1972

A method for generating UTS assignments with an iterative state transition algorithm

Dattatraya Govind Raj-Karne

Follow this and additional works at: http://scholarsmine.mst.edu/doctoral_dissertations

Part of the Electrical and Computer Engineering Commons

Department: Electrical and Computer Engineering

Recommended Citation


This Dissertation - Open Access is brought to you for free and open access by the Student Research & Creative Works at Scholars' Mine. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of Scholars' Mine. For more information, please contact weaverjr@mst.edu.
A METHOD FOR GENERATING UTS ASSIGNMENTS WITH AN ITERATIVE STATE TRANSITION ALGORITHM

by

DATTATRAYA GOVIND RAJ-KARNE, 1937-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

1972

Advisor

Herbert R. Atcom

James H. Casey

Paul W. Stgeft

Jasim M. Taylor

T2783

84 pages

c.l

237263
ABSTRACT

There is a lack of systematic procedures that can be used to find uni-code totally sequential (UTS) assignments from a flow table description of an asynchronous sequential circuit. Presented here is an iterative internal state assignment method. This method consists of three algorithms. The first generates a minimum variable initial assignment from a flow table description. The second tests the validity of this assignment by constructing minimum length transition paths without crossover and the third augments this assignment by adding an internal state variable in the event that all transition paths cannot be constructed without crossover. The second and the third algorithms are used iteratively until a valid non-universal UTS assignment is produced.

The iterative state assignment method is systematic in all its phases. Every phase of the method includes more than one algorithm to perform the same function. The algorithm producing minimum length transition paths is very powerful in that it can also be used in conjunction with other state assignment methods producing either universal or non-universal UTS assignments.

After one obtains a valid UTS assignment an algorithm is provided to replace some or all of the totally sequential transitions with mixed mode transitions. This reduces the number of subtransitions in a given transition path and therefore speeds up the transition time considerably.
ACKNOWLEDGEMENT

The author wishes to express his appreciation and extend his sincere thanks to Dr. James H. Tracey for his guidance during this project. The author is indebted to him for the understanding and the personal interest shown by him during the entire period of the author's doctoral studies.

The author wishes to thank Dr. Javin M. Taylor for his quick and careful review of this dissertation.

The author also wishes to express his gratitude to his wife Anuradha and son Shrikant for their interest and constant encouragement rendered during the entire period of the author's graduate study.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. ISAM: AN ITERATIVE STATE ASSIGNMENT METHOD</td>
<td>10</td>
</tr>
<tr>
<td>A. ALIAS: An Algorithm for Initial Assignment</td>
<td>14</td>
</tr>
<tr>
<td>1. PRIME</td>
<td>14</td>
</tr>
<tr>
<td>2. DIAGRAM</td>
<td>18</td>
</tr>
<tr>
<td>a. PART</td>
<td>20</td>
</tr>
<tr>
<td>b. JOIN</td>
<td>24</td>
</tr>
<tr>
<td>B. TRAPAGAL: A Transition Path Generation Algorithm</td>
<td>33</td>
</tr>
<tr>
<td>1. COUNT</td>
<td>35</td>
</tr>
<tr>
<td>2. COUNT-DEMAND</td>
<td>39</td>
</tr>
<tr>
<td>3. PATH</td>
<td>43</td>
</tr>
<tr>
<td>C. AAA: An Algorithm to Augment an Assignment</td>
<td>51</td>
</tr>
<tr>
<td>1. STRUCTUR</td>
<td>51</td>
</tr>
<tr>
<td>2. GAIN</td>
<td>54</td>
</tr>
<tr>
<td>3. MIXMÖD</td>
<td>56</td>
</tr>
<tr>
<td>D. ALSPT: An Algorithm to Speed up Transitions</td>
<td>61</td>
</tr>
<tr>
<td>III. SUMMARY AND DISCUSSION</td>
<td>70</td>
</tr>
</tbody>
</table>
Table of Contents continued

<table>
<thead>
<tr>
<th>IV. RELATED AREAS OF FUTURE WORK</th>
<th>74</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Routing and Transition Path Generation</td>
<td>74</td>
</tr>
<tr>
<td>B. Unified State Assignment Techniques</td>
<td>75</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>76</td>
</tr>
<tr>
<td>VITA</td>
<td>78</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

Sequential switching circuits denote a class of devices whose outputs depend not only on the present inputs but also on previous inputs. These circuits are further classified as being synchronous or asynchronous. In synchronous circuits, clock pulses synchronize the operation of the circuit while in asynchronous circuits, it is usually assumed that no such clock is available. A desirable feature of asynchronous design is that the resulting circuit does not have to wait for the arrival of clock pulses before effecting a transition. However, the absence of clock pulses introduces the problem of insuring that the circuit functions according to specifications independent of variations in transmission delays of signals.

The operation of an asynchronous sequential circuit can be described by means of a flow table. As shown in Figure 1, it is a two-dimensional array consisting of next-state entries, with its columns representing the input states and its rows representing the internal states of the circuit. The flow table usually shows the output states, too, but since this paper is concerned only with the internal operation of a sequential circuit, the output states are not shown in the flow table.

The row in which the circuit is currently operating is often referred to as the present internal state or just the present state. For example, if the present state of the circuit described by Figure 1 is "a" and then
Input States

<table>
<thead>
<tr>
<th>I_1</th>
<th>I_2</th>
<th>I_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>e</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>e</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

Figure 1. Flow Table for an Asynchronous Sequential Circuit.

The combination of input state and present internal state is called the total circuit state. In some flow tables, particular total circuit states are never entered and the corresponding next-state entries are unspecified. The unspecified next-state entry is called a "don't care" state. Flow tables with "don't care" states are called incompletely specified flow tables. Since a "don't care" state is never entered in the synthesis of the actual circuit, it is permissible to assign any value to such a state to simplify the final design. The material presented in this paper applies to completely specified and incompletely specified flow tables.

Definition: An asynchronous sequential circuit is said to be operating in fundamental mode if the inputs are never changed unless the
circuit is in a stable state.

Definition: A transition from an unstable state to a stable state is called a **direct transition** if all internal state variables that are to undergo a change of state are simultaneously excited.

One of the basic steps in the synthesis procedure of designing an asynchronous sequential circuit is obtaining an internal state assignment. The internal state assignment consists basically of encoding each of the internal states of a sequential circuit with a binary n-tuple or set of n-tuples. The n-tuples are encoded by n internal state variables, \( y_1, y_2, \ldots, y_n \). With n internal state variables at most \( 2^n \) internal states can be encoded.

In an asynchronous circuit, the internal state assignment must be made so that each internal transition always leads to a definite and appropriate stable state independent of the relative speeds of the circuit elements.

Definition: A **race** exists in an asynchronous sequential circuit whenever a transition between a pair of states requires simultaneous change of two or more internal state variables. If the result of a race leads to false operation of the circuit, it is designated as a **critical race**; otherwise, it is a **non-critical race**.

Every internal state assignment must permit circuit operation free of critical races. One basic approach for doing this is to allow no races at all, thereby eliminating critical races. The second approach is to obtain an assignment that permits races, where all races are non-critical.
Based on these two approaches, two main types of internal state assignment techniques have evolved.

In an assignment for an asynchronous sequential circuit where each unstable state leads directly to a stable state, all internal state variables that are to change state during a transition are excited simultaneously at the beginning of the transition. Such assignments are called single transition time (STT) assignments [1]. Further, if only a single coding is associated with each internal state, it is called a uni-code single transition time (USTT) assignment [11]. A uni-code totally sequential (UTS) assignment also assigns a unique binary code to each internal state but all transitions between an unstable state and a stable state are accomplished through the change of a single internal state variable at a time. It is clear that races may exist in USTT assignments, but not in UTS assignments.

Single transition time assignment techniques have received considerable attention from researchers over the last decade [1,2,3,4,5]. As a result, there are well-known established methods for generating the USTT assignments and the corresponding next-state equations. Totally sequential assignment techniques, on the other hand, have received considerably less attention. The main contributions in this area are due to Hazeltine [6], Maki [7,8], and Saucier [9].

Definition: A transition path for a flow table with a UTS assignment is an ordered set of adjacent internal states traversed in going from an
unstable state $S_a$ to a stable state $S_b$. The transition path includes $S_a$ and $S_b$.

Definition: The distance $d$ between two internal states $S_a$ and $S_b$ is the number of bit positions in which the binary code of $S_a$ differs from the binary code of $S_b$.

Definition: Let the distance between two internal states be $d$. If $d$ state variables are excited, each one only once, in effecting a transition between the states, then the transition is called a minimum length (ML) transition and the transition path is called a minimum length transition path.

Definition: A crossover is present when transition paths for any two transitions $S_a \rightarrow S_b$ and $S_c \rightarrow S_d$, such that $S_b \neq S_d$, within the same flow table column have at least one internal state common.

Definition: A $k$-set of a flow table column consists of $k-1$ unstable states leading to the same stable state together with that stable state.

Definition: A UTS assignment is a valid UTS assignment if and only if it is possible to construct all transition paths such that the transition paths among the states of one $k$-set do not crossover transition paths among the states of any other $k$-set within the same flow table column.

Definition: A state assignment is said to be universal if its validity depends only on the number of flow table rows; otherwise, it is non-universal.

Definition: Let $s_o$ be the minimum number of internal state variables to uniquely code an $r$-row flow table. An $n$ variable assignment for this
flow table is called a near-minimal assignment if \( s_o \leq n \leq s_o + \left\lfloor \frac{s_o}{2} \right\rfloor \), where \( \lfloor x \rfloor \) indicates the smallest integer \( \leq x \).

Hazeltine's [6] method consists of attempting to construct transition paths between all stable states and their corresponding unstable states on per column basis. There is a trial and error associated with obtaining the assignment and the transition paths. This method generates a non-universal internal state assignment and transition paths are non-minimum length.

Maki [7] not only developed a method to generate universal assignments but also suggested a new improved bound on the number of internal state variables for such assignments. Even though Maki did not provide a proof in support of this bound no counterexample has yet been found. For a particular class of flow tables, Maki has also developed an algorithm to generate transition paths for all transitions, on per column basis.

Maki's [8] method of generating a non-universal internal state assignment consists of an iterative procedure of adding an internal state variable to an initial minimum variable assignment based on characteristics of a given flow table, until either all transitions could be realized without a conflict or an upper bound on the number of internal state variables is reached. The method does not include any systematic way of generating either an initial assignment or transition paths.

Saucier [9] uses a graph-theoretic approach to generate a non-universal UTS assignment. The assignment technique is based on
attempting to construct transition paths without a crossover. Successful construction of all transition paths without crossover guarantees validity of an internal state assignment generated by this technique. In this respect, Saucier's and Hazeltine's approaches of producing an internal state assignment are similar. However, the former uses minimum length transition paths, while the latter uses non-minimum length transition paths.

