NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

On the Formal Methods for Protocol Conformance Testing

by

Yueping Lu

A M.Sc. Thesis

Submitted to the School of Graduate Studies and Research

in partial fufilment of the requirements for the

Master of Computer Science Degree*

University of Ottawa

Ottawa, Ontario

Canada

* The Master of Computer Science program is a joint program with Carleton University, administrated by the Ottawa-Carleton Institute for Computer Science

ISBN 0-315-60104-3

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

The use of formal methods allows automated generation and optimization of test sequences. Developing formal methods for generating communications protocol conformance tests has drawn considerable attention in recent years.

This thesis describes the implementation of five formal methods of protocol conformance test sequence generation proposed in the literature. These methods are: Transition tour (T) method, Distinguishing sequence (D) method, Characterizing sequence (W) method, Unique Input/Output sequence (UIO) method and Multiple UIO-method. Some related graph theoretic optimization techniques for the Chinese Postman Problem (CPP) and Rural Chinese Postman Problem (RCPP) are applied to T-method and UIO-method respectively to determine minimum-cost test sequences.

It is shown in this thesis that the solution to the RCPP can also be applied to D-method and W-method to derive minimum-cost test sequences. The application of five formal methods to a real protocol, Transport protocol class 4, is then discussed.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude and appreciation to my thesis supervisor, Dr. Hasan Ural. He has provided extremely valuable advice and guidance in every stage of this thesis.

I wish to acknowledge the financial support provided by National Sciences and Engineering Research Council of Canada and Telecommunications Research Institute of Ontario.

I wish to thank the Protocols Research Group for providing an excellent research environment for the entire period of my study leading to this thesis work.

Finally, my special thanks go to my family and all my friends in Canada for their generous help and encouragement during my study at University of Ottawa.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

1.1 The Background and the Motivation of the Thesis

The development of international standards for Open Systems Interconnection (OSI) makes the interworking between different kinds of computer systems viable. Using public data networks, computers and terminals in different parts of the world can exchange information provided that they agree on the use of OSI communication protocols.

A **protocol** defines a set of rules governing orderly exchange of information among communicating entities. A **protocol specification** describes the externally observable behavior of a (protocol) entity in terms of possible sequences of interactions exchanged between the protocol entity and its environment.

However, for any pair of interworking computer systems, it is not sufficient to agree on the use of a particular standardized set of protocols, it is also necessary that each of the communicating computer systems implement these protocols in a correct manner. Thus, to ensure reliable communication among communicating systems, the protocol implementations within each system should be tested for their conformance to their specifications. This is known as **Protocol Conformance Testing.**

The wide range and high complexity of services expected from a communication protocol has increased the complexity of protocols and made the process of protocol conformance testing increasingly important and challenging. The use of formal description techniques (FDT) has been advocated to construct unambiguous, precise and concise specifications of communication protocols. The use of FDTs also promotes the automated derivation of partial implementations and the development of formal methods for generating tests from formal specifications. ESTELLE [ISO9074], LOTOS [ISO8807], and SDL [SaTi87] are three FDTs standardized by international standardization organizations ISO and CCITT, respectively. LOTOS is based on process algebra [BeKl84] and abstract data

types [EhMa85]. SDL and ESTELLE are based on Extended Finite State Machine Models (EFSM) [Boch80].

A protocol specified as an EFSM describes both control and data flow aspects of a protocol. Control flow aspects of a protocol is expressed in terms of sequences of interaction identifiers whereas data flow aspects of the protocol is expressed in terms of the relationships among interaction parameters.

Testing the data flow aspects of a protocol is based on tests that are generated by applying principles of data flow analysis [FoOs76] and functional testing [Howd80] to protocol specifications. The methods that exploit principles of data flow analysis and functional testing for generating test sequences are reported in [Ural87, SBC87]. These methods are appliable to EFSM-based protocol specifications.

The selection of interaction sequences for testing the control flow aspects of a protocol is facilitated by finite state machine (FSM)-based formal test selection methods since the control flow aspects of a protocol can be modeled as a FSM. Typically, a FSM-based formal test selection method generates a test sequence to test every transition in a FSM.

Naito[NaTs81] suggested the transition tour approach (T-method) for FSM representations of sequential circuits and gave a heuristic for generating a tour of the FSM. T-method was later applied to testing control flow aspects of a protocol by [SaBo82]. In [UyDa86], Uyar and Dahbura observed that obtaining an optimal transition tour can be reduced to the well-known Chinese Postman Problem (CPP) [Kuan62] and adapted the solution of CPP to T-method in obtaining minimum-cost test sequences.

All the other formal methods such as Distinguishing sequence (D-method), Characterizing set (W-method), originate from "Checking Experiments" designed for sequential FSM testing [Gill62, Henn64, Gone70, Chow78]. The UIO-method is the only formal method which is proposed originally for protocol testing [SaDa85]. Test sequences generated by the UIO-method are optimized by applying a graph traversal technique called

Rural Chinese Postman Tour (RCPT) [ADLU88]. The latest improvement of the UIO-method called Multiple UIO-method is reported in [SLD89] .

With the rapid development of the data communications and the increasing complexities of communication protocols, formal methods are required to automate the selection of the shortest possible (i.e., optimized) interaction sequences to test the implementations of these complex protocols. The formal methods that are implemented in this thesis fulfill this requirement for testing the control flow aspects of protocols modeled as FSM's.

1.2 The Contributions of the Thesis

This thesis is intended to

a) improve the existing test selection strategies based on the D-method and the W-method so that they can be used to generate optimized test sequences;

b) provide the software tools implementing five formal methods for automatic, systematic and optimal test sequence selections for protocol conformance testing;

c) ease some assumptions (such as minimality and completeness) of some of the formal methods on FSM based specifications so that these formal methods can be applied to real protocols which may not be modeled by minimal and /or completely specified FSM's;

d) select test sequences for a real communication protocol : Transport protocol class 4.

1.3 The Organization of the Thesis

Chapter 2 gives a brief review of the basic terms and concepts in protocol conformance testing. Chapter 3 describes the widely used FSM-based testing principles. Chapter 4 provides detailed descriptions of the five formal methods for protocol conformance test sequence generation in the literature : T-method, D-method, W-method, Single UIO-method and Multiple UIO-method. Chapter 5 presents possible extensions of the D-method and W-method. Chapter 6 discusses implementation issues of these formal methods. Complexities

of these methods are also given in this chapter. Chapter 7 shows examples of applications of the five methods to a real protocol - Transport protocol class 4. Conclusions and future work are discussed in the last chapter.

CHAPTER 2 PROTOCOL CONFORMANCE TESTING

This chapter reviews basic terms and concepts developed by the ISO and the CCITT for the conformance testing of the OSI protocol implementations.

The OSI reference model [Zimm80] consists of seven layers (Figure 2.1). Layer N provides certain services called **N_Services** to the next upper layer. To provide N_services, peer **N_Entities** communicate with each other by exchanging N_Protocol Data Units (**N_PDUs**) (Figure 2.2) over the service provided by the (N-1)_service provider.

Interactions of an N_entity with its upper and lower layer entities are defined by N_ and (N-1)_Abstract Service Primitives (**ASP**), respectively. An **N_Protocol** defines rules for the communication between two N_Entities. An **N_protocol specification** describes the behavior of an N_entity in terms of its responses to requests (i.e., N_ASPs) from its user , indications (i.e., (N-1)_ASPs) from the (N-1)_service provider, messages (i.e., N_PDUs) from its peer , and internal events such as timeouts.

The objective of the conformance testing [ISO9646] of an N_entity implementation is to determine if the behavior of the implementation is in accordance with the N_protocol specification. During the protocol conformance testing of an N_entity, an external tester applies a sequence of inputs (e.g., requests, indications, messages, etc.) to the implementation and verifies that the behavior of the implementation is as described in its specification.

Since the protocol implementation details (source code, etc) are generally not available to the tester, test sequences are derived from the particular protocol specification of a implementation. Thus, during the conformance testing, the implementation under test (IUT) must be treated as a "**black box**". This approach which only considers the externally observable behavior of the IUT is called **black box testing.**

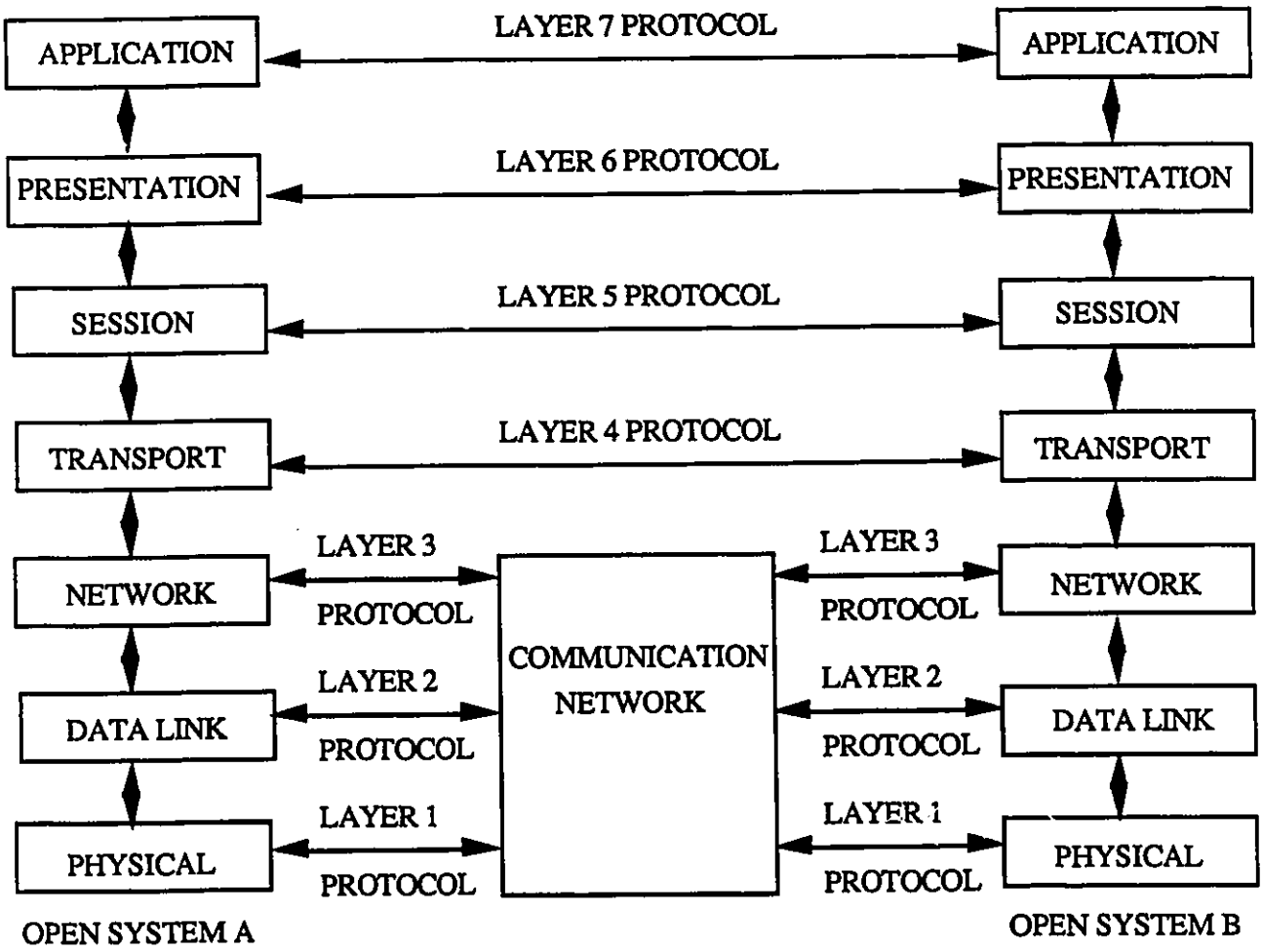


Figure 2.1 Open Systems Interconnection Reference Model

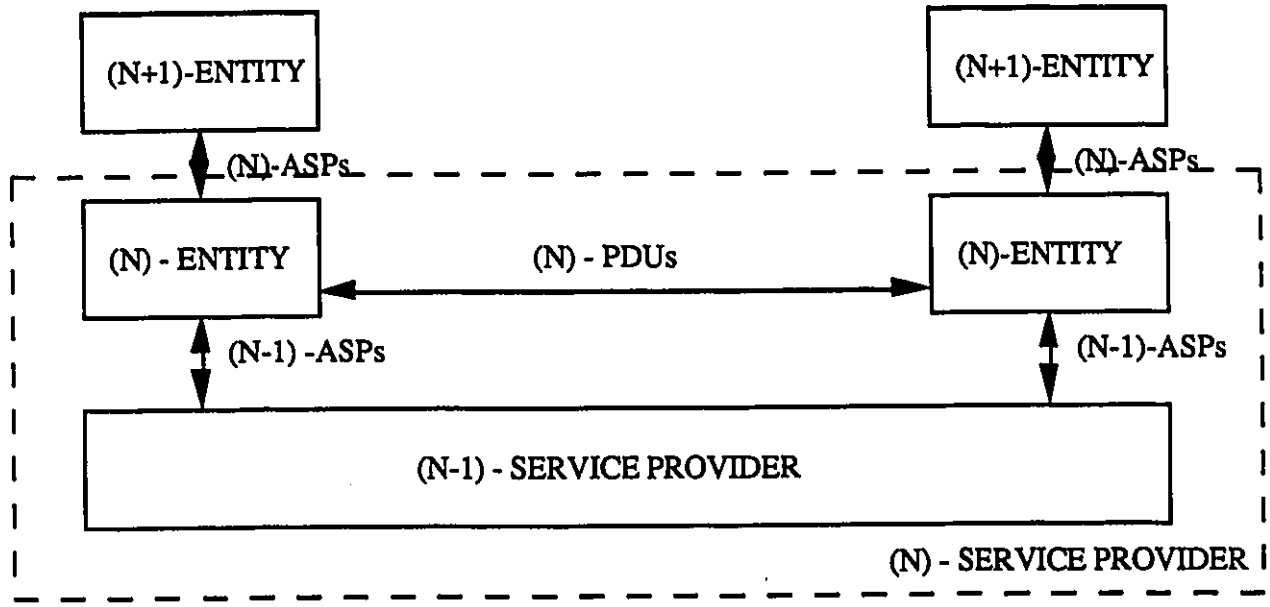


Figure 2.2 Abstraction of an (N)-layer in the OSI Reference Model

2.1 Abstract Test Methods

The black box testing is complicated by the limitations on the controllability and the observability of the protocol implementation. The point at which the exchange of (N-1)_ and N_ASPs and N_PDUs can be controlled and observed during conformance testing is called the **Point of Control and Observation (PCO).** Depending on the available PCOs, various abstract test methods have been identified, each of which is explained next.

Local Testing Method : In this abstract test method, points of control and observation are defined at the lower and upper service boundaries of the IUT (Figure 2.3). In other words, (N)_PDUs, (N)_ and (N-1)_ASPs are assumed to be available such that the upper and lower testers can control and observe them. In Figure 2.3, the **upper tester (UT)** refers to the part of the function that control and observe (N)_ASPs. It acts as the user of the IUT. Similarly, the **lower tester (LT)** controls and observes (N)_PDUs and (N-1)_ASPs. It exchanges (N)_PDUs with the IUT , acting as a peer entity of the IUT. The test cases are interpreted and evaluated by the LT. During testing, the coordination between the actions to be taken by the UT and LT are provided by *test coordination procedures* (**TCP**). Test architectures based on local testing method are mainly used for in-house testing of IUTs by their implementers.

The following three abstract test methods are appliable to testing carried out by users and the third party test centers.

Distributed Testing Method : In this abstract test method, the lower interface of an IUT is not an exposed interface, thus (N-1)_ASPs and (N)_PDUs cannot be directly controlled and observed. (N-1)_ASPs and (N)_PDUs are controlled and observed indirectly (remotely) at the opposite side of the (N-1)_Service Provider by the LT (Figure 2.4). Another PCO is defined at the upper service boundary of the IUT. LT and UT reside in physically separate locations in this method. This method (as well as the next method) is

very suitable for testing end-to-end protocols (i.e., protocols for the transport layer and higher in the OSI reference model).

Coordinated Testing Method : This is an enhanced version of the distributed testing method. The control and observation of (N)_ASPs are performed by the TCPs. The LT is considered to be the master of the UT. In this method, there is only a single PCO, at the opposite side of the (N-1)_Service Provider from the (N)_entity under test. The actions of the UT are controlled by the LT through a *test management protocol* (TMP). The TMP uses TM_PDUs to coordinate the actions between the UT and LT (Figure 2.5). TM-PDU's are exchanged via out-of-band services.

Remote Testing Method : This abstract test method is applied to IUTs which do not have exposed upper interfaces (Figure 2.6). A LT and an optional UT reside in different locations physically as in the last two abstract test methods. The only PCO is defined on the opposite side of the (N-1)_Service Provider from the (N)_entity under test. Coordination between the LT and UT is managed by means of TCPs. This method is the most restrictive of all the methods since it does not allow the control and observation of N_ASPs. Thus, it is suitable to test protocols that are not end-to-end.

For lower layers (1-3) where it may be unrealistic to specify observation and control of (N-1)_ASPs, the LT observation and control are specified in terms of (N)_PDUs and when necessary the state of underlying connection.

The above three abstract testing methods all have the advantage of being appliable to a realistic communications environment. Depending on the availability of PCOs in an IUT, testers can implement one or several methods as their test architectures for fully realizing the conformance testing of an IUT.

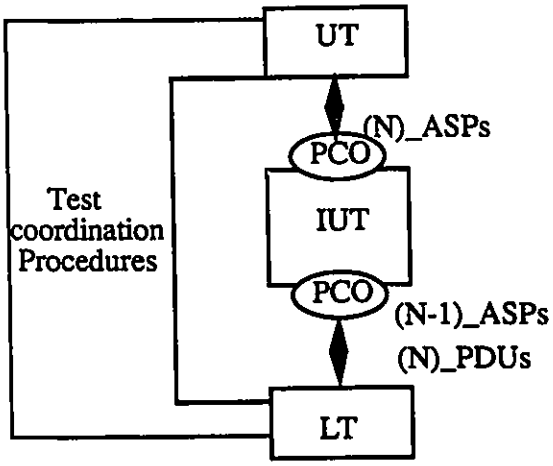


Figure 2.3 The Local Test Method

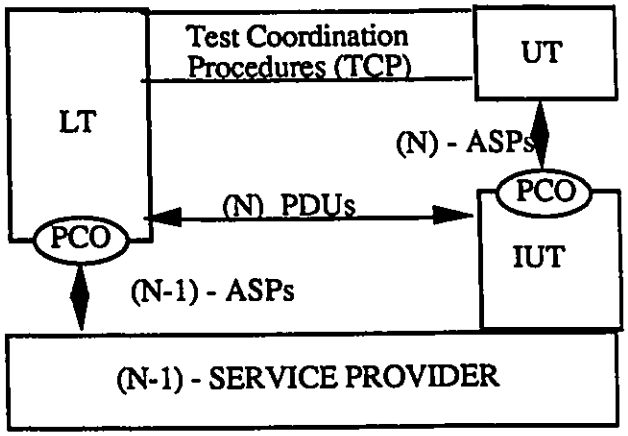


Figure 2.4 The Distributed Test Method

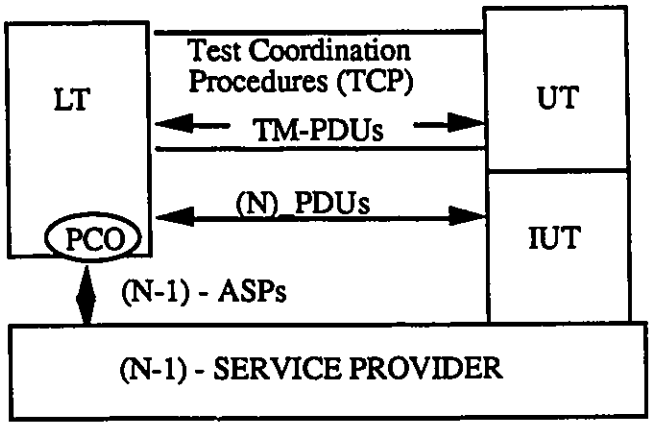


Figure 2.5 The Coordinated Test Method

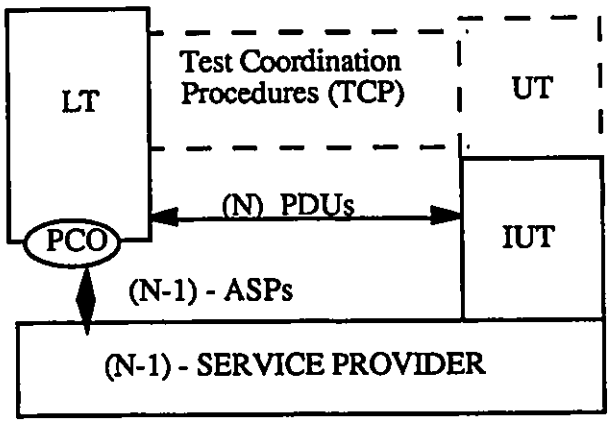


Figure 2.6 The Remote Test Method

2.2 TTCN

TTCN (Tree and Tabular Combined Notation) is an informal abstract notation [ISO 9646]. It is defined to represent conformance test suites in a standard fashion. In protocol conformance testing, **test suites** have a hierarchical structure (Figure 2.7). Within a test suite, nested **test groups** are used to provide a logical ordering of the test cases. Test groups may be nested to any arbitrary depth. A **test case** consists of a series of test events. **Test events** are named for atomic interactions between the IUT and an upper or lower tester. Each test case has a particular purpose to verify a certain capability of the IUT [Rayn87].

In [ISO 9646], four types of tests constitute a conformance test suite :

Basic Interconnection Tests : provide a limited testing to check that the IUT can establish a basic interconnection before thorough testing is performed.

Capability Tests : are used to check whether the IUT can provide the observable capabilities based on the static conformance requirements which are the requirements describing the options, ranges of values for parameters and timers, etc.

Behavior Tests : are used to check the dynamic conformance requirements of an IUT which are the requirements (and options) defining the observable behavior of a protocol. A large

part of behavior tests, which constitute the major portion of conformance tests, can be generated by the formal methods discussed later in this thesis.

Conformance Resolution Tests : are used to provide definite diagnostic answers to specific requirements for the IUT.

In this thesis, the formal methods that we consider are used to generate behavior tests. The remaining tests can be generated by using ad hoc methods. Also, it is important to note that the "*input/output* (i/o)" operation we use corresponds to *send/receive* a single PDU or ASP *to/from* the IUT, which is a test event.

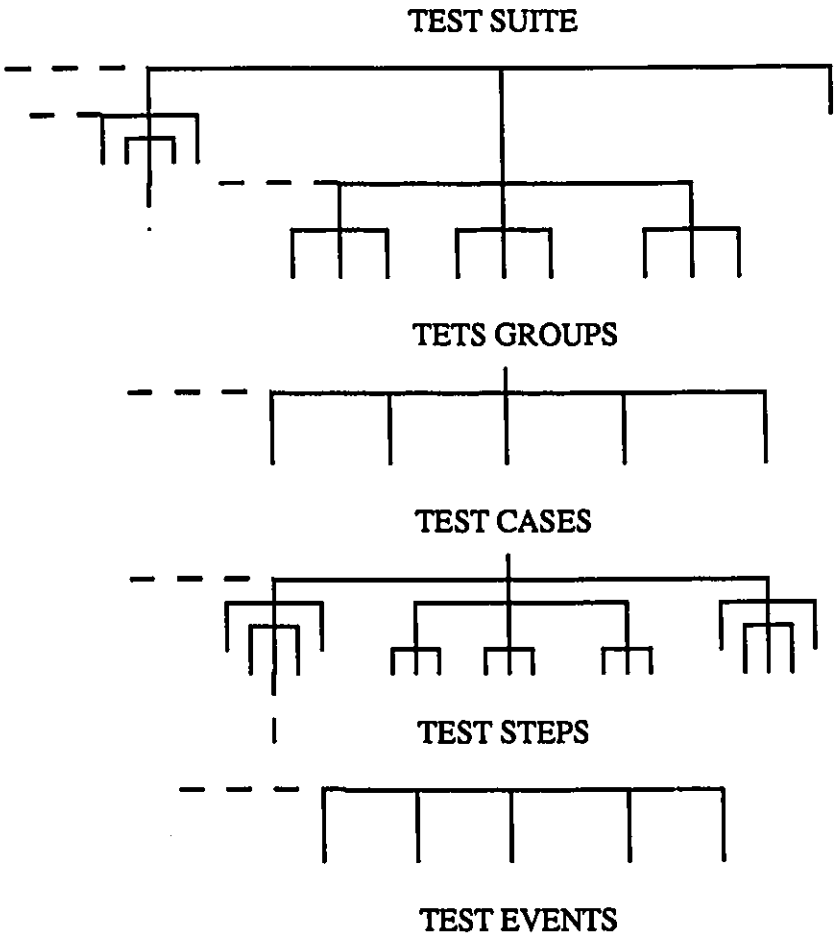


Figure 2.7 Test Suite Structure

CHAPTER 3 FINITE STATE MACHINE BASED TESTING

3.1 The Use of FSM Model in Testing

Protocols are traditionally modeled as Extended Finite State Machines (EFSM) [BoSu83] where the control portion which consists of interaction sequences can be represented by a Finite State Machine (FSM), and the data portion consists of operations on interaction parameters and context variables.

The control portion of a protocol can be tested by FSM based test methods. Some methods such as data flow analysis can be used to test the data portion of a protocol [Ural87]. The Formal methods to be described in the following sections are FSM based test methods.

A FSM describes the control portion of a protocol in terms of possible sequences of state transitions. The present state of a FSM is defined as a stable condition in which the FSM rests until a *stimulus*, called *input* is applied. The FSM generates a response to the stimulus, called *output*, and moves to its next state. The purpose of FSM-based testing is to check whether the control portion of an implementation of a given FSM M is as defined by M. Those specific stimuli that force the implementation under test to undergo certain transitions are called a **Test Sequence(TS).**

A **minimal test sequence** is a TS with the least number of stimuli among all the possible TSs generated by the same formal method. The test sequences generated by formal methods which will be described later can be viewed as an ordered set of test events, the smallest units of a conformance test suite.

3.2 The FSM Model and Its Graph Representation

3.2.1 The FSM Model

A Mealy machine M is a FSM which can be characterized by a quintuple [Koha78]:

$M = (S, I, O, \delta, \lambda)$, where

S : is a set of states $\{s_1, s_2, ..., s_n\}$;

I : is a set of input symbols $\{i_1, i_2, ..., i_q\}$;

O : is a set of output symbols $\{o_1, o_2, ..., o_r\}$;

δ : is a mapping $S \times I \rightarrow S$, called **state transition function;**

λ : is a mapping $S \times I \rightarrow O$, called **output function;**

We define $\Delta \subseteq S \times I \times S$, $\Lambda \subseteq S \times I \times O$.

A state transition from state s_j to state s_k in FSM is denoted by $T_{jk} = (s_j, s_k; i/o)$ where $(s_j, i, s_k) \in \Delta$ and $(s_j, i, o) \in \Lambda$.

The start state $s_S \in S$ of M is a designated state for M. Very often, there is a designated input "*reset*" (or "ri") $\in I$, which takes M to s_S from any other state with a single transition producing a "null" output.

A FSM M is said to be **deterministic** if for each input $i \in I$, there is only one transition defined for each state of M.

