STAN-CS-70-170 SU-SEL-70-024

Analysis and Synthesis of Concurrent Sequential Programs

by

T. H. Bredt

May 1970

Technical Report No. 6

This work was supported in part by the Joint Services Electronic Programs U.S. Army, U.S. Navy, and U.S. Air Force under Contract N-00014-67-A-0112-0044 and by the National Aeronautics and Space Administration under Grant 05-020-377.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

STAN-CS-70-170 SEL-70-024

ANALYSIS AND SYNTHESIS OF CONCURRENT SEQUENTIAL PROGRAMS

by

T. H. Bredt

May 1970

Technical Report No. 6

DIGITAL SYSTEMS LABORATORY

Stanford Electronics Laboratories Computer Science Department

Stanford University

Stanford, California

This work was supported in part by the Joint Services Electronic Programs U.S. Army, U.S. Navy, and U.S. Air Force under Contract N-00014-67-A-0112-0044 and by the National Aeronautics and Space Administration under Grant 05-020-337.

STANFORD UNIVERSITY
Digital Systems Laboratory
Stanford Electronics Laboratories   Computer Science Department

Technical Report Number 6
May, 1970

ANALYSIS AND SYNTHESIS OF CONCURRENT SEQUENTIAL PROGRAMS

by

T. H. Bredt

ABSTRACT

This paper presents analysis and synthesis procedures for a class of sequential programs. These procedures aid in the design of programs for parallel computer systems. In particular, the interactions of a given program with other programs or circuits in a system can be described precisely. The basis for this work is a model for parallel computer systems in which the operation of each component is described by a flow table and the components interact by changing values on interconnecting lines. The details of this model are discussed in another paper*. The analysis procedure produces a flow table description of a program. In program synthesis, a flow table description is converted to a sequential program. Using flow table design procedures, a control program for the two-program mutual exclusion problem is produced.

* Bredt, T.H. and McCluskey, E.J. A model for parallel computer systems. Technical Report No. 5, SEL Digital Systems Laboratory, Stanford University, Stanford, California (April 1970).

TABLE OF CONTENTS

ABSTRACT i

TABLE OF CONTENTS ii

LIST OF TABLES iv

LIST OF FIGURES v

INTRODUCTION 1

ANALYSIS OF PROGRAMS 5

Programs of the Class $\mathcal{P}$ 8

Control Assignment Statement 9

Wait Statement 10

Go To Statement 12

Dummy Statement 12

Halt Statement 13

State Tables and Programs 15

Preliminaries 15

Determination of Immediate Successor Output States . . . 18

Determination of Immediate Successor Row Number 18

Determination of Next-State Entries 19

Examples. 19

State Tables and Flow Tables 23

Inaccessible States and Inaccessible Sets of States . . . 25

Indistinguishable State Tables 25

TABLE OF CONTENTS (Continued)

SYNTHESIS OF PROGRAMS 33

Procedure to Obtain a Program of the Class $\mathcal{P}$ for Any Flow Table 36

CONCLUSIONS 43

ACKNOWLEDGEMENTS 44

REFERENCES 45

LIST OF TABLES

1. Flow Tables for the Two-Process Mutual Exclusion Problem . . . 4

2. General Form of a Flow Table 7

3. Example Program of Class $\mathcal{P}$ 14

4. Statement Forms and Next Statement Numbers 17

5. Program of Table 3 With State Table 20

6. Example of a Program 22

7. Construction of the State Table for the Program of Table 6 . . 22

8. State Table for the Program of Table 6 24

9. Example of Inaccessible States 26

10. Example of an Inaccessible Set of States 26

11. Example of a Program and State Table With a Cycle 28

12. Flow Table for State Table in Table 5 32

13. Flow Table for the State Table of Table 8 and Program of Table 6 34

14. Control Program for the Two-Program Mutual Exclusion Problem . 38

15. Another Control Program for the Two-Program Mutual Exclusion Problem 40

16. Program 1 Without Dummy Statements 41

17. Program 1 With Dummy Statements 42

LIST OF FIGURES

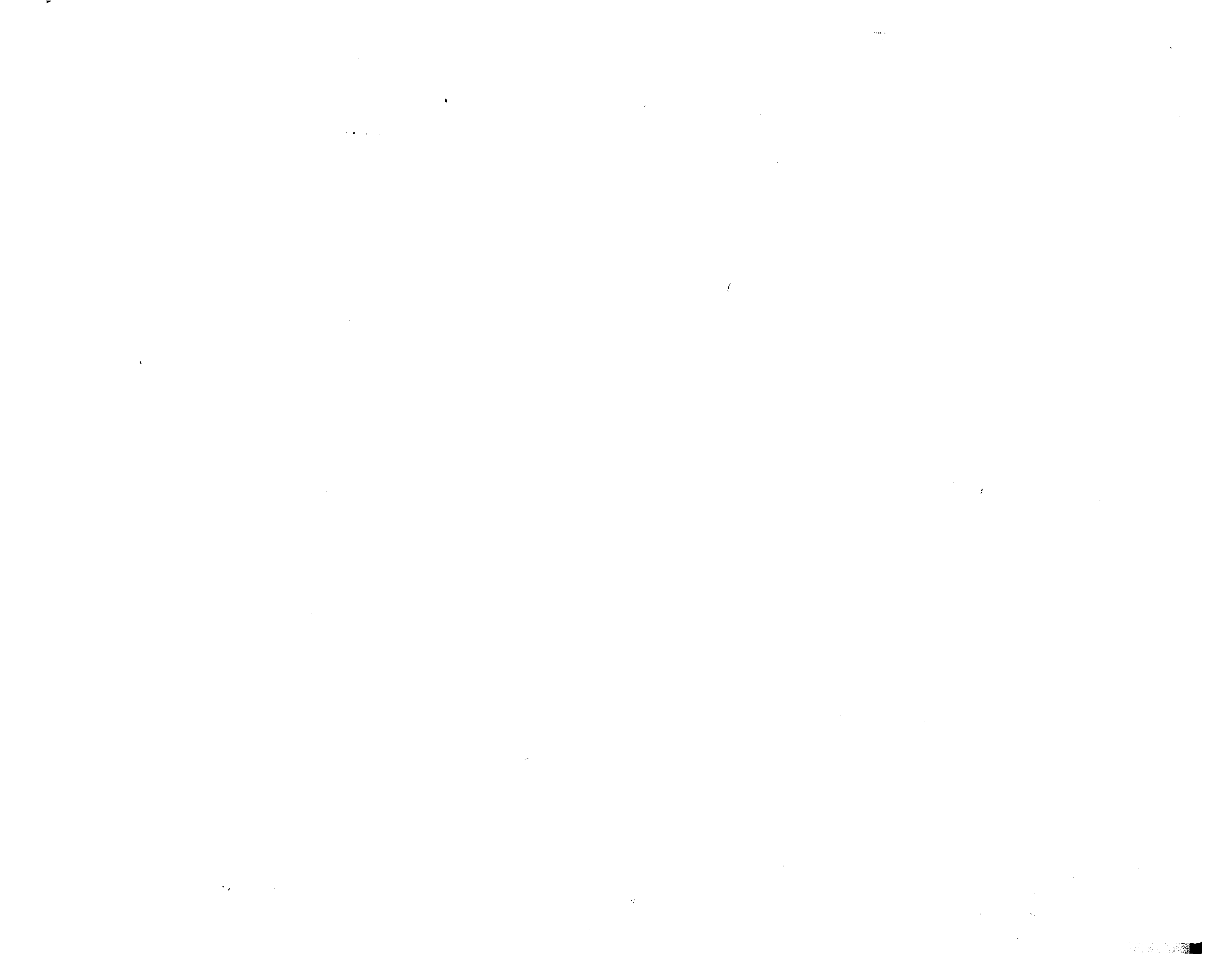
1. Parallel system configuration for the two-process mutual exclusion problem 3