A UTS assignment in general requires fewer internal state variables than a USTT assignment. Though universal UTS assignments are available with minimal effort, these usually require more internal state variables than non-universal UTS assignments. Under a particular cost speed trade-off criterion, a minimal or near-minimal non-universal UTS assignment may prove to be the most efficient internal state assignment.

In USTT assignments, all the transitions between an unstable state and a stable state are direct. Therefore, there is a unique transition path associated with all of the transitions. However, in both the universal and non-universal UTS assignments, a problem always remains: given an assignment, how does one proceed through the construction of transition paths? Even when an assignment is known to be satisfactory, it is not always a straightforward procedure to construct transition paths free of crossovers.

A number of methods for producing UTS assignments have been mentioned above. Some of these have a capability of generating transition paths. However, such capability is considerably limited in that
it can only be used for UTS assignments generated by respective methods.

The preceding discussion points to the following limitations of existing methods:

1) non-availability of a generalized transition path generation algorithm that can be used with any UTS assignment technique;

2) lack of systematic generation of initial assignment;

3) after a valid UTS assignment has been obtained, no means of speeding up some or all of the totally sequential transitions by introducing races.

This paper presents for the first time an iterative state assignment method that is systematic and algorithmic in all its phases. These different phases eliminate completely the limitations of the present methods. The state assignment method produces a near-minimal non-universal UTS assignment for completely or incompletely specified flow tables.

The method consists of a collection of algorithms. At the heart of the method is a transition path generation algorithm, which is unique in that it can also be used in conjunction with any other UTS assignment techniques, either universal or non-universal, to produce transition paths. An algorithm to systematically produce a minimum variable initial assignment is also included.
Lastly, a procedure to speed up totally sequential transitions, by making use of non-critical races, is presented. This procedure is simple, but does not guarantee to speed up all transitions even when it may be possible to do so.
II. ISAM: AN ITERATIVE STATE ASSIGNMENT METHOD

One step in the synthesis procedure for realizing an asynchronous sequential circuit is the generation of next-state and output state equations from a flow table description of the circuit. The normal approach in determining these equations is to first choose an internal state assignment based on the structure of flow table and second to construct a state table based on this assignment.

A state table [10] differs from the flow table in that a state table shows all of the internal states that a sequential circuit can assume along with the next-state entries, whereas a flow table indicates only the unstable and stable states. For example, a flow table with seven internal states and coded with a four variable internal state assignment may have a corresponding state table with sixteen internal states. Of these sixteen states those that are not involved in any transitions in the state table are referred to as unspecified states.

Maki [8] has presented an iterative approach of producing a non-universal UTS assignment. The main limitations of this method are non-availability of a systematic approach to produce an initial assignment and lack of an algorithm to generate transition paths for a flow table. The method described here completely eliminates these limitations and therefore for the first time provides a user with a totally systematic method for generating a non-universal UTS assignment for any flow table.

The basis of this assignment procedure is to find a state assignment where all of the transitions can be accomplished such that the resulting
transition paths are minimum length. ML transitions are desirable in that the state variables which must change state are the only ones excited in effecting a transition. Also the number of states in an ML transition path is less than the number of states in a non-ML transition path for the same transition. Since a satisfactory set of transition paths is more likely when fewer states are required in the corresponding transition paths, ML transitions are more desirable.

The method presented here is called ISAM, an iterative state assignment method, and consists of four algorithms. It accepts a flow table description of an asynchronous sequential circuit as an input and produces a near-minimal valid UTS assignment along with a state table as output. The output of ISAM may then be used for generating next-state and output state equations.

ISAM consists of the following four algorithms:

A. **ALIAS**: **AL**gorithm for **I**nitial **AS**ignment

B. **TRAPAGAL**: **TR**ansition **PA**th **G**eneration **AL**gorithm

C. **AAA**: Algorithm to **A**ugment an **A**ssignment

D. **ALSPT**: **AL**gorithm to **SP**eed up **T**ransitions

ALIAS is the first step in the assignment procedure and encodes the states of a given flow table with minimum number of variables so that a maximum number of transitions are satisfied. The binary code thus obtained is called an initial assignment. The next step, TRAPAGAL, of the procedure involves construction of transition paths for this initial assignment. If all transition paths so constructed are valid, one has
generated a valid UTS assignment; otherwise, the initial assignment is augmented by adding an internal state variable. This step is performed by AAA. TRAPAGAL and AAA are now iteratively used until a valid UTS assignment is produced. At this point one has an option of either accepting the valid assignment and associated totally sequential transition paths or using ALSPT to speed up these transitions by introducing non-critical races.

All individual algorithms, except ALSPT, have the capability of using more than one technique to perform the same function. For example, ALIAS uses two different techniques to produce an initial assignment. The user has an option of choosing any of these techniques. Figure 2 shows a flowchart for ISAM showing relationship between its constituent algorithms. Different techniques for each algorithm are enumerated on the flowchart.

An operations manual for ISAM [15] is available. This includes a detailed source listing and flowcharts for the various algorithms. It also explains multiple options available to the user and provides instructions on how to use them.

A detailed description of each algorithm follows. An illustrative example is included at the end of each algorithm to bring out its salient points.
Call AAA to augment generated assignment by using any of the following: 1. STRUCTUR 2. GAIN 3. MIXMOD

List unsuccessful transitions and columns in which they appeared

Call TRAPAGAL Construct transition paths using any of the following: 1. COUNT 2. COUNT-DEMAND 3. PATH

Can all transitions be accomplished without crossover?

YES

Is search for reduction in transition time needed?

YES

Use ALSPT to speed up transitions.

NO

Read Flow Table & Choice of Algorithms

Call ALIAS Generate initial assignment using either PRIME or DIAGRAM

PRINT Flow Table, State Table and Transition Paths

END

Figure 2. Flowchart of ISAM: An Iterative State Assignment Method.
A. ALIAS: An Algorithm for Initial Assignment

Consider an \( r \)-row flow table. \( S_0 = \lceil \log_2 r \rceil \), where \( \lceil x \rceil \) indicates the smallest integer \( \geq x \), internal state variables are needed to uniquely code each row of the flow table. The basis of the initial assignment procedure is to code the internal states with a single but unique binary code with minimum number of variables. The states having transitions between them are given codes close together while codes for those states having no transitions between them are given codes that are maximum distance apart. Depending on the structure of the flow table it is in general difficult to satisfy all of these conflicting requirements. At times it is also hard to determine the amount of effort that should go into finding a good initial assignment. A good initial assignment is one which permits the completion of a near maximum number of transitions within a given flow table. Two initial assignment methods are presented.

1. PRIME

This is an initial assignment technique based on USTT assignments. As mentioned earlier UTS assignments generally require less internal state variables than USTT assignments. This suggests that by choosing a proper subset of the internal state variables of a valid USTT assignment one can obtain a valid UTS assignment. Since USTT assignment techniques are readily available, such an approach seems quite promising. Tracey [1] has developed algorithmic techniques of producing
USTT assignments. In one of these techniques all dichotomies under each input column are listed and a Boolean matrix is formed. The algorithm then determines an intersection or partition that will cover a maximum or near maximum number of rows of this Boolean matrix. Then covered rows are discarded and the same process is repeated on the subset of the original matrix. This process is terminated when all rows of the matrix are covered. The covering corresponds to a valid USTT assignment. It is clear from the basic strategy of this technique that the first internal state variable covers a larger number of dichotomies than the second internal state variable and so on. This leads to the conjecture that an initial assignment arrived at by this technique would be a good initial assignment.

Tracey's [1] matrix reduction algorithm has been programmed and the test problems run indicated the following limitations of this algorithm:

(a) $S_0$ internal state variables do not always guarantee a distinct code for each row of a flow table. Under this condition one has to use an initial assignment with $S_0 + 1$ internal state variables.

(b) For larger flow tables, the size of Boolean matrix increases considerably. Increased size of the Boolean matrix along with the technique for finding a cover results in a long running computer program.
Example #1: Refer to the flow table of Figure 3.

![Flow Table for Example #1](image)

Step 1: List all dichotomies under each input column.

<table>
<thead>
<tr>
<th>Input States</th>
<th>( I_1 )</th>
<th>( I_2 )</th>
<th>( I_3 )</th>
<th>( I_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-25</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13-46</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>25-46</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14-26</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14-35</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>26-35</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16-32</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16-45</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32-45</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>51-24</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>51-63</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>24-63</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 2: Form a Boolean matrix.

Step 3: Since \( S_0 = 3 \), use Tracey's Boolean matrix reduction algorithm [1] three times. The first iteration yields a partition.
Delete the rows of the Boolean matrix that are covered by this partition. The reduced Boolean matrix is given below.

\[
\begin{array}{cccccc}
13-25 & 0 & 1 & 0 & 1 \\
25-46 & 0 & 1 & 0 & 1 \\
35-14 & 1 & 0 & 1 & 0 \\
14-26 & 0 & 1 & 0 & 1 \\
54-32 & 1 & 1 & 0 & 0 \\
32-61 & 1 & 0 & 0 & 1 \\
54-61 & 1 & 0 & 0 & 1 \\
42-63 & 0 & 1 & 0 & 1 \\
51-63 & 0 & 1 & 0 & 1 \\
\end{array}
\]

The second iteration on the reduced Boolean matrix yields a partition \([(145), (236)]\). Delete rows of the Boolean matrix covered by this partition and obtain the reduced Boolean matrix shown below.

\[
\begin{array}{cccccc}
25-13 & 1 & 0 & 1 & 0 \\
25-64 & 0 & 1 & 0 & 1 \\
35-14 & 1 & 0 & 1 & 0 \\
32-61 & 1 & 0 & 0 & 1 \\
54-61 & 1 & 0 & 0 & 1 \\
42-63 & 0 & 1 & 0 & 1 \\
\end{array}
\]

The third iteration yields a partition \([(146), (235)]\) generating a three variable initial assignment given below.

\[
\begin{align*}
y_1 & : [(135), (246)] \\
y_2 & : [(145), (236)] \\
y_3 & : [(146), (235)]
\end{align*}
\]
When this initial assignment is next tested for validity by constructing all transition paths, it is seen that all transition paths can be constructed without crossover. The initial assignment generated above is therefore a valid UTS assignment. Since after the third iteration of the Boolean matrix reduction algorithm the reduced Boolean matrix is not empty the initial assignment is not a valid USTT assignment.