A FSM M is said to be **completely specified** if for each input $i \in I$, there is a transition defined for each state of M.

In practice, a given FSM is often incompletely specified and input interactions received at a given state of the FSM can be classified as follows:

proper : syntactically correct and a transition defined for it ;

inopportune : syntactically correct and no transition defined for it ;

illegal : syntactically incorrect.

Often, a completeness assumption for an incompletely specified FSM M is used. Two such assumptions are given below :

1) Assume that M ignores each inopportune and illegal input by producing a null output and remaining in its present state.

2) Assume that M produces an error output and enters an error state whenever it receives an inopportune or illegal input. M remains in error state where it ignores all inputs until it receives a reset input.

Two states s_i and s_j of a FSM M are **distinguishable** *if and only if* there exists at least one finite input sequence which, when applied to M, causes different output sequences [Koha78].

States s_i and s_j of a FSM M are said to be **equivalent** *if and only if* for every possible input sequence, the same output sequence will be produced regardless of whether s_i or s_j is the initial state.

A FSM M is said to be **minimal** (reduced) if any pair of its states is distinguishable (i.e. no equivalent states in M).

3.2.2 Graph Representation of a FSM

A FSM M can be represented by a directed graph G=(V, E) where a finite set of vertices $V = \{v_1, v_2, ..., v_n\}$ represents the set S of states of M and a set of directed edges $E = \{(v_j, v_k; i/o): v_j, v_k \in V\}$ represents all specified transitions of M. Each edge $(v_j, v_k; i/o) \in E$ represents a state transition from s_j to s_k under input i and generating output o.

G may contain multiple edges and **self-loops** which stand for transitions starting and ending with the same state of M. A directed graph G is said to be **strongly connected** if for any pair of vertices v_j and v_k, there exists a path from v_j to v_k. A directed graph G is said to be **weakly connected** if its underlying undirected graph is connected.

In subsequent chapters, a FSM M and its graph representation G; the states of M and the vertices of G; the transitions of M and the edges of G are used interchangeably as long as it does not cause confusion.

3.2.3 Assumptions

In the following discussion, we assume that the control portion of a protocol entity is specified by a FSM (denoted henceforth by FSM_S). The FSM_S is assumed to be :

1) deterministic,

2) strongly connected,

3) completely specified, and

4) minimal .

We also assume that the IUT (denoted henceforth by FSM_i) is an implementation of the given FSM_S and that FSM_i is tested remotely over a communication network.

3.3 Experiments

This section reviews the concepts of machine identification and checking experiments, as well as the interelationships between these experiments and FSM-based testing.

3.3.1 Checking Experiment vs Machine Identification

Machine identification experiment [Koha78] is an experiment designed to construct a FSM which accurately represents the behavior of a given black box (implementation of some FSM) provided that input symbols and the upper bound of the number of states in the FSM are known.

Fault-detection (Checking) experiment [Koha78] is an experiment designed to determine whether a given FSM_i behaves in accordance with , or conforms to the given FSM_S. A checking experiment is a restricted case of the machine identification experiment [Koha78]. Its problem formalization is exactly the same as of conformance testing. This is why some of the checking experiment techniques can be applied to protocol conformance testing.

In general, a checking experiment consists of three phases :

1. Initialization phase

FSM_i is first brought to a specific state;

2. State identification phase

   FSM_i is checked whether it has n different states where n is the number of states in FSM_S;

3. Transition verification phase

   FSM_i is checked whether it implements each transition of FSMs correctly.

Each phase of a checking experiment is facilitated by input sequences derived from the FSM_S. The third phase of a checking experiment is based on a transition-level approach in which a test sequence is constructed to verify the correct implementation of each transition of FSM_S. Accordingly, the procedure of checking a transition $T_{jk} = (s_j, s_k; i/o)$ of FSM_S consists of three steps :

1) FSM_i is brought to state s_j;
2) Input i is applied to FSM_i and the output produced by FSM_i is checked to verify it is o;
3) The state FSM_i reaches after the application of input i is checked to verify that it is s_k.

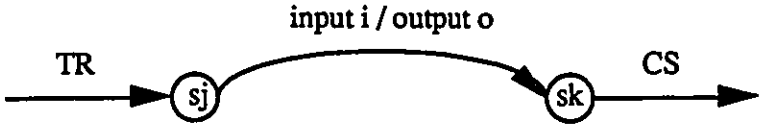


Figure 3.1 Definition of a Test

The input sequence required to test the transition $T_{jk} = (s_j, s_k; i/o)$ is denoted by $\mathbf{TEST_{jk}}$ and consists of

a) an optional transfer sequence $TR_x(j)$ (which is the shortest sequence of inputs to bring FSM_i from its current state x to state s_j);
b) input symbol i for stimulating transition T_{jk};
c) a checking sequence CS for realizing step 3) above.

The same transition-level approach above is adopted in FSM-based conformance testing such that a checking experiment is a test sequence (re. Figure 3.1).

3.3.2 Controllability and Observability Issues

In protocol conformance testing, the difficulty of realizing $TEST_{jk}$ is due to :

A) the limited controllability of FSM_i , so that it is not possible to bring FSM_i to s_j in step 1 without using a TR (with length longer than two input symbols), and

B) the limited observability of FSM_i, so that it is not possible to directly verify that FSM_i is in s_k in step 3.

These problems can be tackled by the following experiments :

i) Final state identification experiment (e.g., a Homing experiment, which together with a TR can solve problem A)) [Koha78].

ii) Initial state identification experiment (for solving problem B)) [Koha78];

Nevertheless, our approach to protocol conformance testing is to solve the **controllability** problem without using experiment i) by arranging the end of one test $TEST_{jk}$ as the beginning of another test $TEST_{kl}$.

3.4 Fault Models

There are four classes of faults (errors) which could exist in an FSM_i implementing FSM_S :

Output error :

A FSM_i is said to have output function (operation) errors , if FSM_i is not equivalent to FSM_S , and can be modified to be equivalent to the FSM_S by changing only the output function of FSM_i (without deleting or adding states in FSM_i).

State transition error :

A FSM_i is said to have state transition (transfer or tail state) errors if FSM_i is not equivalent to FSM_S and can be modified to be equivalent to FSM_S by changing only the next state function of FSM_i (without adding or deleting states in FSM_i).

Extra states (Missing states) :

A FSM_i is said to have extra (missing) states if in order to make FSM_i equivalent to FSM_S, the number of states in FSM_i must be reduced (increased).

Since both FSM_i and FSM_S are minimal machines, an unequal number of states implies that FSM_i and FSM_S are not equivalent.

In the following discussion, if no $\mathbf{FSM_S}$ (denotes a FSM specification) or $\mathbf{FSM_i}$ (denotes an implementation of the given FSM_S) is specified, **FSM** implies the specification only.

CHAPTER 4 THE FIVE FORMAL METHODS

In this chapter, several definitions from graph theory used in the remainder of the thesis are introduced and detailed descriptions of formal methods (i.e., T-method, D-method, W-method, Single UIO-method, and Multiple UIO-method) are given.

4.1 T-Method

4.1.1 Definitions and a Lemma Related to the T-Method

Given a strongly connected directed graph G(V, E), the indegree and outdegree of each vertex of G is defined as:

$d_{in}(v)^E = |\{(w, v; i/o) : (w, v; i/o) \in E\}|$

$d_{out}(v)^E = |\{(v, w; i/o) : (v, w; i/o) \in E\}|$

G is said to be **symmetric** if for every $v \in V$, $d_{in}(v)^E = d_{out}(v)^E$.

A **walk** w in G is a finite sequence of adjacent and not necessarily distinct edges. That is, $w = (v_1, v_2; L_1)\ (v_2, v_3; L_2)\ \ldots\ (v_{n-1}, v_n; L_{n-1})$ where vertex v_1 is called the **head** of the walk, denoted by HEAD(w) and vertex v_n is called the **tail** of the walk, denoted by TAIL(w).

The number of edges contained in a walk is called the **length** of the walk.

A **tour** in G is a walk in G which starts and ends at the same vertex of G.

An **Euler Tour** of G is a tour which contains every edge in E exactly once.

A **Chinese Postman Tour** (CPT) of G is *a shortest tour* which contains every edge in E at least once.

A connected graph is called a **tree** if it has no cycles [Hara72].

A **spanning tree** is a subgraph of G, which is a tree and contains all the vertices of G.

A **spanning arborescence** T is a spanning tree of G, in which each node $v \in T$ satisfies $d_{out}(v) = 1$ except the root r : $d_{out}(r) = 0$ [EdJo73].

Lemma 1 : A directed graph G contains an Euler tour *if and only if* G is strongly connected and symmetric [EdJo73].

4.1.2 The Definition of the T-Method

A Transition Tour (TT) of a FSM M is an input sequence which takes M from its start state s_S, executes every transition in M at least once , and returns M to state s_S. This method of obtaining a test sequence is called **T-method.**

T-method is the most straight forward among the five formal methods. Nevertheless, the simplicity of the approach is offset by its limited fault detecting capability : it can only detect output function errors. The state M reaches after the execution of a transition is not checked as specified in step 3) of the transition checking procedure mentioned in Section 3.3.1.

Previous approaches to generating a TT have been ad hoc which cause redundancies and thus result in TTs that are not minimal. Recently, some graph theoretic optimization techniques have been used in generating minimal test sequences [UyDa86]. The one which is applied to T-method is known as the **Chinese Postman Problem (CPP)** solution. CPP was first studied by a Chinese mathematician Mei-ku Kuan in 1962 as a problem of finding the shortest tour such that each street is traversed at least once [Kuan62]. We call this kind of tour a **Chinese Postman Tour (CPT).** Edmonds proposed an elegant algorithm for solving the CPP later in 1973 [EdJo73]. Our objective here is to generate a test sequence that covers each transition at least once in a manner that minimizes the total number of transitions traversed. This *total* is just the length of the TS (of input symbols which covers all the transitions of M). Consequently, finding a minimal TT of M is equivalent to the problem of finding a minimum-cost postman tour of G (representing M). In the following section, the algorithm proposed by Edmonds and Jonhson which computes a CPT in G will be described. We assume throughout the thesis that the cost of each edge of G is one.

4.1.3 Formulation of the Problem of Finding a Minimal TT

Given a strongly connected directed graph G(V,E) representing a FSM M, the problem of obtaining a minimal TT is equivalent to the problem of finding a CPT in G.

There are two cases which must be considered :

Case A : if G is symmetric, then finding a CPT of G is equivalent to finding an Euler tour of G. This is because the necessary and sufficient condition for G containing an Euler tour is that G is symmetric [Lemma 1 in Section 4.1.1].

Case B : if G is not symmetric, then every edge of E is contained in an optimal tour (CPT of G) at least once. That means that some edges of E may be contained more than once in the tour. Then, finding a CPT of G is reduced to : first, constructing a **symmetric augmentation** G* of G (i.e., a symmetric and strongly connected digraph G*) such that the number of replicated edges is minimized; and then, finding an Euler tour of G* which is a CPT of G.

The key issue of this method is to find a G* of G. As we have just mentioned in Section 4.1.1 and the above paragraph, finding a G* of G is actually the problem of duplicating some edges of E in such a way that ΣXe is minimized, where Xe denotes number of additional copies of an edge $e \in E$. Thus, the above problem can be formulated as :

minimize $Z = \Sigma(Ce*Xe),\ e \in E$

subject to

(1) $Xe \geq 0$, integer, and

(2) $\Sigma(Xe : v = HEAD(e)) - \Sigma(Xe : v = TAIL(e)) = b_v,\ v \in V.$

where

Ce denotes the cost associated with edge e (Since we assume the cost is unit for every edge of E, minimum-cost of Z becomes the goal of minimizing the total number of duplicated edges),

HEAD(e) means edge e emanating from vertex v,

TAIL(e) means edge e going into vertex v,

b_v denotes "$d_{in}(v) - d_{out}(v)$". The left hand side of condition (2) can be interpreted as "$d_{out}(v) - d_{in}(v)$". It is clear that since b_v is integer, the condition (1) that each Xe is an integer is guaranteed.

The above formula is a standard **linear programming** (LP) problem. It can be solved by corresponding techniques, e.g., minimum-cost maximum-flow algorithms since network flow problem is a special structured LP problem. Details of this problem will be given in Section 4.4.4.

Minimal TT Generation :

In general, to compute a minimal TS based on T-method consists of the following major steps :

1. computing duplicated edges such that ΣXe is minimized, where $e \in E$;
2. constructing G* by adding the duplicated edges calculated from step1 to G.
3. finding an Euler tour of G*, hence the CPT of G.

Step 3 includes two sub-steps :

3-1. constructing a **spanning arborescence** T of G* such that the root *r* of T is the start state s_S of M ;

3-2. traversing G* by beginning at r and following the rule that whenever vertex v_i is reached, leave v_i by the next unused outgoing edge from v_i, provided that the edges in T are traversed last in the order (i.e. only use $e \in T$ when there is no other edges emanating from v_i which have not been used, where v_i = HEAD(e)). All the other edges which are not in T are traversed in an arbitrary order in the tour [EdJo73]. Since the edges in T are always traversed last, it is guaranteed that the tour returns to its starting state r.

An example of generating a minimal TT is given in the next section.

4.1.4 An Example

Consider the FSM M in Figure 4.1. M is deterministic, strongly connected, completely specified and minimal.

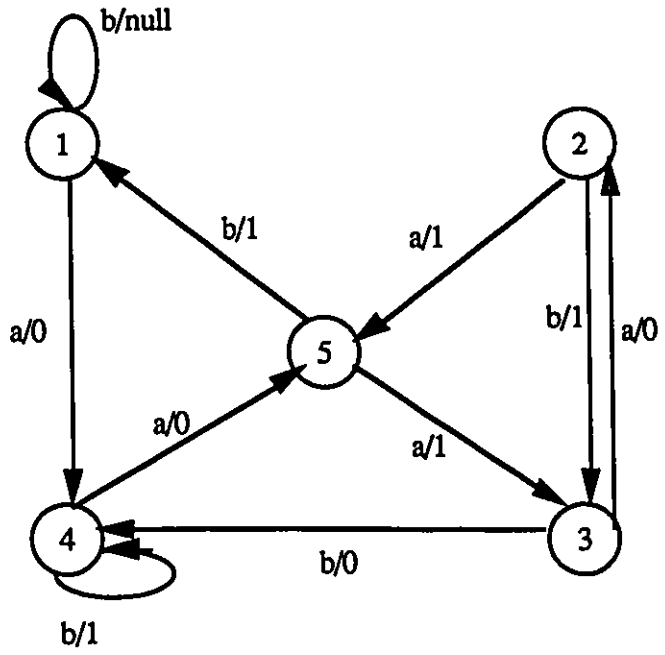


Figure 4.1 A Graph Representation (G) of a FSM M (All edge costs are equal, *reset* edges are not shown).

Application of the T-method :

step 1: Formulate the TT problem as a standard LP problem :

min Z $= \Sigma(Xe_i: e_i \in E)$, where i = 1, 2, ..., 10 (where 10 is no. of edges in G)

(1) $X_{14} - X_{51} = 0$

(2) $X_{23} + X_{25} - X_{32} = -1$

(3) $X_{32} + X_{34} - X_{23} - X_{53} = 0$

(4) $X_{45} - X_{14} - X_{34} = 1$

(5) $X_{51} + X_{53} - X_{25} - X_{45} = 0$

Solution of the above LP problem is :

$X_{32} = X_{45} = X_{53} = 1$; all the rest of $X_{ij} = 0$; and $Z = 3$.

step 2: Add edges e_{32}, e_{45} and e_{53} to G to obtain a symmetric augmentation G* of G as shown in Figure 4.2;

step 3: Find an Euler tour of G* :

a) find a spanning arborescence T of G* as shown in Figure 4.3.

b) find an Euler tour of G* :

An Euler tour of G* is a CPT of G as depicted in Table 4.1.

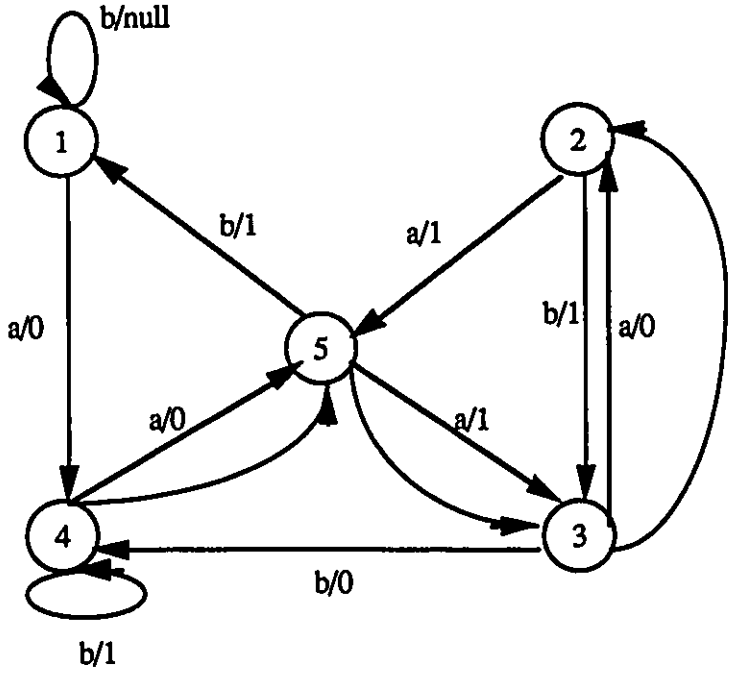


Figure 4.2 A Symmetric Augmentation G* of G in Figure 4.1

(The unlabeled arcs are the additional edges added to G).

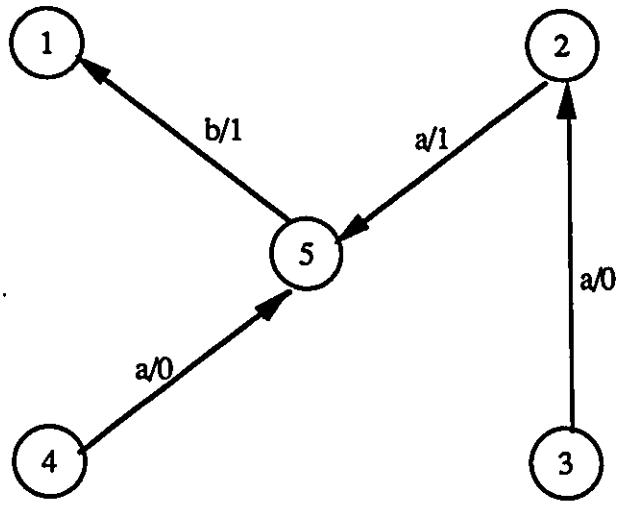


Figure 4.3 A Spanning Arborescence T of the Graph G* in Figure 4.2

I/O sequence :	a/0, b/1, a/0, a/1, b/0, a/0, a/1, a/0, b/1, a/0, a/1, b/1, b/null														
State sequence :	1	4	4	5	3	4	5	3	2	3	2	5	1		1
TT :		a	b	a	a	b	a	a	a	b	a	a	b	b	

TS length = 13

Table 4.1 A Minimal Test Sequence for the FSM in Figure 4.1

4.1.5 About Status Message

In certain protocol implementations, there is a desirable feature called *status message*. It is a single message which identifies each specified state s_i of M by producing a unique output for s_i and causes M to remain at state s_i. For example, in the case of an IUT in which the tester can access the state of the implementation, the status message would be : "respond with the state of the IUT". In this case, step3 of the transition-level testing approach can be easily incorporated into the transition tour method by using the "status" input to verify the state after each transition is checked for the first time [ADLU88]. The observability problem therefore is solved in this special case. Note that the status message

is also checked as an ordinary transition for the first time it appears in the checking procedure. The initial *reset* applied before the test of IUT starts is not verified.

As an example, a transition tour generated for the FSM in Figure 4.1 with *status message* (depicted in Figure 4.4) is listed below :

ri/null, a/0, status/4, status/4, b/1, status/4, a/0, status/5, status/5, ri/null, status/1, a/0, ri/null, status/1, a/0, a/0, a/1, status/3, status/3, b/0, status/4, a/0, a/1, ri/null, status/1, a/0, a/0, a/1, a/0, status/2, status/2, ri/null, status/1, a/0, a/0, a/1, a/0, b/1, status/3, a/0, a/1, status/5, b/1, status/1, status/1, b/null, status/1, ri/null, status/1.

TS length = 49.

For implementations which have no "status" message , other checking sequences such as Unique Input/Output (UIO)sequences, Distinguishing Sequences (DS) and Characterizing set (W-set) can be used to check the state transition function in step3 of the transition checking procedure. In this case, the observability problem is solved, and the transition-level approach for testing the transition $T_{jk} = (s_j, s_k; i/o)$ can be applied as follows :

step 1. FSM_i is brought to state s_j;

step 2. input i is applied to FSM_i and the output produced by FSM_i is checked to verify it is o as expected;

step 3. the state FSM_i reaches after i is applied is checked by UIO_k , DS or W-set , to verify it is s_k as expected

In the following sections, the other four formal methods of conformance test sequence generation are described. These four formal methods (D-method, W-method, Single UIO-method and Multiple UIO-method) are closely associated with Distinguishing sequences, Characterizing sets and UIO sequences, respectively. The test sequences generated by these methods all have the capability of detecting output errors and state transition errors [ADLU88, SiLe86, SLD89].

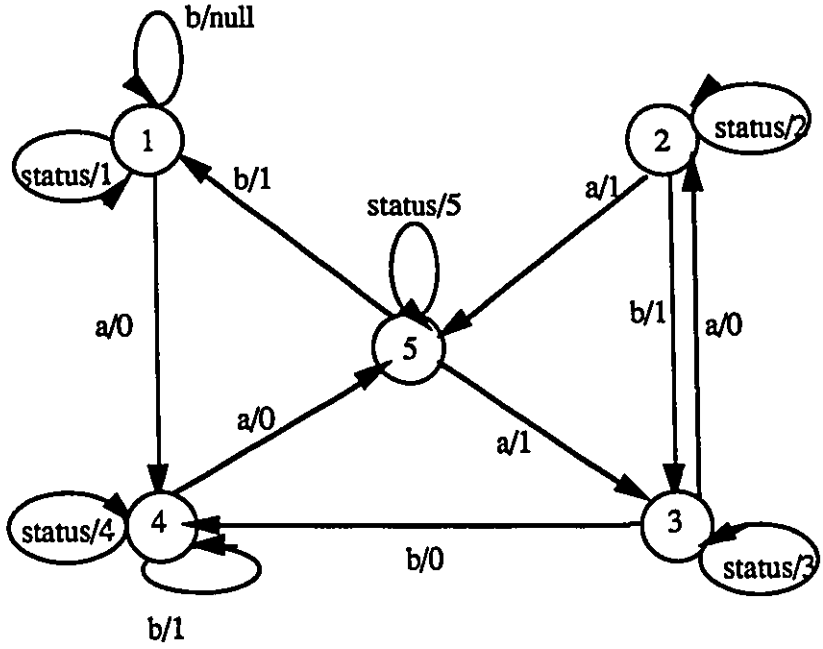


Figure 4.4 A FSM with *Status* Message (*Reset* edges are not shown).

4.2 D-Method

4.2.1 The Definitions Related to the D-Method

A **path** $P = (v_1, v_2; L_1)\ (v_2, v_3; L_2)\ \ldots\ (v_{n-1}, v_n; L_{n-1})$, $n > 1$, is a sequence of adjacent edges.

A path is **simple** if it does not use the same edge twice.

A **covering** is a partition of all the edges of a graph into simple paths which are disjoint (i.e., having no common edges).

A **minimal covering** of G is a covering of G which involves the fewest possible simple paths [BuSa65].

Given a strongly connected directed graph G'(V', E'), an **edge-induced subgraph G[Ec] of G'** is the subgraph of G' whose vertex set is the set of terminal vertices of edges

in Ec, and whose edge set is Ec, where E' $\supseteq$ Ec [ADLU88]. G[Ec] is called an **edge-induced spanning subgraph of G'** if its vertex set is V'.

4.2.2 The Definition of the D-Method

This method is designed for a FSM M which possesses a Distinguishing Sequence (DS). An input sequence is a DS of a FSM M if the output sequence produced by M in response to the DS is unique for each state of M. For example, the sequence "bb" is a DS for the FSM in Figure 4.1. The sequences of outputs produced by the FSM in response to the DS for each state are listed in Table 4.2.

The main idea of the D-method proposed by Gonenc is to construct an *input*@DS-diagram, where *input* $\in$ I, "@" means concatenation of two strings and then, to find a minimal covering of the *input*@DS-diagram, which constitutes a test sequence for M [Gone70].

STATE	DS/Output
1	null null
2	1 0
3	0 1
4	1 1
5	1 null
DS = bb	

Table 4.2 A DS-Table for the FSM in Figure 4.1

4.2.3 The Procedure for the D-Method

The D-method of deriving a test sequence (TS) involves the following steps:

1. compute a DS of minimum length (note that there may exist more than one DS for a FSM);
2. draw an i@DS-diagram denoted by G[Ec] = (V, Ec), where
   Ec = {(v_j, v_l; i@DS) : (v_j, v_k; i/o) ∈ E, i ∈ I, o ∈ O, and TAIL(DS) = v_l ∈ V};
3. find a minimal covering of G[Ec], it is the resulting test sequence.

In the next section, we will provide an algorithm for each step of the D-method.

4.2.4 The Algorithms

Given a FSM M, assume that M possesses at least one DS. The algorithms for each step in Section 4.2.3 are given as follows:

Computing a minimum-length DS :

A minimum-length DS can be obtained by constructing a Distinguishing-tree (DS-tree)[Koha78];

Constructing i@DS-diagram G[Ec]:

G[Ec] is a directed graph constructed by applying successive cells of the form **i@DS**, for each i ∈ I at each state of M. The output and state transition functions are thus checked. Traversing such a graph corresponds to realizing the test of M. A **path** in G[Ec] corresponds to successive application of cells to M.

An example i@DS-diagram of M in Figure 4.1 is given in Figure 4.5. As we can see that the new graph G[Ec] has the same number of states and edges as that in G (representing M) in Figure 4.1 since every transition of M should be checked.

Finding a minimal covering of G[Ec] :

First, we define three sets:

$R = \{ v \in V \mid d_{out}^{Ec}(v) = d_{in}^{Ec}(v) \}$

$F = \{ v \in V \mid d_{out}^{Ec}(v) > d_{in}^{Ec}(v) \}$

$P = \{ v \in V \mid d_{out}^{Ec}(v) < d_{in}^{Ec}(v) \}$.

If G[Ec] is strongly connected and R = V, viz., G[Ec] is symmetric. According to the Lemma 1, an Euler tour exists and the Euler tour is the unique (in terms of length) minimal covering of G[Ec]. This problem can be solved by using the same approach as the one suggested for the T-method.

If $R \neq V$, for either a weakly or strongly connected digraph G[Ec], there are k paths in every minimal covering of G[Ec], where

$k = \Sigma(d_{out}^{Ec}(v) - d_{in}^{Ec}(v) : v \in F) = \Sigma(d_{in}^{Ec}(v) - d_{out}^{Ec}(v) : v \in P)$ and

each path joins a vertex in F to one in P [BuSa65]. A minimal covering of G[Ec] can be found by following the following rules:

1) start from a state in F,
2) each time an edge is followed, erase it,
3) do not use an edge which is an isthmus (i.e, an edge whose deletion divides the graph into two components), if there exists any other edge which can be followed.
4) when it is not possible to go any further, start again from a state in F in such a way that eventually from each state $v_k \in F$ exactly λ_k starts will be made, where

   $\lambda_k = d_{out}^{Ec}(v_k) - d_{in}^{Ec}(v_k)$.

