LOOP-FREE SYNTHESIS OF FINITE STATE MACHINES

by

H. PAUL ZEIGER

B.S., Massachusetts Institute of Technology (1958)

S.M., Massachusetts Institute of Technology (1960)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF
DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September, 1964

Signature of Author

Department of Electrical Engineering, August 11, 1964

Certified by

Thesis Supervisor

Chairman, Departmental Committee
on Graduate Students
LOOP-FREE SYNTHESIS OF FINITE STATE MACHINES

By

Howard Paul Zeiger

Submitted to the Department of Electrical Engineering, Massachusetts Institute of Technology, August 21, 1964, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Techniques are developed for synthesizing a finite state machine as a loop-free interconnection of simpler machines. Methods are given for:

1) Constructing any machine from machines (hereafter called P-R machines) whose inputs either permute the states or reset them all to one state.

2) Constructing any P-R machine from other P-R machines whose state permutations constitute simple groups.

3) Constructing any P-R machine from a machine whose inputs only permute the states plus a machine whose inputs either a) leave the state unchanged or b) reset all states to one state.

The above methods provide a new way of reaching the result that for any given machine, the minimum set of groups arising in its loop-free construction is uniquely determined.

Thesis Supervisor: Professor Trenchard More, Jr.

Title: Assistant Professor of Electrical Engineering
ACKNOWLEDGMENTS

The research reported in this thesis was made possible in part by support extended the Massachusetts Institute of Technology, Research Laboratory of Electronics, jointly by the U. S. Army, the U. S. Navy and the U. S. Air Force.

The author is greatly indebted to his thesis supervisor, Professor Trenchard More, Jr., whose careful criticism has engendered whatever clear writing and logical thinking may be found in these pages. The author has further benefitted from the criticism and suggestions of Professor Frederick C. Hennie III, Professor J. B. Dennis and Professor Alfred K. Susskind. Thanks are due to Miss Barbara Johnson, who typed the manuscript.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>CHAPTER I. FUNDAMENTALS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-1</td>
<td>Finite State Machines</td>
<td>9</td>
</tr>
<tr>
<td>I-2</td>
<td>Notation</td>
<td>9</td>
</tr>
<tr>
<td>I-3</td>
<td>Compound Machines</td>
<td>11</td>
</tr>
<tr>
<td>I-4</td>
<td>Transport Mappings</td>
<td>13</td>
</tr>
<tr>
<td>I-5</td>
<td>State Assignments</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>CHAPTER II. ORDER</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>II-1</td>
<td>Ordered Compound Machines</td>
<td>18</td>
</tr>
<tr>
<td>II-2</td>
<td>Partitions</td>
<td>22</td>
</tr>
<tr>
<td>II-3</td>
<td>Ordered State Assignments</td>
<td>23</td>
</tr>
<tr>
<td>II-4</td>
<td>Ordered State Assignments from Covers</td>
<td>25</td>
</tr>
<tr>
<td>CHAPTER III. SPECIFIC COVERS</td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>III-1</td>
<td>A Cover Sequence Possessed by Any Machine</td>
<td>31</td>
</tr>
<tr>
<td>III-2</td>
<td>Range Covers</td>
<td>37</td>
</tr>
<tr>
<td>III-3</td>
<td>Covers from the Partitions of Transformations</td>
<td>43</td>
</tr>
<tr>
<td>III-4</td>
<td>Refinement of Covers</td>
<td>46</td>
</tr>
<tr>
<td>CHAPTER IV. LABELING</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>IV-1</td>
<td>Introduction</td>
<td>48</td>
</tr>
<tr>
<td>IV-2</td>
<td>Labeling Equivalent Bundles</td>
<td>49</td>
</tr>
<tr>
<td>IV-3</td>
<td>Weaker Conditions for Labeling</td>
<td>54</td>
</tr>
<tr>
<td>IV-4</td>
<td>A Procedure for Refining Covers</td>
<td>55</td>
</tr>
<tr>
<td>IV-5</td>
<td>Properties of the Refinement Procedure</td>
<td>59</td>
</tr>
<tr>
<td>IV-6</td>
<td>Discussion</td>
<td>60</td>
</tr>
</tbody>
</table>
INTRODUCTION

Whenever we wish to build a sequential machine, we face the problem of synthesizing the desired machine from basic building blocks. If the signals that flow from block to block can travel in loops, then any sequential machine can be built from combinational components. In this paper we force the sequential character of the whole machine to be reflected in the sequential character of the building blocks by requiring that the signals not flow in loops. The questions that naturally arise are: 1) What basic building blocks are needed to construct any sequential machine under loop-free interconnection? 2) Given the terminal behavior of a sequential machine, how can we build it from a given set of blocks? Question 2) was first approached by J. Hartmanis, who showed that if the state set of a machine has a partition that is preserved by every state transformation of the machine, then the desired machine can be built from two smaller sequential components connected in series. Unfortunately, many machines do not have such a partition; the preserved partitions, when they exist, may be hard to find. Question 1) was answered by J. L. Rhodes and K. B. Krohn, who showed that two-state machines plus machines whose state transformations constitute simple permutation groups form a universal set of building blocks. Unfortunately the algebraic techniques used to reach this result gave few clues about how to make an efficient synthesis.

The purpose of this paper is to bridge the gap between the works of Hartmanis and those of Rhodes and Krohn, making maximum use of techniques that are of practical interest to the designer of circuits because they provide efficient syntheses.
Chapter I contains a review of the fundamental ideas about sequential machines. Innovations here are: 1) the use of the transport mapping as a descriptive tool, and 2) a simple description of the idea of a state assignment.

In Chapter II we introduce the notion of an ordered state assignment. Finding an ordered state assignment for a machine amounts to synthesizing the machine as a loop-free interconnection of other machines. Hartmanis's theory is reviewed (from a slightly different point of view) and extended to the use of covers (whose blocks may overlap) instead of partitions. This extension was discovered independently by this author and several others. A simple procedure is given for constructing an ordered state assignment from a sequence of preserved covers.

While preserved partitions frustrate the designer by their scarcity, preserved covers embarrass him by their profusion. The material in Chapters III and IV, all of it new, brings a little order to the maze of preserved covers possessed by any machine. Ways of finding preserved covers are given, and bounds on the complexity of the resulting component machines are established. The main new idea of these chapters, and probably of the whole paper, is that of a permutation-reset machine. Permutation-reset machines appear to be the most natural components for loop-free synthesis. They are simple enough that the operation of each component is easy to understand, yet powerful enough that not too many of them are needed to synthesize any given machine.

Chapters V and VI provide the connecting link between the ideas developed in this paper and the theorems of Rhodes and Krohn. Algebra
has been held to a minimum in an effort to make at least the main ideas accessible to the reader who lacks a background in group theory. Consequently, all the constructions are new although the main results are not.

The primary objective throughout has been to give the reader an intuitive feeling for how sequential machines break into smaller components under loop-free interconnection. Passages that seem overly algebraic can safely be skipped on a first reading, to be read later when the necessary motivation arises.
I. FUNDAMENTALS

I-1 FINITE STATE MACHINES

Three different devices, the synchronous sequential switching circuit, the unilateral iterative system, and the one-way one-tape automaton can all be described by the same mathematical model. This model consists of two tables. The first, called the state table, lists the variable Y as a function of the variables x and y; the second, called the output table, lists the variable z as a function of the same x and y. An example is shown in Figure I-1.

In a synchronous sequential switching circuit y denotes the present state of the circuit, x the present input signal, Y the next state of the circuit, and z the present output signal. In a unilateral iterative system y denotes the left input signal of a typical cell, x the primary input signal, Y the right output signal, and z the primary output signal. In a one-way one-tape automaton the tables describe the synchronous sequential switching circuit that forms the head of the automaton.

The familiar state-diagram alternative to the description in Figure I-1 is given in Figure I-2.
Figure I-1. A tabular description

\[
\begin{array}{c|ccc}
  & 1 & 2 & 3 \\ 
1 & 2 & 3 & \ \\
2 & 1 & 3 & \\
3 & 3 & 4 & \\
4 & 1 & 2 & \\
\end{array}
\]

\[Y = (y, x)M\]

\[
\begin{array}{c|ccc}
  & 1 & 2 & 3 \\ 
1 & 1 & 2 & \ \\
2 & 2 & 1 & \\
3 & 1 & 1 & \\
4 & 2 & 1 & \\
\end{array}
\]

\[z = (y, x)V\]

Figure I-2. An alternative description to Figure I-1
The mathematical model described above will be called a finite state machine. The finite set of values assumed by \( y \) and \( X \) will be called \( Q \), the state set of the machine; the finite sets of values assumed by \( x \) and \( z \) will be called \( I \), the input set; and \( O \), the output set; the functions defined by the state table and the output table will be called \( M \), the excitation function, and \( V \), the output function. The above letters will retain the given meanings throughout this paper; different sets, variables, or functions denoted by the same letter will be distinguished by means of subscripts or superscripts.

In order to facilitate the writing of compositions of functions the argument of a function will always be written on the left of the function symbol. For example, in this notation \((y, x)M = Y\) and \((y, x)V = z\). A formula in this notation reads like a recipe for computing a function; \((y, x)MBC\) means: Take \((y, x)\), apply the function \( M \), then apply the function \( B \), then apply the function \( C \).