2. DIAGRAM

An $S_0$ variable initial assignment for an $r$-row flow table consists of $S_0$ distinct two block partitions on the state set of this flow table. PRIME used a classical approach of using an existing state assignment technique to generate $S_0$ distinct partitions. Since any classical approach is based on satisfying a particular class of constraints, one in general cannot expect such a technique to always produce the best result for a different class of constraints. Tracey's fundamental theorem [1] states that the constraint list for obtaining a USTT assignment consists of all the dichotomies as well as the pairs of states that do not appear in either the left or the right side of any of the dichotomies. The Boolean matrix reduction algorithm operating on such a set of constraints was not intended to insure a unique code for each internal state until the complete Boolean matrix was covered. This explains the reason for PRIME not being always able to produce a distinct code for each internal state with $S_0$ internal state variables. The method described here eliminates this limitation by using successive partitioning of the total state set.
DIAGRAM consists of a two step iterative procedure. The first step involves successive partitioning of the total state set, of a given flow table, into disjoint blocks using a partitioning algorithm called PART. This process is continued until each block contains a single element and therefore ensures a distinct code for each internal state. After the second and subsequent iterations of PART, the resulting multiple blocks are merged to form a two block partition. This is accomplished by using JOIN. Each time JOIN is used a new internal state variable is generated. The iterative procedure of using PART and JOIN terminates when $S_0$ internal state variables are generated. Use of the partitioning approach ensures that with generation of $S_0$ internal state variables each row of the flow table has a distinct code.

The effectiveness of the initial assignment in satisfying transitions within a flow table clearly depends upon the two constituent algorithms PART and JOIN. It is apparent, therefore, that making these two algorithms complex and exhaustive would result in an increase in the effectiveness of the initial assignment at the expense of a long running computer program. Since the transition path generation algorithm is used to determine validity of the initial assignment (see Figure 2), a long running initial assignment algorithm (in relation to the running time of TRAPAGAL) adding only a marginal effectiveness to the initial assignment, is undesirable. This led to the incorporation of quick and simple techniques for both PART and JOIN.
a. PART

An inspection of valid internal state assignments indicates that irrespective of the methods of obtaining such assignments the codes for the states having transitions between them are closer to each other while codes for those states having no transitions between them are a large distance apart. This important inherent property shared by all state assignments is the basis for PART. The earliest methods of obtaining the secondary assignments used transition diagrams [12], which enabled one to visualize this important property. However with an increase in the size of flow tables, the transition diagrams become more complex. This results in considerably diminishing the effectiveness of transition diagrams for obtaining secondary assignments through manual search technique. However when manual search is replaced by machine search, even large and complex transition diagrams can be systematically used for obtaining secondary assignments. PART uses transition diagrams in conjunction with a computer search technique to produce an initial assignment. The first step of the algorithm consists of incorporating a weighting scheme. One could use either a weighting scheme on the states of a flow table or on the transitions within a flow table. Since the validity of any internal state assignment depends on successfully completing all transitions, assigning weight to each transition rather than to the individual elements of a k-set seems appropriate. This was also confirmed by comparing different weighting schemes on individual states and transitions.
The weighting scheme used here is based on the number of elements in a k-set. The transition between elements of a k-set with only two elements is given the highest weight since in general such a transition has to satisfy the most stringent requirements for successful completion. As the number of elements in a k-set increase, these requirements become more and more relaxed as now there is more than one way of completing a transition between k-set elements. Hence transitions between states of a k-set with multiple elements are assigned weights inversely proportional to the number of k-set elements. For example a transition between states belonging to a 3-set is assigned a greater weight than a transition between states belonging to a 4-set and so on. The weighting scheme also takes care of "don't care" states in a flow table. The details of the weighting scheme are given later. The following definitions will be useful in the understanding of the algorithm.

**Definition:** A **connectivity matrix** for an r-row flow table is an r x r binary matrix. Element \((i,j) = element (j,i) = 1\) if there is a transition between rows \(i\) and \(j\) of the flow table; otherwise, element \((i,j) = element (j,i) = 0\).

**Definition:** A **transition weight matrix** for an r-row flow table is an r x r matrix. Element \((i,j) = element (j,i) = \text{weight of a transition between rows } i \text{ and } j\). For all rows \(m\) and \(n\) between which there is no transition element \((m,n) = element (n,m) = 0\). From this definition it is clear that the transition weight matrix is symmetric.
Let $S_p = \{\text{Input state set to PART}\}$

$S_{p1} = \{\text{Left block of two block partition produced by PART on } S_p\}$

$S_{pr} = \{\text{Right block of two block partition produced by PART from } S_p\}$

It is clear that before one uses PART the sets $S_{p1}$ and $S_{pr}$ are null while after using PART the sets $S_{p1}$ and $S_{pr}$ are mutually disjoint and their set union is an improper subset of $S_p$ [17].

The first iteration of PART partitions the total state set of the flow table into a two block partition while the second and all the subsequent iterations successively partition each block produced in the previous iteration into two blocks. Thus the second iteration uses PART twice to produce a four block partition of the total state set of the given flow table and in general the $m$th iteration of PART uses PART $2^{m-1}$ times and produces a $2^m$ block partition of the total state set.

(i) The first iteration of PART

During the first iteration of PART the input state set $S_p$ is an improper subset of the total state set of the given flow table. Therefore at the beginning of the first iteration one has the following:

$S_p = \{1, 2, 3, \ldots, r\}$

$S_{p1} = \{\emptyset\}$

$S_{pr} = \{\emptyset\}$, where $\emptyset$ indicates a null set.

The algorithm begins by selecting an element $s_{p1}$ of $S_p$, assigns this element to $S_{p1}$ and deletes it from the input set $S_p$. The connectivity matrix is used to determine a subset $S_r$ of the input set $S_p$ such
that there is no transition between $s_{pl}$ and any element of the subset $S_r$. Next an element $s_{pr}$ of the set $S_r$ is chosen such that the graph-theoretic distance [13] between $s_{pr}$ and $s_{pl}$ is the largest. If there is more than one such element the choice is arbitrarily made. This element is assigned to the set $S_{pr}$ and is simultaneously deleted from the input set $S_p$.

(ii) Enlarging of $S_{pl}$ and $S_{pr}$

Both $S_{pl}$ and $S_{pr}$ are enlarged alternately by adding one element to each set at a time. The same procedure is used to enlarge both these sets and therefore the procedure to enlarge $S_{pl}$ is given below.

One selects an element $s_p$ contained in the set $S_p$ such that the following two conditions are concurrently satisfied. (a) There is a transition between $s_p$ and the largest number of elements of $S_{pl}$ and (b) there is a transition between $s_p$ and the least number of elements of $S_{pr}$. If more than one such $s_p$ exists then the transition weight matrix is used to make a proper choice. Now $s_p$ is deleted from $S_p$ and is assigned to the set $S_{pl}$. The set $S_{pr}$ is enlarged using the same procedure except that $S_{pr}$ is substituted for $S_{pl}$ in the above procedure.

The enlarging of both sets terminates when the input state set $S_p$ to PART is empty. At this point the first iteration of PART is complete and the first internal state variable $y_1$ is given by:

$$y_1 : [(S_{pl}), (S_{pr})]$$

The second and all the subsequent iterations of PART use the procedure outlined above sequentially. For example during the second
iteration PART is used on $S_{pl}$ producing another two block partition $[(S_{pl1}), (S_{pl2})]$. Next it is used on $S_{pr}$ generating yet another two block partition $[(S_{pr1}), (S_{pr2})]$. Hence at the end of the second iteration of PART one has in effect four blocks. In order to generate next internal state variable one must find a way of recombining these multiple blocks into a distinct two block partition. This is achieved by using an algorithm called JOIN. For third and successive iterations of PART, each of the multiple blocks produced at the end of its previous iteration are used as input to PART. For example, for the third iteration of PART, each of $S_{pl1}, S_{pl2}, S_{pr1}$ and $S_{pr2}$ will be used as input to PART. The iterative use of PART and JOIN is continued until each of the multiple blocks generated after using PART contains only a single element.

b. JOIN

After each iteration of PART the number of blocks nearly doubles. For example after $m$th iteration of PART number of blocks may be between $2^{m-1}$ and $2^m$. JOIN operates on these multiple blocks and combines relevant blocks in all possible ways to produce a series of two block partitions. For each of these partitions it computes a figure of merit, and the partition with the least figure of merit is selected for generating a new internal state variable.
Assume that after \((m-1)\)th iteration of PART the following \(2^{m-1}\) block partition of the total state set \(S\) is produced.

\[
S = [A_1, A_2, A_3, \ldots, A_{2^{m-1}}]
\]

Now during the \(m\)th iteration of PART each \(A_i\) of \((m-1)\)th iteration is further partitioned into two blocks \(A_{i1}\) and \(A_{i2}\) generating a \(2^m\) block partition of the total state set \(S\). It is clear that one should not attempt to combine \(A_{i1}\) and \(A_{i2}\) while forming a two block partition to generate a new internal state variable. The recombination procedure uses a \(2^{m-1}\) bit binary counter and lets this counter count from \(000\ldots0\) to \(011\ldots1\). At each count, depending on the binary value of each bit, a group of blocks is merged to form the left block of a two block partition. Therefore each binary count of this counter represents a two block partition. Let the count be \(b_1, b_2, \ldots, b_{2^m}\) then for any \(b_i, b_i = 0\) points to block \(A_{i,1}\) while \(b_i = 1\) points to \(A_{i,2}\), hence a partition indicated by this count is given by

\[
[A_1, 1, 1, 1, 1, 1, \ldots, 1, \bar{b}_1 + A_{1,2} \cdot b_1, \text{(remaining states of flow table)}]
\]

Let this be denoted by \([(B_i/i=1,u), (C_j/j=1, v)]\) where \(u + v = r\), the number of flow table rows. \(A_{1,1}\) has been included in the left block of the partition to ensure that when all state variables are generated and the left blocks are coded with a 0, flow table state 1 will always have a code \(000\ldots0\).

By using the procedure outlined above, all of the \(2^{2^{m-1}-1}\) partitions are generated during the \(m\)th step. In order to select a proper
partition out of these, a figure of merit based on transition weights is computed for all these partitions using the following formula:

\[
\text{Figure of Merit} = \sum_{i=1}^{u} \sum_{j=1}^{v} \text{Transition Weight of (} B_i \rightarrow C_j \text{)} ,
\]

where \( B_i \rightarrow C_j \) indicates a transition between any elements of sets \( B_i \) and \( C_j \). The partition with the least figure of merit is used to generate a new internal state variable.

A step by step description of producing an initial assignment for an \( r \)-row flow table is given below.

Step 1: All \( k \)-sets are listed under each input column.

Step 2: Transition weights are assigned, on per column basis, using the rules given below.

Let \( d \) be the number of states with "don't care" next-state entries in a particular column.

(i) each transition between states of a \( k \)-set with two elements is assigned a weight of \( (r - d) \).

(ii) each transition between an unstable and stable state of a \( k \)-set with more than two elements is assigned a weight of \( (r - d - k) \).