2. General form of a system component 6

3. Cyclic transitions for the state table in Table 11 28

4. Example of a state table with a cycle 30

5. Another example of a state table with a cycle 30



INTRODUCTION

In another paper [1], a model for parallel computer systems was proposed. This model provides a basis for the formal study of program and circuit interaction in computer systems. The motivation for the model is the desire to solve control problems such as the mutual exclusion or interlock problem, which has been studied by Dijkstra and many others [2, 3, 4, 5, 7, 9]. Its solution requires the control of two or more programs which are executed concurrently and contain special functions enclosed in "critical sections". It is necessary to ensure that (1) at most one program is executing a critical section at any instant and (2) if a program wants to enter a critical section, it is eventually allowed to do so.*

In our model flow tables [13] are used to describe the behavior of each component (program, circuit) in a system. Components interact by changing the values on lines which interconnect them. These lines carry binary level signals. The use of lines for intercomponent communication differs from the more common use of shared memory cells [2 , 3 , 4 , 5 , 9]. Our model is on a more primitive level and can be used to describe implementations in which memory cells are shared. It is assumed that the delays in the interconnecting lines are finite and unbounded. Component internal delays are assumed to be finite and bounded.

---

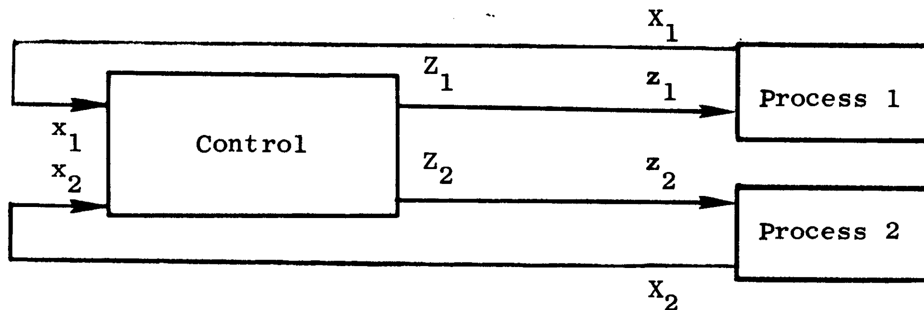
* This is a slightly different version of the problem considered by Dijkstra. Dijkstra did not require that a given program must enter its critical section but rather that the decision as to which program would enter its critical section could not be postponed indefinitely.

In this paper, analysis and synthesis procedures are defined which make it possible to relate program implementations with the flow table model. These procedures are applied in the analysis and synthesis of programs for use in the mutual exclusion problem. A configuration for a parallel system which is appropriate for a discussion of the mutual exclusion problem was specified in [1]. This configuration is reproduced in Fig. 1.* In [1], flow table descriptions for the three components in the system were produced. These flow tables are shown in Table 1.

In [1], a special mode of operation was defined for the components in a system. Each component operates independently of other components using the following operation cycle. First, the present input state of the component is determined and recorded in a rank of flip-flops. Second, the input state and the present internal state of the component determine the component response. When the response is complete, the input state is determined again and the cycle repeats. With this mode of operation, component inputs may change at any time without adverse effects. The flow tables, such as those shown in Table 1, specify the operation of the component during the second phase discussed above.

The sequential programs studied in this paper contain only a limited subset of the facilities available in high-level programming languages such as ALGOL or FORTRAN. The limitations have been made to facilitate the description of the analysis and synthesis procedures. It is reasonable to make these limitations because we are interested

* Lower case denotes input variables and upper case, output variables.



$x_i = 1$	Process i wants to enter critical section i (CSi) <u>or</u> process i is in CSi.
$x_i = 0$	Process i does not want to enter CSi <u>and</u> process i is not in CSi.
$z_i = 1$	Process i may enter CSi.
$z_i = 0$	Process i may not enter CSi.

Figure 1. Parallel system configuration for the two-process mutual exclusion problem.

Table 1. Flow Tables for the Two-Process Mutual Exclusion Problem

(a) Process 1

	x_1 = 0	x_1 = 1	z_1
1	2	(1)	0
2	(2)	1	1

(b) Process 2

	x_2 = 0	x_2 = 1	z_2
1	2	(1)	0
2	(2)	1	1

(c) Control

		x_1x_2 = 00	01	11	10	z_1z_2
(2 last)	1	(1)	2	3	3	00
(2 gets)	2	1	(2)	(2)	3	01
(1 gets)	3	4	2	(3)	(3)	10
(1 last)	4	(4)	2	2	3	00

(Initially each component is in internal state 1)

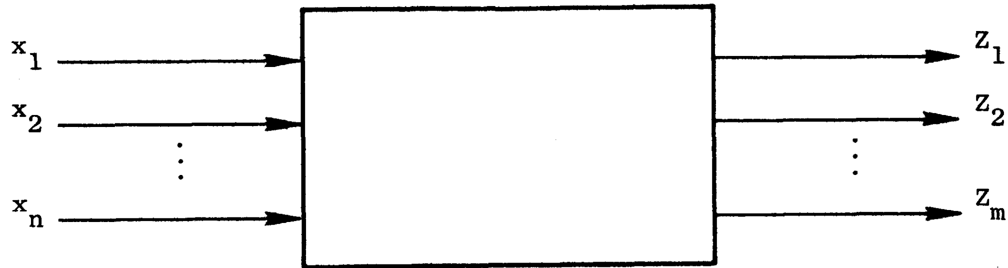
in the interactions of programs with other programs and circuits and not in the analysis of the computations carried out by the programs. In this regard, our work has quite a different emphasis from that of Floyd [6], Knuth [9], and Manna [10 , 11 , 12].

