INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Test sequence generation methods for communication protocols

Yu, Shwuhwa Sarah, Ph.D.
The Ohio State University, 1993
TEST SEQUENCE GENERATION METHODS FOR COMMUNICATION PROTOCOLS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Shwuhwa Sarah Yu, B.S., M.S.

* * * * *

The Ohio State University

1993

Dissertation Committee:
Professor Ming T. Liu
Professor Dik L. Lee
Professor Mukesh Singhal

Approved by

Professor Ming T. Liu
Adviser
Department of Computer
and Information Science
To My Parents and Family
ACKNOWLEDGMENTS

I thank my advisor, Professor Ming T. Liu, for introducing me into the area of Protocol Engineering. Without his guidance and supervision, this dissertation might not exist. Deepest appreciation is given to Professors Dik Lee and Mukesh Singhal for serving as members of the reading committee and reviewing the manuscript.

Special thanks go to my dear husband, Huaiyu Huey Yu, who has always been supportive throughout my graduate study at the Ohio State University. Also, thanks to my parents and all family members for their encouragement and understanding.

I am grateful to my friends Yow-Wei Yao, Wen-Shyen Eric Chen and Chang-Jia Wang for their helpful discussions on this work. Special thanks also to Ing-Miin Hsu, Li-Fen Wu, and Wang-Chien Lee for their moral and technical support. In addition, I thank all the members of the Protocol Engineering research group for their comments on this work.

The research reported in this dissertation has been supported by U.S. Army Research Office under contract No. DAAL03-91-G-0093 and No. DAAL03-92-G-0184.
VITA

December 18, 1960....................................... Born Taiwan, Republic of China.

Sep. 1980 - June 1984................................B.S., Department of Computer
Engineering
National Chaio-Tung University
Hsin-Chu, Taiwan.

Sep. 1984 - June 1986................................M.S., Department of Computer
Engineering
National Chaio-Tung University
Hsin-Chu, Taiwan.

Sep. 1986 - Dec. 1987................................M.S., Department of Computer and
Information Science
Ohio State University
Columbus, Ohio.

Jan. 1987 - June 1992..................................Teaching Associate, Department of
Computer and Information Science
Ohio State University
Columbus, Ohio.

Publication

An Improved Method for Automatic Validation of Communication Protocols. In
Yao and M.T. Liu)

226-231, September 1991. (co-authors: M.T. Liu)

A New Protocol Test Sequence Generation Method Based on UIOS. In Proc. IEEE
INFOCOM '92, pages 2068-2077, May 1992. (co-authors: M.T. Liu)
Utilizing Multiple UIO Sequences and Segment Overlap to Shorten Test Sequences.

Fields of Study

Major Field: Computer and Information Science

Studies in:
  Computer Architecture and Networking: Prof. Ming T. Liu
  Databases: Prof. Dik L. Lee
  Artificial Intelligence: Prof. Terry Patten
# Table of Contents

DEDI CATION ................................................................. ii
ACKNOWLEDGMENTS ........................................ iii
VITA ........................................................................ iv
LIST OF TABLES ..................................................... ix
LIST OF FIGURES ................................................... xii

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Introduction ................................................ 1</td>
<td></td>
</tr>
<tr>
<td>1.1 Motivations ............................................... 1</td>
<td></td>
</tr>
<tr>
<td>1.2 The Problem of Conformance Testing .................. 2</td>
<td></td>
</tr>
<tr>
<td>1.3 Formal Models for Protocol Specification ............ 4</td>
<td></td>
</tr>
<tr>
<td>1.3.1 The Finite State Machine Model .................... 6</td>
<td></td>
</tr>
<tr>
<td>1.3.2 Various Models for Conformance Testing .......... 6</td>
<td></td>
</tr>
<tr>
<td>1.4 Related Works ............................................. 9</td>
<td></td>
</tr>
<tr>
<td>1.5 Major Contributions .................................... 12</td>
<td></td>
</tr>
<tr>
<td>1.5.1 Test Sequence Generation without Length Minimization 12</td>
<td></td>
</tr>
<tr>
<td>1.5.2 Test Sequence Generation with Length Minimization 14</td>
<td></td>
</tr>
<tr>
<td>1.6 Organization of the Dissertation ..................... 15</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td></td>
</tr>
<tr>
<td>Survey of the Literature .................................... 17</td>
<td></td>
</tr>
<tr>
<td>2.1 Review of Test Sequence Generation Methods without Length Minimization 18</td>
<td></td>
</tr>
<tr>
<td>2.1.1 The Transition Tour Method (T method) ............ 19</td>
<td></td>
</tr>
<tr>
<td>2.1.2 The Unique Input/Output (UIO) Method ............. 20</td>
<td></td>
</tr>
<tr>
<td>2.1.3 The Revision of The Unique Input/Output (UIOv) Method 21</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1.4</td>
<td>The Distinguishing Sequence Method (D method)</td>
</tr>
<tr>
<td>2.1.5</td>
<td>The Characteristic Set Method (W method)</td>
</tr>
<tr>
<td>2.1.6</td>
<td>The Partial Characteristic Set Method (Wp method)</td>
</tr>
<tr>
<td>2.2</td>
<td>Review of Test Sequence Generation Methods with Length Minimization</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Aho's Approach</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Shen's Approach</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Chen's Approach</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Miller's Approach</td>
</tr>
<tr>
<td>III</td>
<td>The Unique Input/Output method with Partial Verification (UIO pv)</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic Concepts</td>
</tr>
<tr>
<td>3.2</td>
<td>A New Algorithm for Test Sequence Generation</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparisons and Analyses</td>
</tr>
<tr>
<td>3.4</td>
<td>Conclusions</td>
</tr>
<tr>
<td>IV</td>
<td>The Transition-State Pairs method (TSP)</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic Concept and Definitions</td>
</tr>
<tr>
<td>4.2</td>
<td>Algorithm for Finding USOTSPs</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithm for Transforming Transition-State Pairs into Input/Output</td>
</tr>
<tr>
<td></td>
<td>Sequences</td>
</tr>
<tr>
<td>4.4</td>
<td>Increasing Fault Coverage</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparisons and Analyses</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusions</td>
</tr>
<tr>
<td>V</td>
<td>Minimal Length Test Sequence Generation (MLTG) Method</td>
</tr>
<tr>
<td>5.1</td>
<td>Analysis of Miller's Approach</td>
</tr>
<tr>
<td>5.2</td>
<td>Approach without Verifying the Uniqueness of UIO Sequences</td>
</tr>
<tr>
<td>5.3</td>
<td>Approach with Verifying the Uniqueness of UIO Sequences and Signature Sets</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison with Other Approaches</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusions</td>
</tr>
<tr>
<td>VI</td>
<td>Conclusions and Future Research Directions</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Research</td>
</tr>
</tbody>
</table>

vii
APPENDICES

A  Program for the UIOpv Method ........................................... 126
B  Program for the TSP Method ............................................... 155
C  Program for the MLTG Method .......................................... 186
D  Sample Output of the UIOpv Method ................................. 245
E  Sample Output of the TSP Method ..................................... 247
F  Sample Output of the MLTG Method ................................. 249

BIBLIOGRAPHY ............................................................... 257
# List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
</tr>
<tr>
<td>11</td>
<td>43</td>
</tr>
<tr>
<td>12</td>
<td>46</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>13</td>
<td>The optimal test sequence derived using Miller's approach using both multiple UIOSs and segment overlap</td>
</tr>
<tr>
<td>14</td>
<td>Signature and UIOS of each state generated by the UIO method</td>
</tr>
<tr>
<td>15</td>
<td>Signature set and UIOS of each state generated by the UIOv method and our first method</td>
</tr>
<tr>
<td>16</td>
<td>UIOS of each state generated by the UIO and the UIOv methods</td>
</tr>
<tr>
<td>17</td>
<td>UIOS of each state generated by our improved method</td>
</tr>
<tr>
<td>18</td>
<td>Non-optimized UIO-based test sequence (Uv in the first portion, and transition testing in the second)</td>
</tr>
<tr>
<td>19</td>
<td>Non-optimized UIOv-based test sequence (Uv in the first portion, * Uv in the second, and transition testing in the last)</td>
</tr>
<tr>
<td>20</td>
<td>Non-optimized test sequence generated by our improved method (Uv in the first portion, and transition testing in the last)</td>
</tr>
<tr>
<td>21</td>
<td>UIOS of each state generated by the UIO and the UIOv methods</td>
</tr>
<tr>
<td>22</td>
<td>USIOS of each state generated by the TSP method</td>
</tr>
<tr>
<td>23</td>
<td>Non-optimized UIO-based test sequence (Uv in the first portion and transition testing in the second)</td>
</tr>
<tr>
<td>24</td>
<td>Non-optimized UIOv-based test sequence (Uv in the first portion, * Uv in the second, and transition testing in the last)</td>
</tr>
<tr>
<td>25</td>
<td>Non-optimized TSP-based test sequence (Uv in the first portion and transition testing in the last)</td>
</tr>
<tr>
<td>26</td>
<td>USIOS of each state generated by the TSP method</td>
</tr>
<tr>
<td>27</td>
<td>A summary of comparisons made among five approaches</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conformance testing</td>
</tr>
<tr>
<td>2</td>
<td>Domain of protocol engineering</td>
</tr>
<tr>
<td>3</td>
<td>An FSM specification</td>
</tr>
<tr>
<td>4</td>
<td>Classification of test sequence generation methods</td>
</tr>
<tr>
<td>5</td>
<td>A graph representation of an FSM</td>
</tr>
<tr>
<td>6</td>
<td>The new FSM $G'$ for the rural Chinese postman approach (white arrows represent non-essential edges and black arrows represent essential edges)</td>
</tr>
<tr>
<td>7</td>
<td>The rural symmetric augmentation $G^*$ for the rural Chinese postman approach (values correspond to the number of times each edge is used)</td>
</tr>
<tr>
<td>8</td>
<td>An alternative FSM $G'$ for the approach using multiple UIOSs (white arrows represent non-essential edges and black arrows represent essential edges)</td>
</tr>
<tr>
<td>9</td>
<td>The rural symmetric augmentation $G^*$ for the approach using multiple UIOSs (values correspond to the number of times each edge is used)</td>
</tr>
<tr>
<td>10</td>
<td>(a) An FSM with no UIOS for State C and State D; (b) A faulty FSM with a transfer fault</td>
</tr>
<tr>
<td>11</td>
<td>(a) An FSM specification (b) A faulty FSM with a merged state; (c) A faulty FSM with an extra transition</td>
</tr>
</tbody>
</table>
12 (a) An FSM specification; (b) A faulty FSM with a transfer fault

13 An FSM with no UIOS for States A, B and C

14 An FSM

15 (a) An FSM with converging edges; (b) The FSM after converging edges are removed; (c) Two sets of disjoint paths

16 (a) The test sequence generated using the first set of disjoint paths; (b) The test sequence generated using the second set of disjoint paths

17 (a) An FSM; (b) A faulty FSM

18 (a) An FSM with State C having no UIOS; (b) List of segments generated by the new-Algorithm-4; (c) The test sequence generated by the new Algorithm-4

19 (a) Segments derived for verifying the uniqueness of UIO sequences and signature sets of states; (b) Test sequence generated by the second approach

20 (a) An FSM; (b) A faulty FSM
CHAPTER I

Introduction

1.1 Motivations

New electronic technologies have made computer networking and distributed computing possible. In the last two decades, many computer-communication networks have been designed, implemented and put into service around the world. Many users in academia, business, industry and government use computer networks to help them work more efficiently. In particular, electronic mail has become an important means for people to communicate throughout the world.

Since communications between a pair of end users in a computer network may take on different forms, enabling an orderly exchange of information between physically separated computers is essential. A set of rules governing the interaction between the communicating entities is called a computer-communication protocol, or simply protocol. It prescribes the manner in which communications take place, and the meaning of the information exchanged.

Designing protocols is a challenging task since protocols specify the interaction between communicating entities residing on different nodes of a network, running in parallel, and communicating through possibly unreliable channels. During the trans-
lation from a protocol specification to its implementation, errors may be introduced due to misinterpretation of the specification or other human errors. As a result, after a protocol is implemented, this implementation may not conform to its specification. At present there are no methods that can completely translate or compile a formal specification into implementation code. Therefore, there is a need to test an implementation to determine its conformance with its protocol specification; such a test is commonly called *conformance testing* [41].

1.2 The Problem of Conformance Testing

A test sequence is an input/output sequence which is derived from a protocol specification and applied to a protocol implementation, called an *Implementation Under Test* (IUT). There are two stages in conformance testing (See Figure 1). The first stage is to develop a test sequence and the second is to apply the test sequence to an IUT to determine whether the IUT behaves as expected. Since the internal behavior of the IUT cannot be observed, an IUT is viewed as a black box with an input port and an output port. The second stage of conformance testing can be divided into the following three steps. First, an input sequence is sent to an IUT. Second, the output sequence sent out by the IUT is observed. Third, the output sequence is compared to the one derived from the protocol specification. If the output sequence obtained from the IUT is inconsistent with the one in the test sequence, the IUT must contain errors and is said to be faulty. An IUT conforms to its specification if every specified transition occurs and enters the correct state in the IUT. This approach is called a checking experiment [31, 26]. A test sequence contains a number of subsequences of
Figure 1: Conformance testing
which each either verifies the existence of a transition or the state that a transition enters.

Three different types of faults are checked during conformance testing. The first type are output faults, which cause an IUT to generate an unexpected output. The second are transfer faults, which cause an IUT to enter an incorrect state. The third are merged states, which cause an IUT to enter an incorrect state and to have fewer states than expected.

1.3 Formal Models for Protocol Specification

A Protocol specification describes what the protocol should do and how it should react to external stimuli such as service primitives. The protocol is then implemented based on its specification. Since most protocols are very complex [44, 51], a two-step procedure is commonly used to reduce the complexity of protocol design. A protocol is specified abstractly in the first step of design, and then filled out the implementation details that are incidental to the protocol's behavior in the second step.

The main objective for the abstraction is to facilitate the validation and verification of the protocol before its actual implementation. Since formal methods for protocol specification provide frameworks that one can specify, develop, and verify protocols in a systematic, rather than in an ad hoc manner, they have been applied to protocol design and analysis over the last decade. The formal models have been proved effective for each aspect of Protocol Engineering [35], including protocol service specifications, validation and verification, synthesis, conversion, performance analysis, automatic implementation, and conformance testing (see Figure 2).
Figure 2: Domain of protocol engineering
1.3.1 The Finite State Machine Model

The protocol specification model used in this dissertation is the deterministic finite state machine (FSM) with a finite set of states $S = \{s_1, ..., s_n\}$, a finite set of inputs $I = \{i_1, ..., i_j\}$, and a finite set of outputs $O = \{o_1, ..., o_k\}$. The next state function (NS) is defined as a mapping $NS : S \times I \rightarrow S$, and the output function ($Z$) is defined as a mapping: $Z : S \times I \rightarrow O$.

A finite state machine (FSM) is represented as a directed graph, $G=(V, E)$, where $V$ is a set of nodes, and $E$ is a set of arcs. Each node represents a state and one of the nodes in a graph is identified as the initial state. Each arc, labeled by i/o, represents a state transition in the FSM, where $i$ is an input, and $o$ is an output. Also, a set of nodes in the graph can be specified as the final states if the protocol terminates at them. The head state of a transition is defined as the state that the transition is applied to (i.e. the present state). The tail state of a transition is defined as the state entered after the transition is applied (i.e. the next state). It is also assumed that any state in a directed graph can be reached from the initial state, and the initial state can be reached from any state through a reset input $r$ with a null output. Figure 3 shows an example modeled using an FSM.

1.3.2 Various Models for Conformance Testing

The choice of a formal behavioral model is an important decision for automating test sequence generation. A number of test sequence generation methods have been proposed for protocol specifications based on FSMs [1, 2, 10, 11, 14, 21, 38, 39, 42,
Figure 3: An FSM specification
54, 74, 75, 68, 32, 37, 72, 12, 13, 50, 4, 20, 25, 29, 44, 47, 48, 62] or non-FSM formal description techniques (FDTs) including Extended Finite State Machines (EFSMs) [67, 70, 71], Estelle [44, 46, 27, 19, 61, 60], LOTOS [56, 6] and SDL [30, 7]. The techniques proposed for FSMs can also be applied directly to non-FSM FDTs if certain restrictions are imposed on the protocol specifications.

Among non-FSM FDTs, SDL and Estelle are based on the extended FSM model, and LOTOS is based on behavior trees. [5, 59]. These non-FSM FDTs have larger expressive power than FSMs; therefore, they are capable of specifying complicated protocols. However, when non-FSM FDTs are used to specify protocols, there is the potential loss of controllability and observability of an implementation, which can make testing extremely difficult. Hence, these methods have not been successfully applied to protocol specifications of realistic size and complexity [66]. Many authors use limited versions of the current non-FSM FDTs to show the advances made towards a solution of automatic test sequence generation. What they proposed include heuristic algorithms, formal frameworks, semi-automated algorithms, and non-general solutions.

Since the non-FSM FDTs are very difficult for automatic test sequence generation, this dissertation is concentrated on an FSM model. There are four advantages in choosing FSMs to specify protocols for test sequence generations. First, formal algorithms for protocol specifications using FSMs can be easily obtained and automated. Second, these algorithms can also be applied to other FDTs when certain limitations are imposed. Third, existing length minimization techniques based on FSMs, such as
Rural Chinese Postman Tours, multiple UIOSs, and segment overlap, can be easily incorporated into new methods based on FSMs. Fourth, a number of existing validation techniques such as reachability analysis are available for protocol specifications based on FSMs. The major drawback of the FSM model is its expressive power. In some cases, it is difficult or impossible to model complicated protocols using the FSM model.

1.4 Related Works

Over the past 20 years a number of methods have been proposed in the literature for test sequence generation [36, 62, 3, 53, 73, 24, 14, 42, 10, 21, 2, 2, 11, 38, 54, 74]. These methods are classified into two categories based on their purposes shown in Figure 4. The major distinction for these two categories is that the lengths of test sequences derived by the methods in Category 1 are not minimized while those in Category 2 are minimized. The three most popular methods in Category 1 are the Distinguishing Sequence (D method) method [24], the Characteristic Set (W method) method [14], and the Unique Input/Output (UIO method) method [42]. Those methods focus on deriving unique identities for states and generating segments (or subsequences) for transitions without worrying about how to connect segments to form a short test sequence. A segment is an input/output sequence and is formed by attaching a state's identity to the label of an transition. A state's identity consists of one input/output sequence for states with UIOSs or a set of input/output sequences for states without UIOSs. The UIOv method [10] (a revision of the UIO method) and the Wp method [21] (a revision of the W method) are also included in the first category.
Figure 4: Classification of test sequence generation methods
Two important measurements for test sequence generation methods are applicability and fault coverage. A test sequence generation method's applicability is defined as the number of protocol specifications from which it can derive test sequences. A test sequence generation method's fault coverage is defined as the number of errors that the generated test sequence can detect. The better the applicability of a test sequence generation method is, the more the specifications from which the method can derive test sequences; and the better the fault coverage of a test sequence generation method is, the more the faults that the method can detect. Among the three most popular test sequence generation methods in Category 1, the W method has the widest applicability and the best fault coverage. However, a major drawback of the W method is that it derives too many segments, which lead to a long test sequence. On the contrary, the UIO method can generate a minimal number of segments but has the worst fault coverage and limited applicability. Although the D method has the same fault coverage as the W method, it has very limited applicability.

Four popular test sequence generation methods in Category 2 are Aho's approach [1, 2], Shen's approach [54], Chen's approach [11] and Miller's approach [38]. They all assume that the identities of states are already derived and they focus only on connecting segments together to form a short test sequence. Aho's approach uses the Rural Chinese Postman algorithm to derive test sequences. Further improvement is made in Shen's approach by using multiple UIO sequences. In Chen's approach, segment overlap is considered with a single UIO sequence. In addition, Chen proposes a heuristic to generate short test sequences with the use of multiple UIO sequences.
and segment overlap. Miller claims that, under certain conditions, his approach yields an optimal solution. If these conditions cannot be satisfied, a sub-optimal solution may be obtained.

1.5 Major Contributions

In this dissertation, three methods for test sequence generation are proposed, two for deriving test sequences without length minimization and one with length minimization. These three methods are coded in C, the programs of the first, the second, and the third methods are listed in Appendices A, B, and C, respectively. The input protocols to each of the program can be completely or partial specified, and should be strongly connected and minimal. The output of each program shows the UIOS or signature set of each state. In addition, the output of the third program shows the test sequence and its length. Three outputs are shown in Appendices D, E, F, one for each program using Figure 10 as the sample input protocol specification.

1.5.1 Test Sequence Generation without Length Minimization

The first part of this dissertation deals with deriving a test sequence without length minimization. Two test sequence generation methods are proposed. It is observed that the UIOv method always generates a complete verification part and many segments in a verification part are redundant in detecting transfer errors. To remedy this drawback, the Unique Input/Output method with Partial Verification (UIOpv)
is proposed, which retains the UIOv method's advantages and generates fewer input/output sequences in the verification part.

An efficient algorithm, Algorithm 1, for generating UIOSs and signature sets is proposed in the UIOpv method. In addition, the UIOpv method proves that the upper bound on the length of a UIOS is \((n - 1) \times n\) rather than \(n^n\), where \(n\) is the number of states. Algorithm 1 finds UIOSs and signature sets within this new upper bound; therefore, it can find UIOSs and signature sets faster than the UIOv method.

There are five improvements in Algorithm 1 as compared with the UIO method's algorithm [42]. First, finding UIOSs is faster. For a unique transition-state pair, concatenating the transition and the UIOS of the transition's tail state forms a UIOS if the UIOS of the tail state has already been found. A transition-state pair is defined as a pair consisting of a transition and the tail state of the transition. Second, Algorithm 1 cuts off subtrees that are no longer useful for generating UIOSs to save time. Third, a heuristic to form a smaller signature set is added. Fourth, two other heuristics for selecting input/output sequences are included. Fifth, the new upper bound, \((n - 1) \times n\), is used.

The second method is called the Transition-State Pairs method (TSP). The key concept is that each state can be uniquely identified by the set of its outgoing transition-state pairs. In the TSP method, not only the transition but also the tail state of the transition is considered. Therefore, it is guaranteed that every state can be uniquely identified. On the contrary, without considering a transition's tail state, two states may potentially have the same set of outgoing transitions.
The TSP method has the same fault coverage as the W method, but outperforms the W method in applicability, than those derived by the W method. Therefore, the TSP method retains the advantages of both the W method and the UIO method, but eliminates their disadvantages.

1.5.2 Test Sequence Generation with Length Minimization

The second part of this dissertation deals with deriving test sequences with length minimization for a protocol specification using FSMs.

From the literature it is observed that the most two important factors in generating minimal length test sequences are multiple UIO sequences and segment overlap. If one can efficiently take advantage of both multiple UIO sequences and segment overlap, an optimal test sequence can be derived. Aho’s approach [1, 2] uses the Rural Chinese Postman algorithm to derive test sequences without taking advantage of either multiple UIO sequences or segment overlap. Further improvement is made in Shen’s approach [54] by using multiple UIO sequences. Although Chen’s approach [11] considers both multiple UIO sequences and segment overlap, it has not taken full advantages of multiple UIO sequences. Only a heuristic is proposed to take advantages of both multiple UIO sequences and segment overlap. Miller [38] claims that the shortest test sequence can be derived, but we have found that not only the conditions stated in Miller’s approach can be strengthened but also several corrections and improvements can be applied. We intend to improve Chen’s approach in the utilization of the multiple UIO sequences and Miller’s approach in correctness, applicability and fault coverage. A test sequence generation method with length minimization, called
Minimal Length Test Sequence Generation method (MLTG), is proposed to generate minimal length test sequences based on UIOS. The MLTG method takes advantage of both multiple UIO sequences and segment overlap eliminates the drawbacks of Chen’s and Miller’s approaches.

