National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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.

Canada

TEST CASE GENERATION AND FAULT DIAGNOSIS METHODS FOR COMMUNICATION PROTOCOLS BASED ON FSM AND EFSM MODELS

Ramalingom Thavasinadar

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

September 1994

© Ramalingom Thavasinadar, 1994

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01277-8

Canada

ABSTRACT

TEST CASE GENERATION AND FAULT DIAGNOSIS METHODS FOR COMMUNICATION PROTOCOLS BASED ON FSM AND EFSM MODELS

Ramalingom Thavasinadar, Ph.D.,
Concordia University, 1994

Conformance testing is essential for assuring if an implementation of a computer communication protocol is according to its standard specification. This thesis addresses the problem of test case generation as well as the problem of fault diagnosis from a specification which is represented as a Finite State Machine (FSM) or an Extended Finite State Machine (EFSM).

A new problem, called the Basic UIS Assignment Problem (BUAP), is defined. An efficient algorithm based on the Maximum Cardinality Two Matroid Intersection Problem is proposed for selecting a minimum number of transitions and an assignment of an Unique Input Sequence (UIS) for each of these transitions such that the resulting test graph has the minimum number of connected components. Three heuristic algorithms are developed for solving the asymmetric Rural Postperson Problem (RPP). Finally, a new method which combines the solutions to the BUAP and the RPP with the MU-method [SL92] is proposed for generating a minimal length test sequence for any strongly connected FSM which has at least one UIS for each state.

After analyzing the existing FSM-based test sequence generation methods for their fault detection and diagnosis capabilities, we propose two new fault diagnosis methods: the UIDD-method and the CSDD-method for implementations with at most one output or transfer fault. The UIDD-method minimizes the length of the test sequence using the RPP and it provides better fault diagnosis capability than all the UIS-based methods analyzed. This method is illustrated on the NBS TP4

transport protocol. The CSDD-method uses the Characterizing Sequences instead of the UISs and guarantees a superior fault diagnosis capability.

A new methodology is proposed for generating test cases from an EFSM. A new type of UIS, called a Context Independent Unique Sequence (CIUS), is defined and an algorithm is developed for computing a CIUS set. Our trans-CIUS-set control flow criterion is to apply the CIUS of every state at the tail state of every transition. Our def-use-ob data flow criterion enables the def-use associations to be externally observable. A two phase breadth first search algorithm is presented for generating a set of executable test cases for the required criteria. The methodology is demonstrated on a class 2 transport protocol.

TO MY PARENTS AND TEACHERS

Acknowledgments

I express my deepest gratitude to my thesis supervisors Professor K. Thulasiraman and Dr. Anindya Das. The technical guidance and the long discussions we had throughout the course of this thesis were truly valuable.

I am thankful to Professor Jeremiah F. Hayes and Dr. Marc A. Comeau for their suggestions and moral support.

I owe thanks to the members of the PROSOFT research group of University of Montreal which enriched my knowledge in Protocol Engineering. In particular, I am grateful to Professor G. v. Bochmann and Dr. Rachida Dssouli.

I am indebted to all my friends in Montreal for making my stay here pleasant and enjoyable.

Last but not the least, I thank my mother, wife and my late father for their constant inspiration and support.

Contents

	LIST OF ABBREVIATIONS	x
	LIST OF FIGURES	xii
	LIST OF TABLES	xiv
1	**INTRODUCTION**	**1**
1.1	Conformance Testing	3
1.2	Finite State Machine Model	4
1.2.1	Fault Model for FSM	6
1.3	Review of Literature	7
1.3.1	FSM-Based Test Sequence Generation	7
1.3.2	Fault Diagnosis Methods	11
1.3.3	EFSM-Based Test Sequence Generation	12
1.4	Expressing Algorithms	19
1.5	Scope of the Thesis	19
2	**TEST CASE GENERATION FROM FSM MODEL**	**22**
2.1	Graphs and Matroids	23
2.2	UIS-Based Methods and the Need for a New Method	26
2.2.1	The U-method	27
2.2.2	MU-method	28

2.2.3 Motivation for the New method 29
2.3 Basic UIS Assignment Problem 31
2.3.1 An Illustration of BUAP 42
2.4 Algorithms for the Rural Postperson Problem 47
2.4.1 A Heuristic Algorithm Based on Augmentations 47
2.4.2 A Heuristic Algorithm Based on MST 51
2.4.3 A Heuristic Algorithm Based on Symmetry 56
2.5 Generalized UIS testing method 59
2.5.1 The GU-method 59
2.5.2 An Illustration 65
2.6 Summary 66

3 ANALYSIS OF FSM-BASED TEST SEQUENCE GENERATION METHODS 67

3.1 Test Sequence Generation Methods: Review and Analysis 68
3.1.1 Distinguishing Sequence Method 69
3.1.2 Characterizing Sequences Method 73
3.1.3 W-method 79
3.1.4 Transition Tour Method 81
3.1.5 UIS-Based Methods 82
3.1.6 Wp-method 89
3.2 Comparison of Test Sequence Generation Methods 93
3.2.1 Fault Coverage and Diagnosis 94
3.2.2 Length of Test Sequences Generated 95
3.3 Summary 95

4 FAULT DIAGNOSIS METHODS 97

4.1 UIDD-method 98
4.2 State Cover Tree and UIS Set Computation 107

4.3 An Illustration of the UIDD-method 114

4.4 CSDD-method 123

4.5 Fault Localization 129

4.6 Summary 132

5 TEST CASE GENERATION FROM THE EFSM MODEL 133

5.1 The EFSM Model 135

5.1.1 An Example 139

5.1.2 Unique Input Sequence 143

5.1.3 Control Flow Fault Model 144

5.2 Test Case Selection Criteria 145

5.2.1 Control Flow Coverage 145

5.2.2 Data Flow Coverage 147

5.3 CIUS Computation Algorithm 154

5.3.1 An Illustration 170

5.4 Data Flow Graph Manipulation 173

5.5 Automatic Test Case Generation 182

5.5.1 The Two-Phase Algorithm 182

5.5.2 Fault Coverage 197

5.6 Transport Protocol Test Case Generation 200

5.7 The Feasibility Problem 212

5.8 Summary 219

6 SUMMARY AND PROBLEMS FOR FUTURE STUDY 221

6.1 Summary 221

6.2 Problems for Future Study 225

BIBLIOGRAPHY 227

List of Abbreviations

BUAP	Basic UIS Assignment Problem
CCITT	Comité Consultatif International Télégraphique et Téléphonique
CCS	Cyclic Characterizing Sequence
CIUS	Context Independent Unique Sequence
CS	Characterizing Sequence
CSP	Constraint Satisfaction Problem
DFG	Data Flow Graph
DNF	Disjunctive Normal Form
EFSM	Extended Finite State Machine
EUIO	Extended Unique Input Output
FDT	Formal Description Technique
FSM	Finite State Machine
GC	Gomory Cut
IPP	Integer Programming Problem
IS	Identifying Sequence
ISO	International Organization for Standardization
IUT	FSM/EFSM Representation of an Implementation Under Test
LP	Linear Programming
MCMIP	Maximum Cardinality Matroid Intersection Problem
MC2MIP	Maximum Cardinality Two-Matroid Intersection Problem
MST	Minimum Weight Spanning Tree
OSI	Open Systems Interconnection
PET	Partial Enumeration Technique

RPP	Asymmetric Rural Postperson Problem
RPT	Asymmetric Rural Postperson Tour
SDL	Specification and Description Language
SPEC	FSM/EFSM Representation of a Specification
TCSD Property	Tree Characterizing Set Disjoint Property
TUISD Property	Tree UIS Set Disjoint Property
UAP	UIS Assignment Problem
UIO	Unique Input Output
UIS	Unique Input Sequence

List of Figures

1.1 ISO reference model for OSI 2

1.2 Various aspects of Protocol Engineering 2

2.1 Simplified transport protocol 30

2.2 An FSM based on the INRES protocol: *responder* 31

2.3 Simplified alternating bit protocol (*receiver*) 33

2.4 Cycles in G' 39

2.5 G' for the FSM given in Figure 2.2. 44

2.6 G_H in the first iteration of *basic_assignment*. 44

2.7 G_H in the second iteration of *basic_assignment*. 45

2.8 G_H in the third iteration of *basic_assignment*. 46

2.9 Flow graph for the FSM given in Figure 2.2. 61

3.1 An Example for the D-method 71

3.2 Protocol passed by the C-method 75

3.3 Illustration of the 1-fault resolution capability of the C-method 76

3.4 An Example for the U-method 83

3.5 Protocol passed by the Uv-method 86

3.6 An Example for the Uv-method 88

3.7 State cover tree of the SPEC given in Figure 3.1(a) 91

4.1 SPEC and a state cover tree of an abstract protocol 115

4.2 A SPEC of a subset of NBS TP4 transport protocol 119

4.3 A state cover tree of the transport protocol 121

5.1 An EFSM for the AP-module in the Class 2 transport protocol 140

5.2 A data flow graph for $t3$ with respect to the walk $t1t3t8$ 152

5.3 Different walks with the same behavior sequence a/o 154

5.4 Data flow graph for the transition $t1$ in the AP-module 204

5.5 Data flow graphs for $t1, t3, t8$ with respect to the walk $t1t3t8$ of the AP-module 205

5.6 Data flow graphs for $t3$ and $t8^1$ with respect to $t1t3t8^1t8^2$ and $t8$ with respect to $t1t3t8t12$ 206

List of Tables

2.1 Labels for transitions in Figure 2.1 30

2.2 UISs for states of the FSM shown in Figure 2.1 30

2.3 Labels of the transitions in Figure 2.2 31

2.4 Valid UIS assignment without connected test graph 32

2.5 Valid UIS assignment with connected test graph 33

2.6 Edge description of G' for the FSM given in Figure 2.2. 43

2.7 Parameters on the edges of the flow graph G_f 61

2.8 UIS assignment using min-cost flow 64

3.1 Responses of the SPEC for the distinguishing sequence 70

4.1 UISs selected by *set_uis* for SPEC of Figure 4.1(a) 114

4.2 Test subtours generated in the first phase of DD-method 116

4.3 Subtours for testing $\mathcal{U}$-transitions 116

4.4 Subtours for testing the tail states of the UISs 117

4.5 Tour for testing all non-T-transitions 118

4.6 Explanation of state symbol notations in transport protocol 118

4.7 Labels of the transitions of NBS-TP4 transport protocol 120

4.8 UISs generated for the transport protocol 121

4.9 Subtours for testing the $\mathcal{U}$-transitions 122

4.10 An optimal tour for testing non-T- non-$\mathcal{U}$-transitions 123

5.1 Core transitions in the transport protocol 141

5.2 Core transitions in the transport protocol (Contd.) 142

5.3 Non-core transitions in the transport protocol 142

5.4 CIUSs for the states in the AP-module 172

5.5 Data flow coverage in the AP-module 201

5.6 Data flow coverage in the AP-module (contd.) 202

5.7 Data flow coverage in the AP-module (Contd.) 203

5.8 Preambles and control flow test tours for the transport protocol . . . 207

5.9 Preambles and control flow test tours for the transport protocol (Contd.) 208

Chapter 1

INTRODUCTION

The Open Systems Interconnection (OSI) reference model [IS7498] of International Organization for Standardization (ISO) proposes a seven layer architecture for designing computer communication systems, as shown in Figure 1.1. Each layer performs a well defined function and it consists of many entities. Entities of a given layer communicate with the layer above (below) through service access points. Each layer functions according to a set of well defined rules known as the **protocol** of that layer. As protocols become highly complex systems, protocol design is becoming a challenging problem [Sar93]. A protocol engineering [Liu89] life cycle includes service and protocol specification, validation and verification, implementation, and conformance testing. Figure 1.2 shows the relationship between the different phases of the life cycle.

Traditionally, many protocols are specified in deterministic Finite State Machines (FSMs) [Boc78]. Standardization institutions such as ISO and Comité Consultatif International Télégraphique et Téléphonique (CCITT) have standardized specification languages, also called Formal Description Techniques (FDTs), such as LOTOS [IS8807, BB87], Estelle [IS9074, BD87] and the Specification and Description Language (SDL) [SDL88] for specifying distributed systems, in general, and protocol standards, in particular. While Estelle and SDL are based on the Extended Finite

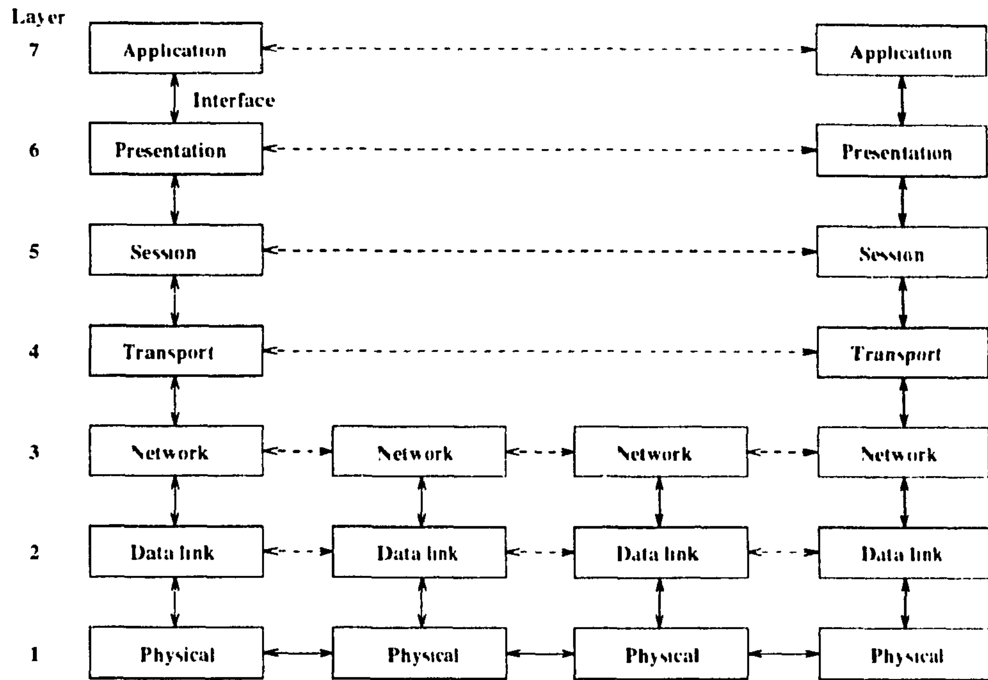


Figure 1.1: ISO reference model for OSI

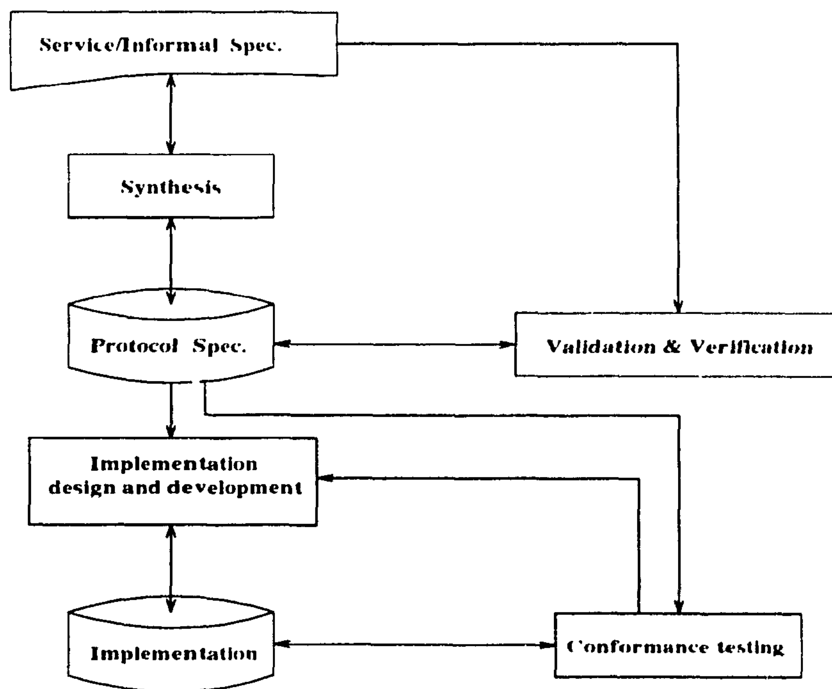


Figure 1.2: Various aspects of Protocol Engineering

State Machine (EFSM) model. LOTOS is based on the Calculus of Communicating Systems [Mil89], the Communicating Sequential Processes [Hoa85] and ACTONE [EM85].

Validation of a protocol is a process of checking the (standard) specification for logical self-consistency; it is normally done by checking if the specification satisfies the safety and liveness properties [Zaf78, ZWR+80]. Protocol verification is the process of verifying whether the specification provides the required functions of the protocol as per its service specification [Sab88]. Once the specification is validated and verified, the implementation phase begins. Semi-automatic tools are being developed for deriving an implementation of a protocol from its specification [BGS87, SF87, JJ89, SS91b, SS91a, Bud92, Tel93]. Since the implementation phase involves human interactions, the implementation is error-prone [Nai92]. Therefore, in order to achieve the interworking of heterogeneous computers, the ultimate goal of OSI, it is important to test for the correctness of the implementations of a protocol against its formal specification. Conformance testing [IS9646] is the most practical way of testing an implementation and it has been an active area of research.

1.1 Conformance Testing

Conformance testing of a protocol implementation is intended to assure that the implementation is equivalent to the standard specification of the protocol [Sta93, IS9646]. It involves the generation of test suite from the specification, execution of the test suite on the implementation under a suitable test environment, and analysis of the test results [IS9646, Ray87]. The OSI conformance testing methodology and framework [IS9646] defines a test suite as a set of test cases, where a test case is a set of protocol event sequences. Each test case has an associated purpose of testing a specific protocol function. Also, a **verdict** of pass, fail, or inconclusive is assigned to each event sequence in a test case. A verdict of a sequence depends on the purpose of

the test case for which it is a member as well as the specification. During conformance testing the implementation under test is viewed as a black box with some **Points of Control and Observation (PCOs)**. Various test architectures for executing the test suite on an implementation are also standardized in [IS9646].

ISO has recently established a working group for studying the application of Formal Methods in Conformance Testing (FMCT) [FMC'93]. One of the primary aims of the FMCT group is to enable computer-aided test suite derivation from protocol standards specified in FDTs. As a result, there has been a growing interest in developing methods for automatic test case generation from protocol standards. Since testing involves heavy costs, only a limited number of test cases of finite length are usually selected for testing. Hence, the quality of test cases generated has direct influence on the quality of the test result. Over the last fifteen years or so, research has been carried out for generating test cases of minimum cost with maximum fault coverage. Before reviewing the literature on the test case generation methods, we next introduce the widely used FSM model.

1.2 Finite State Machine Model

An FSM M can be formally defined as a 5-tuple $M = (S, s_1, I, O, T)$ where $S = \{s_1, s_2, \ldots, s_n\}$ is the nonempty finite set of states of M, s_1 is a designated state called the **initial state**, and I and O are nonempty finite sets of possible inputs and outputs of the protocol, respectively. The transition function T is a partial function defined as $T : S \times I \rightarrow S \times O$. $T(s_i, a) = (s_j, o)$ means that the FSM M at state s_i makes a transition to state s_j when the input a is applied producing the output o. If $t = ((s_i, a), (s_j, o)) \in T$, then t is called a transition in M. s_i and s_j are called the **starting state** and the **tail state** of t, respectively. We shall represent t as $(s_i, s_j; a/o)$.

An FSM $M = (S, s_1, I, O, T)$ can also be represented by a directed labeled graph

$G(V, E)$, where $S = V$ and each transition $(s_i, s_j; a/o)$ corresponds to an arc in E directed from s_i to s_j with label $tid : a/o$, where tid is a unique transition identification for the transition/arc. The identification tid in the label of the transition is in fact optional. An FSM is said to have **reset capability** if for each state s_i in S there exists a transition $(s_i, s_1; r/-)$, called a **reset transition** which resets the FSM to its initial state where 'r' denotes the 'reset' command and '$-$' denotes that the FSM does not produce any output for the reset command. Whenever there is no confusion, we represent all the reset transitions in a protocol simply by r.

We call an FSM M **completely specified** if at each state s_i in M and for each input a in I, there is an outgoing transition from s_i with input a. An FSM can be modified into a completely specified one by using what is called a **completeness assumption** [DS88]. The completeness assumption requires that a self-loop transition with input a and output '$-$' be added for a state s_i if it does not have an outgoing transition with input a.

An input (output) sequence is a sequence of input (output) symbols. We denote an ordered pair (a, b) of input and output by a/b. An **input-output sequence** is a sequence of input and output pairs. We use the operator '$\bullet$' for concatenating two inputs (outputs) symbols as well as input-output pairs. '@' is an operator for concatenating two input (output) sequences as well as input-output sequences. These operators are omitted in certain sequences whenever there is no confusion.

A **walk** is a sequence of a finite number of transitions in the graph G of M, where the tail state of every transition, except the last transition, is the starting state of its successive transition. We use '@' for concatenating walks. This operator is often omitted in the concatenated walks. Note that the same operator is used for concatenating input, output, and input-output sequences. The usage of '@' can easily be distinguished from the context. The starting state (tail state) of the first (last) transition in a walk is called the **starting state (tail state)** of the walk. A walk is said to be **closed** if its starting state and the tail state are the same. A

tour is a closed walk whose starting state is the initial state of the FSM. We often identify a state (s_i) in M by simply referring to its index (i). By **Walk(j, inseq)**, we denote a walk from s_j of the graph of M such that $inseq$ is the concatenation of the inputs along this walk. It actually denotes the sequence of transitions the FSM will go through when $inseq$ is applied to it when it is at s_j.