We begin with a discussion of how sequential programs can be analyzed to produce a flow table description of their operation. Later, synthesis, the specification of a sequential program from a flow table description, is considered.

ANALYSIS OF PROGRAMS

The general form for components is shown in Fig.2. Interaction is accomplished by changes in input and output variable values. The analysis procedure consists of constructing first a state table and then a flow table as shown in Table 2. If the table entry is the same as the row number, the entry is said to be stable; otherwise, it is unstable. For a flow table, we require that every unstable entry specify a stable entry. A state table has the same form as a flow table except that it is not necessary for this condition to be satisfied. The analysis method is analogous to the procedures used for sequential circuits [13].

Each program is assumed to be a sequential program, a program in which only a single instruction or statement is executed at a time. The execution of a program begins with the first statement. The next statement to be executed may be the statement following the one



$x_1, x_2, \ldots, x_n$ are input variables (lower case)

$Z_1, Z_2, \ldots, Z_m$ are output variables (upper case)

Figure 2. General form of a system component.

Table 2. General Form of a Flow Table

Input State

$x_1x_2 \ . \ . \ . \ x_n$

Output State

Internal State			00...1		11...1			$Z_1Z_2\ldots Z_m$
i			(i)		j			10 . . . 0
j					(j)			11 . . . 0
r								

S (next state)

just executed or a statement elsewhere in the program at a location determined by the current values of the input variables or by conditional and branching statements which use control information internal to the program such as the values of internal program variables. Internal control information can be included in a flow table by adding extra rows for the possible internal control states. This can be done as long as the control can be described by finite means. The addition of internal control information complicates the flow table analysis. Since we are interested in the interactions of programs with other components, the types of internal control allowed will be limited. We assume that programs can be put in a form where the next statement to be executed is either the statement following the one just executed or is specified by a "go to" statement or determined by the input state of the component.

Programs of the Class $\mathcal{P}$

A precise description of the class of programs to be considered follows.

Definition 1:

A program of the class $\mathcal{P}$ is a finite sequence of statements of the following types:

a. Control Assignment

b. Wait

c. Go to

d. Dummy

e. Halt

The statements may be numbered with integer labels. Each such number is followed by a colon (:). Statements are separated by semi-colons. The last statement is followed by a period. Each program includes <u>declarations</u> of the input and output variables. The declarations have the form:

INPUT $X_1, X_2, \ldots, X_n$ where X_i is an input variable

OUTPUT $Z_1, Z_2, \ldots, Z_m$ where Z_i is an output variable

The initial values of the output variables (the initial output state) must be specified for each program.

The format and interpretation of the statement types are as follows.

<u>Control Assignment Statement</u>. The control assignment statement is used to assign a binary value, either 0 or 1, to an output variable. This statement has the form:

variable := value

The following are examples of control assignment statements.

$X := 1$

$Z_2 := 0$

Subscripts may be used on variables to keep clear the correspondence between programs and flow tables. When a control assignment statement is executed, the program component must include suitable mechanisms to maintain the designated value on the output line.

Wait Statement. The wait statement is used to test the present input state of a program component and to transfer to the appropriate next statement when a designated input state is recognized. This statement has the form:

$$\text{WAIT } (s_1,i_1),(s_2,i_2),\ldots,(s_k,i_k)$$

Each pair (s_j,i_j) consists of an input state s_j and a statement number i_j. The input state is represented by a binary number $b_1b_2\ldots b_n$ where b_i is the value of the input variable x_i, $b_i = 0$ or 1, for $i = 1,\ldots,n$. The statement number specifies a statement in the program. Examples of wait statements are given below:

(single input variable)

WAIT (0,3)

(two input variables)

WAIT (01,4),(11,7)

Each input state must appear at most once in a wait statement. When a wait statement is executed, the current input state must be determined. This is done in the manner described in [1]. If the present input state is the same as one of the specified input states, the number of the next statement is given by the statement number in the input-state number pair. If the input state does not appear in the wait statement, the execution of the program must be suspended. The first wait statement example given above is used with a component with one input variable, say X_1. If X_1 has the value 0 when the statement is executed, statement number 3 is executed next; otherwise, execution of the program is suspended. When execution is suspended, there are

two alternative actions. The processor which is executing the program may loop and repeatedly execute the wait statement until the actual input state of the component matches one of those specified. Looping is not always desirable since a processor is occupied which could be assigned to the execution of another program. A second alternative frees the processor to execute other programs. The suspended program is added to a queue of programs which are waiting for changes in their input states. The interpretation of the control assignment statement must then be modified so that, when a control assignment statement is executed, the input states of waiting programs are determined and any programs which are ready to continue execution are either re-assigned to processors or put in another queue of programs which are ready to resume execution as soon as processors are available. The overhead required for this alternative may be substantial. The creation and testing of queue entries must be made critical sections and protected by mechanisms of the type we are investigating. We will not adopt either alternative but mention them as possible implementations. Both alternatives have been proposed before [5 , 9].

The combination of the control assignment statement and the wait statement provide a means of communication which is similar to the use of wakeup and block functions proposed by Saltzer [14] and discussed by Lampson [9]. The wakeup function corresponds roughly to the control assignment statement and the block function to the wait statement. In Saltzer's approach, each process has an associated work queue and wakeup waiting switch. The block function causes execution of a program to be suspended until some form of enabling

signal is received. The wakeup function provides the enabling signal. The wakeup waiting switch is used to prevent enabling signals from being lost. When a process reaches a point where further progress depends on the arrival of a enabling signal, it performs the following steps [14].