1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

In Chapter II, a survey of the literature on the existing methods for test sequence generation is provided. Six methods of test sequence generation without length minimization are described, namely, the T method, the UIO method, the UIOv method, the D method, the W method, and the Wp method. Four methods of test sequence generation with length minimization are described, namely, (1) Aho’s approach: test sequence generation based on the Rural Chinese Postman Tour, (2) Shen’s approach: test sequence generation using multiple UIO sequences, (3) Chen’s approach: test sequence generation using segment overlap plus multiple UIO sequences, and (4) Miller’s approach: test sequence generation using multiple UIO sequences and segment overlap. Pros and cons of each method are discussed.

In Chapter III, the UIOpv method for deriving unique identities of states based on UIOS is presented. It is compared with the UIO method, the UIOv method, and the Partial UIO (PUIO) method [15]. The proof of the new upper bound on the length of a UIOS and the proof of the applicability of the UIOv method are also included.
In Chapter IV, the TSP method for deriving unique identities of states is presented. It is compared with the T method, the UIO method, the UIOv method, the D method, the W method, the Wp method, and the UIOpv method.

In Chapter V, the MLTG method using multiple UIOS and segment overlap to shorten test sequences is presented. It is compared with Aho's, Shen's, Chen's, and Miller's methods.

Finally, Chapter VI provides a summary of this dissertation, discusses some important properties of the three methods proposed in the dissertation, and suggests future research directions.

In addition, the three proposed methods are coded using C. Appendices A, B, and C list the programs for the UIOpv, the TSP, and the MLTG methods, respectively, and Appendices D, E, and F list the outputs for running each program using Figure 10 as the input protocol.
CHAPTER II

Survey of the Literature

In this chapter, a brief description is made for each of the ten test sequence generation methods reported in the literature. The first six methods do not use length minimization, but the remaining four do. All ten methods assume that a given protocol specification is modeled using an FSM.

Conformance testing can be done for an IUT by checking for two things. First, every transition specified in an FSM should exist in the IUT. Second, after a transition is traversed, the IUT should end with the correct state based on the specification. Before starting the literature survey, some definitions that will be referred later are given.

Definition 1: A Finite State Machine (FSM) model has a finite set of states \( S = \{s_1, \ldots, s_n\} \), a finite set of inputs \( I = \{i_1, \ldots, i_j\} \), and a finite set of outputs \( O = \{o_1, \ldots, o_k\} \). The Next State (NS) function associated with an FSM is defined as a mapping, \( NS : S \times I \rightarrow S \), and the output function \( (Z) \) is defined as a mapping, \( Z : S \times I \rightarrow O \).

Definition 2: Two FSMs are said to be equivalent if they generate the same output strings for the same input strings.
Definition 3: An FSM $M$ is minimal if the number of states of $M$ is less than or equal to the number of states of any FSM $M'$ equivalent to $M$.

Definition 4: An FSM is completely (or fully) specified if each state of $M$ has a transition for each input symbol. Otherwise, $M$ is said to be incompletely specified.

Definition 5: When the Completeness Assumption is in effect, a state in an FSM reacts to the expected input symbols according to the FSM specification, ignores unexpected input symbols and remains in its current state.

Definition 6: An FSM $M$ is strongly connected if for each state pair $(s_1, s_2)$ there exists a path from state $s_1$ to state $s_2$.

Definition 7: A test sequence for an FSM $M$ is a string of input/output symbols for checking the conformance $M$'s implementation to $M$'s specification.

Definition 8: A test subsequence (or called segment) for an FSM $M$ is a string of input/output symbols for testing either a state or a transition of $M$.

2.1 Review of Test Sequence Generation Methods without Length Minimization

The six test sequence generation methods without length minimization, namely, the Transition Tour, the Unique Input/Output, the Revision of the Unique Input/Output, the Distinguish Sequence, the Characteristic Set, and the Partial Characteristic Set methods, are described in the following six sections.
Table 1: A Transition Tour Sequence


Cost = 15

2.1.1 The Transition Tour Method (T method)

The T method [39] is simple. It derives a test sequence (i.e. a transition-tour sequence) by randomly generating inputs and applying the test sequence to an FSM until each transition in the FSM has been traversed at least once. A test sequence derived by the T method may contain redundant inputs that are generated by loops in the transition tour. A reduction procedure is then used to remove the redundant inputs. A transition tour sequence for the FSM in Figure 3 is shown in Table 1.

One advantage of the T method is its simplicity. In addition, the T method can find a test sequence for any FSMs that are minimal, strongly connected and completely specified. However, its drawback is that the test sequence derived only checks for the existence of transitions, but not for the correctness of the tail states that the transitions enter. Therefore, some faulty FSMs may not be detected by the test sequences derived by the T method.
2.1.2 The Unique Input/Output (UIO) Method

In the UIO method [42], a unique input/output sequence (UIOS) is used as the identity of a state in an FSM. If a state has no UIOS or has a UIOS with length of more than $2n^2$, where $n$ is the number of states in an FSM, a signature is used instead. A signature for a state $s_i$ is defined as follows:

$$LD(s_i) = \left\{ \begin{array}{ll}
IO(i, 1) \otimes \sum_{k=2}^{n} (T_i(k-1) \otimes IO(i, k)) & \text{where } k \neq i \text{ for } i \neq 1 \\
IO(1, 2) \otimes \sum_{k=3}^{n} (T_i(k-1) \otimes IO(1, k)) & \text{for } i = 1
\end{array} \right.$$  

where the symbol $\otimes$ represents concatenation, $IO(i, k)$ represents an input/output sequence that distinguishes $s_i$ from $s_k$, and $T_i(k)$ represents a shortest input/output sequence that brings the tail state of the application of $IO(i, k)$ back to $s_i$. A state's UIOS is unique but its signature is not. Thus a UIOS can be used to distinguish one state from all other states but a signature cannot.

A test sequence generated by the UIO method consists of two parts. The first part, denoted by $Uv$, checks for the existence of every state in the IUT using UIOSs. The second part, denoted by $Tr$, consists of many segments, which check for the existence of transitions. After the IUT has been brought to a particular head state of a transition by applying an appropriate input sequence, and after the input labeled on the transition is applied, a segment first verifies if the IUT ends with the correct tail state, and then it resets the IUT to the initial state for the next transition. The UIOS of each state for the FSM in Figure 3 is shown in Table 2.
The UIO method can only derive test sequences for FSMs that are minimal and strongly connected. It has two major advantages. First, it does not require a completely specified protocol specification. Second, a UIOS consists of only one sequence, which can uniquely identify a state; therefore, a short test sequence is derived. In addition, the test sequence derived can check for both the existence of transitions and the correctness of the tail states that the transitions enter. However, the UIO method's drawback is that the generated signatures are not unique. As a result, many faulty FSMs will not be detected by the test sequences derived by the UIO method.

2.1.3 The Revision of The Unique Input/Output (UIOv) Method

An important observation is made in [10]; that is, although UIOSs generated by the UIO method are unique to a protocol specification, they may not be unique to the IUT of the protocol. Since UIOSs may not be unique to an IUT, and the signatures derived by the UIO method are not unique, some faulty IUTs could not be detected by the test sequence generated by the UIO method. Therefore, the UIOv method was proposed, which is a revision of the UIO method.

There are two differences between the UIOv and the UIO methods. First, the UIOv method proposes a new way to form a signature. After deriving a set of input/output sequences (IOs) for a state without a UIOS, the UIO method concatenates the set of IOs to form a signature, but the UIOv method uses the set of IOs as a signature set. Although the signature generated by the UIO method is not unique, the signature set generated by the UIOv method is unique. As a result, some faulty
<table>
<thead>
<tr>
<th>state</th>
<th>UIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B/2</td>
</tr>
<tr>
<td>1</td>
<td>A/1.A/1</td>
</tr>
<tr>
<td>2</td>
<td>B/0</td>
</tr>
<tr>
<td>3</td>
<td>B/1.B/1</td>
</tr>
<tr>
<td>4</td>
<td>A/1.A/0</td>
</tr>
</tbody>
</table>
IUT missed by the UIO method can be detected by the UIOv method. Second, the UIOv method introduces a verification part, denoted as $U_v$, to check for the uniqueness of the UIOSs and the signature sets. Verifying the uniqueness of a state's UIOS or signature set is done by applying the state's UIOS or signature set to all other states and then checks the output against the expected one. Thus, a verification part ensures that the UIOS or the signature set is not only unique in the specification, but also unique in the IUT.

Since a signature set in the UIOv method set is a set of IOs rather than a long concatenated sequence, each element, which is an IO, in the signature set is verified one by one as though it were a UIOS. The test sequence generated by the UIOv method consists of three parts. In addition to the $U_v$ and the $T_r$ parts, the UIOv method has a verification part, denoted as $U_v$.

One advantage of the UIOv method is that it can find a test sequence for any FSMs that are minimal, strongly connected and completely specified. In addition, the derived test sequence can check for both the existence of transitions and the correctness of the tail states that transitions enter. However, the drawback of the UIOv method is that more segments are derived than the UIO method.

2.1.4 The Distinguishing Sequence Method (D method)

In the D method [24], an input sequence is called a distinguishing sequence (DS) if, when it is applied to each state, a different output sequence will be produced by each state. The DS is used as the identity of every state in an FSM. The key objective of the D method is to compute a DS (if it exists) for a given FSM. A DS can be found by
constructing a “distinguishing tree” inductively on tree levels [31]. By using Figure 3 as an example, the input sequence BB is the DS derived by the D method and the output sequences obtained by applying BB to each state, are listed in Table 3.

As with the UIOv method, the D method can only derive test sequences for FSMs that are minimal, strongly connected and completely specified. The advantage of the D method is that a single sequence can uniquely identify all states. In addition, the derived test sequence can check for both the existence of transitions and the correctness of the tail states that transitions enter. However, the major drawback is that not every FSM possesses a DS. As a result, test sequence cannot be derived for those that do not have a DS. Another drawback of the D method is that the length of a DS usually is longer than the length of a UIOS.

2.1.5 The Characteristic Set Method (W method)

In the W method [14], a set of input sequences is called a characteristic set (W) if, when the set is applied to each state, each state outputs a different set of sequences. The W set is used to distinguish every state in an FSM. The key objective of the W method is to compute a W set, which can be found by constructing a “P₄ table” inductively on the number of states for a given FSM [23]. For example, {A, B, AA} is the W set for the FSM in Figure 3, and Table 4 shows the last output symbols obtained by applying the W set to each state.

One advantage of the W method is that it can find a test sequence for any FSMs that are minimal, strongly connected and completely specified. In addition, the derived test sequence can check for both the existence of transitions and the correctness
Table 3: Outputs on DS

<table>
<thead>
<tr>
<th>state</th>
<th>DS: BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>
Table 4: Last output symbol on W

<table>
<thead>
<tr>
<th>state</th>
<th>A</th>
<th>B</th>
<th>AA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

of the tail states that transitions enter. However, the drawback of the W method is that more segments are derived than the UIO method.

2.1.6 The Partial Characteristic Set Method (Wp method)

The Wp method [21] is a revision of the UIO method. The only difference between them is that the Wp method derives a partial rather than a whole W set for each state. The partial W set of each state in Figure 3 is shown in Table 5. As compared with the W method, the Wp method has the same applicability, fault coverage, advantages,
and disadvantage, but it derives fewer segments. The price paid by the Wp method is that more computation time is needed to derive a partial W set for each state.

2.2 Review of Test Sequence Generation Methods with Length Minimization

All of the four test sequence generation methods with length minimization, namely, Aho's [1, 2], Shen's [54], Chen's [11] and Miller's approaches [38], are based on the UIO method. A four-step procedure used by most existing methods to find minimal length test sequences is described below.

1. Compute UIO sequences for each state.

2. Construct a segment for each transition. A segment is formed by appending a UIO sequence of the transition's tail state to the label of the transition.

3. Remove those segments that are completely contained in other segments.

4. Generate a minimal length test sequence by connecting segments in an optimal way.

Different approaches propose various ways to connect segments. An inefficient way to form a test sequence from a set of segments is to find a path for every pair of adjacent two segments; each path links the tail state of the first segment to the head state of the second segment. This inefficient way is improved in Aho's approach [1, 2]. The least cost path between segments can be found by solving the rural Chinese postman tour problem. Shen's approach [54] improves Aho's by, using multiple
<table>
<thead>
<tr>
<th>State</th>
<th>Partial W Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{B}</td>
</tr>
<tr>
<td>1</td>
<td>{A, AA}</td>
</tr>
<tr>
<td>2</td>
<td>{B}</td>
</tr>
<tr>
<td>3</td>
<td>{B, AA}</td>
</tr>
<tr>
<td>4</td>
<td>{B, AA}</td>
</tr>
</tbody>
</table>
UIO sequences to shorten test sequence. Chen's approach [11] improves Aho's by using segment overlap. In addition, Chen proposes a heuristic to benefit from segment overlap and multiple UIO sequences. Miller's approach uses both multiple UIO sequences and segment overlap so his approach yields an optimal solution under certain conditions. Each of the four test sequence generation methods is described in the following sections.

2.2.1 Aho's Approach

Aho proposes an optimization technique for protocol test sequence generation based on UIO sequence and rural Chinese postman tour [1, 2]. A postman tour of a given FSM is a tour that contains every edge in the FSM at least once. The Chinese postman tour (so called because the Chinese mathematician Kuan first studied this problem [33]) is a minimum-cost postman tour. A rural postman tour is a tour such that each edge in a specified subset of the edges of a given FSM is traversed at least once. The rural Chinese postman tour is a minimum-cost rural postman tour. Computing the rural Chinese postman tour is known to be NP-complete [34, 17]. However, most protocol specifications modeled as FSMs possess a certain structure that can be exploited to yield a polynomial-time algorithm for rural Chinese postman tour [1, 2].

In order to derive a short test sequence, each edge should be traversed as few times as possible and each essential edge must be traversed at least once. The major work of deriving an optimized test sequence is to determine (1) which edges should be replicated and (2) how many replicates are needed. In [1, 2], Aho observes that if
an Euler tour exists in an FSM, the Euler tour is the optimized test sequence. Aho also observes that each symmetric FSM has an Euler tour and each non-symmetric FSM can be augmented to become symmetric by replicating some edges a few times.

Aho’s approach can be divided into the following four steps. An example based on Figure 5 is used to illustrate each step.

**Step 1:** Use the UIO method to derive a UIOS for each state. The UIOS of each state in Figure 5 is shown in Table 6.

**Step 2:** Use the UIO method to derive segments for testing the existence of each transition and each state. Use an edge to represent a segment and add those edges into the original FSM $G$. The new FSM $G'$ contains the added edges (called essential edges) and the original ones (called non-essential edges). The new FSM $G'$ is shown in Figure 6.

**Step 3:** Augment a non-symmetric FSM $G' = (V', E')$ to a rural symmetric one $G^* = (V^*, E^*)$ as follows. Let $V^* = V'$. Each non-essential edge in $E'$ is included in $E^*$ zero or more times and each essential edge in $E'$ is included in $E^*$ at least once so that the total cost of edges in $E^*$ is minimized. Figure 7 shows the rural symmetric augmentation $G^*$ of the FSM in Figure 6. Each number by the edge represents the number of replicates.

**Step 4:** Solve the rural Chinese postman tour problem for the rural symmetric FSM obtained in step 3. The optimal test sequence for the FSM in Figure 5 is shown in Table 7.
Figure 5: A graph representation of an FSM
Table 6: The UIOS for each state

<table>
<thead>
<tr>
<th>STATE</th>
<th>UIO SEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>b/x.a/x</td>
</tr>
<tr>
<td>V2</td>
<td>b/y</td>
</tr>
<tr>
<td>V3</td>
<td>b/x.c/z</td>
</tr>
<tr>
<td>V4</td>
<td>b/x.c/y</td>
</tr>
<tr>
<td>V5</td>
<td>c/z</td>
</tr>
</tbody>
</table>
Figure 6: The new FSM $G'$ for the rural Chinese postman approach (white arrows represent non-essential edges and black arrows represent essential edges)
Figure 7: The rural symmetric augmentation $G^*$ for the rural Chinese postman approach (values correspond to the number of times each edge is used)
Table 7: The optimal test sequence for Aho's approach using the rural Chinese postman tour

<table>
<thead>
<tr>
<th></th>
<th>b/x.b/y</th>
<th>c/y.c/z</th>
<th>a/x.b/x.a/x</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/null</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a/z.b/x.c/y</td>
<td>c/z.b/x.a/x</td>
<td>r/null.b/x.a/x</td>
<td>a/z</td>
</tr>
<tr>
<td>a/x.c/z</td>
<td>c/y.b/x.c/y</td>
<td>a/z</td>
<td>b/x</td>
</tr>
<tr>
<td>b/x.c/z</td>
<td>b/x</td>
<td>b/y.b/x.c/z</td>
<td>b/x</td>
</tr>
<tr>
<td>a/x.c/z</td>
<td>b/x</td>
<td>r/null.b/x.a/x</td>
<td>a/z</td>
</tr>
<tr>
<td>b/x</td>
<td>r/null.b/x.a/x</td>
<td>a/z</td>
<td>b/x.b/x.c/z</td>
</tr>
<tr>
<td>r/null.b/x.a/x</td>
<td>a/z</td>
<td>r/null.b/x.a/x</td>
<td>c/z</td>
</tr>
</tbody>
</table>

TOTAL = 55 input/output pairs
2.2.2 Shen’s Approach

In [54], Shen presents new results for automatically generating test sequences for communication protocols by means of UIO Sequences and the rural Chinese postman algorithm [1, 2]. Shen shows that if multiple minimum-length UIO sequences are used, test sequences shorter than those derived by Aho’s approach can be derived. Shen’s approach selects one UIOS from the set of minimum-length UIOSs for each state using an algorithm for optimal assignment. The goal of the optimal assignment is to reduce the number of replicates of edges needed to augment a non-symmetric FSM to a rural symmetric one.

Shen’s approach results in a substantial reduction (4%-37%) in the length of a test sequence with no noticeable increase in the time required to generate the test sequence. Similar to Aho’s approach, Shen solves rural Chinese postman tour problem to obtain an optimal length test sequence. However, unlike Aho’s approach, which uses a fixed UIOS for each state, Shen considers multiple UIOSs, so the most suitable UIOS can be selected whenever necessary.

Aho’s approach generates only a unique rural symmetric FSM since exactly one UIO sequence per state is computed. As a result, the minimal number of replicates is fixed. A key observation made in Shen’s approach is that there may exist several minimum-length UIO sequences for a given state. For example, for the FSM in Figure 5 there are four UIO sequences for V1, one for V2, two for V3, one for V4 and two for V5 shown in Table 8. A judicious choice of UIO sequences for each transition could reduce the number of replicates, thereby shortening the overall test sequence.
For example, replacing \{b/x.a/x\} by \{c/y.b/x\} for the transitions going into state \(V1\) reduces the number of essential edges going into \(V5\) and increases the number of essential edges going into \(V3\). As a result, the number of replicates for rural symmetric augmentation is reduced as is the length of the resulting test sequence.

Shen's approach, similar to Aho's approach, uses the four steps given in Section 2.2. However, in Step 1, instead of deriving one UIO sequence per state, multiple UIO sequences are derived shown in Table 8. Using multiple UIOSs, a new FSM \(G'\) is generated in Step 2 and shown in Figure 8, and a rural symmetric FSM is generated in Step 3 and shown in Figure 9. In Step 4 of Shen's approach, a resulting rural Chinese postman tour is generated and shown in Table 9.

### 2.2.3 Chen's Approach

There are two important factors in minimizing the connection cost of segments: multiple UIO sequences and segment overlap. A segment, \(S_i\), is said to overlap with another segment, \(S_j\), if the final portion of \(S_i\) is identical to the initial portion of \(S_j\). The length of an overlap is the number of edges in the identical portion. For example, Table 10 shows the set of segments and Table 11 shows the overlap between segments derived from the FSM in Figure 5.

In [11], Chen considers the overlap between segments of transitions. Two procedures are presented in Chen's approach to exploit the overlap property. The first procedure incorporates the consideration of segment overlap into the rural Chinese postman problem. In the second procedure, segment overlap as well as multiple UIO sequences are also taken into consideration, but only a heuristic is proposed. Although
Table 8: The complete set of minimum-length UIO sequences

<table>
<thead>
<tr>
<th>STATE</th>
<th>UIO SEQUENCE</th>
<th>TAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>b/x.a/x</td>
<td>V5</td>
</tr>
<tr>
<td></td>
<td>a/x.b/x</td>
<td>V2</td>
</tr>
<tr>
<td></td>
<td>c/y.b/x</td>
<td>V3</td>
</tr>
<tr>
<td></td>
<td>a/x.c/y</td>
<td>V4</td>
</tr>
<tr>
<td>V2</td>
<td>b/y</td>
<td>V3</td>
</tr>
<tr>
<td>V3</td>
<td>b/x.c/z</td>
<td>V1</td>
</tr>
<tr>
<td></td>
<td>b/x.a/z</td>
<td>V4</td>
</tr>
<tr>
<td>V4</td>
<td>b/x.c/y</td>
<td>V5</td>
</tr>
<tr>
<td>V5</td>
<td>c/z</td>
<td>V1</td>
</tr>
<tr>
<td></td>
<td>a/z</td>
<td>V4</td>
</tr>
</tbody>
</table>
Figure 8: An alternative FSM $G'$ for the approach using multiple UIOSs (white arrows represent non-essential edges and black arrows represent essential edges)
Figure 9: The rural symmetric augmentation $G^*$ for the approach using multiple UXOSs (values correspond to the number of times each edge is used)
Table 9: A lower-cost conformance test sequence derived by Shen’s approach using multiple UIOSs

<table>
<thead>
<tr>
<th>r/null</th>
<th>a/x.a/x.b/x</th>
<th>r/null.a/x.b/x</th>
<th>b/y.b/x.c/z</th>
</tr>
</thead>
<tbody>
<tr>
<td>b/x.b/y</td>
<td>r/null.c/y.b/x</td>
<td>b/x.c/z</td>
<td>r/null.c/y.b/x</td>
</tr>
<tr>
<td>c/y.c/z</td>
<td>c/y.b/x.c/y</td>
<td>a/z.b/x.c/y</td>
<td>c/z.b/x.a/x</td>
</tr>
<tr>
<td>r/null.b/x.a/z</td>
<td>a/z</td>
<td>a/x.a/z</td>
<td>b/x.b/x.a/z</td>
</tr>
<tr>
<td>r/null.a/x.b/x</td>
<td>a/x.c/z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cost = 45
Table 10: Set of segments

<table>
<thead>
<tr>
<th>Edge</th>
<th>Segment #</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V2, a/x, V5)</td>
<td>1</td>
<td>(V2, a/x, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V2, b/y, V3)</td>
<td>2</td>
<td>(V2, b/y, V3) (V3, b/x, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V3, b/x, V5)</td>
<td></td>
<td>(V3, b/x, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V3, c/y, V5)</td>
<td>3</td>
<td>(V3, c/y, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V1, c/y, V4)</td>
<td>4</td>
<td>(V1, c/y, V4) (V4, b/x, V3) (V3, c/y, V5)</td>
</tr>
<tr>
<td>(V1, a/x, V1)</td>
<td>5</td>
<td>(V1, a/x, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V1, r/null, V1)</td>
<td>6</td>
<td>(V1, r/null, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V5, a/z, V4)</td>
<td>7</td>
<td>(V5, a/z, V4) (V4, b/x, V3) (V3, c/y, V5)</td>
</tr>
<tr>
<td>(V5, r/null, V1)</td>
<td>8</td>
<td>(V5, r/null, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V2, r/null, V1)</td>
<td>9</td>
<td>(V2, r/null, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V5, c/z, V1)</td>
<td>10</td>
<td>(V5, c/z, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V3, r/null, V1)</td>
<td>11</td>
<td>(V3, r/null, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V4, r/null, V1)</td>
<td>12</td>
<td>(V4, r/null, V1) (V1, b/x, V2) (V2, a/x, V5)</td>
</tr>
<tr>
<td>(V4, a/x, V5)</td>
<td>13</td>
<td>(V4, a/x, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V4, b/x, V3)</td>
<td>14</td>
<td>(V4, b/x, V3) (V3, b/x, V5) (V5, c/z, V1)</td>
</tr>
<tr>
<td>(V1, b/x, V2)</td>
<td>15</td>
<td>(V1, b/x, V2) (V2, b/y, V3)</td>
</tr>
</tbody>
</table>
Table 11: Overlap between segments

<table>
<thead>
<tr>
<th>From Segment #</th>
<th>To Segment #</th>
<th>Overlap Portion</th>
<th>Length of Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>(V5, c/z, V1)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>(V5, c/z, V1)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>(V5, c/z, V1)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>(V3, c/y, V5)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>(V3, c/y, V5)</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>(V2, a/x, V5)</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>(V5, c/z, V1)</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
<td>(V5, c/z, V1)</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>(V2, b/y, V3)</td>
<td>1</td>
</tr>
</tbody>
</table>
the use of this heuristic does not guarantee that the shortest test sequence will be found, Chen shows that considering segment overlap and multiple UIO sequences can shorten test sequences substantially.