   To reach another restart $v_k \in F$, a "jump" , that is a $TR_j(k)$ must be used, where j represents current simple path terminating state v_j , and $v_j \in P$. TR's are selected from G.

Since there are k paths, the total number of TR's needed is (k-1).

Note that in step 4), there is the problem of choosing the state from which a restart is to be made. Suppose that the minimal covering reaches state v_j, then always chose the restarting state, say v_k, which requires the shortest $TR_j(k)$, and there is at least one unerased edge emanating from v_k.

As an example, consider the G[Ec] in Figure 4.5 for M in Figure 4.1,

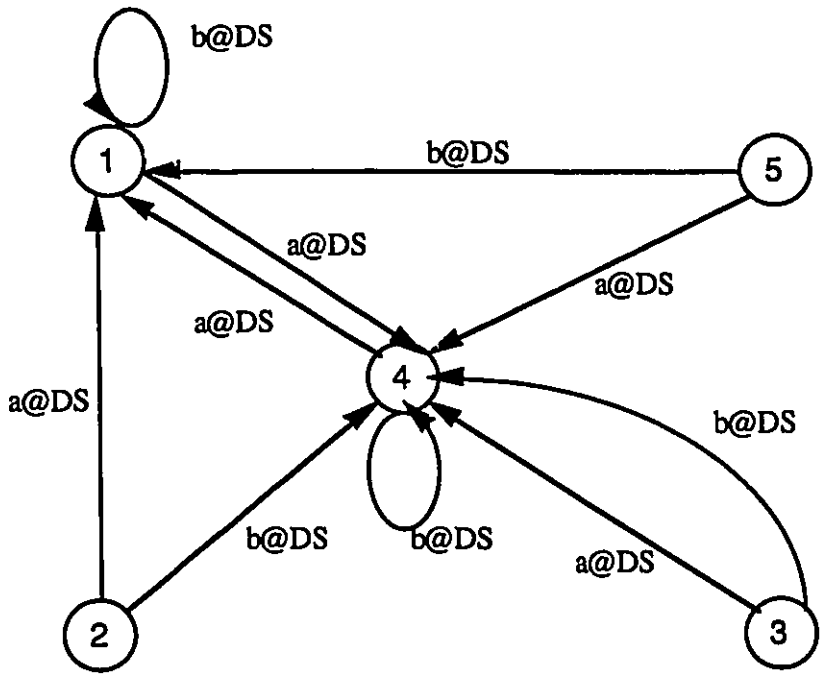


Figure 4.5 The input@DS-diagram G[Ec] of the FSM in Figure 4.1

$R = \text{null}$, $F = \{2,3,5\}$, $P = \{1,4\}$, $k = \lambda_2 + \lambda_3 + \lambda_5 = 2 + 2 + 2 = 6$.

Hence, the minimal covering consists of six paths and five TR's(for "jumps") if the test sequence starts from state 2, 3 or 5 in F.

One of the possible minimal coverings is (state sequence only) :

2 4 - 2 1 1 4 4 1 - 3 4 - 3 4 - 5 4 - 5 1

where the hyphens represent TR's for the "jump"s between distinct paths.

A test sequence based on the minimal covering is :

b@DS, aaa, a@DS, b@DS, a@DS, b@DS, a@DS, aaa, a@DS, aa, b@DS, a, a@DS, a, b@DS

The total length of the test sequence is 40.

In our implementation of this method, the outputs correspond to the TS are also generated together with the TS to form a Test Sequence Table.

4.2.5 Problems

There are some drawbacks for this method. First, since very few real protocols have distinguishing sequences, the applications of the D-method in protocol conformance testing is very limited. Second, the i@DS-diagram (G[Ec]) may not be weakly connected. In this case, the following heuristic approaches are suggested :

a) find edges from G to add to G[Ec] until G[Ec] is weakly connected. Then the above algorithms can be applied to find a minimal covering of the weakly connected G[Ec];

b) test each weakly connected component separately. That is to generate and apply the tests for each of these components respectively. It is clear that the above algorithm is valid for each weakly connected component.

Third, since there are no constraints for terminating a minimal covering, the resulting TS may not end with the start state s_S where the test starts. The minimal covering of the example shown in Section 4.2.4 is an example of this case. In order to bring M to its start state s_S (state2), a TR can be used if M has no reset feature. Also, a Homing sequence (HS) [Koha78] followed by a TR can be used to identify the TAIL(HS) and to bring M to its s_S from TAIL(HS). Note that a HS always exists for a minimal FSM.

It can be readily seen that Gonenc's proposal for finding a minimal covering has many similarities with the approach used in the T-method if the connectivity assumption holds for G[Ec] in the D-method. Notice that a difference between the two methods in terms of the ways of traversing a graph is that instead of finding all the TR("jump")s randomly along the way of finding a minimal covering in the D-method, they are preset through solving a LP problem in the T-method. This leads us to consider the possibility of applying the solution of CPP to the D-method in order to generate a minimal test sequence systematically. Later in Chapter 5, we will give an extension of the D-method based on the application of the solution of the CPP.

4.3 W-Method

4.3.1 The Definition of the W-Method

A set of input sequences which can distinguish the behaviors of every pair of states in a minimal FSM is called **W-set** and denoted by "**W**". In this method, a W-set is applied after each transition to confirm whether the tail state of the transition is as expected. Hence, it is called **W-method** [Chow78].

Every minimal FSM has a W-set, therefore, for FSM's which do not possess distinguishing sequences, W-method is an alternative method for generating test sequences. However, this method yields longer test sequences compared with the other formal methods in most cases [SiLe86].

The test sequences generated by W-method have the capability of detecting output errors, state transition errors, extra states and missing states errors.

4.3.2 The Procedure for the W-Method

Given a FSM_S M with n states and its implementation FSM_i with m states, W-method of generating a test sequence consists of four major steps :

1) compute a minimal W-set **W**:

   a minimal W-set can be found by applying a Multiple-experiment on M [Gill62].

2) estimate the maximum number of states in IUT (FSM_i);

   compute a characterizing set **Z** such that

   $Z = W \cup I@W \cup I^2@W \cup \ldots \cup I^{m-n}@W$, where I is the input symbols of M;

3) construct a **Testing-Tree T**, and obtain the set **P** of all partial paths in T that start at the root of T;

4) concatenate P and Z to obtain a test sequence.

Estimating the Maximum Number of States in the FSMi :

Since FSM_i is tested as a black box, whether it has extra (missing) states is unknown to the tester. The estimation of m is usually based on the FSM_S and available knowledge about FSM_i. If the estimation is correct, test sequences generated by the W-method can detect up to $(m-n)$ extra states in FSM_i.

Characterizing set $Z = W \cup I@W \cup I^2@W \cup \ldots \cup I^{m-n}@W$, is a set of input sequences, where U is the union operator, A@B denotes the concatenation of the sets A and B, and $A^0 = \text{null}$, $A^{i+1} = A@A^i$.

A Procedure of Constructing a Testing-tree T :

(1) label the root of T with the s_S of M. This is level 1 of T;

(2) suppose we have already built T to a level k, the (k+1)th level is built by examining nodes in the kth level from left to right. A node at the kth level is terminated if its label is the same as a nonterminal node at some level j, $j \leq k$. Otherwise, let s_j denote its label. If on input i, machine M goes from states s_j to state s_k, attach a branch with input i with its successor node s_k to node s_j in T.

The above procedure always terminates, since there are only a finite number of states in M. Also, depending on the left to right order in which the successor nodes are placed, a different tree may result.

A **partial path** in T is a sequence of consecutive branches. It starts at the root of the tree and ends at either a terminal or a nonterminal node. It is actually a sequence of inputs which takes M from its s_S (viz. the root of the testing tree T) to a particular state s_j which is the head state of the transition to be tested next. A set P of all partial paths in T can be found by building a test-tree T as we mentioned in the above procedure (1) and (2). An *empty* sequence is also considered a member of P.

The test sequence is the concatenation of set P and Z , where $Z = \{z_1, z_2, \ldots, z_k\}$, each z_i $(1 \leq i \leq k)$ is a sequence of consecutive input symbols which are constructed in the way stated in step2 above and $P = \{p_i \mid p_i \text{ is a partial path in T}\}$.

A complete test sequence (TS) is :

$$TS = \#_{i=0}^{r} \#_{j=1}^{k} (ri\ @P_i@z_j) \quad , r = |E|, \quad p_0 = null,$$

where "ri" stands for reset function, "#" denotes string concatenations.

In the case where M has no reset function, a transfer sequence TR is generated to replace each "ri".

The method described above can be considered as checking whether the FSM_i is P@Z equivalent to the FSM_S, provided that the estimation of m (step2, number of states in FSM_i) is correct [Chow78]. Since both FSM_S and its FSM_i are minimal, for a correct FSM_i, m should be equal to n (number of states in FSM_S).

4.3.3 An Example

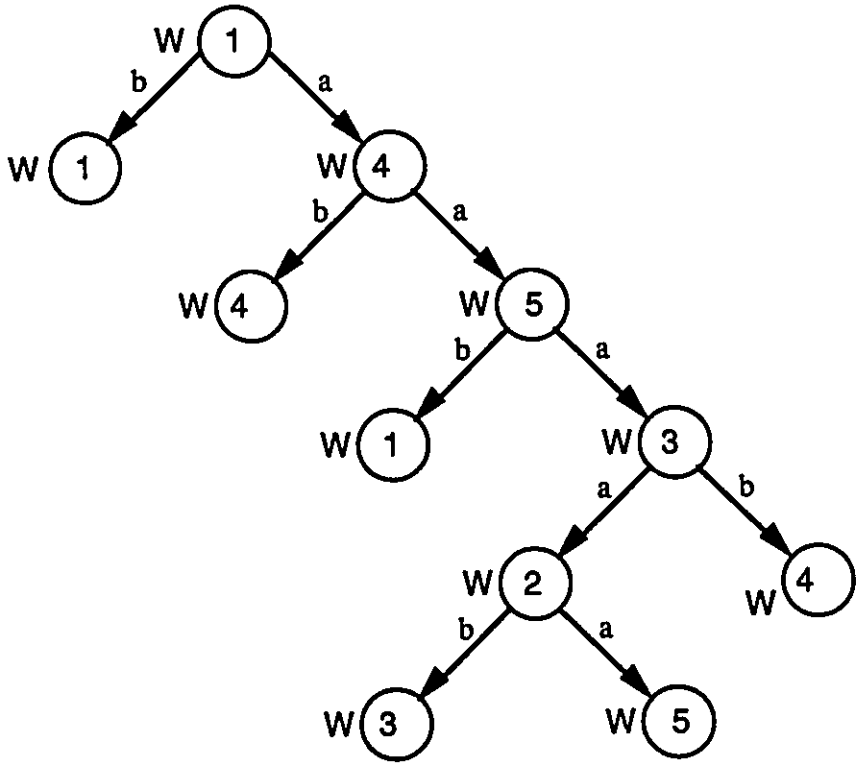
A FSM M as an example is depicted in Figure 4.1. As has been stated previously, M satisfies the four conditions, therefore possesses a W-set . Furthermore, we assume that M has a reset function.

1) a W-set is found by applying a Multiple-experiment on M [Gill62]:

   W = {a, aa, b};

2) let m = n. Then Z is reduced to a W-set ;

3) build a Testing-Tree T and attach the W-set to each node of the tree T :



where W = { a, aa, b }.

Figure 4.6 A Testing-Tree T of the FSM in Figure 4.1

4) a set of test sub-sequences :

W
a@W
b@W
aaaaa@W
aaaab@W
aaaa@W
aaab@W
aa@W
ab@W
aaa@W
aab@W

Table 4.3 (a) A Set of Test Sub-sequences

A complete test sequence is :

ri a	ri aaaba
ri aa	ri aaabaa
ri b	ri aaabb
ri aa	ri aaa
ri aaa	ri aaaa
ri ab	ri aab
ri ba	ri aba
ri baa	ri abaa
ri bb	ri abb
ri aaaaaa	ri aaaa
ri aaaaaaa	ri aaaaa
ri aaaaab	ri aaab
ri aaaaba	ri aaba
ri aaaabaa	ri aabaa
ri aaaabb	ri aabb
ri aaaaa	
ri aaaaaa	
ri aaaab	

TOTAL = 167 inputs

Table 4.3 (b). A Test Sequence for the FSM in Figure 4.1

4.4 Single UIO-Method

4.4.1 Overview of the Single UIO-Method

In the SUIO-method [SaDa88], a graph traversal technique for constructing Rural Chinese postman Tour (RCPT) is applied in combination with UIO sequences to obtain a minimum-length test sequence for an FSM which has reaset or self-loop feature. An input/output behavior which is unique for a state s_j (i.e., not exhibited by any other state) of a given FSM M is called a UIO sequence for s_j, and denoded by UIO(s_j) or UIO_j. Here, by a UIO sequence of a state, we mean either a Unique Input/Output Sequence or a Unique

Signature for that state [SaDa88]. A UIO sequence is applied to check whether M reaches the expected state after the execution of each transition as specified in step 3) of the transition checking procedure mentioned in Section 3.3.1.

4.4.2 Rural Chinese Postman Problem and Minimal Test Sequences

Given a directed graph G'(V', E') with nonnegative costs on the edges in E', a **Rural Chinese Postman Problem** is to find a minimum-cost tour, **Rural Chinese Postman Tour (RCPT)**, containing a set $E' \supseteq E^*$ of required edges [LeKa76]. Computing such a tour is known to be NP-complete in the most general case [LeKa76]. Nevertheless, if the edge-induced spanning subgraph G[Ec] of G' is weakly connected, a polynomial algorithm exists. [ADLU88] shows that as long as a protocol possesses reset or self-loop feature, the corresponding G[Ec] is weakly connected. Furthermore, the symmetric directed graph G*(V*, E*) obtained by adding edges from G(V, E) to G[Ec] is guaranteed to be strongly connected, therefore G* yields an Euler tour which is a RCPT of G. The RCPT of G is a minimum-cost test sequence (hereinafter called **a minimal test sequence**) for the given FSM represented by the digraph G.

4.4.3 Network Flow Problem and Edge Duplications

This section reviews some concepts and definitions from network flow problems. Two sufficient conditions for a subgraph G[Ec] of G' to be connected are given. It is a prelude to solve the RCPP by means of relating the edge duplication problem in the RCPP to the network flow problems.

Networks are directed graphs which are connected and have no self-loops.

Given a network G_f (V_f, E_f), a **flow** in G_f is a nonnegative integer valued function F defined on E_f. The integer F(e) is called the flow on edge e.

A flow in G_f satisfies the following conditions (each edge label is as it is defined before):

(1) $0 \le F(v_j, v_k; i/o) \le capacity(v_j, v_k; i/o)$

(2) for every $v_j \in V_f - \{s, t\}$,

$$0 = \sum_k F(v_j, v_k; i/o) - \sum_k F(v_k, v_j; i/o)$$

where

$(v_j, v_k; i/o)$ and $(v_k, v_j; i/o) \in E_f$, and

s and t are *source* and *sink* nodes of G_f, respectively.

The *total flow F* is evaluated at sink t [Even79] :

(3) $$F = \sum_j F(v_j, t) - \sum_j F(t, v_j)$$

A *maximum flow* with *minimum-cost* is called a **minimum-cost maximum flow** (*min-cost max flow* for short) in G_f.

Recall that finding a minimum-length transition tour was reduced to finding the CPT of G in Section 4.1. Finding a CPT is then reduced to the problem of edge duplications in G so that a symmetric augmentation G* of G is obtained in order to form a CPT of G. The duplication problem can be solved by solving a min-cost max flow problem. A similar approach is used in the Single UIO-method.

Let G'(V', E') be a strongly connected digraph such that V' = V and E' = E U Ec, where Ec is a set of edges each of which is a concatenation of the form *transition@UIO*. Its corresponding digraph is denoted by G[Ec]=(V, Ec). A digraph G*(V*, E*) is called a **Rural Symmetric Augmentation (RSA)** of G'(V', E'), if G* is symmetric and strongly connected, where V* = V', E*= Ec U{e : e∈E, Xe > 0}. Xe is the number of copies of edge e ∈ E. Accordingly, an Euler tour of G* is the RCPT of G' which in turn, is a minimal test sequence for G.

A necessary condition for there to exist an Euler tour in G* is that G[Ec] must be weakly connected. The following two lemmas and a theorem provide sufficient conditions for a G[Ec] to be weakly connected, and for G* to contain an Euler tour.

Lemma 2 : If a FSM M represented by G has the reset capability then, the edge-induced subgraph G[Ec] of G' is a spanning subgraph of G' and is weakly connected [ADLU88].

Lemma 3 : If the graph G of a FSM M is strongly connected and for each vertex $v_j \in V$ there exists an edge $(v_j, v_j; i/o) \in E$ then the edge-induced subgraph G[Ec] of G' is a spanning subgraph of G' and is weakly connected [ADLU88].

Theorem : If the edge-induced subgraph G[Ec] is a weakly connected spanning graph of G' then any rural symmetric augmentation G* of G' is strongly connected and therefore has an Euler tour [ADLU88].

4.4.4 The Procedure for the Single UIO-Method (SUIO-Method)

We explain in this section how to derive a minimal test sequence (TS) by using network flow algorithms.

Consider a directed graph G(V, E) which represents a FSM M. Assume that M has reset capability. Then, there are five steps to compute a minimal TS by using the SUIO-method :

step 1 : compute a minimum-length UIO for each state of M [SaDa88];

step 2 : construct graph G'(V', E') such that,

$V' = V$, $E' = E \cup Ec$

where

$Ec = \{(v_j, v_l; i/o@UIO_k) : (v_j, v_k; i/o) \in E , TAIL(UIO_k) = v_l\}$

the cost of $(v_j, v_l; i/o@UIO_k)$ is defined as

$C(v_j, v_l; i/o@UIO_k) = C(v_j, v_k; i/o) + C(UIO_k)$;

step 3 : reduce the duplication problem to the standard network flow problem by calculating the *index* at each vertex of G[Ec] of G' and constructing the network model by using the indices of G[Ec] of G' as capacities at source and sink, and the structure of G as

the structure of the network. A detailed algorithm will be given next after the discussion of the derivation of UIO-sequences.

step 4 : construct a rural symmetric augmentation G* (V*, E*) of G' by adding duplicated edges calculated in step3 into graph G';

step 5 : find a Euler tour in G*, it is a RCPT of the graph G'.

A minimal TS of G is thus obtained.

UIO-sequences :

An algorithm to calculate a minimum-length UIO sequence for each state of M is given in [SaDa88]. For each state which does not possess a UIO sequence, a Unique Signature (US) is computed. The idea is originated from W-set. Since M is minimal, there exists an I/O sequence IO(j,k) with length less than n which can distinguish s_j from a state s_k (it is called a diagnozing sequence of s_j and s_k [Gill62]). To compute US(s_j), start with IO(j,1) which distinguishes s_j from s_1. Let M end up in state s_{j-1} after IO(j, 1) is applied at s_j. Then a transfer sequence $TR_{j-1}(j)$ is used to bring M back to s_j from s_{j-1}. Next, IO(j, 2) is applied to distinguish s_j from s_2, and so on. The following sequence is defined as US(s_j)'s:

$$US(s_j) = \begin{cases} IO(j, 1)@ \#_{k=2}^{n} TR_{k-1}(j)@IO(j, k), & \text{where } k \neq j \text{ for } j \neq 1 \\ IO(1, 2)@ \#_{k=3}^{n} TR_{k-1}(1)@IO(1, k), & j = 1 \end{cases}$$

Where IO(j, k) distinguishes s_j from s_k, and $TR_k(j)$ is a transfer sequence which brings M back to s_j from the state in which the machine enters after the application of IO(j, k). Figure 4.7 shows an example FSM. State1 of the FSM does not have a UIO sequence.

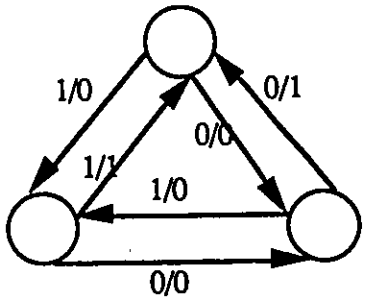


Figure 4.7 An Example FSM

Thus, a US(1) is calculated for state1. The other two UIO's for state2 and state3 are listed together with US(1) in Table 4.4.

US(1) : 0/0, 0/1, 1/0
UIO(2) : 0/1
UIO(3) : 1/1

Table 4.4 A UIO-Table for the FSM in Figure 4.7

In G' (step 2), traversing an edge (v_j, v_l; i/o@UIO_k)$\in$ Ec corresponds to realizing $TEST_{jk}$ in G. Since G[Ec] is a weakly connected spanning subgraph of G' [Lemma 3], a complete tour which covers every edge in Ec will be a complete test sequence for M. There are two cases in finding a RCPT in G :

Case A : if G[Ec] is symmetric (which is a very rare case in SUIO-method) and strongly connected, then no additional edges are needed. In this case, G* = G[Ec] of G'= (V, Ec). An Euler tour exists in G[Ec] which is the minimal test sequence;

Case B : if G[Ec] is not symmetric, then some edges in E need to be replicated in forming a G* of G'. Again it is an edge duplication problem which can be solved by applying the network flow algorithms. According to [ADLU88], every RSA G* of a weakly connected digraph G[Ec] is strongly connected.

Solving the Edge Duplication Problem :

Let graph $G_f = (V_f, E_f)$ be the standard network flow model constructed from G' as follows (ignoring all self-loops at each vertex of G'):

For a vertex $v_i \in G'$, let $\xi(v_i) = d_{in}^{Ec}(v_i) - d_{out}^{Ec}(v_i)$. $\xi(v_i)$ is called the **index** of vertex v_i. If $\xi(v_i) = 0$, for all vertices of G', then E* = Ec. That is, G* = G[Ec] (G[Ec] is symmetric). An Euler tour can be found in G[Ec] which is a minimal test sequence of G. If $\xi(v_i) \neq 0$ for some vertices of G', then a network flow model G_f is constructed as follows:

$V_f = V' \cup \{s, t\}$, where s and t are the source and sink of G_f

$E_f = E' \cup \{(s, v_i) : v_i \in C\} \cup \{(v_j, t) : v_j \in D\}$

where $V_f \supset C$ ($V_f \supset D$) is the set of vertices in G' with positive (negative) indices. In this way, all the vertices with positive indices are connected to the source s, and all the vertices with negative indices are connected to the sink t of G_f. The rest of the vertices are connected in the way that they are in G.

Determining the Cost and Capacity of Every Edge in G_f :

Zero cost is assigned to every edge (s, v_i) in G_f with the flow capacitiy of $\xi(v_i)$, where $\xi(v_i) > 0$, $i = 1, 2, \ldots, n$;

Zero cost is assigned to every edge (v_i, t) in G_f with flow capacity of $-\xi(v_i)$, where $\xi(v_i) < 0$, $i = 1, 2, \ldots, n$;

The rest of edges in G_f are with unit cost and infinite flow capacities.

The total flow at the sink t equals to the total number of duplicated edges needed going into those vertices with negative indices in order to obtain a G* of G', while all the individual flows which are assigned to those intermediate edges are the number of copies of those edges required for a G* of G'. Finding the flow in G_f follows the principle of choosing the shortest path between s and t with maximum amount of feasible flows along that path. When flows are saturated at sink t, the min-cost max flow is found which is the optimal solution to the edge duplication problem.

Constructing a G* of G' :

Given a min-cost max flow F on G_f, let $F(v_j, v_k; i/o)$ denote the number of copies of edge $(v_j, v_k; i/o) \in E$ needed to construct a RSA G* of G'. We define the following function :

$$\chi(v_j, v_k; i/o) = \begin{cases} 1 & \text{if } (v_j, v_k; i/o) \in Ec \\ F(v_j, v_k; i/o) & \text{if } (v_j, v_k; i/o) \in E \end{cases}$$

Function χ corresponds to the construction of RSA G* of G' by repeating the edge $(v_j, v_k; i/o)$ in E $F(v_j, v_k; i/o)$ times, and including every edge of Ec in G* exactly once.

Finding an Euler Tour of G* (it is a RCPT of G') :

Following the same procedure as it is stated in Section 4.1.3 (in T-method).

4.4.5 An Example

Consider the FSM M in Figure 4.8. Every state of M has a UIO sequence, and they are listed in Table 4.5(a). The graph G' for G (or M) is depicted in Figure 4.9. A network flow model G_f for the graph G' in Figure 4.9 is given in Figure 4.10. There are eight units of flow in the network G_f. Total number of replicated edges of E is 11. Since the total number of inputs is 43 in Ec, the length of a minimal test sequence is 55 (including an initial reset input). The rural symmetric augmentation G* of G is shown in Figure 4.11. A minimal test sequence is listed in Table 4.6.

UIO(V1) : b/x, a/x

UIO(v2) : b/y

UIO(v3) : b/x, c/z

UIO(v4) : b/x, c/y

UIO(v5) : c/z

Table 4.5(a) UIO Sequences for the States of the FSM in Figure 4.8

The following table lists all the possible edges in Ec of graph G', each of which is a concatenation of a specified i/o for transition $T_{jk} = (v_j, v_k; i/o)$ with the $UIO(v_k)$.

TRANSITION	i/o@UIO(TAIL(i/o))	TAIL
T_{11}	ri/null b/x a/x	v5
T_{11}	a/x b/x a/x	v5
T_{14}	c/y b/x c/y	v5
T_{12}	b/x b/y	v3
T_{21}	ri/null b/x a/x	v5
T_{23}	b/y b/x c/z	v1
T_{25}	a/x c/z	v1
T_{31}	ri/null b/x a/x	v5
T_{35}	b/x c/z	v1
T_{35}	c/y c/z	v1
T_{41}	ri/null b/x a/x	v5
T_{43}	b/x b/x c/z	v1
T_{45}	a/x c/z	v1
T_{51}	ri/null b/x a/x	v5
T_{51}	c/z b/x a/x	v5
T_{54}	a/z b/x c/y	v5

Table 4.5(b) A Transition@UIO-Table for the FSM in Figure 4.8

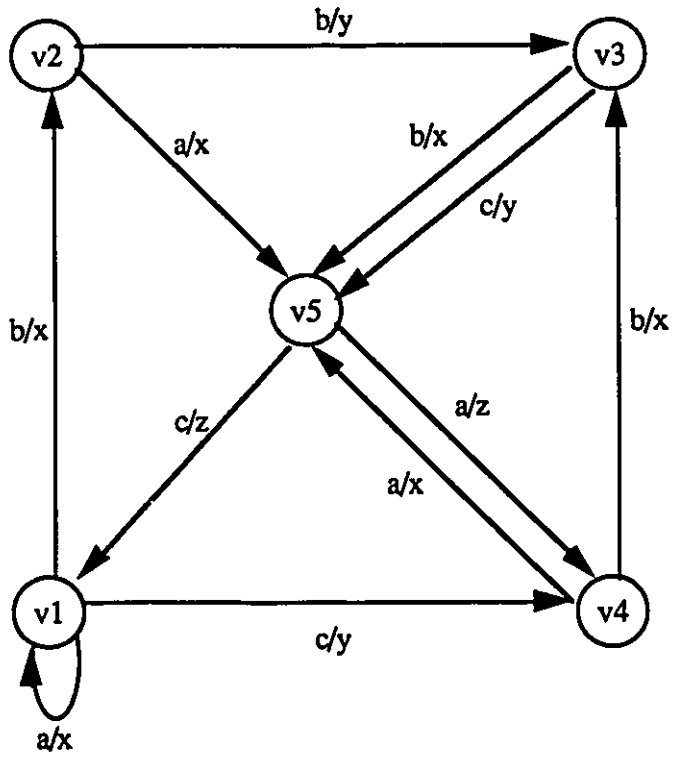


Figure 4.8 A graph representation G of a FSM M

(*reset* edges are not shown).

ri/null	b/x b/y	c/y c/z	a/x b/x a/x
a/z b/x c/y	c/z b/x a/x	ri/null b/x a/x	a/z
a/x c/z	c/y b/x c/y	a/z	b/x
b/x c/z	b/x	b/y b/x c/z	b/x
a/x c/z	b/x	ri/null b/x a/x	a/z
b/x	ri/null b/x a/x	a/z	b/x b/x c/z
ri/null b/x a/x	a/z	ri/null b/x a/x	c/z

TOTAL = 55 inputs

Table 4.6 A Minimal Test Sequence for the FSM in Fig 4.8

(To be read from left to right).