1. Resets the wakeup waiting switch to off.
2. Look in the work queue. If there is an entry continue; otherwise, go to step 3.
3. Call the block function. This function returns if the wakeup waiting switch is on.

When a process wishes to send an enabling signal to another process, the steps are as follows:

1. Make an entry in the work queue of the other process.
2. Call the wakeup function for the process, turning the wakeup waiting switch on.

The two alternative forms of waiting used with the wait statement, a loop or entry in a queue and release of the processor, can be used with the block function.

Go To Statement. The go to statement is the standard statement used for branching in most programming languages. It has the form:

GO TO i

where i is the number of some statement in the program.

Dummy Statement. The purpose of the dummy statement is to gather together those portions of programs which are not essential to interaction with other system components. To an observer, the execution of

this statement represents a delay of unknown duration which has no effect on the input or output variables of the component. In the mutual exclusion problem, the critical section is represented by a dummy statement. This statement has the form:

DUMMY

Halt Statement. This statement is provided to indicate that a process is to be terminated unconditionally. It has the form:

HALT

In the following discussion, the use of the word program will refer to a program of the class $\mathcal{P}$ as just described. An example of such a program is shown in Table 3. The only variables which appear in these programs are input variables and output variables for the component. In the example of Table 3, X_1 is an output variable and Z_1 is an input variable. These variables are global variables defined for the entire system and are associated with the lines which interconnect system components. If a variable is used in both a control assignment statement and a wait statement in the same program, the interconnection must be with the component itself. The primary function of the input and output variables is for inter-process control rather than intra-process control and no examples of this latter use will be given.

Table 3. Example Program of the Class $\mathcal{P}$

```
    INPUT   Z1 ;
    OUTPUT  X1 ;    (initially X1 is 0)
1:  X1 := 0 ;
2:  DUMMY ;
3:  WAIT (0,4) ;
4:  X1 := 1 ;
5:  WAIT (1,6) ;
6:  DUMMY ;
7:  GO TO 1 .
```

State Tables and Programs

We now describe how programs of the class $\mathcal{P}$ can be analyzed to produce a state table. This procedure is analogous to the construction of the state table which describes the operation of a sequential circuit. Each internal state or row in the state table describes the execution of some statement in the program. We assume, in this section, that the statements in a program are numbered in ascending order, starting with 1. We use the Moore model in which an output state is associated with each internal state.

Preliminaries. Given a program with t statements, n input variables, and m output variables, define a state table with t rows, row i corresponding to statement i ($1 \le i \le t$) and 2^n columns, one column for each input state. The initial row (internal state) of the state table is row 1. The output state of row 1 is the initial output state of the program. If the first statement is a control assignment statement, the value assigned to the control variable should appear in the output state for row 1.

To fill in the state table we must specify the output state and next internal state for each row in the table. For a circuit, the output state can be determined once the internal state is known. For a program, the output state is determined first by the initial output state and subsequently by the most recently executed control assignment statement for each output variable. In general, a given statement may have more than one immediate predecessor (statement which is executed immediately before) and the statement may be entered with

different output states. In this case, there must be more than one row in the state table for the given statement, one row associated with the execution of the statement for each possible output state. This problem complicates the procedure for determining the state table.

The procedure to complete the state table can be outlined as follows:

Consider each row of the state table in order.

1. If the output state of the row is completely specified, then for each immediate successor of the corresponding statement:

   a. Determine the immediate successor output state.

   b. Determine the immediate successor row number.

   c. Determine the next-state entries for the row.

2. If the output state of a row is not completely specified, pass over that row and consider it again after all other rows. If after all rows have been processed, there remain rows with unspecified output states, these rows correspond to inaccessible statements, statements which are never executed. Such rows can be deleted from the table.

We now give the details for performing steps a-c above. The statement forms and immediate successor statement numbers for each form are given in Table 4. Before filling in the next-state entries

Table 4. Statement Forms and Immediate Successor Statement Numbers

Form of Statement i	Immediate Successor Statement Number
i: $Z_j := \alpha$ $1 \leq j \leq m$ $\alpha = 0$ or 1	i + 1
i: DUMMY	i + 1
i: HALT	i
i: GO TO t	t
i: WAIT $(s_1,i_1),(s_2,i_2),\ldots,(s_k,i_k)$	$i_1,\ldots,i_k$

for a row, the output state and number of the row corresponding to each immediate successor statement must be determined. This is done in the following manner.

Determination of Immediate Successor Output States. Let statement number w be an immediate successor of statement number i and let the output state of the row corresponding to statement i be completely specified. We wish to determine the output state for the row which corresponds to statement w. This output state is the same as the output state for the row associated with statement number i unless statement w is a control assignment statement

$$w: \quad Z_j := \alpha \qquad 1 \leq j \leq m, \quad \alpha = 0 \text{ or } 1$$

in which case, the value of Z_j in the output state is α .

Having determined this output state, we determine the number of the row corresponding to the immediate successor statement.

Determination of Immediate Successor Row Number. If two output states differ in the value of some output variable which is specified (0 or 1) in both output states, the output states are said to conflict. If the output state determined for the row corresponding to statement number w and the output state for row w do not conflict, row w corresponds to statement w and is given the output state determined above. Otherwise, a row with the conflicting output state must be added to the state table. This row should be tagged with a w to indicate that it corresponds to statement w.

When adding rows, it is unnecessary to add more than one row for a given statement with a given output state. If a state table has no conflicting output states, the number of the row corresponding to a statement is the same as the statement number.

Determination of Next-State Entries. Let i' be the number of the row corresponding to statement number i. If statement i is a control assignment, dummy, go to, or halt statement, each next-state entry in row i' is the number of the row corresponding to the immediate successor statement.

In a row corresponding to a wait statement of the form

$$i:\ \text{WAIT}\ (s_1,i_1),(s_2,i_2),\ldots,(s_k,i_k)$$

the entry in the column for input state s_j $(1 \leq j \leq k)$ is the number of the row corresponding to immediate successor statement i_j. The next-state entries in columns which do not correspond to one of the input states $s_1,s_2,\ldots,s_k$ are stable entries, i'.

Examples. For many programs, conflicting row output states never occur. The program of Table 3 is such a program. This program and its state table, as produced by the procedure just described, are shown in Table 5.