Later we shall want to investigate how the states of a machine are transformed by a given input. To this end we define, for each input \( x \), the corresponding state transformation \( M_x \) so that for every \( y \), \((y, x)M = (y)M_x\). Similarly we can associate with \( x \) an output mapping \( V_x \) so that for every \( y \), \((y, x)V = (y)V_x\). These mappings are described by single columns in the appropriate table of Figure I-1.
A formal description of a finite state machine reads as follows: A finite state machine is a quintuple \((Q, I, O, M, V)\) for which

1. \(Q\), \(I\), and \(O\) are nonempty finite sets
2. \(M: Q \times I \rightarrow Q\)
3. \(V: Q \times I \rightarrow O\).
I-3 COMPOUND MACHINES

In a sequential switching circuit one state may be represented by the states of several flip-flops. In a unilateral iterative system one intercell signal may be represented by the voltage levels on several wires. To incorporate these situations into our mathematical model it is necessary to consider a finite state machine whose state variable \( y \) is represented by an ordered \( k \)-tuple \((y_1, y_2, \ldots, y_k)\) where \( y_1, y_2, \ldots, y_k \) take values in the finite sets \( Q_1, Q_2, \ldots, Q_k \). Such a finite state machine will be called a compound machine.

In other words a compound machine is a finite state machine \((Q, I, \sigma, M, V)\) for which there exist \( k \) \((\geq 2)\) finite sets \( Q_1, Q_2, \ldots, Q_k \) so that \( Q \) is a subset of \( Q_1 \times Q_2 \times \ldots \times Q_k \).
The concept of a transport mapping, discussed below, is useful in describing 1) the idea of a state assignment, 2) cascade connection of machines, 3) state reduction, and 4) the relative computational power of two finite state machines.

Suppose that $Q$ is the state set of a finite state machine, $M_x$ is one of its state transformations, and $T$ is an arbitrary mapping that takes $Q$ onto some other set $R$. Under what circumstances does $T$ carry $M_x$ over into a transformation on $R$? The situation is shown pictorially in Figure I-3.

To define the desired transformation on some element $w$ in $R$ we might select an arbitrary $y$ in $Q$ for which $yT = w$. Then we could map $y$ into $yM_x$ and define $w'$, the image of $w$, as $yM_xT$. This process makes sense only if every possible choice of $y$ yields the same $w'$, and only if this happy coincidence is repeated for each $w$ in $R$.

To write the process explicitly we define $\overrightarrow{T}$, the relation converse to $T$, by requiring that $\overrightarrow{T}$ relate $w$ to $y$ whenever $yT = w$. For each $w$ in $R$ let $\{w\}$, the singleton of $w$, be the set having $w$ as its only element; then $\{w\}^{\overrightarrow{T}}$ is the set of all $y$ for which $yT = w$. The desired $w'$ will be uniquely defined whenever $\{w\}^{\overrightarrow{T}} M_x T$ is also a singleton set $\{w'\}$. If this happens for every $w$ we say: $T$ transports $M_x$, $\overrightarrow{T} M_x T$ is the transported version of $M_x$, and $w' = (w)^{\overrightarrow{T}} M_x T$. 

14.
The reader may wish to check that the T shown in Figure I-4 transports the $M_1$ of Figure I-1 but not the $M_2$.

Remarks:

1. For any 1-1 mapping we will write $T^{-1}$ instead of $\mathcal{T}$.
2. If T is 1-1 it transports any transformation.
3. If T transports both $M_1$ and $M_2$ then $(\mathcal{T}M_1T)(\mathcal{T}M_2T) = \mathcal{T}(M_1M_2)T$.
4. If $M_1$ is 1-1 and T transports $M_1$ then $\mathcal{T}M_1T$ is 1-1 and $(\mathcal{T}M_1T)^{-1} = \mathcal{T}(M_1^{-1})T$. 

15.
Figure I-3. The effect of a transport mapping

\[
y \quad yT \\
1 \rightarrow 1 \\
2 \rightarrow 1 \\
3 \rightarrow 2 \\
4 \rightarrow 1
\]

Figure I-4. A mapping that transports $M_1$ but not $M_2$
I-5 STATE ASSIGNMENTS

The traditional state assignment problem is to represent the states of a finite state machine by k-tuples; that is given a machine \((Q,I,O,M,V)\) to find a new machine \((Q^*,I,O,M^*,V^*)\) called the state assigned version of \((Q,I,O,M,V)\) and an assignment function \(Z\) from \(Q^*\) onto \(Q\) for which:

1. \((Q^*,I,O,M^*,V^*)\) is a compound machine.
2. \(M^*\) imitates \(M\), that is, for all \(x\) in \(I\) \(Z_{x} M^* Z = M_x\).
3. \(V^*\) imitates \(V\), that is, for all \(x\) in \(I\) \(V^* Z = Z V_x\).

Remarks:

1. Condition 2 includes the requirement that \(Z\) transport \(M^*_x\).
2. \(Q^*\) is a set of k-tuples, i.e. a subset of \(Q_1 \times Q_2 \cdots \times Q_k\).
3. The variables \(y_1, y_2, \ldots, y_k\) that take values in \(Q_1, Q_2, \ldots, Q_k\) are commonly called secondary variables.
4. We do not assume that the secondary variables are two-valued.
5. \(Z\) and \(V_x\) uniquely determine \(V^*_x\), but \(Z\) and \(M^*_x\) by no means uniquely determine \(M^*_x\).
II. ORDER

II-1 ORDERED COMPOUND MACHINES

Given any compound machine \((Q,I,0,M,V)\) we will say that each secondary variable is independent of all those that follow it if it is possible, for each state \(y\) and each \(M_x\), to calculate the first \(j\) coordinates of \(yM_x\) given only the first \(j\) coordinates of \(y\). To write this computation explicitly we define the function \(\text{proj}_1^j\) that selects the first \(j\) coordinates of a \(k\)-tuple, i.e.

\[
(y_1,y_2, \ldots y_k)\text{proj}_1^j = (y_1,y_2, \ldots y_j) \quad \text{where} \quad j \leq k.
\]

By definition of transport, \(\text{proj}_1^j\) transports \(M_x\) if and only if \(\bigcup \text{proj}_1^j M_x \text{proj}_1^j\) is a mapping, i.e. single valued. But if \(\bigcup \text{proj}_1^j M_x \text{proj}_1^j\) is a mapping, it maps the first \(j\) coordinates of \(y\) onto the first \(j\) coordinates of \(yM_x\) (see Figure II-1). Thus the intuitive requirement that each secondary variable be independent of all those that follow it is expressed by the algebraic requirement that \(\text{proj}_1^j\) transport every \(M_x\).

A compound machine will be called ordered if, for each \(j\), \(\text{proj}_1^j\) transports every \(M_x\). The state table of an ordered compound machine is shown in Figure II-2. A block diagram for the sequential switching circuit that corresponds to an ordered compound machine is shown in Figure II-3. Those portions of the diagram enclosed in dotted lines are themselves sequential circuits; the corresponding machines will be called the components of the ordered compound machine.
Figure II-1. \( \text{proj}_1^j \) transports \( M_x \)

\[
\begin{array}{c|ccc}
 & 1 & 2 \\
y_1 & 111 & 211 & 231 \\
y_2 & 112 & 212 & 231 \\
y_3 & 121 & 222 & 212 \\
 & 122 & 221 & 211 \\
 & 131 & 231 & 222 \\
 & 132 & 232 & 222 \\
 & 211 & 122 & 211 \\
 & 212 & 121 & 212 \\
 & 221 & 131 & 221 \\
 & 222 & 132 & 221 \\
 & 231 & 111 & 232 \\
 & 232 & 112 & 232 \\
\end{array}
\]

Figure II-2. The state table for an ordered compound machine
Figure II-3. Block diagram of an ordered sequential circuit
Remarks:

1. Each feedback loop occurs within a component, hence the interconnection of components is loop-free.

2. Each component gives as output its present state independently of its present input.
PARTITIONS

Given an ordered compound machine, we can form a partition \( P_j \) of the state set \( Q \) by placing in the same partition block (where a block is a subset of \( Q \)) all states whose first \( j \) coordinates are identical.

If \( 1 \leq j \) then any states that agree in their first \( j \) coordinates certainly agree in their first \( i \) coordinates; therefore every block of \( P_j \) is a subset of some block of \( P_i \). In this case we write \( P_j \preceq P_i \) (\( P_j \) is finer than \( P_i \)). Since \( P_k \preceq \ldots \preceq P_2 \preceq P_1 \) we call the set of partitions nested.

The above definition of \( \preceq \) need not be restricted to partitions. In the future we shall use it on arbitrary collections of subsets of \( Q \).

Consider what happens to the blocks of \( P_j \) under a state transformation \( M_x \). Any block each of whose states has \( y_1 \ldots y_j \) for its first \( j \) coordinates will map into a block each of whose states has \( (y_1 \ldots y_j)^{\text{proj}_i^j} M_x \text{proj}_j^j \) for its first \( j \) coordinates. Thus \( M_x \) maps each block inside another block, i.e. for all \( M_x \) and for all \( j \), \( P_j M_x \preceq P_j \). A partition with this property will be called preserved.
II-3 ORDERED STATE ASSIGNMENTS

Suppose that \((Q^*, I^*, O^*, M^*, V^*)\) is a state assigned version of \((Q, I, O, M, V)\) and that \((Q^*, I^*, O^*, M^*, V^*)\) is ordered. What special properties must \((Q, I, O, M, V)\) have?

Let us map each partition \(P_j\) of \(Q^*\) onto a collection \(P_jZ\) of subsets of \(Q\) by means of the assignment function \(Z\). Every state in \(Q\) belongs to at least one block of \(P_jZ\) because \(Z\) is onto. The blocks of \(P_jZ\) may overlap, since \(Z\) might coalesce states of \(Q^*\) that lie in different blocks of \(P_j\). In fact, \(Z\) might map one block of \(P_j\) inside the image under \(Z\) of some other block of \(P_j\). We shall agree to discard any blocks of \(P_jZ\) that lie inside other blocks of \(P_jZ\) and shall call the resulting collection \(P_jZ\text{max}\). Note that \text{max} is an operator that operates on a collection of sets to get rid of certain sets. This max operator will be used in several contexts. Let \(C_j = P_jZ\text{max}\).