(iii) if any transition appears in more than one input column, the net weight for this transition is the sum of the weights in each column.
Using rules (i), (ii) and (iii) weights for all transitions are calculated and a transition weight matrix is obtained. It can be seen that the weighting scheme takes into account specified and unspecified flow tables and flexibility offered in completing transitions between states of a multiple element k-set.

Step 3: PART is used on the total state set of given flow table to generate first internal state variable.

Step 4: Each block obtained in the previous iteration of PART is further partitioned into two blocks by using PART.

Step 5: JOIN operates on the multiple blocks produced by PART and by computing a figure of merit based on transition weights, produces a new internal state variable.

An iterative use of steps 4 and 5 is terminated when \( S_0 \) internal state variable initial assignment with distinct code for each internal state is produced.
Example #2: Refer to the flow table of Figure 4.

Input States

<table>
<thead>
<tr>
<th>Internal States</th>
<th>I₁</th>
<th>I₂</th>
<th>I₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4. Flow Table for Example #2.

Step 1: List all k-sets and number of "don't care" states under each input column.

I₁: 125, 36, d₁ (1)
I₂: 14, 23, 56, d₂ (0)
I₃: 123, 456, d₃ (0)

where d₁ (n) indicates n "don't care" states under input column I.

Step 2: Using the rules outlined for assigning transition weights, complete the transition weight matrix given below.

<table>
<thead>
<tr>
<th>Internal States</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 3: Use PART to generate the first two block partition from the total state set $S$. As explained in the description, initially

$S_p = \{1, 2, 3, 4, 5, 6\}$

$S_{pl} = S_{pr} = \emptyset$

(a) Delete state 1 from $S_p$ and assign it to the set $S_{pl}$.

Now use the connectivity matrix and determine a subset $A$ of $S_p$ such that there is no transition between state 1 and the elements of subset $A$. It can be seen that $A = \{5, 6\}$ satisfies this condition. Next compare the graph-theoretic distance between states 1, 5, and 6 and assign the element of $A$ with the largest distance to the set $S_{pr}$. In this example there is a tie in the graph-theoretic distances hence state 5 is arbitrarily assigned to the set $S_{pr}$. Delete this element from the input set $S_p$. At this point the first elements of both the sets $S_{pl}$ and $S_{pr}$ are known and are given below:

$S_p = \{2, 3, 4, 6\}$

$S_{pl} = \{1\}$ and

$S_{pr} = \{5\}$

(b)Enlarge $S_{pl}$ and $S_{pr}$ until every state of the flow table is an element of either $S_{pl}$ or $S_{pr}$. To enlarge $S_{pl}$ choose an element $s_p$ from the set $S_p$ so that there is a transition between $s_p$ and state 1 and there is no transition between $s_p$ and state 5. The only element that satisfies these conditions concurrently is state 3. Therefore $S_{pl}$ is enlarged by adding state
3 and simultaneously state 3 is deleted from $S_p$. At this point one has

$$S_p = \{2, 4, 6\}$$

$$S_{pl} = \{1, 3\} \text{ and } S_{pr} = \{5\}$$

Now in order to enlarge $S_{pr}$ select an element $s_j$ from the input set $S_p$ such that there is a transition between $s_j$ and state 5 and either there is no transition between $s_j$ and all the elements of $S_{pl}$ or there is a transition between $s_j$ and the least number of elements of $S_{pl}$. Only states 4 and 6 satisfy these conditions however, state 6 is chosen to enlarge $S_{pr}$ since the transition weight of transition 56 is greater than that of transition 54.

State 6 is deleted from the input set $S_p$ giving

$$S_p = \{3, 4\}$$

$$S_{pl} = \{1, 3\} \text{ and } S_{pr} = \{5, 6\}$$

(c) Continuing the procedure outlined above, the following two block partition is obtained generating the first internal state variable.

$$\Lambda_1 \quad A_2$$

$$y_1 : [(1 \ 3 \ 2), (5 \ 6 \ 4)] \quad (1)$$

Step 4. Use PART on both blocks $A_1$ and $A_2$ of (1) above to produce a four block partition.

$$[(A_{11}) (A_{12}) (A_{21}) (A_{22})] \quad (2)$$
where

\[ A_{11} = (12), A_{12} = (3) \]
\[ A_{21} = (56), A_{22} = (4) \]

Step 5. Use JOIN on this four block partition to obtain a second two block partition and the second internal state variable. Since this is the second iteration, form a \(2^{(2-1)} = 2\) bit counter. As explained in the description of JOIN, each count of this counter represents a two block partition. The two block partition with the least figure of merit is chosen to generate a new internal state variable. The two bit counter counts from 00 to 01. Count 00 implies a partition \([ (1256), (34)] = P_{21}\) and count 01 implies a partition \([ (124), (356)] = P_{22}\), where \(P_{ij}\) indicates a jth two block partition in the ith iteration.

Let \(F_{ij}\) indicate the figure of merit for the jth partition in the ith iteration then \(F_{21} = 26\) and \(F_{22} = 17\). Since \(F_{22}\) is the least figure of merit, \(P_{22}\) is used to generate the second internal state variable.

\[ y_2: [ (124), (356)] \quad (3) \]

Step 6. Use PART on each block of (2) to produce an eight block partition. If any blocks of (2) contain only one element a dummy null block is associated with it.

\[ [ (A_{111}) (A_{112}) (A_{121}) (A_{122}) (A_{211}) (A_{212}) (A_{221}) (A_{222}) ] \quad (4) \]
where \( A_{111} = (1), A_{112} = (2), A_{121} = (3), A_{122} = (\emptyset) \)
\( A_{211} = (5), A_{212} = (6), A_{221} = (4), A_{222} = (\emptyset) \)

Step 7. Use JOIN on this multiple block partition and repeat the procedure
of step 6. The resulting count, partitions and figure of merit
for each partition are given.

<table>
<thead>
<tr>
<th>Count</th>
<th>Partitions</th>
<th>Figure of Merit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>( P_{31} = [(1354), (26)] )</td>
<td>( F_{31} = 27 )</td>
</tr>
<tr>
<td>0001</td>
<td>( P_{32} = [(135), (246)] )</td>
<td>( F_{32} = 33 )</td>
</tr>
<tr>
<td>0010</td>
<td>( P_{33} = [(1364), (25)] )</td>
<td>( F_{33} = 20 )</td>
</tr>
<tr>
<td>0011</td>
<td>( P_{34} = [(136), (245)] )</td>
<td>( F_{34} = 26 )</td>
</tr>
<tr>
<td>0100</td>
<td>( P_{35} = [(154), (236)] )</td>
<td>( F_{35} = 19 )</td>
</tr>
<tr>
<td>0101</td>
<td>( P_{36} = [(15), (2346)] )</td>
<td>( F_{36} = 25 )</td>
</tr>
<tr>
<td>0110</td>
<td>( P_{37} = [(164), (235)] )</td>
<td>( F_{37} = 22 )</td>
</tr>
<tr>
<td>0111</td>
<td>( P_{38} = [(16), (2345)] )</td>
<td>( F_{38} = 28 )</td>
</tr>
</tbody>
</table>

Since \( P_{35} \) has the least figure of merit it is chosen to generate
the third internal state variable.

\[ y_3 : [(154), (236)] \]  \hspace{1cm} (5)

Now that each block of multiple partition in (4) has a single
element, the minimum variable initial assignment has been
obtained and is given below.

\[ y_1 : [(123), (456)] \]
\[ y_2 : [(124), (356)] \]
\[ y_3 : [(145), (236)] \]
B. TRAPAGAL: A Transition Path Generation Algorithm

The transition path generation algorithm presented in this section is at the heart of the iterative state assignment method. None of the available UTS assignment techniques have incorporated an algorithm, as flexible, powerful and efficient as TRAPAGAL, for generation of totally sequential ML transition paths. TRAPAGAL can rightly be classified as a universal transition path generation algorithm since it can be used with the same effectiveness for generating minimum length transition paths for completely or incompletely specified flow tables with universal or nonuniversal UTS assignments.

Definition: Consider an n-variable internal state assignment for an r-row flow table \((n \geq \log_2 r)\). \(2^n - r\) unused binary codes are assigned to spare states which initially correspond to state table \([10]\) rows with unspecified next-state entries.

Definition: If a transition is required between a pair of flow table states \(S_a\) and \(S_b\), then the pair \((S_a, S_b)\) is called a transition pair.

TRAPAGAL generates minimum length transition paths on a per column basis without crossover. In constructing a transition path for a given transition pair \((S_a, S_b)\), the algorithm considers the use of unspecified next-state flow table entries, spare states and remaining elements of a k-set containing \(S_a\) and \(S_b\). Three techniques of varying complexity are presented. Although they do not generate a complete set of transition paths for all known valid UTS assignments,
they have been highly successful on a large number of test problems. These techniques are relatively simple and are embodied in a very efficient and short running computer program. Each of the techniques can either be used independently or in conjunction with any of the remaining techniques. The three techniques are

1. COUNT
2. COUNT-DEMAND
3. PATH

The COUNT-DEMAND is an extension of COUNT while PATH is completely independent of and different from the two. Each of these techniques has been individually debugged and tested. Because of the simplicity of COUNT, it has the shortest execution time while incorporation of a near exhaustive reservation scheme for PATH makes it considerably slower. The algorithmic structure for both the COUNT and PATH has been so chosen that those examples for which COUNT cannot generate transition paths, PATH generates them very efficiently, while for those examples for which PATH needs an extensive search, COUNT generates transition paths very efficiently. Such a complimentary property for these techniques is one of the most powerful characteristics of TRAFAGAL.

Definition: A transition $S_a \rightarrow S_b$ under an input column $I$ will be called a 1-transition with respect to variable $y_i$ if $y_i = 0$ for $S_a$ and $y_i = 1$ for $S_b$ and will be called a 0-transition if $y_i = 1$ for $S_a$ and $y_i = 0$ for $S_b$. 
Definition: If $Y_1, Y_2 \ldots \ldots, Y_n$ are $n$ internal state variables for an $n$ bit assignment then $1$-count ($0$-count) for the internal state variable $y_i$ is the number of states, with specified next state entries, in which $y_i = 1$ ($y_i = 0$).

1. **COUNT**

Consider a transition between states $S_a$ and $S_b$ distance $d$ apart. This transition can be realized through a sequence of changing a single internal state variable at a time. Each such change will be called a unit transition. The minimum length transition path between $S_a$ and $S_b$ requires $(d-1)$ intermediate states and $d$ unit transitions. These may all be $1$-transitions, $0$-transitions or a combination of both. Successful completion of the transition path depends upon the proper ordering of the $d$ unit transitions. A decision scheme which will properly order these individual unit transitions such that the resulting intermediate states are directed away from states of other $k$-sets and near or into states of the same $k$-set or unspecified states (states with unspecified next state entries) is desirable.