To illustrate the procedure when row output states do conflict, consider the following situation. We wish to control the operation of two concurrent programs, program 1 and program 2, such that

1. Program 1 is enabled whenever it requests to be enabled.

Table 5. Program of Table 3 With State Table

```
      INPUT Z1 ;
      OUTPUT X1 ;    (initially X1 is 0)
  1:  X1 := 0 ;
  2:  DUMMY ;
  3:  WAIT (0,4) ;
  4:  X1 := 1 ;
  5:  WAIT (1,6) ;
  6:  DUMMY ;
  7:  GO TO 1 .
```

a) Program

	z_1 = 0	z_1 = 1	X_1
1	2	2	0
2	3	3	0
3	4	(3)	0
4	5	5	1
5	(5)	6	1
6	7	7	1
7	1	1	1

b) State Table

2. Program 2 may be enabled if and only if program 2 requests to be enabled and program 1 is not requesting to be enabled.

The system configuration for this problem is the same as for the mutual exclusion problem and is shown in Fig. 1. The interpretation of the variable values is the following. If x_i is 1, program i is requesting an enabling signal. Program i is enabled by placing a 1 value on the Z_i line. In different terms, the problem is to guarantee that Z_2 is never set to 1 unless the x_1 input to the control mechanism is 0. Suppose that instead of using flow tables to design a solution to this problem, a program is written directly using control assignment and wait statements. One possible control program is shown in Table 6. To determine if this program controls program 1 and 2 as intended, the corresponding state table will be constructed. The output state for row 6 is specified as 10 when row 3 is filled in. When statement 6 is reached from statement 5, the output state for statement 6 must be $Z_1Z_2 = 11$. Therefore, the output states conflict and an additionsl row must be added to the table (row 12). Row 12 also describes statement 6 but with output state $Z_1Z_2 = 11$. The state table after the completion of row 5 is shown in Table 7a. No more output state conflicts occur until row 12 is reached. The state table at this stage is shown in Table 7b.

Table 6. Example of a Program

(initially Z_1Z_2 = 00)

```
    INPUT X1,X2;
    OUTPUT Z1,Z2;
1:  Z1 := 0;
2:  Z2 := 0;
3:  WAIT (01,4),(11,6),(10,6);
4:  Z2 := 1;
5:  WAIT (00,2),(11,6),(10,10);
6:  Z1 := 1;
7:  WAIT (00,1),(01,8);
8:  Z1 := 0;
9:  GO TO 4;
10: Z2 := 0;
11: GO TO 6.
```

Table 7. Construction of the State Table for the Program of Table 6

		x_1x_2 00	01	11	10	Z_1Z_2
	1	2	2	2	2	00
	2	3	3	3	3	00
	3	3	4	6	6	00
	4	5	5	5	5	01
	5	2	5	12	10	01
	6					10
	7					
	8					
	9					
	10					00
	11					
(6)	12					11

(a)

		x_1x_2 00	01	11	10	Z_1Z_2
	1	2	2	2	2	00
	2	3	3	3	3	00
	3	3	4	6	6	00
	4	5	5	5	5	01
	5	2	5	12	10	01
	6	7	7	7	7	10
	7	1	8	7	7	10
	8	9	9	9	9	00
	9	4	4	4	4	00
	10	11	11	11	11	00
	11	6	6	6	6	00
(6)	12					11

(b)

Row 12 describes the execution of statement 6. This statement is followed by statement 7 in the program. But again, the output states conflict so an additional row (row 13) must be added to correspond to the execution of statement 7 with output state Z_1Z_2 =11. Additional rows must be added for statements 1 , 8 , and 9 as well. The complete state table is shown in Table 8. The analysis of this program will be continued later; however, a careful examination of the state table will reveal that the program is incorrect. Suppose program 2 sets X_2 to 1 and is enabled by the control program (Z_2 is set to 1). The control program enters internal state 5. Now let program 1 set X_1 to 1. The control program next enters internal state 13. But, it is now possible for program 2 to set X_2 to 0 then reset it to 1 and be re-enabled immediately because the control does not set Z_2 to 0 in the case when x_2 goes to 0 when x_1 is 1.

State Tables and Flow Tables

It is always possible to produce a state table which corresponds to every program of the class $\mathcal{P}$, just as it is always possible to produce a state table corresponding to every sequential circuit. However, it is not always possible to reduce this state table to an "equivalent" flow table for circuits or for programs. We now consider under what conditions and how the reduction of a state table to a flow table can be performed. A flow table differs from a state table only in that for a flow table every unstable table entry must specify a next internal state entry that is stable. This condition is not required for state tables. A flow table may reduce the number of internal states and provide a more concise description of component operation.

Table 8. State Table for the Program of Table 6

		x_1x_2 = 00	01	11	10	Z_1Z_2
	1	2	2	2	2	00
	2	3	3	3	3	00
	3	(3)	4	6	6	00
	4	5	5	5	5	01
	5	2	(5)	12	10	01
	6	7	7	7	7	10
	7	1	8	(7)	(7)	10
	8	9	9	9	9	00
	9	4	4	4	4	00
	10	11	11	11	11	00
	11	6	6	6	6	00
(6)	12	13	13	13	13	11
(7)	13	14	15	(13)	(13)	11
(1)	14	2	2	2	2	01
(8)	15	16	16	16	16	01
(9)	16	4	4	4	4	01

Inaccessible States and Inaccessible Sets of States. It is possible that when a state table is constructed some of the internal states are never entered. For circuits, there may be states that are not the initial state and which do not appear as the next-state entry in any row of the state table other than their own row. Such states are called inaccessible states. It is also possible to have an inaccessible set of states which is a set of states which does not include the initial state such that no state of the set is entered from any state not in the set [13]. For programs, the procedure for obtaining the state table is such that all states which are retained may be entered. For example in Table 9, internal states 6 and 7 are inaccessible and are removed. In Table 10, internal states 5, 6, 7, and 8 form an inaccessible set of states and also are removed.

Indistinguishable State Tables. The notion of equivalence we will use is called indistinguishability and is defined as follows [13]:

> Definition 2:
>
> Two state tables are said to be indistinguishable if and only if when both tables are in their respective initial states, any input sequence applied to both state tables results in identical output sequences from both tables.