So far we know about the \(C_j\) only that:

1. every state in \(Q\) is an element of some block of \(C_j\)
2. no block of \(C_j\) is a subset of another block of \(C_j\), i.e.

\[C_j\text{max} = C_j\]

Any collection of blocks satisfying these two conditions will be called a cover.

We know that if \(i \leq j\), \(P_j \leq P_i\). Since \(P_j \leq P_i\) if and only if every block of \(P_j\) is a subset of some block of \(P_i\), and since set-inclusion is preserved by any mapping, in particular by \(Z\), it follows that \(C_j \leq C_i\). Thus the covers are nested also.

23.
We know that each $P_j$ is preserved, i.e. for all $M_x$

$P_jM_x \preceq P_j$. By the same reasoning used in the last paragraph it follows that $C_jM_x \preceq C_j$. Thus the covers are preserved.

A nested sequence of preserved covers for the machine of Figure II-4a is shown in Figure II-4b.
II-4 ORDERED STATE ASSIGNMENTS FROM COVERS

In the previous section we saw that if a machine has an ordered state assignment, then it has a nested sequence of preserved covers. In this section we show the converse. That is, if a machine \((Q, I, O, M, V)\) has a nested sequence of preserved covers \(C_1, C_2, \ldots, C_k\), where \(C_k = \text{the set of all singleton subsets of } Q\), then there is a \((Q^*, I, O, M^*, V^*)\) for which

1. \((Q^*, I, O, M^*, V^*)\) is a state assigned version of \((Q, I, O, M, V)\)

2. \((Q^*, I, O, M^*, V^*)\) is ordered

3. the partitions \(P_j\) of \((Q^*, I, O, M^*, V^*)\) are mapped by the assignment function \(Z\) onto the covers \(C_j\), i.e. \(P_jZ\text{max} = C_j\).

We shall show this by giving a procedure for constructing the desired \((Q^*, I, O, M^*, V^*)\). The idea is that the first component will tell which block of \(C_1\) the machine is in; the second component will tell which block of \(C_2\) the machine is in given which block of \(C_1\) it is in, and so on. The hierarchy of possibilities so generated can be described by a tree diagram like the one shown in Figure II-5, which was constructed from the cover sequence of Figure II-4b.

The details of the construction are discussed under A below. The states of each component can be listed on the appropriate level of the tree diagram as shown in Figure II-6. The assignment of state labels to branches is discussed under B below. Finally, the state transformations of the desired compound machine can be viewed as transformations on paths in the tree diagram. The assignment of an appropriate path transformation to each original input is discussed under C below.

25.
c_1 = \{\{1346\}, \{2456\}\}
c_2 = \{\{146\}, \{346\}, \{25\}\}
c_3 = \{\{14\}, \{46\}, \{36\}, \{25\}\}
c_4 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}\}

Figure II-4. A machine with a nested sequence of preserved covers

Figure II-5. The tree diagram for the machine of Figure II-4
We now consider these steps in detail. For ease of understanding we look at them in the order B, C, A.

B. Label the branches on the \( j^{th} \) level of the diagram with elements of \( Q_j \), making sure that all branches leaving one node bear distinct labels. This guarantees that each path of length \( k \) from left to right through the diagram will have a different sequence of branch labels. These sequences will be the states of \( Q^* \). \( Z \) will map each sequence into the state in \( Q \) that appears at the tip of the corresponding path.

C. Since the machine we are constructing is to be ordered, the \( M^*_x \) associated with each \( M_x \) must be transported by \( \text{proj}_j^j \), i.e. the action of \( M^*_x \) on the first \( j \) branch labels must be independent of the remaining branch labels. We force this behavior by deciding the action of \( M^*_x \) one level at a time, proceeding from left to right through the diagram. For example, to determine \( M^*_1 \), the path transformation for \( x = 1 \), we proceed as shown in Figure II-7. Since input 1 maps both 1346 and 2546 into 2546, \( M^*_1 \) acts on the first labels as shown at (a). The action of \( M^*_1 \) on the second level is decided consistently with the first-level decision, as shown at (b). After proceeding in this way through the entire tree diagram, we get the first column of the state table shown in Figure II-8. If more than one \( M^*_x \) can result from this procedure, we pick one arbitrarily.

A. Start by listing the blocks of \( C_1 \) on the first level of the tree diagram (see Figure II-5). From each block \( B \) of \( C_1 \) draw
Figure II-6. Labeling the tree diagram

Figure II-7. Determining path transformations for x = 1
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>211</td>
<td>121</td>
</tr>
<tr>
<td>111</td>
<td>211</td>
<td>121</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>122</td>
</tr>
<tr>
<td>112</td>
<td>212</td>
<td>122</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>122</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>122</td>
</tr>
<tr>
<td>111</td>
<td>212</td>
<td>111</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>112</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>112</td>
</tr>
<tr>
<td>112</td>
<td>211</td>
<td>112</td>
</tr>
<tr>
<td>211</td>
<td>221</td>
<td>112</td>
</tr>
<tr>
<td>211</td>
<td>221</td>
<td>112</td>
</tr>
<tr>
<td>211</td>
<td>222</td>
<td>111</td>
</tr>
<tr>
<td>211</td>
<td>212</td>
<td>112</td>
</tr>
<tr>
<td>211</td>
<td>212</td>
<td>112</td>
</tr>
</tbody>
</table>

Figure II-8. Ordered state assignment for the machine of Figure II-4.
a set of branches each of which terminates in a block of $C_2$ that is a subset of $B$. This accounts for all the blocks on the second level of the diagram in Figure II-5 except 46, a block that is needed because $M_1$ maps $25\,\!46$ onto itself and takes $25$ onto $46$. In general, additional blocks like this one can be added as needed, or they can all be included at the outset by the following scheme. Every block appearing at the end of a branch emanating from $B$ must be:

1. a subset of $B$, and

2. a subset of some block of $C_2$ (because $C_2$ is preserved).

Thus we can get a sufficient set of blocks by selecting all subsets of $Q$ that satisfy 1 and 2, then apply max to get rid of needless blocks. The tree diagram can be completed by iteration of the above procedure.

We have now shown how, given $(Q, I, O, M, V)$ and $C_1 \ldots C_k$, to construct an ordered state assigned version $(Q^*, I, O, M^*, V^*)$. The method of construction guarantees that the assignment function $Z$ will map each $P_j$ onto $C_j$. 

30.
III. SPECIFIC COVERS

III-1 A COVER SEQUENCE POSSESSED BY ANY MACHINE

So far we have shown how to produce an ordered state assignment given a nested sequence of preserved covers. We now turn to the problem of finding such cover sequences. Our first example will be a cover sequence that is possessed by any machine and that leads to particularly simple components.

Given a machine $(Q, I, O, M, V)$ with $n$ states we let $C_j = \{\} \in \mathcal{P}(Q)$ of all $n-j$ element subsets of $Q$. The $C_j$ are obviously nested. They are also preserved since mapping by any $M_x$ cannot increase the number of states in a set. For example, consider the machine whose state table is shown in Figure III-1. The tree diagram for the above sequence of covers is shown in Figure III-2. If we select path transformations as described in the previous chapter, then we find that no paths map into any of the paths that pass through $123$ or $234$. These needless paths can be discarded to give the diagram of Figure III-3. By a similar argument we can discard the paths that pass through $134$ and $13$ and those that pass through $124$ and $14$ to give the diagram of Figure III-4. We construct an ordered compound machine based on this tree diagram to get the state table shown in Figure III-5.

Next we show that the components of this compound machine are very simple. The state tables of the components are shown in Figure III-6. For example, the table for the second component
Figure III-1. A state table

Figure III-2. The tree diagram for the sequence of covers described in section III-1
Figure III-3. An abbreviated tree diagram

Figure III-4. A further abbreviated tree diagram
Figure III-5. The ordered state assignment for the machine of Figure III-1

Figure III-6. The components of the ordered compound machine of Figure III-5
(Figure III-6b) is derived from the table of Figure III-5 by:
1. omitting \( Y_1 \) and \( Y_3 \) entries
2. omitting \( y_3 \) entries
3. collapsing identical rows
4. rearranging the table so \( y_1 \) is viewed as an input.

Observe that for each component, every component input either permutes the states or resets them all to one state. We shall call a machine with this property a P-R machine.

To see that the components resulting from the above sequence of covers will always be P-R, we define a \( j \)-bundle as a set of all paths of length \( j \) that agree in their first \( j-1 \) branches. We observe that each state transformation of the \( j \)th component is determined by the behavior of some \( j \)-bundle under some \( M_x \). For example, the state transformation \( 1 \rightarrow 2, 2 \rightarrow 1 \) is produced in the third component of the machine of Figure III-5 when the input \( x = 1 \) acts on the 3-bundle whose common first two branches are labeled \( y_1 = 1, y_2 = 1 \). Consider the mappings of bundles shown in Figure III-7. If \( B \) is the block of states that constitute the foot of the bundle as shown in Figure III-7 and \( B \) contains \( b \) states, then \( BM_x \) contains \( \leq b \) states. If fewer, then \( BM_x \) is contained in some block on the next level of the diagram, because of the way the diagram was constructed. In this case a reset is produced in the \( j \)th component as shown in Figure III-7a. On the other hand, if \( BM_x \) contains just \( b \) states, then \( M_x \) is l-l on \( B \) and therefore maps the b-l element subsets of \( B \) l-l onto the b-l
element subsets of $E_{x}$. In this case a permutation is produced in the $j^{th}$ component as shown in Figure III-7b.