The major contribution in Chen's approach is to extend a given FSM not only with segment edges (or essential edges) as in [54] but also with overlap links. Overlap links are used to represent the overlap relationship between segments. In addition to a segment edge, two segment nodes are added to represent the head and tail states of the segment. Each segment node is then connected to its corresponding node via zero length directed virtual links. A virtual link is drawn to the head segment node from the corresponding real node, and another is drawn from the tail segment node to its corresponding real node. If segment $S_i$ overlaps with segment $S_j$, an overlap link is drawn from the tail state of segment $S_i$ to the head state of segment $S_j$. The overlap link has a negative length equal to the length of the overlap between segment $S_i$ and segment $S_j$. The negative lengths denote the number of transitions saved when overlapped segments are considered.

The overlap links enforce the constraint that a tour must have segment $S_i$ immediately followed by segment $S_j$. With the addition of segment nodes, segment links are no longer weakly connected, and the optimal solution may not be obtainable using the technique discussed in Section 2.2.2. The formulated rural Chinese postman tour problem can then be solved as a minimum cost maximum cardinality matching problem in a bipartite graph. If the solution is a tour, then the shortest test sequence is found. Otherwise, only a near optimal test sequence can be obtained via heuristics.
The optimal test sequence derived using Chen's approach for the FSM in Figure 5 is shown in Table 12.

2.2.4 Miller's Approach

Miller's approach proposed two algorithms which use both segment overlap and multiple UIO sequences implicitly for test sequence generation. The condition for deciding which algorithm to use is as follows.

\[
\begin{align*}
\text{If the FSM is definitely diagnosable then} & \\
\text{perform Algorithm-2} & \\
\text{else} & \\
\text{perform Algorithm-3} & \\
\text{endif}
\end{align*}
\]

An edge is a converging edge if at least one other edge has the same label and enters the same tail state as the edge does. Miller defines that an FSM is definitely diagnosable if there is no converging edge. Disjoint paths are defined as paths having no intersection. Algorithm-2 and Algorithm-3 in Miller's approach are defined as follows.

**Miller's Algorithm-2**

If an FSM \( G \) has an Euler Tour then

\[
\text{Test Seq.} = \text{Any Euler Tour of the FSM beginning with any state } u_i \in UIO_i
\]

else if \( G \) has an Euler Path starting and ending
Table 12: The optimal test sequence derived using Chen's approach using segment overlap

| Seg 15 -> Seg 2 -> Seg 10 -> Seg 1 -> Seg 4 -> |
| Seg 3 -> Seg 5 -> Seg 7 -> Seg 8 -> a/z -> |
| Seg 13 -> Seg 6 -> a/z -> Seg 14 -> b/x -> |
| Seg 9 -> a/z -> Seg 12 -> a/z. b/x -> Seg 11 |

(a) Execution Sequence of Segments

| b/x. b/y. b/x. c/z. b/x. a/x. c/z. c/y. b/x. c/y. c/z. a/x. |
| b/x. a/x. a/x. b/x. c/y. r/null. b/x. a/x. a/z. a/x. a/x. |
| c/z. r/null. b/x. a/x. a/z. b/x. b/x. c/z. b/x. r/null. |
| b/x. a/x. a/z. r/null. b/x. a/x. a/z. b/x. r/null. b/x. a/x. |

Cost = 43

(b) Execution Sequence of Input/Output Sequences
at state $v_k$ and state $v_i$, respectively, then

Test Seq. = Any Euler Path of the FSM
from state $v_k$ to state $v_i \text{ UIO}_i$

else

(1) augment the FSM minimally to get a new graph $G'$ that has an Euler Path
(2) Test Seq. = Any Euler Path of $G' \text{ UIO}_i$
where state $v_i$ is the tail state of the Euler Path

endif

Although the test sequence generated by Algorithm-2 may not begin with an initial state, one can easily start the test sequence from the initial state by modifying Algorithm-2 to choose the Euler Tour and Euler Path beginning with the initial state.

Miller's Algorithm-3

1. Compute $\text{UIO}_{i,j} \forall v_i, v_j \in V$ where $\text{UIO}_{i,j} = \text{minimum length UIO sequence from } v_i \text{ to } v_j$.

2. Construct a new graph $G' = (V', E')$ from $G = (V, E)$ such that $E' = E - E_c$ where $E_c = CE_1, ..., CE_n$ and $CE_i$ is a converging edge.

3. Compute disjoint paths $P_1, ..., P_n$ from $G'$.

4. Construct a bipartite graph $G_b = (V_b, E_b)$ where $V_b = V_s \cup V_e$
\[ V_t = \{ i : i = Head(Seg_i) \text{ for some } j \} \]
\[ V_e = \{ i : i = Tail(Seg_i) \text{ for some } j \} \]
\[ E_h = \{ (i,j) : i \in V_t, j \in V_e; i \neq Tail(Seg_k), j \neq Head(Seg_h) \text{ for some } k \} \]

Assign costs \( C_{i,j} \) to each \( (i,j) \in E_h \) based on Miller's cost function described below.

\( Seg_i \) is a segment. A segment is either a converging edge with two end states or a disjoint path with two end states. The two end states are the state starting the converging edge or the disjoint path, and the state at which the converging edge or the disjoint path ends. \( Head(Seg_i) \) is the state starting the segment, and \( Tail(Seg_i) \) is the state at which the segment ends.

5. Use the maximum cardinality minimum cost matching algorithm for graph \( G_h \).

Every converging edge and disjoint path is considered as a segment. The solution to the above matching problem is to concatenate given segments to form a sequence using transfer paths to connect those segments. A transfer path between States A and B is a path starting at State A and ending at State B. If an Euler Path (Tour) can be found then a valid test sequence is derived by appending the UIO sequence of the last state in the Euler Path (Tour) to the end of the Euler Path (Tour). Otherwise, these disconnected components are connected by some heuristics similar to Chen's approach [11]. The cost function \( C_{i,j} \) for an edge from State i to State j is defined as follows.
Miller's Cost Function:

1. \( i = \text{Tail}(CE_k) \) and \( j = \text{Head}(P_l) \) for some \( k \) and \( l \). \( C_{i,j} = \text{length of the shortest UIO sequence of } i \text{ ending with } j \) for \( i \neq j \) and \( C_{i,i} = 0 \) \( \forall i \). \( CE_k \) is a converging edge and \( P_l \) is a disjoint path.

2. \( i = \text{Tail}(CE_k) \) and \( j = \text{Head}(CE_i) \) for some \( k \) and \( l \). \( C_{i,j} = \min \{ \text{length of the shortest UIO sequence of } i \text{ to } j, \text{length of the shortest UIO sequence of } i \text{ to } \text{Tail}(CE_i) \text{ ending with the converging edge } CE_i - 1 \} \).

3. \( i = \text{Tail}(P_k) \) and \( j = \text{Head}(CE_i) \) for some \( k \) and \( l \). \( C_{i,j} = \min \{ \text{length of the shortest UIO sequence of } i \text{ to } j, \text{length of the shortest UIO sequence of } i \text{ to } \text{Tail}(CE_i) \text{ ending with the converging edge } CE_i - 1 \} \).

4. \( i = \text{Tail}(P_k) \) and \( j = \text{Head}(P_l) \) for some \( k \) and \( l \). \( C_{i,j} = \min \{ \text{length of the shortest UIO sequence of } i \text{ ending with } j, \min \{ \text{length of the shortest UIO sequence of } i \text{ ending with } k \text{ along } P_l - \text{number of transitions from } j \text{ to } k \text{ where } k \text{ is a state on } P_l \} \} \).

As shown above, there are four possible ways to construct Cost \( C_{i,j} \). First, a shortest UIO sequence between State A to State B is found when State A is the tail state of a converging edge and State B is the head state of a disjoint path and State A differs from State B. If State A is the same as State B, the length of the shortest UIO sequence is zero. Second, a shortest UIO sequence between State A to State B is found when State A is the tail state of a converging edge and State B is the head state of another converging edge. Third, a shortest UIO sequence between State A
Table 13: The optimal test sequence derived using Miller's approach using both multiple UIOSs and segment overlap

| b/x. b/y. b/x. c/z. a/x. c/y. b/x. c/y. a/z. a/x. |
| c/z. b/x. a/x. a/z |

Cost = 14

to State B is found when State A is the tail state of a disjoint path and State B is the head state of a converging edge. Fourth, a shortest UIO sequence between State A to State B is found when State A is the tail state of a disjoint path and State B is the head state of another disjoint path.

Miller's approach generates shorter test sequences than Aho's, Shen', and Chen's approaches by using multiple UIO and segment overlap techniques. Table 13 shows the optimal test sequence derived using Miller's approach for the FSM in Figure 5.
CHAPTER III

The Unique Input/Output method with Partial Verification (UIOpv)

The UIO method is a very popular test sequence generation method. Its advantage is that it derives short test sequences, but its disadvantage is that it generates non-unique signatures for states with no UIOS. Faulty IUTs may not be detected due to the non-unique signature. The UIOv method improves the UIO method in two ways. First, the UIOv method generates unique signature sets for states with no UIOS. Second, it uses a verification part to ensure the uniqueness of UIOSs and signature sets. Therefore, some fault IUTs that are not detected by the UIO method may be detected by the UIOv method. In this chapter, the Unique Input/Output method with Partial Verification (UIOpv) [74] is presented. It retains the UIOv method's advantages but requires only a partial verification part. The UIOpv method is coded using C and the detailed program is listed in Appendix A.

The UIOpv method has a number of improvements over the UIOv method. First, the incorrect claim made in the UIOv method is proved to be false. Second, the UIOpv method generates a partial verification part. Third, the UIOpv method uses less time to generate a test sequence. The detail of each improvement is described in the following sections.
3.1 Basic Concepts

Chan claims that when a generated signature set is not unique, the UIOv method cannot handle this case. This claim is proved to be incorrect by proving that the UIOv method can always generate a test sequence for any minimal, completely specified and strongly connected FSM.

Theorem 1:
If a given FSM is minimal, completely specified and strongly connected, the UIOv method can derive a non-unique signature set for every state in the FSM.

Proof: Assume that there are n states, and States A and B have the same signature set. Based on the UIOv method's algorithm, there are (n-1) IO sequences in State A's signature set, and one of them is IO(A,B). IO(A,B) represents an input/output sequence that can distinguish State A from State B, In other words, there exists a path beginning from State with transitions labeled as IO(A,B), but no such path exists for State B. There are two cases to be considered. First, if such a sequence is not in State B's signature set, it violates the assumption that State B has the same signature set as State A. Second, if such a sequence is in State B's signature set, it means the same sequence is used by State B to distinguish another state. Then, it violates the definition of IO(A,B). By contradiction, it is proved that no two states can have the same signature set. □

As an example, Tables 14 through 15 show the different identities for State C and State D in Figure 10(a) derived by the UIO method, the UIOv method and the UIOAv method. The signatures of State C and State D derived by the UIO
Table 14: Signature and UIOS of each state generated by the UIO method

<table>
<thead>
<tr>
<th>State Name</th>
<th>Signature or UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>State A</td>
<td>0/1</td>
</tr>
<tr>
<td>State B</td>
<td>1/1</td>
</tr>
<tr>
<td>State C</td>
<td>2/1.1/0.1/1.1/0 (IO(C,A)T(A,C)IO(C,B))</td>
</tr>
<tr>
<td>State D</td>
<td>2/1.0/1.2/2.1/0 (IO(D,A)T(A,D)IO(D,B))</td>
</tr>
<tr>
<td>State X</td>
<td>3/2</td>
</tr>
</tbody>
</table>

Table 15: Signature set and UIOS of each state generated by the UIOv method and our first method

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS or Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>State A</td>
<td>0/1</td>
</tr>
<tr>
<td>State B</td>
<td>1/1</td>
</tr>
<tr>
<td>State C</td>
<td>{2/1.0/1, 1/0.1/1}</td>
</tr>
<tr>
<td>State D</td>
<td>{2/1.2/1, 1/0.1/0}</td>
</tr>
<tr>
<td>State X</td>
<td>3/2</td>
</tr>
</tbody>
</table>

division method are not unique; therefore, the test sequence derived by the UIO method cannot detect the faulty FSM shown in Figure 10(b). However, the UIOv method and the UIOv method can derive a test sequence that can distinguish the difference between Figures 10(a) and 10(b). Appendix D lists the sample output generated by the program implementing the UIOv method using Figure 10(a) as the input protocol specification.

The purpose of a verification part in the UIOv method is to guarantee that each UIOS and signature set derived from a given specification FSM is not only unique
Figure 10: (a) An FSM with no UIOS for State C and State D; (b) A faulty FSM with a transfer fault.
in the specification but also unique in the protocol implementation. Three situations can cause a UIOS and a unique signature set to become non-unique in a protocol implementation: (1) merged states, (2) extra transitions and (3) transfer faults. Figures 11 and 12 illustrate each of the situations. Figure 11(a) is a FSM specification with three states. Figure 11(b) illustrates State 2 and State 3 merged into one. Figure 11(c) shows State 2 with an extra transition i/j, which ends with State 1. A transfer fault is illustrated in Figure 12(b); the tail state of transition b/1 leading from State 1 is State 3 in Figure 12(a), but it is changed to State 1 in Figure 12(b).

Since there is only a slightly difference between a protocol FSM and its implementation when transfer faults are present, such errors are most likely to occur among the three situation listed above. Therefore, it is reasonable to consider a verification part only for detecting transfer faults. The only way to detect merged states and extra transitions is to use a complete verification part as in the UIOv method. In contrast, detecting transfer faults can be easily achieved without a verification part or with a partial verification part. Examining whether or not a verification part is needed for detection of transfer faults is given in Lemma 1.

Lemma 1:
A verification part is needed for detection of transfer faults iff some states have the same transitions but some tail states of the transitions are different.

Proof. (⇒) If a verification part is needed for detection of transfer faults, it implies that the tail state of a duplicate transition is changed to the same tail state of another duplicate transition in an IUT. This situation can happen only if some
states have the same transitions but some tail states of the transitions are different in the specification.

(⇐) If two states have the same transition but different tail state, once one tail state of the transition is changed to the other in an IUT, these two states not only have the same transition but also the same tail state. Therefore, a verification part is needed.

From Lemma 1, we know that if no transition has duplicates, then no verification part for detection of transfer faults is needed. Otherwise, if some states have the same transition but different tail states, heuristics are proposed for deriving a partial verification part.

Three heuristic are proposed below for generating a partial verification part for detecting transfer faults. If no verification is needed for detecting transfer faults, the verification part is omitted to make a test sequence shorter. In addition, when a verification part cannot be completely eliminated for detecting transfer faults, Heuristics 1 and 2 are proposed to choose a better UIOS and a better unique signature set. The UIOpv method can identify which states need to be verified for which UIOSs and/or signature sets so that a verification part can be kept as small as possible for detecting transfer faults. A better UIOS and a better unique signature set mean that they can be used to detect more faulty IUTs than other UIOSs and other unique signature sets when a verification part is absent.

Heuristics 1 and 2 (presented below) are applied to Algorithm 1 in Section 3.2 to choose a better UIOS and a better signature set. Heuristic 1 is described as follows.
Figure 11: (a) An FSM specification (b) A faulty FSM with a merged state; (c) A faulty FSM with an extra transition
Figure 12: (a) An FSM specification; (b) A faulty FSM with a transfer fault
In Step 2(c) of Algorithm 1, if there is more than one unique transition-state pair and the the UIOS or signature set of the tail state in the transition-state pair is derived, choose the transition-state pair with a minimal number of duplicates of the transition. In Step 4 of Algorithm 1, if there is more than one input/output sequence (or called IO), choose the input/output sequence with a minimal number of duplicates of the first transition in the input/output sequence.

Heuristic 2 is proposed to identify which states need to be verified against which UIOSs and/or which signature sets. There are two cases to be considered. Case 1 deals with a UIOS, and Case 2 deals with a signature set. The detail of Heuristic 2 is described as follows. In Case 1, if a duplicate transition is chosen as the first transition in a UIOS, a verification part is needed only for those states with the same transition and different tail state. Furthermore, if a state (called State A) has the same transition but different tail state, and the transition is either the first transition of a UIOS of State A or the first transition in an input/output sequence (or called IO) in a signature set of State A, no verification is needed for State A. The reason for this is if the tail state of the transition in State A is changed, it can be detected by the Uv part of a test sequence. Therefore, only a few states need to be verified. In Case 2, once a state's signature set is formed, a verification part is needed only for those with the same first transition in every input/output sequence in the signature set. Since states seldom have the same first transition in every input/output sequence in a given signature set, verification for the uniqueness of a signature set is seldom needed.
Table 16: UIOS of each state generated by the UIO and the UIOv methods

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>a/1 (a/1.a/0-non-minimal UIO)</td>
</tr>
<tr>
<td>State 2</td>
<td>a/0.a/1</td>
</tr>
<tr>
<td>State 3</td>
<td>a/0.a/0 and b/1.a/1</td>
</tr>
</tbody>
</table>

Heuristic 3 is proposed to make a signature set as small as possible; it is applied to Step 4 of Algorithm 1. The detail of Heuristic 3 is as follows. If more than one input/output sequence can be chosen to distinguish State A from State B, choose the one distinguishing State A from State B as well as from many other states. Heuristic 3 can be easily implemented by applying each candidate input/output sequence to every state to find out which input/output sequence can distinguish the most states.

Using Figure 12(a) as an example, the UIOSs of State 1 and State 2 are a/1 and a/0.a/1, respectively, and either a/0 < 2 > or b/1 < 1 > can be chosen for State 3. Based on our Heuristic 1, the transition-state pair a/0 < 2 > is chosen since transition a/0 only has one duplicate, but transition b/1 has two. Although State 2 has the same transition a/0, since transition a/0 is part of the UIOS of State 2, no verification is needed for State 2 for detecting transfer faults. As shown in Tables 16 through 20, the test sequence generated by the UIOv method can distinguish the difference between Figure 12(a) and Figure 12(b) without a verification part. Neither the UIOv method without a verification part nor the UIO method can make this distinction.
Table 17: UIOS of each state generated by our improved method

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>a/1</td>
</tr>
<tr>
<td>State 2</td>
<td>a/0.a/1</td>
</tr>
<tr>
<td>State 3</td>
<td>a/0.a/0.a/1</td>
</tr>
</tbody>
</table>

Table 18: Non-optimized UIO-based test sequence (Uv in the first portion, and transition testing in the second)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/-.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>
Table 19: Non-optimized UIOv-based test sequence (Uv in the first portion, * Uv in the second, and transition testing in the last)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/-.a/l</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/l</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/l</td>
</tr>
<tr>
<td>r/-.a/l</td>
</tr>
<tr>
<td>r/-.b/1.a/0</td>
</tr>
<tr>
<td>r/-.a/1.a/0</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0/a/0</td>
</tr>
<tr>
<td>r/-.a/1.a/0/a/1</td>
</tr>
<tr>
<td>r/-.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0.a/0/a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>

Table 20: Non-optimized test sequence generated by our improved method (Uv in the first portion, and transition testing in the last)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/-.a/l</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/l</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-.a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>
3.2 A New Algorithm for Test Sequence Generation

This section presents an efficient algorithm implementing the concepts proposed in Section 3.1. A new upper bound on the length of a UIOS is proved in this section to be \((n - 1) \times n\) rather than \(n^n\), where \(n\) is the number of states. The new upper bound then is used in Algorithm 1 so that Step 2 of Algorithm 1 can terminate sooner if there is no UIOS.

**Theorem 2:** The upper bound on the length of a UIOS is \((n - 1) \times n\), where \(n\) is the number of states.

The proof is as follows. From [42] we know that any two states can be distinguished by an input/output sequence of length less than or equal to \(n\). It is observed that distinguishing State A from both State B and State C can be done in two steps.

The first step is to distinguish State A from State B with an input/output sequence of length less than or equal to \(n\). When the input/output sequence is applied to State C, there are two possible outcomes. The first possible outcome is that the same input/output sequence can distinguish State A from State C as well as State B. This input/output sequence has a length less than or equal to \(n\). The second possible outcome is that the input/output sequence cannot distinguish State A from State C. Assume that States A and B enter States E and F, respectively, after the input/output sequence is applied to States A and B.

Then, the second step is to distinguish States E from F by another input/output sequence of length less than or equal to \(n\). By concatenating the sequences obtained from the first and the second parts, we can derive a new input/output sequence of
length less than or equal to $2 \times n$, which can distinguish State A from both States B and C. If a state needs to be distinguished from $n-1$ states, the same method described can be applied. The final input/output sequence can be viewed as a concatenation of no more than $n-1$ input/output sequences. Therefore, the upper bound on the length of the final input/output sequence is $(n - 1) \times n$. □

The four lemmas are proposed to determine whether a state has a UIOS or a unique signature set before using the test sequence generation procedure. If some states have neither UIOSs nor unique signature sets, there is no need to generate a test sequence because it will not be able to detect some faulty FSMs.

Lemma 2:
A state has a UIOS (1) if it has a unique outgoing transition, or (2) if it has a unique outgoing transition-state pair and the tail state of the transition in the transition-state pair has a UIOS.

The proof is as follows. (1) If a state has a unique outgoing transition, that transition can be used as its UIOS. (2) If a state has a unique outgoing transition-state pair and the tail state of the transition in the transition-state pair has a UIOS, the concatenation of the transition and the UIOS of the tail state can be used as its UIOS. □

Lemma 3:
A state has no UIOS if every transition-state pair leading from the state is not unique.

The proof is as follows. If a transition-state pair is not unique, at least one duplicate exists. Regardless of which transition-state pair is chosen for generating
the UIOS for the state, the UIOS is not unique because duplicate transitions can produce the same sequence.

Divide the states with no UIOS into groups. Two states are put into the same group if they have the same set of outgoing transitions, or if the set of outgoing transitions of one state is a subset of another.

Lemma 4:
A state has a unique signature set (1) if no other state has the same outgoing transition set, or (2) if the state has a unique transition-state pair that is unique in its group and the tail state of the transition in the transition-state pair has a UIOS or a unique signature set, or (3) if the state has a transition leading from its group to another group and no other state in its group has transitions leading from its group to the same group.

The proof is as follows. (1) A state can use its outgoing transition set as its unique signature set if no other state has the same outgoing transition set. (2) If the state has a unique transition-state pair that is unique in its group and the tail state of the transition in that pair has a UIOS or a unique signature set, use that pair to distinguish the state from other states in its group and use those transitions in its group but not in other groups to distinguish the state from other states in other groups. (3) If the state has a transition leading from its group to another, and no other state in its group has transitions leading from its group to the same group, the concatenation of the transition and the outgoing transition set of its tail state can be used to distinguish the state from others in its group since no other state in its
group can be connected to the group the transition leading to. In addition, use the
transitions in its group but not in others to distinguish the state from other states in
other groups.