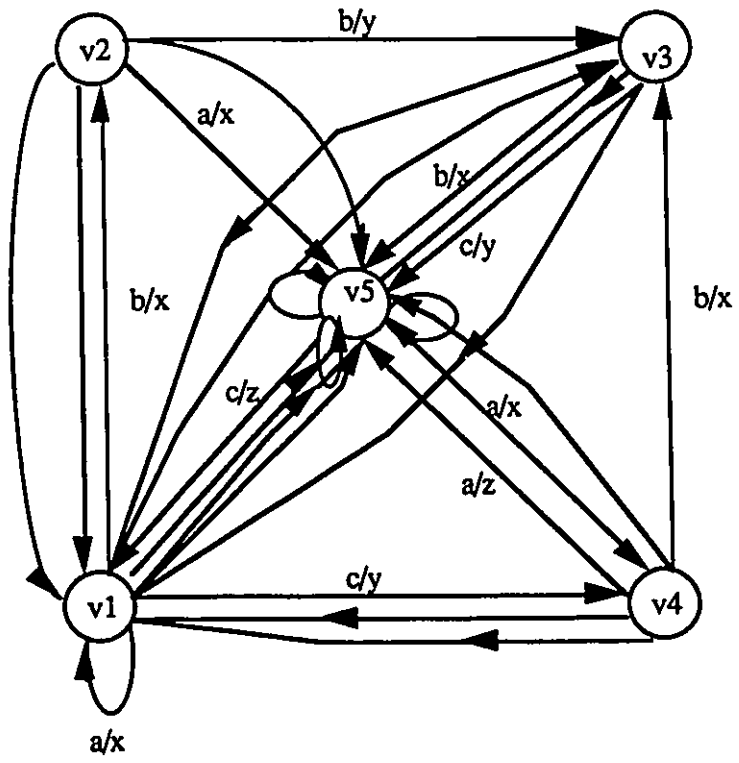


Figure 4.9 The Graph G' for the Graph in Fig 4.8

(unlabeled edges are the edges of Ec)

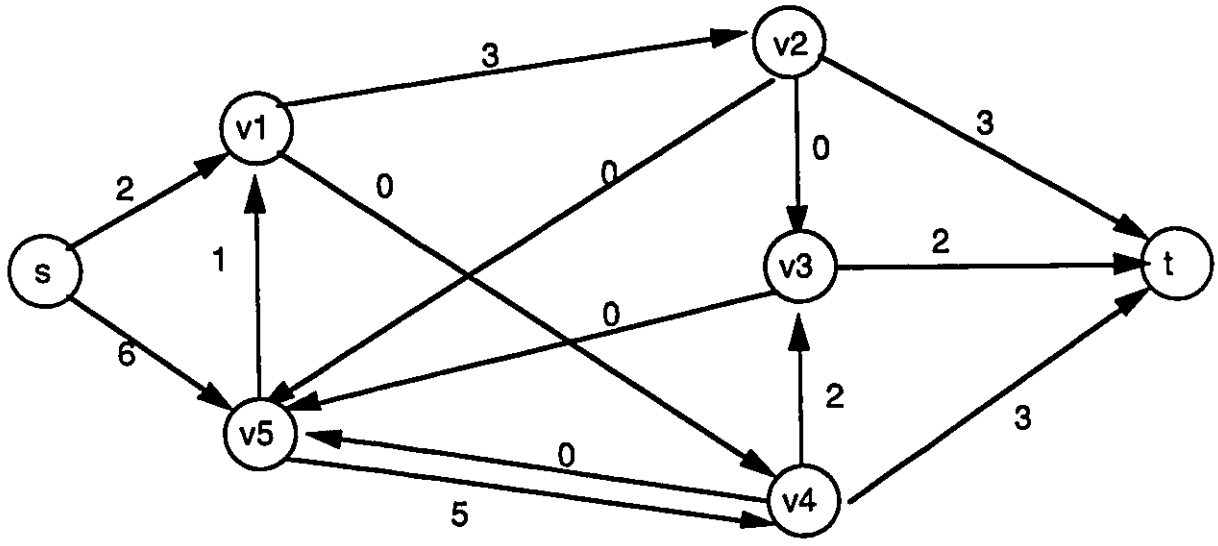


Figure 4.10 The Network Model G_f of the Graph G' in Figure 4.9

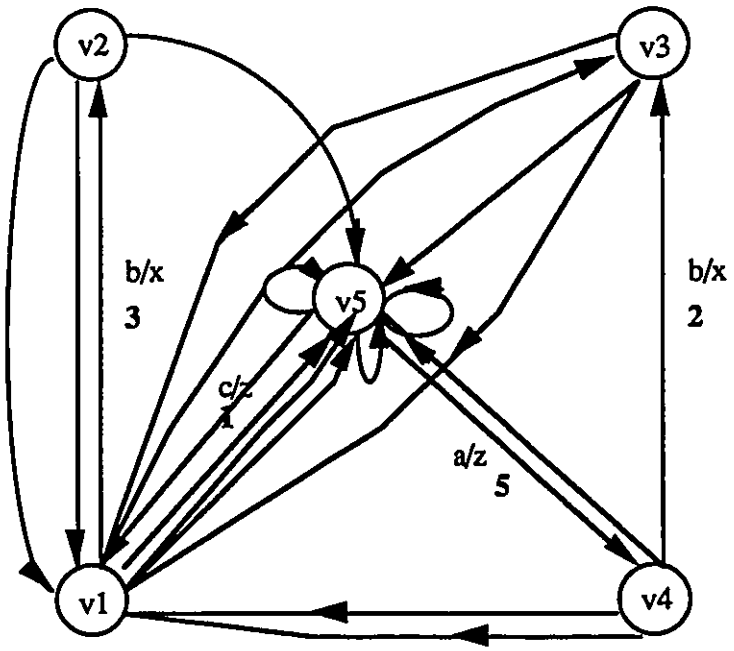


Figure 4.11 The Rural Symmetric Augmentation G* of the Graph G' in Fig 4.9.

In the above figure, the value associated with each labeled edge corresponds to the number of times the edge appears in the G* and the Euler tour. They are selected from E of G in Figure4.8.Unlabeled edges represent Ec of G[Ec] (re. Table 4.5(b)).

4.5 Multiple UIO-Method

4.5.1 Overview of the Multiple UIO-Method

The Multiple UIO-method (MUIO-method) is a variation of the Single UIO-method. The MUIO method was proposed by Shen, Lombardi and Dahbura [SLD89]. The key idea of this method is to use multiple minimum-length UIO sequences for each state of the given FSM. The result is a substantial reduction (4% ~ 37%) in the length of the test sequences with no noticeable increase in the time required to generate such a test sequence.

In the SUIO method, the graph G' is constructed such that the index for each vertex $\xi(v_i)$ is a fixed value and is not equal to zero in most of the cases. The higher the value of $\xi(v_i)$ is, the more edge duplications are needed, consequently, the test sequence will be longer. MUIO method is conceived as the result of attempting to reduce the (absolute) values of those $\xi(v_i)$'s. The idea is based on the observation that there may exist several minimum-length UIO sequences for a state of M. For example, for the M depicted in Figure 4.8, there exist four UIO sequences for v_1 :

UIO_1^1 = {b/x, a/x}, UIO_1^2 = {a/x, b/x}, UIO_1^3 = {c/y, b/x} and UIO_1^4 = {a/x, c/y}.

A set of minimum-length UIO sequences for all the other state of the M is given in Table4.7(a).

A judicious choice of UIO sequences for each state could reduce the length of overall test sequence. For example, consider the M in Figure 4.8. If we use the single minimum-length UIO's shown in Table 4.5(a), we have the following indices in G' in Figure 4.9 :

$\xi(v_1) = 2$, $\xi(v_2) = -3$, $\xi(v_3) = -2$, $\xi(v_4) = -3$, and $\xi(v_5) = 6$.

Recall that the indices in G' are calculated only based on graph G[Ec]. At vertex v_1, there are six edges in Ec going into v_1 in G' while four edges leaving v_1, therefore $\xi(v_1) = 6-4 = 2$; there are no edges in Ec going into v_2 while there are three edges leaving v_2, therefore $\xi(v_2) = 0 -3 = -3$; and so on.

UIO	SEQUENCE	TAIL
UIO(v1) :	b/x, a/x	v5
	a/x, b/x	v2
	c/y, b/x	v3
	a/x, c/y	v4
UIO(v2) :	b/y	v3
UIO(v3) :	b/x, c/z	v1
	b/x, a/z	v4
UIO(v4) :	b/x, c/y	v5
UIO(v5) :	c/z	v1
	a/z	v4

Table 4.7(a) A Set of MUIO's for the States of the FSM in Figure 4.8

Nevertheless, if Multiple UIO sequences are applied for each state of M, a G' can be constructed in the following way : using $UIO_1{}^3$ in replacing $UIO_1{}^1$ for the transition going into state v_1; this will reduce the number of edges in Ec going into v_5 while increasing the number of edges in Ec going into v_3. Thus, $\xi(v_5)$ is decreased while $\xi(v_3)$ is increased; $UIO_1{}^2$ is used to test the state transition of (v_1, v_1; a/x) and $UIO_1{}^3$ is used to test the state transition of (v_1, v_1; ri/null), and so on. Multiple UIO sequences (MUIO's) for each state of M are used for different transitions whose tail state is the same state. In this way, the indices in G' are reduced to the following:

$\xi(v_1) = \xi(v_2) = \xi(v_3) = 0$; $\xi(v_4) = -1$ and $\xi(v_5) = 1$

The result graph G' using the set of MUIO's in Table 4.7(a) is shown in Figure 4.12. As we can see, the number of augmented edges for the symmertric G* is reduced; the length of the test sequence which is the Euler tour of G* is reduced to 45 as shown in Table 4.8 accordingly, compared with that of 55 in SUIO approach in which the same FSM (in Figure

4.8) is used. A rural symmetric augmentation G* of the graph G' in Figure 4.12 is depicted in Figure 4.13.

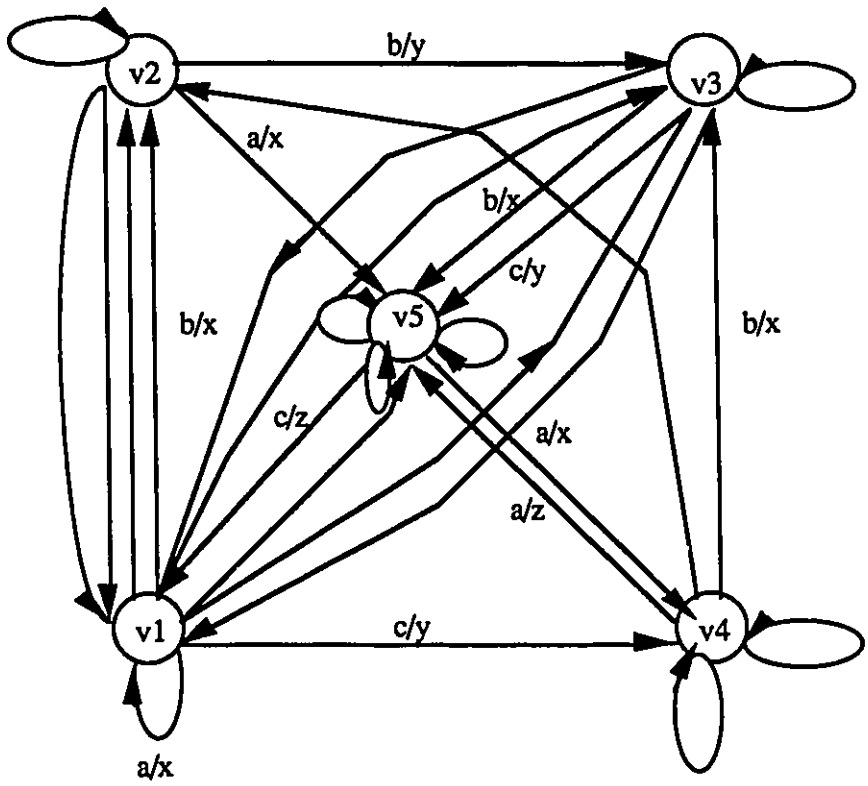


Figure 4.12 An Alternative Graph G' Derived by the MUIO-method (The labeled edges represent E; unlabeled edges represent Ec).

The following table lists all the edge labels in Ec :

TRANSITION	i/o@UIO(TAIL(i/o))	TAIL
T_{11}	ri/null c/y b/x	v3
T_{11}	a/x a/x b/x	v2
T_{12}	b/x b/y	v3
T_{14}	c/y b/x c/y	v5
T_{21}	ri/null a/x b/x	v2
T_{23}	b/y b/x c/z	v1
T_{25}	a/x c/z	v1

T_{35}	b/x c/z	v1
T_{35}	c/y c/z	v1
T31	ri/null c/y b/x	v3
T_{41}	ri/null a/x b/x	v2
T_{45}	a/x a/z	v4
T_{43}	b/x b/x a/z	v4
T_{54}	a/z b/x c/y	v5
T_{51}	c/z b/x a/x	v5
T_{51}	ri/null b/x a/x	v5

Table 4.7(b) Transition Table for G' in Figure 4.12

4.5.2 The Procedure for the MUIO-Method

MUIO-method of deriving a minimal test sequence consists of the following major steps:

step 1. compute the set $MUIO_k$ of all the minimum-length UIO's for each state v_k of M;

step 2. construct a network model G_m (V_m, E_m);

step 3. compute a minimum-cost maximum flow in G_m to solve the duplication problem on UIO sequences;

step 4. assign the selected UIO_k^{α} to each edge (v_j, v_k; i/o) of G accordingly to obtain the graph G' of G, where k = 1, 2, ... , n;

step 5. apply the RCPP algorithm on G': construct a network flow model $G_f(E_f, V_f)$, compute a min-cost max flow in G_f; construct a RSA G* of G'; find an Euler tour of G* which is the resulting minimal test sequence.

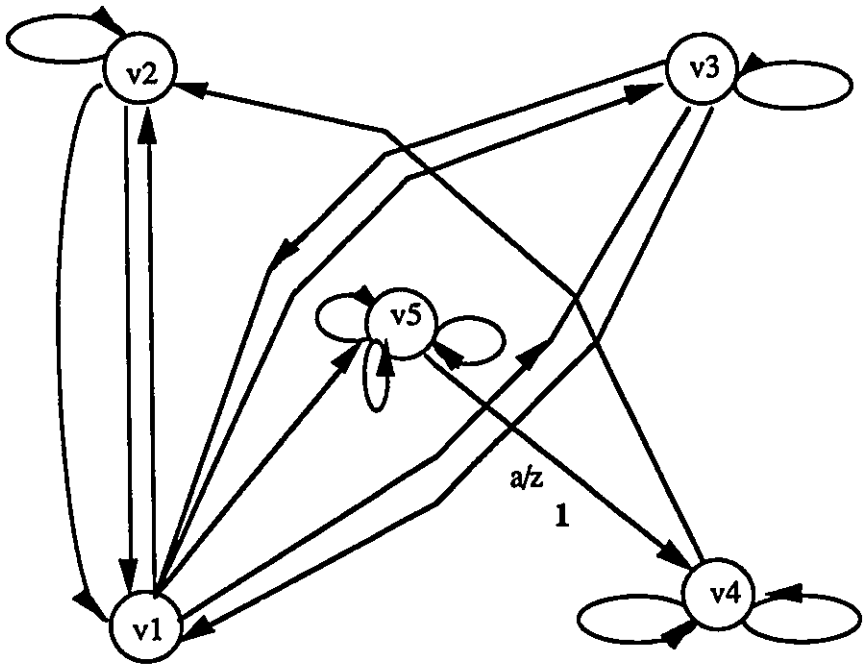


Figure 4.13 The Rural Symmetric Augmentation G* of G' in Figure 4.12 (labeled edges are the copies selected from E; the unlabeled edges represent Ec as specified in Table 4.7(b)).

ri/null	a/x a/x b/x	ri/null a/x b/x	b/y b/x c/z
b/x b/y	ri/null c/y b/x	b/x c/z	ri/null c/y b/x
c/y c/z	c/y b/x c/y	a/z b/x c/y	c/z b/x a/x
ri/null b/x a/x	a/z	a/x a/z	b/x b/x a/z
ri/null a/x b/x	a/x c/z		

TOTAL = 45 inputs

Table 4.8 A Minimal Test Sequence for the FSM in Figure 4.8

In the following section, we will give the algorithms for each step above in the method.

4.5.3 The Algorithms for the MUIO-method

Given a digraph G(V, E), representing the given FSM M, G' is as defined in the SUIO-method. We assume that there are r_k minimum-length UIO sequences for state s_k, where r_k

> 0, and all the tail states of UIO_k^{α} are distinct and they are different from their head states [SLD89] (i.e., $HEAD(UIO_k^{\alpha}) <> TAIL(UIO_k^{\alpha})$), k = 1, 2, ... , n, $\alpha = 1, 2, \ldots, r_k$.

Compute MUIO's :

By using a modified version of the algorithm in [SaDa88], MUIO's can be computed for each state of M;

Construct Network Flow Model G_m :

Let the degree of G' be $\Delta(G') = \sum_{i=1}^{n} |\xi(v_i)|$, where $v_i \in G[Ec]$. The purpose of this step is to solve "Multiple UIO assignment problem (MAP)". Formally, MAP is defined as follows : Given G and a set of $MUIO_k$ for each state v_k, where $MUIO_k = \{UIO_k^1, \ldots, UIO_k^{r_k}\}$, for v_k, k = 1, 2, ... , n,
find an element $UIO_k^{\alpha} \in MUIO_k$ for each edge $(v_j, v_k, i/o) \in E$ such that $\Delta(G')$ is minimized.

Note that each edge of Ec is a concatenation of an edge $(v_j, v_k, i/o) \in E$ with the UIO sequence : $UIO(v_k)$. Since every edge in E should be checked, the number of edges in G[Ec] is exactly the same as that of G, and the number of outgoing edges in Ec at each vertex v_j of G' is equivalent to $d_{out}^E(v_j)$ in G. In constructing G', there should be exactly $d_{in}^E(v_k)$ UIO_k^{α}'s assigned to the edges going into v_k, because there are $d_{in}^E(v_k)$ transitions terminated at v_k which should be verified. This problem is reduced to the problem of assigning an appropriate value to each $UIO_k^{\alpha} \in MUIO_k$, where k = 1, 2, ... , n and $1 \le \alpha \le r_k$. This value is equivalent to the number of UIO_k^{α}'s needed to construct a G' such that $\Delta(G')$ is minimized. It is again a network flow problem.

Given G, let $G_m = (V_m, E_m)$ be a directed graph such that

$V_m = \{s, t\} \cup H \cup T$

where

$H = \{h_1, h_2, \ldots, h_n\}$ and $T = \{t_1, t_2, \ldots, t_n\}$, $h_i, t_i \in V$ for all i.

H and T are the sets of all the possible head states of $MUIO_k$ and tail states of $MUIO_k$ (k = 1, 2, ... , n), respectively. The flows to be decided on them are the number of copies needed for each of the UIO's in order to construct a G' of G.

Let $E_m = E_S \cup E_t^- \cup E_t^+ \cup E^*$,

where

$E_S = \{(s, h_k) : h_k \in H\}$, $E_t^- = \{(t_l, t) : t_l \in T\}$, $E_t^+ = \{(t_l, t) : t_l \in T\}$, and

$E^* = \{(h_k, t_l) :$ there exists a UIO_k^{α} such that $TAIL(UIO_k^{\alpha}) = t_l\}$

We use $(t_l, t)^-$ and $(t_k, t)^+$ denote an edge $(t_l, t) \in E_t^-$ and an edge $(t_l, t) \in E_t^+$, respectively. The E_t^+ is called a supplementary network component.

So far, a network model is constructed. The cost and flow capacity of the G_m are decided as follows:

each edge $(s, h_k) \in E_S$ has cost zero and capacity $d_{in}^E(v_k)$;

each edge $(t_l, t)^-$ has cost -1 and capacity $d_{out}^E(v_l)$;

each edge $(t_l, t)^+$ has cost +1 and infinite capacity.

each edge $(h_j, t_k) \in E^*$ has cost zero and infinite capacity.

As an example, the network $G_m(V_m, E_m)$ constructed from G of Figure 4.8 with a set of $MUIO_k$ shown in Table 4.7(a) is given in Figure 4.14, where k = 1, 2, 3, 4, 5.

Compute a min-cost max flow in G_m :

A flow in G_m is a function $F : E_m \rightarrow Z^+$, where Z^+ is a set of non-negative integers, s.t.,

1) for each vertex $h_k \in H$,

$F(s, h_k) = \Sigma F(h_k, t_l)$, $(h_k, t_l) \in E^*$;

2) for each vertex $t_l \in T$,

$F(t_l, t)^- + F(t_l, t)^+ = \Sigma F(h_k, t_l)$, $(h_k, t_l) \in E^*$;

3) for each edge $(s, h_k) \in E_S$,

$F(s, h_k) \leq$ capacity$(s, h_k) = d_{in}^E(h_k)$, and

4) for each edge $(t_j, t)^-$,

$F(t_j, t)^- \leq \text{capacity}(t_j, t)^- = d_{out}^{E}(t_j)$

Note that $G_m(V_m, E_m)$ here is not a standard network flow model. A modified Busacker-Gawen's min-cost flow algorithm [SDK83] can be used to compute the flow F in G_m without changing the upper bound of time complexity of the algorithm.

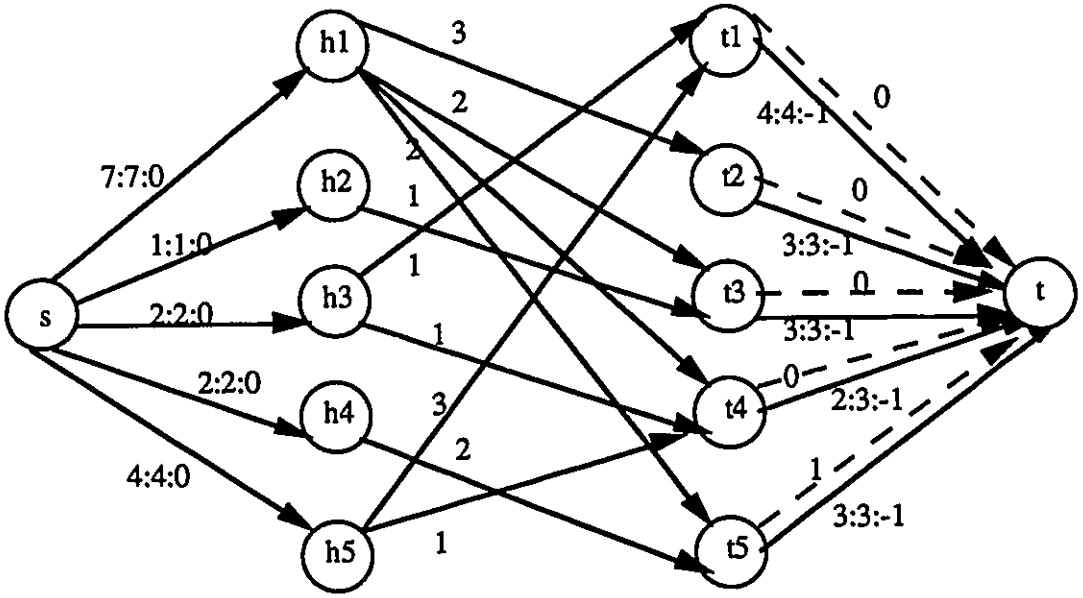


Figure 4.14 The Network Model G_m Constructed Based on a Set of MUIO's Listed in Table 4.7(a), and a min-cost max flow in G_m.

In Figure 4.14, the label format for edges connected with s and t is : (flow : capacity : cost). Dashed edges labeled only with the flows. They are with infinite capacity and unit cost. All the intermediate edges labeled with flows, each of them with infinite capacity and zero cost.

MUIO Assignment Procedure:

Given a min-cost max flow F in G_m, the Multiple UIO sequence assignment procedure (for constructing the G' of G) is given below:

For each $v_k \in V$ ($k = 1, 2, \ldots, n$), assign to each edge $(v_j, v_k) \in E$ a UIO sequence $UIO_k^{\alpha} \in MUIO_k$ such that each UIO_k^{α} in $MUIO_k$ is used exactly $F(h_k, t_l)$ times, where $(h_k, t_l) \in E^*$ is an edge in G_m representing UIO_k^{α}, $h_k = HEAD(UIO_k^{\alpha})$, $t_l = TAIL(UIO_k^{\alpha})$. For example, for the min-cost max flow F of G_m shown in Figure 4.14, $F(h_1, t_5) = 2$; therefore UIO_1^1 is assigned to two edges going into vertex v_1 in G : (v_5, v_1; c/z) , and (v_5, v_1; ri/null). Similarly, since $F(h_1, t_3) = 2$, UIO_1^3 is assigned to two edges going into vertex v_1 : (v_3, v_1; ri/null) and (v_1, v_1; ri/null). Following the same procedure for other vertices of G, a G' is constructed.

It has been proved that if F is a min-cost max flow in G_m then the corresponding assignment of UIO sequences to the edges of G is such that the degree $\Delta(G')$ is minimized [SLD89].

Apply the RCPP Solution :

Follow the same procedure as it is described in Section 4.4.4 : first, construct a standard network flow model $G_f = (V_f, E_f)$ from G' and G; second, compute a min-cost max flow in G_f to obtain a RSA G* of G'; then, find an Euler tour of G*, which is the RCPT of G'. A minimal test sequence is thus obtained.

The MUIO approach is implemented at SUN workstation in the Computer Science Dept. of University of Ottawa, the tool was tested by using the same example M in Figure 4.8. It was found that after a min-cost flow is found in G_m, the index at each vertex is equal to zero. That means that the graph G[Ec] is symmetric and contains an Euler tour which is the minimal test sequence. The G[Ec] is constructed according to the flow calculated in network G_m shown in Figure 4.15. The graph G[Ec] and a set of MUIO's are depicted in Figure4.16 and Table 4.9, respectively. The minimal test sequence is listed in Table 4.10. The length of the test sequence is 44, which is one input less than the TS in Table 4.8.