**COUNT** is based on the above scheme. A count list of $1$-count and $0$-count is formed for all the internal state variables of the given assignment. The transitions are ordered in an increasing order of distance between the initial and the final states under an input column of the flow table. This means that the transitions between adjacent states are followed by those between states distance $2$ apart and so on.
For any transition between states distance \( d \) apart (\( d > 1 \)), 1-count and 0-count are known from the count list. The \( d \) unit transitions can therefore be ordered by ordering the numerical values of these counts. This establishes the order in which the \( d \) unit transitions are made. After each of these unit transitions is completed the transition path constructed so far is checked for crossover with all other completed transitions paths under the same input column. The next unit transition is attempted only when the previous unit transition has been successfully constructed. In the event of a tie in counts, the internal state variables are changed in a cyclic order. If this results in an invalid intermediate state, then one attempts the change of variables in a different cyclic order by using permutation if necessary. The attempt is terminated when either a valid transition path results or all cyclic changes of the relevant state variables do not yield a valid transition path. For example if \( S_a \) and \( S_b \) differ in state variables \( y_1, y_2 \) and \( y_3 \) then one first attempts a transition by changing \( y_1, y_2 \) and \( y_3 \), in that order. If a valid transition path does not result then any of the remaining cycles and permutations of \( y_1, y_2, y_3 \) are tried. Upon successful completion of a transition path, the "count list" is modified and the remaining transitions are considered.
Example #3: Refer to the flow table column of Figure 5.

\[
\begin{array}{cc|c}
Y_1 & Y_2 & Y_3 & I_1 \\
000-1 & 3 \\
001-2 & 5 \\
101-3 & 3 \\
011-4 & 5 \ldots \\
110-5 & 5 \\
100-6 & \\
\end{array}
\]

Figure 5. Flow Table Column With a Distinct Binary Code for Each Flow Table Row.

Step 1: Form the count list using the given assignment.

**Count list**

<table>
<thead>
<tr>
<th>Bit position</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-count</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0-count</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 2: Order transitions using distance between the initial and final states of a transition pair.

**Stable states**

3, 5

**Distance 2 transitions**

1 \rightarrow 3 T_{21}

4 \rightarrow 5 T_{22}

**Distance 3 transitions**

2 \rightarrow 5 T_{31}

where \( T_{ij} \) is the jth, distance i transition.

Step 3: For transition \( T_{21} \) states '1' and '3' differ in bit positions 1 and 3 and both are 1-transitions. From the count list the order of complementing is bit position 1 followed by bit position 3.

000 \rightarrow 100 bit position 1 is complemented.
'100' being an unspecified state is a valid intermediate state. Hence bit position 3 is complemented generating the following transition path:

$$000 \rightarrow 100 \rightarrow 101$$

**Step 4:** Modify the count list to include intermediate state '100'.

**Modified count list**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-count</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0-count</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**Step 5:** For transition $T_{22}$ states '4' and '5' differ in bit positions 1 and 3. Bit position 1 is a 1-transition and bit position 3 is a 0-transition. Count list indicates that both the counts are equal hence a note is made and bit position 1 is complemented first.

$$011 \rightarrow 111$$ bit position 1 complemented.

Since '111' is a spare state it is a valid intermediate state and construction of the transition path can be continued by complementing bit position 3.

$$011 \rightarrow 111 \rightarrow 110$$

**Step 6:** Modify the count list to include intermediate state '111'.
Modified count list

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-count</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0-count</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Step 7: For transition $T_{31}$ states '2' and '5' differ in all three bit positions. Bit positions 1 and 2 are 1-transitions while bit position 3 is 0-transition. This results in an ordering of complementing bits 2, 3 and 1.

001 → 011  bit position 2 complemented.

'011' is a state with specified next state entry; however, it belongs to the same k-set as the transition under consideration and is therefore a valid intermediate state. Transition 011 → 110 has already been completed giving the following transition path for $T_{31}$:

001 → 011 → 111 → 110

This completes generation of all the transition paths.

2. COUNT-DEMAND

The principal difficulty with COUNT is that tie counts are frequently encountered and with limited look-ahead capability, considerable trial and error can be involved in the generation of a valid transition path. A demand routine has been added which assigns weights to all candidates for the next step in a transition path. This weighting scheme considerably improves the decision making capability of COUNT. The combination of a demand routine with COUNT
gives the COUNT-DEMAND technique.

In COUNT-DEMAND all specified states (states with specified next state entries) are assigned a fixed arbitrary demand of, say 200. A count list is formed and transitions are ordered as before. All states adjacent to the initial state $S_a$ and obtained by complementing one bit out of $d$ bits are given an additional demand of 1. A demand list is formed. The count list and demand list are merged to form a count-demand list. Based on this, proper order of complementing the bits is determined. After successful completion of a transition, the count list is modified, the intermediate states are assigned a demand of 200 and a new count-demand list is formed to initiate the next transition.

Example #4: Refer to the flow table column of Figure 6.

```
Y_1Y_2Y_3   I_1
000-1       5
011-2       3
100-3        3   ... 
111-4        4
101-5        5
```

Figure 6. Flow Table Column With a Distinct Binary Code for Each Flow Table Row.

Step 1: Form the count list using the given assignment.
Count list

Bit Position  1  2  3
1-count       3  2  3
0-count       2  3  2

Step 2: Order transitions using distance between the initial and final states of a transition pair.

Stable states 3, 4, 5

Distance 2 transition $1 \rightarrow 5$ $T_{21}$
Distance 3 transition $2 \rightarrow 3$ $T_{31}$

where $T_{ij}$ is the $j$th, distance $i$ transition.

Step 3: For transition $T_{21}$ states '1' and '5' differ in bit positions 1 and 3 and both are 1-transition. One gets state '100' by complementing bit position 1 and state '001' by complementing bit position 3. Assign an additional demand of 1 to both these states, and generate the demand list.

Demand list

Bit Position changed  1  3
Demand               201  1

Step 4: Combining the count list and the demand list, form the count-demand list for transition $T_{21}$.

Count-demand list

Bit Position changed  1  3
Count-demand        204  4

From the count-demand list proper ordering of complementing bit positions is 3, 1. This results in the following transition
Step 5: Modify the count list and assign a demand of 200 to the intermediate state '001'.

**Modified count list**

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-count</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>0-count</td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 6: For transition $T_{31}$ states '2' and '3' differ in all three bit positions. Bit position 1 is a 1-transition and bit positions 2 and 3 are 0-transitions. Repeating the procedure outlined in step 3, the following demand list is obtained.

**Demand list**

<table>
<thead>
<tr>
<th>Bit position changed</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand</td>
<td>201</td>
<td>201</td>
<td>1</td>
</tr>
</tbody>
</table>

Combining the count list and the demand list, form the count-demand list for transition $T_{31}$.

**Count-demand list**

<table>
<thead>
<tr>
<th>Bit position changed</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count-demand</td>
<td>204</td>
<td>205</td>
<td>3</td>
</tr>
</tbody>
</table>

Hence the proper ordering for complementing the bit positions is 3, 1 and 2. This results in a valid transition path given below:

000 $\rightarrow$ 001 $\rightarrow$ 101
3. PATH

The COUNT-DEMAND technique considerably improves the effectiveness of the decision scheme used by COUNT. However since the demands are determined for only one transition at a time, the order in which these transitions are attempted has a considerable effect on successful completion of all transitions under an input column. The main advantage of COUNT-DEMAND over the COUNT is that by modifying the count list one considerably improves the decision scheme used by COUNT.

The PATH incorporates a near exhaustive reservation scheme by considering simultaneously the claim of all transitions within a flow table column on all the states. All such claims are added to generate a net demand for each state. Instead of ordering transitions by considering the distance between transition pairs as in COUNT and COUNT-DEMAND, PATH orders these on the basis of the number of valid minimum length transition paths for each transition.

Initially all the specified states are assigned an arbitrarily high demand of 200. For each transition, all minimum length transition paths are generated. Every intermediate state in these transition paths is given an additional demand of 1. This is repeated for all transitions. By adding the demands for each state a demand array is obtained. The entries in this array indicate the demands on all states by considering in parallel all transitions under any input column.
All minimum length transition paths for a particular transition are now known. Intermediate states in each of these transition paths may consist of specified states of the same k-set (as the transition under consideration), specified states of other k-sets and unspecified states. Because of the basic weighting scheme all specified states are assigned an initial demand of 200. However, specified states belonging to the same k-set are valid intermediate states. Hence when such a state appears in a transition path its demand is modified (and is taken as the demand from the demand array less 200) to distinguish it from specified states of other k-sets. A product demand is now formed for each transition path by multiplying the demand of each of the intermediate states in this path.

By repeating the procedure given above, product demands for all transition paths within a flow table column are generated. It is obvious that valid transition paths would be those that have a product demand of less than 200. For all transitions, a count of valid transition paths is maintained. All transitions within an input column are ordered according to increasing counts.

One does not have any choice for completing transitions with a single valid transition path, while considerable flexibility is available for transitions as number of valid transition paths increase. Therefore transitions with only one valid path are considered first followed by those with two such paths and so on. For transitions with multiple
valid paths, the one with the least product demand is chosen. If all
valid paths for a transition have the same product demand, this tran-
sition is temporarily skipped and is reconsidered after all the remaining
transitions are attempted. After construction of valid transition path
is tried once for each transition if there are still some transitions with
multiple valid paths with the same product demand, then a transition
with minimum number of such paths is chosen and any of the multiple
valid paths is selected arbitrarily. However, if all the remaining
transitions have the same number of valid paths with equal product
demand, any of these transitions is chosen and any of the multiple
valid paths is selected arbitrarily.

As each transition path is constructed those unspecified states
appearing as intermediate states in this transition path are assigned
proper next-state entries and an additional demand of 200 to distin-
guish them from the remaining unspecified states. Once a transition
has been successfully completed, demand array is modified so as to
reflect demands only for the remaining incomplete transitions. The
procedure is repeated for all transitions.

Example #5: Refer to the flow table column of Figure 7.
Step 1: Assign a demand of 200 to all the specified states and a demand of zero to all the remaining states. The initial values for the demand array are:

<table>
<thead>
<tr>
<th>States</th>
<th>Demand</th>
<th>States</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>200</td>
<td>1100</td>
<td>200</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>200</td>
<td>1111</td>
<td>200</td>
</tr>
<tr>
<td>0010</td>
<td>200</td>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1010</td>
<td>200</td>
</tr>
<tr>
<td>0111</td>
<td>200</td>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>200</td>
<td>1001</td>
<td>200</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>
Step 2: The following transitions are to be completed.