Thus, given any $n$-state machine we can find an ordered state assignment for it in which:

1. the components are P-R
2. there are at most $n-1$ components
3. the $j^{th}$ component has at most $n-j+1$ states.
III-2 RANGE COVERS

The cover sequence described in the previous section has several shortcomings. First, it tends to generate many extraneous cover blocks that have to be eliminated by ad hoc arguments of the type used in the example. Second, if the method is used on the machine shown in Figure III-8, then the first component turns out to have the same form as the original machine, so the remaining n-2 components are useless. (The complaint is not that the method fails, for no cover sequence will do any better for this machine, but that it gives us no warning that the machine is indecomposable.)

In this section we develop a cover sequence that eliminates the above difficulties. If we examine the cover blocks that remain in Figure III-4 after the ad hoc eliminations, we find that each block is the image of the state set Q under the composition \( M_r \) of several state transformations \( M_1, M_2, \ldots, M_m \). We will call such a block the range of the transformation \( M_r \), and will call the number of states in the block the rank of \( M_r \). Our observation suggests that we might shortcut the ad hoc eliminations if we always constructed covers out of the ranges of state transformations.

One difficulty appears immediately. Certain states may not be contained in any range. We need a way of expanding a collection of ranges until it actually covers Q. To this end we define the \( \text{join} \ C_1 \lor C_2 \) of any two collections of subsets of Q as \{all blocks of \( C_1 \) or of \( C_2 \)\} max. Observe that if \( C_1 \) and \( C_2 \) are both preserved,
(a) \hspace{5cm} (b)

Figure III-7. Component state transformations

\begin{align*}
x & \\
\begin{array}{c|cc}
y & 1 & 2 \\
1 & 5 & 2 \\
2 & 1 & 2 \\
3 & 2 & 2 \\
4 & 3 & 2 \\
5 & 4 & 2 \\
\end{array} & \text{Figure III-8. A machine without usable preserved covers}
\end{align*}

38.
so is \( C_1 \cup C_2 \). Next we define the special cover \( O = \{ \text{all singleton subsets of } Q \} \). Now if \( C \) is a preserved collection of ranges that fails to cover \( Q \), \( C \cup O \) is a preserved cover.

Next we assert that \( \{ \text{all ranges containing } \leq i \text{ states} \} \) is a preserved collection, since the image under \( M_r \) of any range is another range, and no \( M_r \) can expand a set. Then \( \{ \text{all ranges containing } \leq i \text{ states} \} \cup O \) will be a preserved cover. By using different \( i \)'s we get a nested sequence of such covers. For example, the machine of Figure III-9a has the ranges developed in Figure III-9b. In this example:

1. \( \{ \text{ranges having } \leq 3 \text{ states} \} \cup O = \{ 123,134,12,13,1 \} \cup \{ 1,2,3,4 \} = \{ 123,134 \} \)
2. \( \{ \text{ranges having } \leq 2 \text{ states} \} \cup O = \{ 12,13,1 \} \cup \{ 1,2,3,4 \} = \{ 12,13,4 \} \)
3. \( \{ \text{ranges having } \leq 1 \text{ state} \} \cup O = \{ 1 \} \cup \{ 1,2,3,4 \} = \{ 1,2,3,4 \} \).

We shall call the sequence of covers resulting from this process the range cover sequence. It seems to be more economical than the sequence described in the previous section, but has one disadvantage: it need not always lead to a state assignment that has P-R components. To see why, we study the machine of Figure III-10a. Its range cover sequence is developed in Figures III-10b and III-10c, and the resulting tree diagram appears in Figure III-11. The interesting component is the second, whose state table is shown in Figure III-12. First observe that there are don't-care entries in
Figure III-9. A machine and its ranges

\[
\begin{array}{c|cc}
  & 1 & 2 \\
\hline
  y & 1 & 1 \\
  & 2 & 3 \\
  & 3 & 3 \\
  & 4 & 4 \\
\end{array}
\quad y
\begin{array}{c|ccc}
  & 1 & 2 & 3 \\
\hline
  & 123 & 12 & 13 \\
  & 134 & 13 & 134 \\
  & 12 & 12 & 13 \\
  & 13 & 1 & 13 \\
  & 1 & 1 & 1 \\
\end{array}
\]

(a) \quad (b)

Figure III-10. A counterexample to the conjecture that range covers give P-R components

\[
\begin{array}{c|cc}
  & 1 & 2 \\
\hline
  y & 1 & 3 \\
  & 2 & 4 \\
  & 3 & 6 \\
  & 4 & 1 \\
  & 5 & 1 \\
  & 6 & 5 \\
\end{array}
\quad y
\begin{array}{c|ccc}
  & 1 & 2 & 3 \\
\hline
  & 123 & 123 & 456 \\
  & 456 & 1 & 45 \\
  & 45 & 1 & 4 \\
  & 1 & 3 & 4 \\
  & 2 & 1 & 5 \\
  & 3 & 2 & 6 \\
  & 4 & 1 & 4 \\
  & 5 & 1 & 4 \\
  & 6 & 1 & 5 \\
\end{array}
\]

(a) \quad (b) \quad (c)

\[c_1 = \{\{123\}, \{456\}\} \quad c_2 = \{\{45\}, \{1\}, \{2\}, \{3\}, \{6\}\} \quad c_3 = 0\]
Figure III-11. A tree diagram for the covers of Figure III-10

Figure III-12. The second component derived from Figures III-10 and 11. The second column is neither permutation nor reset.
the table. They correspond to conditions that can never arise. They do not upset the P-R character, however, since any transformation that is a permutation or a reset on some subset of $Q_j$ can always be extended to a permutation or a reset on all of $Q_j$.

We next observe that the component input $y_1 = 1, x = 2$ leads to a state transformation that is neither a permutation nor a reset. Thus the range cover sequence need not always give P-R components.
III-3 COVERS FROM THE PARTITIONS OF TRANSFORMATIONS

The cover sequences described so far have been universal; every machine has them. In this section we describe a class of covers that may, for a given machine, be empty. When nonempty, however, it is very useful.

We shall construct preserved covers (in fact, partitions) from the natural partitions (see below) of state transformations. If \( M_r \) is a state transformation, its natural partition \( P_r \) is the set of all images under \( M_r \) of singleton subsets of \( Q \). Thus each block of \( P_r \) is mapped into a singleton by \( M_r \). For example, in the machine of Figure III-13: \( M_1 \) has the natural partition \( P_1 = \{15, 24, 3\} \), \( M_2 \) has the natural partition \( P_2 = \{1, 2, 4, 35\} \), \( M_2 M_1 \) has the natural partition \( P_{21} = \{135, 24\} \). The set of all natural partitions for the machine of Figure III-13 is developed in the redundant backward table shown in Figure III-14.

The natural partitions are not necessarily preserved, but we can get preserved partitions by intersecting natural partitions. To this end we define the meet \( C_1 \land C_2 \) of two arbitrary collections of subsets of \( Q \) as \{all sets that are subsets of both a block of \( C_1 \) and a block of \( C_2 \}\}_\text{max}. Observe that this reduces to the usual definition of intersection of partitions when \( C_1 \) and \( C_2 \) are partitions.

Now consider the meet of all natural partitions of transformations of rank \( \leq i^+ \). In the example of Figure III-13: when

\[ + \text{ For definition of rank, see page 37, second paragraph.} \]
Figure III-13. A machine with preserved input partitions

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure III-14. The backward table for the machine of Figure III-13

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>135</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>35</td>
</tr>
<tr>
<td>-</td>
<td>35</td>
<td>4</td>
</tr>
<tr>
<td>135</td>
<td>24</td>
<td>135</td>
</tr>
<tr>
<td>24</td>
<td>135</td>
<td>24</td>
</tr>
</tbody>
</table>
\[ i = 2, \text{ this meet } = \{135,24\}; \text{ when } i = 3, \text{ this meet } = \{15,24,3\} \cap \{135,24\} = \{15,24,3\}; \text{ when } i = 4, \text{ this meet } = \{1,2,3,4,5\} = 0. \] These meets will always be preserved. To see this, recall that the blocks of the meet of all partitions of transformations of rank \( \preceq i \) are just those blocks mapped into a singleton by every state transformation of rank \( \preceq i \). If \( M \) is any state transformation of rank \( \preceq i \) and \( M' \) is any state transformation, \( M'M \) is a state transformation of rank \( \preceq i \). If \( B \) is a block of the above partition, then \( B(M'M) = (BM')M \) is a singleton; so \( BM' \) is also mapped into a singleton by every transformation of rank \( \preceq i \).

The set of preserved partitions generated by the method of this section (using various \( i \)'s) is nested. It often contains only the partition 0. If the machine has finite memory, then the above sequence of partitions gives an ordered state assignment in which each component is a unit delay. This has been shown (using different methods from ours) by Liu in "Some Memory Aspects of Finite Automata", RLE Technical Report 411.
III-4 REFINEMENT OF COVERS

It is often desirable, given a preserved cover \( C_1 \), to find a preserved cover \( C_2 \) that is, intuitively, a slight refinement of \( C_1 \).

Suppose the number of states in the largest block of \( C_1 \) is \( m \). A simple procedure is to form the collection of all images of blocks \( C_1 \) together with all blocks of \( C_1 \) itself; discard the blocks or images of blocks containing exactly \( m \) states; then join with 0.

For example, let \( C_1 = \{13, 46, 25\} \) as in Figure III-15. The collection of all images of blocks of \( C_1 \) is developed in Figure III-16. Discarding the four-element block and joining with 0 (which does nothing in this case) we have \( C_2 = \{25, 46, 14, 36\} \).

If the problem is to insert a preserved cover between \( C_1 \) and \( C_3 \), where \( C_3 \leq C_1 \), it is reasonable to form \( C_2 \) from \( C_1 \) as above, then join with \( C_3 \).