UIO	SEQUENCE	TAIL
UIO(v1) :	b/x, a/x	v5
	a/x, b/x	v2
	b/x, b/y	v3
	a/x, c/y	v4
UIO(v2) :	b/y	v3
UIO(v3) :	b/x, c/z	v1
	b/x, a/z	v4
UIO(v4) :	b/x, b/x	v5
UIO(v5) :	c/z	v1
	a/z	v4

Table 4.9 A Set of MUIO's for the States of the FSM in Figure 4.8

ri/null	c/y b/x b/x	a/z b/x b/x	c/z b/x a/x
ri/null a/x c/y	a/x a/z	ri/null b/x b/y	ri/null b/x b/y
c/y a/z	b/x b/x c/z	b/x b/y	b/x c/z
a/x a/x b/x	ri/null a/x b/x	b/y b/x c/z	ri/null a/x b/x
a/x c/z			

TOTAL = 44 inputs

Table 4.10 A Minimal Test Sequence for the FSM in Figure 4.8

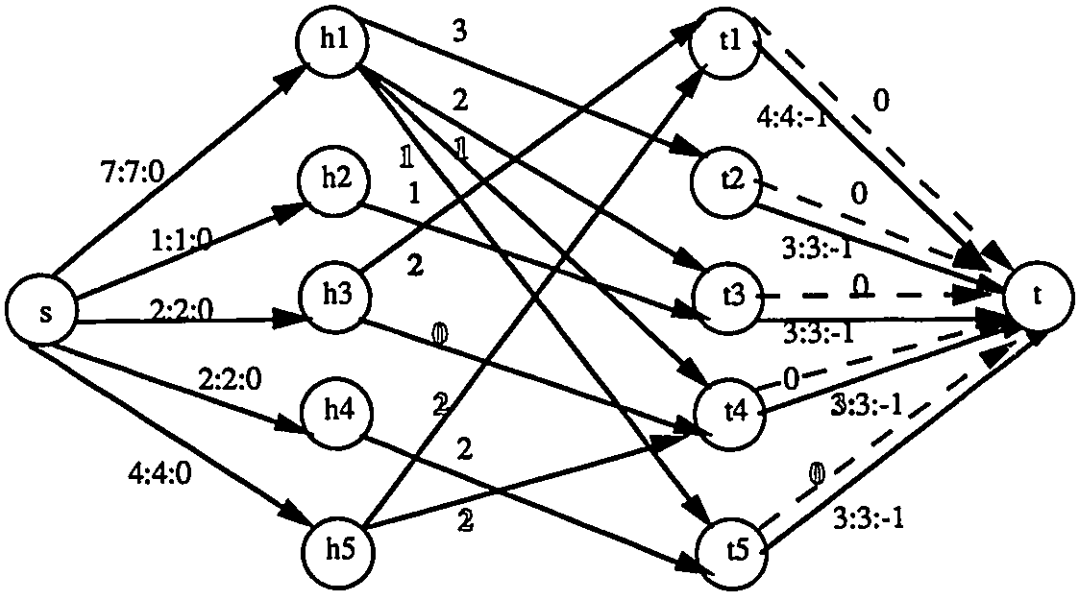


Figure 4.15 The Network Flow Model Gm Based on the FSM in Figure 4.8, and a set MUIO's in Table 4.9, and a flow in G_m. The flow values in double character style are the flows which are different from that of G_m in Figure 4.14.

4.5.4 Some Open Issues

The MUIO - method uses multiple UIO sequences for each state of M aimed at reducing the amount of edge duplications in constructing G* of G'. When combined with the RCPT techniques, this method has some significant improvement in reducing the length of the test sequences compared with the SUIO-method [SLD89]. However, there are some problems which still remain open :

a) the assignment of edges to UIO sequences which minimizes $\Delta(G')$ may not minimize the total length of the tour of G' theoretically, if we consider the cost of each edge;

b) it is possible that G' is not connected even if M has reset and/or self-loop properties when MUIO-method is applied. We suggest two possible ways to deal with this situation:

1) select a fixed UIO_1 for state v_1 (representing s_S in M);

2) add edges from E into Ec until G[Ec] is weakly connected.

These issues will be discussed further in Chapter 6.

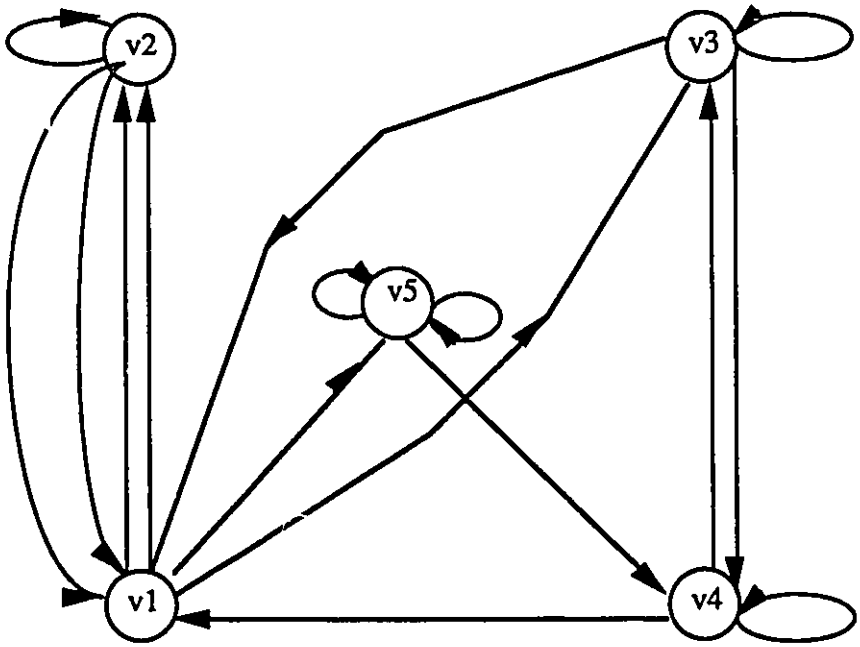


Figure 4.16 The G[Ec] Derived from G in Figure 4.8 and a flow in Gm (G[Ec] is symmetric therefore has an Euler tour).

The following table gives the label of every edge of G[Ec] in Figure 4.16 :

TRANSITION	i/o@UIO(TAIL(i/o))	TAIL
T_{11}	ri/null a/x b/x	v2
T_{11}	a/x a/x b/x	v2
T_{12}	b/x b/y	v3
T_{14}	c/y b/x b/x	v5
T_{21}	ri/null a/x b/x	v2
T_{23}	b/y b/x c/z	v1
T_{25}	a/x c/z	v1
T_{31}	ri/null b/x b/y	v3
T_{35}	b/x c/z	v1
T_{35}	c/y a/z	v4

T_{41}	ri/null b/x b/y	v3
T_{43}	b/x b/x c/z	v1
T_{45}	a/x a/z	v4
T_{51}	ri/null a/x c/y	v4
T_{51}	c/z b/x a/x	v5
T_{54}	a/z b/x b/x	v5

Table 4.11 The Transition@MUIO-Table for the G[Ec] in Figure 4.16.

CHAPTER 5 EXTENSIONS OF THE D- AND W-METHODS

In Sections 4.4 and 4.5, we show that RCPT technique is applied to the SUIO-method and the MUIO-method to generate minimal test sequences. This chapter demonstrates that RCPT algorithm can also be used in the D- and W-methods in generating minimal test sequences systematically.

5.1 Application of the RCPT Approach to the D-Method

Recall that Gonenc's approach described in Section 4.2.3 is to a find minimal covering of the i@DS-diagram G[Ec]. His approach of connecting all the simple paths of a G[Ec] in forming a minimal covering of the G[Ec] is heuristic and is time consuming when applied to large and complex protocols. It is because in step 4 of the procedure, k-1 TR's are needed when G[Ec] has k simple paths. Each of $TR_j(k)$ brings M from the terminal state s_j of one path to the initial state s_k of another path, where $s_k \in F$ (a set of vertices whose in-degree $\leq$ out-degree). In order to find the shortest $TR_j(k)$, all the possible $TR_j(k)$'s have to be generated and compared. In the worst case, k = (n-1), then (n-2) TR's are required. To compute each TR needs maximum $O(n^2)$ steps [Even79]. The overal complexity only for TR's will be (n-2) times n^2 which is $O(n^3)$. This does not count on comparisons between TR's which are recognized as one of the most time consuming job in any implementation. Also, there is no optimal solutions for obtaining a complete test sequence when G[Ec] is not weakly connected. Therefore, in considering efficiency, a systematic approach is needed for the D-method. Based on the experience on the SUIO-method, we found that RCPT can also be applied to the D-method by modifying the way of traversing the graph G[Ec] (note that the complexity for max-flow method is $O(n^2)$, and the complexity for finding an Euler tour is linear [GoLu80, SDK83]). The procedure of deriving a minimal test sequence by the D-method using the RCPT approach is as follows :

1) expand the graph G[Ec] to G'(V', E'), where V' = V, E' = Ec U E;

2) follow the same procedure as it is described in the SUIO-method : construct a network flow model G_f based on graph G[Ec] (capacity) and G (structure); compute a min-cost max flow in G_f which is the solution to the duplication problem; construct a RSA G* of G'; find a Euler tour of G* which is the RCPT of G' and a minimal test sequence consequently.

Assuming that M has reset capability or self-loops, then G[Ec] is guaranteed to be weakly connected.

An example minimal test sequence for the M shown in Figure 4.1 is given in Table 5.1. The total length of the test sequence is 44. It is four inputs longer than that of Gonenc's version (in Section 4.2.4) but the test starts and ends with the same state (state 1 of M) and it is obtained by a systematic and less time consuming way.

aa, b@DS, aa, a@DS, b@DS, aa, b@DS, aa, a@DS, aaa, b@DS, a@DS, a@DS, aaa, a@DS, b@DS

TOTAL = 44 inputs

Table 5.1 A Minimal Test Sequence for the FSM in Figure 4.1

5.2 Application of the RCPT Approach to the W-method

The original W-method proposed by Chow [Chow78] only stresses the observability aspect. Each test $TEST_{jk}$ is a sub-test sequence of the whole test, and the terminal state of one test $TEST_{jk}$ is not necessarily coincident to the beginning state of another test $TEST_{kl}$.

When RCPT is applied to the W-method, both observability and controllability aspects are covered to generate minimal test sequences. A procedure is given below :

Assume that M has reset function or self-loops.

1) compute a W-set (W) by applying multiple experiment on M;

2) construct a G' = (V', E')

where

$V' = V$, $E' = Ec \cup E$,

$Ec = \{ (v_j, v_l; i/o@W) \}$, where $(v_j, v_k; i/o) \in E$, $v_l = TAIL(i/o@W)$.

let $W = \{w_1, w_2, \ldots, w_\mu\}$, then

$$i/o@W = \{i/o@w_1@TR\#i/o@w_2@TR\#\ldots\#i/o@w_\mu\}$$

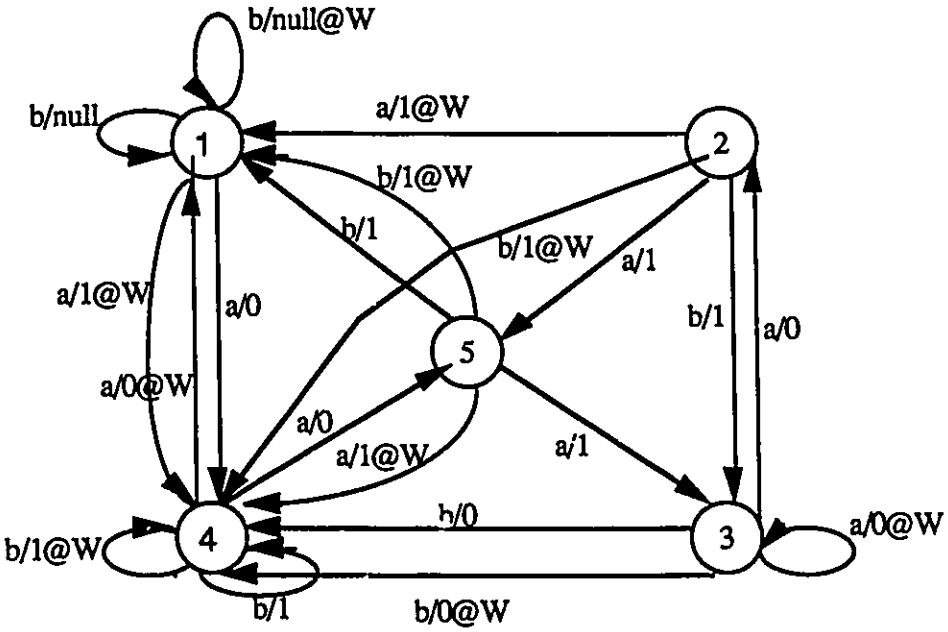
$$= \#_{j=1}^{\mu} (i/o@w_j)$$

where $\mu = |W|$, each TR is a transfer sequence which brings M from a $TAIL(i/o@w_j)$ to the HEAD(i/o) of the same i/o, where $j = 1, 2, \ldots, \mu$;

3) solving duplication problem by network flow method described in the SUIO-method;

4) construct a RSA G* of G' and find an Euler tour in G* which is the minimal test sequence.

As an example, the G' for the FSM in Figure 4.1 is depicted in Figure 5.1. The reset function (ri) is not assumed in this particular example, since G[Ec] is connected without ri. A RSA G* of G' is shown in Figure 5.2. A minimal test sequence is given in Table 5.2 with the length of 110 (including the initial reset before the test starts), compared with that of 167 in example test sequence shown in Table 4.3(b) in which Chow's original W-method is applied. If the same FSM is assumed to have *ri* function, the resulting test sequence (RCPT) is of length 164, including the initial reset ri.



where W = {a, aa, b}

Figure 5.1 The Graph G' of the FSM in Figure 4.1
(The edges labeled with I/O@W represent Ec, and
the rest of edges represent E of G in Figure 4.1).

ri a abaa bb aa baabaabb aa aaaaaaab babababbb aa baababb aaa babaaaabb ababaabbab ababaaaabab aa aaaaaaab a aaaaaaab baabbaabbb

TOTAL = 110 inputs

Table 5.2 A Minimal Test Sequence for the FSM in Figure 4.1.

In the above table (Table 5.2), the bold inputs represent transitions to be tested; the underlined inputs are those w_i's in the W-set; the rest of inputs are transfer sequences.

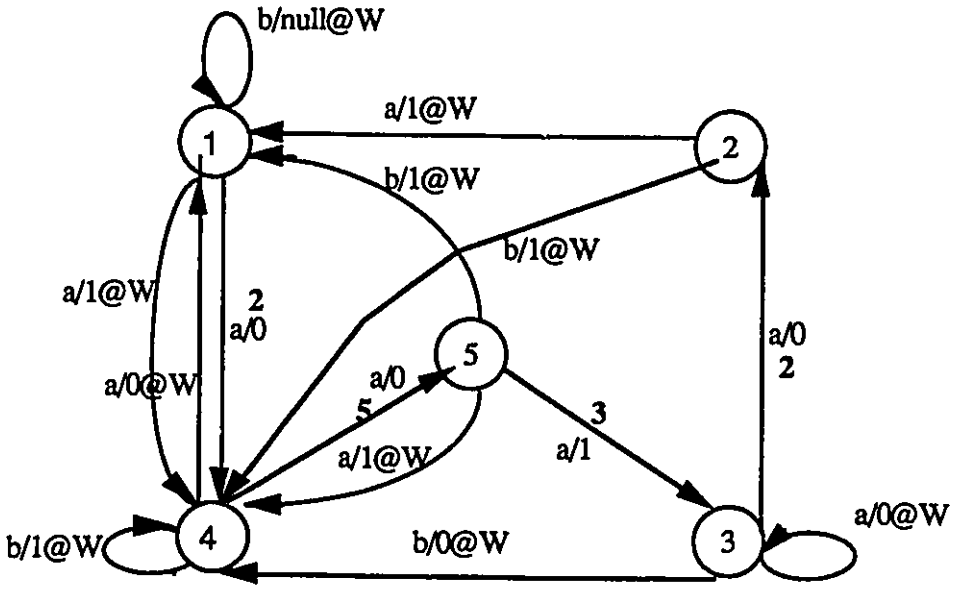


Figure 5.2 The Rural Symmetric Augmentation G* of the G' in Figure 5.1 (There are four edges from E which are labeled with values additional to the i/o's. The values are the number of appearings of the corresponding edges in a RCPT of G' which is a minimal test sequence for the FSM in Figure 4.1).

CHAPTER 6 COMPLEXITY ANALYSIS AND IMPLEMENTATION ISSUES

This chapter discusses issues regarding implementations of the five formal methods described in the previous chapters after pointing out the upper bounds on the lengths of the test sequences generated by all these formal methods including the two variations of the D-method and the W-method. A brief comparison of the five methods is given at the end of the complexity analysis.

6.1 Complexity Analysis of the Five Formal Methods

Given a graph G representing a FSM M, let n be the number of states in M, k be the number of input symbols of M. The upper bounds on the length of a test sequence (TS) generated by each formal method described in this thesis are as follows :

The T-Method :

The upper bound on the length of a TS is $O(n^2k)$.

It is based on the fact that the maximum flow in $G_f \leq$ total number of edges of G which is n*k. The flow distributed at each node in order to get balance is maximum n*k, therefore n*(nk) is the upper bound of edge duplications. The number of transitions in a completely specified M is n*k, hence the upper bound for the T-method is : $n^2k + n*k \rightarrow O(n^2k)$.

The D-Method :

The upper bound on the length of a TS derived by Gonenc's D-method is $O(n^{n+2}k)$ [SaBo84].

When combined with the solution of the RCPP, the upper bound on the length of a TS derived by the D-method is $O(n^{n+1}k)$.

Proof :

The upper bound on the length of a DS is n^n in the worst case [Koha78]. In constructing the graph G[Ec], the total cost (equals to the total number of edges implied in Ec) is :

$nk + nk*n^n = nk(n^n+1)$. This is derived as follows :

nk is the number of edges in G. It is equal to the number of edges in Ec. Since each edge of G[Ec] is a concatenation of one edge in G with the DS, the actual length of a basic tour contains all the edges of G[Ec] is: nk + nk*(length of DS), which comes up with the above result.

On the other hand , the maximum number of possible edge duplications is n^2k (re. T-method). Thus, the upper bound on the length of a TS is :

$nk(n^n+1) + n^2k = nk(n^n+n+1)$ -> $O(n^{n+1}k)$.

The above bound is one level lower than that of Gonenc's D-method.

The W-Method

The upper bound on the length of the TS generated by Chow's W-method is given in [SaBo84] as : $O(n^4k)$

When combined with the solution of the RCPP, the upper bound on the length for a minimal test sequence derived by this method is of :

$n^2 + (nk*(n-1)) + n^2*nk + n^2k^2(n-2) + n^2k$ -> $O(n^3k^2)$.

Proof :

The bound on length of a W-set is : $(n-1)^2$ -> $O(n^2)$ [Gill62];

The maximum number of w_i's in a W-set is (n-1) [Gill62]. The left hand side of the above complexity formula means the following :

|(the initial)W-set| + (number of edges in G)*(number of w_i's in a_W-set) + |W-set|*(number of edges in G) + (sum of the TR's from s_t to s_j) + (edge replications).

Where s_t is the tail state of the present test $TEST_{st}$, s_j is the head state of the next transition to be tested. The upper bound of the shortest path (for TR's) between s_t and s_j is nk which equals to the number of edges in G. There are (nk*|w_i's|)-nk TR's are needed for constructing a graph G' (in fact G[Ec]) of G, where "minus nk" means that nk edges of M are artificially connected one after another in forming the graph G[Ec] so that they do not need TR's. Hence, the second last item in the above formula is nk*[nk*(n-1)-nk].

The RCPT approach of W-method has a better bound on TS length than the $O(n^4k)$ of Chow's W-method.

The SUIO-Method :

The bound on length for a UIO sequence is $O(n^2)$ [SaDa88], assuming that a UIO sequence exists for every state of M;

If some state has no UIO,the upper bound for computing a Unique Signiture would be:

C_n^2*|A diagnosing sequence for a pair of states of M| + |a_TR|

= [n(n-1)/2]*(n-1 + nk) -> $O(n^3k)$.

If we consider complexity of $O(n^2)$ for UIO sequences, the upper bound on the length of a test sequence derived by the SUIO-method will be :

$|UIO|_{max}$*(total number of UIO's in G[Ec]) + (number of transitions in M) + (total edge duplications) = n^2*nk + nk + n^2k ->

$O(n^3k)$.

It is better than that for the W-method. Experiences show that most protocols' UIO sequences are no longer than 6 i/o symbols, and the resulting test sequence is much shorter than that of using the W-method or D-method [ADLU88].

The MUIO-Method :

MUIO-method has the same upper bound on the length for UIO sequences as that of the SUIO-method. This is because that generating a set of MUIO for each state of M is the worst case of generating a SUIO sequence for each state if the UIO is found in the last branch of the tree when one considers the search of a UIO for a state of M to be like building a successor tree [Koha78] for that state. In MUIO approach, when a minimal UIO is found at some level of the tree, the search still continue to find all the UIO's at the same level (i.e., seeking for all the minimal UIO's for that state). Whereas in the SUIO-method, the search stops as soon as a minimal UIO is found.

6.2 Comparisons of the Five Formal Methods

Theoretically T-method has the lowest complexity on the length of a TS among all the implemented formal methods. It is also true in practice. However, it can only detect output errors of an IUT. All the other four formal methods (i.e., SUIO-method, MUIO-method, D-method and W-method) have the capabilities of detecting both output errors and state transition errors. Theoretically, the SUIO-method and MUIO-method have the lowest bounds on the length of a TS compared with the D-method and the W-method. The D-method has the highest upper bound on the length of a TS. In practice, the length of a TS depends, to a large extent, on the length of the corresponding checking sequences (i.e., UIO sequences (UIO's) in the SUIO-method, MUIO sequences (MUIO's) in the MUIO-method, Distinguishing Sequence (DS) in D-method, and W-set in W-method). The MUIO-method is the best among the four formal methods in terms of generating shortest TS's; the SUIO-method comes the second best, since for most protocols, the length of the UIO's (and MUIO's) do not exceed six i/o symbols. The minimal W-set is usually longer than the minimal DS for a FSM. This is one of the reasons why the W-method often yields the longest TS among all the formal methods. As for their applicability, the T-method can always be applied to a FSM as long as the FSM is strongly connected; a FSM which

satisfies the four conditions (re.Section 3.2.3) always has a W-set [Gill62]; most FSM's have UIO's (MUIO's), but not many protocols (modeled as FSM's) possess DS's. Also, deriving a DS can be extremely time consuming [SaBo84]. This restricts the applications of the D-method to the real world protocols.

In the next section, we discuss the implementation issues of the five formal methods. We first outline the overall design of the software tool developed at University of Ottawa, Department of Computer Science, named "TSG", meaning Conformance Test Sequence Generator; then, we give an explanation of each function in the design. Some important algorithms implemented in TSG are also described.

6.3 The Overall Design of TSG

TSG is programmed in C under UNIX operating system on a SUN workstation. TSG consists of two major components as they are depicted in Figure 6.1: one is called "USER INTERFACE", the other is called "GENERATE TEST SEQUENCE". As a matter of fact, TSG implemented seven formal methods, two of which are the variations of the original methods. They are the applications of the RCPT to the D- and W-methods (Chapter 5).

6.3.1 The User Interface

The USER INTERFACE component has the following basic functions:

1) read in a user provided FSM

The user can choose to enter the FSM from standard input (i.e., the keyboard), or from a preset data file with a designated file name (re. User Manual in Appendix). An input menu is maintained for the user during the input period. The purpose of the menu is to provide the user the chance to make changes to any part of the input FSM (re. Figure 6.2).

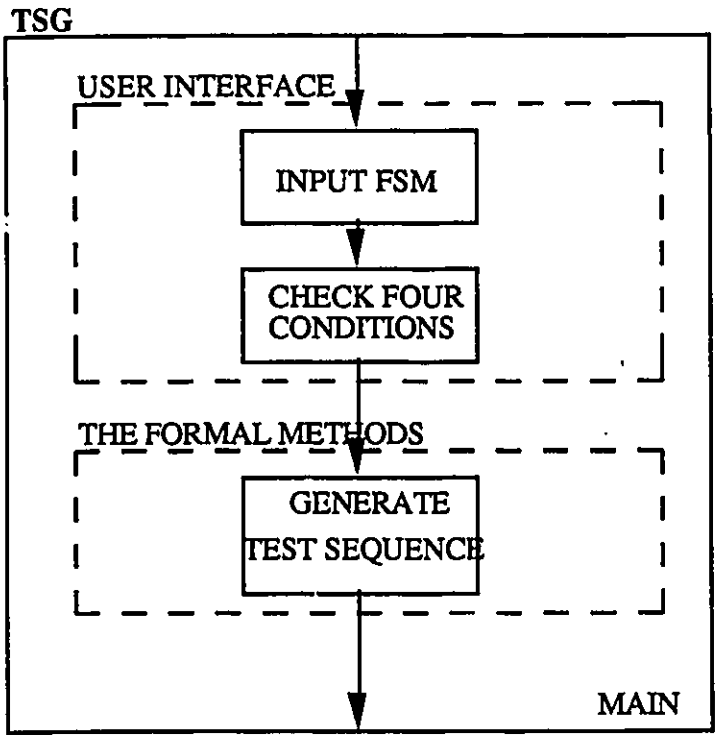


Figure 6.1 The Overall Design of TSG

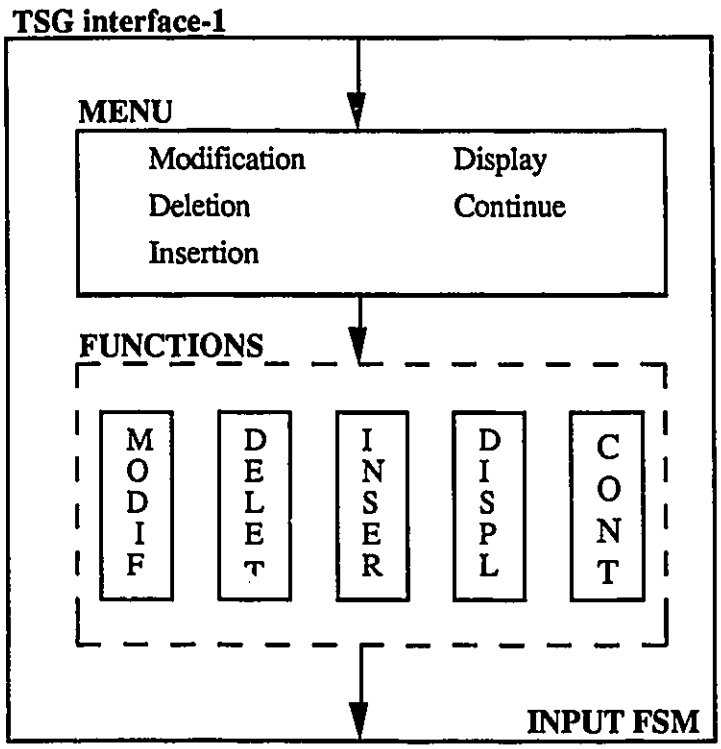


Figure 6.2 TSG interface-1 : Input a FSM

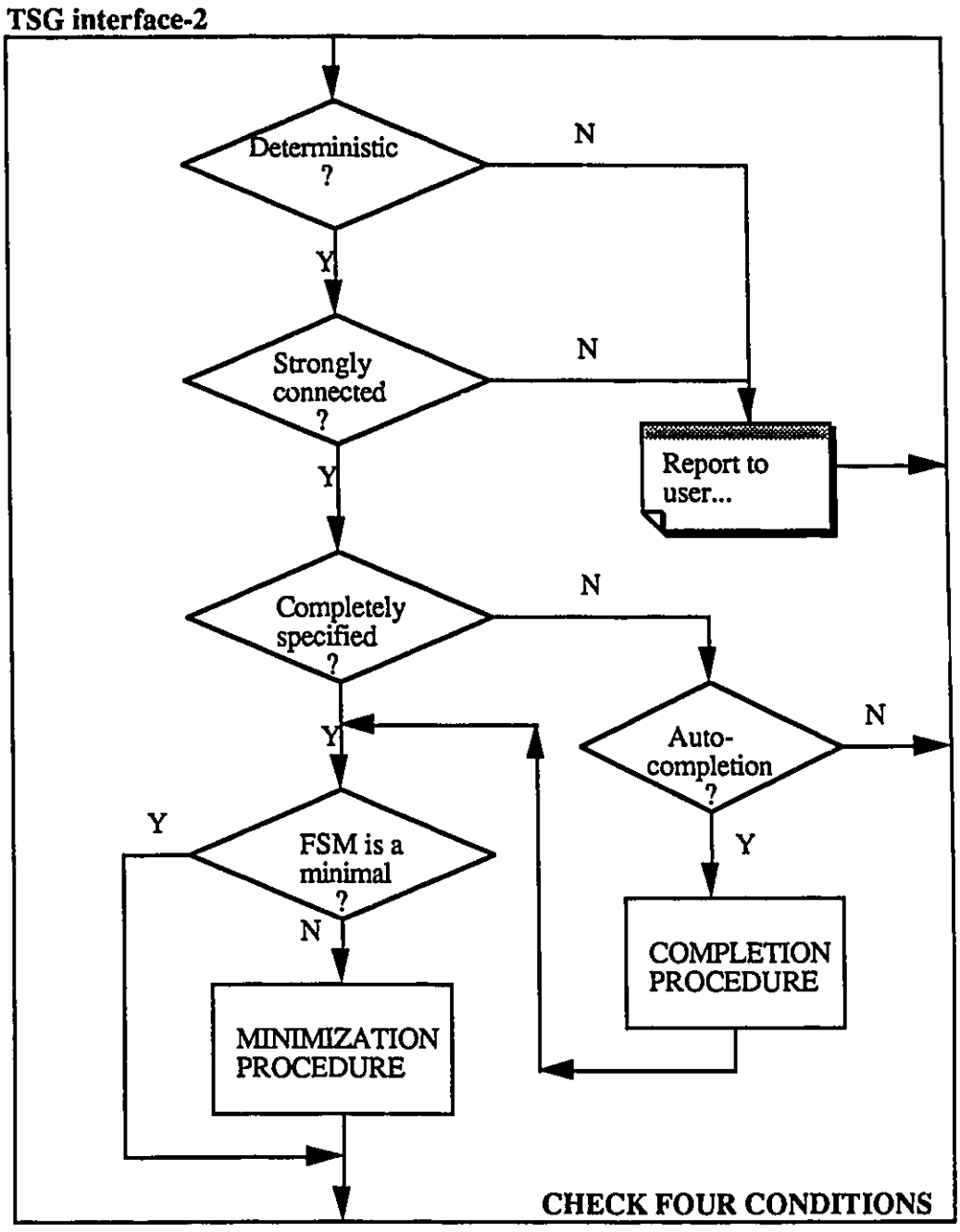


Figure 6.3 TSG User Interface-2, Checking Four Conditions

2) check the four conditions

The design for this function is outlined in Figure 6.3. After the FSM M is read into the system, it is first checked for the following four conditions as required by the seven formal methods :

a) deterministic,

b) strongly connected,

c) completely specified, and

d) minimal.