Lemma 5:
No state has a unique signature set if there is only one group.

The proof is as follows. If there is only one group, each state in this group has the
same outgoing transition set and the tail state of a transition is also in this group.
There are no means to distinguish any two states in the group; therefore no state has
a unique signature set.

After Lemmas 2 through 5 are applied to a given FSM, if some states do not have
unique UIOS and signature set, generating a test sequence is not needed. Otherwise,
Algorithm 1, the new algorithm for generating UIOSs and unique signature sets,
is activated. There are five improvements in Algorithm 1 as compared with the
Algorithm proposed in [42]. First, a faster way to find new UIOSs is suggested in
Step 2(c). For a unique transition-state pair, concatenating the transition and the
UIOS of the tail state of the transition in the transition-state pair forms a new UIOS
if the UIOS of the tail state is derived. Second, cutting off subtrees that are no longer
useful for generating UIOSs is done in Step 2(d) and Step 2(e) to save time. Third,
Heuristic 3 is included in Step 4 to form a smaller signature set. Next, Heuristics
1 and 2 are included in Algorithm 1 so that better fault coverage can be achieved
without a verification part or with a partial verification part. Finally, the new upper
bound, \((n - 1) \times n\), is used in Algorithm 1.
Algorithm 1:

1. For each different transition, count the number of times it occurs, denoted by $Count(t)$ where $t$ is a transition.

2. Repeat the following steps until either every state's UIOS has been found or the tree level has reached the upper bound.
   
   (a) Use $(n - 1) \times n$ as the upper bound on the length of a UIOS to generate UIOSs for states whose UIOSs are not found yet. Level by level explore all the possibilities for all states in parallel by incrementing the length of the tree by one.
   
   (b) When a level is finished, check to see whether any state's UIOS has been found. If none is found at this level, continue until one is found.
   
   (c) Do the following things for every state. If a state has a unique transition-state pair and the tail state of the transition in the transition-state pair has a UIOS found in Step 2(b) and Step 2(c), use the concatenation of the transition and the tail state's UIOS as its UIOS. Apply Heuristic 1 to choose the transition-state pair that has a minimal number of duplicate transitions. This step may be repeated multiple times until no new UIOS can be derived.
   
   (d) For each state whose UIOS is found in Step 2(b) and Step 2(c), decrease $Count(t)$ for each transition $t$ in the state's outgoing transition set.
   
   (e) For each transition $t$ whose $Count(t)$ equals 0, cut off the subtrees leading
from it at the root level. (Note that since a transition may have duplicates, each subtree of the same transition can be cut off at the same time.)

3. Calculate $IO(i,k)$s for every pair of states, where $i \neq k$ and $IO(i,k)$ is an input/output sequence of length less than or equal to $n$ that can distinguish state $S_i$ from state $S_k$.

4. A signature set of state $S_i$ can be formed by selecting one $IO(i,k)$ for each state $S_k$, where $k \neq i$. There may be more than one input/output sequence of the same length that can distinguish two states, called State A and State B. It is better to choose the sequence that can distinguish State A from State B as well as others. Heuristic 3 is applied to form a smaller signature set and Heuristic 1 is applied to choose the IO sequence that has a minimal number of duplicate transitions.

5. Heuristic 2 is applied to find the necessary input/output sequences for verification purposes.

6. Once a signature set is formed for a state, a verification part is only needed for the states that have the same first transition in every input/output sequence in the signature set.

The test sequence generated by the UIOpv method (Algorithm 1) consists of two or three parts. The Uv and the Tr parts always exist, whereas the Uv is present if verification is needed.
3.3 Comparisons and Analyses

The UIOpv method is compared with the UIO method, the PUIO method [15] and the UIOv method. The PUIO method made improvements on the UIO method. It can derive UIO sequences for states with UIOSs and sets of Partial UIO sequences (PUIOSs) for states without UIOSs. Five different aspects are used for comparison, namely, (1) applicability, (2) fault coverage, (3) time for finding UIOSs and signature sets, (4) length of UIOSs, and (5) number of input/output sequences in a signature set and their lengths.

Applicability

The UIOpv method has the same applicability, which is the widest, as the UIOv method and the PUIO method. It generates test sequences for the protocols in which some states have UIOSs and some do not. It is well known that the UIOv method has wider applicability than the UIO method since the UIO method can only be applied to the protocols that every state has a UIOS. Since the PUIO method can derive a set for states without UIOSs, it has the same applicability as the UIOv and the UIOpv methods.

In summary, the applicability among the different methods is:

\[ UIO < UIOv = PUIO = UIOpv \]

Fault Coverage

Better fault coverage can be achieved by verifying the uniqueness of states' UIO sequences and signature sets. Heuristics 1 and 2 are proposed to choose a better UIOS and/or a better signature set and to identify which states need to be verified
to keep the verification part small. Therefore, even without a verification part, the UIOpv method can achieve better fault coverage than the version of the UIOv method without a verification part, denoted as UIOv'.

However, when a complete verification part is included in both the UIOv method and the UIOpv method, both methods can have the same fault coverage, which is better than the UIO method. Since a signature in the UIO method is not unique and the UIO method does not derive a verification part, some faulty IUTs will be missed by the UIO method. However, they can be detected by both the UIOpv, and UIOv methods. Although the PUIO method derives unique sets of PUIOSs for states, it does not derive a verification part. Therefore, some faulty IUTs will be missed by the PUIO method.

In summary, the fault coverage among the different methods is:

a. without a verification part:
   \[ UIO < PUIO = UIOv' < UIOpv \]

b. with a complete verification part:
   \[ UIO < PUIO < UIOv = UIOpv \]

**Time Complexity**

The UIOpv method can find UIOSs and signature sets faster than the UIOv method since Algorithm 1 is better than the algorithm used by the UIOv method. By using concatenation in Step 2(c) of Algorithm 1, it is faster to find a UIOS for a state. By using \((n - 1) \times n\) as the new upper bound on the length of a UIOS, the time complexity of the UIOpv method is reduced from \(O(n^2 \times D^{2 \times n^2})\) to \(O(n^2 \times D^{(n-1) \times n})\),
where \( n \) is the number of states and \( D \) is the maximum degree of outgoing transitions of a state. In addition, a pruning technique is also included in Algorithm 1 to save more time. The PUIO method proposes two theorems, which are used in its algorithm to improve its efficiency. As compared with the PUIO method, the UIOpv method can derive UIOSs faster if every state in a given protocol has a UIOS. The reason for this is that the UIOpv method's Algorithm 1 uses concatenation of an existing UIOS and a transition to form a new UIOS. Otherwise, the PUIO method does better.

In summary, the time complexity in the worst case among the different methods is:

- **UIO and UIOv:** \( O(n^2 \times D^{2n^2}) \)
- **PUIO:** \( O(n^2 \times D \times 3^{n-1}) \)
- **UIOpv:** \( O(n^2 \times D^{(n-1)n}) \)

**Length of a UIOS**

Since concatenation is used in Step 2(c) of Algorithm 1, the lengths of some UIOSs generated may not be the shortest. However, the upper bound on the length of a UIOS derived by the UIOpv method is shorter than that derived by the UIO and the UIOv methods by using the new upper bound proved in Algorithm 1. Since the PUIO method can generate \( n \times 3^{n-1} \) possible vertices, in the extreme case all possible vertices will be on the same path. Therefore, the upper bound on the length of a UIOS derived by the PUIO method is \( n \times 3^{n-1} \).
In summary, the length of a UIOS for the different methods is:

- UIO and UIOv: \(1 \leq \text{length} \leq 2 \times n^2\)
- PUIO: \(1 \leq \text{length} \leq n \times 3^{n-1}\)
- UIOpv: \(1 \leq \text{length} \leq (n - 1) \times n\)

**Size and Length of a Signature Set**

Since the UIOpv method and the UIOv method use a similar procedure to derive the signature sets, the number of input/output sequences and their lengths are about the same. There is no comparison made between the UIOpv method and the PUIO method in terms of the size and length of a signature set or a set of PUIOSs since these two methods are based on different procedures.

### 3.4 Conclusions

The UIOpv method is proposed and coded using C. It has a number of improvements over the UIOv method. First, it proves that the UIOv method will not derive non-unique signature set by showing that the UIOv method can find a test sequence for any minimal, strongly connected and completely specified FSM. Second, the UIOpv method generates only a partial verification part for detecting transfer faults. Third, a new upper bound on the length of a UIOS was proved to be \((n - 1) \times n\), which is decreased by a factor of \(n^{n-2}\). Fourth, the UIOpv method's Algorithm 1 is more efficient since it introduces a faster way to form UIOSs, a means to cut off subtrees, and a new upper bound on the length of a UIOS.
The major advantages of the UIOpv method are widest applicability, better fault coverage and less computation time. However, the only drawback of the UIOpv method is that it may not generate the shortest UIOSs for some states. If generating shortest UIOSs is important, Algorithm 1 can be easily modified to derive all the possible UIOSs using \((n - 1) \times n\) as the upper bound on the length of a UIOS.
CHAPTER IV

The Transition-State Pairs method (TSP)

The key concept behind the Transition-State Pairs method (TSP) is that each state can be uniquely identified by the set of its outgoing transition-state pairs. The TSP method considers not only the transition but also the tail state of the transition. Without considering the tail state of a transition, two states could potentially have the same set of outgoing transitions.

This chapter presents a new test sequence generation method based on the unique set of transition-state pairs. Since the internal behavior of an FSM cannot be observed, a transition’s tail state cannot be verified directly. Instead of using a state’s name in a transition-state pair, its identity should be used.

The TSP method consists of two steps. In the first step, a unique set of transition-state pairs is derived for each state. In order to generate the test sequence efficiently, only the minimal number of distinguishing transition-state pairs needed. In the second step, a unique set of input/output sequences is derived for each state. After the first step derives USOTSPs for each state, the second step transforms each transition-state pair into a set of input/output sequences. The TSP method is coded using C and the detailed program is listed in Appendix B.
4.1 Basic Concept and Definitions

Assume that there are \( n \) states, \( S = \{s_1, s_2, \ldots, s_n\} \) in an FSM. A set of outgoing transitions of a state \( s_j \), denoted as \( \text{Trans}(s_j) \), consists of all the outgoing transitions leading from \( s_j \). A set of outgoing transition-state pairs of a state \( s_j \), denoted as \( \text{STSP}(s_j) \), consists of all the outgoing transition-state pairs leading from \( s_j \). A unique set of transition-state pairs of a state \( s_j \), denoted by \( \text{USOTSP}(s_j) \). It consists of some transitions, denoted by i/o, and/or transition-state pairs, denoted by \( i/o < s > \), leading from \( s_j \) that can uniquely identify the state \( s_j \). A unique set of input/output sequences of a state \( s_j \), denoted by \( \text{USIOS}(s_j) \), consists of some input/output sequences. The only difference between \( \text{USOTSP}(s_j) \) and \( \text{USIOS}(s_j) \) is that every transition-state pair in \( \text{USOTSP}(s_j) \) is transformed into the concatenation of the transition and the whole or partial set of its tail's identity. A difference set of state \( s_j \) and state \( s_k \), is denoted by \( \text{DIF}(s_j, s_k) \), which is equivalent to \( \text{Trans}(s_j) - \text{Trans}(s_k) \). A transition is in \( \text{DIF}(s_j, s_k) \) if the transition is in \( \text{Trans}(s_j) \) but not in \( \text{Trans}(s_k) \).

The \( TSP \) method consists of two algorithms: Algorithm 2 and Algorithm 3. Algorithm 2 derives a unique set of transition-state pairs (USOTSP) for each state. Since a state's name is used in a USOTSP and the state's name cannot be observed through the output port of an IUT, Algorithm 3 transforms each transition-state pair into a set of input/output sequences. The set of sequences can be obtained by concatenating the transition in the transition-state pair and the identity of its tail state. Since the identity of a state in the \( TSP \) method is always a set of input/output
sequences, the concatenation of the transition and the identity of its tail state is also a set of input/output sequences. For example, if a transition-state pair is 1/2 and State 3, and the identity of State 3 is \{2/3, 4/5\}, then the representation of the transition-state pair in the TSP method is \{1/2.2/3, 1/2.4/5\} where \("\) is used to separate two consecutive input/output symbols. In order to derive a short test sequence, a partial set of a tail state's identity rather than the whole set is used if the concatenation of the transition and the partial set can distinguish states.

4.2 Algorithm for Finding USOTSPs

Algorithm 2 derives the minimal USOTSP for each state. Case 1 of Algorithm 2 can find USOTSPs and USIOSs for states with a unique outgoing transition. The unique outgoing transition of a state is used as the USOTSP and the USIOS. In Case 2, USOTSPs and USIOSs are found for states that have (1) a unique transition-state pair and (2) the USIOS of the tail state of the transition in the transition-state pair found either in Case 1 or Case 2. Case 2 can be repeated many times until no new USIOS are found. The USIOSs found in Case 1 and Case 2 consist of only one element so they can be viewed as UIOSs.

In Case 3, states are divided into groups. Two states are put into the same group if they have the same set of outgoing transitions, or if the set of outgoing transitions for one is a subset of the other. Step 1 of Case 3 finds a difference set of transitions between a state and every state that is not in the same group. Step 2 of Case 3 finds a difference set of transition-state pairs between a state and every state in the same group as the state is in. In order to derive a minimal USOTSP, the transition or the
transition-state pair in a difference set that has the most duplicates is chosen. Steps 3, 4, and 5 of Case 3 decide which transition or which transition-state pair to select by counting the number of occurrences of every transition in difference sets and by sorting difference sets based on the number of elements in a set.

Algorithm 2

Case 1: For each of the states with unique outgoing transitions, select one unique outgoing transition as its USOTSP and as its USIOS.

Case 2: For each of the states with unique transition-state pairs and a USIOS for its tail state, select one unique transition-state pair. Use the concatenation of the transition and the USIOS as its USOTSP and USIOS. This step may be repeated many times until no new USIOS is derived.

Case 3: Perform the following steps to find the USOTSPs for states.

1. For a state, called State B, divide all states into groups based on the relationship of State B and each state. State A and State B should be put into the same group if (1) $\text{Trans}(A) \subseteq \text{Trans}(B)$ or (2) $\text{Trans}(B) \subseteq \text{Trans}(A)$; otherwise, State A and State B should be put in different groups.

2. Find $\text{DIF}(B, s_i)$ for every $s_i \in S$, where $s_i$ and State B are not in the same group.

3. For State B, find $\text{DIF}(\text{STSP}(B), \text{STSP}(s_i))$ for every $s_j \in S$, where $s_j$ and $B$ are in the same group, $s_j \neq B$. ($\text{DIF}(\text{STSP}(B), \text{STSP}(s_i))$ is a difference set of transition-state pairs for State B and State $s_i$.)
4. For every different transition \( t \) in the DIF sets created in Step 2 and Step 3, count the number of times the transition occurs, denoted by \( \text{Count}(t) \).

5. Sort the DIF sets created in Step 2 and Step 3 in ascending order based on the number of elements in a DIF set.

6. Choose a transition or a transition-state pair from each DIF set based on the sorted sequence. Select the transition or the transition-state pair that has the maximum \( \text{Count} \) from each DIF set. Add the transition or the transition-state pair into the USOTSP of State \( B \).

4.3 Algorithm for Transforming Transition-State Pairs into Input/Output Sequences

Some elements in a USOTSP may be transition-state pairs that have the a format \( i/o < s > \). Distinguishing State \( s \) from other states is problematic. Instead of using a state's name, a set of input/output sequences is used to identify the state. Therefore, a transition-state pair needs to be transformed into the concatenation of the transition and the set of input/output sequences. The set of the input/output sequences is either the whole set or partial set of the identity of the transition's tail state.

Algorithm 3 transforms every transition-state pair into a set of input/output sequences. The same concept used in the UIO method \([42]\) to determine a sequence, denoted as \( IO(s, s_i) \), that can distinguish State \( s \) from State \( s_i \), is used in Algorithm 3. Step 4 and Step 5 of Algorithm 3 guarantees that the number of input/output sequences is smaller than or equal to the number of states in a given FSM.
Algorithm 3

For each State $s_i$ whose USIOs has not been found, do the following:

1. Determine the number of transitions in $USOTSP(s_i)$, denoted as $NumTrans(s_i)$.

2. For each transition-state pair, $TS_j$ in $USOTSP(s_i)$, determine the number of transition-state pairs with the same transition but tail states, denoted as $Number(TS_j)$.

3. Let $Sum = NumTrans(s_i) + \sum Number(TS_j)$ where $TS_j$ is a transition-state pair in $USOTSP(s_i)$.

4. If $Sum \leq n$ then
   
   (a) Copy each transition in $USOTSP(s_i)$ into $USIOS(s_i)$.
   
   (b) For each transition-state pair do the following:
      
      i. Assume that there are $k$ different tail states, $s_1, s_2, \ldots, s_k$ from State $s$ where State $s$ is the tail state of the transition-state pair.
      
      ii. Find $IO(s, s_j) \forall j$ $1 \leq j \leq k$, where $IO(s, s_j)$ is an input/output sequence of length less than or equal to $n$.
      
      iii. The transition-state pair is transformed into the concatenation of the transition and the set of IOs found in the previous step. The result of concatenation is put into $USIOS(s_i)$.

5. If $Sum > n$ then find $IO(s_i, s_j) \forall j$ $1 \leq j \leq n$ and $s_i \neq s_j$, and $IO(s_i, s_j)$ is put into $USIOS(s_i)$. 
Figure 10 is used as an example to go through the procedures of Algorithm 2 and Algorithm 3. The result obtained in Step 1 of Algorithm 2 is as follows.

\[ \text{USOTSP}(A) = \text{USIOS}(A): \{0/1\} \]
\[ \text{USOTSP}(B) = \text{USIOS}(B): \{1/1\} \]
\[ \text{USOTSP}(X) = \text{USIOS}(C): \{3/2\} \]

The result obtained in Step 2 of Algorithm 2 is as follows.

\[ \text{DIF}(C,A) = \{2/1\} \]
\[ \text{DIF}(C,B) = \{1/0\} \]
\[ \text{DIF}(C,D) = \{2/1, 1/0\} \]
\[ \text{DIF}(C,X) = \{1/0\} \]
Therefore, \[ \text{USOTSP}(C): \{2/1, 1/0\} \]

\[ \text{DIF}(D,A) = \{2/1\} \]
\[ \text{DIF}(D,B) = \{1/0\} \]
\[ \text{DIF}(D,C) = \{2/1, 1/0\} \]
\[ \text{DIF}(D,X) = \{1/0, 2/1\} \]
Therefore, \[ \text{USOTSP}(D): \{2/1, 1/0\} \]

The result obtained in Algorithm 3 is as follows.

\[ \text{USIOS}(A): \{0/1\} \]
USIOS(B): {1/1}
USIOS(C): {2/1.0/1, 1/0.1/1}
USIOS(D): {2/1.2/1, 1/0.1/0}
USIOS(X): {3/2}

Appendix E lists the sample output generated by the program implementing the TSP method using Figure 10(a) as the input protocol specification. Although the TSP method differs from other methods in generating a state's unique identity, the segments for transitions can be easily derived by using the same procedure used in the UIO method.

The following three lemmas can be used to check whether or not a test sequence can be found for a given FSM. The three lemmas should be applied to a given FSM before Algorithm 2 is used. If no test sequence can possibly be found, there is no need to waste time on test sequence generation.

Lemma 6:
An FSM is minimal and completely specified iff each state is in a different group after the \( P_k \) table method as described in [23] is applied to the FSM for \( n \) times, where \( n \) is the number of states.

Proof: (⇒) Since the FSM is minimal and completely specified, no states can have the same set of output sequences when a set of input sequences is applied to each state. Therefore, \( n \) states will be in \( n \) different groups after the \( P_k \) table method is applied.
(⇐) If each state is in a different group, no states can have the same set of output sequences when a set of input sequences is applied to each state. Therefore, the FSM is minimal and completely specified.

Lemma 7:
A USOTSP can be found for each state if the given FSM is minimal and completely specified.

Proof. (⇒) If a USOTSP can be found for each state, it implies that each state has a unique set of outgoing transition-state pairs. Therefore, the FSM is minimal and completely specified.

(⇐) If an FSM is minimal and completely specified, no states can have the same set of outgoing transition-state pairs. Therefore, there exists a USOTSP for each state.

Lemma 8:
Every USOTSP can be transformed into a USIOS.

Proof. This is obvious as shown in Algorithm 3.

An example, as shown in Figure 13, illustrates that after Lemma 6 is applied, one can conclude that the given FSM is not minimal. Based on Lemmas 7 and 8, one can conclude that the USIOSs of some states will not be found even before the test generation procedure starts.
Figure 13: An FSM with no UIOS for States A, B and C
4.4 Increasing Fault Coverage

The test sequences generated by both the W and the Wp methods always consist of complete verification parts. The advantage of using a complete verification part is to detect more errors, but its disadvantage is that it causes a long test sequence. Since it is observed that for some protocols a partial verification part is good enough to detect transfer errors, two heuristics and a lemma are proposed in this section to derive a partial verification part.

The advantage of a partial verification part was illustrated in Section 3.1. The same concept of using a partial verification part for detecting transfer faults is adopted in the TSP method. Two heuristics are proposed for forming a partial verification part. Heuristic 4 is proposed to select a better element of a USOTSP, and Heuristic 5 is proposed to identify which states need to be verified for which USIOSs to keep the verification part small. A better element of a USOTSP is the one that has better fault coverage than other candidates when a verification part is not considered. Both heuristics can be applied to each Step of Algorithm 2, so better fault coverage can be achieved when the verification of uniqueness of USOTSPs is not considered. Heuristic 4 says that if there is more than one candidate (transition-state pairs or transitions), select the one with the fewest duplicates. Heuristic 5 says that once a USOTSP is formed for a state, a verification part is needed only for the states having the same USOTSP set. Since states seldom have the same USOTSP set, verification for uniqueness of a USOTSP is seldom needed.
Table 21: UIOS of each state generated by the UIO and the UIOv methods

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>a/1 (a/1.a/0-non-minimal UIO)</td>
</tr>
<tr>
<td>State 2</td>
<td>a/0.a/1</td>
</tr>
<tr>
<td>State 3</td>
<td>a/0.a/0 and b/1.a/1</td>
</tr>
</tbody>
</table>

The test sequence generated by the TSP method with Heuristic 4 has better fault coverage than the UIO method and the version of the UIOv method with no verification part. If both Heuristic 4 and Heuristic 5 are used by the TSP method, the test sequence has the same ability as the UIOv method to detect transfer faults, but fewer segments in its verification part.