1 → 2, 3 → 4, 5 → 7, 8 → 2, 9 → 10

Step 3: Generate all minimum length transition paths for transition 1 → 2.

0000 → 0100 → 0101 \( P_{11} \)
0000 → 0001 → 0101 \( P_{12} \)

where \( P_{ij} \) indicates jth path for ith transition. Modify the demand array by adding a demand of 1 to the accumulated demand of the states in the generated paths. The modified demand array after generating minimum length transition paths for transition 1 → 2 is given below:

<table>
<thead>
<tr>
<th>States</th>
<th>Demand</th>
<th>States</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>200</td>
<td>1100</td>
<td>200</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>200</td>
<td>1111</td>
<td>200</td>
</tr>
<tr>
<td>0010</td>
<td>200</td>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1010</td>
<td>200</td>
</tr>
<tr>
<td>0111</td>
<td>200</td>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>200</td>
<td>1001</td>
<td>200</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

Step 4: Repeat step 3 for all the remaining transitions and generate a demand array which takes into account the demands of all transitions simultaneously. The following minimum length transition paths will be generated.
Paths for transition 3 → 4:

1100 → 1101 → 1111 → 0111  3 → 4
1100 → 1101 → 0101 → 0111  3 → 4
1100 → 0100 → 0110 → 0111  3 → 4
1100 → 0100 → 0101 → 0111  3 → 4
1100 → 1110 → 1111 → 0111  3 → 4
1100 → 1110 → 0110 → 0111  3 → 4

Paths for transition 5 → 7:

1001 → 1011 → 0011  5 → 7
1001 → 0001 → 0011  5 → 7

Paths for transition 8 → 2:

0010 → 0000 → 0001 → 0101  8 → 2
0010 → 0000 → 0100 → 0101  8 → 2
0010 → 0110 → 0111 → 0101  8 → 2
0010 → 0110 → 0100 → 0101  8 → 2
0010 → 0011 → 0001 → 0101  8 → 2
0010 → 0011 → 0111 → 0101  8 → 2

Paths for transition 9 → 10:

1111 → 1011 → 1010  9 → 10
1111 → 1110 → 1010  9 → 10

Based on these transition paths, the final demand array is given below:

<table>
<thead>
<tr>
<th>States</th>
<th>Demand</th>
<th>States</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>202</td>
<td>1100</td>
<td>200</td>
</tr>
<tr>
<td>0001</td>
<td>4</td>
<td>1101</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>202</td>
<td>1111</td>
<td>202</td>
</tr>
<tr>
<td>0010</td>
<td>200</td>
<td>1110</td>
<td>3</td>
</tr>
<tr>
<td>0110</td>
<td>4</td>
<td>1010</td>
<td>200</td>
</tr>
<tr>
<td>0111</td>
<td>202</td>
<td>1011</td>
<td>2</td>
</tr>
<tr>
<td>0101</td>
<td>202</td>
<td>1001</td>
<td>200</td>
</tr>
<tr>
<td>0100</td>
<td>5</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>
Step 5: From the final demand array form product demand for all the transition paths listed above. Let \( d_{ij} \) denote the product demand for the \( j \)th path of the \( i \)th transition. Then:

1) \( d_{11} = 5, \ d_{12} = 4 \)
2) \( d_{21} = 2 \times 202, \ d_{22} = 2 \times 202, \ d_{23} = 5 \times 4 \)
   \( d_{24} = 5 \times 202, \ d_{25} = 3 \times 202, \ d_{26} = 3 \times 4 \)
3) \( d_{31} = 2, \ d_{32} = 4 \)
4) \( d_{41} = (202 - 200) \times 4, \ d_{42} = (202 - 200) \times 5 \)
   \( d_{43} = 4 \times 202, \ d_{44} = 4 \times 5, \ d_{45} = 202 \times 4 \)
   \( d_{46} = 202 \times 202 \)

*It can be seen that intermediate state "0000" in paths \( P_{41} \) and \( P_{42} \) is a specified state leading to stable state '0101' which is the stable state for the transition under consideration. Therefore the demand for '0000' is equal to the demand from the demand array less 200.

5) \( d_{51} = 2, \ d_{52} = 3 \)

Let \( K_i \) denote the number of transition paths for the \( i \)th transition for which the product is less than 200. Then:

\( K_1 = 2 \) and the paths are \( P_{11} \) and \( P_{12} \)
\( K_2 = 2 \) and the paths are \( P_{23} \) and \( P_{26} \)
\( K_3 = 2 \) and the paths are \( P_{31} \) and \( P_{32} \)
\( K_4 = 3 \) and the paths are \( P_{41}, P_{42} \) and \( P_{44} \)
\( K_5 = 2 \) and the paths are \( P_{51} \) and \( P_{52} \).
Step 6: Ordering of the transitions is 1, 2, 3, 5, and 4. For transition 1 & 2 path \( P_{12} \) has the least product demand and it is used to complete this transition.

\[ 1 \rightarrow 2: 0000 \rightarrow 0001 \rightarrow 0101 \]

The demand array is modified by subtracting the demands for transition 1 & 2. The intermediate state '0001' is given an additional demand of 200 to distinguish it from the remaining unspecified states. The modified demand array is given below:

<table>
<thead>
<tr>
<th>States</th>
<th>Demand</th>
<th>States</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>201</td>
<td>1100</td>
<td>200</td>
</tr>
<tr>
<td>0001</td>
<td>203</td>
<td>1101</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>201</td>
<td>1111</td>
<td>202</td>
</tr>
<tr>
<td>0010</td>
<td>200</td>
<td>1110</td>
<td>3</td>
</tr>
<tr>
<td>0110</td>
<td>4</td>
<td>1010</td>
<td>200</td>
</tr>
<tr>
<td>0111</td>
<td>202</td>
<td>1011</td>
<td>2</td>
</tr>
<tr>
<td>0101</td>
<td>202</td>
<td>1001</td>
<td>200</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1000</td>
<td>0</td>
</tr>
</tbody>
</table>

Step 7: Repeating steps 5 and 6, all the transitions are successfully completed. The valid transition paths in the order of their generation are given below:

\[ 1 \rightarrow 2: 0000 \rightarrow 0001 \rightarrow 0101 \]
\[ 5 \rightarrow 7: 1001 \rightarrow 1011 \rightarrow 0011 \]
\[ 9 \rightarrow 10: 1111 \rightarrow 1110 \rightarrow 1010 \]
\[ 3 \rightarrow 4: 1100 \rightarrow 0100 \rightarrow 0110 \rightarrow 0111 \]
\[ 8 \rightarrow 2: 0010 \rightarrow 0000 \rightarrow 0001 \rightarrow 0101 \]
C. AAA: An Algorithm to Augment an Assignment

This algorithm consists of adding a variable to an assignment so that the augmented assignment will either be a valid UTS assignment or result in the completion of a near maximum number of the unsuccessful transitions. Three different techniques are included. These are:

1. STRUCTUR
2. GAIN
3. MIXMÖD

STRUCTUR was developed by Maki [8] for realizing a non-universal UTS assignment. GAIN is a modification of STRUCTUR while MIXMÖD uses an entirely different approach to augment an assignment.

1. STRUCTUR

This is based on the relationship between flow table and its initial assignment.

Definition: A set of state variables \( \{y_1, y_2, \ldots, y_k\} \) is an independent set of state variables if no \( y_i \) in the set is equal to the mod 2 sum of a subset of the other state variables of the set.

STRUCTUR is used only when all valid transition paths cannot be constructed, therefore there is a good chance of it being used immediately after the initial assignment is tested for validity. During the first iteration of STRUCTUR a count list is formed for each state variable.
of the initial assignment. The count list contains a count of number of times each state variable changes when all distance two and more transitions are realized. Next the state variables are ordered in a decreasing order of the counts.

For the first and all subsequent iterations of STRUCTUR the first two variables in this list are considered to be independent state variables and using mod 2 operation a third, dependent state variable is produced. The independent variables used to produce the dependent state variables are deleted from the ordered list of variables to eliminate any conflicts in parity sets during any subsequent iteration.

This algorithm has been programmed. For further details refer to [8].

![Figure 8. Flow Table for Example #'s 6, 7, 8 and 9.](image)
### Internal State Variables


d| \( y_1 \) | \( y_2 \) | \( y_3 \) | \( y_4 \) |
---|---|---|---|---|
1 | 0 | 0 | 0 | 0 |
2 | 0 | 0 | 0 | 1 |
3 | 0 | 1 | 1 | 0 |
4 | 0 | 0 | 1 | 0 |
5 | 0 | 1 | 0 | 1 |
6 | 1 | 0 | 0 | 1 |
7 | 1 | 0 | 0 | 0 |
8 | 1 | 1 | 1 | 0 |
9 | 1 | 1 | 0 | 1 |
10 | 1 | 0 | 1 | 1 |

*Figure 9. Initial Assignment for the Flow Table of Figure 8.*

Example # 6: Refer to the flow table of Figure 8 and initial assignment of Figure 9.

**Step 1.** Form a count list of number of times each variable changes when all distance two and more, transitions are attempted.

| State variable | \( y_1 \) | \( y_2 \) | \( y_3 \) | \( y_4 \) |
---|---|---|---|---|
| # of times changed | 6 | 10 | 10 | 8 |

**Step 2.** Since \( y_2 \) and \( y_3 \) vary a maximum number of times add a variable \( y_5 \) such that \( y_5 = y_2 \oplus y_3 \). The augmented assignment is given in Figure 10.
2. **GAIN**

STRUCTUR does not capitalize on the knowledge of the limitation of initial assignment in completing some of the transitions, but adds a variable by analyzing initial assignment in relation to all transitions within a flow table. Since unsuccessful transitions are only indirectly involved in this approach addition of a variable does not necessarily guarantee the completion of all unsuccessful transitions. GAIN, on the other hand, uses the information provided by unsuccessful transitions to add a state variable. Whenever AAA is called, a list of unsuccessful transitions is supplied to it. GAIN uses this information to form a count list similar to the one generated in STRUCTUR. After
ordering this count list, in a decreasing order of counts, the first two
state variables in this list are used to generate a new variable. The
augmented assignment is now tested for validity.

Since GAIN takes into account unsuccessful transitions directly,
it is fair to say that the augmented assignment obtained by using GAIN
has a better chance of success in completing unsuccessful transitions
than an augmented assignment obtained using STRUCTUR. Another
obvious advantage of GAIN emerges from the fact that the number of
unsuccessful transitions are a very small percentage of the total
number of transitions within a flow table. Therefore GAIN is much
faster in running time than STRUCTUR.

Example #7. Refer to the flow table of Figure 8 and initial assignment
of Figure 9.

Step 1: List all transitions that cannot be constructed using initial/
augmented assignment. Form a count list of the number of
times each internal state varies for each such transition.