If a) is not satisfied, TSG reports to the user that the given FSM is a non-deterministic machine, and exits the system.

If b) is not satisfied, TSG reports to the user that the given FSM is not strongly connected and exits the system. If the above two situation happens, it is the user's responsibility to correct his design. Whereas, it is not the case in condition c) and d).

If c) is not satisfied by the given M, TSG provides the following choices to the user after listing all the unspecified inputs :

. complete M by the user, then re-run TSG;

. complete M by TSG.

In the first case, TSG gives the user suggestions to complete M according to either the completeness assumption 1) or assumption 2) in which an error state is introduced for M as it is mentioned in Section 3.2.1.

If condition d) is not satisfied, a minimization function is called and M will be minimized.

The above implementation means that the condition c) and d) can be relaxed for a given FSM since our tool can fulfill the tasks of completing and minimizing a FSM.

6.3.2 Access to the Five Formal Methods

In this component of TSG, a menu provides the user all the conveniences of accessing the functions of the five formal methods. There are seven major functions in this component. Each of the them implements one of the following formal methods : T-method, D-method, W-method, UIO-method and MUIO-method. Figure 6.4(a), 6.4(b), 6.4(c),

6.4(d) and Figure6.4(e) sketches all the functions inside this component : the seven formal methods as a whole, and each individual formal method as an independent function which can be executed independently.

In order to realize the above functionalities, each formal method is implemented in a separate program. The seven programs are linked by the USER INTERFACE so that they can be executed consecutively upon user's request.

TSG also provided an option for the user : except the T-method and MUIO-method, the user can choose to apply his own checking sequences in all the other formal methods (re. Table 6.1).

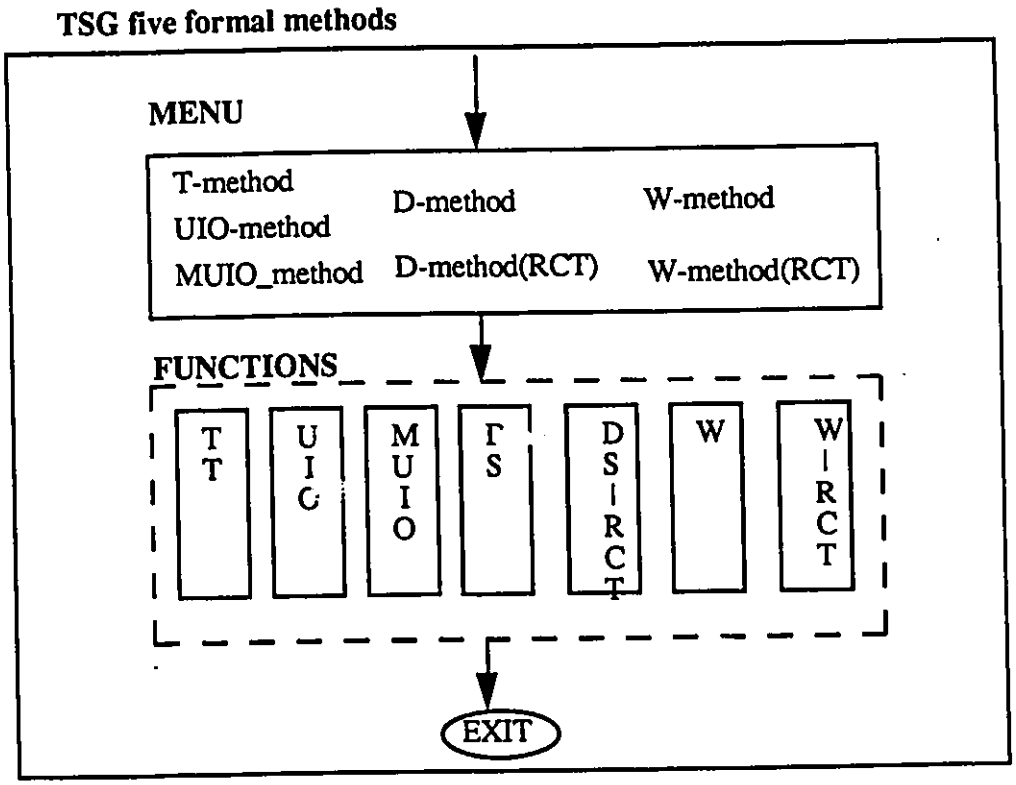


Figure 6.4(a). The Outline of the Five Formal Methods

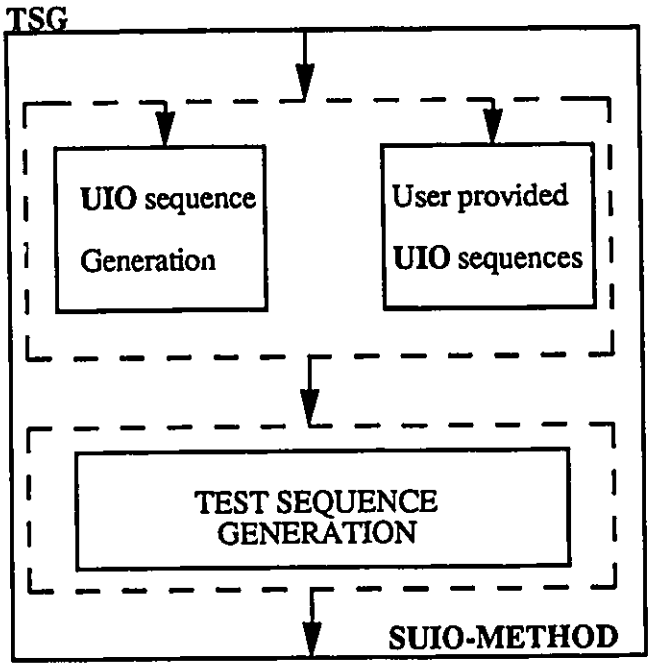


Figure 6.4(b) The SUIO-method

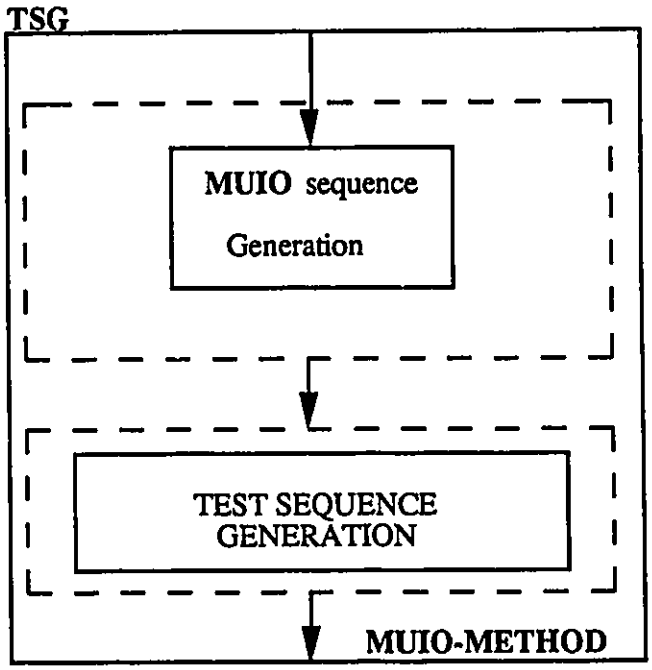


Figure 6.4(c) The MUIO-method

Note that the MUIO-method function in TSG does not have user provided MUIO selection.

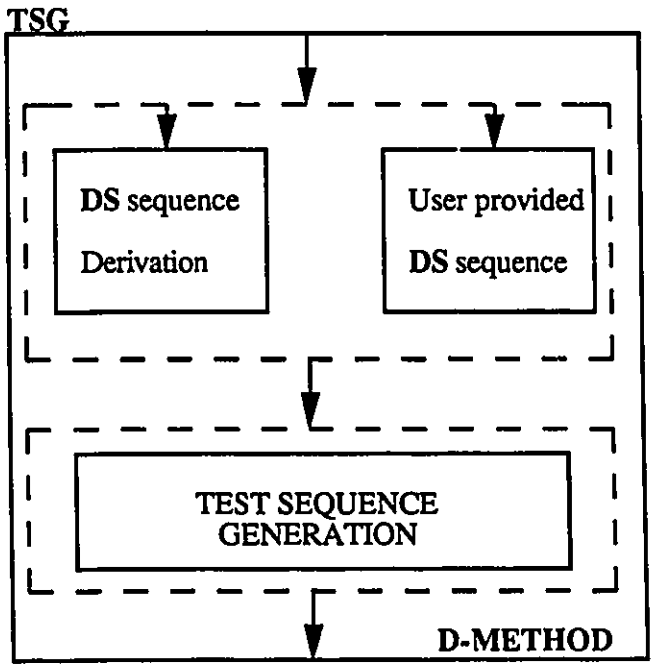


Figure 6.4(d) The D-method

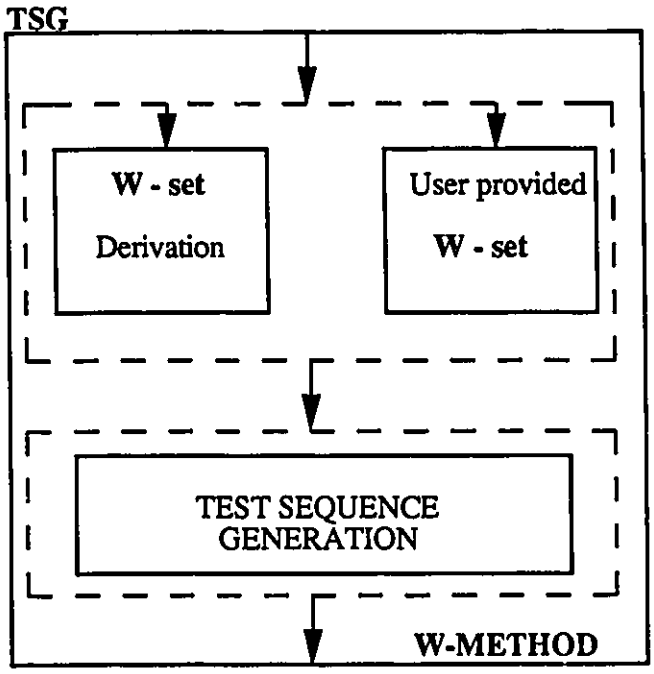


Figure 6.4(e) The W-method

The above two models are appliable for both D-method, W-method, and their variations.

6.3.3 Weak and Strong Conformance Test Sequences

In TSG, the resulting test sequence is represented by an input-output table. The input portion is the test sequence; the output portion specifies the expected output from the IUT [Appendix]. During testing, the test sequence is applied to the IUT at its input port, and the output from the IUT is observed at the output port of the IUT and compared with the expected output in the test table. If both outputs are identical, the IUT is said to conform to its specification. Otherwise, the IUT is said to have fault(s). As it is mentioned in Chapter 3, the protocol specification is modeled as a FSM. The specified edges in the FSM_S are refered to as **core edges**; the unspecified edges added as self-loops in the FSM_S are called **non-core edges**. Since most protocols are not completely specified, conformance is therefore defined at two levels : *weak* and *strong conformance* .

Definition 1 : An implementation has **strong conformance** to the specification if both the FSM_I and FSM_S generate the same outputs for all test sequences [SiLe86].

Definition 2 : An implementation has **weak conformance** to its specification if the FSM_I has the same intput/output behavior as the FSM_S with respect to the core edges only.

Table 6.1 lists types of conformance test sequences which TSG can generate based on the seven implemented formal methods. Weak conformance test sequences can be generated based on the D- and W-methods only if the user provides his/her own DS and W-set. Otherwise, only strong conformance test sequences can be generated based on these two formal methods including their variations. TSG can generate both weak and strong conformance test sequences based on the T-method, SUIO-method and MUIO-method no matter whether the user provides his/her own UIO sequences or not.

TSG Test Selection Table

Method	User Provided Checking Seq's Allowed	Type of Conformance Test Sequences Weak	Strong
T-Method	N/A	Yes	Yes
SUIO-Method	Yes	Yes	Yes
MUIO-Method	No	Yes	Yes
D-Method	Yes	Yes*	Yes
D-Method(RCPT)	Yes	Yes*	Yes
W-Method	Yes	Yes*	Yes
W-Method(RCPT)	Yes	Yes*	Yes

* -- Only if the User provides his/her own checking sequences, weak conformance test sequences will be generated.

Table 6.1 TSG Test Selection Table

6.4 Implemention Details of the Five Formal Methods

T-method, SUIO-method and MUIO-method Implementations

In TSG, the given FSM M does not have to be completely specified when applying T-method, SUIO-method or MUIO-method. As long as M is strongly connected, a tour can be found in M, which covers every transition of M. Users can choose to have test sequences generated from a partially specified M or a completely specified M. Note that the exception is that if some states of M do not have UIO sequences in the SUIO-method, M must be completed in order to generate alternative Unique Signatures for those states (re.Section 4.4.4).

In the MUIO-method, the graph G[Ec] may not be connected. If it is the case, there are two possible ways that are suggested in the thesis to deal with the situation :

1) Allow the use of self-loop UIO sequences; select only one minimum-length UIO sequence (minimal UIO) for the initial state s_S of M, denoted by UIO(s_S).

The reason why a fixed UIO(s_S) can avoid nonconnectivity of the graph G[Ec], provided that M has reset function, is that reset function brings M back to the initial state s_S after each transition being checked. According to the method, all the reset functions are also checked with UIO(s_S)'s to verify that they do take M back to s_S from every other state of M. In the graph G (representing M), since each vertex has an outgoing edge labeled with *ri*, terminating at s_S, a UIO(s_S) applied after each ri will take M to the tail state of the UIO(s_S). Since UIO(s_S) is fixed for all incoming reset edges, the graph G[Ec] is guarranteed to connect every vertex of M to the tail state of the UIO(s_S), and thus G[Ec] is weakly connected. Recall that each edge of G[Ec] is a concatenation of an edge of M with the UIO sequence for the tail state of that edge.

2) in TSG, we do not restrict self-loop UIO sequences in deriving MUIO's. If G[Ec] is not weakly connected, users are provided with the graph G[Ec] in a data file under the name of "graphEc". Users can review a hard copy of the G[Ec] and figure out how to add more edges into G[Ec] from M so that G[Ec] is weakly connected. After making changes to the G[Ec], the user can retry by following the instructions provided by TSG.

In TSG, only the second approach is implemented. The general procedure can be summarized as follows :

step1. compute minimal MUIO's for the given FSM M;

step2. construct the graph G[Ec];

step3. verify the connectivity of G[Ec] :

if G[Ec] is not weakly connected, then

output the G[Ec] to user, exit;

/* if obtain a modified G[Ec] from user, then

re-entry is at step3; */

step4. construct a RSA G* of G', where G' = (V, EUEc);

step5. find an Euler tour of G*;

step6. output the minimal test sequence (the Euler tour of G*).

Duplication problem is merely a min-cost max-flow problem in disguise as it is mentioned in the SUIO-method. The T-method is implemented by following the same approach. The only difference between the SUIO-method networks and the T-method networks is that the indices calculated in the T-method is based on graph G itself instead of being based on G[Ec] in the SUIO-method.

Consider the example for the T-method in Section 4.1.4, i.e., the FSM M given in Figure 4.1. The indices of all the vertices of M are listed on the right hand side of the LP equations from (1) to (5) on page 23. They represent the indices at vertex 1, 2, 3, 4 and 5 respectively. Among them, vertex1, 3 and 5 are intermediate nodes in the network with zero indices; vertex 2 with negative index connected to sink t; vertex 4 with positive index connected to source s. The resulting network model is shown below :

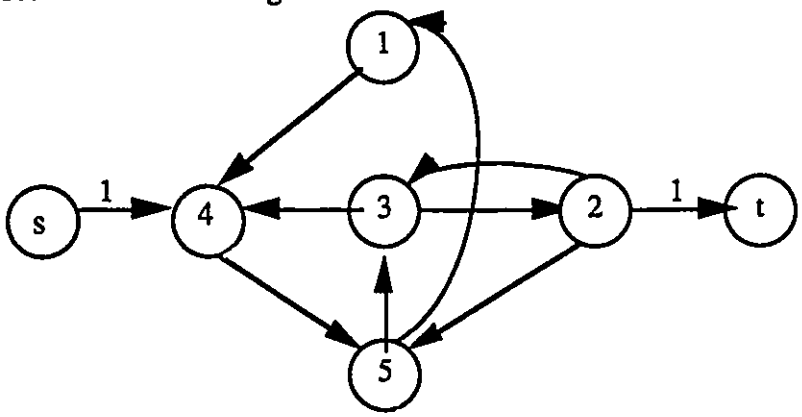


Figure 6.5 A Network Flow Model G_f for the FSM in Figure 4.1
(capacities are shown on those edges connected to s and t. All the other edges have infinite capacity).

In the above network, at most one unit flow can be sent from s to t, and the unit flow must leave s by way of node 4. At the same time, at most one unit flow can arrive at t by way of node 2. To leave node 4, only edge$_{45}$ can be used. To leave node 5, only edge$_{53}$ can be used. Therefore, the only path with min-cost max flow from s to t is :

s -> 4 -> 5 -> 3 ->2 -> t

The total min-cost max flow is one unit. The unit flow is attributed to edge e_{45}, e_{53}, and e_{32} respectively. That means that each of the three edges need to be replicated once in the Euler tour later, which is the minimal test sequence.

The above solution is the same as that of derived by LP algorithm shown in Section 4.1.3, but the latter is less efficiency.

Minimum-cost Maximum Flow Algorithm

TSG implements Busaker and Gowen *dual method* of min-cost flow algorithm to solve the duplication problem [SDK83]. The main idea of this algorithm is described as follows :

Given a network flow model G_f and a target flow F which is the total of the flow capacity on each edge which emenats from source s of G_f, first, find a shortest-path (least-cost path) from s to t by using Dijkstra's algorithm [Even79]; then, send the maximum amount of feasible flow along this path. To be accurate, let delta δ be the difference of the capacity and current flow on a edge in G_f. A maximum feasible flow is the minimum value of those δ's along that path. If the target flow value F is achieved, the program halts, the result is passed to the next function in TSG; otherwise, modify the network G_f by increasing δ unit flow on the forward edges in the path, decreasing δ unit flow on the backward edges in the path. Then a modified network G_f' is obtained. Continuously apply the above procedure on the modified G_f' until the target flow F is achieved or there is no more path left from s to t. An outline of the algorithm is given below :

```
begin
    Gf' <- Gf;
    while (flow value < F) and ( a path exists from s to t in Gf') do
    begin
        find a shortest path SP from s to t in Gf';
        if no such path exists then exit;
        send additional flow along SP until either SP is saturated or the flow reaches F;
        if SP saturates then obtain a modified network Gf'
    end
end
```

In our implementation, capacity, cost and flow are stored in three arrays representing the three matrices, respectively.

Euler Tour Algorithm

There are quite a few linear time algorithms exist for finding an Euler tour in a symmetric graph (G*). For example, Fleury's *isthmus-avoiding path* algorithm. Gonenc's minimal covering finding procedure is a variation of this algorithm. Others like *splitting algorithm* [BeWi83] and Edmonds' *end-pairing* algorithm [EdJo73], etc. In our implementation, Edmonds' approach is combined with Golumbic's algorithm [Golu80] to find an Euler tour of G*. The outline of the algorithm is given below :

```
begin
    Tour <- null;
    construct a spanning arborescence T of the G*;
    move all the edges of T to the end of the array where all the edges of G* stored;
    /* the above step is to ensure all the edges of T are traversed at last in the tour */
    Tour <- sS;                                        /* start traversal of G*  */
```

```
    cv <- s_S                          /* cv represents current state being visited */
    while (there are uncovered edges ) do
      select an unused edge e such that TAIL(e) = cv;
      /* meaning e coming into cv */
      Tour <- HEAD(e);
      cv <- HEAD(e);
    end
    output the above result sequence in the reverse order which is an Euler tour of G*;
end
```

The above algorithm starts at the initial state s_S of G* and traverses G* in the opposite direction of the edges until the the tour returns to s_S. All the edges of G* are visited in the order that they are stored in the array. This is equivalent to tracing a simple closed path C_1. If all the edges of G* are covered in C_1, stop the traversal. Otherwise, starting from an unused edge $e \in E^*$, such that TAIL(e) is already in C_1, trace another simple closed path C_2, and so on, until all the edges are covered by some C_i. All the C_i's are connected because of the way that they are exploited. Then output the entire trace in the reverse order, an Euler tour is thus obtained.

Implementation of Gonenc's D-Method

As mentioned in Section 4.2.5, Gonenc's approach does not consider optimal solutions for a nonconnected graph G[Ec] to obtain a complete test sequence. This thesis suggested the following three ways of tackling this problem :

(1) add reset function into G. This is proved to ensure a weakly connected G[Ec].

(2) add edges from G to G[Ec] until it is weakly connected (heuristics).

(3) generate separate test for each weakly connected component, and carry out the tests separately based on the components, which can also achieve the same goal of conformance testing.

TSG implemented proposal (1) and (2). The user has the choices of choosing either one.

Derivation of Minimum-length Distinguishing Sequences

Deriving a minimum-length Distinguishing Sequence (minimal DS for short) is the most time consuming task for an implementation of the D-method. It has the time complexity of $O(n^n + n^2k^2)$ [SaBo84], which is the highest among all the formal methods covered by this thesis. Deriving a minimal DS involves constructing and searching a DS-tree [Koha78]. BFS and DFS are considered two major algorithms in graph traversal techniques. BFS can be used for finding shortest paths in a graph, and is an efficient algorithm in finding a minimal DS for a FSM. Nevertheless, BFS tree (DS-tree) expands exponentially (k^n), it is impractical to implement (due to limited computer memory) for our real applications. DFS occupies much less memory but is very inefficient in finding minimal DS's. For the sake of both practicality and efficiency, we adopted the **branch-and-bound** technique in our implementation to find a minimal DS for a FSM. It is much more efficient than the straight forward DFS. The outline of the procedure is as follows :

(1) initially, assume the existence of a DS not necessarily with minimal length (if any, say, length = n(n-1)/2);

(2) starting from the root of the DS-tree, continue the search for another shorter DS (if any) in DFS manner. The root of a DS-tree is called an Uncertainty Vector (UV) consists of all states of a FSM [Koha78].

(3) at each level of the tree search, current possible solution is compared with the existing DS :

if another DS' is found and its length is shorter than that of DS, then DS is replaced by DS', continue;

(4) at any level of the search, if the length of the current input sequence (which equals to the depth of the tree at that level) is equal to or longer than that of DS', then stop this branch, start another branch from the root to seek another shorter DS.

The above procedure stops when the whole DS-tree is searched no matter whether M has a DS or not because there are finite number of states in a FSM and each DS is bound with length of n(n-1)/2 [Lee76]. As a matter of fact, we can choose n-1 as the actual bound for a minimum-length DS of a minimal FSM in the implementation. This is based on Lee's theory in which the lower bound (LB) on a DS for a non-i-mergeable minimal FSM is $\log_o n$, and the upper bound (UB) on a DS is n(n-1)/2, where o is number of distinct output in FSM. A FSM M is said **i-mergeable** if for each $i \in I$, there exists at least a pair of states s_j, s_k such that $\delta(s_j, i) = \delta(s_k, i)$, and $\lambda(s_j, i) = \lambda(s, i)$ [Lee76]. Since an i-mergeable FSM has no DS [Lee76], a FSM M which possesses a DS must be a non-i-mergeable M. n-1 is the maximum number of partitions possible for a FSM. It is a value between LB and UB of a DS. A real protocol represented by a FSM is usually sparse in terms of number of transitions with respect to the number of states. Those transitions are often distinct from each other at different state (re. TP Class 4 in Chapter 7). At each level of the DS-tree, every input symbol is applied to the M, and the output at each state of M is observed and grouped into different partitions if their output are different. The states of a FSM usually can be partitioned into at least two groups in the first level of the DS-tree; each additional level, is a refinement of the previous level [Koha78], and there are maximum n-1 levels are needed in this general case.