The function **Tail(j, inseq)** accepts the index of the state s_j, $1 \leq j \leq n$ and an input sequence $inseq$ and returns the index of the state the FSM M will reach on applying the $inseq$ to M when it stays at s_j. That is, $Tail(j, inseq)$ is the tail state of $Walk(j, inseq)$. Similarly **Dest(t)** will denote the tail state of the transition t. We shall denote the input sequence, the output sequence and the input-output sequence on a walk W by **Inseq(W)**, **Outseq(W)** and **IOseq(W)**, respectively.

A completely specified FSM M is said to be minimal if for every pair of distinct states in M there exists at least one input sequence of finite length such that M produces different output sequences when this input sequence is applied to M when it is in these two different states. Formally, a completely specified FSM M is **minimal** if for every pair of states $(s_i, s_j), i \neq j$, there exists an input sequence D_{ij} of finite length such that $Outseq(Walk(i, D_{ij})) \neq Outseq(Walk(j, D_{ij}))$

In FSM-based testing methods, the specifications and the implementations are assumed to be represented as FSMs. We shall refer to the FSM representations of a specification and an implementation of a protocol as SPEC and IUT, respectively. The methods also assume that the SPEC is strongly connected[1].

1.2.1 Fault Model for FSM

Our fault model for an FSM-based protocol is similar to the one given in [BDD+91]. The faulty IUTs are assumed to have only two types of faults, namely, the output fault and the transfer fault [BDD+91, DS88]. A transition $(s_i, s_j; a/o)$ of the SPEC is said to have an **output fault** in the IUT if for the state corresponding to s_i and for the

[1]An FSM M is said to be **strongly connected** if each state is reachable from all the states.

input a, the IUT produces an output different from o. A transition $(s_i, s_j; a/o)$ of the SPEC is said to have a **transfer fault** in the IUT if for the state corresponding to s_i and for the input a, the IUT makes a transition to a state which does not correspond to s_j. The **fault coverage** of a test sequence generation method is the percentage of faulty IUTs the method can detect from the set of all IUTs with any number of output faults and/or transfer faults. Thus, a method is said to have **complete fault coverage** if the test sequence generated by the method has 100% fault coverage.

In order to detect and diagnose the fault, the test sequence generated is applied to the IUT one by one and the output is observed. If the output is different from the expected one, the testing process is stopped and the IUT is declared faulty. The output sequence obtained thus far is analyzed for diagnosing the fault. A test sequence generation method has **k-fault resolution capability of level l**, where $k, l \geq 1$, if for any IUT with at most k faulty transitions, a test sequence generated by the method can localize at least one faulty transition to within a set of l transitions provided the presence of a fault is detected.

1.3 Review of Literature

1.3.1 FSM-Based Test Sequence Generation

The control flow in a protocol can be represented by an FSM. A major part of the research contributions in test sequence generation so far is based on the FSM model. Some of the earlier work had been done with respect to hardware and software testing [Gil62, Gon70, Koh78, Cho78]. The test suite for an FSM is simply a linear input sequence. The expected output sequence for this input sequence is also known to the tester. Though some of the existing methods, for example, the U-method [SD85], consider the test sequence as an input-output sequence, we uniformly treat the test sequence of an FSM as an input sequence.

In all FSM-based test sequence generation methods, a transition is tested by

bringing the IUT to the starting state of the transition under test and applying the input symbol of the transition and observing the output produced by the IUT to see if it matches with the output of the transition. Most of the methods also confirm the tail state of the transition in the IUT by applying a sequence which can uniquely identify the state [Koh78]. Since such an identification sequence is generated from the SPEC, some methods verify this sequence in the IUT before using it.

The transition tour method (in short, the T-method) [NT81] requires traversal of each of the transitions at least once. Thus, the required test sequence is an input sequence along a minimum length tour which traverses all the transitions. It is reported that this method has only a limited capability of detecting faults in an IUT since it does not confirm the intermediate states as it traverses the transitions [SL89].

The distinguishing sequence method [Hen64, Gon70], also known as the D-method, assumes that the SPEC is completely specified and minimal. The IUT is assumed to have at most the same number of states as the SPEC. The method uses what is called a **distinguishing sequence** for identifying the states. The method is discussed in detail, later in Chapter 3. It consists of two phases. The test sequence generated in the first phase verifies if the distinguishing sequence of the SPEC is also a distinguishing sequence of the IUT. The second phase is to generate a test sequence for testing all the transitions. A test subsequence for a transition consists of a sequence to put the IUT in the starting state of the transition followed by the input of the transition and the distinguishing sequence. The method has complete fault coverage [FBK+91]. The primary limitation of the method is that some SPECs may not have a distinguishing sequence.

The characterizing sequence method, henceforth referred to as the C-method, proposed by Kohavi *et al* [KRK74, Koh78] assumes that the SPEC is completely specified and minimal. It uses a set of sequences known as the **Characterizing Sequence set (CS set)** for identifying the states. C-method is also discussed in Chapter 3. For each state, it defines an **Identifying Sequence set (IS set)**, a

subset of the CS set. It first computes an **identifying sequence** for each state using the IS set of that state. As in the D-method, the C-method also has two phases. In the first phase, it generates a test sequence for verifying the identifying sequence of each state in the IUT. The sequence generated in the second phase is for testing the transitions. A test sequence for a transition consists of a sequence to put the IUT in the starting state of the transition under test and confirming the state followed by the input symbol of the transition and a sequence to verify the tail state of the transition in the IUT. The latter sequence is based on the identifying sequence of the tail state as well as the characterizing sequences. Unlike the D-method, the C-method can be applied on any SPEC which is minimal.

In [Cho78], Chow proposes a test sequence generation method for SPECs which have the reset capability. The method, called the W-method, assumes that the SPEC and the IUT are completely specified and minimal. It further assumes that the IUT is strongly connected and it has the reset capability. The number of states in the IUT is assumed to be within a known bound. The method first computes the CS set of the IUT from the CS set of the SPEC. The CS set of the IUT is used in the test sequence generation. The W-method uses a fixed path to reach a given state from the initial state. The reset transition is used to reach the initial state from a given state. The test sequence for a transition consists of a set of subsequences, one for each sequence in the CS set. A subsequence in the above set lies along the tour consisting of the fixed path from the initial state to the starting state of the transition under test, the transition itself, the walk starting from the tail state of the transition with the characterizing sequence as its input sequence and the reset transition. The W-method guarantees complete fault coverage [Cho78].

The Wp-method of Fujiwara *et al* [FBK+91] is an improvement of the W-method in that the length of the test sequence is reduced without compromising the fault coverage. It actually splits the W-method into two phases. Only the transitions in the fixed paths are tested in the first phase. For this the scheme as described in the

W-method is used. The remaining transitions are tested in the second phase. The same scheme is applied in this phase for testing the remaining transitions. However, for confirming the tail state of the transition under test, it uses only the IS set (in the IUT) of the state, instead of the whole CS set of the IUT. The W-method and the Wp-method are described in detail in Chapter 3.

The U-method presented in [SD85, ADLU88, DSU90a, SD88] defines what is called a **Unique Input Output (UIO) sequence** of each state for state identification. An UIO-sequence of a state is an input-output sequence of shortest length which lies along a walk starting from that state such that no walk starting from any other state has the same sequence. Since we express the test sequence as an input sequence, we extract only the input part of an UIO-sequence and call it as the **Unique Input Sequence (UIS)** as in [YLB93]. A test sequence for each transition is an input sequence along a walk which comprises of the transition followed by the walk starting from the tail state of the transition along the UIS of the state. The problem of generating a minimum length test sequence according to the above scheme is formulated in [ADLU88] as an **asymmetric Rural Postperson Problem (RPP)** [EJ73, Kua62, TS92]. The eficient solution for RPP proposed in [ADLU88] requires a certain auxiliary graph derived from the SPEC for a given set of UISs, to be weakly connected[2] [ADLU88].

In order to further minimize the length of the test sequence generated by the U-method, Shen *et al* [SLD92, SL92] have recently proposed a method, known as the MU-method. Instead of a single UIS for each state, it uses a number of UISs. The improvement is achieved by suitably assigning an UIS for testing every transition incoming at a state from the set of UISs of that state such that the final test sequence is of minimum length. They have formulated the problem as an **UIS Assignment Problem (UAP)** and presented an efficient algorithm. The algorithm presented in [SL92] is based on a minimum cost maximum flow problem [Tar83] and it requires

[2]A directed graph is said to be **weakly connected**, if there exists a path between every pair of vertices in the underlying undirected graph.

that a certain graph derived from the SPEC for the given assignment of UISs to the transitions be weakly connected [SL92].

Methods for further minimizing the length of the test sequence by overlapping test subsequences of the transitions are presented in [CCK90, MP91, LJH92]. Boyd and Ural have shown in [BU91] that the problem of finding an optimum length test sequence is NP-complete if the overlapping of the test subsequences of the transitions is allowed.

Though the UIS-based methods as described in [ADLU88, SLD92, SL92, CCK90, MP91, LJH92] have very high fault coverage [MCS93], they do not have complete fault coverage [CVI89]. The improved UIS method of Chan *et al* [CVI89] is intended to rectify this limitation by verifying the UISs in the IUT as it is done in other methods (for example, D-method, C-method, W-method) for verifying the state identification sequences used in the respective methods. This method assumes that both the SPEC and the IUT are completely specified.

The recent method by Yao *et al* [YPB93] is for generating test sequences for SPECs which may not have the reset capability. It assumes that the IUT has at most the same number of states as the SPEC, and that the IUT is completely specified and minimal. The test sequence also includes subsequences for verifying the UISs in the IUT and it guarantees complete fault coverage. The length of the test sequence generated is often very high in this method due to the state verification sequences.

1.3.2 Fault Diagnosis Methods

When an implementation fails the conformance testing, it goes back to the implementor for correction. Apart from assigning a fail verdict to the implementation, if the tester also provides some diagnostic information on the possible functions/operations which are not implemented as per the standard specification, then it certainly speeds up the correction process. All the test case generation methods available in the literature basically detect the presence of faults, if any, in the implementation. Because

of its practical importance, researchers have started designing methods for generating test cases for detecting as well as diagnosing faults in an implementation.

The work by Vuong and Ko [VK90] is for the FSM model and is based on the Constraint Satisfaction Problem (CSP) [Mac77]. Using CSP resolution, they first generate the set of all FSMs whose behavior is the same as that of the IUT for a given test sequence. These FSMs are then checked for their equivalence with the SPEC. If none of them is equivalent to the SPEC, then clearly the IUT is faulty. Therefore, the method generates some additional test sequences to identify from the above set the FSM which is identical to the IUT. The complexity of this method is quite high, as the CSP is an NP-complete problem.

The recent method by Ghedamsi [Ghe92] takes an alternate approach for diagnosing faults in an FSM. As the first step, it applies the test sequence generated by any test generation method, such as the T-method, U-method, W-method, or the Wp-method to the IUT in order to find a set of transitions which could be faulty. We shall refer to this set as the **initial fault resolution set**. In the second step, it computes a set of additional test sequences for further localizing the fault.

The size of the initial fault resolution set depends on the fault detection and diagnosis capability of the test generation method used. Some of the existing methods may produce almost all the transitions in the SPEC as the initial fault resolution set. In this case, the first step is not very useful; also it severely affects the second step since the method needs to avoid all the transitions in the initial fault resolution set while computing the **limited characterizing set** [Ghe92]. Some of the test generation methods, for instance the T-method, may produce an empty initial fault resolution set despite the fact that the IUT is faulty.

1.3.3 EFSM-Based Test Sequence Generation

As its name suggests, the Extended Finite State Machine (EFSM) model is an extension of the FSM model. Here, we provide an informal description of the EFSM

model considered in this thesis. A formal treatment is given in Chapter 5 where we present a test case generation method for this model. While a transition in an FSM has only an input and an output, the one in an EFSM can have an input interaction, a sequence of output interactions, and a predicate. A transition in an EFSM can also use and operate on a set of local variables of the EFSM. The input and output interactions also have a set of interaction parameters. A transition in an EFSM can be executed, if the input interaction specified in the transition is input to the EFSM when it is in the starting state of the transition and the values of the input interaction parameters and the local variables are such that the predicate of the transition is satisfied. When a transition is executed, the EFSM produces a sequence of output interactions as given in the transition, some variables are set to new values as specified by the transition, and the EFSM moves to the tail state of the transition. Specification languages Estelle, and SDL are based on a more powerful EFSM model than the one considered in this thesis [CA91]. For instance, this model supports communicating EFSMs, whereas our model has a single EFSM only. There are techniques available in the literature to transform a protocol specification written in Estelle or SDL into an EFSM model [SB86, Tri92, LL91, CA91, UW93], which is similar to the one considered in this thesis. Protocols expressed in LOTOS have also been transformed into a labeled transition system in order to generate test cases for the protocols [Kar88, Tri92]. Henceforth, by an EFSM, we refer to our restricted model.

As evident from the EFSM model, the input and output interactions are allowed to have parameters, and it has a set of local variables which can be manipulated by executing the transitions. Therefore, as part of conformance testing, data flow aspects of the EFSM have to be tested in addition to the control flow aspects. As the associated predicates have to be satisfied in order to execute the transitions, the feasibility of a walk cannot be taken for granted [UY91, UW93, CZ93]. Therefore, generating test cases which adequately cover the data flow and control flow aspects

of a protocol represented in an EFSM is a challenging problem. In this section we review the EFSM-based test sequence generation methods available in the literature.

The Functional Approach

The semi-automatic test case generation method of Sarikaya *et al* [SBC87, FS92] is for a normal form Estelle module [SB86], which is similar to an EFSM. This method is based on the functional approach for software testing [How87]. It selects a set of tours for each functional block. In order to test every function thoroughly a set of test data is selected for executing each tour generated for the function.

The method has limited control flow coverage since the tail states of the transitions are not verified while traversing the transitions. The functional coverage is affected since the feasibility of a tour is considered only after selecting a set of tours for the function. The method does not address the data flow aspects effectively.

The Static Data Flow Approach

Ural and Yang [UY91] uses the static data flow analysis technique [RW85, Ura87] for selecting test cases for protocols represented as a single normal form Estelle module. An **IO-df-chain** with respect to a variable is a walk from the starting state of a transition where the variable is defined by an input interaction parameter. The value of the variable affects a chain of variables along this walk. The walk terminates in a transition where this chain ends affecting an output interaction parameter or the predicate in the transition. In order to have finite length, the walk does not include more than two sub-walks for any def-use pair in this walk. A data flow coverage criterion known as the **IO-df-chain criterion** is used to select a set of tours such that every IO-df-chain in the module is covered by at least one tour in the set.

The tours are not tested for feasibility while selecting them. Due to the limitation imposed on the IO-df-chain for achieving finite length, a number of tours selected may be infeasible. This phenomenon is also observed in [Wey90] with respect to the

all-du-path criterion [RW85]. For the control flow aspects of testing, the method only guarantees the traversal of each transition in the module.

In their recent work [UW93], Ural and Williams have used the **all-uses** [RW85] data flow coverage criterion to select test cases for protocols specified in SDL. The all-uses criterion requires the selection of a set of tours such that at least one def-clear path for every def-use pair in the protocol is covered by some tour. Though the number of tours required to satisfy the all-uses criterion is only quadratic in the number of transitions in the protocol, the use of a variable in the def-use pair may not be observable in the selected tour. Although Ural and Williams do not directly address the feasibility of the selected tours, they have emphasized its importance.

UIO-Based Approaches

In [CA91], Chun and Amer have applied the U-method [ADLU88, SD88] for selecting test cases for an Estelle module in normal form. As in the U-method, their control flow coverage criterion is to traverse each transition followed by an UIO-sequence of its tail state. Henceforth, we refer to this criterion as the **trans-UIO criterion**. In general, we define the **trans-state-id criterion** as the one which requires a set of tours such that for each transition, the set has at least one tour which traverses the transition followed by a state identification sequence of the tail state of the transition [Koh78, Cho78, FBK+91]. They have also presented a scheme for generating an executable sequence of transitions. They do not, however, consider how this scheme could be used for computing feasible UIO-sequences or for selecting feasible test cases which satisfy the required control flow coverage criterion. The data flow aspect is not addressed in this approach. Due to the tail state confirmation requirement in the transition testing, the test cases selected in this method provide better control flow fault coverage than all the EFSM-based methods discussed thus far.

In [LHHT94], Li *et al* propose a method for generating control flow test cases from an EFSM. The EFSM is assumed to have only integer type of local variables

and input interaction parameters. Also, the predicates are assumed to be linear. A new type of UIO-sequence known as the Extended UIO-sequence (**EUIO-sequence**) is introduced. Let $W1$ ($W2$) be a walk ending (starting) at a state. The sequence of input and output interactions along $W1W2$ is called an EUIO-sequence of the state if $W1W2$ is always feasible and the sequence of input and output interactions along $W2$ is an UIO-sequence of the state. We refer to $W1$ as a **pre-walk** for the EUIO-sequence. Two feasible tours are selected for testing a transition such that they contain the same walk from the initial state to the starting state of the transition under test. Also, while the first tour contains the underlying walk of an EUIO-sequence of the starting state of the transition, the second tour contains the underlying walk of an EUIO-sequence of the tail state of the transition such that the pre-walk has the transition under test as its last transition. The feasibility of the walks are checked using integer linear programming problem.

In order to test all the incoming transitions at a state, more than one EUIO-sequence may be required for that state. Also, the problem of finding if a given UIO-sequence has an EUIO-sequence is, in general, undecidable [LHHT94]. The method does not consider the data flow testing. It assumes a reset transition from every state to the initial state.

Combined Testing Approaches

The test case selection method of Miller and Paul, presented in [MP92], covers both the control flow and the data flow aspects of protocols which are specified as a single normal form Estelle module. The trans-UIO criterion is used for control flow coverage. It has been established in [MP91] that, in order to guarantee the trans-UIO criterion, it is enough to find a tour such that (i) it covers all the transitions, and (ii) for each occurrence of any converging transition[3], the tour covers the transition followed

[3]A transition ending at a state is said to be converging if there exists another transition ending at the same state such that both transitions have the same input and the same output

by the path along the UIO-sequence of its tail state. A **def-ob path (def-puse path)** with respect to a variable in a data flow graph - a graph extracted from the Estelle module - is a path in the graph such that the variable is defined by an input interaction parameter in the first node of the path and the value flows through other local variables along the path until it is assigned to an output interaction parameter (a variable in the predicate) at the end of the path. Corresponding to each def-ob path (def-puse path) in the data flow graph, there exist **def-ob walks (def-puse walks)** in the Estelle module such that the required data flow occurs in these walks in the same order as in the def-ob path (def-puse path). The **def-ob/def-puse criterion** is to cover at least one feasible def ob walk for each def-ob path and at least one feasible def-puse walk for each def-puse path. The method provides a back-tracking approach for selecting a set of tours which cover the set of walks satisfying the combined data flow and the control flow criteria.

Miller and Paul's method takes the white box approach of testing, in contrast to the traditional black-box approach which is suitable for conformance testing of protocol implementations. The method does not address the feasibility issue effectively. For instance, it does not consider the issue of obtaining a feasible def-ob walk (def-puse walk) for a def-ob path (def-puse path) even if such a walk is known to be present. The back-tracking approach of combining the walks is inefficient as the partial tours obtained at a certain point of time may have to be undone if there exists no enabling context for all the uncovered walks.

In their recent work [CZ93], Chanson and Zhu have presented another unified approach for selecting test cases for both control flow and data flow aspects of protocols represented as EFSMs. They use the trans-state-id criterion for control flow coverage. A **Cyclic Characterizing Sequence (CCS)** [CZ93] is used as the state identification sequence for each state. They propose to use either the all-du-path criterion or the all-uses criterion for the data flow coverage. A set of paths satisfying the required def-use association is first determined. Each path is then augmented

into a tour by prefixing it with a shortest path from the initial state of the EFSM to the starting state of the path and suffixing the path with a shortest path from the ending state of the path to the initial state followed by a CCS of the initial state. The tours are then augmented with a CCS at each converging state[4] [MP91] provided the CCS does not disturb the data flow coverage criterion. In this way, the above [illegible] also covers the control flow criterion for some of the transitions. Some additional tours are also selected for covering the remaining transitions for their control flow. Traditional CSP techniques [Mac77, JL87] are used to check the feasibility of the selected tours. If a tour is infeasible, then the self-loops within the tour which influence the control flow are analyzed for determining the number of times they have to be included so that the tour becomes feasible, if such an augmentation is possible. The remaining infeasible tours are discarded.

The self-loop analysis requires solutions to two mathematically intractable problems: the recurrence relation problem and the feasibility problem [CZ93]. Since the feasibility of the tours are considered only after selecting them to cover the criteria, the control flow and the data flow coverage criteria are affected if some of the tours remain infeasible after the self-loop analysis.