<table>
<thead>
<tr>
<th>Internal State Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>57</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>107</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Unsuccessful transitions</td>
</tr>
<tr>
<td>( y_1 )</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Step 2: Since \( y_1 \) and \( y_4 \) are excited a maximum number of times, add a variable \( y_5 \) such that \( y_5 = y_1 \oplus y_4 \). The augmented assignment is given in Figure 11.

<table>
<thead>
<tr>
<th>Internal State Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

Figure 11. Augmented Assignment for the Flow Table of Figure 8 using GAIN.

3. MIMMOD

The main advantage of using mod 2 operation in generating a new variable is that, for a given assignment, the maximum distance between states of any transition pair remains unchanged. This however does not mean that if the distance between states of a transition pair is less than maximum, then this distance will not increase. An increase in distance between the states of a transition pair results in a longer transition path and consequent increase in transition time. One would therefore like to add a variable so that
i) a maximum number of unsuccessful transitions can be completed and

ii) the distance between the majority of transition pairs is unchanged.

MIXMØD first uses a dichotomy approach to complete as many of the unsuccessful transitions as possible and then goes on to use a covering process to partially satisfy the second requirement. The dichotomy covering approach has an inherent property of permitting direct transitions, therefore one can easily see that MIXMØD not only provides a method to add a variable to augment an assignment but also allows speeding up of some of the totally sequential transitions in the process. A step by step description of the algorithm follows.

Step 1: From the given assignment obtain the list of those transitions that cannot be completed.

Step 2: Under each column of the flow table all those dichotomies that do not satisfy Tracey conditions [1] are listed. If any of these dichotomies includes transitions that could not be completed, such dichotomies are starred.

Step 3: Dichotomies listed under each input column are covered. All those input columns with only one such cover are ordered in a decreasing order of the number of starred dichotomies. If none of the input columns with just one cover has any starred dichotomies, an input column with
maximum number of starred dichotomies is chosen and a
cover is obtained to include a maximum number of these.

Step 4: The cover obtained in step 3 is now enlarged to include
(i) as many of the remaining starred dichotomies,
(ii) as many of other dichotomies
and (iii) to ensure that the maximum distance between as
many of the maximum or near maximum distance
transitions remains unchanged when a new state
variable is added.

Step 5: The enlargement of a cover is discontinued when the set
union of both blocks of the cover equals the total state
set of flow table. A new internal state variable is added
to augment the assignment.

Example #8: Refer to the flow table of Figure 8 and initial assignment
of Figure 9.

Step 1: List those transitions that cannot be completed using the
initial/augmented assignment.

56 in Column $I_1$
57 or 13 in Column $I_2$
16 in Column $I_5$
48 or 107 in Column $I_5$

Step 2: Under each column list all those dichotomies that do not
satisfy the Tracey conditions.
dichotomies denote that one or both blocks of these dichotomies include an unsuccessful transition.

Step 3: Order input columns having only one cover in a decreasing order of number of starred dichotomies. Let this set be $C_1$. Then,

$$C_1 : \{I_5, I_1, I_3, I_4\}$$

Step 4: Cover for $I_5$ is 164895-23107.

Step 5: Since $(164895) \cup (23107) = S$ the total state set, this cover cannot be enlarged any further and a variable is added using this cover. The augmented assignment obtained is shown in Figure 12.
<table>
<thead>
<tr>
<th>Internal States</th>
<th>Internal State Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>y_1 y_2 y_3 y_4 y_5</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 1 1 0 1</td>
</tr>
<tr>
<td>4</td>
<td>0 0 1 0 0</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>6</td>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>7</td>
<td>1 0 0 0 1</td>
</tr>
<tr>
<td>8</td>
<td>1 1 1 0 0</td>
</tr>
<tr>
<td>9</td>
<td>1 1 0 1 0</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 1 1</td>
</tr>
</tbody>
</table>

Figure 12. Augmented Assignment for the Flow Table of Figure 8 using MIXMØD.
D. **ALSPT: An Algorithm to Speed up Transitions**

For a uni-code totally sequential assignment transition between states of a transition pair is realized by changing only one variable at a time. If an average amount of time for each internal state variable to change is $\Delta t$, then for USTT operation average time for each transition between an unstable and stable state would be $\Delta t$. For UTS operation, the average time for each transition would be $n \cdot \Delta t$, where $n$ is equal to the number of intermediate states the circuit sequenced through before arriving at the stable state. It is clear, then, that saving in hardware resulting from fewer number of internal state variables needed for UTS operation results in a much slower circuit.

After a valid UTS assignment is produced all transition paths can be constructed with totally sequential transitions. However if some or all of these transitions could either be realized as direct or if a number of subtransitions in their transition paths could be reduced by using non-critical races without crossover, then resulting transitions would be considerably faster.

Definition: In a mixed mode (MM) transition subtransitions in transition path are realized through a combination of totally sequential transition and non-critical races.

Langdon [14] has defined almost totally sequential (ATS) assignments as those where each subtransition in a transition path may
involve a change in more than one state variable to speed up transition between a transition pair. On the surface there seems to be a lot of similarity between ATS assignments and UTS assignments with MM transitions. However construction of transition paths with MM transitions is much more flexible than those for ATS assignments. Use of MIXMOD to augment an assignment followed by construction of MM transitions will result in a considerable improvement in speed of a circuit realization of a given flow table and will go a long way in reducing the main limitation of UTS assignments.

The algorithm to speed up transitions generates transition paths on per column basis. A step by step description for column $I_t$ follows:

Step 1: All dichotomies that do not satisfy the Tracey conditions are listed for this column.

Step 2: Let $C_i$ be the set of transitions appearing $i$ times in the list of dichotomies in step 1. Transitions in each $C_i$ are ordered in a decreasing order of distance between transition pairs.

Step 3: PATH is used to generate totally sequential transition paths for all transitions within this flow table column.

Step 4: From the totally sequential path for all transitions, a 1-cube for each subtransition is formed. For example, for a distance 3 transition $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4$ all 1-cubes for $s_1 \rightarrow s_2$, $s_2 \rightarrow s_3$ and $s_3 \rightarrow s_4$ are generated.
Step 5: Consider, $ij$, the first transition in set $C_1$. Let $(ij, mn)$ be the dicotomy that does not satisfy the Tracey conditions. Since every element of $C_1$ is involved in only a single dichotomy that does not satisfy the Tracey conditions, one has to consider only the intersection between subcubes for the transition being considered and its incompatible counterpart $mn$. If each subtransition of $ij$ is a p-cube, then all $(p+1)$-cubes are formed. Each of these $(p+1)$-cubes is intersected with each subtransition cube of $mn$, using intersection rules given in Figure 13. $\emptyset$ indicates a null intersection between two subcubes. An intersection between two subcubes is valid if

i) an intersection is null,

ii) an intersection contains don't care state(s),

or

iii) an intersection contains state(s) that are elements of the same $k$-set as the present transition.

Any $(p+1)$-cube having a valid intersection with each transition subcube of its incompatible counterpart can be

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$\emptyset$</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 13. Intersection Rules for P-cubes
realized as a direct transition.

Each such enlarged subcube replaces the transition subcubes from which it was obtained. If more than one valid (p+1) cube could be realized, these in turn are enlarged to form (p+2) cubes and same intersection process is repeated. The process is terminated when no more subcubes could be enlarged. At this point one has constructed a mixed mode transition path for transition ij, and therefore transition ij is removed from set C₁ and next element of C₁ is chosen. If C₁ is empty then the first element of the next set is considered.

It is possible that none of the (p+1) cubes for transition ij could result in a valid intersection with transition subcubes of mn. Under this condition one considers the next element of set C₁ and so on. However if the same results are obtained for all transitions in all the sets then none of the transitions can be realized as mixed mode transition paths and one must be content with totally sequential transition paths.

Step 6: Whenever any transition is realized with a mixed mode transition path all intermediate states in this path are assigned proper destinations and again PATH is called upon to generate all remaining transition paths. Now one uses relevant
portions of step 5 and step 6 iteratively till all transitions are considered.

\[
\begin{array}{c|c}
Y_1Y_2Y_3Y_4 & I_1 \\
0000 & 2 \\
0111 & 2 \\
0010 & 4 \\
1100 & 4 \\
1001 & 6 \\
1010 & 6 \\
\end{array}
\]

Figure 14. Flow Table Column with a Valid UTS Assignment for Example #9

Example #9: Refer to flow table column with a valid UTS assignment of Figure 14.

Step 1: List all dichotomies that do not satisfy Tracey conditions.

12-34, 56-34.

Step 2: Order transitions in sets on the basis of number of appearances of each in the dichotomies listed in step 1. Let \( C_i \) denote the set of transitions appearing \( i \) times. Then,

\( C_1 = \{12, 56\} \) and \( C_2 = \{34\} \)

Step 3: Call PATH to generate totally sequential transition path for each transition. These paths are given below.

\[1 \rightarrow 2, \quad 0000 \rightarrow 0100 \rightarrow 0101 \rightarrow 0111\]
3 \rightarrow 4: \quad 0010 \rightarrow 0110 \rightarrow 1110 \rightarrow 1100
5 \rightarrow 6: \quad 1001 \rightarrow 1000 \rightarrow 1010

Step 4: Form subcubes for each subtransition.

1 \rightarrow 2, \quad 0000 \rightarrow 0100 : 0X00
0100 \rightarrow 0101 : 010X
0101 \rightarrow 0111 : 01X1

3 \rightarrow 4, \quad 0010 \rightarrow 0110 : 0X10
0110 \rightarrow 1110 : X110
1110 \rightarrow 1100 : 11X0

5 \rightarrow 6, \quad 1001 \rightarrow 1000 : 100X
1000 \rightarrow 1010 : 10X0

Step 5: (i) Consider the first transition in $C_1$. Generate all 2-cubes.

Since the subcube of transition 12 intersects the subcube
of transition 34, intersect all 2-cubes for transition 12 with
each subtransition subcube for transition 34. Two 2-cubes
for transition 12 are:

0000 \rightarrow 0101 : 0X0X
0100 \rightarrow 0111 : 01XX

Intersect 0X0X: \quad 0X0X \cap 0X10 = \emptyset
0X0X \cap X110 = \emptyset
0X0X \cap 11X0 = \emptyset

Since the intersection is null 0X0X is a valid 2-cube.

Intersect 01XX: \quad 01XX \cap 0X10 = 0110
01XX \cap X110 = 0110
01XX \cap 11X0 = \emptyset

This is not a valid 2-cube since 0110 is an intermediate state
in a transition path for a transition in a different k-set. Now
one of the two 2-cubes is valid hence transition 12 can be
realized as a mixed mode transition as follows:
\[ X \]

1 \rightarrow 2, \ 0000 \rightarrow 0101 : 0X0X
0101 \rightarrow 0111 : 01X1

\[ X \]

where \(-\) indicates a direct subtransition. Replace original subtransitions for \(1 \rightarrow 2\) with those given above and delete transition \(12\) from \(C_1\).