The state table obtained for a program may indicate that the program produces multiple changes for the value of an output variable during some unstable internal-state transition. In an environment

Table 9. Example of Inaccessible States

```
     INPUT Z; (initially X is 1)
     OUTPUT X;
1:   X := 1;
2:   WAIT (0,3);
3:   DUMMY;
4:   X := 0;
5:   GO TO 1;
6:   X := 1;
7:   GO TO 2 .
```

a) Program

	Z = 0	Z = 1	X
1	2	2	1
2	3	(2)	1
3	4	4	1
4	5	5	0
5	1	1	0
6			
7			

b) State Table

Table 10. Example of an Inaccessible Set of States

```
     INPUT X; (Z is 0 initially)
     OUTPUT Z;
1:   Z := 0;
2:   WAIT (1,3);
3:   Z := 1;
4:   WAIT (0,1);
5:   Z := 0;
6:   WAIT (1,7);
7:   Z := 1;
8:   WAIT (0,5).
```

a) Program

	X = 0	X = 1	Z
1	2	2	0
2	(2)	3	0
3	4	4	1
4	1	(4)	1
5			
6			
7			
8			

b) State Table

where line delays are unbounded, such multiple changes should not be used since it cannot be known if the intermediate output values are recognized. The effects of multiple output changes are discussed in more detail in [1]. For this reason, we assume that each output variable changes value at most once during each unstable transition.

It is possible that the state table obtained for a given program will contain a cycle.

Definition 3:

A state table is said to contain a cycle if for a fixed input state the component represented may operate indefinitely without ever becoming stable.

An example of a program and associated state table containing a cycle is shown in Table 11. The cyclic transitions are shown in Fig. 3. During the cycle, the value of output variable X_1 changes more than once. The program of Table 11 could be used as program 1 in a solution to the mutual exclusion problem; however this should not be done because of the multiple-value transitions during the cycle. A further discussion of the difficulties with this program is given in [1, p.23].

The assumption that output variables change value at most once during unstable transitions allows us to restrict our attention to cycles in which this assumption holds. Associated with each cycle is a set of internal states of the state table which are entered during the indefinite operation which defines the cycle. Call this set C. Since output variables change value at most once during a cycle and

Table 11. Example of a Program and State Table With a Cycle

```
      (initially X1 is 0)
      INPUT Z1;
      OUTPUT X1;
1:    X1 := 0;
2:    DUMMY;
3:    X1 := 1;
4:    WAIT (1,5);
5:    DUMMY;
6:    GO TO 1.
```

a) Program

	z_1 = 0	z_1 = 1	X_1
1	2	2	0
2	3	3	0
3	4	4	1
4	(4)	5	1
5	6	6	1
6	1	1	1

b) State Table

	z_1 = 0	z_1 = 1	X_1
1			0
2			0
3			1
4			1
5			1
6			1

Figure 3. Cyclic transitions for the state table in Table 11.

the states in C are entered an unspecified number of times, all the internal states in C must have the same output state. Two examples of state tables with such cycles are shown in Figs. 4a and 5a. In Fig. 4, if input variable x changes from 0 to 1 during the cycle, the next stable entry is 3. As a result, this state table cannot be distinguished from the flow table shown in Fig. 4b in which the cycle has been replaced by a single stable entry and the internal states renumbered. Unfortunately, the example of Fig. 5 shows that it is not always possible to simply replace cycles by stable entries. In this example, if an x_1x_2:00 ⟶ 11 transition is recognized during the cycle and while the component is in internal state 1, the next internal state will be state 3. If the input transition is recognized while in internal state 4, the next internal state will be state 2. States 2 and 3 have different output states. If the cycle is removed by replacing rows 1 and 4 by a single row, identical to row 1 except for a stable entry in the 00 column, the flow table shown in Fig 5b is obtained. However it is possible for the state table and the flow table to be presented with the 00,11 input sequence and for them to produce different output sequences. Unfortunately, we are as yet unable to characterize in a general way the conditions under which cycles can be replaced by stable entries.

In most cases, the state tables obtained for programs will not contain cycles. Such tables are said to be cycle-free. If a state table is cycle-free and each output variable changes value at most

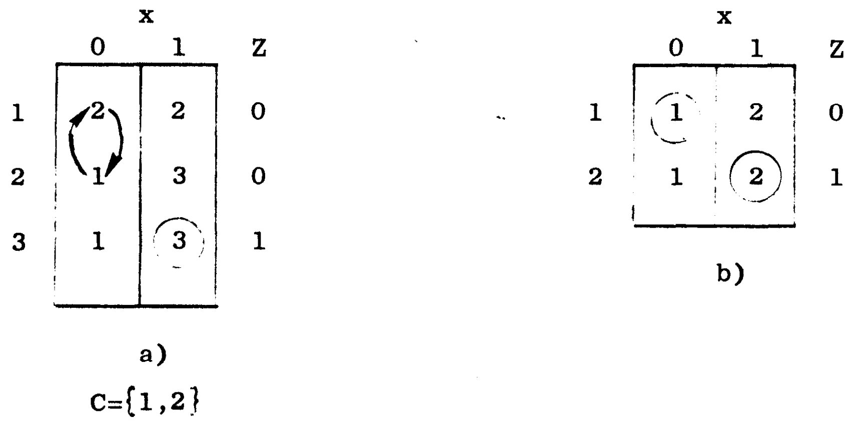


Figure 4. Example of a state table with a cycle.

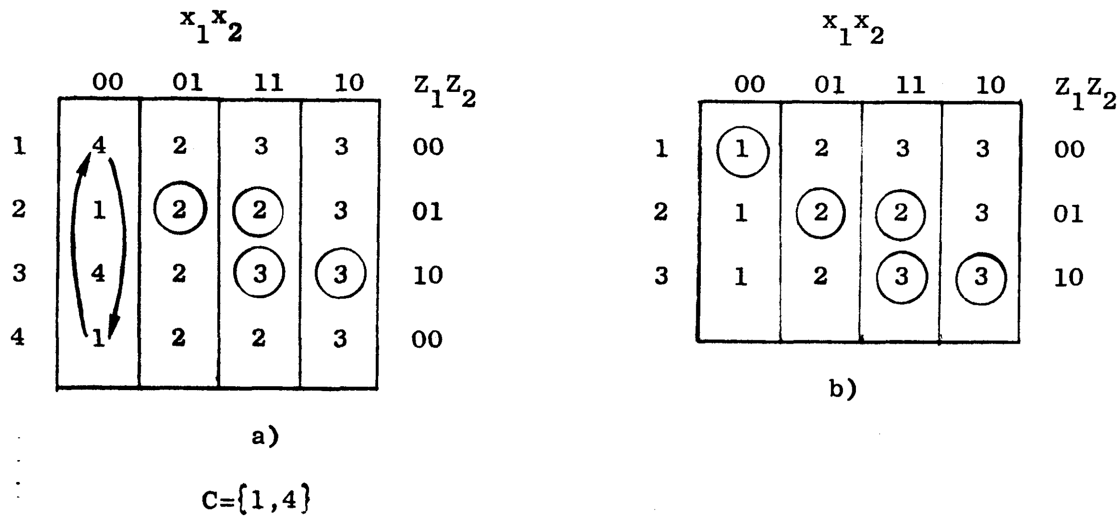


Figure 5. Another example of a state table with a cycle.