In TSG, only the information about current branch and the DS (if any) are stored and updated when necessary. Meanwhile, the user can choose to stop searching at any intermediate step when he feels satisfied with the current DS. This approach has been used to derive DS's for X.25 and TPclass4.

The purpose of using this branch-and-bound technique is to reduce the amount of search to be conducted for the optimal solution (a minimal DS). After each branching, lower bound

of a DS is computed. The next solution space to be searched is therefore reduced to only those input sequences with shorter length than that of the current DS computed.

CHAPTER 7 MINIMAL TEST SEQUENCES GENERATION FOR TRANSPORT PROTOCOL CLASS 4

This chapter reports on the applications of TSG to transport protocol class4 (TP4). Minimal test sequences generated by TSG are based on the seven formal methods described in Chapter 4 and Chapter 5.

7.1 NBS Class4 Transport Protocol

The National Institute of Science and Technology (former National Bureau of Standards NBS) has developed a set of standards for the transport layer in the OSI reference model. It consists of five classes (i.e., TP0, TP1, TP2, TP3 and TP4) based on the reliability of the underlying networks. The TP4 is the most complex transport layer protocol among the four classes [NBS83]. It ensures that data arrives at destinations in the order that they are sent, recovers from errors and losses occurring during transmission, etc. There are five types of service primitives in TP4 :

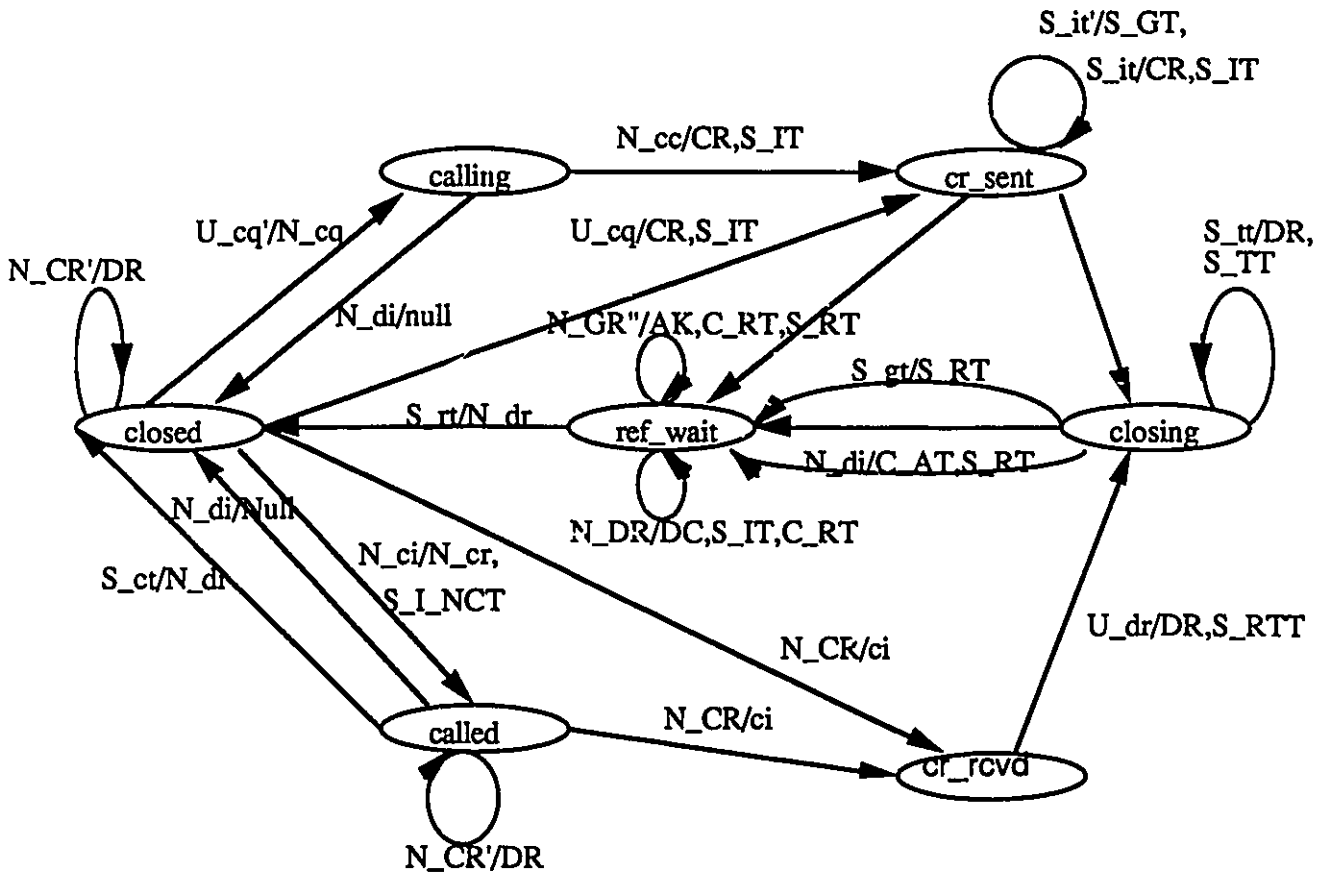
request, indication, response, confirm and **cancel**, and there are ten TPDUs :

CR, CC, DR, DC, GR, ERR, DT, XPD, AK and **XAK**.

We refer readers to [NBS83] for more details.

7.2 Minimal Test Sequence Generations for NBS TP4

The test sequence generator TSG is applied to a subset of the NBS TP4 shown in Figure 7.1(a) and (b). This subset excludes transitions for unit data, expedited data, data transfer and close request service primitives. The FSM M for this subset of NBS TP4 has sixteen states, fifty-nine core (specified) transitions. It is an incompletely specified FSM. There are twenty-seven distinct input symbols (service primitives and TPDUs). Table 7.1 lists the names and abbreviations for all the distinct inputs to the M across the user (U), network (N) and system interfaces (S). Table 7.2 lists the abbreviations for the output primitives used in TP4.



from CLOSING to REF_WAIT : N_DC/C_AT, S_RT, N_DR/C_AT, S_RT

from CR_SENT to REF_WAIT : S_gt/dc, S_RT; N_DR/DC, dc, C_AT, S_RT

from CR_SENT to CLOSING : N_CC'/DR, dc, S_TT, C_IT, C_GT

Figure 7.1(a) A Subset of NBS Class 4 Transport Protocol (con't in Fig 7.1b)

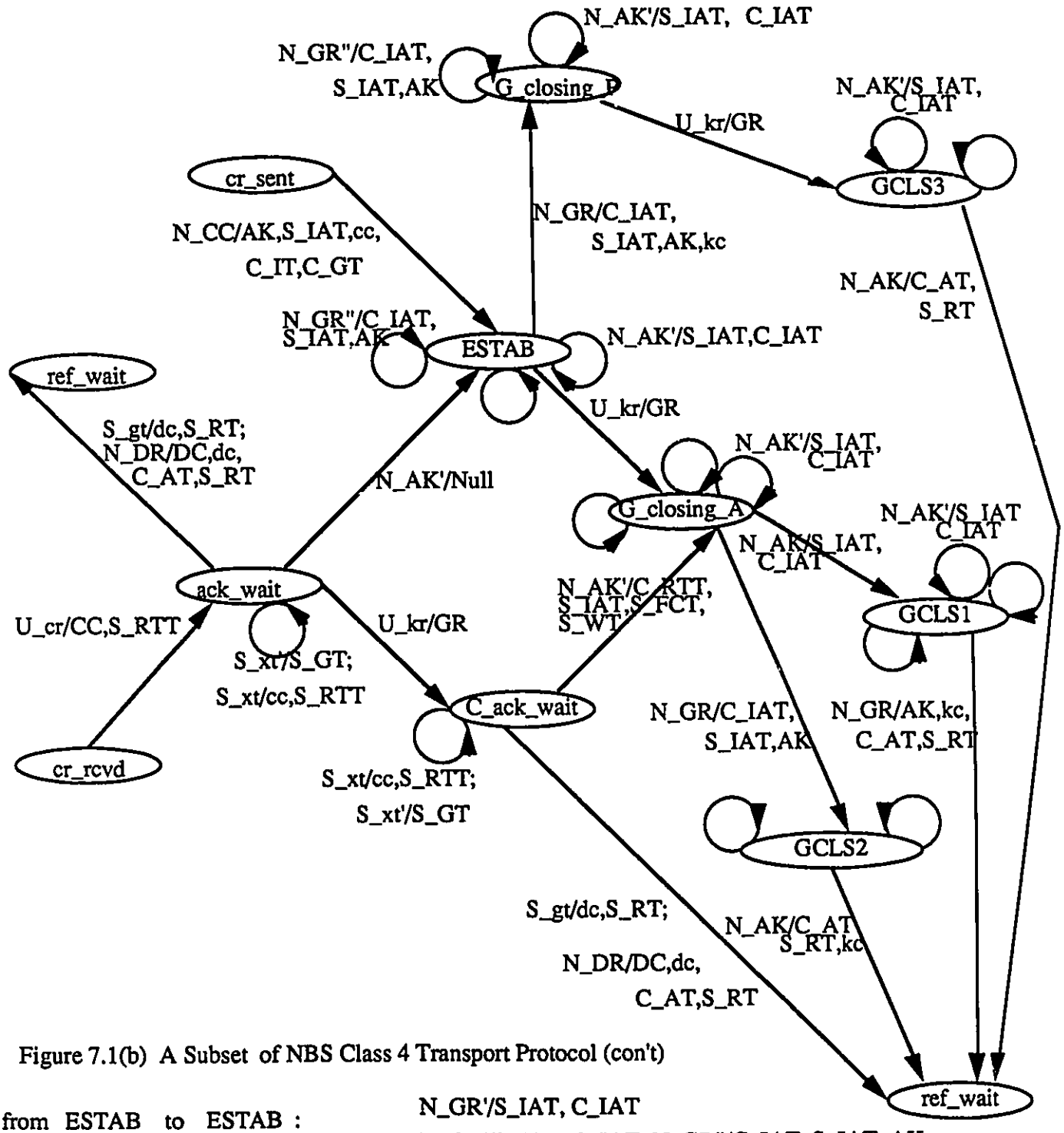


Figure 7.1(b) A Subset of NBS Class 4 Transport Protocol (con't)

from ESTAB to ESTAB :	N_GR'/S_IAT, C_IAT
from GCLS1 to GCLS1 :	N_GR'/S_IAT, C_IAT; N_GR"/C_IAT, S_IAT, AK
from GCLS2 to GCLS2 :	N_AK'/S_IAT, C_IAT; N_GR"/C_IAT, S_IAT, AK
from GCLS3 to GCLS3 :	N_GR"/C_IAT, S_IAT, AK
from G_closing_A to G_closing_A :	N_GR'/S_IAT, C_IAT; N_GR"/C_IAT, S_IAT, AK

INPUT PRIMITIVES

N_CR	from N: N_DATA.indication (connection_request)
N_CR'	from N: N_DATA.indication (bad CR)
N_CC	from N: N_DATA.indication (connection_confirm)
N_CC'	from N: N_DATA.indication (bad CC)
N_DR	from N: N_DATA.indication (disconnect_request)
N_DC	from N: N_DATA.indication (disconnect_confirm)
N_AK	from N: N_DATA.indication (acknowlegement)
N_AK'	from N: N_DATA.indication (AK and AK_ok)
N_GR	from N: N_DATA.indication (graceful_close_request and GR_arrived)
N_GR'	from N: N_DATA.indication (graceful_close_request and not GR_arrived and $cond_1$)
N_GR"	from N: N_DATA.indication (graceful_close_request and $cond_2$)
U_cq	from U: CONNECT.request
U_cq'	from U: CONNECT.request and nc_required
U_cr	from U: CONNECT.response
U_dr	from U: DISCONNECT.request
U_kr	from U: T_CLOSE.request
N_ci	from N: CONNECT.indication
N_cc	from N: CONNECT.confirm
N_di	from N: DISCONNECT.indication
S_tt	from S: S_TIMER.response (terminate_timer)
S_it	from S: S_TIMER.response (initiate_timer)
S_it'	from S: S_TIMER.response (initiate_timer) and $cond_3$
S_gt	from S: S_TIMER.response (giveup_timer)
S_rt	from S: S_TIMER.response (reference_timer)
S_xt	from S: S_TIMER.response (retransmit_CC_timer)
S_xt'	from S: S_TIMER.response (retransmit_CC_timer) and $cond_3$
S_ct	from S: S_TIMER.response (incoming_nc_timer)

Table 7.1 Abbreviations for the Inputs in TP4.

($cond_1$ is [from N: TPDU.tpdu_nr] ≤ rcv_window;

cond2 is [from N: TPDU.tpdu_nr] < rcv_nxt or >= rcv_window;

cond is [from S: Datum] >= rcount).

OUTPUT EVENTS

CR_TPDU (CR)
CC_TPDU (CC)
DR_TPDU (DR)
DC_TPDU (DC)
acknowlegement (AK)
set reference timer (S_RT)
set initiate timer (S_IT)
set giveup timer (S_GT)
set retransmit timer (S_RTT)
set terminate timer (S_TT)
cancel all timer (C_AT)
null output (Null)
N_connect_request (N_cq)
N_disconnect_request (N_dr)
N_connect_response (N_cr)
set incoming_nc timer (S_I_NCT)
connect.indication (ci)
connect.confirm (cc)
disconnect.confirm (dc)
cancel initiate timer (C_IT)
cancel giveup timer (C_GT)
cancel inactive timer (C_IAT)
set inactive timer (S_IAT)
set flow control timer (S_FCT)
set window timer (S_WT)
cancel retransmit timer (C_RTT)
cancel reference timer (C_RT)
close.confirm (kc)

Table 7.2 Abbreviations for the Outputs in TP4.

Minimal test sequences are generated based on the T-method, D-method, W-method, SUIO-method and MUIO-method. In applying T-method and two UIO-methods, TSG generates two types of conformance test sequences for TP4 (M) :

A weak conformance minimal test sequence for the original (partially specified) M;

A strong conformance minimal test sequence for a completely specified M.

TSG completes the given M according to the completeness assumption 1) in Section 3.2.1.

After M is fully specified, it contains 405 transitions.

The D-method and the W-method require M be complete, therefore only strong conformance minimal test sequences can be generated based on these two formal methods including their variations (i.e., the D- and W-method in combining with the RCPT approach).

A minimum-length DS and W-set derived for M by TSG are listed below :

DS : N_CR', U_cq', N_cc, N_AK', U_kr, N_DR, N_AK

W-set : N_cc, N_di, S_gt, N_DR, U_kr, N_GR, N_AK, N_CR'

We use the following natural numbers to represent different states of the FSM modeled TP4 :

1 CLOSED

2 CALLING

3 CR_SENT

4 ESTAB

5 ACK_WAIT

6 CR_RCVD

7 CLOSING

8 REF_WAIT

9 CALLED

10 C_ACK_WAIT

11 G_CLOSING_A

12 GCLS1

13 GCLS2

14 GCLS3

15 G_CLOSING_P

The following table (Table 7.3) lists all the UIO sequences for the states of TP4.

UIO sequences for TP4 :

STATE	UIO (I/O)	TAIL STATE
1	U-cq'/ N_cq	2
2	N-di / Null	1
3	S_it' / S_GT	3
4	N_GT / C_IAT,S_IAT,AK,kc	15
5	S_xt',U_kr / S_GT, GR	10
6	U_cr / CC, S_RTT	5
7	S_tt / DR, S_TT	7
8	S_rt / N_dr	1
9	N_di / C_I_NCT	1
10	N_AK' / C_RTT,C_GT,S_IAT,S_FCT,S_WT	11
11	N_AK / S_IAT, C_IAT	12
12	N_GR / AK, kc, C_IAT, S_RT	8
13	N_AK / C_AT, S_RT, kc	8
14	N_AK / C_AT, S_RT	8
15	U_kr, N_AK / GR, C_AT, S_RT	8

Table 7.3 A List of UIO's for TP4

A set of multiple UIO sequences for TP4 is listed below :

The Multiple UIO sequences for TP4 :

STATE	UIO (I/O)	TAIL STATE
1	U-cq'/ N_cq	2
1	U_cq / CR, S_IT	3
1	N_ci / N_cr, S_I_NCT	9
2	N-di / Null	1
2	N_cc / CR, S_IT	3
3	S_it' / S_GT	3
3	N_CC / AK,S_IAT, cc, C_IT, C_GT	4
3	N_CC' / DR, dc, S_TT, C_IT, C_GT	7
4	N_GT / C_IAT,S_IAT,AK,kc	15
5	S_xt',U_kr / S_GT, GR	10
5	U_kr, S_gt / GR, dc, S_RT	8
5	U_kr, N_AK'/ GR,C_RTT,C_GT,S_IAT,S_FCT,S_WT	11
6	U_cr / CC, S_RTT	5
6	U_dr / DR, S_TT	7
7	S_tt / DR, S_TT	7
7	S_gt / S_RT	8
8	S_rt / N_dr	1
8	N_GR" / AK, C_RT, S_RT	8
9	N_di / C_I_NCT	1
9	N_CR / ci	6
10	N_AK' / C_RTT,C_GT,S_IAT,S_FCT,S_WT	11
11	N_AK / S_IAT, C_IAT	12
11	N_GR / C_IAT, S_IAT, AK	13

12	N_GR / AK, kc, C_IAT, S_RT	8
13	N_AK / C_AT, S_RT, kc	8
14	N_AK / C_AT, S_RT	8
15	U_kr, N_AK / GR, C_AT, S_RT	8

Table 7.4 A Set of MUIO's for TP4

Table 7.5 gives a summary of the resulting test sequences generated by TSG based on the seven implemented formal methods. Minimal test sequences are generated by applying the T-method, SUIO-method and MUIO-method for both weak and strong conformance testing of the NBS TP4. Strong conformance test sequences for NBS TP4 using D-method (Gonenec), D-method (RCPT), W-method (Chow), and W-method (RCPT) are also generated by TSG.

TP4 Test Sequence Information Table

Method	Type of Test Sequences	TS - Length
T-Method	Weak	98
	Strong	443
SUIO-Method	Weak	207
	Strong	1320
MUIO-Method	Weak	196
	Strong	1232
D-Method	Strong	3135
D-Method(RCPT)	Strong	3934
W-method	Strong	17513
W-Method(RCPT)	Strong	9360

Table 7.5 TP4 Test Sequence Information Table

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This thesis describes the implementations of five formal methods for protocol conformance test sequence generation in the literature : T-method, D-method, W-method, SUIO-method and MUIO-method. Variations for the D-method and the W-method are also introduced in this thesis. The techniques for solving Chinese postman problem (CPP) is applied to the T-method and Rural Chinese postman problem (RCPP) is applied to other four formal methods in generating minimal test sequences for conformance testing. The RCPP has a low degree polynomial-time solution when the graph G[Ec] (which is the edge-induced subgraph of G') is weakly connected, provided that the FSM has reset or self-loop capability.

Among the five methods (considering only the variations of the D-method and the W-mehtod), T-method is the only one which stresses only controllability aspect of a test. This method has a limited test capability of only detecting output errors if the given FSM has no status message available. Whereas all other methods (i.e., D-method, W-method, SUIO-method and MUIO-method) emphasize both the controllability and observability aspects of a protocol. They have the capabilities of detecting both output errors and state transition errors.

The upper bounds on the length of test sequences generated by all the implemented formal methods are also given. A real protocol example is described.

8.2 Future Work

Developing formal methods of protocol conformance test sequence generation has been a very active research field. Much work is to be done to exploit the existing FSM based testing techniques. As far as this thesis is concerned, some of the possible future work could be related to :

futher optimizations of the existing formal methods. The following is a list of such optimizations :

1) Reduction of Distinguishing Sequence (DS)

   The reduction of a DS is based on a property of a DS. Suppose that the DS = 001 for some FSM. If δ(state1, 00) = state2 and δ(state1, 001) = state3, then δ(state2, 1) = state3, Therefore, the last transition from state2 to state3 with input "1" need not be checked in transition checking procedure, because it is checked in DS already. A similar idea can be applied to the SUIO-method.

2) Deletion of the Redundancies of the Subsequences of a Test Sequence

   For example, consider the FSM in Figure 4.8. In the SUIO-method, the tests TEST(v_1, v_2; b/x) = (b/x)(b/y) and TEST(v_2, v_3; b/y) = (b/y)(b/x)(c/z) can be optimized by partially overlapping these tests as one test : (b/x)(b/y)(b/x)(c/z), which tests two transition at once.

3) Optimization of W-set

   In line with the idea in 2), W-set can be optimized by combining some redundant subsequences. According to the definition of a W-set, it should be able to identify every pair of states of a given FSM. The idea of the optimization is based on the fact that if given two input sequences I_1 and $I_2@I_3$, I_1 is a subsequence of $I_2@I_3$, then I_1 can be omited since it is implied by $I_2@I_3$. Only $I_2@I_3$ need to be actually applied to the FSM.

   Consider the example FSM in Figure 4.1. One of its W-set is {w1, w2, w3 }, where

   w1 = {a}, w2 = {aa}, w3 = {b}.

   The multiple experimental tree for the FSM is given in Figure 8.1.

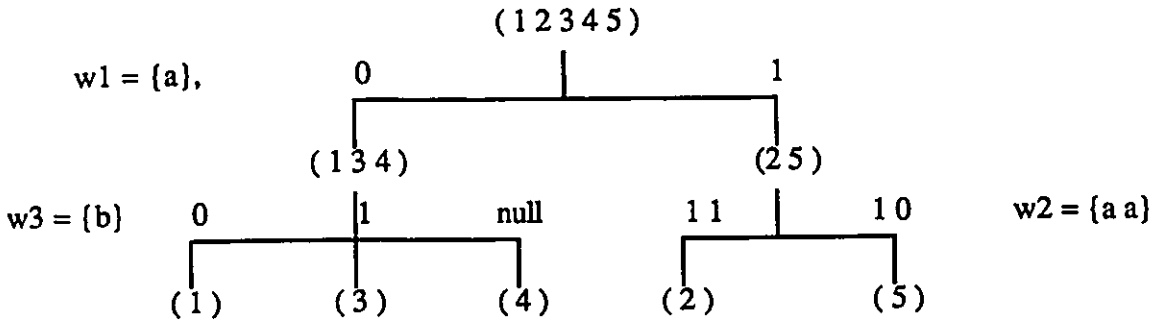


Figure 8.1 A Multiple Experiment Tree for the FSM in Figure 4.1

We can see from the tree that all the states are identified at the third level of the tree , and w2 ⊃ w1. If only w2 and w3 are applied to the FSM the result is then shown as below :

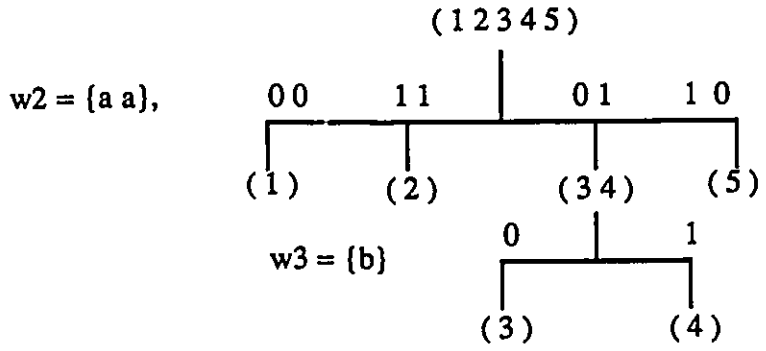


Figure 8.2 A Multiple Experiment Tree for the FSM in Figure 4.1.

It can be readily seen that w1 need not to be applied to the FSM, every pair of states of the FSM is still identified (i.e., all the leave nodes are singletons). Formally, let "every pair of FSM is identified" be a function on B and C, the result domain is A, then,

B -> A, C -> A => BUC -> AUA = A;

since C ⊃ B, we have:

B -> A.

As a result, the length of the final test sequence (TS) is significantly reduced. Still using the same example, but considering this time, W= {aa, b}, a minimal TS with 116 (including the initial ri) inputs for the FSM in Figure 4.1 derived by using Chow's approach of the W-method is given below compared with the 167 inputs in Table 4.3(b) :

ri ri aa ri b ri baa ri bb ri aaa ri ab ri aaaabaa ri aaaabb
ri aaaaaaa ri aaaaab ri aaaaaa ri aaaab ri aaabaa ri aaabb
ri abaa ri abb ri aaaa ri aab ri aabaa ri aabb ri aaaaa ri aaab

Total = 116 inputs

With this W-set, the minimal test sequence generated by using the RCPT approach of the W-method is with only 76 inputs for the FSM without reset feature, and 115 inputs with reset feature.

4) Optimal Selection of Checking Sequences

Another possible way of reducing the length of a TS is to select a DS, a set of UIO's and a W-set wisely. However, we usually do not see the effects of a particular selection until the test sequences are generated. Optimal selection of checking sequences is considered to be a hard problem. Of course, the optimal selection can be studied by some experimental approach. Whether any systematic approach exists is an open research problem.

5) Synchronization Problems

In the remote test architecture (re. Figure 2.6), formal mechanisms for the synchronization between the upper tester and lower tester are needed. It is shown in [SaBo84] that TSs can be checked for synchronization problems by making some reasonable assumptions and using modified formal methods. Their approach is only

applied to T-method and W-method [SaBo84]. The immediate result is longer TSs. Also, whether the intrinsic synchronization problems are avoidable for all times is not clear [SaBo84]. How to avoid the synchronization problems without losing applicabilities of the formal methods to real protocols is also an open research problem.

REFERENCES

[ADLU88] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar, "An Optimization Technique for Protocol Conformance Test Sequence Generation Based on UIO Sequences and Rural Chinese Postman Tours," Proc. of the 8th. Int'l Symp. on Protocol Specification, Testing, and Verification, North-Holland, ed. S. Aggarwal and K. Sabnani, 1988.

[BeKl84] J. Bergstra and J.W. Klop, "Process Algebra for Synchronous Communication," Information and Control, Vol.60, pp. 109-137, 1984.

[BeWi83] L.W. Beineke and R.J. Wilson, "Selected Topics in Graph Theory 2," Academic Press Inc., London, 1983.

[Boch80] G.V. Bochmann, "A General Transition Model for Protocols and Communication Services," IEEE Trans. on Comm., Vol. Com-28 , pp.643-650, April, 1980.

[BoSu83] G.V. Bochmann and C.A. Sunshine, "A Survey of Formal Methods," Computer Networks and Protocols," P.E. Green, ed., pp. 561-578, New York: Plenum Press, 1983.

[BuSa65] R.G. Busacker and T.L. Saaty, "Finite Graphs and Networks," New York : MacGraw-Hill Company, New York, 1965.

[Chow78] T.S. Chow, "Testing Software Designs Modeled by Finite-State Machines," IEEE Trans. on Software Engineering, Vol.SE-4, no.3, May 1978.

[EdJo73] J.Edmonds and E.L. Johnson, "Matching, Euler Tours and the Chinese Postman," Mathematical Programming, Vol.5, pp.80-124, 1973.

[EhMa85] H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specification 1," Springer-Verlag, Berlin, 1985.

[Even79] S. Even, "Graph Algorithms," Maryland : Computer Science Press, 1979.

[FoOs76] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," Comput. Surveys, Vol. 8, pp. 305-330, September 1976.

[Gill62] A.Gill, "Introduction to the Theory of Finite-state Machines," New York : MacGraw-Hill, 1962.