The Fault-Based Method

In [WL93], Wang and Liu present a fault model based test case generation method for an EFSM model which is somewhat similar to the EFSM model considered in this thesis. While all the other methods generate test cases independent of any specific fault model, this method selects test cases which have the capability of detecting faults in a prespecified fault model. The method takes an EFSM representation of a protocol specification and a fault model as its input and produces a test case that detects errors in the model. The feasibility issue is not addressed effectively in this

[4] A set of states are said to be converging if there exists a set of transitions such that it has one transition from every state in this set to a common state and the transitions in this set has the same input and the same output

method.

1.4 Expressing Algorithms

In this thesis algorithms are described using plain English combined with the widely known Pascal-like syntax [JW74] such as the assignment and the **if** statements, the **for**, the **while**, and the **repeat..until** structures, the block and the record structures etc. Comments in the algorithm descriptions are surrounded with curly brackets (braces) { }, for example {Input: Specification graph G_s.}. Curly brackets are also used to denote a set of elements, for example E := $\{s_i \mid 1 \leq i \leq n\}$. Both uses are fairly standard and easy to recognize from the context.

1.5 Scope of the Thesis

This thesis is concerned with the development of new methods for generating test cases for protocols specified either as an FSM or as an EFSM.

Chapter 2 presents a new method for generating a test sequence for a given FSM which has at least one UIS for each state. This method is based on the MU-method and addresses certain shortcomings of this method. The novelty of the U-method and the MU-method is that they minimize the length of the test sequence using optimization techniques such as RPP and minimum cost maximum flow problems [Tar83]. These methods have very high fault coverage [MCS93]. The length minimization techniques of the U-method and the MU-method can be applied only when certain auxiliary graphs constructed from the SPEC are weakly connected. There are real life protocols for which this condition may not be satisfied. This shortcoming is addressed in our method. It requires solutions to two sub-problems: Basic UIS Assignment Problem (BUAP), and the general RPP. The BUAP is formally defined and is shown to be a maximum cardinality two matroid intersection problem [Law76].

An efficient algorithm is proposed based on the algorithms of Lawler [Law75] and Edmonds [Edm79] as given in [NW88]. Heuristic algorithms for the RPP are then presented. Finally, our test sequence generation algorithm carefully combines the above algorithms with the MU-method.

In Chapter 3, we formally analyze the fault detection and fault diagnosis capabilities of the existing FSM-based test sequence generation methods. These methods are now being used for detecting the presence of faults in an implementation. Recently, test sequences generated by these methods have been taken as the initial sequences for diagnosing faults in an implementation [Ghe92]. While the fault detection capabilities of only some of the methods are available in the literature, none of these methods have formally been analyzed for their fault diagnosis capabilities. Therefore, our analysis in Chapter 3 will be useful for the practitioners to choose an appropriate method for testing and diagnosing protocol implementations. The need for similar study is also indicated in [Ura92], Moreover, this study gives us an insight on the complexities involved in designing new methods with better fault diagnosis capabilities.

Two new methods are proposed in Chapter 4 for generating test sequences from specification FSMs for diagnosing a single fault in an implementation. Both methods are based on the Wp-method [FBK+91]. The first method uses a UIS for identifying a state. It applies the RPP optimization techniques at appropriate places in order to minimize the length of the test sequence. This method guarantees the best fault diagnosis capability among all the UIS based methods analyzed in Chapter 3. The second method uses a CS set for identifying the states and it provides superior fault diagnosis capability than the Wp-method.

Chapter 5 presents a new approach for generating test cases for testing both the control flow and the data flow aspects for protocols which are represented as EFSMs. A new type of state identification sequence, namely the Context Independent Unique Sequence (CIUS), is defined and an algorithm for computing a CIUS of a given

state is developed. The trans-CIUS-set criterion proposed for control flow coverage is superior to the existing control flow coverage criteria for the EFSM. In order to provide observability, the "all-uses" data flow coverage criterion is extended to what is called the def-use-ob criterion. Finally, a two-phase breadth-first search algorithm is designed for generating a set of executable test tours for covering the selected criteria.

In Chapter 6, we summarize our contributions in this thesis and point out certain problems for future study.

Chapter 2

TEST CASE GENERATION FROM FSM MODEL

As pointed out in Chapter 1, the U-method [ADLU88] and the MU-method [SLD92, SL92] generate test sequences of minimal length with very high fault coverage [MCS93] and have practical applications [SU90]. However, these can be applied only for limited classes of protocols which are represented as strongly connected FSMs having at least one Unique Input Sequence for each state. In this chapter, we propose a generalized approach that can be used for generating test sequences for any such protocol.

The necessary definitions from graph theory and matroid theory are provided in Section 2.1. The U-method, the MU-method and the need for a new method for test case generation are discussed in Section 2.2. Our method requires solutions to two sub-problems: the Basic UIS Assignment Problem (BUAP), and the general Rural Postperson Problem (RPP) [EJ73]. In Section 2.3, the BUAP is formally defined and an efficient solution is presented. Heuristic algorithms for the RPP are proposed in Section 2.4. Our generalized method presented in Section 2.5 carefully combines the above algorithms with the MU-method to derive test sequences for a given protocol.

2.1 Graphs and Matroids

The following definitions are taken from [TS92, Law76, PR88].

A **graph G** is a pair (V, E), where V and E are two finite disjoint sets. Elements of V and E are called **vertices** and **edges**, respectively. Each edge is identified with an unordered pair of vertices referred to as the **end vertices** of the edge. An edge is said to be **incident** on its end vertices. Two edges are said to be **adjacent** if they have a common end vertex. An edge is called a **self-loop** if both its end vertices are identical. A **walk** in a graph is a finite sequence of vertices $(v_0, v_1, \ldots, v_k)$, $k \geq 1$ such that (v_{i-1}, v_i) for $i = 1, 2, \ldots, k$, are edges in G. v_0 and v_k are called the **end vertices** of the walk and all other vertices are called its internal vertices. The walk $(v_0, v_1, \ldots, v_k)$ can also be denoted as a sequence of edges $(t_1, t_2, \ldots, t_k)$ if $t_i = (v_{i-1}, v_i)$, for $i = 1, 2, \ldots, k$. A walk is **open** if its end vertices are distinct; otherwise it is **closed**. A closed walk is referred to as a **tour**. A walk is called a **trail** if all its edges are distinct. An open trail is a **path** if all its vertices are distinct. A closed trail is called a **cycle** if all its vertices except the end vertices are distinct. A graph is said to be **connected** if there exists a path between every pair of vertices in the graph.

Weighted graphs are those graphs in which a real number, usually called the **cost**, is associated with each edge. The graph $G' = (V', E')$ is said to be a **subgraph** of the graph $G = (V, E)$ if $V' \subseteq V$,$E' \subseteq E$ and both end vertices of every edge in E' are in V'. A subgraph $G' = (V', E')$ of $G = (V, E)$ is called a **spanning subgraph** if $V' = V$. A subgraph $G' = (V', E')$ of $G = (V, E)$ is said to be **induced by** $F \subseteq E$ if $E' = F$ and V' is the set of end vertices of all the edges in F. This induced subgraph is denoted by $G[F]$. If F is a **bag**[1]. of edges from E, then in $G[F]$ each edge in F is repeated as many times as it occurs in F. If K is a set of edges having both their end vertices in V, then $G + K$ denotes the graph obtained from $G = (V, E)$ by adding all

[1] A **bag** is a collection of elements over some domain. Unlike sets, bags can have multiple occurrences of the same element

the edges in K to (the edge set of) G.

If each edge of a graph $G = (V, E)$ is identified with an ordered pair of vertices then G is called a **directed graph** or simply a **digraph**. The edges in a digraph are also referred to as **arcs**. Let $a = (u, v) \in E$, where u and v are the vertices in V. a is called an **outgoing (incoming)** arc at $u(v)$. Also, u and v are called the **starting vertex** and the **ending vertex** of a, respectively. A **walk** in a digraph G is a finite sequence of vertices $(v_0, v_1, \ldots, v_k)$, $k \geq 1$ such that (v_{i-1}, v_i) for $i = 1, 2, \ldots, k$, are edges in G. v_0 and v_k are called the **starting vertex** and **ending vertex** of the walk, respectively and all other vertices are called its internal vertices. A tour, a path, and a cycle in a digraph can be defined analogous to those in the undirected graph. A digraph is said to be **strongly connected** if there exists a path from any given vertex to any other vertex. A digraph is **weakly connected** if the underlying undirected graph is connected. The digraph $G = (V, E)$ is said to be **symmetric** if the number of incoming arcs at every vertex in V is the same as the number of outgoing arcs at that vertex. Given a strongly connected digraph $G = (V, E)$ with weighted arcs, and a subset of arcs $F \subseteq E$, the **Asymmetric Rural Postperson Problem** (RPP) with respect to F is to find a tour with minimum cost such that it covers each arc in F at least once [EJ73, Kua62, TS92]. Such a minimum cost tour is referred to as a **Asymmetric Rural Postperson Tour** (RPT) with respect to F. The RPP is known to be an NP-complete problem [LRK76, Pap76].

A **matroid** $M = (E, \mathcal{I})$ is a structure in which E is a finite set of elements and $\mathcal{I}$ is a family of subsets of E such that

1. The empty set is a member of $\mathcal{I}$;

2. $F_1 \subset F$ and $F \in \mathcal{I}$ imply $F_1 \in \mathcal{I}$;

3. If F_p and F_{p+1} are sets in $\mathcal{I}$ having p and $p+1$ elements respectively, then there exists an element $e \in F_{p+1} - F_p$ such that $F_p \cup \{e\} \in \mathcal{I}$.

Elements in $\mathcal{I}$ are called the **independent sets** in M.

A **graphic matroid** of a graph $G = (V, E)$ is a matroid $(E, \mathcal{I})$ such that $F \subseteq E$ is in $\mathcal{I}$ iff $G[F]$ has no cycle.

Let $P = \{E_1, E_2, \ldots, E_k\}$ be a partition of the edge set E of a graph $G = (V, E)$. Let $Q = \{i_1, i_2, \ldots, i_k\}$ be a given set of non-negative integers. Let $(E, \mathcal{I})$ be a system such that $F \in \mathcal{I}$ iff $|E_j \cap F| \leq i_j$ for $j = 1, 2, \ldots, k$. It is known that $(E, \mathcal{I})$ is a matroid [Law76]. It is called the **partition matroid** of G with respect to the partition P and the index Q.

Let $(E, \mathcal{I}_j)$, $j = 1, 2, \ldots, k$ for some $k \geq 2$, be a system of matroids over E. The **Maximum Cardinality Matroid Intersection Problem (MCMIP)** is to find a maximum subset H of E such that H is independent in $(E, \mathcal{I}_j)$ for $j = 1, 2, \ldots k$. If $k = 2$, then the above problem is referred to as the **Maximum Cardinality Two Matroid Intersection Problem (MC2MIP)**. Though the general MCMIP is NP-complete [PS82, PR88], its special case MC2MIP is polynomially solvable [Law76].

An input sequence, U_i is called a **Unique Input Sequence (UIS)** of the state s_i of an FSM M if U_i is a sequence of shortest length such that (i) there exists a walk W from s_i such that Inseq(i,W)=U_i and (ii) for each state $s_j, j \neq i$, either there is no walk from s_j with input sequence U_i or $Outseq(j, U_i) \neq Outseq(i, U_i)$. Moreover, the input-output sequence IOseq(W) is called an **Unique Input Output (UIO) sequence** of s_i.

Let U_i be an UIS of s_i, $1 \leq i \leq n$. The set $\mathcal{U} = \{U_i \mid 1 \leq i \leq n\}$ is referred to as an **UIS set** of the FSM M.

We assign a unit cost to each edge in the specification graph $G_s = (S, E)$ since we are concerned with test sequence length minimization. Let MU_i be a nonempty set of UISs for each state $s_i \in S$. We assume that the functions $head$ and $tail$ will return the starting state and the ending state of any UIS, respectively. Also, the functions $start(e)$, $label(e)$ and $end(e)$ will return the starting state, label and the ending state of any transition e, respectively. Let $MU = MU_1 \cup MU_2 \cup \ldots \cup MU_n$. Define the relation $R \subseteq E \times MU$ such that $(e, u) \in R$ iff $end(e) = head(u)$. Clearly, R denotes

the set of all possible assignments of UIS from MUC for all the transitions in E. We call any subset $B \subseteq R$ a **valid UIS assignment** or simply an **UIS assignment** for the set of transitions $D \subseteq E$ if $dom(B) = D$ and $|\{u \mid (e, u) \in B\}| = 1$, for each $e \in D$. That is, each element in D has exactly one UIS assigned in B. A valid UIS assignment for E is also referred to as a **(valid) UIS assignment of the protocol** G_s.

Consider the undirected graph $G'' = (S, E')$ where $E' = \{(start(e), tail(u); label(e)@u) \mid (e, u) \in R\}$. An edge $e' \in E'$ which corresponds to $(e, u) \in R$ is often referred to as a **test edge** for the transition e since e can be tested by applying the sequence along e'. For each edge $e' = (start(e), tail(u); label(e)@u) \in E'$ which corresponds to $(e, u) \in R$, the length of the input sequence in $label(e)@u$ is taken as the cost of e'. It is easy to see that there is a one-to-one correspondence between R and E'. An element of R is often treated as an edge in E' and vice versa. Let B be a valid UIS assignment for $D \subseteq E$. The subgraph $G''[B]$ of G'' induced by B is called a **test graph** for D. The test graph induced by an UIS assignment of the protocol G_s is simply referred to as a **test graph for the protocol**. Observe that every test graph for a strongly connected protocol always spans all the states of the protocol. Subgraphs of G'' are often extended by adding edges from G_s, and vice versa. Suppose H' is a subgraph of G'' and $F \subseteq E$, then $H' + F$ will be treated as an undirected graph as H' is undirected. On the other hand, if H is a subgraph of G_s and $F' \subseteq E'$, then $H + F'$ will be treated as a directed graph as H is directed, the orientation of the edges in F' coinciding with that of the corresponding edge in G_s.

2.2 UIS-Based Methods and the Need for a New Method

Two important methods based on the UISs are described in this section. We study the need for a new method by analyzing the scope of these methods. Recall that SPEC

and IUT denote the FSM representation of a specification and an implementation of a protocol, respectively.

2.2.1 The U-method

The U-method [ADLU88] [DSU90b] requires that the representation graph $G_s = (S, E)$ of SPEC be strongly connected. Each state of G_s is assumed to have an UIS. Let U_j be an UIS for s_j, $1 \leq j \leq n$. The U-method tests each transition $(s_i, s_j; a/o)$, as follows:

> The protocol implementation IUT is first put in state s_i. Then the input a is applied and the output is verified for o. Finally, to check for state s_j, the UIS U_j is applied to the current state of the IUT and the output is examined against the expected output according to the SPEC.

Thus the input sequence $a@U_j$ is the test subsequence for the transition $(s_i, s_j; a/o)$. By considering $MU_j = \{U_j\}$, $1 \leq j \leq n$, we get $G' = (S, E')$, where $E' = \{(s_i, tail(U_j); a@U_j) | (s_i, s_j; a/o) \in E\}$. Clearly, G' is the unique test graph of G_s. Let $G^* = G_s + E'$. In the U-method, each transition in G_s is tested by applying the subsequence along its test edge in E'. Thus an optimai test sequence for G_s lies along an RPT of G^* with respect to E'. In other words, the optimal test sequence generation problem is equivalent to the problem of finding an RPT of G^* with respect to E'. Before proceeding further, we introduce a definition. A **rural symmetric augmentation** of a weighted digraph $G = (V, E)$ with respect to $F \subseteq E$ is a digraph $G[F \cup E_1]$ such that (i) $G[F \cup E_1]$ is symmetric, and (ii) E_1 is a minimum cost bag in E satisfying (i). The polynomial algorithm given in [SD85] for finding an RPT first computes a rural symmetric augmentation $G^*[E' \cup E_1]$ of G^* with respect to E', where E_1 is a bag containing elements in $E \cup E'$. It then generates a test sequence by concatenating the subsequences and/or inputs along an euler tour of $G^*[E' \cup E_1]$. This algorithm can be successfully applied to a protocol G_s if the test graph G' is

connected [ADLU88]. Note that this is only a sufficient condition. It is also shown that protocols which have either a self-loop at each state or the reset capability always meet this requirement.

2.2.2 MU-method

In the MU-method [SLD92, SL92], Shen *et al* have proposed an improvement for the U-method. While the U-method uses only one UIS for each state, this method uses multiple (≥ 1) UIS(s) for each state. The improvement is obtained by suitably assigning an UIS for each transition from the set of multiple UISs of its tail state in order to reduce the length of the test sequence. The approach of using multiple UISs for minimizing the length of the test sequence as introduced in [SLD89, SLD92] has two problems:

(1) The test graph resulting from the UIS assignment computed in the method may not be connected.

(2) The UIS assignment computed in the method does not necessarily minimize the length of the resulting test sequence.

These two problems were rectified in their recent method as described in [SL92] and we shall consider this modified method as the MU-method. We would like to note that an alternate approach of solving the above two problems is proposed in [Ura92]. Given a set MU_i of multiple UISs of the minimum length for each state $s_i, i = 1, 2, \ldots n$ of the protocol $G_s = (S, E)$, the **UIS Assignment Problem (UAP)** is to find a valid UIS assignment B of the protocol such that the RPT of $G_s + B$ with respect to B is of minimum length among all valid UIS assignments of the protocol. The MU-method solves certain specific instances of this problem efficiently by transforming it into an equivalent multi-stage minimum cost maximum flow problem [SL92]. As in the U-method, a minimum length test sequence is obtained by concatenating the test subsequences and/or the input of the transitions along the minimum cost RPT. The

MU-method guarantees an optimal test sequence for a protocol G_s if the test graph $G''[B]$ is connected. It has also been proved that protocols which have either the reset capability or a self-loop at each state always meet this requirement [ADLU88, SLD92]. As we shall see later in this section, this approach of obtaining minimum length test sequences does not work for all protocols.

Methods for further minimizing the length of a test sequence by overlapping test subsequences of the transitions are presented in [CCK90, MP91, LJH92]. In this thesis, we do not consider optimization through overlapping. We shall now illustrate the need for extending the U- and the MU-methods.

2.2.3 Motivation for the New method

It has been reported in [ADLU88] and [CS92] that the U-method can be applied to generate test sequence for any protocol G_s which satisfies one of the conditions (i) through (v) below. Note that conditions (i) through (iv) are independent of UISs whereas condition (v) is with respect to a particular UIS for each state.

(i) G_s has the reset capability [ADLU88].

(ii) G_s has a self-loop at each state [ADLU88].

(iii) G_s has a state, say s_e, with a self-loop and a reset edge, and each state has a self-loop, or a reset edge, or an edge to the state s_e [CS92].

(iv) For every partition of S into two nonempty subsets S_A and $S - S_A, \exists s_i \in S_A$ and $s_j \in S - S_A$ such that there is an edge to some state s_k from both s_i and s_j [CS92].

(v) For every partition of S into two nonempty subsets S_A and $S - S_A, \exists s_i \in S_A$ and $s_j \in S - S_A$ such that state $s_i(s_j)$ has an edge to a state $s_p(s_q)$ in S and $tail(U_p) = tail(U_q)$. Here, U_j is an UIS for the state s_j, $j = 1, 2, \ldots, n$ and it is used for testing every incoming transition at the state s_j [CS92].