**Step 6:** Assign proper next-state entries for all intermediate states of the transitions just completed and call \(\text{PATH}\) to generate transition paths for the remaining transitions.

In this case \(\text{PATH}\) generates the same transition paths as before for transitions \(34\) and \(56\). Therefore their sub-transitions and 1-cubes are repeated while subtransition subcubes for transition \(12\) are those obtained in step 5.

\[ 3 \rightarrow 4, \ 0010 \rightarrow 0110 : 0X10 \]
0110 \rightarrow 1110 : X110
1110 \rightarrow 1100 : 11X0

\[ 5 \rightarrow 6, \ 1001 \rightarrow 1000 : 100X \]
1000 \rightarrow 1010 : 10X0

\[ X \]

1 \rightarrow 2, \ 0000 \rightarrow 0101 : 0X0X
0101 \rightarrow 0111 : 01X1

(ii) Consider transition \(56\) in \(C_1\). Generate the only 2-cube. Since the subcube of transition \(56\) intersects with the sub-cube of transition \(34\), intersect this 2-cube with each of the 1-cubes of transition \(34\).

\[ 1001 \rightarrow 1010 : 10XX \]

Intersect \(10XX\) : 
\[ 10XX \cap 0X10 = \phi \]
\[ 10XX \cap X110 = \phi \]
\[ 10XX \cap 11X0 = \phi \]
The intersection results in a null set, hence $10XX$ is a valid 2-cube and since this was the only 2-cube, transition 56 can be realized directly as follows.

$1001 \overset{X}{\rightarrow} 1010$.

Replace the original subtransitions for $5 \rightarrow 6$ with that given above and delete transition 56 from $C_1$. Assign the proper next-state entries for all intermediate states of the transitions just completed and call PATH to generate transition paths for the remaining transitions. PATH generates the same transition path as before for the only remaining transition 34 and therefore its subtransitions and 1-cubes are repeated while subtransition subcubes for transitions 12 and 56 are appropriately modified.

$3 \rightarrow 4, \quad 0010 \rightarrow 0110 : 0X10$
$\qquad \quad \quad \quad 0110 \rightarrow 1110 : X110$
$\qquad \quad \quad \quad 1110 \rightarrow 1100 : 11X0$

$1 \rightarrow 2, \quad 0000 \overset{X}{\rightarrow} 0101 : 0X0X$
$\quad \quad \quad \quad \quad 0101 \rightarrow 0111 : 01X1$

$5 \rightarrow 6, \quad 1001 \overset{X}{\rightarrow} 1010 : 10XX$

(iii) Since $C_1$ is empty, select first the transition 34 in $C_2$. Generate all 2-cubes. The subcube of transition 34 intersects the subcube of transitions 12 and 56 hence intersect each subcube of 34 with each subtransition subcube of transitions 12 and 56.
Intersect XX10 with subtransition subcubes of transition 12:

\[ XX10 \cap 0X0X = \emptyset \]
\[ XX10 \cap 01X1 = \emptyset \]

Intersect XX10 with subtransition subcubes of transition 56

\[ XX10 \cap 10XX = 1010 \]

Since 1010 is a stable state, the intersection is not valid and therefore XX10 is not a valid 2-cube.

Intersect X1X0 with subtransition subcubes of transition 12

\[ X1X0 \cap 0X0X = 0100 \]
\[ X1X0 \cap 01X1 = \emptyset \]

0100 is an intermediate state in a transition path of different k-set. Hence X1X0 is not a valid 2-cube.

Now both 2-cubes are not valid, this means that transition 34 cannot have a mixed mode transition path.

List all transitions along with their transition paths:

1 \rightarrow 2, 0000 \overset{X}{\rightarrow} 0101 \rightarrow 0111

3 \rightarrow 4, 0010 \rightarrow 0110 \rightarrow 1110 \rightarrow 1100

5 \rightarrow 6, 1001 \overset{X}{\rightarrow} 1010.
III. SUMMARY AND DISCUSSION

The research effort for this project was mainly concentrated around developing a generalized but extremely powerful method to construct minimum-length transition paths. Of the very few methods to generate transition paths that have been reported in literature some are useful for only a restricted class of flow tables and assignments while others are useful in producing a particular type of internal state assignment. In the context of these available methods a question always arises: given a valid unicode totally sequential assignment either universal or nonuniversal, how does one systematically produce transition paths without crossover?

The method developed, as a result of this research, eliminates this limitation entirely and provides a universal transition path generation algorithm for any type of UTS assignment. Three independent algorithms are included in this method. These algorithms are of varying complexity and the computer running time for each is directly proportional to the complexity of the particular algorithm. COUNT is the simplest and has a shorter running time while PATH incorporates an exhaustive search technique and consequently has a longer running time. However these two algorithms have a complimentary characteristic in that for a class of flow tables for which COUNT may not be able to generate a solution, PATH finds one in comparatively shorter time while for classes of flow tables for which PATH needs use of all
its near exhaustive search techniques, COUNT produces a solution much faster. Based on the diversity of test problems, these algorithms have proved to be extremely efficient and very successful in generating transition paths. However since a totally exhaustive search technique has not been used in any of the algorithms no claim of 100% success is made. It is clear however that incorporation of a totally exhaustive search technique, though feasible and superficially desirable, will not be of practical value since it, in general, will need prohibitively large amounts of computer time.

As a direct consequence of the availability of this almost perfect algorithm it seems possible that by reversing the technique used in this method it could be used to generate state assignments. It is easy to see that the transition path generation method constructs transition paths on per column basis while in order to generate an internal state assignment interaction among transitions under all input columns of a flow table has to be considered simultaneously. Hence transition path generation methods cannot be used as a one step procedure to produce internal state assignment.

Maki [8] proposed an interactive approach in producing non-universal UTS assignments. Such an approach has a limitation in that it cannot guarantee generation of a minimum variable assignment. A minimum variable internal state assignment does not necessarily result in a circuit realization with least cost. Hence in many cases a near-minimal assignment is acceptable and under a certain cost criterion
may indeed be the most desirable assignment. In this context an iterative state assignment approach appears very promising.

Although Maki proposed a broad outline of such an approach and established an upper bound on the number of internal state variables, he failed to systematize all phases of his method. This paper has completely eliminated these limitations by developing a new iterative state assignment method (ISAM). The method incorporates a simple procedure to produce a "good" minimum variable initial assignment and then uses a transition path generation algorithm to determine the validity of this assignment. If the initial assignment is not valid it is augmented by adding a variable. This augmented assignment is again tested for validity. The transition path generation algorithm and the algorithm to augment an assignment are used iteratively until a valid UTS assignment is produced. It can be seen that transition path generation algorithm has been very effectively used in an iterative manner for generating valid UTS assignments and hence can indeed be considered to be at the heart of ISAM.

Since an iterative assignment algorithm produces a near-minimal UTS assignment it has more spare states than a minimum variable UTS assignment. Some of these spare states could easily be used to introduce non-critical races to reduce the number of subtransitions in totally sequential transition paths. This has been accomplished with the inclusion of a very straightforward algorithm. After a valid UTS
assignment is generated this algorithm attempts to speed up transitions by reducing the number of subtransitions in a totally sequential transition path by introducing non-critical races.

An iterative state assignment method can be used as a very powerful algorithm by logic designers. It provides a means of generating multiple UTS assignments, each being valid, with varying number of internal state variables. Availability of multiple assignments could be used to compare their effectiveness under a variety of performance criteria. It could also be used as an adaptive method to develop (i) better algorithms for generating initial assignments and (ii) an insight into the information provided by successful and unsuccessful transitions and how this could be used best to add an internal state variable. The method has a tremendous potential of being used as a very effective computer-aided design tool for asynchronous sequential circuits.
IV. RELATED AREAS OF FUTURE WORK

It is the opinion of the author that the research leading to the
development of iterative state assignment method can be extended to
the following research areas.

A. Routing and Transition Path Generation

The routing problem [16] deals with specifying an interconnection
path between circuit elements on a board. The generally specified
constraints to realize this are either to use minimum wire length or
least number of crossovers. The transition path generation problem
similarly involves interconnection between internal states by generally
using minimum length transition paths but without crossover. If all
transition paths cannot be constructed within these constraints then
another internal state variable is added to the assignment. Similarly,
if all interconnections cannot be computed for routing problems by
satisfying specified constraints, another layer is added.

This indicates that there is considerable similarity between two
problems. The routing problem has received considerable attention
over the years and it is therefore imperative to investigate the relation­
ship between these two problems. Such an investigation may lead to
the application of some of the existing techniques for routing problem
to transition path generation or even development of a common approach
to solve both problems.
B. **Unified State Assignment Techniques**

USTT assignments have to satisfy more stringent constraints than the UTS assignments. This results in a faster circuit realization for the former at a higher cost and a slower circuit realization for the latter at a lower cost. Since Tracey's [1] enunciation of the necessary and sufficient conditions, constraints for USTT assignments are better understood. No such conditions are presently available for UTS assignments and therefore constraints for UTS operation are identified with a modification of those for USTT operation. However, if such conditions are established for UTS assignments, the relationship between the constraints of UTS and USTT assignments can be better understood. Such an understanding may eventually lead to a unified theory of state assignments embracing both the USTT and UTS assignments.
BIBLIOGRAPHY


VITA

Dattatraya Govind Raj-karne was born in Nagpur, India on December 17, 1937 and received his primary and secondary education there. He received Bachelor of Engineering (Honors) degree in Electrical Engineering from Jabalpur University, MP, India in May of 1959.

He underwent graduate apprenticeship at the Heavy Electicals (India) Ltd., Bhopal, MP, India from March 1, 1960 to February 28, 1962 and was then appointed as a Transformer Design Engineer. After two years in this position he was promoted to the rank of Assistant Section Engineer and worked in this capacity until May of 1967 when he came to the United States of America for graduate study.

He joined Tulane University in New Orleans, Louisiana in June of 1967 and received the Master of Science in Electrical Engineering in August of 1969. During the entire period of his graduate work at Tulane he was a Research Assistant in the Department of Electrical Engineering.

He joined the University of Missouri at Rolla in the fall of 1969 for doctoral studies in Electrical Engineering. He was a Graduate Teaching Assistant from September of 1969 to May of 1972 and a Research Assistant from February of 1970 to July of 1972 in the Department of Electrical Engineering. He is presently a visiting Assistant Professor in the Department of Electrical Engineering at the University of Texas at Austin.

The author is a member of IEEE and Eta Kappa Nu.