once during each unstable internal state transition, the state table can always be transformed into a flow table. The procedure is the following. For each row with a stable entry, replace all unstable entries with the number of the stable state entered at the end of the internal-state transition. All rows which have only unstable entries can then be eliminated. The initial internal state of the flow table is the first row with a stable entry which is entered as a successor of the initial internal state in the state table.

The state table in Table 5 is cycle-free and has at most single output-variable changes during unstable transitions. The flow table, with states renumbered is shown in Table 12. This flow table is identical to the flow table in Table 1 which was specified in the synthesis of program 1 for the mutual exclusion problem. Therefore, the program of Table 3, which was analyzed to produce the flow table in Table 12, is a suitable program for the mutual exclusion problem.

The flow table in Table 12 is completely specified. That is, each next-state entry and output state is specified. In general, all state tables and flow tables obtained for programs will be completely specified. This is a direct consequence of the definition of the

Table 12. Flow Table for State Table in Table 5

	z_1 = 0	z_1 = 1	X_1
1	2	(1)	0
2	(2)	1	1

procedure to obtain these tables. A result from switching theory is the following [13]. For any completely specified flow table it is always possible to obtain a unique* flow table with a minimum number of internal states which is indistinguishable from the original table.

Another example of the reduction of a state table to a flow table, the flow table for the state table of Table 8, is given in Table 13. The difficulty discussed earlier is detectable by examining internal state 4 when x_1x_2: 11 ➤ 10 ➤ 11. If the table entry in row 4, column 10 is changed from ④ to 3, the difficulties are avoided.

SYNTHESIS OF PROGRAMS

In this section, formal synthesis procedures are specified which make it possible to produce, from a flow table, a program of the class $\mathcal{P}$. As a result, it is possible to design control programs for problems such as the mutual exclusion problem.

The first step in the synthesis procedure is to obtain a flow table of the form shown in Fig. 2 with a minimum number of internal states. Techniques for obtaining such flow tables are well known [13]. The flow table obtained may or may not be completely specified. Let us consider the possibility that the flow table is not completely specified. If a next internal-state entry is unspecified in the table, it will be assumed that the designer intended that this entry never

---

* Except possibly for the numbering of the internal states.

Table 13. Flow Table for the State Table of Table 8 and the Program of Table 6

	x_1x_2 00	01	11	10	Z_1Z_2
1	(1)	2	3	3	00
2	1	(2)	4	3	01
3	1	2	(3)	(3)	10
4	1	2	(4)	(4)	11

be entered during the operation of the component. If the value of an output variable is unspecified for some internal state, it will be assumed that the designer does not care what the value of the output variable is when the component is in that internal state. Consider first, the implementation of the flow table as a sequential circuit. When the excitation functions for the circuit are determined, all unspecified entries in the table, both internal-state entries and output-state entries, become completely specified. Now consider the implementation of the flow table as a program. Since unspecified internal-state entries are never entered, they can be changed to stable entries without affecting the operation of the component. When this is done a flow table is obtained in which all next internal-state entries are completely specified and only the values of certain output variables may be unspecified. For a program, the present output state depends on the most recently executed control assignment statement for each output variable. Furthermore, once a value is assigned to an output variable in a program, the value of that variable will always be defined for as long as the program exists. Therefore, for a program, it is possible for the value of an output variable to be undefined only from the time the execution of the program starts until the first control assignment statement is executed for the variable (it is assumed that a value is assigned to each output variable at least once in every program). Rather than allow this interval when the value of an output variable is undefined, we require that the initial state of every program be completely specified. This

means that, if a flow table is to be implemented as a program, the output state associated with the initial internal state must be completely specified. In the remainder of this discussion, it will be assumed that all next internal-state entries are specified and that the initial output state is completely specified for any flow table which is to be implemented as a program.

Given any completely specified flow table, the following procedure constructs a program of the class $\mathcal{P}$.

Procedure to Obtain a Program of the Class $\mathcal{P}$ for Any Flow Table

1. Declare input and output variables; specify the initial output state.
2. Define a wait statement for each row in the flow table. For each unstable entry in a given row, define the pair (s_j, i_j) in the wait statement, where s_j is the input state for the unstable entry and i_j is the number of the next internal state. If all entries in a row are stable, replace the wait statement by a halt statement.

3. Before each statement from step 2, place control assignment statements to define the output state for the corresponding row in the flow table. Number the first of these statements with the number of the corresponding row in the flow table.

The program obtained for the control flow table in Table 1 is shown in Table 14. This procedure results in a program of the class $\mathcal{P}$ which does not contain any "go to" statements. It follows that given any flow table F, it is always possible to obtain a program of the class $\mathcal{P}$ which is indistinguishable from all other programs which are implementations of F and which does not contain any go to statements.