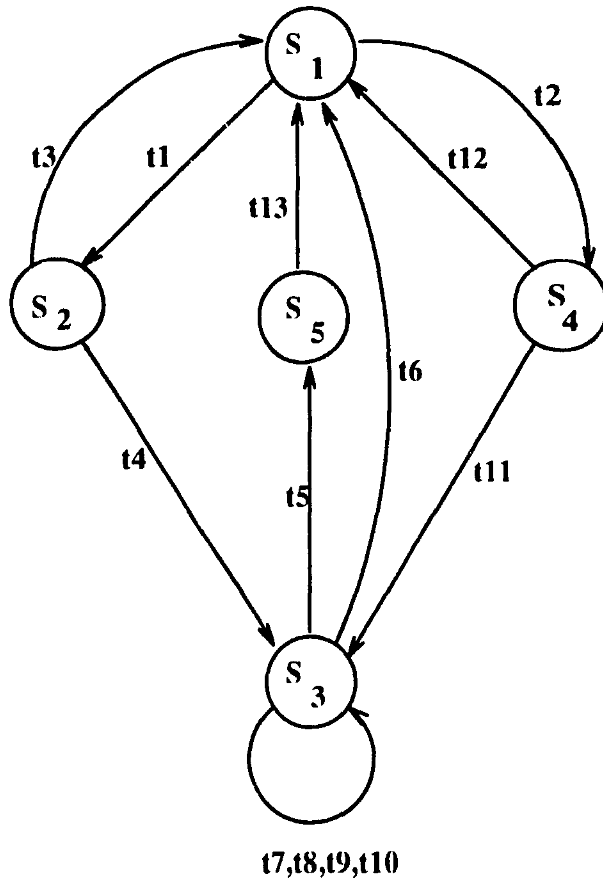


Figure 2.1: Simplified transport protocol

Transition	Label	Transition	Label
t1	TCONreq/s-CR	t2	r-CR/TCONind
t3	r-DR/TDISind	t4	r-CC/TCONconf
t5	TDISreq/s-DR	t6	r-DR/s-DC&TDISind
t7	null/s-AK	t8	r-AK/null
t9	r-DT/TDATAind	t10	TDATAreq/s-DT
t11	TCONresp/s-CC	t12	TDISreq/s-DR
t13	r-DC/TDISconf		

Table 2.1: Labels for transitions in Figure 2.1

State	UIS	State	UIS
S_1	t2 (r-CR)	S_2	t4 (r-CC)
S_3	t6 (r-DR)	S_4	t11 (TCONresp)
S_5	t13 (r-DC)		

Table 2.2: UISs for states of the FSM shown in Figure 2.1

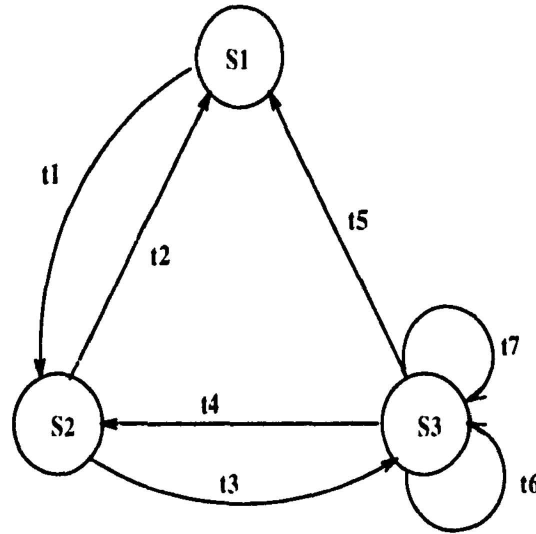


Figure 2.2: An FSM based on the INRES protocol: *responder*

Transition	Label	Transition	Label
t1	CR/ICONind1	t2	IDISreq/DR1
t3	ICONresp/CC	t4	CR/ICONind2
t5	IDISreq/DR2	t6	DT2/AK
t7	DT1/IDATind&AK		

Table 2.3: Labels of the transitions in Figure 2.2

We would like to note that the above conditions are only sufficient conditions. There are real life protocols which do not satisfy any of these conditions, yet the U-method can successfully generate test sequences for these protocols provided suitable UISs are chosen. For example, consider the FSM representation of a simplified transport protocol as given in [BDZ89] and shown in Figure 2.1. The labels for the transitions are shown in Table 2.1. This protocol does not satisfy conditions (i) - (iv). With the UISs generated as in Table 2.2, condition (v) is not met. However, the test graph of the protocol is connected if the UISs given in Table 2.2 are used. This example suggests that even if a protocol does not have any of the structures stated in the literature, a suitable assignment of UISs for some of the transitions for testing their tail states would facilitate the U-method to obtain an optimum test sequence.

Transition	UIS	Transition	UIS
t1	t2	t2, t5	t1
t3, t6, t7	t6	t4	t3

Table 2.4: Valid UIS assignment without connected test graph

Careful assignment of UISs to transitions is necessary since an arbitrary assignment may not produce a connected test graph despite the existence of such assignments. For example, consider the abstract FSM protocol as given in Figure 2.2, based on the *responder* module of the INRES protocol [Hog92]. Only the core transitions are considered here. The states s_1, s_2, and s_3 correspond to the states *DISCONNECTED*, *WAIT*, and *CONNECTED* of the *responder* module, respectively. We have slightly modified the original labels of the transitions so that the FSM has multiple UISs. The labels of the transitions are given in Table 2.3. Let $MU_1 = \{t1\}$, $MU_2 = \{t2, t3\}$, and $MU_3 = \{t4, t5, t6\}$ be the set of UISs for the states s_1, s_2, and s_3, respectively. Note that the UISs are denoted by their corresponding transitions. Let $MU = MU_1 \cup MU_2 \cup MU_3$. Clearly, the assignments A_1 and A_2 given in Table 2.4 and Table 2.5, respectively, are valid UIS assignments of the protocol. Also both these assignments are obtainable in the MU-method while it attempts to solve the UAP. Note that the test graph $G''[A_1]$ is not connected whereas the other test graph $G''[A_2]$ is connected. If the UIS-based methods assign UISs to the transitions as per A_1 then they cannot generate a test sequence for this protocol. On the other hand, A_2 facilitates the UIS-based methods to generate an optimal test sequence for the protocol. The above discussion implies that the UIS-based methods as described in [ADLU88, SLD92, SL92] may not always produce a test sequence even if the protocol has a connected test graph. Unfortunately, there is no way to ensure that the min-cost flow approach will lead to a test graph which is connected.

It should also be emphasized that certain protocols may not even have any connected test graph. Consider the FSM representation of a simplified alternating bit protocol (*receiver*) shown in Figure 2.3. $m0$ and $m1$ are the only UISs for the

Transition	UIS	Transition	UIS
t1	t3	t2, t5	t1
t3, t6, t7	t6	t4	t2

Table 2.5: Valid UIS assignment with connected test graph

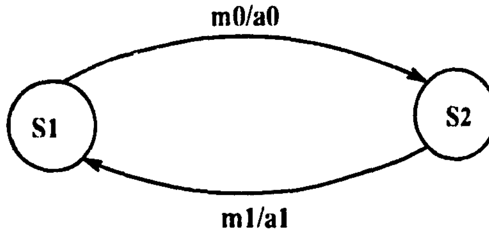


Figure 2.3: Simplified alternating bit protocol (*receiver*)

states s_1 and s_2, respectively. The FSM neither satisfies the requirement stated in conditions (i) through (v) nor has a valid UIS assignment so that the resulting test graph is connected.

Thus the following questions arise: Given a set of multiple UISs for each state, does the protocol have a set $BE \subseteq E$ of transitions and a valid UIS assignment for BE such that the resulting test graph for BE is connected and spans all the states of the protocol? If so, how to find a minimum set of transitions satisfying the above condition? (This problem is formalized in Section 2.3 as the Basic UIS Assignment Problem (BUAP)) If not so, how to minimize the length of the test sequence for this protocol? These questions are addressed in Sections 2.4 and 2.5. In section 2.4 we provide three heuristic algorithms for solving the general RPP. In Section 2.5, we propose a new method which combines the solutions to BUAP and general RPP. This method can be used to generate a test sequence from any protocol which is represented as strongly connected FSM having at least one UIS for each state.

2.3 Basic UIS Assignment Problem

As defined earlier, let MU_i be a nonempty set of UISs for each state s_i of the strongly connected specification digraph $G_s = (S, E)$; $MU = MU_1 \cup MU_2 \cup \ldots \cup MU_n$. $R \subseteq E \times MU$ is a relation such that $(e, u) \in R$ iff $end(e) = head(u)$. Consider the undirected graph $G' = (S, E')$ where $E' = \{(start(e), tail(u); label(e)@u) | (e, u) \in R\}$. Observe that for each valid UIS assignment B, the induced graph $G'[B]$ is a test graph for $dom(B)$.

> **The Basic UIS Assignment Problem (BUAP) is to find a minimum set $K \subseteq E$ and a valid UIS assignment B of K such that $G'[B]$ has the minimum number of connected components spanning G'.**

The BUAP can be efficiently solved using the matroid theoretic approach. We demonstrate this by mapping the BUAP into an equivalent MC2MIP which is solvable in polynomial steps. To start with, let us assume that G' is connected and that it has no self-loop. Let $M_1 = (E', \mathcal{I}_1)$ be the graphic matroid of G'. Let Q_e be the set of all possible UIS assignments from MU for the transition e. Clearly, $Q_e \subseteq R$ and $dom(Q_e) = \{e\}$. Let $P = \{Q_e \mid e \in E\}$. Then clearly, P is a partition of E'. Let $M_2 = (E', \mathcal{I}_2)$ be the partition matroid over the partition P and integers $i_e = 1$ for all $e \in E$. Suppose that I_{max} is a maximum set such that it is independent in M_1 as well as in M_2, then it is a valid assignment for $dom(I_{max})$ and $G'[I_{max}]$ is acyclic. Since I_{max} is a maximum set, it spans G'. These properties, in turn, imply that $G'[I_{max}]$ contains the minimum number of components (see Proof of Theorem 2.2). Hence, $dom(I_{max})$ and I_{max} form a solution to the BUAP.

We now present an efficient algorithm called *basic_assignment* for solving the BUAP. We assume that G' is connected and it has no self-loop. This algorithm is based on the algorithms for the MC2MIP by Lawler [Law75, Law76], and Edmonds

[Edm79] as given in [NW88]. These algorithms are for computing a maximum cardinality intersection of any two matroids over the same set of elements. Algorithm *basic_assignment* is obtained from the above algorithms by adapting them for computing the maximum cardinality intersection of the graphic matroid M_1 and the partition matroid M_2 given above; thereby reducing the overall time complexity. The *basic_assignment* algorithm starts with an empty set of edges. That is, initially $H = \emptyset$. At each iteration of the **repeat..until** loop of the algorithm, it computes a valid UIS assignment H such that $G''[H]$ is acyclic and H has one element more than the number of elements it had in the previous iteration. The algorithm terminates when there is no such H in the current iteration. The UIS assignment H output by the algorithm and $dom(H)$ form a solution to the BUAP. A formal description of the algorithm is given below. For the sake of simplicity in notation, we shall let an element $j = (e, u) \in E'$ also refer to the edge e.

Algorithm *basic_assignment*(G_s, MU, G', H);
{ **Input:** The digraph $G_s = (S, E)$, graph $G' = (S, E')$, set of UISs MU }
{ **Output:** a set of edges H from E' }
    $H \leftarrow \emptyset$;
    $V_H \leftarrow \{s, t\} \cup E'$;
    repeat
        {Construct the digraph $G_H = (V_H, E_H)$}
        $E_H \leftarrow \emptyset$;
        for each $j = (e, u) \in E' - H$ **do**
        begin
            if ($G'[H \cup \{j\}]$ is acyclic) **then**
                $E_H \leftarrow E_H \cup \{(s, j)\}$;
            if ($e \notin dom(H)$) **then**
                $E_H \leftarrow E_H \cup \{(j, t)\}$;
            for each $k = (e', u') \in H$ **do**
            begin
                if ($e = e'$) **then**
                    $E_H \leftarrow E_H \cup \{(j, k)\}$;

```
            if (G'[H ∪ {j}] has a unique cycle containing k) then
                E_H ← E_H ∪ {(k, j)} ;
        end
    end
    if (the digraph G_H = (V_H, E_H) has a path from s to t) then
    begin
        find a shortest path (s, j_1, k_1, ..., j_{p-1}, k_{p-1}, j_p, t) from s to t in G_H;
        H ← (H ∪ {j_1, j_2, ..., j_p}) − {k_1, k_2, ..., k_{p-1}}
    end
    else
    begin
        output(H);
        stop
    end
  for ever
end basic_assignment.
```

An iteration of the **repeat...until** loop first constructs a digraph $G_H = (V_H, E_H)$ for a given H. Here, $V_H = \{s, t\} \cup E'$, where s and t are two designated vertices in V_H. The set of vertices in V_H which represents the edges in E' is partitioned into two sets: H and $E' - H$. Then, the graph G_H is constructed in such a way that the presence of a path from s to t in G_H guarantees that the cardinality of H in the current iteration can be increased by one. In order to construct the edge set E_H, the following is done with respect to each $j = (c, u) \in E' - H$.

> If $G'[H \cup \{j\}]$ is acyclic, then an edge from s to j is added to E_H. If $c \notin dom(H)$, then an edge from j to t is added to E_H. For each $k = (c', u') \in H$, an edge from j to k is added to E_H if k and j are test edges for the same transition (that is, if $c = c'$). Also, if j and k are contained in a cycle of $G'[H \cup \{j\}]$, then an edge is added to E_H from k to j.

If the digraph G_H thus constructed has a path from s to t, then let $(s, j_1, k_1, \ldots, j_{p-1}, k_{p-1}, j_p, t)$ be a shortest path from s to t. As established in Theorem 2.1, $H' = (H \cup \{j_1, j_2, \ldots, j_p\}) - \{k_1, k_2, \ldots, k_{p-1}\}$ is a valid assignment such that $G''[H']$ is acyclic. Therefore, the algorithm proceeds to the next iteration of the **repeat...until** loop. On the other hand, if G_H has no path from s to t, then the algorithm terminates since H computed in the previous iteration and $dom(H)$ form a solution to the BUAP (refer to Theorem 2.2).

Suppose that the cardinality of H computed in the current iteration is $n-1$, where n is the number of states in G_s. Then, $G''[H \cup \{j\}]$ will have a cycle for each $j \in E' - H$. Therefore, G_H computed in the next iteration will not have any outgoing edge from s. In other words, G_H has no path from s to t. Also, the algorithm starts with H as an empty set and each iteration adds one edge to H. As a result, the algorithm terminates within n iterations.

Let m and ν denote the number of transitions and the maximum number of UISs in MU for any state of the protocol, respectively. Since G_s is strongly connected, $m \geq n$, where n is the number of states in G_s. Suppose that the computation needed to check if a given set is independent in a given matroid is considered as one step. Then, our algorithm requires $O(n(m\nu)^2)$ steps. This complexity can easily be derived since the outer and the inner **for** loop of the **repeat..until** loop are executed at most $m \times \nu$ times. Whereas the **repeat..until** loop itself is executed at most n times. Note that this complexity is better than the complexity $(O(m\nu)^3))$ of the general maximum cardinality two-matroid intersection algorithms. When the time required to complete each step is also taken into account, the complexity of *basic_assignment* become $O(n^2m^2\nu^2)$ time units. The following theorems establish the correctness of the algorithm.

Theorem 2.1 *Suppose that $H \subseteq E'$ is a valid UIS assignment and $G''[H]$ is acyclic at the beginning of a given iteration of the* **repeat..until** *loop of basic_assignment algorithm. If $P = (s, j_1, k_1, \ldots, j_{p-1}, k_{p-1}, j_p, t)$ is a shortest path from s to t in G_H*

then $H' = (H \cup \{j_1, j_2, \ldots, j_p\}) - \{k_1, k_2, \ldots, k_{p-1}\}$ *is a valid UIS assignment for* $dom(H')$ *and* $G''[H']$ *is acyclic.*

Proof:

We will first establish that $H' = (H \cup \{j_1, j_2, \ldots, j_p\}) - \{k_1, k_2, \ldots, k_{p-1}\}$ is a valid UIS assignment.

No pair of edges j_i and j_l, where $1 \leq i < l < p$, can be the UIS assignments for the same transition in E. Assume the contrary. Then, j_i and k_l will be UIS assignments for the same transition due to the fact that j_l and k_l assign UIS for the same transition. This means that $(j_i, k_l) \in E_H$ and that $(s, j_1, k_1, \ldots j_i, k_l, j_l + 1, k_l + 1, \ldots, j_{p-1}, k_{p-1}, j_p, t)$ is a path in G_H shorter than P. A contradiction.

Similarly, j_i and j_p, where $1 \leq i < p$, cannot be UIS assignments for the same transition. If this is not true, then let $e \in E$ be the transition for which j_i and j_p are UIS assignments. We know that, $e \notin dom(H)$ because $(j_p, t) \in E_H$. So, $(j_i, t) \in E_H$. This means G_H has a path shorter than P. A contradiction.

Also, no j_i, $1 \leq i < p$ and $h_m \in H$, ($h_m \neq k_l$, for any l) can both assign UIS for the same transition. If this is not so, then h_m and k_i will assign UIS for the same transition. It contradicts that H is a valid assignment.

From the above it follows that $(H \cup \{j_1, j_2, \ldots, j_p\}) - \{k_1, k_2, \ldots, k_{p-1}\}$ is a valid UIS assignment.

Next, we show that $G''[H']$ is acyclic. The fact that $(s, j_1) \in E_H$ implies that $G[H_1]$ is acyclic, where $H_1 = H \cup \{j_1\}$. Let $H_i = H \cup \{j_1, j_2, \ldots, j_i\} - \{k_1, k_2, \ldots, k_{i-1}\}$, for $2 \leq i \leq p$. We shall prove by induction that $G''[H_i]$ is acyclic, for $i = 2, 3, \ldots, p$. Note that $G''[H_p]$ is nothing but $G''[H']$. Since $(k_1, j_2) \in E_H$, k_1 is an edge in the unique cycle C_1 of $G''[H \cup \{j_2\}]$. As shown in Figure 2.4, let us denote this cycle as $R\ j_2\ S\ k_1$, where R and S are some paths in G'' whose edges are from H only.

We shall prove by contradiction that $G''[H_2]$ is acyclic. Suppose that $G''[H_2]$ has a cycle, then as $G''[H \cup \{j_2\} - \{k_1\}]$ is acyclic, j_1 and j_2 are in a cycle, say C_2, of $G''[H_2]$. Let $C_2 = X\ j_1\ Y\ j_2$ where X and Y are paths in G'' whose edges are from

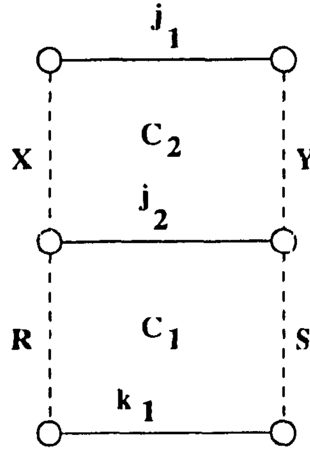


Figure 2.4: Cycles in G'

$H - \{k_1\}$. This cycle is also shown in Figure 2.4. Then $C_1 \oplus C_2 \subseteq H_1$ has a cycle[2] [TS92], contradicting that $G''[H_1]$ is acyclic.

By assuming that $G''[H_i]$, where [illegible] $i < p$ is acyclic, we prove that $G''[H_{i+1}]$ is acyclic. Let $H_0 = H - \{k_1, k_2, \ldots, k_{i-1}\}$. Clearly, $H_i = H_0 \cup \{j_1, j_2, \ldots j_i\}$ and $H_{i+1} = H_0 \cup \{j_1, j_2, \ldots j_{i+1}\} - \{k_i\}$.

As $(k_i, j_{i+1}) \in E_H$, we know that k_i is contained in the unique cycle of $G''[H \cup \{j_{i+1}\}]$. Also, no edge from $\{k_1, k_2, \ldots, k_{i-1}\}$ is contained in this unique cycle of $G''[H \cup \{j_{i+1}\}]$. For, suppose that k_r, for some r, $1 \le r \le i-1$, is an edge in the cycle, then $(k_r, j_{i+1}) \in E_H$ and $(s, j_1, k_1, \ldots, k_r, j_{i+1}, k_{i+1}, \ldots, j_p, t)$ is a path in G_H shorter than P. This is a contradiction since P is a shortest path in G_H. By the induction hypothesis, we know that $G''[H_0 \cup \{j_1, j_2, \ldots j_i\}]$ is acyclic. Therefore, $G''[H_0 \cup \{j_1, j_2, \ldots j_i, j_{i+1}\} - \{k_i\}]$ is acyclic. This completes the induction and the proof of the theorem.

□

At the beginning of the first iteration of the **repeat..until** loop of the algorithm *basic_assignment* H is a valid assignment and $G''[H]$ is acyclic since $H = \emptyset$. At the start of a subsequent iteration, H corresponds to H' of the previous iteration

[2] $C_1 \oplus C_2 = (C_1 \cup C_2) - (C_1 \cap C_2)$

Therefore, as per Theorem 2.1, we know that H is a valid UIS assignment and $G''[H]$ is acyclic.

Theorem 2.2 *At the end of a given iteration of the* **repeat..until** *loop of the algorithm* basic_assignment, *if G_H has no path from s to t then H and $dom(H)$ form a solution to the BUAP.*

Proof:

In order to prove the theorem, we define the functions t and a from the power set of E' to the set of natural numbers including zero and the functions T and A from the powerset of E' to itself. Let X be a subset of E'.

$t(X)$ = number of edges in the largest acyclic subgraph of $G''[X]$.
$a(X)$ = number of elements in a maximum valid UIS assignment in X.
$T(X) = F$ if F is a largest superset of X in E' such that $t(F) = t(X)$.
$A(X) = F$ if F is a largest superset of X in E' such that $a(F) = a(X)$.

We will first establish that H is a maximum valid UIS assignment in E' such that $G''[H]$ is acyclic. Let R denote the set of all vertices in G_H which are reachable from s. Formally, $R = \{v \in E' \mid \exists$ a path from s to v in $G_H\}$. Let $NR = E' - R$, $RH = R \cap H$, and $NRH = NR \cap H$.

We claim that $R \subseteq A(RH)$. RH is a valid UIS assignment since its superset H itself is a valid UIS assignment. So, $RH \subseteq A(RH)$. Let $j \in R - RH$. In order to prove our claim it is enough to prove that RH is a maximum valid UIS assignment in $RH \cup \{j\}$. In other words, we have to prove that $RH \cup \{j\}$ is not a valid UIS assignment. Let j be an UIS assignment for the transition e. Suppose that $RH \cup \{j\}$ is a valid UIS assignment then $e \notin dom(RH)$. Also, $e \notin dom(H - R)$ for otherwise an element in $H - R$ will be reachable from s. Therefore, $e \notin dom(H)$ and $(j, t) \in E_H$. But then there exists a path from s to t in G_H because $j \in R$. This is a contradiction.