Note that the sequence of range covers described in section III-2 is generated by successive application of this refinement procedure to the cover sequence \( \{Q\}, 0 \).
Figure III-15. A machine with a cover that we wish to refine

\[ c_1 = \{\{1346\}, \{25\}\} \]

\[
\begin{array}{c|c|c}
1 & 2 & 3 \\
2 & 4 & 4 \\
3 & 5 & 1 \\
4 & 5 & 6 \\
5 & 6 & 1 \\
6 & 2 & 4 \\
\end{array}
\]

Figure III-16. Construction of refined cover

\[ c_2 = \{\{25\}, \{46\}, \{14\}, \{36\}\} \]

\[
\begin{array}{c|c|c}
\text{Images} & 1346 & 25 \\
& 1346 & 14 \\
25 & 46 & 14 \\
46 & 25 & 46 \\
14 & 25 & 36 \\
36 & 25 & 14 \\
\end{array}
\]
IV. LABELING

IV-1 INTRODUCTION

Given any finite state machine we might find some covers for it by the methods of chapter III, then produce an ordered state assignment by the methods of chapter II. During this process we would be confronted with the problem of labeling the tree diagram; the complexity of the resulting components might be quite sensitive to the labeling selected. In this chapter we explore possible ways of intelligently selecting a labeling.

We shall begin by exhibiting very narrow conditions under which the labeling problem has a satisfactory answer. We shall then weaken the conditions slightly. Next we show how, for any machine, to find a sequence of covers that gives a tree diagram that meets our conditions. Finally we shall investigate the structure of the ordered state assignment that results from this process.
IV-2 LABELING EQUIVALENT BUNDLES

As an example of the effect of labeling on the form of the components, consider the machine of Figure IV-1a with the preserved covers of IV-1b. Two possible choices of labeling are shown in Figure IV-2, and the resulting ordered state assigned versions are shown in Figure IV-3. The third component of the state assigned version shown in Figure IV-1a has for state transformations only the identity transformation and the two resets, while the third component of the state assigned version shown in Figure IV-1b has, in addition to those transformations, the transformation that interchanges states 1 and 2. This is an example of the extraneous transformations we wish to avoid by proper labeling.

Each state transformation of the $j^{th}$ component is produced by some mapping on a $j$-bundle of the tree diagram. The bundle mapping responsible for the extra transformation in Figure IV-3b is shown in Figure IV-4. We note that the mapping goes from one bundle to another; so in this case, it appears that our problem lies in the relative labeling of the two bundles. Indeed, it is clear that the labeling for one bundle can always be chosen arbitrarily; then the problem is to label the other bundles properly relative to it.

Suppose, then, that $D$ is a $j$-bundle and we have labeled it arbitrarily, i.e. we have selected a mapping $W_d$ that is 1-1 from $D$ into $Q_j$. Then certain state transformations of the $j^{th}$ component
Figure IV-1. A machine with preserved covers

Figure IV-2. Two possible labelings
Figure IV-3. State tables for the two state assignments of Figure IV-2

Figure IV-4. The pertinent 3-bundles
are inevitable, namely those that result from path transformations that take D into itself. When can some other j-bundle, E, be labeled so as to introduce no additional state transformations in the jth component? We assert that if there are input sequences whose associated path transformations map the bundles D and E 1-1 onto each other, then these path transformations can be used to transport a labeling from D to E in a way that introduces no new state transformations in the jth component, at least as far as transformations among these two bundles are concerned. To say this more precisely, we recall that each input sequence produces a mapping on every j-bundle according to the construction of section II-4, part C+. We examine all the mappings R on D and S on E so produced. (Although the construction of II-4 assumed a previous labeling, it is clear that the actual path transformations in no way depend on this preliminary labeling.) We call D and E equivalent if we can find R and S so that \( S = R^{-1} \). (This definition includes the requirement that R and S be 1-1.) If D and E are equivalent and we have chosen \( W_D \) arbitrarily, then the choice \( W_E = S W_D \) introduces no new transformations in the jth component. To see why, consider any transformation T on E produced by a sequence of inputs. Suppose first that T maps E into E (see Figure IV-5). The component state transformation \( X = W_E^{-1} T \) \( W_E = W_D^{-1} \) RTS \( W_D \); so X results from a transformation that maps D into itself, namely RTS. Similar arguments dispose of the two remaining cases, i.e. when T maps E into D and when T maps D into E.

+ In general, an input sequence will take a j-bundle into some other bundle, as do R and S in Figure IV-5.
If D is equivalent to both E and F, then E is equivalent to F, for the needed transformations can be constructed from the transformations that make D equivalent to E and F. Furthermore, if the above technique is used to assign labels, then mappings among D, E and F introduce no more component state transformations than are produced by mappings that leave D fixed.

Thus, having chosen a labeling for D, we can always find satisfactory labelings for all bundles equivalent to D. If we are so fortunate as to have a tree diagram in which, for all j, all the j-bundles are equivalent, we can construct the entire labeling in a satisfactory way. Since such tree diagrams are rarely encountered, we have to weaken the requirements on the tree diagram.
IV-3  WEAKER CONDITIONS FOR LABELING

We first observe that we can weaken the conditions on the equivalence of bundles. Instead of requiring that \( R = S^{-1} \), we need only require that \( R \) and \( S \) be 1-1, for if \( R \) and \( S \) are 1-1, we can find an appropriate \( S' \) that is inverse to \( R \). To compute \( S' \), observe that \( SR \) is a permutation on \( E \), so there is an integer \( k \) for which \( (SR)^k \) is the identity. Then let \( S' = (SR)^{k-1}S \) so \( S'R = \text{identity} \). Furthermore, \( S' \) results from a well-defined input sequence. Henceforth we use the weaker statement for equivalence of bundles.

Next we observe that there is one type of \( j \)-bundle that never has to satisfy any equivalence conditions. Call a \( j \)-bundle singular if it contains only one path of length \( j \). A singular \( j \)-bundle results, for example, whenever some block of \( C_{j-1} \) is also a block of \( C_j \). Obviously, any path transformation from any bundle into a singular bundle must produce a reset. Furthermore, any path transformation from a singular bundle into any bundle produces a component state transformation that is defined on only one element of \( Q_j \), hence can be extended to a reset.

Therefore we can construct the entire labeling in a satisfactory way if, for each \( j \), all nonsingular \( j \)-bundles are equivalent.
A PROCEDURE FOR REFINING COVERS

We now know how to label any tree diagram in which, for each \( j \), all nonsingular \( j \)-bundles are equivalent. Our next project is to show how, given any machine, we can find a nested sequence of preserved covers whose tree diagram meets the above requirement. The intuitive idea is to discover the slightest possible preserved refinement of a given preserved cover \( C_1 \).

We need the following definitions:

1. Two blocks of \( C_1 \) are similar if each is the image of the other under some \( M_x \).
2. A block \( B \) of \( C_1 \) is initial in \( C_1 \) if it is not the image under any \( M_x \) of any other block \( A \) of \( C_1 \), unless \( A \) is similar to \( B \).

If \( B \) is initial in \( C_1 \), then all blocks similar to \( B \) are also initial in \( C_1 \), so they form an initial similarity class. Now we can refine \( C_1 \) by the following procedure:

1. Form \( \overline{C}_1 \), the collection of all blocks of \( C_1 \) together with all images under every \( M_r \) of all blocks of \( C_1 \).
2. Discard from \( \overline{C}_1 \) any initial similarity class of blocks.
3. Join with 0. (Remember that this includes a max operation.)

The cover \( C_2 \) that results from this process is a refinement of \( C_1 \) because \( C_1 \) is preserved, and therefore every image in \( \overline{C}_1 \) is a subset of some block of \( C_1 \). \( C_2 \) is a proper refinement since it lacks those blocks that belong to the discarded similarity class.
C_2 is preserved because discarding an initial similarity class does not wreck the preservedness of \( C_1 \); no block in the discarded class is an image of any block outside the discarded class. When we construct a tree diagram from \( C_1 \) and \( C_2 \), any block of \( C_1 \) not belonging to the discarded similarity class reappears in \( C_2 \) to produce a singular 2-bundle. The only nonsingular 2-bundles will be those that pass through a block of \( C_1 \) that was discarded in going from \( C_1 \) to \( C_2 \).

Suppose that a nested sequence of preserved covers is got by repeated application of the above procedure, starting from the trivial cover \( \{ \emptyset \} \). We wish to show that in the resulting tree diagram, all nonsingular j-bundles are equivalent. As above, if a j-bundle is nonsingular, then its foot (like A or B in Figure IV-6) belonged to the primitive similarity class that was discarded in going from \( C_{j-1} \) to \( C_j \). Because all such blocks are similar, we can find a state transformation that takes any one onto any other; in particular, we can find an \( M_1 \) that takes A onto B and an \( M_2 \) that takes B onto A. Let the path transformations on D and E produced by \( M_1 \) and \( M_2 \) be R and S. We want to show that RS permutes D. Suppose not: then RS must take two paths of D into one path of D; hence \( M_1 M_2 \) must map two sub-blocks of A, say \( A_1 \) and \( A_2 \), into some sub-block of A, say \( A_3 \). Since \( M_1 M_2 \) permutes A, there is an integer k for which \( (M_1 M_2)^k = \text{identity on A} \); the state transformation \( (M_1 M_2)^{k-1} \) is inverse to \( M_1 M_2 \). Since
Figure IV-5. $X = W_D^{-1} TW_E = W_D^{-1}(RTS)W_D$

Figure IV-6. Bundles resulting from the refinement procedure
$C_j$ is preserved, the image of $A_3$ under $(M_1M_2)^{k-1}$ is contained in some sub-block $A_4$ of $A$; so both $A_1$ and $A_2$ must be subsets of $A_4$. This is impossible because the max operation in the construction of the tree diagram (section II-4, part A) guarantees that no sub-block of $A$ is a proper subset of any other sub-block of $A$. Thus we have shown by contradiction that $RS$ is 1-1 on $D$.