The programs produced by this procedure may be inefficient in the following two ways. First, they may contain unnecessary control assignment statements. For example, in the program of Table 14 which corresponds to the control flow table in Table 1, internal state 1 is always entered when Z_1 has the value 0; therefore, statement 1 can be removed and the second statement given the number 1. Similarly, internal state 4 is always entered with Z_2 equal to 0. Consequently the statement Z_2:=0 after statement number 4 is unnecessary. A second source of inefficient operation is the following. A control assignment statement may be executed when in fact the output variable already has the value specified by the assignment statement. In such cases, it may be possible to transfer to a different statement and avoid executing the assignment statement. There are many instances of

Table 14. Control Program for the Two-Program Mutual Exclusion Problem

```
        (initially Z1,Z2 are 0)
        INPUT X1,X2;
        OUTPUT Z1,Z2;
1:      Z1 := 0;
        Z2 := 0;
        WAIT (01,2),(11,3),(10,3);
2:      Z1 := 0;
        Z2 := 1;
        WAIT (00,1),(10,3);
3:      Z1 := 1;
        Z2 := 0;
        WAIT (00,4),(01,2);
4:      Z1 := 0;
        Z2 := 0;
        WAIT (01,2),(11,2),(10,3).
```

a) Program

		x_1x_2 00	01	11	10	Z_1Z_2
(2 last)	1	(1)	2	3	3	00
(2 gets)	2	1	(2)	(2)	3	01
(1 gets)	3	4	2	(3)	(3)	10
(1 last)	4	(4)	2	2	3	00

b) flow table

this inefficency in the program of Table 14. For example if we reorder the assignment statements for statement 3 and its immediate successor it is possible to transfer directly to the Z_1 : = 1 statement when entering internal state 3 from internal states 1 and 4. After taking advantage of these and similar economies, the program shown in Table 15 can be obtained. We claim that this program has the fewest number of statements of any program which implements the control flow table in Table 1. Furthermore, control assignment statements are executed only when the value of an output variable must be changed. We have not yet obtained a general algorithm to perform this simplification of programs. The interested reader may verify that the programs of Tables 14 and 15 are indistinguishable by constructing the flow tables for these programs using the analysis procedures from the proceeding section.

The programs produced by the synthesis procedure do not contain any dummy statements. The reason for this is that dummy statements represent computations which have no effect on the output variables and which are not affected by the values of the input variables. These statements may be inserted in any sequence of statements, with appropriate statement renumberings and possible introduction of go to statements, without affecting the external behavior of the component other than to introduce delays in value transitions for line variables. To illustrate the synthesis procedure when dummy statements are inserted, a program 1 for the mutual exclusion problem will be designed. The flow table is given in Table 1. The program obtained by applying the synthesis procedure is given in Table 16. The critical section is

Table 15. Another Control Program for the Two-Program Mutual Exclusion Problem

```
        (initially Z1,Z2 are 0)
        INPUT X1,X2;
        OUTPUT Z1,Z2;
  1:    Z2 := 0;
  2:    WAIT (01,4),(11,7),(10,7);
  3:    Z1 := 0;
  4:    Z2 := 1;
  5:    WAIT (00,1),(10,6);
  6:    Z2 := 0;
  7:    Z1 := 1;
  8:    WAIT (00,9),(01,3);
  9:    Z1 := 0;
 10:    WAIT (01,4),(11,4),(10,7).
```

Table 16. Program 1 Without Dummy Statements

```
        (initially X1 is 0)
        INPUT Z1;
        OUTPUT X1;
1:      X1 := 0;
        WAIT (0,2);
2:      X1 := 1;
        WAIT (1,1).
```

a) Program 1

z_1

	0	1	X_1
1	2	(1)	0
2	(2)	1	1

b) flow table

entered after the execution of the second wait statement and before execution of statement number 1. To represent the critical section a dummy statement can be inserted at the end of the program followed by a go to statement. After statement number 1, another dummy statement can be inserted to represent computation performed outside the critical section. The modified program is shown in Table 17. Except for the numbering of the statements, this program is identical to the program of Table 3 which was analyzed earlier to produce the same flow table used to start this synthesis example.

Table 17. Program 1 With Dummy Statements

```
        (initially X1 is 0)
        INPUT Z1;
        OUTPUT X1;
1:      X1 := 0;
        DUMMY;          (remainder of program 1)
        WAIT (0,2);
2:      X1 := 1;
        WAIT (1,3);
3:      DUMMY;          (critical section)
        GO TO 1.
```

CONCLUSIONS

The flow table model presented in [1] and the analysis and synthesis procedures presented in this paper have been valuable in the design of a solution or control algorithm for the mutual exclusion problem. An important feature of this model is the ability to consider both program and circuit implementations. By restricting control operations to be finite (which does not appear to be a restriction for the mutual exclusion problem) it is possible to find "optimal" implementations in the sense that the number of internal states required in a flow table is minimized. The procedures we have defined make it possible to determine if a program or circuit is correct in the sense that its operation is described by a given flow table. The determination of correct operation for an entire parallel system has not been considered in this paper but will be the subject of an additional paper in which we continue our investigation of flow table models for parallel computer systems.

ACKNOWLEDGEMENTS

The author would like to thank Professor Edward J. McCluskey for his many comments and criticisms. Thanks are also due to Harold S. Stone for proofreading a draft of this report.

REFERENCES

[1] Bredt, T.H. and McCluskey, E.J. A model for parallel computer systems. Technical Report No. 5 , SEL Digital Systems Laboratory, Stanford University, Stanford, California (Apr 1970).

[2] Dennis, J.B. and Van Horn, E.C. Programming semantics for multiprogrammed computations. Comm. ACM, 9 (March 1966), 143-155.

[3] Dijkstra, E. W. Solution of a problem in concurrent programming control. Comm. ACM, 8 (Sept 1965), 569.

[4] Dijkstra, E. W. The structure of the "THE" multiprogramming system. Comm. ACM, 11 (May 1968), 341-346.

[5] Dijkstra, E. W. Co-operating sequential processes. in Programming Languages, Genuys, F. (Ed.), Academic Press, New York (1968).

[6] Floyd, R.W. Assigning meanings to programs. Proc. of Symposium on Applied Mathematics, Vol. 19, American Mathematical Society (1967), 19-32.

[7] Knuth, D.E. Additional comments on a problem in concurrent programming control. Comm. ACM, 9 (May 1966), 321-322.

[8] Knuth, D. E. The Art of Computer Programming. Vol. 1, Addison-Wesley Publishing Co., Reading, Mass. (1968).

[9] Lampson, B. W. A scheduling philosophy for multiprocessing systems. Comm. ACM, 11 (May 1968), 347-360.

[10] Manna, Z. Termination of algorithms. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania (Apr 1968).

[11] Manna, Z. Properties of programs and the first-order predicate calculus. J. ACM (Apr. 1969).

[12] Manna, Z. The correctness of programs. J. of Computer and System Sciences, 3 (May 1969).

[13] McCluskey, E.J. Introduction to the Theory of Switching Circuits. McGraw-Hill Book Co., New York, N.Y. (1965).

[14] Saltzer, J.H. Traffic control in a multiplexed computer system. MAC-TR-30, Massachusetts Institute of Technology, Cambridge, Mass. (July 1966).