Our next claim is that $NR \subseteq T(NRH)$. Since $NRH \subseteq T(NRH)$, it is enough to prove that $NR - NRH \subseteq T(NRH)$. Let $j \in NR - NRH$. Note that $G''[NRH]$ is acyclic. We have to prove that the largest acyclic subgraph of $G''[NRH \cup \{j\}]$ is $G''[NRH]$. If not, then $G''[NRH \cup \{j\}]$ is acyclic. But, $G''[H \cup \{j\}]$ has a cycle because $j \notin R$ and so $(s, j) \notin E_H$. Therefore, there exists a $k \in RH$ such that it is contained in the unique cycle in $G''[H \cup \{j\}]$. Then, $(k, j) \in E_H$. $k \in RH$ and $(k, j) \in E_H$ together imply $j \in R$. This is a contradiction as $j \in NR - NRH$.

Let H_{max} be a maximum valid UIS assignment in E' such that $G''[H_{max}]$ is acyclic. Clearly,

$$|H| \leq |H_{max}| \tag{2.1}$$

The following derivation is obtained using the above claims.

$$\begin{aligned}
|H_{max}| &= |H_{max} - R| + |H_{max} \cap R| \\
&= t(H_{max} - R) + a(H_{max} \cap R) \\
&\leq t(E' - R) + a(R) \\
&= t(NR) + a(R) \\
&\leq t(T(NRH)) + a(A(RH)) \\
&= |NRH| + |RH| \\
&= |H| \qquad (2.2)
\end{aligned}$$

Combining the inequalities (2.1) and (2.2) we obtain that H is a maximum valid UIS assignment in E' such that $G''[H]$ is acyclic.

Suppose H does not span all the vertices in G''. Let v be a vertex in G'' which is not spanned by H. Let $e' \in G''$ be an edge incident at v such that $e' = (e, u) \in R$ and $start(e) = v$. Clearly, $H \cup \{e'\}$ is a valid UIS assignment and $G''[H \cup \{e'\}]$ is acyclic. This contradicts that H is a maximum valid UIS assignment. Therefore, H spans all the vertices is G''.

We shall prove by contradiction that $G''[H]$ has the minimum number of connected components. Suppose it is not true. Then, let P be an UIS assignment such that $G''[P]$ is acyclic, spans every vertex in G', and has the minimum number of connected components. Let p and q be the number of connected components in $G''[H]$ and $G''[P]$ respectively. Clearly, $p > q$. Since $G''[H]$ and $G''[P]$ are both acyclic it follows that the number of edges in H and P are $n - p$ and $n - q$, respectively. Since $p > q$, we have $n - p < n - q$. That is, H has less number of edges than P. This is a contradiction since H is a maximum valid UIS assignment with $G''[H]$ acyclic. In other words, $G''[H]$ has the minimum number of connected components. Thus we have proved that H is a maximum valid UIS assignment such that $G''[H]$ is acyclic. In other words, H and $dom(H)$ provide a solution to the BUAP.

□

In the presentation of the solution to the BUAP, we have assumed that G'' is connected and it has no self-loop. Our approach works for the general case as well. Suppose that the graph obtained from G' by removing all the self-loops is not connected. Let $C_1, C_2, \ldots, C_p$ be the connected components of the resulting graph, where $p \geq 2$. Let K_i and B_i be the solutions for the BUAP for C_i, where K_i and B_i denote the edge set and the UIS assignment for K_i, respectively. If C_i is just a single vertex, say s_k, then $B_i = \{e'\}$ and $K_i = \{e\}$, where $e' = (e, u) \in R$ is some self-loop at s_k in G''. If C_i is not a single vertex then its solution is obtained using the algorithm *basic_assignment*. Then it is easy to see that $K = K_1 \cup K_2 \cup \ldots \cup K_p$ and $B = B_1 \cup B_2 \cup \ldots \cup B_p$ form a solution to the original BUAP.

2.3.1 An Illustration of BUAP

As an illustration for the *basic_assignment* algorithm, we consider the FSM as given in Figure 2.2 with $MU_1 = \{t1\}$, $MU_2 = \{t2, t3\}$, and $MU_3 = \{t4, t5, t6\}$ as the sets of multiple UISs for the states s_1, s_2, and s_3, respectively. Note that, we have referred to

Edge	Description	Edge	Description
$e1$	$(s_2, s_2; t2\ t1)$	$e2$	$(s_3, s_2; t5\ t1)$
$e3$	$(s_1, s_1; t1\ t2)$	$e4$	$(s_1, s_3; t1\ t3)$
$e5$	$(s_3, s_1; t4\ t2)$	$e6$	$(s_3, s_3; t4\ t3)$
$e7$	$(s_2, s_2; t3\ t4)$	$e8$	$(s_2, s_1; t3\ t5)$
$e9$	$(s_2, s_3; t3\ t6)$	$e10$	$(s_3, s_2; t6\ t4)$
$e11$	$(s_3, s_1; t6\ t5)$	$e12$	$(s_3, s_3; t6\ t6)$
$e13$	$(s_3, s_2; t7\ t4)$	$e14$	$(s_3, s_1; t7\ t5)$
$e15$	$(s_3, s_3; t7\ t6)$		

Table 2.6: Edge description of G' for the FSM given in Figure 2.2.

an UIS of a state by the corresponding transition along which the sequence lies. The graph $G' = (S, E')$ for the FSM is given in Figure 2.5. Here, $E' = \{e1, e2, \ldots e15\}$. The edges are described in Table 2.6. For convenience, we represent the label of an edge in E' as the sequence of transitions along which the label of the edge lies. As discussed before, *basic_assignment* algorithm considers only the non-self-loop edges in G'. Thus the set $\{e1, e3, e6, e7, e12, e15\}$ of edges are removed from E'.

To start with $H = \emptyset$. The first part of the **repeat...until** loop constructs a digraph $G_H = (V_H, E_H)$, where $V_H = \{s, t\} \cup E'$. E_H is the empty set at the beginning of the first iteration. The edge set E_H after completely executing the first **for** loop is shown in Figure 2.6 as part of the digraph G_H. Observe that, for instance, an edge from s to $e8$ was added to E_H in this loop since $G'[H \cup \{e8\}]$ is acyclic. Also, an edge from $e8$ to t was added to E_H since the transition $t3$, for which $e8$ is a test edge, does not obviously belong to $dom(H) = \emptyset$. Since $(s, e8, t)$ is a shortest path from s to t in G_H, $e8$ is added to H. Therefore, $H = \{e8\}$ and $dom(H) = \{t3\}$. The algorithm enters into the second iteration of the **repeat...until** loop.

The digraph G_H constructed in the second iteration of the **repeat...until** loop is shown in Figure 2.7. Note that an edge from $e9$ to $e8$ is present in G_H since $e9$ and $e8$ are test edges for the same transition $t3$. Since $(s, e5, t)$ is a shortest path from s to t in the current G_H, $e5$ is added to H. Thus $H = \{e8, e5\}$ and $dom(H) = \{t3, t4\}$ at the end of the second iteration of the **repeat...until** loop. Figure 2.8 shows the

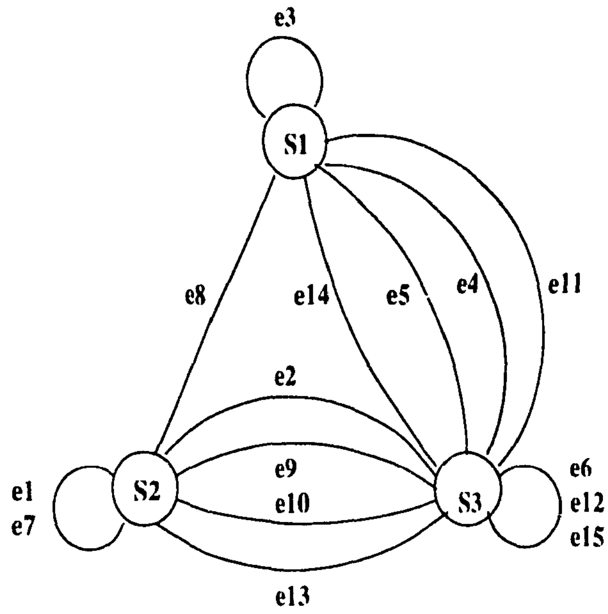


Figure 2.5: G' for the FSM given in Figure 2.2.

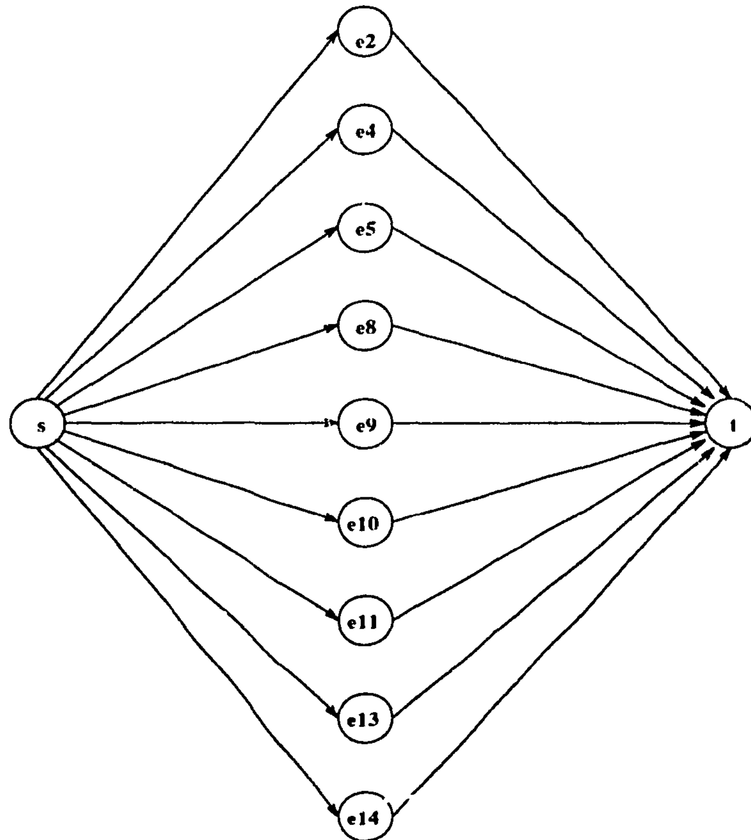


Figure 2.6: G_H in the first iteration of *basic_assignment*.

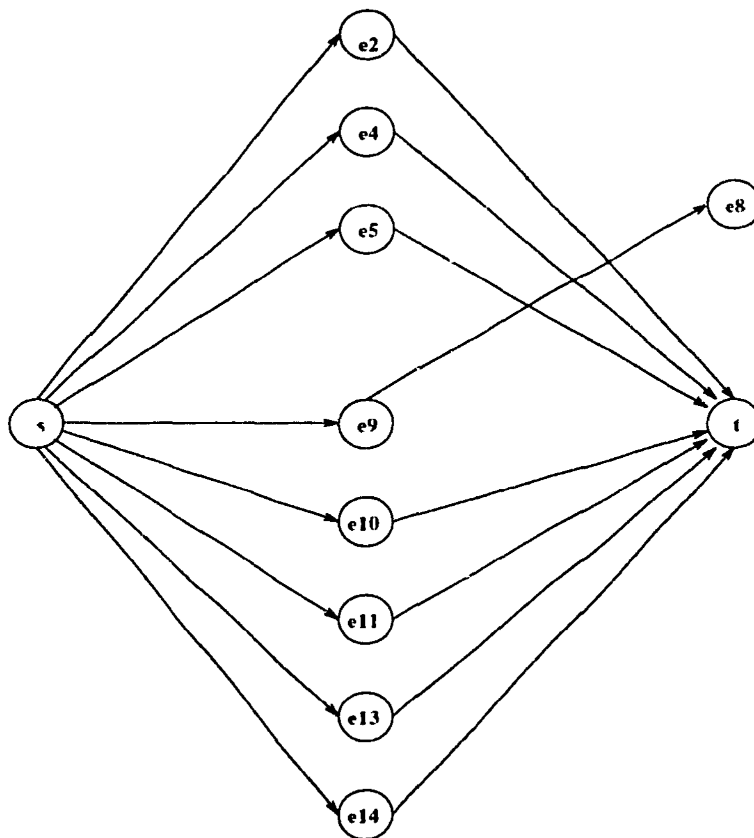


Figure 2.7: G_H in the second iteration of *basic_assignment*.

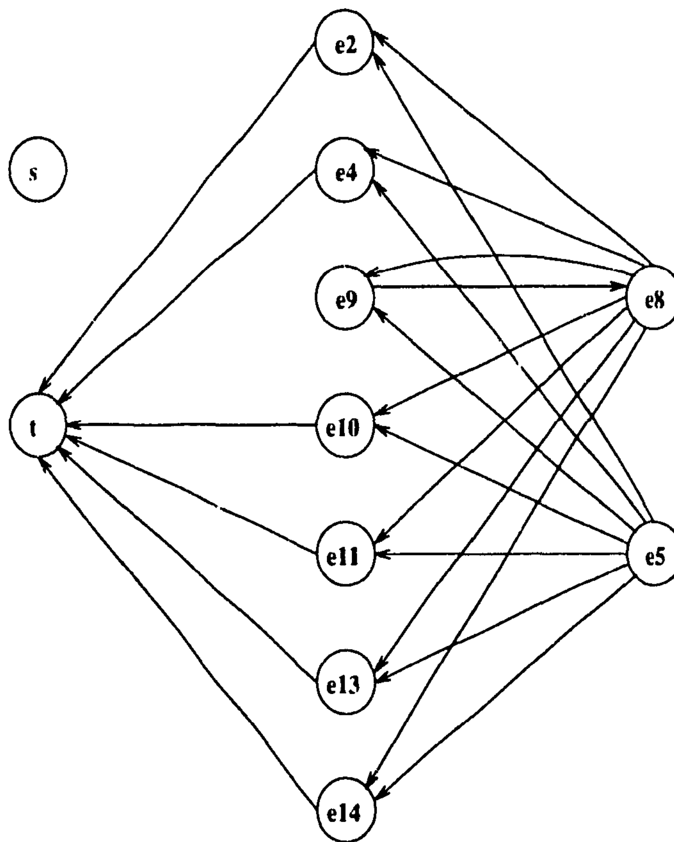


Figure 2.8: G_H in the third iteration of *basic_assignment*.

graph G_H constructed in the third iteration of the loop. Observe that the edge from $e5$ to $e2$, for instance, is present in G_H since $e5$ and $e2$ are contained in the unique cycle in $G''[H \cup \{e2\}]$. The algorithm terminates in this iteration since there is no path from s to t in G_H. Thus the solution to the basic assignment problem at hand is $H = \{e8, e5\}$ with $dom(H) = \{t3, t4\}$. That is, to assign the UIS along $t5$ to the transition $t3$ and the UIS along $t2$ to the transition $t4$. Observe that $G''[H]$ is a spanning tree of G''.

2.4 Algorithms for the Rural Postperson Problem

As stated earlier, given a strongly connected weighted digraph $G = (V, E)$ and an arc subset FF of E, the RPP is to find a tour with minimum cost which traverses each arc in FF at least once. This problem is known to be NP-complete. In this section we present three heuristic algorithms for the RPP.

2.4.1 A Heuristic Algorithm Based on Augmentations

Our first heuristic algorithm *app_rpt* repeatedly applies the rural symmetric augmentation algorithm of Aho *et al* [ADLU88]. We refer to this as *rural_symm_aug* (G, F, G_1, E_1). This algorithm accepts a weighted digraph $G = (V, E)$, and an edge set $F \subseteq E$ and computes a rural symmetric augmentation $G_1 = G[F \cup E_1]$ of G with respect to F by finding a minimum cost bag E_1 of edges from E such that G_1 is symmetric.

The algorithm *app_rpt* consists of three steps. Step 1 calls the algorithm *rural_symm_aug*(G, FF, G_0, E_0) to compute a rural symmetric augmentation G_0 of G with respect to the given set $FF \subseteq E$. If G_0 is weakly connected, then the algorithm outputs an euler tour of G_0 as the required tour and terminates. Otherwise

it proceeds to Step 2. As explained below, this step joins the subtours in G_0 in an iterative fashion by computing rural symmetric augmentations of different auxiliary graphs with respect to some subsets of FF. The idea of joining the subtours is also applied by Frieze *et al* in their heuristic algorithm [FGM82] for the asymmetric traveling salesperson problem [TS92, PS82]. Step 3 further minimizes the cost of the tour obtained at the end of Step 2. The formal description of the algorithm is given below.

Algorithm *app_rpt*(G, FF, Γ)

{ The algorithm finds an approximate RPT Γ of G with respect to FF.}

{ where $G = (V, E)$ is a directed weighted graph with a cost of one unit on }

{ each edge and $FF \subseteq E$. }

Step 1 {Initial rural symmetric augmentation }

rural_symm_aug(G, FF, G_0, E_0);

if (G_0 is weakly connected) **then begin**

Compute an euler tour Γ of G_0 ;

Stop

end

else begin

Let $C_1, C_2, \ldots, C_{|c|}$ be the components of G_0;

$T := FF \cup E_0$; $K := |c|$;

Compute all pair shortest paths in G

end

Step 2 {Compute rural symmetric augmentations of auxiliary graphs }

repeat

{ Construct an auxiliary weighted digraph $G' = (V', E')$ }

$V' := \emptyset$; $E' := \emptyset$;

$F' := \emptyset$;

$V_f := \emptyset$; $V_t := \emptyset$;

for $i := 1$ **to** K **do begin**

Choose an edge $e = (v_f, v_t) \in FF \cap C_i$;

Add v_f to V_f and to V';

Add v_t to V_t and to V';

Add e to E' and to F';

Associate the cost of e in E as the cost of e in E';

end

for each $v_t \in V_t$ **do**

for each $v_f \in V_f$ such that $(v_f, v_t) \notin F'$ **do begin**

Add an edge $e' = (v_t, v_f)$ to E';

Let the cost of e' be that of a shortest path from v_t to v_f in G;

end

Let $G'' = (V', E')$;

rural_symm_aug(G'', F', G^*, E^*) ;

Let T' be the bag of all underlying edges in E for the edges in $E^* \cup F'$;

Add all the edges in T' to T;

Let $C_1, C_2, \ldots, C_h$ be the components of G^*;

$K := h$;

until$(K = 1)$

Step 3 { Delete unwanted edges from T and compute the final tour }

Construct an undirected graph G''' from $G[T]$ by fusing the end vertices of each edge in FF and ignoring the orientation of the remaining edges;

Compute an MST T'' of G''';

Let F'' be the set of edges in E corresponding to the edges in T'';

rural_symm_aug$(G, FF \cup F'', \hat{G}, \hat{E})$;

Compute an euler tour Γ of $\hat{G}$;

end *app_rpt*.

Suppose that G_0 is not weakly connected. Let $C_1, C_2, \ldots C_{|c|}$ be the set of components of G_0. Observe that each component contains at least one edge from FF; for otherwise E_0 will not be a minimum cost bag such that $G_0 = G[FF \cup E_0]$ is symmetric. In this case, the algorithm proceeds to Step 2 after storing $FF \cup E_0$ in the bag T. In general, each iteration of the **repeat..until** loop of this step accepts the set of all components $C_1, C_2, \ldots C_K$, $K \geq 2$, of a symmetric disconnected digraph of the previous iteration and a bag T of edges computed thus far. Each component will have at least one edge from FF. Let $F' \subseteq FF$ such that it has exactly one edge from each component. Let V_f (V_t) be the set of all starting (ending) vertices of the edges in F'. Step 2 first constructs the weighted digraph $G'' = (V', E')$, where $V' = V_f \cup V_t$ and $E' = F' \cup P$, where $P = \{(v_t, v_f) \mid v_t \in V_t \text{ and } v_f \in V_f \text{ and } (v_f, v_t) \notin F'\}$. That is, an edge is added from the ending vertex v_t of each edge in F' to the starting vertex v_f of every other edge in F'. The cost associated with this edge is the cost of a shortest path in G from v_t to v_f. Also, the cost associated with an edge in F' is the cost assigned for this edge in G.

Step 2 then invokes *rural_symm_aug*(G'', F', G^*, E^*) to find a rural symmetric augmentation $G^* = G''[F' \cup E^*]$ of G'' with respect to F'. Let T'' be the bag of all the underlying edges for the edges in $F' \cup E^*$. T'' is added to T. Note that if G^* is weakly connected, then so is $G[T]$. Also, $G[T]$ is a symmetric digraph containing each edge in FF at least once. Therefore, the algorithm proceeds to Step 3 to derive a tour using edges in T. Suppose that G^* is not weakly connected. Then, let $C_1, C_2, \ldots, C_k$ be the components of G^*. In this case, the algorithm repeats the **repeat..until** loop of the Step 2 for this set of components. As shown in Theorem 2.3, the number of components at the starting of a given iteration of the loop is at most half of the number of components in the starting of the previous iteration. Therefore, Step 2 terminates within $\lceil \log |c| \rceil$ iterations.

Step 3 computes a set F'' of edges of minimum cost from T such that $G[FF \cup F'']$

is weakly connected. For this purpose, it reduces G[T] into the graph G'' by fusing the end vertices of each edge in FF and ignoring the orientation of the remaining edges. It then computes a Minimum weight Spanning Tree (MST) T'' [TS92] of G''. F'' is nothing but the set of edges in E corresponding to the edges in T''. The step finally outputs the euler tour of the rural symmetric augmentation $\hat{G}$ of G with respect to $FF \cup F''$ as the required approximate tour. Note that since G[$FF \cup F''$] is weakly connected, the euler tour is guaranteed.