Therefore both $R$ and $S$ must be 1-1, so $D$ and $E$ are equivalent. This is the desired result. If the given procedure is used to generate a nested sequence of preserved covers, then in the resulting tree diagram, for all $j$, all nonsingular $j$-bundles are equivalent.
IV-5. PROPERTIES OF THE REFINEMENT PROCEDURE

Suppose we are given a finite state machine and are asked to find an ordered state assignment with the simplest possible components. We start with the trivial cover \( \{q\} \) and apply the refinement procedure of the previous section to get a sequence of preserved covers \( Q_1, Q_2, \ldots, Q_k = 0 \). In the resulting tree diagram, for each \( j \), all nonsingular \( j \)-bundles are equivalent, so we can label the diagram by the methods of section IV-2.

We first assert that the components in the resulting ordered state assignment are P-R. According to section IV-2, each non-reset transformation of the \( j \)-th component results from the action of some \( M_x \) on some \( j \)-bundle \( D \) that is mapped into itself by the path transformation corresponding to \( M_x \). Let \( B \) in \( C_{j-1} \) be the foot of \( D \). If \( M_x \) permutes \( B \), then the corresponding path transformation permutes \( D \) (by the same argument used at the end of the previous section). If \( M_x \) does not permute \( B \), then \( B M_x \) is smaller than \( B \); \( B M_x \) can't be similar to \( B \), so \( B M_x \) is contained in some block \( A \) of \( C_j \). Since \( M_x \) maps \( B \) into \( A \) it maps every sub-block of \( B \) into \( A \), to produce a reset in the \( j \)-th component. Therefore the \( j \)-th component is P-R.

We next assert that if \( M_x \) permutes \( B \), then the corresponding permutation \( R \) of \( D \) is uniquely determined. If both \( R_1 \) and \( R_2 \) correspond to \( M_x \), then \( R_1 R_2^{-1} \) corresponds to \( M_x M_x^{-1} \) = identity on \( B \); so \( R_1 = R_2 \). Let \( S_B \) be the semigroup of all state transformations.
that permute $B$. The mapping $h$ that takes each $M_x$ in $S_B$ into the corresponding permutation on $D$ is a homomorphism from $S_B$ onto the group $G_j$ of permutation inputs to the $j^{th}$ component. When we later demonstrate the main theorem of Rhodes and Krohn, we shall use the fact that in the ordered state assignment described in this section each component group $G_j$ is a homomorphic image of a subsemigroup $S_B$ of the semigroup of all state transformations of the machine.

We shall actually need a stronger version of this result:

Each $G_j$ is a homomorphic image of a subgroup of the semigroup of state transformations of the machine. That is, if $h$ is a homomorphism from $S_B$ onto $G_j$, then there is a subgroup $G$ of $S_B$ for which $h$ is a homomorphism from $G$ onto $G_j$. We shall show this by selecting an idempotent $e$ in $S_B$ (i.e. an element for which $ee = e$) for which:

1) $h$ maps $eS_Be$ onto $G_j$

2) $eS_Be$ is a group.

Assertion 1) is true for any idempotent $e$. If $e$ is an idempotent in $S_B$, then $eh$ must be the identity in $G_j$; therefore for all $b$ in $S_B$, $bh = (eh)(bh)(eh) = (ebe)h$. To prove 2) we select an idempotent $e$ of smallest rank in $S_B$. Then all elements of the form $ebe$ must have the same natural partition and the same range; otherwise there would be an idempotent of the form $(ebe)^k$ (see Clifford and Preston, page 20) having rank smaller than $e$. A semigroup of transformations all of which have the same natural partition and the same range is necessarily a group, so $eS_Be$ is a subgroup of the semigroup of state transformations.
The above arguments suggest an easy way of finding out what
groups to expect in the components for a given machine. Given the
machine of Figure IV-7a we form the table of ranges shown in
IV-7b. We mark with an asterisk those ranges that are permuted
by input sequences. There is one similarity class \{248\} of
three-element ranges, and two similarity classes, \{37,26\} and
\{24,28,48\}, of two element ranges. The range 248 is permuted
cyclicly. Thus among the component groups we need just one copy
of the cyclic group of degree 3, and at most two copies of the
two-element group. Actually, the tree diagram of Figure IV-8 gives
an ordered state assignment with the following components:
2 unit delays, one 2-state P-R machine, one 3-state P-R machine
with cyclic permutations.
<table>
<thead>
<tr>
<th>y</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>13579</td>
</tr>
<tr>
<td>2468</td>
</tr>
<tr>
<td>159</td>
</tr>
<tr>
<td>246</td>
</tr>
<tr>
<td>*248</td>
</tr>
<tr>
<td>*37</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>*48</td>
</tr>
<tr>
<td>*26</td>
</tr>
<tr>
<td>*24</td>
</tr>
<tr>
<td>*28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

etc.

Figure IV-7. Estimating the component groups

Figure IV-8. A possible tree diagram for the machine of Figure IV-7

62.
IV-6 DISCUSSION

Thus we have a decomposition into P-R components that works for every machine. Is further decomposition possible, i.e. can we find ordered state assignments even for P-R machines? None of the methods so far discussed will find a usable preserved cover for a P-R machine. It can still happen, however, that a P-R machine has some usable preserved covers. Furthermore, useful ordered state assignments can sometimes be found even for P-R machines that have no usable preserved covers. We will attack the ordered state assignment problem for P-R machines by arguing that if machine A is P-R, and machine B is got from A by deleting the reset inputs, the problem of finding ordered state assignments is identical for the two machines. That is, any ordered state assignment for A can be converted to an ordered state assignment for B by deleting some component inputs and every ordered state assignment for B can be converted to an ordered state assignment for A by adding some inputs that merely reset components. Therefore we can restrict our attention to permutation machines, and all our results will apply also to the P-R case. The next chapter will treat permutation machines.
V. PERMUTATION MACHINES

V-1 OBJECTIVES

Our objective in this chapter is to explore the kinds of ordered state assignments possessed by permutation machines. We shall show that for a given permutation machine certain types of components must appear in every ordered state assignment and that only these types are needed. Luck and engineering skill, however, determine how many components of each type are used. In chapter VI we shall generalize this result to arbitrary machines.

The P-R components necessary for a given machine will be identified only up to their groups. For this reason our constructions will repeatedly use as components the most complicated machines having given groups. It is well to remember that in practice, additional engineering is necessary to reduce each component to simplest form without changing its group.
V-2 INTRANSITIVE PERMUTATION MACHINES

Call two states $y$ and $Y$ of a permutation machine connected if there is a sequence $r$ of inputs for which $yM^r_r = Y$. The relation of connectedness is reflexive, symmetric and transitive; it therefore partitions the state set $Q$ into connectedness classes. We shall call a machine transitive if there is just one such class; intransitive if there are two or more. Since an intransitive machine remains forever in the connectedness class in which it starts, any intransitive machine can be imitated by a number of transitive machines preceded by a machine that remembers forever its initial state, hereafter called an identity machine. This representation corresponds to an ordered state assignment based on the nested sequence $C_1 \ldots C_k$ of preserved covers got by the following rules:

1. $C_1$ = the partition consisting of all connectedness classes of states,

2. $C_2 = (C_1$ with one connectedness class discarded) $\lor$ 0,

3. $C_3 = (C_2$ with one connectedness class discarded) $\lor$ 0,

4. et cetera.

An example is shown in Figures V-1, 2 and 3. Note that $y_3$ is logically independent of $y_2$, indicating parallel connection of the second and third components.
Figure V-1. An intransitive permutation machine

Figure V-2. Tree diagram for the machine of Figure V-1

Figure V-3. Components from Figure V-2
In this section we explore the relations between a transitive permutation machine and the group generated by the action of its inputs on its states.

The state table of a transitive permutation machine is shown in Figure V-4. The set of all permutations in its group can be exhibited by forming the state table of an extended machine whose inputs are input sequences of the original machine. Such a table is shown in Figure V-5.

The Cayley table for the group of an n-state permutation machine can be approached by connecting n copies of the extended machine in parallel and forming the resulting state table. Such a state table is shown in Figure V-6. Compare its form with that of the Cayley table for the group as shown in Figure V-7.

The Cayley table of Figure V-7 can be viewed as the state table of a finite state machine. This machine will be called the group accumulator. If we append to this group accumulator the output network shown in Figure V-8, then the output assigned to each state is just the image of 1 under the permutation represented by that state. By merging equivalent states we get Figure V-9 from Figure V-8, and Figure V-9 has the same form as Figure V-5. Thus we have imitated the extended machine in Figure V-5 by appending an appropriate output network to the accumulator of its group.