For example, as shown in Figure 12(a), Figure 12(b) and Tables 21 through 25, the TSP method outperforms both the UIO and the UIOv methods. In this example, the identity of State 3 found by the TSP method is different from the ones found by the UIO and the UIOv methods; therefore, the test sequences generated by them are slightly different. The test sequence generated by the UIO method cannot distinguish the difference between Figure 12(a) and Figure 12(b). Using the verification part of the UIOv method, the difference between Figure 12(a) and Figure 12(b) can be detected. After Lemma 1 and Heuristic 4 are applied to Figure 12(a), it is observed that without a verification part the TSP method can still detect the difference between Figure 12(a) and Figure 12(b).
Table 22: USIOS of each state generated by the TSP method

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>a/1</td>
</tr>
<tr>
<td>State 2</td>
<td>a/0.a/1</td>
</tr>
<tr>
<td>State 3</td>
<td>a/0.a/0.a/1</td>
</tr>
</tbody>
</table>

Table 23: Non-optimized UIO-based test sequence (Uv in the first portion and transition testing in the second)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/.a/1</td>
</tr>
<tr>
<td>r/.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/.a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/.a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/.a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/.a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/.a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>
Table 24: Non-optimized UIOv-based test sequence (Uv in the first portion, * Uv in the second, and transition testing in the last)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/-a/1</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-a/1</td>
</tr>
<tr>
<td>r/-b/1.a/0</td>
</tr>
<tr>
<td>r/-a/1.a/0</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0/0.a/0</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.b/1.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>

Table 25: Non-optimized TSP-based test sequence (Uv in the first portion and transition testing in the last)

<table>
<thead>
<tr>
<th>Segments of the Test Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>r/-a/1</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.a/0.a/0.a/1</td>
</tr>
<tr>
<td>r/-a/1.b/1.b/1.a/1</td>
</tr>
</tbody>
</table>
Table 26: USIOS of each state generated by the TSP method

<table>
<thead>
<tr>
<th>State Name</th>
<th>UIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>state 0</td>
<td>B/2</td>
</tr>
<tr>
<td>state 1</td>
<td>B/1.B/0</td>
</tr>
<tr>
<td>state 2</td>
<td>B/0</td>
</tr>
<tr>
<td>state 3</td>
<td>A/0.B/1.B/2</td>
</tr>
<tr>
<td>state 4</td>
<td>B/1.B/2</td>
</tr>
</tbody>
</table>

The TSP method has an option to choose no verification part, a partial verification part, or a complete verification part. When no verification part or a partial verification part is chosen, the TSP method generates shorter test sequences than the W and the Wp methods. For some protocols, the fault coverage of the test sequences derived by the TSP method is as good as the W method and the Wp method. When a complete verification part is chosen, the TSP method may generate shorter test sequences than the W and the Wp methods since it usually generates fewer sequences as a state's identity. An example as shown in Figure 3, Table 4, Table 5, and Table 26, illustrates that the TSP method only generates one sequence as a state’s identity, while the W method generates three and the Wp method generates one or two.

4.5 Comparisons and Analyses

Five different methods, the UIOpv method, and the TSP method are compared using the following six criteria: (1) fault coverage; (2) applicability; (3) time complexity; (4)
length of an input/output sequence and the number of input/output sequences in a state’s identity; (5) number of segments produced; and (6) ability to test applicability.

**Fault Coverage**

Better fault coverage can be achieved by verifying the uniqueness of each state’s identity. Heuristic 4 is proposed to select a better element for a USOTSP and Heuristic 5 is proposed to identify which states need to be verified for which USIOSs to keep the verification part as small as possible. Therefore, without a verification part, some transfer faults can be detected by the TSP method but may be missed by the version of the UIOv method with no verification part, denoted as $UIOv'$. Hence, the TSP method can achieve better fault coverage than the $UIOv'$ method.

However, when a complete verification part is included in both the UIOv method and the TSP method, the same fault coverage is achieved, which is better than the UIO method but the same as the D-method. Since a signature in the UIO method is not unique, some faulty IUTs will be missed by the UIO method, but can be detected by both the TSP method and the UIOv method. Since both the UIOpv and the TSP methods use similar heuristics to select an input/output sequence, they have the same fault coverage.

The $W$ and $Wp$ methods can detect extra states as well. The price paid by them is that more long segments will be derived. The TSP method can be modified easily to detect extra states by adding more long segments. When a complete verification part is used and extra segments for detecting extra states are added, the TSP method has the same fault coverage as both the $W$ and the $Wp$ methods.
In summary, the fault coverage among the different methods is:

a. without a verification part

\[ \text{UIO} < \text{UIO}_v < \text{UIO}_{pv} = \text{TSP} < \text{D} < \text{W}_p = \text{W} \]

b. with a complete verification part

\[ \text{UIO} < \text{UIO}_v = \text{UIO}_{pv} = \text{TSP} = \text{D} < \text{W}_p = \text{W} \]

**Applicability**

As shown in Lemmas 7 and 8, if a given FSM is minimal, the TSP method can find a unique identity for each state, and so can the UIO\(_v\), the W, and W\(_p\) methods as stated in [74, 14, 21]. Although Chan claims in [10] that the UIO\(_v\) method cannot handle the case in which a signature set is not unique, it was proved in Chapter 3 that the UIO\(_v\) method can derive a test sequence for any FSM that is minimal.

The UIO\(_v\), the UIO\(_{pv}\), the TSP methods have the same applicability as the W and the W\(_p\) methods. In addition, they have better applicability than both the D and the UIO methods since there are some cases in which a state cannot be uniquely identified by the D and the UIO methods.

In summary, the applicability among the different methods is:

\[ \text{D} < \text{UIO} < \text{UIO}_v = \text{UIO}_{pv} = \text{TSP} = \text{W}_p = \text{W} \]
Time Complexity

Although deriving "distinguish tree" is similar to deriving a UIOS, the upper bound on the length of the distinguish sequence is much larger than those used by the UIO and the UIOv methods. Thus, the time complexity of the D method is much higher than those of the UIO and the UIOv methods. The TSP method generates different sets for different states while the W method derives only a single set for all states. Although the TSP method appears more complicated than the W method, the TSP method uses less time. Among the seven methods, the TSP method uses the least amount of time.

In summary, the time complexity among the different methods is:

D: \( O(K^n(n+1)) \)
UIO and UIOv: \( O(n^2 \times D^2 \times n^2) \)
W and Wp: \( O(n \times K^n) \)
UIOpv: \( O(n^2 \times D^{n \times (n-1)}) \)
TSP: \( O(n \times D^n) \)

Length and Size

The upper bound on the length of an input/output sequence in a USIOS in the TSP method is \( n \), which is the best upper bound. In addition, the number of input/output sequences in the USIOS in our TSP method is bounded by \( n \).

In summary, the length and the size of a state's identity among the different methods is:
D: \[ 1 \leq \text{length} \leq (n - 1) \times (n^n) \text{ and } size = 1 \]

UIO and UIOV: \[ 1 \leq \text{length} \leq 2 \times (n^2) \text{ and } 1 \leq size \leq n \]

W and Wp: \[ 1 \leq \text{length} \leq n \text{ and } 1 \leq size \leq k^{\text{length}} \]

UIOPv: \[ 1 \leq \text{length} \leq (n - 1) \times n \text{ and } 1 \leq size \leq n \]

TSP: \[ 1 \leq \text{length} \leq n \text{ and } 1 \leq size \leq n \]

**Number of Segments**

In most cases, the TSP method can derive fewer number of segments than the W method and the Wp method due to the following two reasons. The first is that the TSP method usually generates fewer sequences as a state's identity as shown in Figure 3, Tables 4, 5, and 26, and the second is that the TSP method can choose no verification part, a partial verification part, or a complete verification part.

Since our TSP method has a partial verification part to detect transfer errors, it has fewer segments than the UIOV method in most cases. However, when a complete verification part is included into the TSP method, the TSP method has about the same number of segments as the UIOV method. In conclusion, the number of segments derived by the TSP method is between the number of segments derived by UIO method and by the W method. Usually, the more segments there are, the longer the test sequence is.
**Testing Applicability**

The D and the UIO methods do not provide any means to examine whether a test sequence can be derived for a given FSM. In contrast, the UIOv, the W, the Wp, the UIOpv, and the TSP methods can apply the $P_k$ table method to a given FSM to determine whether the given FSM is minimal before the test sequence generation procedure starts. If the given FSM is not minimal, these methods will not be able to derive a test sequence that will successfully detect faulty IUTs. Therefore, these methods can decide not to start the whole test sequence generation procedure.

In summary, the ability to test applicability among the different methods is:

- **D**: no method
- **UIO**: no method
- **UIOv**: $P_k$ table with time complexity $O(k \times n^2)$
- **UIOpv**: $P_k$ table
- **W and Wp**: $P_k$ table
- **TSP**: $P_k$ table

### 4.6 Conclusions

A new approach for test sequence generation is proposed in this chapter. This method is called the Transition-State Pairs (TSP) method since the concept of identifying a state with a unique set of transition-state pairs is used.
Once the identities of states have been determined, our TSP method applies the same algorithm used in the UIO method to generate a test sequence. The length of the test sequence derived by our TSP method is greater than that of the UIO method but less than that of the W method.

A verification part is not always needed in the TSP method to detect transfer faults. When a verification part is not considered, the TSP method can achieve better fault coverage than both the UIO and the version of the UIOv method with no verification part. When a verification part is considered for detecting transfer faults, the TSP method needs only a smaller verification part than the UIOv method.

In summary, our TSP method retains the advantages of the UIO and the W methods but eliminates their disadvantages; therefore, our TSP method can have the best applicability and the best fault coverage, while maintaining the shortest possible test sequence. In addition, the TSP method uses the least amount of time.
CHAPTER V

Minimal Length Test Sequence Generation (MLTG) Method

From the literature it is observed that two of the most important techniques in generating shorter test sequences are multiple UIO sequences and segment overlap. The former provides more options for selecting UIOSs and the latter measures the number of the common transitions between any two segments. The shortest test sequence may be obtained by solving the Chinese Postman Tour problem in conjunction with both techniques.

Chen’s and Miller’s methods use both techniques to shorten test sequences. Chen proposes only a heuristic algorithm to connect the overlapped segments; therefore, segments may not be connected in an optimal way. Miller’s method takes both techniques into account when connecting segments and can generate the shortest test sequences. However, both methods cannot handle the protocol specifications in which there are states with no UIOS, and none of them verify the uniqueness of states’ UIOSs or signature sets.

To remedy the drawbacks of Chen’s and Miller’s approaches, the MLTG method[75] is proposed for improving their applicability and fault coverage. The MLTG method consists of two approaches; one without the verification of the uniqueness of states’
UIOSs and signature sets, and the other with the verification. Since the MLTG method is an improvement over the Miller's method, a brief discussion of Miller's method is given in the next section.

5.1 Analysis of Miller's Approach

It is observed that the shortest UIO sequence from State 1 to State 2 is equivalent to the UIO sequence of State 1 plus the shortest sequence from the tail state of State 1's UIO sequence to State 2. By using the shortest UIO sequences to connect edges and disjoint paths, Miller's cost function can take advantage of multiple UIO sequences and segment overlap implicitly. Therefore, the shortest test sequences may be found.

This section describes Miller's two algorithms and two cost functions, and addresses the errors and drawbacks, which can be improved. First, the new definition of definitely diagnosable is not strong enough as compared with the original definition. It is defined in the original definition that an FSM machine is definitely diagnosable if each state can be distinguished by some unique sequences. However, in the new definition, an FSM machine is definitely diagnosable if there are no two states going into the same state with the same input/output label. Figure 14 illustrates that the new definition is different from the original one. The graph in Figure 14 is definitely diagnosable by the new definition, but it is not by the old one since no sequences can distinguish the three states in Figure 14.

Second, the condition for applying Algorithm-2 is incorrect. Miller's intention is to use Algorithm-3 for protocols in which some states have converging edges and to use Algorithm-2 for protocols in which every state has an UIOS and no state has a
converging edge. However, the condition for applying Algorithm-2 only requires that each state has an UIOS. The correct condition is that (1) every state has an UIOS and (2) no state has an converging edge.

Third, Miller's two algorithms may overlook the shorter test sequences. Miller's Algorithm-2 generates a test sequence by appending the UIOS of the last state in a Euler Tour (Path) to the Euler Tour (Path). As a result, a test sequence's length depends on the Euler Tour (Path) that is selected. When there is more than one Euler Path (Tour) in an FSM, Miller's Algorithm-2 selects one randomly. Therefore, shorter test sequences may also be overlooked Algorithm-3 generates disjoint paths and partitions an FSM into disjoint paths. When there is more than one way to do partitions, Miller's Algorithm-3 selects one randomly. Since different partitions results in test sequences with different lengths, randomly selecting one may overlook the shorter test sequence. An example, as shown in Figure 15, illustrates that there
is more than one set of disjoint paths for a given FSM. Figure 16 shows that test sequences generated have different lengths based on the set of disjoint paths chosen in Figure 15(c).

The first improvement over Miller's approach is to increase its fault coverage. Currently, many transfer faults cannot be detected by Miller's approach because it does not verify the uniqueness of states' UIOSs and signature sets. For example, as shown in Figure 17, the test sequence generated for Figure 17(a) is e1-c2-c3-e4-e5-e6-e3, which cannot distinguish Figure 17(a) from Figure 17(b). Also, as noted in [10], many faulty FSMs cannot be detected due to the non-unique UIO sequences of states. A UIO sequence unique to an FSM specification may not unique to its implementation. For example, the test sequence generated by Miller's approach for Figure 12(a) cannot distinguish the difference between Figure 12(a) and Figure 12(b) because it cannot verify the uniqueness of UIO sequences and signature sets for states in the IUT.

5.2 Approach without Verifying the Uniqueness of UIO Sequences

The first approach uses segment overlap as well as multiple UIO sequences to shorten test sequences. Although it is similar to Miller's approach, it fixes most of Miller's errors and drawbacks. In addition, this approach is designed to handle the protocols in which some states do not have UIO sequences but have signature sets [10]. In summary, it generates shorter test sequences and has better applicability than Miller's approach. Since a verification part for ensuring the uniqueness of states' UIO
Sets of Disjoint Paths:

(1) P₁ : 5→3→2→4→5→3  
P₂ : 1→2→3

(2) P₁ : 5→3→2→4→5→3  
P₂ : 2→1→2

Figure 15: (a) An FSM with converging edges; (b) The FSM after converging edges are removed; (c) Two sets of disjoint paths
Figure 16: (a) The test sequence generated using the first set of disjoint paths; (b) The test sequence generated using the second set of disjoint paths.
Figure 17: (a) An FSM; (b) A faulty FSM
sequences is not considered in the first approach, the first approach's fault coverage is the same as Miller's approach.

Three algorithms are proposed to apply to different conditions and two new cost functions are proposed to calculate the cost from one segment to another. These cost functions and algorithms are described in detail below.

**Conditions for Applying Algorithms**

As mentioned earlier, the condition for applying Miller's algorithms is incorrect. The first approach can handle not only the protocols in which every state has UIO sequences but also the protocols in which some states do not have UIO sequences but have signature sets. The condition to decide which algorithm to use is as follows.

If each state in the given FSM has UIO sequences then

if no state has a converging edge then

perform new-Algorithm-2

else

perform new-Algorithm-3

endif

else if each state has either UIO sequences or signature sets then

perform new-Algorithm-4

endif

**New-Algorithm-2**

Similar to Miller's Algorithm-2, the new-Algorithm-2 generates a test sequence that is a Euler Tour (Path) plus a UIO sequence of the last state in the Euler Tour (Path).
In addition, the new-Algorithm-2 ensures that the Euler Tour (Path) begins with the initial state. If more than one Euler Tour (Path) can be selected, the one in which the last state with the shortest UIO sequence is chosen. The new-Algorithm-2 is given as follows.

If an FSM $G$ has a Euler Tour then

Test Seq. = The Euler Tour beginning with the initial state $v_i$ @ $UIO_i$

else if $G$ has a Euler Path beginning with the initial state then

find all Euler Paths beginning with the initial state
select the Euler Path beginning with the initial state, $v_i$, and ending at the State $v_j$ with the shortest UIO sequence
Test Seq. = The Euler Path from State $v_i$ to State $v_j$ @ $UIO_j$

else

augment the FSM minimally to get a new graph $G'$ which has a Euler Path beginning with the initial state
Test Seq. = The Euler Path of $G'$ @ $UIO_i$

where State $v_i$ is the tail state of the Euler Path
endif

The test sequence generated by Miller's Algorithm-2 may not begin with the initial state, while the one generated by the new-Algorithm-2 always begins with the initial
state. If there is more than one Euler Path beginning with the initial state, the new-
Algorithm-2 selects the one that results in the shortest test sequence, while Miller's
Algorithm-2 selects one randomly.

New Cost Function-1

A segment in the first approach differs from a segment in Miller's approach. In
Miller's approach, a segment is either a converging edge or a disjoint path with two
end states, while a segment in the first approach is either a converging edge or a non-
converging edge with two end states. The new cost Function-1 calculates the shortest
UIO sequence from a segment to another. It is used by the new-Algorithm-3. The
new cost function-1 is given as follows.

\[ i = \text{Tail}(Seg_k) \text{ and } j = \text{Head}(Seg_l) \text{ for some segments } Seg_k \text{ and } Seg_l. \]

\[ C_{i,j} = \text{length of the shortest UIO sequence of } i \text{ ending at } j \text{ for } i \neq j. \]

If \( Seg_l \) is not a converging edge then

\[ C_{i,i} = 0 \forall i. \]

else

\[ C_{i,i} = \text{length of the shortest UIO sequence of } i \text{ ending at } i \forall i. \]

endif

Basically, converging edges and non-converging edges are treated the same. The
only difference occurs when the tail state of a segment, called the first segment, is the
same as the head state of another segment, called the second segment. The cost from
the state to itself is zero if the second segment is not a converging edge. Otherwise,
it is determined by the shortest UIO sequence.
New-Algorithm-3

Similar to Miller's Algorithm-3, the new-Algorithm-3 applies the maximum cardinality minimum cost matching algorithm to a bipartite graph [22, 18] to generate a test sequence. The solution to this matching problem is a sequence of given segments and transfer paths connecting those segments. If a Euler Path (Tour) beginning with the initial state can be found then a valid test sequence is formed by appending the UIO sequence of the last state in the Euler Path (Tour) to the Euler Path (Tour). Otherwise, those disconnected components are connected by some heuristics similar to Chen's approach. In addition, the new-Algorithm-3 ensures that the maximum cardinality minimum cost matching algorithm begins with the initial state. The new-Algorithm-3 is given as follows.

1. Compute $UIO_{i,j} \forall v_i, v_j \in V$, where $UIO_{i,j}$ = the shortest UIO sequence from $v_i$ to $v_j$.

2. Let each edge (transition) be a segment.

3. Construct a bipartite graph $G_b = (V_b, E_b)$, where

   $V_b = V_s \cup V_e$

   $V_s = \{ i : i = Head(Seg_i) \text{ for some } j \}$

   $V_e = \{ i : i = Tail(Seg_i) \text{ for some } j \}$

   $E_b = \{ (i, j) : i \in V_e, j \in V_s; i \neq Tail(Seg_k), j \neq Head(Seg_k) \text{ for some } k \}$

   Assign costs $C_{i,j}$ to $(i, j) \in E_b$ based on the new cost Function-1.
4. Use the maximum cardinality minimum cost matching algorithm for graph $G_b$ and begin the process with the initial state.

A segment in Miller's Algorithm-3 is either a converging edge with two end states or a disjoint path with two end states, but a segment in the new-Algorithm-3 is either a converging edge with two end states or a non-converging edge with two end states. Since segments of non-converging edges replace segments of disjoint paths in our new-Algorithm-3, all possible sets of disjoint paths in a graph with all converging edges removed are implicitly considered. Therefore, a better solution can be found. In addition, unlike the new-Algorithm-3, a test sequence generated by Miller's Algorithm-3 may not necessary begin with the initial state, but a test sequence.

**New Cost Function-2**

A segment in the new-Algorithm-4 differs from a segment in the new-Algorithm-3. A segment in the new-Algorithm-3 is either a converging edge or a non-converging edge with two end states, while a segment in the new-Algorithm-4 can be (1) **TYPE 1**: a converging edge with two end states, (2) **TYPE 2**: a non-converging edge with two end states or (3) **TYPE 3**: the concatenation of an edge and an element in a signature set with two end states.

The purpose of the new cost Function-2 is to calculate a path (either the shortest UIO sequence or the shortest sequence) between two segments. If a path is from a segment of **TYPE 1** or **TYPE 2**, the shortest UIO sequence is calculated since a segment of **TYPE 1** or **TYPE 2** does not include the UIO sequence of the tail state of the segment. On the other hand, if a path is from a segment of **TYPE 3**, the shortest
sequence is calculated since a segment of TYPE 3 includes the UIO sequence of its tail state.

The new cost Function-2 used by the new-Algorithm-4 is given as follows.

\[ i = \text{Tail}(Seg_k) \text{ and } j = \text{Head}(Seg_l) \text{ for some segments } Seg_k \text{ and } Seg_l. \]

If Seg\(_k\) is either Type 1 or Type 2 then

Use the new cost Function-1 to calculate the cost.

else

if Seg\(_l\) is Type 3 and Seg\(_k\) is overlapped with the last part of Seg\(_k\) then

\[ C_{i,j} = -(\text{number of transitions overlapped}) \forall j \text{ and } i \neq j. \]

\[ C_{i,i} = 0 \forall i. \]

else

\[ C_{i,j} = \text{length of the shortest sequence of } i \text{ ending at } j \text{ for } i \neq j. \]

\[ C_{i,i} = 0 \forall i. \]

endif

endif

The only difference between the new cost Function-1 and the new cost Function-2 is that the latter can handle the case in which Seg\(_k\) is the concatenation of an edge and an element in a signature set.
New-Algorithm-4

The new-Algorithm-4 is designed to handle the case in which every state has UIO sequences, as well as the case in which some states do not have UIO sequences but have signature sets. Basically, the new-Algorithm-4 is similar to the new-Algorithm-3. The first difference is that the new-Algorithm-4 derives more segments than the new-Algorithm-3. When the tail state of an edge has a UIO sequence, only one segment is formed for the edge. On the other hand, when the tail state of an edge has a signature set, the number of segments formed for the edge is the same as the number of elements in the signature set. The second difference is that segments in the new-Algorithm-4 can be either TYPE 1, TYPE 2, or TYPE 3. After segments are derived and form a bipartite graph, the maximum cardinality minimum cost matching algorithm is then applied to the bipartite graph to generate a test sequence. The new-Algorithm-4 uses the new cost Function-2 to calculate a transfer path between segments. The new-Algorithm-4 is given as follows.

1. Compute $UIO_{ij} \forall v_i, v_j \in V$, where $UIO_{ij} =$ the shortest UIO sequence from $v_i$ to $v_j$.

2. Let each edge (transition) that enters a state with UIO sequences be a segment. For each edge that enters a state with a signature set, concatenate the edge and each element in the signature set to form a segment one at a time.

3. Construct a bipartite graph $G_b = (V_b, E_b)$, where $V_b = V_e \cup V_c$
$V_s = \{ i : i = \text{Head}(\text{Seg}_i) \text{ for some } j \}$

$V_e = \{ i : i = \text{Tail}(\text{Seg}_i) \text{ for some } j \}$

$E_b = \{ (i,j) : i \in V_e, j \in V_s; i \neq \text{Tail}(\text{Seg}_k), j \neq \text{Head}(\text{Seg}_k) \text{ for some } k \}$

Assign costs $C_{i,j}$ to $(i,j) \in E_b$ based on the new cost Function-2.

4. Use the maximum cardinality minimum cost matching algorithm for graph $G_b$
and begin the process with an initial state.

In Figure 18, State C has no UIO sequence. Thus, Miller's approach cannot be applied. Using the new-algorithm-4, the test sequence generated for Figure 18(a) is shown in Figure 18(c).

The new-Algorithm-2 and the new-Algorithm-3 are special cases of the new-Algorithm-4. The new cost Function-1 is a special case of the new cost Function-2. By simply using the new-Algorithm-4 and the new cost Function-2, both of the case in which each state has UIO sequences with and/or without converging edges and the case in which each state has either UIO sequences or a signature set is covered.

The reason for proposing three algorithms and two cost functions is to make the first approach look similar to Miller's approach, so that the differences between them can be easily observed.

5.3 Approach with Verifying the Uniqueness of UIO Sequences and Signature Sets

The second approach, presented in this section, is almost the same as the first. The only difference is that the second approach also considers verifying the uniqueness of
Figure 18: (a) An FSM with State C having no UIOS; (b) List of segments generated by the new-Algorithm-4; (c) The test sequence generated by the new Algorithm-4
UIO sequences and signature sets of states. Therefore, its fault coverage is improved over the first approach and Miller's approach. In summary, the second approach can generate shorter test sequences and has wider applicability and better fault coverage than Miller's approach.