In the following, we establish the correctness of *app_rpt* and the level of approximation of the tour obtained by the algorithm. Let $cost(X)$ be a function which accepts a bag X of edges and output the total cost of all the edges in X, considering each occurrence of an edge in X as being separate.

Lemma 2.1 *Let $G = (V, E)$ be a weighted digraph and let $G[FF \cup E_1]$ be a rural symmetric augmentation of G with respect to $FF \subseteq E$. Suppose that Γ_{opt} is an RPT of G with respect to FF. Then, $cost(FF \cup E_1) \leq cost(\Gamma_{opt})$.*

Proof

Since $G[FF \cup E_1]$ is a rural symmetric augmentation of G with respect to FF, E_1 is a minimum cost bag from E such that $G[FF \cup E_1]$ is symmetric. However, an RPT Γ_{opt} needs a minimum cost bag E_2 from E such that $G[FF \cup E_2]$ is symmetric and weakly connected. Therefore, $cost(FF \cup E_1) \leq cost(FF \cup E_2)$. That is, $cost(FF \cup E_1) \leq cost(\Gamma_{opt})$.

□

Lemma 2.2 *Let $G = (V, E)$ be a weighted digraph and $FF \subseteq E$. Let $F' \subseteq FF$ such that no two edges in F' are adjacent. Let $V' = V_f \cup V_t$, where V_f (V_t) is the set of all starting (ending) vertices of the edges in F'. Let $P = \{(v_t, v_f) \mid v_t \in V_t$ and $v_f \in V_f$ and $(v_f, v_t) \notin F'\}$. Let $G' = (V', E')$ be a weighted digraph, where $E' = F' \cup P$ and the costs assigned to the edges are as follows: For each edge e in F' the cost of e is*

E is assigned as its cost in G'. For each edge $e' = (v_i, v_j) \in P$ the cost of a shortest path in G from v_i to v_j is associated as the cost of e' in G'. Suppose that Γ_{opt} is an RPT of G with respect to F and Γ' is an RPT of G' with respect to F'. Then, $cost(\Gamma') \leq cost(\Gamma_{opt})$.

Proof

In Γ_{opt}, choose one occurrence of each edge in F'. Let f_1 be an arbitrary edge in F'. Label the other edges in F' such that $F' = \{f_1, f_2, \ldots, f_k\}$ and $f_1, f_2, \ldots, f_k$ is the order in which the chosen instances of the edges occur in Γ_{opt}. Let v_f^i and v_t^i be the starting and the ending vertices of f_i, for $i = 1, 2, \ldots, k$. Obtain a tour Υ from Γ_{opt} by replacing the walk in Γ_{opt} from v_t^i to v_f^{i+1}, for $i = 1, 2, \ldots k-1$, by an edge with the cost as the cost of the shortest path in G from v_t^i to v_f^{i+1}. Also, to obtain Υ, replace the walk in Γ_{opt} from v_t^k to v_f^1 by an edge with the cost as the cost of a shortest path in G from v_t^k to v_f^1. Clearly, $cost(\Upsilon) \leq cost(\Gamma_{opt})$. Also, Υ is a tour in G' which visits each edge in F' at least once. Since Γ' is an RPT of G' with respect to F', $cost(\Gamma') \leq cost(\Upsilon)$. Thus, $cost(\Gamma') \leq cost(\Gamma_{opt})$.

□

The lemma given below directly follows from Lemma 2.1 and Lemma 2.2.

Lemma 2.3 *Let G, G', FF, and F' be as defined in Lemma 2.2. Suppose that $G'[F' \cup E^*]$ is a rural symmetric augmentation of G' with respect to F' and Γ_{opt} is an RPT of G with respect to FF. Then, $cost(F' \cup E^*) \leq cost(\Gamma_{opt})$.*

□

We prove in the following theorem that the cost of the tour Γ produced by our *app_rpt* algorithm is $(1 + \lceil log|c| \rceil)$-**approximate**[3] to the cost of an RPT of G with respect to FF, where $|c|$ is the number of weakly connected components of the rural symmetric augmentation obtained in Step 1 of the algorithm.

[3]The cost of a tour A is said to be **k-approximate** to that of a tour B if $\frac{cost(A)}{cost(B)} \leq k$, where $cost(A), cost(B) \geq 0$, $k \geq 1$ and $cost(B) \neq 0$.

Theorem 2.3 *Suppose that Γ_{opt} is an RPT of G with respect to FF and that Γ is the output given by algorithm* app_rpt. *Then Γ is a single tour containing each edge in FF at least once and $cost(\Gamma) \leq (1 + \lceil log|c| \rceil) cost(\Gamma_{opt})$, where $|c|$ is the number of weakly connected components of the rural symmetric augmentation obtained in Step 1 of the algorithm. That is, the cost of Γ is $(1 + \lceil log|c| \rceil)$-approximate to the cost of an RPT of G with respect to FF.*

Proof

If the rural symmetric augmentation G_0 of G with respect to FF is weakly connected, then clearly Γ is an euler tour of G_0 and so it contains each edge in FF at least once. By lemma 2.1, we have $cost(\Gamma) \leq cost(\Gamma_{opt})$. Since G_0 is also weakly connected, from the definition of an RPT, we have $cost(\Gamma_{opt}) \leq cost(\Gamma)$. Thus, $cost(\Gamma) = cost(\Gamma_{opt})$.

If G_0 is not weakly connected, then all the edges in FF as well as the bag (E_0) of edges computed for the rural symmetric augmentation are stored in T and the **repeat...until** loop of Step 2 is executed with this T and the components $C_1, C_2, \ldots C_{|c|}$ of G_0. Note that $cost(T) \leq cost(\Gamma_{opt})$ before entering Step 2. From the construction of F' and G'', we know that each component of the rural symmetric augmentation $G^* = G''[F' \cup E^*]$ of G'' with respect to F' contains at least two edges from F', where E^* is a minimum cost bag in E' such that G^* is symmetric. Therefore, F' is reduced at least by a factor of 2 between successive iterations of the **repeat...until** loop. At the starting of the first iteration $|F'| = |c|$. Therefore, this loop is repeated at most $\lceil \log |c| \rceil$ iterations. In each iteration of this loop, the bag T' of edges in E corresponding to the bag $F' \cup E^*$ of edges is added to T. Since the rural symmetric augmentation G^* in the last iteration of the loop is weakly connected, $G[T]$ is also weakly connected. Also, since FF is added to T in Step 1 and none of the edges in T is deleted in Step 2, T contains all the edges in FF. From the construction of G'' in Step 3, it follows that $G[FF \cup F'']$ is weakly connected, where F'' is the set of edges corresponding to the MST of G''. Therefore, the rural symmetric augmentation $\hat{G}$ of

$\hat{G}$ with respect to $FF \cup F''$ is clearly eulerian and the euler tour Γ of $\hat{G}$ contains all the edges in FF.

From Lemma 2.3, we know that $cost(T') \leq cost(\Gamma_{opt})$ at the end of a given iteration of the **repeat...until** loop of Step 2. Since $cost(T) \leq cost(\Gamma_{opt})$ before entering Step 2 and since T' is added to T at the end of every iteration of this loop which is repeated at most $\lceil \log |c| \rceil$ times, $cost(T) \leq (1 + \lceil \log |c| \rceil) cost(\Gamma_{opt})$. Step 3 computes a set F'' of edges from T such that $G[FF \cup F'']$ is weakly connected. It then finds a rural symmetric augmentation $\hat{G}$ of G with respect to $FF \cup F''$. Let Γ be the euler tour of $\hat{G}$. Since $G[T]$ is symmetric and contains each edge in $FF \cup F''$, $cost(\Gamma) \leq cost(T)$. But $cost(T) \leq (1 + \lceil \log |c| \rceil) cost(\Gamma_{opt})$. Therefore, $cost(\Gamma) \leq (1 + \lceil \log |c| \rceil) cost(\Gamma_{opt})$. This completes the proof .

□

As given in [SLD92], the time complexity of most practical algorithms for finding the rural symmetric augmentation of a graph G with respect to a set of edges FF is $O(m^2 \log n)$, where m and n are the number of edges and the number of vertices in the graph G, respectively. Step 2 of *app_rpt* takes $O(|c|^4 \log |c|)$ time units, where $|c|$ is the number of weakly connected components in G_0, the rural symmetric augmentation computed in Step 1. It follows that the time complexity of the algorithm is $O(m^2 \log n + |c|^4 \log |c|)$.

2.4.2 A Heuristic Algorithm Based on MST

One naive approach of finding a tour covering each arc in FF at least once when the induced graph G[FF] is not weakly connected is by connecting the disconnected components of G[FF] by a set of arcs with minimum cost and finding a rural symmetric augmentation of G with respect to the new set of arcs. The approach is naive because it simply connects the components which would have been connected otherwise through the rural symmetric augmentation itself. Our heuristic algorithm based on

trees takes care of this point. The algorithm is so named because Step 2 and Step 3 of the algorithm involve the computation of minimum weight spanning trees of certain auxiliary graphs. The algorithm is given below. It starts with the computation of a rural symmetric augmentation G_1 of G with respect to FF. If G_1 is weakly connected then we can find the RPT right away. Otherwise in Step 2 we compute a set of arcs F_1 with minimum cost among the arcs added in Step 1 so that the weakly connected components of $G[FF]$ which belong to the same component in G_1 are also in the same component in $G[FF \cup F_1]$. A set of arcs F_2 with minimum cost which makes $G[FF \cup F_1 \cup F_2]$ weakly connected is computed in Step 3. The required tour is obtained in Step 4 by finding the euler tour of the rural symmetric augmentation graph G_4 of G with respect to $FF \cup F_1 \cup F_2$.

Algorithm *mst_rpt*(G FF,Γ)

Step 1 {Rural symmetric augmentation}

- Compute the rural symmetric augmentation G_1 of G with respect to FF.
- If G_1 is weakly connected then compute the euler tour of G_1 and stop.
  Else compute the weakly connected components $C_1, C_2, \ldots C_{|c|}$ of G_1.

Step 2 {Find the MST of the weakly connected components}

- For each component C_i construct the undirected graph C_i' from C_i by fusing the end vertices of arcs in $FF \cap C_i$ and dropping the orientation of the remaining arcs.
- Compute the MST T_i' of C_i', $i = 1, 2, \ldots |c|$. Let $F_1 = T_1 \cup T_2 \cup \ldots T_{|c|}$ where T_i is the set of arcs in G corresponding to the edges in T_i', $i = 1, 2, \ldots, |c|$.

Step 3 {Minimum cost arc set for connecting the components}

- Construct a weighted undirected graph $G_3 = (V_3, E_3)$ where $V_3 = \{v_1^c, v_2^c, \ldots v_{|c|}^c\}$ and $e = (v_i^c, v_j^c) \in E_3$ iff $\exists$ a path in G between a vertex in C_i and a vertex in C_j whose intermediate vertices (possibly empty) are only from $V - V(G_1)$; the cost of the edge e is the minimum cost over all such paths in G. Here, $V(G_1)$ denotes the vertex set of G_1.
- Compute a minimum weight spanning tree T of G_3. Let F_2 be the set of arcs in G corresponding to the edges in T.

Step 4 {Rural symmetric augmentation}

- Compute the rural symmetric augmentation G_4 of G with respect to $FF \cup F_1 \cup F_2$.
- Compute the euler tour Γ of G_4 .

end *mst_rpt*

By the constructions in Step 2 and Step 3, $G[FF \cup F_1 \cup F_2]$ is weakly connected. Thus the algorithm guarantees a tour visiting each arc in FF at least once. The time complexity of this algorithm is dominated by the complexity of finding the rural symmetric augmentations. Thus the algorithm takes $O(m^2 \log n)$ time units.

2.4.3 A Heuristic Algorithm Based on Symmetry

This algorithm differs from the previous algorithm only in Step 3. At the end of Step 2 the induced graph $G[FF \cup F_1]$ may have a number of weakly connected components. In Step 3, we compute a set F_2 of arcs such that $G[FF \cup F_1 \cup F_2]$ is weakly connected and F_2 contributes to an optimal extent in making $G[FF \cup F_1 \cup F_2]$ symmetric. This is done using the procedure *symm_connect*, which is based on a greedy approach. Given a graph $G = (V, E)$ and a set $X \subseteq E$, each vertex is assigned one of the following three attributes: excess, deficient and neutral. A vertex v has the attribute **excess, deficient**, or **neutral** if the number of incoming arcs (from the set X) at vertex v respectively is greater than, less than, or equal to the number of outgoing arcs (from the arc set X) at the vertex v. The algorithm is given below.

Algorithm *symm_rpt*(G,FF,Γ)

Step 1 {Rural symmetric augmentation}

- Compute the rural symmetric augmentation G_1 of G with respect to FF.
- If G_1 is weakly connected then compute the euler tour of G_1 and stop. Else compute the weakly connected components $C_1, C_2, \ldots C_{|c|}$ of G_1.

Step 2 {Find the MST of the weakly connected components}

- For each component C_i construct the undirected graph C_i'' from C_i by fusing the end vertices of arcs in $FF \cap C_i$ and dropping the orientation of the remaining arcs
- Compute the MST T_i' of C_i'', $i = 1, 2, \ldots |c|$. Let $F_1 = T_1 \cup T_2 \cup \ldots T_{|c|}$ where T_i is the set of arcs in G corresponding to the edges in T_i', $i = 1, 2, \ldots, |c|$.

Step 3 {Find an arc set(with contributions to symmetry) for connecting components }

- Perform procedure *symm_connect*(G, $FF \cup F_1$, F_2).

Step 4 { Rural symmetric augmentation }

- Compute the rural symmetric augmentation G_3 of G with respect to $FF \cup F_1 \cup F_2$
- Compute the euler tour Γ of G_3.

end *symm_rpt*

The formal description of the procedure *symm_connect* is given below:

Procedure *symm_connect*(G,FF,F')

Step 1

Compute the weakly connected components of $G[FF]$;

Let F' be empty;

Mark an arbitrary component selected and the others unselected;

Compute the attribute of each vertex in G with respect to $FF \cup F'$.

Step 2

while (there exist an unselected component **and**
a path from an excess vertex of a selected (unselected) component to a deficient vertex of an unselected (selected) component) **do**

begin

Find one such path, say P, with minimum cost;

Mark all unselected components along P as selected;

Add all the arcs along P into F';

Reassign the attributes for the end vertices of P with respect to $FF \cup F'$;

end

<u>**Step 3**</u>

while(there exists an unselected component **and**

(there is a path from an excess vertex of a selected (unselected) component to a neutral vertex of an unselected (selected) component **or**

there exists a path from a neutral vertex of a selected (unselected) component to a deficient vertex of an unselected (selected) component)) **do**

begin

Find one such path say P with minimum cost;

Mark all unselected components along P as selected,

Add all the arcs along P into F';

Reassign the attributes for the end vertices of P with respect to $FF \cup F'$;

Perform Step 2;

end

<u>**Step 4**</u>

while (there is an unselected component) **do**

begin

Choose a path P with minimum cost between a selected component and an unselected component;

Mark all unselected components along P as selected;

Add all the arcs along P into F';

Reassign the attributes for the end vertices of P with respect to $FF \cup F'$;

Perform Step 3;

end

end *symm_connect*

It is easy to see that the graph $G[FF \cup F_1 \cup F_2]$ is weakly connected and so the graph G_3 is eulerian. The time complexity of this algorithm is $O(m^2 \log n + n^2|c|)$, where $|c|$ is the number of weakly connected components in G_1.

As far as the selection of a method for solving the general RPP is concerned, if an explicit bound on the optimality of the tour is of prime importance, then one could use the heuristic algorithm *app_rpt* rather than the other two. The approach taken in this thesis is to apply all three heuristics and choose the tour of shortest length among the tours obtained.

2.5 Generalized UIS testing method

As pointed before the existing UIS-based methods can be applied only to a subset of strongly connected protocols which are represented as FSMs having at least one UIS for each state. These methods, however, have the merit of generating minimum length test sequences with high fault coverage [MCS93]. In this section, we propose a generalized approach which can be applied to any protocol satisfying the above conditions. This method produces test sequences with varied level of optimality depending on the structure of the protocol and the UISs considered.

2.5.1 The GU-method

Our GU-method is based on the BUAP and the RPP. The method is formally described in algorithm: *guio_test*. We assume that the protocol representation graph $G_s = (S, E)$ is strongly connected and each of its states has a nonempty set of UISs. As defined earlier, MU_i is a nonempty set of UISs for each state s_i. Let $MU = MU_1 \cup MU_2 \cup \ldots \cup MU_n$. Let $R \subseteq E \times MU$ such that $(e, u) \in R$ iff $end(e) = head(u)$. Let $G' = (S, E')$, where $E' = \{(start(e), tail(u); label(e)@u) | (e, u) \in R\}$. A cost is assigned to each edge in E'. Suppose $e' \in E'$ corresponds to $(e, u) \in R$, then the cost assigned to e' is the length of the UIS u plus one. The generalized method starts by

checking whether the MU-method can be applied to generate a test sequence for the given protocol. If so, the method computes a minimum length test sequence in Step 1 using the MU-method and terminates. If the MU-method fails to find a solution, then the generalized method uses the UIS assignment B obtained in the MU-method to calculate an approximate RPT Γ_1 of $G_s + B$ with respect to B by choosing a tour with the minimum cost among the tours obtained by the algorithms *app_rpt*, *mst_rpt* and *symm_rpt*. In Step 2, the method finds a minimum set of transitions BE and a valid UIS assignment H for BE such that the resulting test graph $G'[H]$ spans G' and $G'[H]$ has the minimum number of connected components. This is done by invoking the *basic_assignment* algorithm.

In order to minimize the length of the test sequence, a valid UIS assignment H' for the transitions in $E - BE$ and a minimum bag EP of transitions from E are computed in Step 3 such that the graph $G'' = G'[H \cup H'] + EP$ is symmetric. Note that each transition in EP is repeated in G'' as many times as they occur in EP. H' and EP are obtained by computing a minimum cost maximum flow f^* of a multi-stage flow graph $G_f = (V_f, E_f)$ whose construction is described below. $V_f = \{s, t\} \cup V_x \cup V_y \cup V_z$ where $V_x = \{x_1, x_2, \ldots, x_n\}$, $V_y = \{y_1, y_2, \ldots, y_n\}$, and $V_z = \{z_1, z_2, \ldots, z_n\}$. $E_f = E_{sx} \cup E_{xy} \cup E_{yz} \cup E_{zz} \cup E_{zt} \cup E_{xz}$ where, $E_{sx} = \{(s, x_i) \mid 1 \leq i \leq n\}$, $E_{xy} = \{(x_i, y_j) \mid \exists e \in E - BE \wedge start(e) = s_i \wedge end(e) = s_j\}$, $E_{yz} = \{(y_i, z_j) \mid \exists u \in MU_i \wedge tail(u) = s_j\}$, $E_{zz} = \{(z_i, z_j) \mid \exists e \in E \wedge start(e) = s_i \wedge end(e) = s_j\}$, $E_{zt} = \{(z_i, t) \mid 1 \leq i \leq n\}$, and $E_{xz} = \{(x_i, z_j) \mid \exists (e, u) \in H \wedge start(e) = s_i \wedge tail(u) = s_j\}$. The lower bound, cost, and capacity assigned to the edges of G_f are shown in Table 2.7.

Assignment of UISs for the transitions in $E - BE$ is done using the optimum flow f^* in G_f. For instance, an unit flow from x_i to z_k through the vertex y_j indicates that the UIS $u \in MU_j$ with $tail(u) = s_k$ is to be assigned to a transition in $E - BE$ from s_i to s_j. This assignment H' is computed in Step 3. EP is obtained by adding a transition from s_i to s_j to EP as many times as the flow $f^*(z_i, z_j)$ along the edge $(z_i, z_j) \in E_{zz}$.

Edge	lower bound	cost	capacity
(s, x_i)	0	0	# of out trans. from s_i in E
(x_i, y_j)	0	0	# of trans. from s_i to s_j in $E - BE$
(y_i, z_j)	0	$length(u)$, where $head(u) = s_i \wedge tail(u) = s_j$	∞
(z_i, z_j)	0	1	∞
(z_i, t)	0	0	# of out trans. from s_i in E
(x_i, z_j)	1	0	1

Table 2.7: Parameters on the edges of the flow graph G_f

If G'' is connected, then G'' is eulerian and an euler tour Γ_2 of G'' is computed in Step 4. Otherwise, an approximate RPT Γ_2 of $G_s + F$ with respect to F is computed in this step by choosing a tour with the minimum cost among the tours obtained by the heuristic algorithms *app_rpt*, *mst_rpt* and *symm_rpt*, where $F = H \cup H'$. The tour with minimum cost is chosen from Γ_1 and Γ_2 and a test sequence is obtained from this tour by concatenating the subsequences and/or the inputs of the transitions along this tour. The algorithm *guio_test* is described below.

Algorithm *guio_test*(G_s, MU, G', TS).

Step 1

Apply the MU-method;

Let B be the UIS assignment computed in the MU-method;

Let E_1 be a bag in E computed in the MU-method

such that $G'[B] + E_1$ is symmetric;

if ($G'[B] + E_1$ is connected) **then**

begin

Obtain the test sequence TS by concatenating the labels

of the edges along the euler tour Γ of $G'[B] + E_1$;

stop

end

else begin

app_rpt$(G_s + B, B, X)$;

mst_rpt$(G_s + B, B, Y)$;

symm_rpt$(G_s + B, B, Z)$;

Let Γ_1 be the tour with minimum cost among X, Y and Z;

end

Step 2

$basic_assignment(G_s, MU, G', H)$;

$BE \leftarrow dom(H)$;

Step 3

Compute a minimum cost maximum flow f^* of $G_f(V_f, E_f)$.

