k$ is always satisfied and another request can be starved. If not all requests in $(Q_1 \cup Q_2) - \{R_k\}$ can be satisfied, then clearly PS is not maximally compatible since $(Q_1 \cup Q_2 - \{R_k\})$ is compatible. If all requests in $(Q_1 \cup Q_2) - \{R_k\}$ can be satisfied, then let those accesses in $(S_1 - (S_1 \cap S_2))$ terminate and those in $(S_2 - (S_1 \cap S_2))$ continue in progress. Since PS is fair, then requests in $(Q_1 - (Q_1 \cap Q_2))$ can only occur with higher priority than $R_k$ a finite number of times. Thus, there is a configuration which contains requests in $(Q_1 - (Q_1 \cap Q_2))$ with lower priority than $R_k$, $R_k$ and accesses in $(S_2 - \{A_k\})$. Clearly at least one request in $(Q_1 - (Q_1 \cap Q_2))$ can not be satisfied. Thus, the PS is not maximally compatible.
Lemma

If algorithm MXCMPS constructs a maximally compatible production system PS, and there is two minimal incompatible sets of requests $Q_1$ and $Q_2$ such that $|Q_1 \cup Q_2| \geq 3$, $Q_1 \cap Q_2$ is not empty and there is a request $R_k$ in $Q_1 \cap Q_2$ such that $(Q_1 \cup Q_2) - \{R_k\}$ is compatible, then PS is not fair.

Proof: (By contradiction)

Let $S_1$ and $S_2$ be the accesses corresponding to $Q_1$ and $Q_2$ respectively. Assume PS is fair. Consider a configuration $\emptyset$ only containing the requests of $(Q_1 \cup Q_2)$ in which $R_k$ is blocked by requests in $(Q_1 \cup Q_2) - \{R_k\}$. Such a configuration is reachable since requests of a particular process occur independently of requests of other processes and since PS is fair, otherwise, $R_k$ is always satisfied and another request can be starved as a result.

The requests blocking $R_k$ can be satisfied since $\emptyset$ only contains requests in $Q_1 \cup Q_2$, $(Q_1 \cup Q_2) - \{R_k\}$ is compatible and algorithm MXCMPS constructs a maximally compatible PS. Let the accesses in $(S_1 - (S_1 \cap S_2))$ terminate and the accesses in $(S_2 - \{A_k\})$ continue in progress. Since PS is fair, the requests in $(Q_1 - (Q_1 \cap Q_2))$ can only occur with
higher priority than $R_k$ a finite number of times, otherwise $R_k$ can be starved. Thus, there is a configuration which contains requests in $(Q_1 - (Q_1 \cap Q_2))$ with lower priority than $R_k$; $R_k$ and accesses in $(S_2 - \{A_k\})$. Clearly at least one request in $(Q_1 - (Q_1 \cap Q_2))$ can not be satisfied, otherwise the PS is not fair. Since algorithm MXCMPS constructs a maximally compatible PS, then those requests blocked by $R_k$ can be satisfied, thus a contradiction.
3. **Further Research and Problem Areas**

The results obtained in this thesis apply to algorithms which are prototypes of certain classes of solutions. For example, algorithm COMPS represents a class of solutions which allow requests to occur and accesses to terminate nondeterministically, and independently of other requests of other processes and accesses. On the other hand, in algorithm LVCMPS requests are grouped into different sets and all requests in any one set can occur simultaneously, only.

Further research is needed to determine equivalent classes of production systems, so that the results obtained in this thesis can be applied to a wider range of algorithms.

The algorithms developed in this thesis can be used to model concurrent processing systems in a distributed or multiprogrammed environment. It is of interest to further distinguish between these two environments within the model derived to represent the system at hand. As a consequence, we will be able to derive some results which may apply to one environment but not the other. For instance, consider the contention problem, i.e., when two or more events occur simultaneously. In a distributed environment, two requests
to a resource may occur simultaneously. In a multiprogrammed environment, this does not occur since there can only be one process running at any one time.

There are synchronization problems which we have not addressed in this thesis. Of particular interest are the problems of indeterminacy and consistency usually encountered in distributed data bases. A set of operations is indeterminate if the order of executing the operations affects the net outcome of the operations. The consistency problem is to guarantee that all multiple copies of data converge to one consistent copy after a specified number of operations have been performed.

It is also of interest to investigate how these properties interact with other properties such as maximal compatibility, liveness and fairness. In turn, the trade-offs involved can be evaluated to assist the designer in making a decision.

Other problems of concurrent processing include performance issues. We have already come across these issues when we attempted to derive the trade-offs involved in the preemption algorithm PLVCMPS for live production systems. The standard performance issues include throughput,
efficiency, overhead,...etc. The feasibility of studying these issues within production system models is yet to be determined. It is also desirable to study the effects of certain properties on the performance of the system under consideration.

Finally, the procedure we have developed to derive implementations from a production system yields synchronization procedures which are very basic in structure. As a consequence, further work to optimize these procedures is needed. Some of the procedures already appear to be too redundant in their content. Furthermore, concepts such as access control can be enforced by injecting the proper sequence of operations within the synchronization procedures. This is specially feasible when access control is implemented in terms of capabilities [43], [45].
BIBLIOGRAPHY


51. Gligor, V.D. Architecture Implications of Abstract Data Type Implementation. Dept. of Comp. Sci., Univ. of Maryland, College Park, Maryland.


142. SIGCOM/SIGOPS Proceeding (ACM), Interprocess Communications Workshop, Santa Monica, Calif., March 1975.