The basic idea for achieving better fault coverage is to verify the uniqueness of UIO sequences and signature sets of states. Since multiple UIO sequences are implicitly used in this new approach, every UIO sequence used in a cost function should be verified for its uniqueness. The two cost functions proposed in the previous section will be used in this section. The new algorithm that ensures the uniqueness of UIO sequences of states in an FSM implementation is presented in this section. It also handles the case in which some states do not have UIO sequences but have signature sets.

**Condition for Applying the New Algorithm**

If each state has either a UIO sequence or a signature set then

perform new-Algorithm-5

endif

**New-Algorithm-5**

The new-Algorithm-5 is similar to the new-Algorithm-4, but achieves better fault coverage over the new-Algorithm-4 since it verifies the uniqueness of UIO sequences of states. The first difference is that the new-Algorithm-5 derives more segments...
than the new-Algorithm-4. Similar to the new-Algorithm-4, the new-Algorithm-5 can handle three different types of segments: TYPE 1, TYPE 2 and TYPE 3. In addition, the new-Algorithm-5 can handle a new type of segment, which is an input/output sequence with two end states. An input/output sequence is derived by feeding the input part of a UIO sequence of a state to another state. The use of the new type of segment ensures the uniqueness of UIO sequences and signature sets. The cost Function-2 is used to calculate a transfer path between segments. The new type of segment is handled in the same way as the segments of Type 3.

The new-Algorithm-5 records all new UIO sequences that are used and derives segments by applying those recorded UIO sequences to states. It repeats the process to derive segments until no new UIO sequences are recorded. The new-Algorithm-5 is given as follows.

1. Compute $UIO_{i,j} \forall v_i, v_j \in V$, where $UIO_{i,j}$ = the shortest UIO sequence from $v_i$ to $v_j$.

2. Let each edge (transition) that enters a state with UIO sequences be a segment. For each edge (transition) that enters a state with a signature set, concatenate the edge and each element in the signature set to form a segment one at a time.

3. Construct a bipartite graph $G_b = (V_b, E_b)$, where

   $V_b = V_s \cup V_e$

   $V_s = \{i : i = Head(Seg_i) \text{ for some } j\}$

   $V_e = \{i : i = Tail(Seg_i) \text{ for some } j\}$
\[ E_b = (i, j): i \in V_v, j \in V_s; i \neq Tail(Seg_k), j \neq Head(Seg_k) \text{ for some } k \]  

Assign costs \( C_{i,j} \) to \((i, j) \in E_b\) based on the new cost Function-2.

4. Record each new UIO sequence that are used in the bipartite graph.

5. Apply a UIO sequence of a state recorded in Step 4 to other states, and treat it as a new segment. Add it to the bipartite graph if it is not contained in other segments. Assign costs from other segments to it, and from it to other segments based on the new cost Function-2.

6. Repeat Steps 4 to 6 until no new UIO sequences are recorded.

7. Use the maximum cardinality minimum cost matching algorithm for graph \( G_b \) and begin the process with an initial state.

Figure 17 is used as an example to demonstrate that the new-Algorithm-5 can outperforms Miller's approach and the first approach. The test sequence generated by Miller's approach as well as the first approach is a/1.b/1.a/2.b/2.c/1.c/2.a/2. This sequence cannot distinguish the difference between Figure 17(a) and Figure 17(b). In contrast, the test sequence generated by our new-Algorithm-5 is c/-a/1.a/-b/1.b/-c/-a/2.b/2.a/-b/-c/1.c/2.a/2. The sequence, c/-a/1.a/-b/1.b/-c/-a/2.b/2.a/-b/-c/1.c/2.a/2, can distinguish Figure 17(a) from Figure 17(b). As shown in Figure 19, many extra segments are derived by the new-Algorithm-5 to verify the uniqueness of states' UIO sequences and signature sets, and the new test sequence generated is six units longer.
Test Sequence generated: 1→1→3→3→4→4→1→2→2→2→3→4→1
or represented as c/-.a/1.a/-b/1.b/-c/-a/2.b/2.a/-b/-c/1.c/2.a/2

Figure 19: (a) Segments derived for verifying the uniqueness of UIO sequences and signature sets of states; (b) Test sequence generated by the second approach
5.4 Comparison with Other Approaches

In this section a comparison of the two new approaches with three others is made based on the following five criteria: (1) whether multiple UIO sequences and/or segment overlap are used, (2) whether multiple UIO sequence and/or segment overlap are used explicitly or implicitly, (3) whether the methods can handle the case in which each state has UIO sequences and/or the case in which some states have signature sets but no UIOS, (4) whether optimal solutions are guaranteed, and (5) whether verifying the uniqueness of UIO sequences and signature sets can be handled. Among these criteria, the third addresses applicability, and the fifth, fault coverage. Usually a method verifying the uniqueness of UIO sequences and signature sets has better fault coverage than the method without.

Multiple UIO Sequences and Segment Overlap

Many different approaches for shortening test sequences have been proposed so far. It has been noted that multiple UIO sequences and segment overlap are the two main factors that affect the length of a test sequence. Shen’s approach in [54] uses only multiple UIO sequences while Chen’s approach in [11] and Miller’s approach in [38] use multiple UIO sequences and segment overlap. In Chen’s approach, although multiple UIO sequences and segment overlap are used to shorten test sequences, they are used in a heuristic algorithm. In contrast, Miller’s approach and our two approaches use both multiple UIO sequences and segment overlap in non-heuristic algorithms.
Optimal Solutions

Miller’s approach successfully illustrates that it can find better solutions than others that use only either multiple UIO sequences or segment overlap. Both Miller’s and Chen’s approaches use multiple UIO sequences and segment overlap, but Miller’s approach excels since Chen’s approach uses only a greedy heuristic to shorten test sequences. Since our two approaches do not have the errors and the drawbacks of Miller’s approach but have its advantages, ours is superior. When a Euler Path (Tour) can be found to link all segments, optimal solutions are guaranteed by Miller’s approach as well as by our two approaches, but optimal solutions are not guaranteed by the other two methods.

Explicitly v.s. Implicitly

Both Shen’s and Chen’s approaches use either multiple UIO sequences or segment overlap explicitly by introducing extra links between segments. In contrast, Miller’s approach uses both multiple UIO sequences and segment overlap implicitly in the cost function. Instead of finding the shortest sequence between two states, Miller’s approach finds the shortest UIO sequence between two states. The shortest UIO sequence between State A and State B is equivalent to a UIO sequence of State A plus a transfer sequence from the tail state of the UIO sequence to State B. Then, choosing which UIO sequence of State A depends on which overlaps the most with the next segment. Therefore, multiple UIO sequences and segment overlap are fully used. Our two approaches use both the multiple UIO sequences and segment overlap implicitly similar to Miller’s approach to find shortest test sequences.
Applicability

Miller's approach can only handle the case in which every state has a UIO sequence. Based on our observation, Chen's approach can also handle states that have either UIO sequences or signature sets. Our two approaches are designed to handle not only the case in which every state has a UIO sequence but also the case in which some states have signature sets but no UIO sequences. In summary, only Chen's approach and our two approaches have the same applicability, outperforming both Miller's approach and Shen's approach.

Fault Coverage

Better fault coverage can be achieved by verifying the uniqueness of UIO sequences and signature sets of states. Based on our observation, Chen's approach can verify the uniqueness of UIO sequences and signature sets by adding more segments. Our second approach is designed to verify the uniqueness of UIO sequences and signature sets of states. In summary, only Chen's approach and our second approach have the same fault coverage, which is better than our first approach, Miller's approach and Shen's approach.

A summary of the comparison made among Shen's approach, Chen's, Miller's and our two approaches is given in Table 27. An interesting observation was obtained recently about the impact of segment overlap on the fault coverage. We found that although using segment overlap can shorten the lengths of test sequences, the shortened test sequences may not have the same fault coverage. For example, the four segments derived from Figure 20(a) are listed in Table 28. Both Aho's and Shen's
Table 27: A summary of comparisons made among five approaches

<table>
<thead>
<tr>
<th></th>
<th>Shen's</th>
<th>Chen's</th>
<th>Miller's</th>
<th>Our-1</th>
<th>Our-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple UIOS</td>
<td>yes</td>
<td>yes/heuristic</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>overlapping</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>explicity/implicity</td>
<td>exp.</td>
<td>exp.</td>
<td>imp.</td>
<td>imp.</td>
<td>imp.</td>
</tr>
<tr>
<td>each state has UIOS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>states have signature sets</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>optimal solution</td>
<td>no</td>
<td>no</td>
<td>some cases</td>
<td>some cases</td>
<td>some cases</td>
</tr>
<tr>
<td>verification of uniqueness</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Table 28: Four segments

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>String of Input/Output Sequence and State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>&lt;1&gt; a/1 &lt;2&gt; b/1 &lt;3&gt;</td>
</tr>
<tr>
<td>S2</td>
<td>&lt;1&gt; d/1 &lt;2&gt; b/1 &lt;3&gt;</td>
</tr>
<tr>
<td>S3</td>
<td>&lt;2&gt; b/1 &lt;3&gt; c/1 &lt;1&gt;</td>
</tr>
<tr>
<td>S4</td>
<td>&lt;3&gt; c/1 &lt;1&gt; d/1 &lt;2&gt;</td>
</tr>
</tbody>
</table>

approaches do not use the segment overlap technique, the test sequence derived by them is a/1.b/1.d/1.b/1.b/1.c/1.c/1.a/1. In contrast, Miller’s, Chen’s and our two new approaches proposed in this chapter use the segment overlap technique, the test sequence derived by those methods is a/1.b/1.c/1.d/1.b/1. Although the test sequence derived by Aho’s and Shen’s approaches without using the segment overlap technique is longer, it can detect the faulty FSM as shown in Figure 20(b). However, the test sequence derived by Miller’s, Chen’s, and our approaches is shorter but cannot detect the faulty FSM as shown in Figure 20(b).

5.5 Conclusions

The MLTG method is proposed and coded using C and the detailed program is listed in Appendix C. Appendix F lists the sample output generated by the program implementing the MLTG method using Figure 10(a) as the input protocol specification. Multiple UIO sequences and segment overlap are used implicitly to find shortest test sequences. All the errors and drawbacks of Miller’s approach are corrected. The MLTG method consists of two approaches. Both can handle the protocol specifi-
Figure 20: (a) An FSM; (b) A faulty FSM
ocations in which some states do not have UIOSs while Miller's approach cannot. The second approach improves the fault coverage of our first approach by verifying the uniqueness of UIO sequences and signature sets of states. Among test sequence generation methods with length optimization, our two approaches have the best applicability, and our second approach has the best fault coverage. In addition, they guarantee that under some conditions optimal solutions can be found.
As communication networks expand, the need of providing automated conformance testing grows. Therefore, three automatic test sequence generation methods have been proposed. In the following sections, important properties of each proposed method are reviewed and future research issues are discussed.

6.1 Summary

In the three test sequence generation methods proposed in this dissertation, the first two, the UIOpv and the TSP methods, concentrate on deriving segments for each state and transition without worrying about the length of the final test sequence, and the last method, the MLTG method, deals with shortening the overall length of the test sequence. The UIOpv method uses an efficient algorithm to derive a UIOS or a unique signature set for each state. It also derives a partial verification part to verify the uniqueness of UIOSs and signature sets for achieving better fault coverage. As a result, the UIOpv method generates a shorter test sequence than the UIOv method and uses less time.

Unlike the W method, which derives a single characteristic set for all states in an FSM, and the Wp method, which derives a subset of the characteristic set for each
state, the TSP method derives a unique set for each state. Therefore, the number of sequences that are contained in a set is smaller in the TSP method than in the W and the Wp methods. As a result, the TSP method generates fewer segments and thus reduces the time for generating a test sequence. Although the UIOpv and the TSP methods serve the same purpose, they have different strength. The TSP method needs less time for deriving a test sequence; on the other hand, the UIOpv method generates shorter test sequences.

The second part of this dissertation (Chapter 5) proposes the MLTG method, which derives UIOSs and unique signature sets for states, generates segments for each state and each transition, and then connects segments using multiple UIOSs and segment overlap to shorten test sequences. The MLTG has better applicability and better fault coverage than the Aho’s, Shen’s, Chen’s and Miller’s approaches.

6.2 Future Research

Many companies have deployed fiber optics networks to provide high speed data, voice, image, and video communications, and many others will follow. The integrated data, voice, image, and video service and policing quality of service make protocols complicated because many new features such as timing, guaranteed bandwidth, guaranteed delay, forward congestion control, backward congestion control, and discard eligibility need to be addressed in new protocols. Existing high speed protocols that provide quality of service include Frame Relay, Switched Multimega Bit Data Service (SMDS), Fiber Distributed Data Interface (FDDI), Asynchronous Transfer Mode (ATM) Layer, ATM Adaptation Laye(AAL) protocols.
It is not easy to model all the features of quality of service by using current models. Thus, one important research issue is to design new models that can describe quality of service for high speed protocols. The new models should be able to describe timing, delay, congestion, and discard eligibility requirements. Since no existing methods can generate test sequences for these new models, the next research issue is to develop new methods for the new models. Although timed and/or probability models exist, no methods can generate test sequences for them. Therefore, another research issue is to develop test sequence generation methods for protocols specified in timed models, probability models, and both.

Another research issue is to design test sequence generation methods for the non-FSM FDTs such as Extended Finite State Machine, Estelle, LOTOS, and SDL. The flexibility and expressive power of these sophisticated models make complicated protocols easier to specify but also make test sequence generation harder to implement. Although there are methods generating test sequences for these models, they use restricted models and only provide partial solutions such as heuristic algorithms, formal frameworks, semi-automated algorithms, or ad hoc solutions. The new methods should use the more complete features of the models, achieve better fault coverage, provide more general solutions, and hopefully be fully-automated. Length minimization for non-FSM FDTs is another important issue that has not been explored so far.

In summary, test sequence generation for communication protocols is an important yet not fully understood area. The reliability of a communication network depends on
a correct implementation of a protocol, but only a handful of automatic test sequence
generation methods are available. This dissertation contributes three automatic test
sequence generation methods to the state of art and identifies important issues that
need to be resolved in the future research.
Appendix A

Program for the UIOpv Method

```c
#include <stdio.h>
#include <string.h>

int k; /* number of outgoing transitions */
char Edge[100][2]; /* store edges no duplicate */
int Cnt[100];
    /* store the number of duplicates for each edge */
int n_edge;
    /* store the total non-duplicate edges */
int sameE[20][5];
    /* store the states having the same edge */
int n_sameE[20]; /* store the number of states */

int GROUP[20]; /* value=1 or 2; divide into 2 groups */
int F[20];
    /* Flag value = 0: initial; */
/* i 1: Case 1 of Algorithm 1; 2:Case 2 */
```
int N1[20], N2[20];
/* store the number of elements in USOTSP and USIOS */
int TRANS[20]; /* store the number of trans for states */
char I[20][20]; /* input symbols */
char O[20][20]; /* output symbols */
char H[20][20]; /* head states */
char T[20][20]; /* Tail states */
char USOTSP[20][20][20]; /* elements in USOTSP */
char USIOS[20][20][20]; /* elements in USIOS */
int n_found, n_state, n_tran;
int id, n_usotsp;
char s[50], str[50];
char state[20];
int find;
int entry;
char name;
int FOUND[50];
char SAME[50];

main()
{
    Init();
input();

cal_Set(); /* step 1 */

Find_UI0();

Concat_UI0();

Signature();

Output();

}

/******************************/

/* Initialization */

/******************************/

Init()
{

}

/******************************/

/* Determine the set for each edge */

/******************************/

cal_Set();
{ int i, n, x, n, c;

for (i=0; i<=n_state; i++) n_sameE[i] = -1;
n=TRANS[0];
for (x=0; x<=n-1; x++)
{
    Edge[x][0]=I[0][x];
    Edge[x][1]=O[0][x];
    Cnt[x]=0;
    sameE[x][0]=0;
    n_sameE[x]=0;
}
}

n_edge=n-1;
for (i=1; i<= n_state; i++)
{
    n=TRANS[i];
    for (x=0; x<=n; x++)
    {
        c=exist(i,x);
        if (c==0)
        {
            
            
        }
n_edge++;
add(n_edge, i, x);
Cnt[n_edge]=0;
n_sameE[n_edge]=0;
sameE[n_edge][0]=i;
}
else /* existing */
{
    Cnt[c]++;
n_sameE[c]++;
sameE[c][n_sameE[c]]=i;
}
/* check if an edge exists */
exist(i, x)
int i, x;
{
    int m, find;

    m=0;
    find=0;
    while ((m<n_edge) && (find==0))
    {
        if ((Edge[m][0]==I[i][x]) && (Edge[m][1]==O[i][x]))
            find=1;
        else m++;
    }
    if (find==0) return(0);
    else return(m);
}

/**********************************************************/
/* add an edge                                          */
/**********************************************************/

add(n, i, x)
int n, i, x;
{

    Edge[n][0]=I[i][x];
    Edge[n][1]=O[i][x];
}

.Repositories

/*****************************/
//  Find UI0S of length 1   */
/*****************************/

Find_UI00()
{
    int i, m, n;

    for (i=0; i<=n_state; i++)
    {
        N2[i]= -1; F[i]=0;
    }

    for (m=0; m<=n_edge; m++)
    {
        i=sameE[m][0];
        if ((Cnt[m]==0) && (F[i]==0))
        {
        }
F[i]=1;
save(m,i,0);
N2[i]=0;
}
}

/***************************************************************************/
/* save a UIOS into USIOS */
/***************************************************************************/

save(m,i,x)
int m,i,x;
{
    USIOS[i][x][0]=Edge[m][0];
    USIOS[i][x][1]='/';
    USIOS[i][x][2]=Edge[m][1];
    USIOS[i][x][3]='\000';
}

/***************************************************************************/
/* Find UIOS of length > 1 */
/***************************************************************************/
Concat.UIO()
{
    int n_found, y, i, m, j, c, n;
    int new, min, min1, min2, min3, count;

    n_found= -1;
    for (i=0; i<=n_state; i++)
    {
        if (F[i]==1)
        {
            n_found++;
            FOUND[n_found]=i;
        }
    }
    /***** find better UIOS ******/
    new=0;
    while((n_state>n_found)&&(new==0))
    {
        new=1;
        j=0;
    }
while(j<=n_state)
{
    if (F[j]==0)
    {
        min=100;
        min1= -1;
        min2= -1;
        n=TRANS[j];
        for (y=0; y<n; y++)
        {
            for (m=0; m<=n_found; m++)
            {
                i=FOUND[m];
                if (T[j][y]==state[i])
                {
                    c=check_tsp(j,y);
                    if (c==0)
                    {
                        count=find_cnt(j, y);
                        if (count < min)
                        {
                            min=count;
                            mini=j;
                        }
                    }
                }
            }
        }
    }
}
min2=y;
min3=i;
}
}
}
}

if (min != 100)
{
new=0;
n_found++;
FOUND[n_found]=j;
F[min1]=2;
N2[min1]=0;
USIOS[min1][0][0]=I[min1][min2];
USIOS[min1][0][1]='/';
USIOS[min1][0][2]=0[min1][min2];
USIOS[min1][0][3]='.';
USIOS[min1][0][4]='\000';
strcat(USIOS[min1][0],USIOS[min3][0]);
}
}
```c
j++;  
}
}

/*****************************/
/* find out the number of duplicates */
/*****************************/
find_cnt(i, x)  
int i, x;  
{
    int m, c;  

    m=0;  
    while ((c==0) && (m<n_edge))  
    {
        if (((Edge[m][0]==i[x]) && (Edge[m][1]==0[i][x]))  
            c=1;  
        else m++;  
    }
    return(Cnt[m]);  
}
```
/************************•**••********/
/* Find signature set for states   */
/************************•**••********/

Signature()
{
    int i, j, n;
    char str[50];

    for (i=0; i<=n_state; i++)
    {
        if (F[i]==0)
        {
            F[i]=3;
            N2[i]= -1;
            for (j=0; j<=n_state; j++)
            {
                if (i!=j)
                {
                    N2[i]++;
                }
find_IO(i, j, str);
strcpy(USIOS[i][N2[i]], str);

}
if (c==1) /* j contains k */
    USIOS[i][k][0]='#';
else if (c==2) /* k contains j */
    USIOS[i][j][0]='#';

for(i=0; i<n_state; i++)
{
    j = 0;
    printf("\n");
    printf("---------------------------------------------------------\n");
    printf("*** state %c has the following set ***\n", state[i]);
    n=N2[i];
    for (k=0; k<n; k++)
    {
        if (USIOS[i][k][0]!='$')
        {
            j++;
            printf(">>> element %d: %s\n", j, USIOS[i][k]);
        }
    
}
I check if a string is contained in another.

```
contain(i, j, k)
int i, j, k;
{
    int m, n;

    m=0;
    while ((USIOS[i][j][m]!='$0000') && (USIOS[i][k][m]!='$0000') &&
        (USIOS[i][j][m]==USIOS[i][k][m])) m++;
    if (USIOS[i][j][m]='$0000') return(2);
    else if (USIOS[i][k][m]='$0000') return(1);
    else return(0); /* no contain relationship */
}
```
/* find a sequence less than or equal to n */
/* ...........................................................................*/

find_IO(i,j,str)

int i, j;
char str[50];
{
    int x, n, y;

    str[0] = '\000';
    Ass_Group(i);
    n = -1;
    while(((GROUP[i]==GROUP[j])&&(n<=n_state))
    {
        n++;
        x=S_Group(i,j,&y);
        s[0]=I[i][x];
        s[1]='/';
        s[2]=O[i][x];
        s[3]='.';
        s[4]= '\000';
        strcat(str,s);
        i=state_id(T[i][x]);
j=state_id(T[j][y]);
Ass_Group(i);
}
x=D_Group(i,j);
s[0]=I[i][x];
s[1]='/';
s[2]=O[i][x];
s[3]='.';
s[4]='\000';
strcat(str,s);
}

/************************************
/* find the index of a state name */
/* in an array */
/************************************
state_id(ch)
char ch;
{
    int i, find;
i=0;
find=0;
while((find==0)&&(i<n_state))
{
    if (state[i]==ch) find=1;
    else i++;
}
return(i);

/**
 * Divided into 2 groups according to i
 */

Ass_Group(i)
int i;
{
    int j, c;

    GROUP[i]=1;
    for (j=0; j<n_state; j++)
    {
}
if (i!=j)
{
    c=same(i,j);
    if (c==0) GROUP[j]=1;
    else GROUP[j]=2;
}

same(i, j)
int i, j;
{
    int c, x, y, find, n, m;
    int small, big;

    c=0;
    x=0;
    if (TRANS[i] <= TRANS[j])
{ small=i; big=j; }
else { small=j; big=i; }
n=TRANS[small];
while ((c==0) &amp; (x&lt;=n-1))
{
    y=0;
    find= 0;
    m=TRANS[big];
    while (y&lt;=m-1)
    {
        if((I[small][x]==I[big][y]) &amp;&amp; (O[small][x]==O[big][y]))
        {
            find++;
            y=m+10;
        }
    }
    else
    {
        y++;
    }
    if (y==m)
        c++; /* not in the same group */
    else x++;
}
return(c);
}

/*------------------------------------------------------------------------------------------
/* find tsp for the same group          
/*------------------------------------------------------------------------------------------

S_Group(i, j, p)
int i, j, *p;
{
    int c, x, n, m, find, y;
    int small, big;

c=0;
x=0;
if (TRANS[i]<TRANS[j]) {small=i; big=j;}
else {small=j; big=i;}
n=TRANS[small];
while ((c==0) && (x<n-1))
{
    y=0;
    find=0;
m=TRANS[big];
while (((find==0) && (y<=m-1))
{
    if((I[i][x]==I[j][y]) && (O[i][x]==O[j][y]) &&
        (T[i][x]==T[j][y])) find++;
    else y++;
}
if (find !=0) c++;
else x++;
}
*p=y;
return(x);
}

/****************************************************************************
/* find transitions for different group    */
/****************************************************************************