[Golu80] M.C. Golumbic, "Algorithmic Graph Thery and Perfect Graphs, " New York : Academic Press, 1980.

[Gone70] G. Gonenc, "A Method for the Design of Fault Detection Experiments," IEEE Trans., Comput., Vol.C-19, pp.551-558, Jan. 1970.

[Hara72] F. Harary, "Graph Theory," Addison-Wesley Publishing Company, New York, 1972.

[Henn64] F.C. Hennie, "Fault-detecting Experiments for Sequential Circuits," Proc. of the 5th Ann. Symp. on Switching Circuit Theory and Logical Design, pp.95-110, Nov. 1964.

[Howd80] W.E. Howden, "Functional program testing," IEEE Trans. on SE, Vol. SE-6, No. 2, pp. 162-169, 1980.

[ISO8807] International Standard of LOTOS, A FDT Based on Temporal Ordering of Observational Behavior, ISO TC97/SC21 IS8807.

[ISO9074] International Standard of Estelle, A FDT Based on an Extended State Transition Model, ISO TC97/SC21 IS9074.

[ISO9646] ISO/TC97/SC21, OSI conformance testing methodology and framework - Part 1-5, ISO 2nd DP9646-1 revised text, edited by D. Rayner, Vancouver, December 1987.

[Koha78] Z. Kohavi, "Switching and Finite Automata Theory," New York : Mcgraw-Hill, 1978.

[Kuan62] M-K. Kuan, "Graphic Programming Using Odd or Even Points," Chinese Math., Vol.1 pp. 273-277, 1962.

[Lee76] S.C. Lee, "Digital Circuit and Logic Design," Prentice-Hill, Inc. Englewood Cliffs, New Jersey, 1976.

[LeKa76] J. K. Lenstra and A.H.G. Rinnooy Kan, "On General Routing Problems," Networks, Vol.6, pp.273-280, 1976.

[NaTs81] S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours," Proc. of the 11th IEEE Fault Tolerant Compt. Symp., IEEE Comput. Soc., Press, pp. 234-243, 1981.

[NBS83] "Specification of a Transport Protocol for Computer Communication, vol.4 : Protocol Specifications," Report ICST/HLNP-83-84, National Bureau of Standards, Washington, D.C., Jan. 1983.

[Rayn87] D. Rayner, "OSI Conformance Testing," Computer Networks and ISDN Systems, Vol. 14, pp. 79-98, 1987.

[SaBo82] B. Sarikaya and G.V. Bochman, "Some Experience with Test Sequence Generation," Proc. of the Second Int'l Workshop on Protocol Specification, Tesing, and Verification," North-Holland, ed. C. Sunshine, 1982.

[SaBo84] B.Sarikaya and G.V. Bochmann, "Synchronization and Specification Issues in Protocol Testing," IEEE Trans. on Comm., Vol. Comm-32, pp. 389-395, April 1984.

[SBC87] B. Sarikaya, G.V. Bochmann and E. Cerny, "A Test Design Methodology for Protocol Testing," IEEE Trans. on Software Eng., pp.531-540, May 1987.

[SaDa85] K.K. Sabnani and A.T. Dahbura, "A New Technique for Generating Protocol Tests," Proc. of the 9th Data Communications Symp., IEEE Computer Soc. Press, pp.36-43, September 1985.

[SaDa88] K. Sabnani and A. Dahbura, "A Protocol Test Generation Procedure," Computer Networks and ISDN Systems Vol. 15, pp.285-297, 1988.

[SaTi87] R. Saracco and P.A.J. Tilanus, "CCITT SDL : Overview of the Language and Its Applications," Computer Networks and ISDN Systems Vol. 14, pp. 65-74, 1987.

[SDK83] M.M. Syslo, N. Deo and J.S. Kowalik, "Discrete Optimization Algorithms," New Jersey : Prentice-Hall, Inc., Englewood Cliffs, 1983.

[SiLe86] D. Sidhu and T-K. Leung, "Formal Methods for Protocol Testing : A Detailed Study," in Proc. IEEE INFOCOM'88, pp. 68-72, 1986.

[SLD89] Y.N. Shen, F. Lombardi and A.T. Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences," in Protocol Specification, Testing and Verification, IFIP WG 6.1 Inter'l Symp., June 1989, ed., E. Brinksma, G. Scollo, and C.A. Vissers.

[Ural87] H. Ural, "Test Sequence Selection Based on Static Data Flow Analysis," Computer Communications, Vol. 10, No. 5, pp. 234-242, October 1987.

[UyDa86] M.U. Uyar and A.T. Dahbura, "Optimal Test Sequence Generation for Protocols : the Chinese Postman Algorithm applied to Q.931," Proc. of IEEE Global Telecommunications Conference, 1986.

[Zimm80] H. Zimmermann, OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, IEEE Trans. on Comm. Com-28(4), pp. 425-432, 1980.

APPENDIX

TSG USER'S MANUAL

(Version 1.0)

Department of Computer Science

University of Ottawa,

Ottawa, Ontario, Canada K1N 6N5

I. INTRODUCTION

TSG (Test Sequence Generator) is a software package which can generate test sequences from the protocol specifications modeled as finite state machines (FSM). The generation of test sequences is based on the following formal methods :

1. Transition Tour Method (T-method),
2. Distinguishing Sequence Method (D-method),
3. Characterizing Set Method (W-method),
4. Unique Input/Output Method (Single UIO-method or SUIO-method), and
5. Multiple UIO-Method (MUIO-method).

TSG also implements two variations of the D-method and W-method. Since the Rural Chinese Postman Tour (RCPT) approach is combined with these two methods in deriving minimal test sequences, the variations are called :

6. D-method (RCPT), and
7. W-method (RCPT).

TSG can generate the following types of test sequences :

1) weak and strong conformance minimal test sequences based on the T-method, SUIO-method and MUIO-method,
2) strong conformance test sequences based on the D-method (Gonenc) and W-method (Chow),

3) minimal test sequences based on the D-method and W-method in combination with the Rural Chinese postman Tour (RCPT) approach,

4) weak conformance test sequences based on the D-method (Gonenc) and/or the W-method (Chow), and minimal weak conformance test sequences based on the D- and/or W-methods in combination with the RCPT approach with user provided Distinguishing Sequence (DS) and/or W-set.

Table A.1 lists types of conformance test sequences which TSG can generate and the user's options for checking sequences.

TSG Test Selection Table

Method	User Provided Checking Seq's Allowed	Type of Conformance Test Sequences Weak	Strong
T-Method	N/A	Yes	Yes
SUIO-Method	Yes	Yes	Yes
MUIO-Method	No	Yes	Yes
D-Method	Yes	Yes*	Yes
D-Method(RCPT)	Yes	Yes*	Yes
W-Method	Yes	Yes*	Yes
W-Method(RCPT)	Yes	Yes*	Yes

* -- Only if the User provides his/her own checking sequences, weak conformance test sequences will be generated.

Table A.1 TSG Test Selection Table

TSG has a friendly user interface. The entire processing is self-explanatory. By following the menus and prompts provided by TSG, a user can easily apply the package without using this manual. The Manual, however, intends to provide a reference for the user.

Software Capabilities

In the following, the upper limits for the input data to the five formal methods (seven implementations) are listed:

METHOD	(N, M, K)
T-method	(100, 10000, 100)
D-method	(50, 2500, 100)
W-method	(50, 2500, 100)
UIO-method	(50, 2200, 100)
MUIO-method	(50, 2200, 100)
D-method(RCPT)	(50, 2200, 100)
W-method(RCPT)	(50, 2200, 100)

where

N : represents the maximum number of states allowed for a FSM.

M : represents the upper limit for the number of transitions in a FSM.

K : represents the maximum number of input symbols allowed for a FSM.

II. STARTING UP

TSG currently runs on SUN workstations and is portable to any other systems supporting the C language. TSG consists of a User Interface function and seven other functions to realize the five formal methods of protocol conformance test sequence

generation. Each function is implemented as a separate C program which can run either separately or jointly over the SUN workstation.

To invoke TSG, simply type in "tsg" and press RETURN. The screen will then show the following information :

********** Protocol Conformance Testing **********

-- Test Sequence Generator --

(Version 1.0)

. Department of Computer Science,

University of Ottawa

****

which indicates that the program has started.

TSG first accepts a FSM in the form of transitions, storing it in a data file under the name of "**fsm**". The "fsm" is an editable text file. Then, four conditions are checked and (one of) the five formal methods (is) are applied to generate conformance test sequence(s) automatically depending on the user's choice.

III. INPUTING A FSM-MODEL OF A PROTOCOL

3.1 INPUT PHASE

The user will have to answer the following query first :

"Have you had your FSM ready in a file named 'fsm'(y/n)?"

If the answer is "y" meaning "yes", TSG will enter the phase IV;

If the answer is "n" meaning "not yet", TSG gives the user the following choices :

1) Input the FSM from keyboard through screen interaction;

2) Input the FSM to a file named '**fsm**' by using the system editor .

If the user makes the first choice, he is prompted to input all the transitions of the FSM:

Please input the FSM by following the prompts below :

(please input all the transitions emanating from the same state consecutively.)

(use natural numbers to represent the states of the FSM, starting from 1)

Head state : 1

Tail state : 3

Input symbol : a

Output symbol : b

.....

where the underlined information is required to be entered by the user. Each transition of the FSM is entered to the system in the above format.

If the user makes the second choice, the screen will jump to edit status. The user should use "vi' command [Commands Reference Manual] to enter and/or edit his input. Every transition should be entered into a file called 'fsm' in the following format :

1

3

a

b

.....

After the FSM is saved (under the name "fsm" by default), TSG will bring user back to the end of the phase III. Upon the user's request, TSG will display all the transitions entered to the system.

Note that the user should not name one input symbol as part of another distinct input symbol, for example, CRq and CRq'. Also, the user should not use comma "," as part of an input symbol since it will cause confusion in C string processing.

3.2 EDIT PHASE

If the user selectes the interaction mode for inputing his FSM, the input FSM can be changed after each transition is entered to the system : if the user found the current transition is not correct, he can simply re-enter the correct transition. After all the transitions are entered to the system, an "Input Menu" is displayed on the screen :

********** Input Menu **********

(For transition modifications)

Modification ..... (1)

Deletion ..... (2)

Insertion ..... (3)

Display ..... (4)

Continue ..... (C)

****

The user can choose to modify a transition, delete a transition and/or insert a new transition by selecting (1), (2), and/or (3). Display means to display all the transitions. To proceed with the execution of TSG, user selects Continue (C).

IV. CHECKING THE FOUR CONDITIONS

After the FSM is entered to the system, TSG will check the following four conditions required by the five formal methods :

1) deterministic,

2) strongly connected

3) completely specified, and

4) minimal.

If the FSM is non-deterministic and/or not strongly connected, the user will be notified, and the execution will be aborted. In the case that 3) and/or 4) are/is not satisfied, the user has choices to either complete and/or minimize the FSM or not, since the T-method, SUIO-

method and MUIO-method do not require that a FSM satisfy condition 3) and 4). Then the weak conformance test sequence(s) will be generated for the FSM. To complete a FSM, the user can choose to complete the FSM himself or by the tool. In both cases, the TSG will provide the user a complete list of all the unspecified inputs of the given FSM.

After the FSM is checked the above four conditions, it is kept in another data file called "Fsm". The "Fsm" is used for generating test sequences. The "Fsm" can be edited by using "vi" Editor's commands, and can be reused as many times as the user wants. The first line of the file "Fsm" is number of states in it, the second line specifies the number of transitions in the "Fsm".

V. INVOKING THE FIVE FORMAL METHODS

5.1 Integrated Execution of the Seven Implemented Formal Methods

TSG implements seven approaches based on the five formal methods. Two of the seven are the variations of the D-method and the W-method (Table A.1). Hereinafter, we call them seven formal methods.

After the successful completion of the phase IV, the system will show the following menu :

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

This menu assists the user to access the tools to generate seven different conformance test sequences by using seven formal methods. The choice (4) is actually a variation of the (3), and the choice (6) is a variation of the (5).

The entire process is self-explanatory and easy to follow.

In executing choice (5) (W-method of Chow's), the user can assume that the number of states in IUT is equal to the number of states in the SPEC (FSM-modeled specification) [Chow78, APPENDIX A.1].

Any time at this stage, the user can press **CLT-C** to interrupt the current execution, and will be brought back to the above menu.

The resulting test sequence of the Gonenc's D-method ("fdsG") is output to a file called "**TestsqG**"; the resulting test sequence of the Chow's W-method ("fwsc") is output to a file called "**WTS**" ; all the test sequences derived by using the rest of the formal methods are sharing a file named "**TestSequence**". The user should be very careful to make copies of the current result(s), when needed, before he plans to run another example, or apply another formal method which uses the output file under the same name.

5.2 Executions of the Seven Formal Methods Independently

Each subfunction of TSG which implements a formal method of test sequence generation can be executed independently without following the routine of the TSG (i.e., do not use the User Interface function, to invoke the corresponding C programs directly). The commands for invoking the seven C programs are listed below :

1) T-method : ttour ;

2) D-method (Gonenc) : dsG or
   fdsG if the user provides his own DS in a file called "dss";

3) D-method (RCPT) : dsg or
   fds if the user provides his own DS in a file called "dss";

4) W-method (Chow) : wsetc or

fwsc if the user provides his own W-set stored in a file called "wss";

5) W-method (RCPT) : wset or

fws if the user provides his own W-set stored in a file called "wss";

6) SUIO-method : usq or

fuio if the user provides his own UIO sequences in a file called "uios";

7) MUIO-method : muio or

fuiom if the user provides his own MUIO sequences in a file called "muios".

All the input and output files of TSG can be edited (set-up and/or modified) by using Unix Editor "vi" commands.

VI. THE OUTPUT TEST-TABLE

A test sequence generated by TSG is output to screen and to a file in the format of a Test-Table. In the following, we introduce different Test-Tables for different formal methods. Note that the RCPT-based approaches (i.e., the SUIO-, MUIO-, D- and W-methods) make use of the same output file under the name "TestSequence" as they are mentioned in Section V. The user should make copies before he applies another example or different formal methods, if he wants to keep all the resulting test sequences.

6.1 The Test-Table Heading for the T_Method

A Minimal Test Sequence

State Sequence Input / Output Sequence

For examples please refer to [Appendix A.1] of the thesis.

6.2 The Test-Table Headings for the D-, SUIO- and MUIO-Methods

The Test Sequence Table

Present State Transition (I/O) Tail State Xd-sequence (I/O) Tail (Xd)

where the first three columns are the information about a transition to be tested : Head, I/O, Tail; the "Xd-sequence" is a DS on D-method, a UIO sequence and one of the MUIO sequence for the "Tail" of the current transition in the SUIO-method and the MUIO-method, respectively.

6.3 The Test-Table Heading for the W-method

A Test Sequence Table

Present State Transition (I/O) Tail State Tran@W-set (I/O) Tail (Tran@W-set)

where the column four is the concatenation of the transition (I/O) specified in column 2 with a W-set of the given FSM. The last column lists all the tail states of the subsequences of the concatenation of Tran@W-set. For examples please refer to [Appendix A.1] in this thesis.

[Appendix A.1] shows a sample run of TSG. The FSM is the one depicted in Figure4.1 of the thesis.

APPENDIX A.1 A SAMPLE RUN OF THE TSG

The example : the FSM depicted in Figure 4.1 of the thesis.

csid% **tsg**

******* Protocol Conformance Testing *******

-- Test Sequence Generator --

(Version 1.0)

Department of Computer Science

University of ottawa

****

Please press any key to continue.

Have you had your FSM ready in a file named 'fsm' (y / n) ? n

Please make your choice :
Input the FSM via Keyboard -- 1.
Input the FSM via the Editor -- 2 : 1

Please input the FSM by following the prompt below:

(please input all the transitions emanating from the same state consecutively.)

(use natural numbers to represent the states of the FSM, starting from 1)

Head state(enter 'CR' to end the input) : 1

Head: 1

Tail state:1

Tail: 1

Input symbol of this transition :
b

Input symbol : b

Output symbol :
null

Output symbol : null

Is this transition correct(y/n) ? y

Head state :1

Head: 1

Tail state:5

Tail: 5

Input symbol of this transition :
a

Input symbol : a

Output symbol :
0

Output symbol : 0

Is this transition correct(y/n) ? n

Please re_enter the correct transition:

Head state :1

Head: 1

Tail state:4

Tail: 4

Input symbol of this transition :
a

Input symbol : a

Output symbol :
0

Output symbol : 0

Head state :2

Head: 2

Tail state:3

Tail: 3

Input symbol of this transition :
b

Input symbol : b

Output symbol :
1

Output symbol : 1

Is this transition correct(y/n) ? y

Head state :2

Head: 2

Tail state:5

Tail: 5

Input symbol of this transition :
a

Input symbol : a

Output symbol :
1

Output symbol : 1

Is this transition correct(y/n) ? y

Head state :3

Head: 3

Tail state:2

Tail: 2

Input symbol of this transition :
a

Input symbol : a

Output symbol :
0

Output symbol : 0

Is this transition correct(y/n) ? y

Head state :3

Head: 3

Tail state:4

Tail: 4

Input symbol of this transition :
b

Input symbol : b

Output symbol :
0

Output symbol : 0

Is this transition correct(y/n) ? y

Head state :4

Head: 4

Tail state:4

Tail: 4

Input symbol of this transition :
b

Input symbol : b

Output symbol :
1

Output symbol : 1

Is this transition correct(y/n) ? y

Head state :4

Head: 4

Tail state:5

Tail: 5

Input symbol of this transition :
a

Input symbol : a

Output symbol :
0

Output symbol : 0

Is this transition correct(y/n) ? y

Head state :5

Head: 5

Tail state:1

Tail: 1

Input symbol of this transition :
b

Input symbol : b

Output symbol :
1

Output symbol : 1

Is this transition correct(y/n) ? y

Head state :5

Head: 5

Tail state:5

Tail: 5

Input symbol of this transition :
a

Input symbol : a

Output symbol :
0

Output symbol : 0

Is this transition correct(y/n) ? y

Head state :

There are 10 transitions in this machine.

(1) 1 -> 1 b null

(2) 1 -> 4 a 0

(3) 2 -> 3 b 1

(4) 2 -> 5 a 1

(5) 3 -> 2 a 0

(6) 3 -> 4 b 0

(7) 4 -> 4 b 1

(8) 4 -> 5 a 0

(9) 5 -> 1 b 1

(10) 5 -> 5 a 0

********** Input Menu **********

(For transition modifications)

Modification ..... (1)

Deletion ..... (2)

Insertion ..... (3)

Display ..... (4)

Continue ..... (C)

****

Please Input Your Choice : 1

Please enter the order No. of the transition which you want to correct : 10

(10): 5 -> 5 a 0

Is this the transition that you want(y/n)? y

Please re_enter the correct transition:

Head state :5

Head: 5

Tail state:3

Tail: 3

Input symbol of this transition :
a

Input symbol : a

Output symbol :
1

Output symbol : 1

There are more(y/n) ? n

********** Input Menu **********

(For transition modifications)

Modification ..... (1)

Deletion ..... (2)

Insertion ..... (3)

Display ..... (4)

Continue ..... (C)

****

Please Input Your Choice : 4

There are 10 transitions in this machine.

(1)	1 -> 1 b	null
(2)	1 -> 4 a	0
(3)	2 -> 3 b	1
(4)	2 -> 5 a	1
(5)	3 -> 2 a	0
(6)	3 -> 4 b	0
(7)	4 -> 4 b	1
(8)	4 -> 5 a	0
(9)	5 -> 1 b	1
(10)	5 -> 3 a	1

********** Input Menu **********

(For transition modifications)

Modification ..... (1)

Deletion ..... (2)

Insertion ..... (3)

Display ..... (4)

Continue ..... (C)

****

Please Input Your Choice : c

NO. of states in the given design is 5.

Do you want the completeness and minimality to be checked for this machine (y/n) ?

Note that M must be complete and minimal for deriving DS and W-set !
(y/n) : y

(1)(2)(5)(3)(4)

The given FSM is a minimal !

NO. of states is: 5;
NO. of transitions is: 10.

This machine satisfies the four conditions now.

Please press any key to continue.

There are Seven Formal Methods which can be used to generate test sequence(TS) automaticly.

Please follow the menu and select one of them to apply.

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

Your choice : 1

Does this machine have 'Status Message'(y/n) ? n

A Minimal Test Sequence :

States sequence	Input | Output sequence
1	a | 0
4	b | 1
4	a | 0
5	a | 1
3	b | 0
4	a | 0
5	a | 1
3	a | 0
2	b | 1

```
3               a | 0
2               a | 1
5               b | 1
1               b | null
1
```

Test Sequence Length : 13

Please press any key to continue.

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

Your choice : 2

Do you want to use your own UIO_sequences(y/n)? n

Please press any key to continue.

Computing the UIOs, please wait...

-- SUIO Method --

Does this Machine have Reset(ri) function(y/n)? n

Do you want Reset function to be added into the FSM(y/n)? n

The Test Sequence Table

Present State	Transition(I/O)	Tail State	UIO_sequence(I/O)	UIO-Tail
1	ri | null	1		
1	a | 0	4		
4	a | 0	5		
5	b | 1	1	b | null	1
1	a | 0	4		
4	a | 0	5		
5	a | 1	3	b | 0	4
4	b | 1	4	bb | 11	4
4	a | 0	5		
5	a | 1	3		
3	b | 0	4	bb | 11	4
4	a | 0	5		
5	a | 1	3		
3	a | 0	2	bb | 10	4
4	a | 0	5		
5	a | 1	3		

3	a | 0	2		
2	b | 1	3	b | 0	4
4	a | 0	5	bb | 1null	1
1	a | 0	4	bb | 11	4
4	a | 0	5		
5	a | 1	3		
3	a | 0	2		
2	a | 1	5	bb | 1null	1
1	b | null	1	b | null	1

Please press any key to continue.

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

Your choice : 6

Do you want to use your own W-set (y/n) ? n

Please press any key to continue.

********** A W-set for the given FSM : *********

The length of the W-set:

3
b
a
b,b

-- W_Method --

Does this Machine have Reset(ri) function (y/n)? n

Do you want Reset function to be added into the FSM (y/n)? n

A Test Sequence Table:

Present State	Transition(I/O)	Tail State	Transition@Wset_sequence(I/O)	Tail
1	ri / null	1		
1	null	-	b / null	1
1	null	-	a / 0	4
1	null	-	abbb / 01nullnul	1
1	a / 0	4		
4	a / 0	5		
5	b / 1	1	bbaabaabbb / 1null001001nullnull	1
1	a / 0	4		
4	a / 0	5		
5	a / 1	3	abaaaaabb / 100101101	4
4	b / 1	4	bbbababbb / 111010111	4
4	a / 0	5		
5	a / 1	3		

3	b | 0	4	bbaabaabbb | 0101001011	4
4	a | 0	5		
5	a | 1	3		
3	a | 0	2	abaaaabb | 01011010	4
4	a | 0	5		
5	a | 1	3		
3	a | 0	2		
2	b | 1	3	bbaaababbb | 1001010101	4
4	a | 0	5	abaaababb | 01001001null	1
1	a | 0	4	ababaababb | 0101001011	4
4	a | 0	5		
5	a | 1	3		
3	a | 0	2		
2	a | 1	5	abaaaaaaaabb | 11001011011null	1
1	b | null	1	bbbaabbbb | nullnullnull001nullnullnull	1

Please press any key to continue.

```
****************** MENU ******************

                T-method ..... (1)

                SUIO-method ..... (2)

                D-method (Gonenc)..... (3)

                D-method (RCPT)..... (4)

                W-method (Chow)..... (5)

                W-method (RCPT)..... (6)

                MUIO-method ..... (7)

                EXIT ..... (8)

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

        Your choice : 5

    Do you want to use your own W-set (y/n)? n

Please press any key to continue.


    ********** A  W-set for the given FSM : *********

            The length of the W-set:

                3
                b
                a
                b,b


            -- W_Method --

    There are 5 distinct states in the SPEC.
    Please give your appropriate guess on the possible No. of states in IUT( more than or
     equal to that of SPEC)

    : 5
```

A Test Sequence Table:

Present State	Transition(I/O)	Tail State	Transition@Wset_sequence(I/O)	Tail
1	ri | null	1		
1	null	-	rib | -null	1
1	null	-	ria | -0	4
1	null	-	ribb | -nullnull	1
1	b | null	1	ribb | -nullnull	1
1	b | null	1	riba | -null0	4
1	b | null	1	ribbb | -nullnullnull	1
1	a | 0	4	riab | -01	4
1	a | 0	4	riaa | -00	5
1	a | 0	4	riabb | -011	4
2	b | 1	3	riaaaabb | -001010	4
2	b | 1	3	riaaaaba | -001010	2
2	b | 1	3	riaaaabbb | -0010101	4
2	a | 1	5	riaaaaab | -001011	1
2	a | 1	5	riaaaaaa | -001011	3
2	a | 1	5	riaaaaabb | -001011null	1
3	a | 0	2	riaaaab | -00101	3
3	a | 0	2	riaaaaa | -00101	5
3	a | 0	2	riaaaabb | -001010	4
3	b | 0	4	riaaabb | -00101	4
3	b | 0	4	riaaaba | -00100	5
3	b | 0	4	riaaabbb | -001011	4
4	b | 1	4	riabb | -011	4
4	b | 1	4	riaba | -010	5
4	b | 1	4	riabbb | -0111	4

4	a | 0	5	riaab | -001	1
4	a | 0	5	riaaa | -001	3
4	a | 0	5	riaabb | -001null	1
5	b | 1	1	riaabb | -001null	1
5	b | 1	1	riaaba | -0010	4
5	b | 1	1	riaabbb | -001nullnull	1
5	a | 1	3	riaaab | -0010	4
5	a | 1	3	riaaaa | -0010	2
5	a | 1	3	riaaabb | -00101	4

Please press any key to continue.

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

Your choice : 7

Please press any key to continue.

-- Multi_UIO Method --

Does this Machine have Reset(ri) function(y/n)? n

Do you want Reset function to be added into the FSM(y/n)? y

The Test Sequence Table

Present State	Transition(I/O)	Tail State	UIO_sequence(I/O)	UIO-Tail
1	ri | null	1		
1	a | 0	4		
4	a | 0	5		
5	ri | -	1	b | null	1
1	a | 0	4		
4	ri | -	1	b | null	1
1	a | 0	4		
4	a | 0	5		
5	a | 1	3		
3	ri | -	1	b | null	1
1	a | 0	4		
4	a | 0	5		
5	a | 1	3	b | 0	4
4	b | 1	4	bb | 11	4
4	a | 0	5		
5	a | 1	3		
3	b | 0	4	bb | 11	4
4	a | 0	5		
5	a | 1	3		

3	a | 0	2	aa | 11	3
3	a | 0	2		
2	b | 1	3	b | 0	4
4	a | 0	5	aa | 10	2
2	a | 1	5	aa | 10	2
2	ri | -	1	b | null	1
1	a | 0	4	bb | 11	4
4	a | 0	5		
5	b | 1	1	b | null	1
1	b | null	1	b | null	1
1	ri | -	1	b | null	1

Please press any key to continue.

********** MENU **********

T-method ..... (1)

SUIO-method ..... (2)

D-method (Gonenc)..... (3)

D-method (RCPT)..... (4)

W-method (Chow)..... (5)

W-method (RCPT)..... (6)

MUIO-method ..... (7)

EXIT ..... (8)

****

Your choice : 8

Thu Jan 11 23:55:33 EST 1990

********** End of Execution **********