It is an easy exercise in group theory to show that this can always be done, that is, any transitive permutation machine can
Figure V-4. A transitive permutation machine

<table>
<thead>
<tr>
<th>x</th>
<th>y1</th>
<th>y2</th>
<th>y3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2  2</td>
<td>3 1</td>
<td>1 3</td>
</tr>
</tbody>
</table>

Figure V-5. The extended state table for the machine of Figure V-4
\[
\begin{array}{cccccc}
1^3 & 1^2 & 1 & 1^3_2 & 1^2_2 & 12 \\
\hline
y_1y_2y_3 & 123 & 123 & 312 & 231 & 213 & 321 & 132 \\
312 & 312 & 231 & 123 & 321 & 132 & 213 \\
231 & 231 & 123 & 312 & 132 & 213 & 321 \\
213 & 213 & 132 & 321 & 123 & 231 & 312 \\
321 & 321 & 213 & 132 & 312 & 123 & 231 \\
132 & 132 & 321 & 213 & 231 & 312 & 123 \\
\end{array}
\]

\[Y_1 Y_2 Y_3\]

\[
\begin{array}{c}
x \\
\hline
\rightarrow y_1 \\
\rightarrow y_2 \\
\rightarrow y_3
\end{array}
\]

starting state = 1
starting state = 2
starting state = 3

Figure V-6. Three extended machines in parallel

\[
\begin{array}{cccccc}
a & b & c & d & e & f \\
\hline
a & a & b & c & d & e & f \\
b & b & c & a & e & f & d \\
c & c & a & b & f & d & e \\
d & d & f & e & a & c & b \\
e & e & d & f & b & a & c \\
f & f & e & d & c & b & a \\
\end{array}
\]

Figure V-7. The Cayley table for the group of the machine of Figure V-6
<table>
<thead>
<tr>
<th>permutation</th>
<th>output</th>
</tr>
</thead>
</table>

Figure V-8. Figure V-7 rearranged, with output network

![Figure V-8](image)

**Figure V-8.** Figure V-7 rearranged, with output network

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure V-9.** Figure V-8 after merger of equivalent states
be imitated by appending an appropriate output network to the accumulator of its group. The idea is to pick some state \( y \) of the transitive machine and append to the accumulator the output function that maps each permutation \( a \) into \( (y)a \). Then the partition of accumulator states into equivalence classes is identical with a right coset decomposition of the group by the subgroup that leaves \( y \) fixed. Finally check that each input permutes the cosets in the same way that it permutes the states of the transitive machine. (For more detail see Marshall Hall, *The Theory of Groups*, MacMillan, pp. 56-58.)

A surprising consequence of the above construction is that any transitive permutation machine can be imitated by a parallel combination of several copies of any extended permutation machine having the same group. To accomplish this imitation, build the group accumulator out of several copies of the extended machine, as was done in Figure V-6, then imitate the extended behavior of the other machine with the accumulator as was done in Figure V-8.

Thus as far as ordered state assignments are concerned, extended transitive machines having the same group are equally powerful components. Thus it makes no sense to talk about which transitive permutation machines are necessary components in an ordered state assignment. We can only talk about which groups are necessary.
be imitated by appending an appropriate output network to the accumulator of its group. The idea is to pick some state $y$ of the transitive machine and append to the accumulator the output function that maps each permutation $a$ into $(y)a$. Then the partition of accumulator states into equivalence classes is identical with a right coset decomposition of the group by the subgroup that leaves $y$ fixed. Finally check that each input permutes the cosets in the same way that it permutes the states of the transitive machine. (For more detail see Marshall Hall, *The Theory of Groups*, MacMillan, pp. 56-58.)

A surprising consequence of the above construction is that any transitive permutation machine can be imitated by a parallel combination of several copies of any extended permutation machine having the same group. To accomplish this imitation, build the group accumulator out of several copies of the extended machine, as was done in Figure V-6, then imitate the extended behavior of the other machine with the accumulator as was done in Figure V-8.

Thus as far as ordered state assignments are concerned, extended transitive machines having the same group are equally powerful components. Thus it makes no sense to talk about which transitive permutation *machines* are necessary components in an ordered state assignment. We can only talk about which *groups* are necessary.
A STANDARD ORDERED STATE ASSIGNMENT FOR PERMUTATION MACHINES

We shall now develop a standard ordered state assignment that reduces any permutation machine (transitive or intransitive) to components whose groups are small.

Consider the machine whose state table is shown in Figure V-9 and whose group is shown in Figure V-7. Observe that if we lump together the even permutations a, b, c and the odd permutations d, e, f we can form the abbreviated Cayley table shown in Figure V-10. (This is an example of a homomorphism on the group of Figure V-7. The even permutations form the normal subgroup that is the kernal of the homomorphism.) We shall use the above fact to develop an ordered state assigned version whose block diagram is shown in Figure V-11. The second component will attempt to imitate the original machine using only even state permutations. Whenever this component is in error, the output network will give the correct state by translating the state of the second component according to a correction permutation stored in the first component. The correction permutations will be a, the identity (even) permutation, and d, an odd permutation (coset leaders). The action of any input x on a state $y_1$ of the first component is determined by the rule, $y_1 (\text{proj}_1 M_x \text{proj}_1) = a \text{ or } d$ according as $y_1 M_x$ is even or odd. The ordered state assignment developed according to this scheme for the machine of Figure V-9 is shown in Figure V-12. The resulting components are simpler than the original machine because the first
Figure V-10. An abbreviated form of Figure V-7

Input

First component has 2 states, a and d.

Second component has same states as original machine, but only even permutations.

Output network translates the state of the second component according to the permutation stored in the first component.

Output

Figure V-11. Block diagram for the ordered state assigned version for the machine of Figure V-9.

\[
\begin{array}{cccccccc}
\times & a & b & c & d & e & f & z \\
\hline
y_1 y_2 & a_1 & a_3 & a_2 & d_1 & d_3 & d_2 & 1 \\
a_2 & a_2 & a_1 & a_3 & d_2 & d_1 & d_3 & 2 \\
a_3 & a_3 & a_2 & a_1 & d_3 & d_2 & d_1 & 3 \\
d_1 & d_1 & d_2 & d_3 & a_1 & a_2 & a_3 & 2 \\
d_2 & d_2 & d_3 & d_1 & a_2 & a_3 & a_1 & 1 \\
d_3 & d_3 & d_1 & d_2 & a_3 & a_1 & a_2 & 3 \\
\end{array}
\]

Figure V-12. State table for the machine of Figure V-11
component generates only the two-element group and the second component generates the cyclic group of even permutations of three objects, while the original machine generates the group of all permutations of three objects.

The above state assignment worked because the set of all even permutations of three objects is a normal subgroup of the set of all permutations of three objects. In general, if we are given a permutation machine \((Q, I, O, M, V)\) whose group \(G\) has a normal subgroup \(H\) and factor group \(G/H\), then we have the ordered state assigned version shown in Figure \(V-13\). The state set \(Q_1\) of the first component is a set of coset leaders in a right coset decomposition of \(G\) by \(H\), and the state set \(Q_2\) of the second component = \(Q\). To specify the state assignment we specify \(Z\) and \(M_x^*\) as follows:

1. \(Z\) maps \((y_1, y_2)\) into the image of the state \(y_2\) under the coset leader \(y_1\) as shown in Figure \(V-14\).

2. \(M_x^*\) acts on the first component according to multiplication in \(G/H\), i.e. if \(y_1\) is the present state of the first component and \(y_1\) is the next state under input \(x\), then \(y_1 \cdot M_x^*\) is the leader of the coset that contains \(y_1 M_x^*\).

We now have to show that the operation of the second component is uniquely determined by 1. and 2. above, and that every state transformation of the second component is an element of \(H\). Suppose the second component is in state \(y_2\) and it receives the input \((y_1, x)\).
Figure V-13. Block diagram of the standard ordered state assigned version of a permutation machine

Figure V-14. The requirements on Z and $M_x^*$
The present state of the machine being imitated would be \((y_2) y_1\); its next state would be \((y_2) y_1 M x_1\). Since the next state of the first component is \(Y_1\), correct operation requires that the next state \(Y_2\) of the second component be \((y_2) y_1 M x_1 Y_1^{-1}\). Thus under the second component input \((y_1 x)\) the second component state transformation \(Y_1 M x_1 Y_1^{-1}\) is uniquely determined. Finally we show that \(y_1 M x_1 Y_1^{-1}\) belongs to \(H\). Since, by 2. above, \(Y_1\) is the leader of the coset that contains \(y_1 M x_1 Y_1^{-1}\), there is an \(h\) in \(H\) for which \(h Y_1 = y_1 M x_1 Y_1^{-1}\). This \(h\), therefore, \(= y_1 M x_1 Y_1^{-1}\).

In the next section we shall see that the exhaustive application of the procedure described in this section gives a unique set of smallest component groups for any permutation machine.
V-5 COMPOSITION FACTORS

We shall call the technique described in the previous section the standard construction of a two-component ordered state assignment for a permutation machine. Since it can be used on any permutation machine whose group has a proper normal subgroup, exhaustive application of the standard construction to a permutation machine gives an ordered state assigned version each of whose component groups has no proper normal subgroup, i.e. is simple. Call the resulting state assigned version a standard form for the permutation machine.

We first show that all standard forms for a given permutation machine have the same set of component groups. Any standard form for a permutation machine with group $G$ is based on a sequence $G, H_1, H_2, \ldots, H_{k-1}$ of groups in which each group is a maximal normal subgroup of the preceding group. Such a series is called a composition series for $G$; the Jordan-Hölder theorem states that the factor groups $G/H_1, H_1/H_2, \ldots, H_{k-2}/H_{k-1}$, called composition factors for $G$, are unique up to rearrangement. In a standard form for a permutation machine with group $G$, the component groups are exactly these composition factors; so the component groups in a standard form are unique up to rearrangement.

We next show that a standard form has the smallest set of smallest component groups of any ordered state assignment for a given permutation machine. We shall consider an arbitrary state assigned version $A^*$ of a permutation machine $A$. We shall first
show that \(A^*\) can be modified slightly to give a machine \(A^{**}\) whose
component groups are 1) no larger than those of \(A^*\), and 2) the
composition factors of the group of \(A^*\). Then we shall show that
every composition factor of the group of \(A\) is also a composition
factor of the group of \(A^*\).

We compare 1) an arbitrary, 2-component, ordered compound,
permutation machine \(A^* = (Q^*, I, O, M^*, V^*)\) having group \(G^*\) and
first-component group \(G_1\), with 2) that 2-component version \(A^{**}\)
of \(A^*\) having first-component group \(G_1\) and resulting from the
standard construction. That is, since the mapping \(h\) that takes
every \(M^*_r\) of \(A^*\) into \(\text{proj}_1 M^*_r \text{proj}_1\) is a homomorphism on \(G^*\), this
homomorphism can be used to give a 2-component version of \(A^*\) by
the standard construction. Of course, \(A^{**}\) has the same first-
component group as \(A^*\), but the first component may have more states
in \(A^{**}\). This possible proliferation of states is traded for a
possible reduction in the size of the second-component group.