D_Group(i, j)
int i, j;
{
    int find, c, x, y, n, m;
c=0;

x=0;
n=TRANS[i];

while ((c==0) && (x<=n-1))
{
    y=0;
    find=0;
    m=TRANS[j];
    while ((find==0) && (y<=m-1))
    {
        if((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])) find++;
        else y++;
    }
    if (find ==0) c++;
    else x++;
}

return(x);

/***********************************/

/* put info. into arrays */
input()
{
    char dummy;
    int n_tran, i;

    printf("If there is no more state, please type *
        
    printf("Please type the state name
           
   scanf("\%c", &name);
   n_state= -1;
   while (name!='*')
   {
       n_state++;
       state[n_state]=name;
       n_tran= -1;
       printf("Please type the number of transitions
             
             scanf("\%d", &n_tran);
             TRANS[n_state]=n_tran;
             scanf("\%c", &dummy);
for (i=0; i<n_tran; i++)
{
    printf("Please type the input symbol, output symbol, 
    and the tail state (char, char, char): ");
    scanf("%c", &I[n_state][i]);
    scanf("%c", &O[n_state][i]);
    scanf("%c", &T[n_state][i]);
    scanf("%c", &dummy);
}
name='#';
printf("If there is no more state, please type *.
");
printf("Please type the state name
    (one character only):");
scanf("%c", &name);
}

/*****************************/
/* decide if a given transition is unique*/
/*****************************/

check_t(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;
    while ((c==0) && (j<n_state))
    {
        if (i!=j)
        {
            y=0;
            n_tran=TRANS[j];
            while ((c==0) && (y<=n_tran-1))
            {
                if ((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])) c++;
                y++;
            }
        }
    }
    j++;
}

if (c==0) return(0);
else return(1);
/* decide if a given tsp is unique*/

check_tsp(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;
    while ((c==0) && (j<=n_state))
    {
        if (i!=j)
        {
            y=0;
            n_tran=TRANS[j];
            while (((c==0) && (y<n_trans-1))
            {
                if ((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])

}
\&\& (T[i][x]==T[j][y]) c++;

else y++;

j++;

if (c==0) return(0);
else
{
    return(1);
}
}
Appendix B

Program for the TSP Method

#include <stdio.h>
#include <string.h>

int GROUP[20]; /* value=1 or 2; divide into 2 groups */
int F[20];
    /* Flag value = 0: initial; */
    /* 1: Case 1 of Algorithm 1; 2:Case 2 */
int N1[20], N2[20];
    /* store the number of elements in USOTSP and USIOS */
int TRANS[20]; /* store the number of trans for states */
char I[20][20]; /* input symbols */
char O[20][20]; /* output symbols */
char H[20][20]; /* head states */
char T[20][20]; /* Tail states */
char USOTSP[20][20][20]; /* elements in USOTSP */
char USIOS[20][20][20]; /* elements in USIOS */
int n_found, n_state, n_tran;
int id, n_usotsp;
char s[50], str[50];
char state[20];
int find;
int entry;
char name;
int FOUND[50];
char SAME[50];

main()
{
    int i;

    Init();
    input();
    n_found= -1;
    Algo1_Case1();
    Algo1_Case2();
    Algo1_Case3();
    Algo2();
    Output();
}
/*******************************
/* print out result           */
/*******************************

Output()
{
    int c, i, j, k, n;

    for(i=0; i<=n_state; i++)
    {
        n=N2[i];
        for (j=0; j<=n; j++)
            for (k=j+1; k<=n; k++)
            {
                c=contain(i, j, k);
                if (c==1) /* j contains k */
                    USIOS[i][k][0]=#';
                else if (c==2) /* k contains j */
                    USIOS[i][j][0]=#';
            }
    }
}
for(i=0; i<=n_state; i++)
{
    j = 0;
    printf("\n");
    printf("------------------------------------------\n");
    printf("*** state %c has the following set ***\n", state[i]);
    n=N2[i];
    for (k=0; k<=n; k++)
    {
        if (USIOS[i][k][0]!="#")
        {
            j++;
            printf(">>> element %d: %s\n", j, USIOS[i][k]);
        }
    }
}

/*******************************************************************************/

/* check if a string is contained */
/* in another */
/************************************/
contain(i, j, k)
int i, j, k;
{
    int m, n;

    m=0;
    while ((USIOS[i][j][m]!=\'000\') && (USIOS[i][k][m]!=\'000\') &&
        (USIOS[i][j][m]==USIOS[i][k][m])) m++;
    if (USIOS[i][j][m]==\'000\') return(2);
    else if (USIOS[i][k][m]==\'000\') return(1);
    else return(0); /* no contain relationship */
}

/************************************/
/* Algorithm 2 */
/************************************/
Algo2()
{
    int i, j, k, kk, mm, m;
    int n_same;
int id, id2, n_usotsp;

for (i=0; i<n_state; i++)
{
    if (F[i]==3)
    {
        N2[i]=N1[i];
        n_usotsp=N1[i];
        kk=0;
        for (k=0; k<n_usotsp; k++)
        {
            if (USOTSP[i][k][0]!='*')
            {
                strcpy(USIDS[i][kk], USOTSP[i][k]);
                kk++;
            }
            else
            {
                id=state_id(USOTSP[i][k][5]);
                if (F[id]!=3)
                {
                    for (m=0; m<=3; m++)
                }
```c
s[m] = USOTSP[i][k][m+1];
s[4] = '\000';
strcat(s, USIOS[i][0]);
strcpy(USIOS[i][kk], s);
kk++;
}
else /* id doesn't have a unique sequence */
{
    id2 = find_tail(i, k);
    find_EOL(id, id2, str);
    for (m = 0; m <= 3; m++)
        s[m] = USOTSP[i][k][m+1];
    s[4] = '\000';
    strcat(s, str);
    strcpy(USIOS[i][kk], s);
    kk++;
}
}
}
N2[i] = kk - 1;
```
/* find out the states having the same */
/* transition but different tail state */
/**********************/
find_tail(i, x)
int i, x;
{
    int j, y, find, id2, n_tran;

    if (x>i) j=x+1;
    else j=x;
    find=0;
    n_tran=TRANS[j];
    y=0;
    while ((find==0) && (y<=n_tran-1))
    {
        if (((USOTSP[i][x][1]==I[j][y]) && (USOTSP[i][x][3]
            ==O[j][y]) && (USOTSP[i][x][5]!=T[j][y])))
            find=1;
        else y++;
    }
id2=state_id(T[j][y]);
return(id2);
}

/**********************************************************************************/
/* find a sequence less than or equal to n     */
/**********************************************************************************/
find_IO(i,j,str)
int i, j;
char str[50];
{
  int x, n, y;

  str[0] = '\000';
  Ass_Group(i);
  n = -1;
  while((GROUP[i]==GROUP[j]) && (n<=n_state))
  {
    n++;
    x = S_Group(i,j,&y);
  }
s[0]=I[i][x];
s[1]='/';
s[2]=O[i][x];
s[3]='.';
s[4]='$000$';
strcat(str,s);
i=state_id(T[i][x]);
j=state_id(T[j][y]);
Ass_Group(i);
}
x=D_Group(i,j);
s[0]=I[i][x];
s[1]='/';
s[2]=O[i][x];
s[3]='.';
s[4]='$000$';
strcat(str,s);
}

/*****************************/
/
/* find the index of a state name */
/* in an array */

state_id(ch)
char ch;
{
    int i, find;

    i=0;
    find=0;
    while((find==0)&&(i<n_state))
    {
        if (state[i]==ch) find=1;
        else i++;
    }
    return(i);
}

/* Initialize variables */

Init();
{

/* Case 1 in Algorithm 1 */
/* Find unique transitions for states */

Algo1_Case1()
{
    int i, c;

    for (i=0; i<n_state; i++)
    {
        c=Uni_T(i);
        if (c==0)
        {
            n_found++;
            FOUND[n_found]=i;
            F[i]=1;
            N1[i]=0;
            N2[i]=0;
            strcpy(USIOS[i][0],USDTSP[i][0]);
        }
    }
}
/*******************************
/* Case 2 in Algorithm 1       */
/* Find unique taps for states */
/* and its tail state's USOTSP has been found */
/*******************************/

Alg01_Case2()
{
    char st[50];
    int i, j, k, x, y, c, m;

    m=0;
    while(m<=n_found)
    {
        i=FOUND[m];
        for (j=0; j<=n_state; j++)
        {

if ((i!=j) && (F[j]==0))
{
    c=0;
    y=0;
    n_tran=TRANS[j];
    while((c==0) && (y<n_tran-1))
    {
        if (T[j][y]==state[i])
        {
            c=check_tsp(j,y);
            if (c==0)
            {
                n_found++;
                FOUND[n_found]=j;
                F[j]=2;
                strcpy(st,USOTSP[i][0]);
                save_t(j,y,st,0);
                N1[j]=0;
                N2[j]=0;
                strcpy(USIOS[j][0],USOTSP[j][0]);
            }
        }
    }
}
/* Case 3 in Algorithm 1 */
/* Find unique sets for states */

Algo1_Case3()
{
    int i;

    if (n_found < n_state)
    {
        for (i=0; i<n_state; i++)
        {
            
```
if(F[i]==0)
{
    Ass_Group(i);
    F_SEQ(i);
    F[i]=3;
}
}
}

/****************************
/ * Divided into 2 groups according to i  */
/****************************

Ass_Group(i)
int i;
{
    int j, c;

    GROUP[i]=1;
    for (j=0; j<=n_state; j++)
    {
    }
if (i!=j)
{
    c=same(i,j);
    if (c==0) GROUP[j]=1;
    else GROUP[j]=2;
}

/* check if two states having the same transition set */
/* transition set */

same(i, j)
int i, j;
{
    int c, x, y, find, n, m;
    int small, big;

    c=0;
    x=0;
    if (TRANS[i] <= TRANS[j])
{ small=i; big=j; }
else { small=j; big=i; }
n=TRANS[small];
while ((c==0) && (x<=n-1))
{
    y=0;
    find= 0;
    m=TRANS[big];
    while (y<=m-1)
    {
        if ((I[small][x]==I[big][y]) && (O[small][x]==O[big][y]))
        {
            find++;  
            y=m+10;
        }
    else
    
        y++;
    }
    if (y==m)
        c++; /* not in the same group */
    else x++;
}
return(c);
}

/* find sequences to distinguish states */

F_SEQ(i)
int i;
{
    int y, n_usotsp, j, x;

    n_usotsp= -1;
    for (j=0; j<=n_state; j++)
    {
        if ((GROUP[i]==GROUP[j]) && (i!=j))
        {
            x=S_Group(i,j,&y);
            n_usotsp++;
            save_tsp(i,x, n_usotsp);  
        }  
    }  
}
} 
else if (i!=j) 
{ 
    x=D_Group(i,j);
    n_usotsp++;
    save_t(i,x, "",n_usotsp);
}
}
N1[i]=n_usotsp;
{

/*****************************/
/* find tsp's for the same group */
/*****************************/

S_Group(i, j, p)
int i, j, *p;
{
    int c, x, n, m, find, y;
    int small, big;
c=0;
x=0;
if (TRANS[i]<=TRANS[j]) {small=i; big=j;}
else {small=j; big=i;}
n=TRANS[small];
while ((c<=0) && (x<=n-1))
{
    y=0;
    find=0;
    m=TRANS[big];
    while ((find<=0) && (y<=m-1))
    {
        if((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])
            && (T[i][x]!=T[j][y])) find++;
        else y++;
    }
    if (find !=0) c++;
    else x++;
}
*p=y;
return(x);
/*********************************************/
/* find transitions for different group */
/*********************************************/

D_Group(i, j)
int i, j;
{
    int find, c, x, y, n, m;

    c=0;
    x=0;
    n=TRANS[i];
    while ((c==0) & (x<=n-1))
    {
        y=0;
        find=0;
        m=TRANS[j];
        while ((find==0) & (y<=m-1))
        {
            if((I[i][x]==I[j][y]) & (O[i][x]==O[j][y])) find++;
            else y++;
        }
if (find == 0) c++; else x++; }
return(x);
}

/*****************************/
/* put info. into arrays */
/*****************************/

input()
{
    char dummy;
    int n_tran, i;

    printf("If there is no more state, please type *\n");
    printf("Please type the state name
    (one character only): ");
    scanf("%c", &name);
    n_state= -1;
    while (name!='*')
n_state++;
state[n_state]=name;
n_tran=-1;
printf("Please type the number of transitions
STATE \%c has (integer):", name);
scanf("%d", &n_tran);
TRANS[n_state]=n_tran;
scanf("%c", &dummy);
for (i=0; i<n_tran; i++)
{
    printf("Please type the input symbol, output
symbol, and the tail state (char, char, char):");
    scanf("%c", &I[n_state][i]);
    scanf("%c", &O[n_state][i]);
    scanf("%c", &T[n_state][i]);
    scanf("%c", &dummy);
}
name='#';
printf("If there is no more state, please type *.
\n");
printf("Please type the state name
(\one character only):");
    scanf("%c", &name);
    }
}

/****************************/
/* decide if a given transition is unique*/
/****************************/

check_t(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;
    while ((c==0) && (j<n_state))
    {
        if (i!=j)
        {
            y=0;
            n_trant=TRANS[j];
            while ((c==0) && (y<n_tran-1))
            {...}
check_tsp(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;

    if ((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])) c++;
    y++;
}

if (c==0) return(0);
else return(1);
}
while ((c==0) && (j<=n_state))
{
  if (i!=j)
  {
    y=0;
    n_tran=TRANS[j];
    while ((c==0) && (y<=n_tran-1))
    {
      if ((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])
          && (T[i][x]==T[j][y])) c++;
      else y++;
    }
  }
  j++;}
if (c==0) return(0);
else
{
  return(1);
}
}
/************

/* save info. into USOTSP */

save_t(i, x, str, entry)
int i, x, entry;
char *str;
{
    char s[20];

    s[0]=I[i][x];
    s[1]='/';
    s[2]=0[i][x];
    s[3]='.';
    s[4]=\'000\';
    strcat(s,str);
    strcpy(USOTSP[i][entry],s);
}

save_tsp(j, y, entry)
int j, y, entry;
{
    char s[20];

    s[0]='*';
    s[1]=I[j][y];
    s[2]='/';
    s[3]=0[j][y];
    s[4]='.';
    s[5]=T[j][y];
    s[6]='\000';
    strcpy(USOTSP[j][entry], s);
}

/****************************/
/* find a unique transition for a state  */
/****************************/

Uni_T(i)
int i;
{

int c, x, n_tran;

c = -1;
x = 0;
n_tran = TRANS[i];
while ((c != 0) && (x <= n_tran - 1))
{
    c = check_t(i, x);
    if (c == 0)
    {
        N1[i]++;
        save_t(i, x, "", 0);
    }
    else x++;
}
return(c);
}

Thông tin

/=====================================================================
/* check if a unique tsp exists 
/=====================================================================

Uni_TSP(i)


```c
int i;
{
    int c, x, n_tran;

    c = -1;
    x = 0;
    n_tran = TRANS[i];
    while ((c != 0) && (x <= n_tran - 1))
    {
        c = check_tsp(i, x);
    }
    return(c);
}
```
Appendix C

Program for the MLTG Method

#include <stdio.h>
#include <string.h>

char Edge[100][2]; /* store edges no duplicate */
int Cnt[100];
/* store the number of duplicates for each edge */
int n_edge; /* store the total non-duplicate edges */
int sameE[20][6];
/* store the states having the same edge */
int n_sameE[20]; /* store the number of states */
int GROUP[20]; /* value=1 or 2; divide into 2 groups */
int F[20];
/* Flag value = 0: initial; */
/* 1: Case 1 of Algorithm 1; 2: Case 2 */
int N1[20], N2[20];
/* store the number of elements in USOTSP and USIOS */
int TRANS[20]; /* store the number of trans for states */
char I[20][20]; /* input symbols */
char O[20][20]; /* output symbols */
char H[20][20]; /* head states */
char T[20][20]; /* Tail states */
char USOTSP[20][20][20]; /* elements in USOTSP */
char USIOS[20][20][20]; /* elements in USIOS */
char FINAL[20][20];
    /* store the last state in a signature element */
int n_found, n_state, n_tran;
int id, n_usotsp;
char s[50], str[50];
char state[20];
char name;
int FOUND[50];
char SAME[50];
int n_segment;
char SEGMENT[50][40];
char RESULT[100];
char ENDUIOS[50]; /* store the tail state of the UIOS */
char list[10000][50]; /* store the path */
int len[10000]; /* store the length of paths */
int Valid[50];
/* check a segment is connected by another */
int Valid1[50];

/* check a segment is connecting to another */
char TAIL[20][5];

/* store the tail states of transitions having*/
/* the same label*/

main()
{
    int i;

    uios();
    segment();
    connect();
    Output2();
}

/*****************************/
/* print out the connected segments */
/*****************************/
Output2()
{ 
    int idx, j, n, i, k, kk;
    char first, last;
    int found;
    int Valid3[100];
    int length;

    n=0;
    first=0;
    length=0;
    for (i=0; i<n_segment; i++) Valid3[i]=0;
    i=0;
    while (n<=n_segment)
    {
        Valid3[i]=1;
        if (SEGMENT[i][0]!\'3\')
        {
            printf("*******************************\n");
            printf("segment number= %d\n", i);
            printf("type of segment= %c\n", SEGMENT[i][0]);
            printf("starting state= %c,\n", SEGMENT[i][1]);
            printf("input/output=%c%c%c\n", SEGMENT[i][2],
        
    
}
SEGMENT[i][3], SEGMENT[i][4]);

length++;

printf("ending state= %c,\n", SEGMENT[i][6]);
}

else if (SEGMENT[i][0]=='3')
{

printf("*******************************\n.toFloat
segment number= %d\n", i);
printf("type of segment= %c\n", SEGMENT[i][0]);
printf("starting state= %c,\n", SEGMENT[i][1]);
printf("input/output=\%c%\%c\n", SEGMENT[i][2],
SEGMENT[i][3], SEGMENT[i][4]);

length++;

kk=8;

while(SEGMENT[i][kk]!='$000')
{

printf("input/output=\%c\%c\%c\n", SEGMENT[i][kk],
SEGMENT[i][kk+1], SEGMENT[i][kk+2]);

kk=kk+4;

length++;
}

printf("ending state= %c,\n", SEGMENT[i][6]);
} /* else */

last=SEGMENT[i][6];

if (SEGMENT[i][19]== -1) /* transfer path */
{
    k=22;
    printf("transfer path= ");
    while (SEGMENT[i][k+1]!='\000')
    {
        printf("%c/%c.", SEGMENT[i][k+1],
                SEGMENT[i][k+2]);
        last=SEGMENT[i][k];
        k=k+3;
        length++;
    }
    printf("\n");
    printf("the transfer path ends at state=
            %c\n", last);
}

n++;
i=SEGMENT[i][20];
if (((Valid3[i]==1)&&(n<=n_segment))
{

found=0;
j=0;
while((found==0) && (j<=n_segment))
{
    if (Valid3[j]==0)
    {
        found=1;
    }
    else
        j++;
}
if (last!=SEGMENT[j][1])
{
    idx=path(state_id(last), state_id(SEGMENT[j][1]));
    printf("-----------------------------\n");
    printf("connecting two paths from\n");
    printf(" state %c to state %c\n", last, SEGMENT[j][1]);
    printf("using path= ");
    kk=0;
    while(list[idx][kk]!='\000')
    {
        printf("%c/%c.", list[idx][kk+1], list[kk+2]);
kk=kk+3;
length++;}

printf("\n");
printf("-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-\n");
}
i=j;
first=j;

}
}
printf("\n\n");
printf("The length of the test sequence is %d input/output pairs\n", length);
}

/***************************************************************************/
/* derive segments */
/***************************************************************************/

segment()


```c
{
  int i, x, c, m, id, n;

  n_segment = -1;
  for (i=0; i<n_state; i++)
  {
    n=TRANS[i];
    for (x=0; x<n-1; x++)
    {
      id=state_id(T[i][x]);
      if ((F[id]==1)||(F[id]==2))
      {
        n_segment++;
        SEGMENT[n_segment][1]=state[i];
        SEGMENT[n_segment][2]=I[i][x];
        SEGMENT[n_segment][3]='/';
        SEGMENT[n_segment][4]=0[i][x];
        SEGMENT[n_segment][5]='. '
        SEGMENT[n_segment][6]=T[i][x];
        SEGMENT[n_segment][7]='\000';
        c=find_conv(i, x);
        if (c>1)
```
\begin{verbatim}
SEGMENT[n_segment][0]= '1'; /* converging edge */
else
    SEGMENT[n_segment][0]= '2'; /* non-converging */
}
else
{
    for (m=0; m<=N2[id]; m++)
    {
        if (USI0S[id][m][0]!= '#')
        {
            n_segment++;
            SEGMENT[n_segment][1]=state[i];
            SEGMENT[n_segment][2]=I[i][x];
            SEGMENT[n_segment][3]= '/';
            SEGMENT[n_segment][4]=O[i][x];
            SEGMENT[n_segment][5]= '.';
            SEGMENT[n_segment][6]=T[i][x];
            SEGMENT[n_segment][7]= '.';
            SEGMENT[n_segment][8]= '\000';
            SEGMENT[n_segment][0]= '3'; /* id is verified */
            strcat(SEGMENT[n_segment], USI0S[id][m]);
            SEGMENT[n_segment][6]=FINAL[id][m];
        }
    }
\end{verbatim}
connect() {
    int i;

    for (i=0; i<=n_segment; i++) {
        Valid[i]=0;
        Valid1[i]=0;
        SEGMENT[i][20] = -1; /* forward chain */
        SEGMENT[i][21] = -1; /* backward chain */
    }
    combine();
}
combine()
{
    int ii, i, j;
    char minpath[1000];
    int min, midx, idx;
    int minj;

    for (i=0; i<=n_segment; i++)
        for (j=0; j<=n_segment; j++)
            {
                if ((i!=j) && (SEGMENT[i][0]=='3') && (Valid[j]==0)
                    && (SEGMENT[j][0]=='1') && (Valid1[i]==0))
                    {
                        // code goes here
                    }
            }
}
if (SEGMENT[i][6]==SEGMENT[j][1])
{
    Valid[j]=1;
    Valid1[i]=1;
    SEGMENT[i][20]=j;
    SEGMENT[j][21]=i; /*cost=0*/
    SEGMENT[i][19]=0; /*cost=0*/
}
}

for (i=0; i<=n_segment; i++)
{
    if(((Valid1[i]==0) && (SEGMENT[i][0]==3))
    {
        minpath[0]='\000';
        min= 1000;
        minj= -1;
        midx= -1;
        for (j=0; j<=n_segment; j++)
        {
            if ((i!=j)
                && (SEGMENT[j][0]=='1') && (Valid[j]==0))
                

{ 
    idx=path(state_id(SEGMENT[i][6]),
            state_id(SEGMENT[j][1]));
    if (len[idx] < min)
    {
        min=len[idx];
        midx=idx;
        minj=j;
        strcpy(minpath, list[idx]);
    }
}

if (minj != -1)
{
    Valid[minj]=1;
    Valid1[i]=1;
    SEGMENT[i][20]=minj;
    SEGMENT[minj][21]=i; /*cost=0*/
    SEGMENT[i][19]= -1; /* transfer path from index 22 */
    for (ii=0; ii<min-1 ; ii++)
        SEGMENT[i][22+ii]=minpath[ii];
for (i=0; i<n_segment; i++)
{
  for (j=0; j<n_segment; j++)
  {
    /* connect segments having cost=0 first */
    if ((i!=j) && (SEGMENT[i][0]=='2') && (Valid[j]==0)
        && (Valid1[i]==0))
    {
      if ((SEGMENT[j][0]=='2')
          && (SEGMENT[i][6]==SEGMENT[j][1]))
      {
        Valid[j]=1;
        Valid1[i]=1;
        SEGMENT[i][20]=j;
        SEGMENT[j][21]=i; /*cost=0*/
        SEGMENT[i][19]=0; /*cost=0*/
      }
    }
  }
}

else if ((i!=j) && (SEGMENT[i][0]=='3'))
& & (Valid[j]==0) & & (Valid1[i]==0))

{
    if (SEGMENT[i][6]==SEGMENT[j][1])
    {
        Valid[j]=1;
        Valid1[i]=1;
        SEGMENT[i][20]=j;
        SEGMENT[j][21]=i; /*cost=0*/
        SEGMENT[i][19]=0; /*cost=0*/
    }
}

/* connect segments having non-zero cost */
/* using shortest sequence from i to j */
for (i=0; i<n_segment; i++)
{
    min=1000;
    midx=-1;
    minj=-1;
    minpath[0]="\000";
    for (j=0; j<=n_segment; j++)
{ 
    if ((i!=j) && (SEGMENT[i][0]==3') && (Valid[j]==0)
        && (Valid1[i]==0))
    {
        idx=path(state_id(SEGMENT[i][6]),
                  state_id(SEGMENT[j][i]));
        if (len[idx]< min)
        {
            min=len[idx];
            midx=idx;
            minj=j;
            strcpy(minpath, list[idx]);
        }
    }
}
if (midx != -1)
{
    Valid[minj]=1;
    Valid1[i]=1;
    SEGMENT[i][20]=minj;
    SEGMENT[minj][21]=i; /* cost=non-0 */
    SEGMENT[i][19]= -1; /* transfer path from index 22 */
for (ii=0; ii<=min-1 ; ii++)
    SEGMENT[i][22+ii]=minpath[ii];
}

/* connect segments having non-zero cost */
/* using shortest UID sequence from i to j */
for (i=0; i<n_segment; i++)
{
    for (j=0; j<n_segment; j++)
    {
        if ((i!=j) && (Valid[j]==0) && (Valid1[i]==0))
        {
            Valid[j]=1;
            Valid1[i]=1;
            SEGMENT[i][20]=j;
            SEGMENT[j][21]=i; /* cost non=0 */
            SEGMENT[i][19]= -1;
            /* transfer path from index 22 */
            idx=uiopath(state_id(SEGMENT[i][6]),
                        state_id(SEGMENT[j][1]));
            for (ii=0; ii<=len[idx] ; ii++)
SEGMENT[i][22+ii]=list[idx][ii];

} 
} 
} 
} 
} 
} 

/****************************************************************************/
/* shortest path */
/****************************************************************************/
path(statei, statej)
int statei, statej;
{

int found, length, idx, pp, old, idy;
int i, j, k, ii, n, total;

found=0;
length=1;
total=TRANS[statei];
idx=0;
len[idx]=0;
idy=statei;
pp = -1;
old = -1;
for (ii=0; ii<=20; ii++) len[ii] = 0;
while ((found == 0))
{
    n = 0;
    while ((found == 0) && (n<=total-1))
    {
        if (T[id][n] == state[state]) /* end of the path */
        {
            found = 1;
            idy = len[idx];
            list[idx][idy] = T[id][n];
            list[idx][idy+1] = T[id][n];
            list[idx][idy+2] = T[id][n];
            idy = idy + 3;
            list[idx][idy] = '\000';
            len[idx] = idy + 1;
        } /* not the end of the path */ /* record information */
idy=1en[idx]; /* next one to use */
list[idx][idy]=T[id][n];
list[idx][idy+1]=I[id][n];
list[idx][idy+2]=O[id][n];
idy=idy+3;
len[idx]=idy;
}

if (found==0) /* check the next transition of a state */
{
    n++;
    idx++;
}

if (found==0) /* move to the next candidate in the list*/
{
    pp++;
    id=state_id(list[pp][len[pp]-3]);
    total=TRANS[id];
    old=pp;
    if (old>=0) /* copy states in the path */
    {

for (ii=0; ii<=total-1; ii++)
{
    idy=len[old];
    for (i=0; i<=idy-1; i++)
    {
        list[idx+ii][i]=list[old][i];
    }
    len[idx+ii]=len[old];
    list[idx+ii][idy]='\000';
}
}
)
return(idx);
}

/*****************************/
/* find uio-path */
/*****************************/
uiopath(statei, statej)
int statei, statej;
```c
{
int found, length, idx, pp, old, idy;
int i, j, k, ii, n, total;
int answer;