$H' \leftarrow \emptyset$;

for each $e = (s_i, s_j; a/o) \in E - BE$ **do**

begin

$f^*(x_i, y_j) \leftarrow f^*(x_i, y_j) - 1$;

Choose $(y_j, z_k) \in E_f$ such that $f^*(y_j, z_k) > 0$.

$f^*(y_j, z_k) \leftarrow f^*(y_j, z_k) - 1$;

Let $u \in MU_j$ such that $head(u) = s_j$ and $tail(u) = s_k$;

Add (e, u) to H';

end

$F = \{(start(e), tail(u); label(e)@u) | (e, u) \in H \cup H'\}$;

$EP \leftarrow \emptyset$;

for each $(z_i, z_j) \in E_f$ such that $f^*(z_i, z_j) > 0$ **do**

Add a transition from s_i to s_j with minimum cost to EP $f^*(z_i, z_j)$ times ,

Step 4

if ($G'' = G'[H \cup H'] + EP$ is connected) **then**

Find the euler tour Γ_2 of G'',

else begin

$app_rpt(G_s + F, F, X)$;

$mst_rpt(G_s + F, F, Y)$.

$symm_rpt(G_s + F, F, Z)$;

Let Γ_2 be the tour with minimum cost among X, Y and Z;

end

Obtain the test sequence TS by concatenating the labels of

the edges along the tour with minimum cost between Γ_1 and Γ_2.

end *guio_test*.

We would like to note that the multi-stage flow problem formulation used above

is similar to the one given in [SL92] for assigning UISs to the transitions in E. While the assignment obtained in [SL92] may not result in a connected test graph, an optimal flow of our flow graph always yields a connected test graph whenever such a test graph exists. This is due to the fact that any optimal flow in our flow graph always subsumes the UIS assignment obtained as a solution to the BUAP in Step 2. It can be seen that the rural symmetric augmentations made in the first step of *app_rpt*, *mst_rpt*, and *symm_rpt* are redundant, as far as the *guio_test* is concerned, since similar augmentation is already done in Step 1 or Step 3 of *guio_test*. We do not however modify the former algorithms due to their generic application.

The algorithm takes at most $O(n^2m^2\nu^2 + c^4 \log c)$ time units. Here, $c = \max\{|c_1|, |c_2|\}$, where $|c_1|$ and $|c_1|$ are the number of weakly connected components of $G''[B] + E_1$ and G'', respectively. As before, ν denotes the maximum number of UISs in MU for any state. The level of optimality of the test sequence obtained by our generalized method are summarized in the following theorem. Proof of the theorem directly follows from Theorem 2.3 and the algorithm *guio_test*.

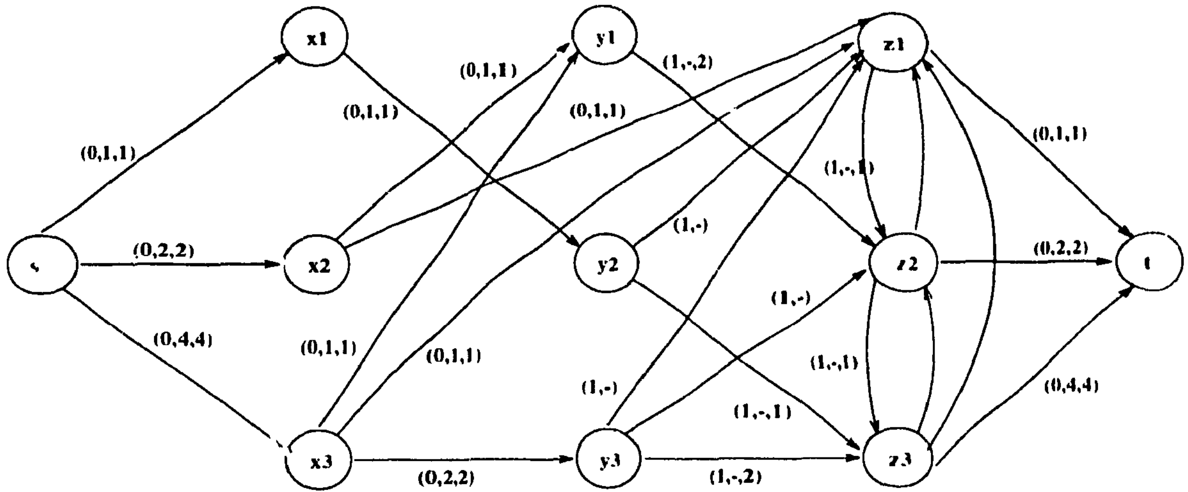
Theorem 2.4 *The length of the test sequence generated in the generalized method has the following levels of optimality.*

(i) if $G''[B] + E_1$ is connected then it is optimum

(ii) if G'' is connected then it is optimum subject to the condition that the edges in $dom(H)$ are preassigned using H, a solution to the BUAP.

(iii) In the worst case, it is always $(1 + \lceil\log(\min\{|c_1|, |c_2|\})\rceil)$-approximate to the length of an optimal test sequence, where $|c_1|$ and $|c_2|$ are the number of connected components of $G''[B] + E_1$ and G'', respectively.

□



Note: (i) '-' indicates the infinite capacity (ii) Edge lables are the triplet (cost, capacity, flow), the last entry (flow) is omitted if it is zero (iii) (1, -) is assumed on all the unlabelled edges (iv) Assign a lower bound of one unit on edges (x2, z1) and (x3, z1) and zero for other edges

Figure 2.9: Flow graph for the FSM given in Figure 2.2.

Transitions	UIS
t1	t3
t2, t5	t1
t6, t7	t6

Table 2.8: UIS assignment using min-cost flow

2.5.2 An Illustration

We now illustrate the proposed method on the FSM given in Figure 2.2. Let us consider the same sets of multiple UISs which are used in Section 2.2.3: $MU_1 = \{t1\}$, $MU_2 = \{t2, t3\}$, and $MU_3 = \{t4, t5, t6\}$. The generalized method finds a test sequence of minimum length (14) if $G'[B] + E_1$, computed in Step 1, is connected. If not, let us suppose that the UIS assignment of the FSM obtained in Step 1 is A_1 as given in Table 2.4. Since $G'[A_1]$ itself is symmetric, $E_1 = \emptyset$. Since $G'[A_1] + E_1$ is not connected the heuristic algorithms have to be invoked for computing an approximate RPT with respect to A_1. Suppose that $\{e3, e9\}$, where $e3 = (s_1, s_1; t1\ t2)$ and $e9 = (s_2, s_3; t3\ t6)$ (refer to Table 2.6), is chosen as F' in the first iteration of the **repeat...until** loop of the algorithm *app_rpt*. Then $T' = \{t1, t5\}$ and *app_rpt* moves to the third step. In Step 3 of *app_rpt*, the edge $t3$ is added so that $G'[A_1] + E_1$, $t1$, $t5$, and $t3$ together form the tour X: *t1 t2 t1 t2 t1 t3 t6 t5 t1 t3 t4 t3 t6 t6 t7 t6 t5* of length 17. Since each edge in G_1 is in fact a test edge, F_1 computed in Step 2 of *mst_rpt* as well as *symm_rpt* is the empty set. Suppose that the transition *t1* is selected in Step 3 of *mst_rpt*. Then the last step of this algorithm adds *t2* to $A_1 \cup \{t1\}$ so that $G[A_1 \cup \{t1\} \cup \{t2\}]$ is symmetric. The algorithm *mst_rpt* returns an approximate RPT Y: *t1 t2 t1 t3 t6 t6 t6 t7 t6 t4 t3 t5 t1 t2 t1 t2* of length 16. Similarly, the algorithm *symm_rpt* produces an approximate RPT Z of length 16 or 17 depending on the selection of the transition *t1 (t2)* or *t5*, respectively, in the third step. Let Γ_1 be the tour Y, the one with the minimum length (16) among X, Y and Z.

The algorithm *basic_assignment* is invoked at Step 2 with the above set of multiple UISs. Suppose that the algorithm *basic_assignment* assigns the UISs $t5$ and $t2$ to transitions $t3$ and $t4$, respectively. Note that this assignment(H) in fact yields a connected test graph. The multi-stage flow graph for computing the UIS assignment for the remaining transitions as well as a set of transitions to be added for obtaining G'' is shown in Figure 2.9. Labels in each edge is a triplet representing the cost.

capacity, and the optimal flow, in that order. The last part of the triplet is omitted if the optimum solution has a zero flow along that edge. Edges (x_2, z_1) and (x_3, z_1) also have a unit lower bound. The resulting UIS assignment(H') for the remaining transitions are shown in Table 2.8. The solution also indicates that *t1* and *t3* are the only additional transitions required for obtaining a rural symmetric augmentation of $G_s + (H \cup H')$ with respect to $H \cup H'$. *t1 t3 t6 t6 t7 t6 t5 t1 t2 t1 t3 t5 t1 t3 t4 t2* is the resulting tour Γ_2. Note that the length of Γ_2 is same as the length of Γ_1. A test sequence is obtained by concatenating the input-output of the transitions along Γ_2. Observe that our generalized method produces a test sequence of length 16, two more than the optimum test sequence, whereas the MU-method by itself does not guarantee a test sequence.

2.6 Summary

The optimal UIS-based test generation methods (U-method [ADLU88] and MU-method [SLD92]) do not cover certain protocols which are represented as strongly connected FSMs having at least one UIS for each state. In this chapter we have generalized the MU-method so that it can be applied on any such protocol. Observe that our method can also be applied for protocols without reset capability. The method generates test sequences of different levels of optimality depending on the structure of the protocol as well as the set of UISs used. The method uses solutions to the Basic UIS assignment Problem, and the Rural Postperson Problem. The BUAP is formulated as an MC2MIP and an efficient algorithm based on the MC2MIP is presented. Three heuristic algorithms, including one with an explicit bound on the optimality of the solution, are proposed for the general RPP.

The work presented in this chapter is reported in [RTD94].

Chapter 3

ANALYSIS OF FSM-BASED TEST SEQUENCE GENERATION METHODS

In this chapter, we review and analyze FSM-based test sequence generation methods for their fault detection and fault diagnosis capabilities. We also summarize results on the lengths of the test sequences they generate. As suggested in [Ura92] this comprehensive study will be useful in selecting suitable methods for generating test sequences for a given protocol. A formal analysis of the fault diagnosis capabilities of the methods will also aid one in choosing a suitable method for diagnosing the faults in an implementation [Ghe92]. In addition, this analysis will help understand the complexities involved in developing test sequence generation methods with greater fault detection and fault diagnosis capabilities. Detailed descriptions of these test sequence generation methods may be found in [Gil61, Hen64, KL67, KK68, Gon70, KRK74, Koh78, Cho78, NT81, ADLU88, SD88, CVI89, SL89, FBK+91, Ura92] . Analysis of fault coverage of some of these methods may also be found in [SL89, CVI89, FBK+91, BPY94].

The chapter is organized as follows. In Section 3.1, we review various FSM-based

test sequence generation methods and analyze their fault detection and diagnosis capabilities. These methods are compared in Section 3.2 with respect to their fault detection and diagnosis capabilities and the length of the test sequences the methods generate. Some of the desirable characteristics a testing method should have in order to have better diagnosis capabilities are discussed in the concluding section.

3.1 Test Sequence Generation Methods: Review and Analysis

In this section we review FSM-based test sequence generation methods and analyze their fault detection and fault diagnosis capabilities with respect to the fault model introduced in Section 1.2.1. We examine the fault coverage and the 1-fault resolution capabilities of these methods. We denote the FSM representations of the protocol specification and an implementation by SPEC and IUT, respectively. In the analysis of the 1-fault resolution capabilities of the methods, the IUTs are assumed to have at most one fault (either a transfer fault or an output fault in a transition). Thus, a faulty IUT is identical to the SPEC except for the unique faulty transition. In all these methods, n and $|E|$ will denote the number of states, and the number of transitions in the SPEC, respectively.

Let $TEST(s_i, s_j; a/o)$ denote a set of walks for testing the transition $(s_i, s_j; a/o)$. That is, $\{Inseq(W) \mid W \in TEST(s_i, s_j; a/o)\}$ is the set of input sequences for testing $(s_i, s_j; a/o)$. Suppose that $TEST(s_i, s_j; a/o)$ is a singleton, then the unique walk in this set itself is denoted by $TEST(s_i, s_j; a/o)$. Each walk W in $TEST(s_i, s_j; a/o)$ can be divided into three parts:

$$W = preamble @ body @ postamble$$

The *preamble* is a path for putting the IUT in the state s_i from its current state. The walk for traversing and testing the transition $(s_i, s_j; a/o)$ constitutes the ***body***. The

postamble is a path for putting the IUT into a known state after traversing the *body*.

3.1.1 Distinguishing Sequence Method

The distinguishing sequence method (in short, the *D-method*) [Hen64, Gon70, Koh78] assumes that the SPEC is strongly connected, minimal, and completely specified. The IUT is assumed to have at most n states. It is assumed that the SPEC has a special type of input sequence called a distinguishing sequence. Formally, an input sequence X_0 is said to be a **distinguishing sequence** of a SPEC if the output sequence obtained while applying X_0 at each state in the SPEC is distinct. Let $D(i)$ denote the walk from s_i with the input sequence X_0. Let $q_i = Tail(i, X_0)$. That is, q_i is the state the SPEC reaches after applying X_0 at s_i, $i = 1, 2, \ldots, n$. Let $T(i, j)$ denote a shortest path from s_i to s_j in the SPEC, where $1 \leq i, j \leq n$. The D-method involves two phases: the state verification phase and the transition testing phase. Given a distinguishing sequence X_0, the state verification phase tests whether (i) the IUT has exactly n states and (ii) X_0 is also a distinguishing sequence of the IUT. This phase is performed using the input sequence along the following walk.

$$D(1)\ T(q_1, 2)\ D(2)\ T(q_2, 3) \ldots D(n)\ T(q_n, 1)\ D(1)$$

If we get the expected output sequence on applying the input sequence along the above walk to the IUT, then clearly, the IUT also has n distinct states. In the transition testing phase each transition $t = (s_i, s_j; a/o) \in E$ is tested using the input sequence along $T(q_p, i-1)\ D(i-1)\ T(q_{i-1}, i)\ t\ D(j)$ where, q_p is the state of the IUT before starting the testing of transition $(s_i, s_j; a/o)$. The prefix $T(q_p, i-1)\ D(i-1)\ T(q_{i-1}, i)$ is to ensure that the IUT is put in state s_i before applying the input of the transition $(s_i, s_j; a/o)$. $D(j)$ is to confirm whether the tail state of the transition under test is in fact s_j. In order to reduce the length of the test sequence one can test some of the transitions in the state verification phase itself. The main advantage of the D-method is that it ensures complete fault coverage [FBK+91]. It is also likely that the

State	output for $axbxx$	State	output for $axbxx$
s_1	00101	s_2	00000
s_3	00001	s_4	20000
s_5	12101	s_6	02101
s_7	20101	s_8	22101

Table 3.1: Responses of the SPEC for the distinguishing sequence

length of the test sequence will be smaller than those obtained by other test sequence generation methods having state verification phase.

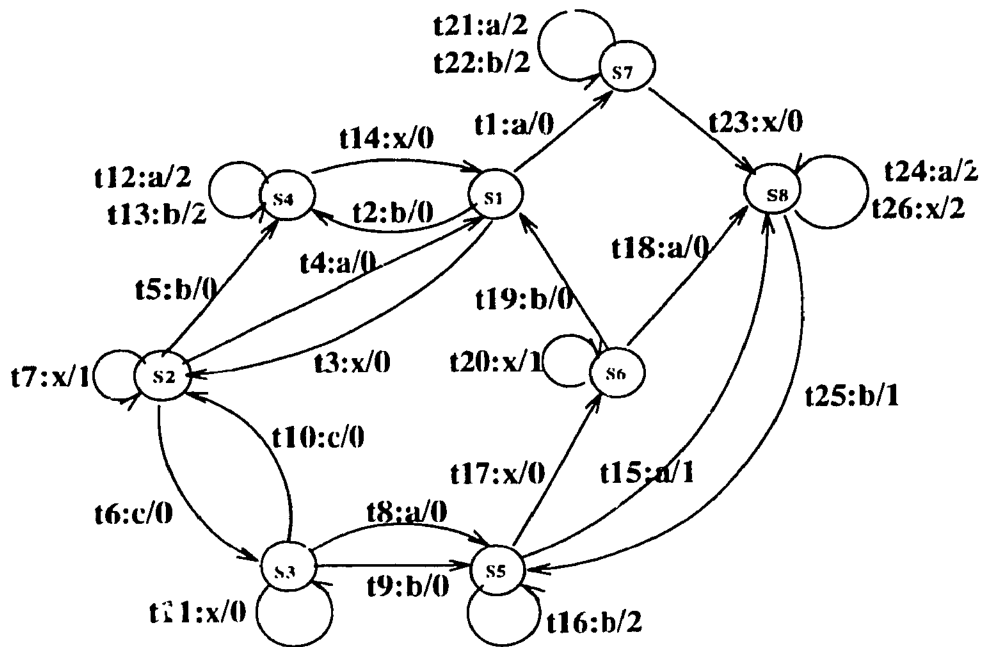
In the following we present our results on the analysis of the 1-fault resolution capability of this method. Our first claim is that its fault diagnosis capability is very limited if the IUT fails in the state verification phase. For example consider the SPEC and the IUT given in Figure 3.1. From Table 3.1, it is easy to see that $X_0 = axbxx$ is a distinguishing sequence of the SPEC. Observe that the transition $t3 = (s_1, s_2; x/0)$ of the SPEC has a transfer fault in the IUT. Consider the input sequence along the walk

$$D(1)\ T(q_1, 2)\ D(2)\ T(q_2, 3)\ D(3)\ T(q_3, 4)\ D(4)$$

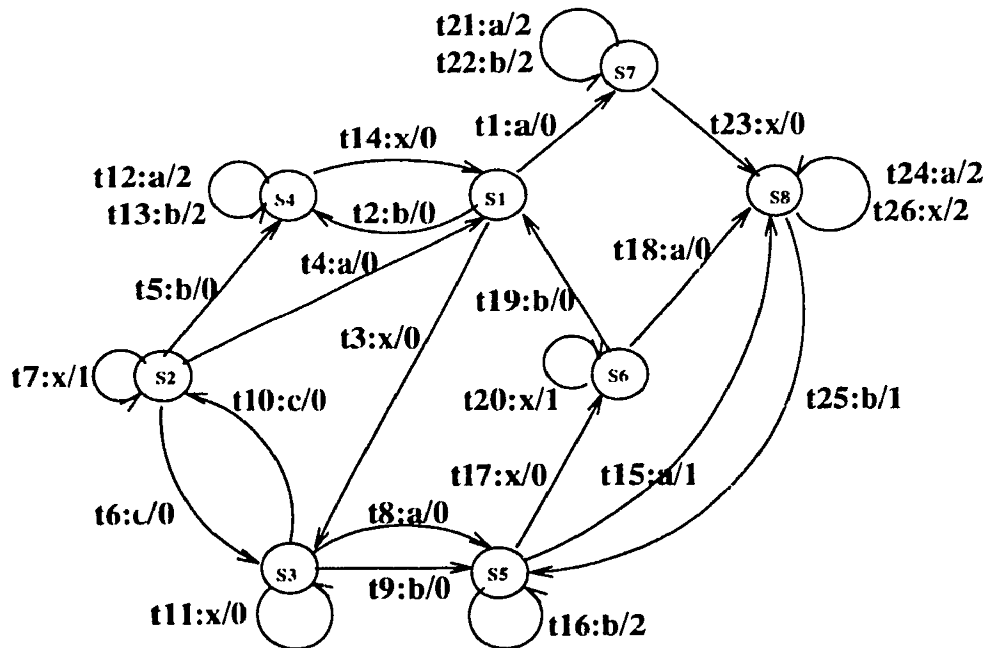
where $q_1 = 6$ and $q_2 = q_3 = 2$. $T(q_1, 2), T(q_2, 3)$, and $T(q_3, 4)$ are the paths $t19\ t3$, $t6$, and $t5$ respectively. Clearly, the input sequence is a prefix of the sequence generated in the state verification phase. The actual input sequence, the expected output sequence, and the output sequence observed on applying the input sequence to the IUT are given below.

Test subsequence	$= axbxx\ bx\ axbxx\ c\ axbxx\ b\ axbxx.$
Expected output sequence	= 00101 00 00000 0 00001 0 20000.
Observed output sequence	= 00101 00 00000 0 00001 0 00101.

Thus the IUT fails while verifying the distinguishing sequence at state s_4. However, the fault is at the transition $t3 = (s_1, s_2; x/0)$, which is a part of $T(q_1, 2), D(2), D(3)$



(a) SPEC



(b) IUT

Figure 3.1: An Example for the D-method

and $D(4)$. In general, if the IUT fails in the state verification phase, then the fault could be in any of the transitions traversed before the detection of the fault. Thus the D-method has the 1-fault resolution capability of level only $|E|$. On the other hand, suppose that the method ensures the following property:

> If an IUT passes the test sequence generated in the state verification phase of the D-method, then for each state s_i of the SPEC, the corresponding state in the IUT responds to X_0 in the same way as s_i.