Indeed, if \(K\) is the group of all those \(M^*_r\) in \(A^*\) that produce the
identity permutation in the first component, then each \(M^*_r\) in \(K\)
must produce a distinct permutation on the second component (since
all these \(M^*_r\) are distinct elements of \(G^*\)). Therefore the group
\(G_2\) of the second component of \(A^*\) contains \(K\) as a subgroup. In
\(A^{**}\), however, the group of the second component just equals \(K\),
since \(K\) is the kernel of the homomorphism \(h\). By iteration of this
idea we see that if \(A^*\) is a many-component, ordered compound machine

78.
then the standard construction can be used to give a version of
A* with component groups no larger than those of the original
version.

We now use the above fact to show that the standard con-
struction gives the smallest component groups of any ordered state
assignment for a given permutation machine. Let A be the given
machine and let A* be any ordered state assigned version for A
whose component groups are simple. By using the standard con-
struction as above, we make a new version A** of A* whose
component groups are at most as large as those of A*. By this
construction the component groups of A** are just the composition
factors of G*, the group of A*. Next we show that every composition
factor of G, the group of A, is also a composition factor of G*.
This follows from the fact that the mapping h* that takes each
M* into \( \tilde{Z} \) (where Z is the assignment function) is a homomorphism
from G* onto G. Let H be its kernel and refine the series G*,H
to a composition series for G*. Then the composition factors
from G* down to H are just the composition factors of G. But
the composition factors of G are just the component groups in
any standard form for A.
V-6 DISCUSSION

In the preceding section we saw that the standard construction leads to a unique set of smallest component groups for every permutation machine. As a corollary we see that any permutation machine whose group is simple has no ordered state assignment all of whose component groups are smaller than the group of the original machine. Thus any permutation machine has at least one ordered state assignment in which the components are either identity machines or transitive machines whose (simple) groups are the composition factors of the group of the original machine. Call such an ordered state assignment minimal. It is minimal as far as groups are concerned since any other ordered state assignment has components with at least as many distinct composition factors. On the other hand, there may be many minimal ordered state assignments for a given machine varying widely in 1) the number of components having a certain group and 2) the number of states per component. It is possible to reduce the number of states per component by increasing the number of components having a given group. To do this, build each transitive component as a parallel connection of several copies of the machine with a minimum number of states that has the same group. This puts lower bounds on the numbers of states in the components. The remaining problem, that of bounding the number of components of each type needed for a given permutation machine, seems much more difficult. J. L. Rhodes has suggested that some progress in this direction might be made using fairly deep theorems on group representations (see Marshall Hall, op. cit., chapter 14).
VI. COMPONENTS FOR ARBITRARY MACHINES

VI-1 A STANDARD FORM FOR ARBITRARY MACHINES

In the previous chapter we saw that identity machines plus transitive permutation machines whose groups are simple form a complete set of components, under loop-free interconnection, for permutation machines; for each permutation machine the simple groups are uniquely determined. In this chapter we show that by adding resets to the identity machines, we get a complete set of components for all machines; again, for each machine the simple groups are uniquely determined. As in the permutation case, we shall demonstrate the result by means of a standard form for finite state machines.

To construct the standard form for a given machine, we first reduce it to P-R components by the method of section IV-5. Remember that each resulting component group is a homomorphic image of some subgroup of the semigroup of state transformations of the machine. Second, we reduce each component to subcomponents whose groups are simple, using the procedure of section V-5. This gives a standard form for each component in which, for each state of that machine consisting of the first k-l subcomponents, there is one state of the final subcomponent for each state of the original component. Therefore the resets can be put back in by having each one reset the final subcomponent while leaving the other subcomponents alone (i.e. producing an identity permutation.
on them). Third, we split any components whose groups are intransitive by the method of section V-2. So far we have reduced the given machine to components whose groups are simple and transitive; some of the components have resets. Fourth, we can separate the resets from the permutations in each P-R component by applying the standard construction of section V-4 using the normal subgroup consisting of only the identity. When the resets are put back in at the end of this step, they are added to a machine whose only previous input was the identity permutation, thus forming an identity machine with resets. An example of this last step is shown in Figure VI-1.

Thus any finite state machine has at least one ordered state assignment in which the components are identity machines with resets and transitive permutation machines whose groups are simple. The set of component groups that appear in the standard form is a subset of the set $U$ of all simple groups that are composition factors of subgroups of the semigroup of state transformations of the machine. Since the extended behavior of any permutation machine with group $H$ can be imitated by any extended permutation machine whose group has a subgroup isomorphic to $H$ (section V-3), we can trim $U$ by discarding those groups that are isomorphic to subgroups of other members of $U$. The resulting set $U_{\text{max}}$ comprises sufficient simple groups for the loop-free construction of the given machine. In the next section we shall show that these groups are also necessary.
Figure VI.1. Separating permutations from resets
NECESSARY COMPONENT GROUPS

Given any machine $A$ we are going to produce a set of simple groups that must be represented in any ordered state assigned version $A^*$ whose components either are identity machines with resets or have simple groups.

To do this we derive from $A$ a new machine $A_F$ by the following steps:

1. Let $S$ be the semigroup generated by the action of $A$'s inputs upon its states. Form the extended machine of $A$, whose individual inputs correspond to input sequences of $A$, with one input to the extended machine for each element of $S$.

2. Pick a set of inputs to the extended machine that generate a subgroup $F$ of $S$, and discard all other inputs.

3. Delete all states not belonging to the common range of all the inputs retained.

The resulting $A_F$ is a permutation machine whose group is $F$.

Hence the unique set of smallest component groups for $A_F$ is made up of the composition factors of $F$. But any ordered state assigned version of $A$ can be converted to one of $A_F$ by applying steps 1, 2, and 3, none of which enlarge any component groups. Hence if $W$ is a set of components that are sufficient to build $A$, then for each $F$, every composition factor of $F$ must be a subgroup of some member of $W$; $W$ contains $U_{\text{max}}$, so all the groups in $U_{\text{max}}$ are necessary. Thus for each finite state machine, we have identified the unique
set of simple groups necessary and sufficient for its loop-free construction.

The results of the preceding paragraph constitute the main theorem of Rhodes and Krohn. It does not tell us how many components with a given group will be needed; if one of the groups in Umax has a simple subgroup, the theorem does not say whether a component having that subgroup is needed or not; the theorem does give us a hint at the minimum numbers of states needed in the components. The identity machines with resets can be built out of two-state components (any binary state assignment will do). For each necessary component group G, there is a minimum number of objects needed for a permutation representation of G (see section V-6), but this number is easy to find only in the case mentioned below.

For most machines encountered in practice, Umax contains only cyclic simple groups. In this case all the objections, except the first one, voiced in the preceding paragraph disappear: cyclic simple groups have no proper subgroups; each cyclic simple group has just one transitive representation.
VI-3 APPLICATIONS

When do the decomposition methods described in this paper provide practical engineering tools for the synthesis of sequential switching circuits? Only in a synchronous system would we even consider using loop-free synthesis, for the timing problems in an asynchronous cascade of sequential circuits would be prohibitive.

For synchronous systems, the synthesis from P-R machines described in chapters I - IV seems attractive. Although somewhat wasteful of equipment, it gives a system whose operation is easy to understand since we can think about it one component at a time. The permutation and reset operation of each component is in good agreement with the way we think about sequential circuits intuitively.

Further decomposition of the P-R components by the methods of section V-4 is less attractive because it can result in such a great proliferation of states. Things are not quite so bad as they seem. This decomposition leads to machines whose groups are simple, and there aren't too many such groups. In fact, the vast majority of simple groups consists of cyclic groups of prime order, whose machines are merely counters. When a P-R machine can be built from counters with resets, the proliferation of states is not too serious and the benefits in terms of understandability are great.
The use of our methods on incompletely specified machines presents no difficulties. In fact, loop-free decomposition seems to provide some of the best clues on how to fill in unspecified entries.

All the above remarks apply to machines whose state tables are given. For machines whose operation is described in some other way (like English), the possibility of describing the machine at the outset in terms of a loop-free interconnection of P-R components merits serious consideration. The conversion of an English statement to a P-R operation is often easy. For example, a machine that tells the parity of the number of 1 inputs since the last 0 input is just a 2-state P-R machine with 1 producing a permutation and 0 a reset. A machine whose state table is too large to write down might be described fairly compactly as a loop-free interconnection of P-R machines. Once the machine has been described, state reduction on it might be carried out by a computer. Then the computer might convert the reduced machine back to a loop-free interconnection of P-R components by the methods already described. The designer could then choose between his representation and the computer's.
BIBLIOGRAPHY

A. Finite State Machines


B. Mathematics


BIOPGRAPHICAL NOTE

Paul Zeiger was born in Niagara Falls, New York, on November 12, 1936. He received a B.S. degree in Electrical Engineering from Massachusetts Institute of Technology in 1958, and an S.M. degree from M.I.T. in 1960.

Mr. Zeiger held a William Barton Rogers scholarship while an undergraduate at M.I.T. He was elected to Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. He was a research assistant at the Research Laboratory of Electronics at M.I.T. during 1958-1960 and 1962-1964.

Mr. Zeiger has been employed by the Electro Metallurgical Company, U. S. Corps of Engineers, and Collins Radio Corporation. He spent a two-year tour of duty as a Lieutenant at Fort Monmouth, New Jersey, where his duties included research and teaching.