found=0;
length=1;
total=TRANS[statei];
idx=0;
len[idx]=0;
id=statei;
pp= -1;
old= -1;
for (ii=0; ii<=20; ii++) len[ii]=0;
while ((found==0))
{
    n=0;
    while ((found==0) && (n<=total-1))
    {
```
if (T[id][n] == state[statej]) /* end of the path */
{
    idy = len[idx];
    list[idx][idy] = T[id][n];
    list[idx][idy+1] = I[id][n];
    list[idx][idy+2] = O[id][n];
    idy = idy+3;
    len[idx] = idy;
    list[idx][idy] = '\000';
    answer = checkuio(statei, idx); /* answer = 0 unique */
    if (answer == 1) found = 0; /* not unique */
    else
    {
        list[idx][idy] = '\000';
        len[idx]++;
        found = 1; /* unique */
    }
}
else /* not the end of the path */ /* record information */
{
    idy = len[idx]; /* next one to use */
list[idx][idy]=T[id][n];
list[idx][idy+1]=I[id][n];
list[idx][idy+2]=0[id][n];
list[idx][idy+3]=\000;
idy=idy+3;
len[idx]=idy;
}

if (found==0) /* check the next transition of a state */
{
    n++;
    idx++;
}
}

if (found==0) /* move to the next candidate in the list*/
{
    pp++;
    id=state_id(list[pp][len[pp]-3]);
    total=TRANS[id];
    old=pp;
    if (old>=0) /* copy states in the path */
    {
        for (ii=0; ii<=total-1; ii++)
{ 
    idy=len[old];
    for (i=0; i<=idy-1; i++)
    {
        list[idx+ii][i]=list[old][i];
    }
    len[idx+ii]=len[old];
    list[idx+ii][idy]='\000';
}

if (found==0)
    printf("***** cannot find UIO transfer path *******\n");
else return(idx);
}

/*****************************/
/* check if a path is uio-path */
/*****************************/
checkuio(statei, idx)
int statei, idx;
{
    int i, j;
    int found;
    int count;
    i=0;
    count= -1;
    found=0;
    while ((found==0) && (i<n_state))
    {
        if (i!=statei)
        {
            found=unique(i, idx);
            if (found==0)
            {
                count++;
            }
            else found=1;
        }
        i++;
if (count==n_state-1) return (0);
else return(1); /* 0: unique; 1: not-unique */

/********************************************************************************
/* check if a path is unique to one state */
/********************************************************************************
unique(i, idx)
int i, idx;
{
    int k, found1, found2, length, j;

    length=len[idx]-2;
j=0;
found1=0;
while((found1==0) && (j<=length))
{
    found2=0;
k=0;
while ((found2==0) && (k<=TRANS[i]-1))
{
if (((T[i][k]==list[idx][j+1]) && (0[i][k]==list[idx][j+2])))
{
    if (T[i][k]==list[idx][j])
    {
        found1=1; found2=1;} /* not-unique */
    else
    {
        i=state_id(T[i][k]);
        j=j+3;
        found2=1; /* check next input and output seq. */
    } /* check next transition of state i */
}
else
{
    k++; /* check next transition of state i */
}
}

if (found2==0) found1=2; /* unique */
}

if (found1==1) return(1);
else if (found1==2) return(0); /* unique */
else if ((found1==0) && (found2==1)) return(1);
else return(0);

}'

/*****************************/
/* find out the converging edges */
/*****************************/

find_conv(i, x)
int i, x;
{
    int j, mm, c;

    mm=0;
    c=0;
    while ((c==0) & (mm<n_edge))
    {
        if ((Edge[mm][0]==I[i][x]) & (Edge[mm][1]==O[i][x]))
        {
            for (j=0; j<n_sameE[mm]; j++)
            {
                if (T[i][x]==TAIL[mm][j]) c++;
            }
        }
    }
}
else mm++; 
} 
return(c); 

/**********************************************************/
/* find tail state */
/**********************************************************/

find_tail(i, m)
int i, m;
{
    int n, c, x, w, tail;

    n=TRANS[i];
c=0;
x=0;

    while((c<0)&&(x<=n-1))
    {
        if (((I[i][x]==Edge[m][0])&(O[i][x]==Edge[m][1]))
            
        
    }
tail=T[i][x];
c=1;
}
else x++;
}
return(x);

/**********************************
/* derive uios and signature set */
/**********************************
uios()
{
Init();
input();
cal_Set(); /* step 1 */
Find_UIO();
Concat_UIO();
Signature();
Output();
}
/*******************************
/* In itia liza tio n */
/*****************************/

Init()
{

}

/*****************************/
/* Determine the set for each edge */
/*****************************/

cal_Set()
{
    int i, m, x, n, c;

    for (i=0; i<n_state; i++) n_sameE[i]=-1;
    n=TRANS[0];
    for (x=0; x<n-1; x++)
    {

    }
```c
Edge[x][0]=I[0][x];
Edge[x][1]=0[0][x];
Cnt[x]=0;
sameE[x][0]=0;
TAIL[x][0]=T[0][x];
n_sameE[x]=0;
}
n_edge=n-1;
for (i=1; i<= n_state; i++)
{
    n=TRANS[i];
    for (x=0; x<=n-1; x++)
    {
        c=exist(i,x);
        if (c==0)
        {
            n_edge++;
            add(n_edge,i,x);
            Cnt[n_edge]=0;
            n_sameE[n_edge]=0;
            TAIL[n_edge][0]=T[i][x];
            sameE[n_edge][0]=i;
```
else /* existing */
{
    Cnt[c]++;
    n_sameE[c]++;
    sameE[c][n_sameE[c]]=i;
    TAIL[c][n_sameE[c]]=T[i][x];
}
}
}

/****************************/
/* check if an edge exists */
/****************************/

exist(i, x)
int i, x;
{
    int m, find;
m=0;
find=0;
while ((m<n_edge) && (find==0))
{
    if ((Edge[m][0]==I[i][x])&&(Edge[m][1]==0[i][x]))
        find=1;
    else m++;
}
if (find==0) return(0);
else return(m);
}

/********************
/* add an edge     */
/********************

add(n, i, x)
int n, i, x;
{
    Edge[n][0]=I[i][x];
Edge[n][i]=0[i][x];
}

/****************************************************************************
/* Find UI0 of length 1 */
/****************************************************************************/

Find_UI0()
{
    int i, m, n;

    for (i=0; i<=n_state; i++)
    {
        N2[i]= -1; F[i]=0;
    }
    for (m=0; m<=n_edge; m++)
    {
        i=sameE[m][0];
        if ((Cnt[m]==0) && (F[i]==0))
        {
            F[i]=1;
            save(m, i, 0);
            N2[i]=0;
        }
    }
}
/* save a UIOS into USIOS */

save(m, i, x)
int m, i, x;
{
    USIOS[i][x][0]=Edge[m][0];
    USIOS[i][x][1]='/';
    USIOS[i][x][2]=Edge[m][1];
    USIOS[i][x][3]='/000';
}

/* Find UIOS of length > 1 */

Concat_UIO()
int n_found, y, i, m, j, c, n;
int k, x, new, min, min1, min2, min3, count;

n_found = -1;
for (i=0; i<=n_state; i++)
{
    if (F[i]--1)
    {
        n_found++;
        FOUND[n_found]=i;
    }
}

/**** find better UIOS *****/
new=0;
while((n_state>n_found)&&(new==0))
{
    new=1;
    j=0;
    while(j<=n_state)
    {
        if (F[j]==0)
        {

    }
min=100;
min1=1;
min2=-1;
n=TRANS[j];
for (y=0; y<n; y++)
{
    for (m=0; m<=n_found; m++)
    {
        i=FOUND[m];
        if (T[j][y]==state[i])
        {
            c=check_tsp(j,y);
            if (c==0)
            {
                count=find_cnt(j,x);
                if (count<min)
                {
                    min=count;
                    min1=j;
                    min2=y;
                    min3=i;
                }
            }
        }
    }
}
if (min != 100)
{
    new=0;
n_found++;
    FOUND[n_found]=j;
P[min1]=2;
N2[min1]=0;
USIOS[min1][0][0]=I[min1][min2];
USIOS[min1][0][1]='/';
USIOS[min1][0][2]=0[min1][min2];
USIOS[min1][0][3]='.';
USIOS[min1][0][4]='/00';
strcat(USIOS[min1][0],USIOS[min3][0]);
}
}
j++;
find_cnt(i, x)
int i, x;
{
    int m, c;

    m=0;
    while ((c==0) && (m<n_edge))
    {
        if ((Edge[m][0]==I[i][x]) && (Edge[m][1]==O[i][x]))
            c=1;
        else m++;
    }
    return(Cnt[m]);
}

/*******************************/
/* find out the number of duplicates */
/*******************************/
/* Find signature set for states */

Signature()
{
  int i, j, n;
  char str[50];
  int last;

  for (i=0; i<=n_state; i++)
  {
    if (F[i] == 0)
      {
        F[i] = 3;
        N2[i] = -1;
        for (j=0; j<=n_state; j++)
        {
          if (i != j)
            {
              N2[i]++;
              last = find_IO(i, j, str);
              FINAL[i][N2[i]] = state[last];
            } // if (i != j)
        } // for (j=0; j<=n_state; j++)
      } // if (F[i] == 0)
  } // for (i=0; i<=n_state; i++)
strcpy(USIOS[i][N2[i]], str);

Output()
{
   int c, i, j, k, n;

   for(i=0; i<=n_state; i++)
   {
      n=N2[i];
      for (j=0; j<=n; j++)
      {
         for (k=j+1; k<=n; k++)
         {
            c=contain(i, j, k);
            if (c==1) /* j contains k */
USIOS[i][k][0] = '#';
else if (c == 2) /* k contains j */
    USIOS[i][j][0] = '#';
}

for(i = 0; i <= n_state; i++)
{
    j = 0;
    printf("\n");
    printf("----------------------------------------------------------\n");
    printf("*** state %c has the following set ***\n", state[i]);
    n = N2[i];
    for (k = 0; k <= n; k++)
    {
        if (USIOS[i][k][0] != '#')
        {
            j++;
            printf(">>> element %d: %s\n", j, USIOS[i][k]);
        }
    }
}
}    
    printf("-------------------------------\n");
}

/********************************************************************************
/* check if a string is contained */
/* in another */
/********************************************************************************
contain(i, j, k)
int i, j, k;
{
    int m, n;
    m=0;
    while ((USIOS[i][j][m]!=\000') &
        (USIOS[i][j][m]==USIOS[i][k][m])) m++;
    if (USIOS[i][j][m]==\000') return(2);
    else if (USIOS[i][k][m]==\000') return(1);
    else return(0); /* no contain relationship */
}

/********************************************************************************
/* find a sequence less than or equal to n */

/***************************************************/

find_ID(i, j, str)

int i, j;
char str[50];
{
    int x, n, y;

    str[0]=\000;
    Ass_Group(i);
    n= -1;
    while((GROUP[i]==GROUP[j]) && (n<=n_state))
    {
        n++;
        x=S_Group(i, j, &y);
        s[0]=I[i][x];
        s[1]='/';
        s[2]=O[i][x];
        s[3]='.';
        s[4]=\000;
        strcat(str, s);
        i=state_id(T[i][x]);
    }
j = state_id(T[i][j]);
Ass_Group(i);
}
x=D_Group(i,j);
s[0]=I[i][x];
s[1]='/';
s[2]=D[i][x];
s[3]='.';
s[4]='/000';
strcat(str,s);
return(state_id(T[i][x]));
}

/*****************************/
/* find the index of a state name */
/* in an array */
/*****************************/
state_id(ch)
char ch;
{
    int i, find;
i=0;
find=0;
while((find==0)&&(i<n_state))
{
    if (state[i]==ch) find=1;
    else i++;
}
return(i);
}

/*****************************/
/* Divided into 2 groups according to i */
/*****************************/

Ass_Group(i)
int i;
{
    int j, c;

    GROUP[i]=1;
    for (j=0; j<=n_state; j++)
```c
{ 
    if (i!=j) 
    { c=same(i,j); 
        if (c==0) GROUP[j]=1; 
        else GROUP[j]=2; 
    }
}

/*************************************************************************/
/* check if two states having the same                        */
/* transition set                                              */
/*************************************************************************/

same(i, j) 
int i, j; 
{ 
    int c, x, y, find, n, m; 
    int small, big;

    c=0;  
    x=0;
```
if (TRANS[i] <= TRANS[j])
    { small=i; big=j; }
else { small=j; big=i; }
n=TRANS[small];
while ((c==0) & (x<=n-1))
{
    y=0;
    find= 0;
    m=TRANS[big];
    while (y<=m-1)
    {
        if((I[small][x]==I[big][y]) & (O[small][x]==O[big][y]))
        {
            find++;
            y=m+10;
        }
        else
            y++;
    }
    if (y==m)
        c++; /* not in the same group */
else x++;

return(c);

/**
 * find tspo for the same group
 */
/***************************/

S_Group(i, j, p)
int i, j, *p;
{
  int c, x, n, m, find, y;
  int small, big;

  c=0;
  x=0;
  if (TRANS[i]<=TRANS[j]) {small=i; big=j;}
  else {small=j; big=i;}
  n=TRANS[small];
  while ((c==0) && (x<n-1))
  {
    y=0;
```c
find=0;
m=TRANS[big];
while ((find==0) && (y<=m-1))
{
    if((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])
        && (T[i][x]!=T[j][y])) find++;
    else y++;
}
if (find !=0) c++;  
else x++;
}
*p=y;
return(x);
}

/*******************************************************************************/
/ * find transitions for different group   */
/*******************************************************************************/

D_Group(i, j)
int i, j;
{

int find, c, x, y, n, m;

c=0;
x=0;
n=TRANS[i];
while ((c=0) & (x<=n-1))
{
    y=0;
    find=0;
    m=TRANS[j];
    while ((find=0) & (y<=m-1))
    {
        if((I[i][x]==I[j][y]) & (0[i][x]==0[j][y])) find++;
        else y++;
    }
    if (find ==0) c++;
    else x++;
}
return(x);

/*********************/
input()
{
    char dummy;
    int n_tran, i;

    scanf("%c", &name);
    n_state= -1;
    while (name!='*')
    {
        n_state++;
        state[n_state]=name;
        n_tran= -1;
        scanf("%d", &n_tran);
        TRANS[n_state]=n_tran;
        scanf("%c", &dummy);
        for (i=0; i<n_tran; i++)
        {
            scanf("%c", &I[n_state][i]);
        }
    }
}
scanf("%c", &O[n.state][i]);
scanf("%c", &T[n.state][i]);
scanf("%c", &dummy);
O[n.state][i], T[n.state][i]);*/
}

name='#';
scanf("%c", &name);
}
}

/*****************************/
/* decide if a given transition is unique*/
/*****************************/

check_t(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;
    while ((c==0) && (j<n.state))
{ 
    if (i!=j) 
    { 
        y=0; 
        n_tran=TRANS[j]; 
        while ((c==0) && (y<n_tran-1)) 
        { 
            if ((1[i][x]==1[j][y]) && (0[i][x]==0[j][y])) c++; 
            y++; 
        } 
    } 
    j++; 
} 
if (c==0) return(0); 
else return(1); 
} 

/****************************/ 
/* decide if a given tsp is unique*/ 
/****************************/ 

check_tsp(i, x)
int i, x;
{
    int j, y, c;

    c=0;
    j=0;
    while ((c==0) && (j<=n_state))
    {
        if (i!=j)
        {
            y=0;
            n_tran=TRANS[j];
            while ((c==0) && (y<=n_tran-1))
            {
                if ((I[i][x]==I[j][y]) && (O[i][x]==O[j][y])
                && (T[i][x]==T[j][y])) c++;
                else y++;
            }
        }
        j++;
    }
    if (c==0) return(0);
else
{
    return(1);
}
}
Appendix D

Sample Output of the UIOpv Method

*** state A has the following set ***
>>> element 1: 0/1

*** state B has the following set ***
>>> element 1: 1/1

*** state C has the following set ***
>>> element 1: 2/1.0/1.
>>> element 2: 1/0.1/1.

*** state D has the following set ***
>>> element 1: 1/0.1/0.
>>> element 2: 2/1.2/1.
state X has the following set

>> element 1: 3/2
Appendix E

Sample Output of the TSP Method

*** state A has the following set ***
>>> element 1: 0/1.

*** state B has the following set ***
>>> element 1: 1/1.

*** state C has the following set ***
>>> element 1: 2/1.0/1.
>>> element 2: 1/0.1/1.

*** state D has the following set ***
>>> element 1: 1/0.1/0.
>>> element 2: 2/1.2/1.
*** state X has the following set ***

>>> element 1: 3/2.
Appendix F

Sample Output of the MLTG Method

---

*** state A has the following set ***
>>> element 1: 0/1

---

*** state B has the following set ***
>>> element 1: 1/1

---

*** state C has the following set ***
>>> element 1: 2/1.0/1.
>>> element 2: 1/0.1/1.

---

*** state D has the following set ***
>>> element 1: 1/0.1/0.
>>> element 2: 2/1.2/1.
*** state X has the following set ***

>>> element 1: 0/2

segment number= 0

type of segment= 2

starting state= A

input/output=0/1

ending state= X

transfer path= 0/2.1/0.1/0.

the transfer path ends at state= B

segment number= 5

starting state= B

input/output=1/1

input/output=1/0

input/output=1/1

ending state= C
segment number = 7

type of segment = 1
starting state = C
input/output = 1/0
ending state = B
transfer path = 1/1.2/1.0/1.0/2.
the transfer path ends at state = D

*******************************************************************************

segment number = 10

type of segment = 3
starting state = D
input/output = 2/1
input/output = 2/1
input/output = 2/1
input/output = 0/1
ending state = X

*******************************************************************************

segment number = 15

type of segment = 3
starting state = X
input/output = 0/2
input/output = 1/0
input/output = 1/0
ending state= B

segment number= 2
starting state= B
input/output= 2/1
input/output= 2/1
input/output= 2/1
input/output= 0/1
ending state= X

segment number= 14
starting state= X
input/output= 2/1
ending state= A
transfer path= 0/1.0/2.
the transfer path ends at state= D

segment number= 11
starting state= D
input/output= 2/1
input/output=1/0
input/output=1/1
ending state= C
transfer path= 1/0.
the transfer path ends at state= B

==================================================
segment number= 3
type of segment= 3
starting state= B
input/output=2/1
input/output=1/0
input/output=1/1
ending state= C

==================================================
segment number= 6
type of segment= 1
starting state= C
input/output=2/1
ending state= A
transfer path= 0/1.0/2.
the transfer path ends at state= D

==================================================
segment number= 9

type of segment= 3

starting state= D

input/output=1/0

input/output=1/0

input/output=1/1

ending state= C

transfer path= 2/1.

the transfer path ends at state= A

*******************************

segment number= 1


type of segment= 1

starting state= A

input/output=1/0

ending state= B

transfer path= 1/1.2/1.0/1.0/2.

the transfer path ends at state= D

*******************************

segment number= 8


type of segment= 3

starting state= D

input/output=1/0
input/output=2/1
input/output=0/1
ending state= X

********************************************
segment number= 13
type of segment= 3
starting state= X
input/output=1/0
input/output=1/0
input/output=1/1
input/output=1/1
ending state= C
transfer path= 1/0.
the transfer path ends at state= B

********************************************
segment number= 4
type of segment= 3
starting state= B
input/output=1/1
input/output=2/1
input/output=0/1
input/output=0/1
ending state= X

********************************************
segment number= 12
type of segment= 3
starting state= X
input/output=1/0
input/output=2/1
input/output=0/1
ending state= X

**********************
segment number= 16
type of segment= 3
starting state= X
input/output=0/2
input/output=2/1
input/output=2/1
ending state= A

**********************

The length of the test sequence is 59 input/output pairs
BIBLIOGRAPHY