Assume that the IUT fails in the second phase while testing the transition $t = (s_i, s_j; a/o)$ using the following walk.

$$TEST(s_i, s_j; a/o) = T(q_p, i-1)\ D(i-1)\ T(q_{i-1}, i)\ t\ D(j).$$

One of the transitions in $T(q_p, i-1)$ or in $D(p)$ is faulty if an unexpected output is observed from the IUT while applying the input sequence along the subwalk $T(q_p, i-1)\ D(i-1)$ of $TEST(s_i, s_j; a/o)$. On the other hand, if the first unexpected output $o' \neq o$ corresponds to the input a of the transition t, then the transition t has an output fault. However, if the first unexpected output corresponds to $D(j)$, then we can conclude that the transition t has a transfer fault. Since the length of $T(q_p, i-1)$ is at most $n-1$, we conclude that the fault can be diagnosed within l_d+n-1 transitions, where l_d is the length of the distinguishing sequence X_0. We summarize this result on the 1-fault resolution capability of the D-method in the following lemma.

Lemma 3.1 *Assume that the IUT has at most one fault. The D-method, in general, has 1-fault resolution capability of level only $|E|$. However, if the successful completion of the state verification phase on an IUT also implies that the response to the distinguishing sequence X_0 at each state s_i in the SPEC is the same as the response to X_0 in the corresponding state in the IUT, then the fault can be located within $l_d + n - 1$ transitions, where l_d is the length of X_0.*

□

A known theoretical upper bound for l_d is $(n-1)n^n$ [Koh78]. Actually, l_d would be much smaller for real life protocols. The method as such does not ensure that the state in the IUT corresponding to each state s_i responds to X_0 in the same way as s_i even if the IUT passes the state verification phase. This is due to the fact that a single transfer fault in a transition can permute the distinguishing sequence's response of one state into the other [KRK74].

3.1.2 Characterizing Sequences Method

The characterizing sequence method (henceforth referred to as the *C-method*) proposed by Kohavi *et al* [KRK74, Koh78] is a fault detection experiment for testing FSMs which may not have any distinguishing sequence. Only a brief description of the method is presented here. It is assumed that the protocol specification SPEC is strongly connected and minimal. The C-method assumes that the SPEC is completely specified. The C-method uses a set, called a characterizing sequence set for identifying the states. A nonempty finite set $\mathcal{C}$ of input sequences is called a characterizing sequence set of a protocol specification SPEC if no two states in SPEC have the same set of output sequences when all the sequences from $\mathcal{C}$ are applied to them. Formally, $\mathcal{C}$ is a **Chararcterizing Sequence set (CS set)** if for any two distinct states s_i and s_j in S, $\{Outseq(Walk(i,C) \mid C \in \mathcal{C}\} \neq \{Outseq(Walk(j,C) \mid C \in \mathcal{C}\}$. Each sequence in $\mathcal{C}$ is called a **Characterizing Sequence (CS)**. When $\mathcal{C}$ is a singleton, the unique characterizing sequence becomes a distinguishing sequence. A set $\mathcal{V}_i \subseteq \mathcal{C}$ is called an **Identifying Sequence set (IS set)** [Koh78] of the state s_i if $\mathcal{V}_i$ is a minimal subset of a CS set $\mathcal{C}$ such that $\{Outseq(Walk(i,V)|V \in \mathcal{V}_i\} \neq \{Outseq(Walk(j,V)|V \in \mathcal{V}_i\}$ for every state $s_j \neq s_i$.

We describe the method for $|\mathcal{C}| = 2$. Let $\mathcal{C} = \{C_1, C_2\}$. In order to identify the states an identifying sequence I_i for each state s_i is computed using its IS set $\mathcal{V}_i$. Suppose $\mathcal{V}_i = \{C_j\} (j = 1 \text{ or } 2)$ for a state s_i, where $1 \leq i \leq n$. Then $I_i = C_j$ is an identifying sequence for s_i and it is called an **identifying sequence of first**

order. On the other hand, if $\mathcal{V}_i = \{C_1, C_2\}$ then an **identifying sequence of second order** is computed as follows. As defined earlier, $T(i,j)$ is the transfer sequence which takes the SPEC from s_i to s_j. Let q_i and r_i denote the states of the SPEC after C_1 and C_2 respectively are applied at s_i. That is, $q_i = Tail(i, C_1)$ and $r_i = Tail(i, C_2)$, $i = 1, 2, \ldots, n$. Suppose β states respond to C_1 in the same way as s_i. Then the C-method uses the sequence $I_i = [C_1\ Inseq(T(q_i, i))]^{\beta+1} C_2$ as the identifying sequence for s_i [KRK74]. The reason for applying $\beta+1$-times the sequence $C_1\ Inseq(T(q_i, i))$ is to assure that the IUT is in the same state corresponding to s_i before an application of C_1 and the application of C_2. Let $C(i) = Walk(i, I_i)$ and $p_i = Tail(i, I_i)$, for $i = 1, 2, \ldots, n$. The following description of the C-method is applicable even if $|\mathcal{C}| > 2$ provided $\mathcal{V}_i$ is a singleton subset of $\mathcal{C}$ for all $i = 1, 2, \ldots, n$. In this case, the unique element in $\mathcal{V}_i$ becomes I_i for all $i = 1, 2, \ldots, n$. Similar to the D-method, this method can also be divided into two phases: state verification phase and the transition testing phase. The test sequence for the state verification phase corresponds to the walk given below.

$$C(1)\ T(p_1, 2)\ C(2)\ T(p_2, 3)\ C(3)\ \ldots\ C(n-1)\ T(p_{n-1}, n)\ C(n).$$

In the transition testing phase each transition of the SPEC is tested. The test sequence corresponding to a transition, say $(s_i, s_j; a/o)$ consists of (i) a sequence along a path required to put the IUT in state s_i and confirming it, (ii) the input symbol of the transition under test, and (iii) a sequence for identifying the tail state s_j. For more details on the C-method, [KRK74, Koh78] may be consulted.

We next analyze the fault coverage and the 1-fault resolution capability of the C-method. We claim that the C-method does not have complete fault coverage. We demonstrate this using the fictitious protocol SPEC and its implementation IUT shown in Figure 3.2. The same example has been used in [CVI89] for analyzing the fault coverage of the U-method. Let $C_1 = aa$ and $C_2 = ba$. Clearly, $\mathcal{C} = \{C_1, C_2\}$ is a CS set. Also, $I_1 = I_2 = C_1$ and $I_3 = C_2$. The test sequence for the state verification

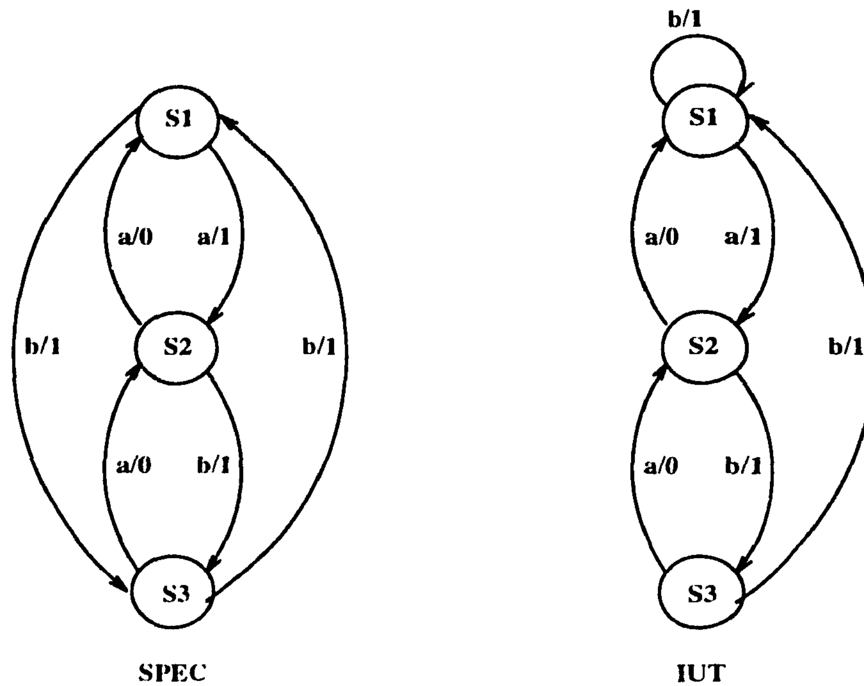


Figure 3.2: Protocol passed by the C-method

Test sequence: *aaaaabbaaaabbaaaaaaaaabbaaaaaaaaaaaabbabbabaaabbaaabbaa*
Expected output sequence: 1010111101011101010101111010101010101111111001111011110.

phase is *aaaaabba*. The corresponding expected output sequence is 10101111. Subsequences for testing the transitions $(s_1, s_2; a/1)$, $(s_2, s_3; b/1)$, $(s_2, s_1; a/0)$, $(s_1, s_3; b/1)$, $(s_3, s_2; a/0)$, and $(s_3, s_1; b/1)$ in that order are *aaabbaaaaa*, *aaaabba*, *aaaaaaa*, *aaaabba*, *bbabaaa*, and *bbaaabbaa*, respectively. Thus, the test sequence generated and the expected output sequence are given below.

The above test sequence passes the IUT given in Figure 3.2. However, the transition $(s_1, s_3; b/1)$ has a tail state fault in the IUT. Observe that though I_3 is an identifying sequence of s_3 in the SPEC, it is not an identifying sequence for s_3 in the IUT (responses for I_3 at both s_1 and s_3 in the IUT are identical).

Suppose that the IUTs of a SPEC are known to have at most one fault. The fact that an IUT fails while testing the transition $(s_i, s_j; a/o)$ using the test subsequence along $TEST(s_i, s_j; a/o)$ does not necessarily imply that a transition in

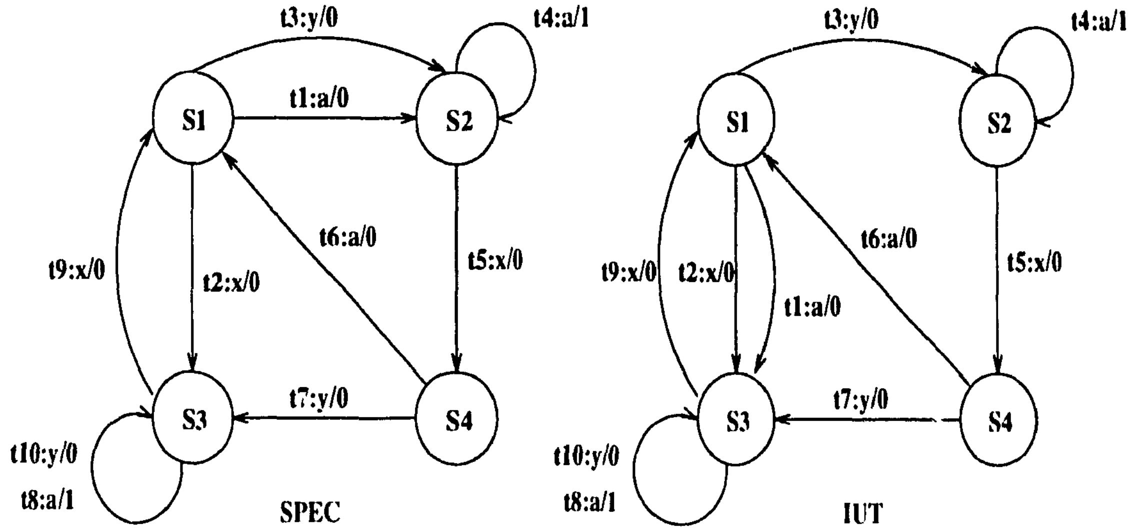


Figure 3.3: Illustration of the 1-fault resolution capability of the C-method

$TEST(s_i, s_j; a/o)$ is faulty. In fact the fault could be in any of the transitions traversed by the test sequence up to the point when the IUT failed. We illustrate this using the SPEC and its IUT as shown in Figure 3.3. Let $C = \{C_1, C_2, C_3\}$, where $C_1 = xa$, $C_2 = xay$, and $C_3 = ay$. Clearly, C is a CS set and $I_i = C_i$, for $1 \leq i \leq 3$ and $I_4 = C_3$. It is easy to see that $p_1 = p_3 = 3$ and $p_2 = p_4 = 2$. Let us take $T(p_1, 4) = t9\ t3\ t5, T(p_2, 4) = t5, T(p_3, 2) = t9\ t3$, and $T(p_4, 1) = t5\ t6$. The walk traversed for the state identification phase, test subsequence ITS, and the expected output sequence are given below.

$$\begin{aligned}
\text{Walk for ITS} &= C(1)\ C(3)\ T(p_3, 2)\ C(2)\ T(p_2, 4)\ C(4) \\
&\quad T(p_4, 1)\ C(1)\ T(p_1, 4)\ C(4)\ T(p_4, 1). \\
ITS &= xa\ ay\ xy\ xay\ x\ ay\ xa\ xa\ xyx\ ay\ xa. \\
\text{Expected output sequence} &= 01\ \ 10\ \ 00\ 000\ 0\ 00\ \ 00\ 01\ 000\ \ 00\ 00.
\end{aligned}$$

Let PTS be the concatenation of the subsequences along $TEST(s_1, s_2; a/0)$, $TEST(s_2, s_2; a/1)$, $TEST(s_2, s_4; x/0)$, and $TEST(s_4, s_1; a/0)$ in that order. These subsequences are defined in the following.

$$
\begin{aligned}
TEST(s_1, s_2; a/0) &= t1\ C(2). \\
\text{Input sequence} &= a\ xay. \\
\text{Expected output sequence} &= 0\ 000. \\
TEST(s_2, s_2; a/1) &= C(2)\ t4\ C(2). \\
\text{Input sequence} &= xay\ a\ xay. \\
\text{Expected output sequence} &= 000\ 1\ 000. \\
TEST(s_2, s_4; x/0) &= C(2)\ t5\ C(4). \\
\text{Input sequence} &= xay\ x\ ay. \\
\text{Expected output sequence} &= 000\ 0\ 00. \\
TEST(s_4, s_1; a/0) &= T(p_2, 4)\ C(4)\ C(2)\ T(p_2, 4)\ t6\ C(1). \\
\text{Input sequence} &= x\ ay\ xay\ x\ a\ xa. \\
\text{Expected output sequence} &= 0\ 00\ 000\ \ 0\ 0\ 01.
\end{aligned}
$$

Clearly PTS is a prefix of a test subsequence obtained in the transition testing phase. It can be verified that the IUT passes the test subsequence ITS. If we apply PTS to the IUT, after applying the ITS, the output observed from the IUT is the same as expected one but for the last input symbol a which lies in the test subsequence for $t6 = (s_4, s_1; a/0)$. When the last input symbol a is applied, the IUT gives the output 0 instead of the expected output 1. One could suspect that one of the transitions in $TEST(s_4, s_1; a/0)$ is faulty. However, the actual fault is at the transition $(s_1, s_2; a/0)$, which is tested at the initial part of PTS. From this we conclude that even if the C-method detects the fault, locating the fault using the test sequence generated by this method is difficult. Thus we conclude that the C-method has 1-fault resolution capability of level $|E|$.

In the following lemma we present our result on the 1-fault resolution capability of the C-method under a requirement on the state verification phase. We consider only the interesting case of all the identifying sequences being of order at most 2. Test sequences generated using the C-method for a SPEC having a CS set of cardinality more than two are in general very long and Kohavi suggests the use of some alternative methods [KL67, KRK74].

Lemma 3.2 *Assume that the IUT has at most one fault. The C-method, in general, has 1-fault resolution capability of level $|E|$. Suppose that the success of the state verification phase ensures that the identifying sequence I_i of each state in the SPEC is also an identifying sequence of the corresponding state in the IUT and that the fault is detected in the transition testing phase. Then the C-method diagnoses a fault to within $(\beta + 3)(l_c + n - 1) + 1$ transitions. Here, $\mathcal{C} = \{C_1, C_2\}$ is the CS set of the SPEC, β is the maximum number of states exhibiting the same response while applying C_1 at those states and l_c is the maximum length of a characterizing sequence in $\mathcal{C}$.*

Proof:

The first part of the lemma directly follows from the above discussion. Let W_{ij} denote $Walk(i, C_j)$, $1 \leq i \leq n, 1 \leq j \leq 2$. Suppose that the IUT fails in the second phase while testing the transition $t = (s_i, s_j, a/o)$ using the sequence along $TEST(s_i, s_j, a/o)$. Let s_k be the state of the IUT before applying this sequence. Let $t' = (s_g, s_h; b/o')$ be the last transition tested and s_f be the state of the IUT before applying the sequence $TEST(s_g, s_h; b/o')$. Let us assume that the identifying sequences for s_i, s_j, s_k, s_g, s_h and s_f are of second order. Clearly,

$$TEST(s_i, s_j, a/o) = C(k)\ T(p_k, i)\ W_{i1}\ T(q_i, k)\ C(k)\ T(p_k, i)\ W_{i2}\ T(r_i, k)$$
$$C(k)\ T(p_k, i)\ t\ W_{j1}\ T(q_j, k)\ C(k)\ T(p_k, i)\ t\ W_{j2}.$$

We know that $I_k = [C_1\ Inseq(T(q_k, k))]^{\beta+1}\ C_2$. If an unexpected output is observed in the IUT while applying the sequence along the first instance of $C(k)$ in the

above walk, then one of the transitions along the walk $C(f)\ T(p_f, g)\ t'\ W_{h2}$ is faulty. Note that $C(f)\ T(p_f, g)\ t'\ W_{h2}$ is a suffix of $TEST(s_g, s_h; b/o')$. As the maximum length of any identifying sequence and any transfer sequence are $(\beta+2)l_c+(\beta+1)(n-1)$ and $n-1$, respectively, the fault can be localized within $1+(\beta+3)l_c+(\beta+2)(n-1)$ transitions. However, if the first unexpected output corresponds to $T(p_k, i)\ W_{i1}$ of $TEST(s_i, s_j, a/o)$, then one of the transitions in $C(k)$ or in $T(p_k, i)$ is faulty. Thus, the fault is localized within $(\beta+2)(l_c+n-1)$ transitions. If the first mismatch between the expected output and the observed output is found while applying the first occurrence of $T(q_i, k)$ then it can be concluded that one of the transitions in $C(k)$, $T(p_k, i)$, W_{i1} and $T(q_i, k)$ is faulty. Proceeding this way, it is easy to see that if the first unexpected output is observed while applying the sequence $T(q_j, k)$, then the faulty transition is either $(s_i, s_j, a/o)$, or one among the transitions in $C(k)$, $T(p_k, i)$, W_{j1}, and $T(q_j, k)$. Thus the fault is localized within $(\beta+3)(l_c+n-1)+1$ transitions. Similarly in all the instances of first output mismatch while applying the sequence along $TEST(s_i, s_j, a/o)$, it can be shown that the fault is localized within $(\beta+3)(l_c+n-1)+1$ transitions.

□

3.1.3 W-method

In [Cho78] Chow proposed the W-method for testing the control structure of software designs modeled by FSMs. In recent years, this method has been widely applied for generating test sequences for protocol testing [SL89, FBK+91]. In this method, it is assumed that the protocol specification (SPEC) and its implementation (IUT) are strongly connected, minimal, completely specified and they accept the same input set, say I. The method also assumes that the SPEC has reset capability which is correctly implemented in the IUT. The IUT is allowed to have any number of states bounded by a finite estimate.

For state verification purposes the W-method uses a CS set of the IUT. It provides a scheme for obtaining a CS set of the IUT from a CS set of the SPEC. Given two sets of input sequences A and B, its concatenation AB is defined as $AB = \{a @ b \mid a \in A \wedge b \in B\}$. By A^i we denote the concatenation of A i times for any non-negative integer i. A^0 is the empty sequence ϵ. Let $I[j]$ denote the set of all possible input sequences of length at most j, for some integer j. Clearly $I[j] = \{\epsilon\} \cup I \cup I^2 \cup \ldots \cup I^j$ if $j \geq 0$. $I[j]$ is $\{\epsilon\}$ if $j < 0$. Let C be a CS set of the SPEC. Let n and n' denote the number of states in the SPEC and an upper bound on the number of states in the IUT, respectively. It is proved in [Cho78] that $\mathcal{W} = I[n' - n]\, C$ is a CS set for any fault free IUT. In order to confirm a state in the IUT, the W-method applies all the sequences from $\mathcal{W}$ at that state.

A directed spanning tree rooted at the initial state is used for reaching any state from the initial state. We refer to this tree as a **state cover tree**. Let P_i denote the unique path in a state cover tree T from the initial state to a given state s_i, for $i = 1, 2, \ldots, n$. Reset transitions are traversed for putting the SPEC or the IUT into the initial state from any given state. Thus, P_i, for $i = 1, 2, \ldots, n$ and r are the only transfer paths considered in this method. The set $TEST(s_i, s_j, a/o)$ of walks for testing a given transition $t = (s_i, s_j, a/o)$ is obtained using reset transitions, P_i, $(s_i, s_j, a/o)$, and $\mathcal{W}$. That is, $TEST(s_i, s_j, a/o) = \{r\ P_i\ t\ Walk(j, W') \mid W' \in \mathcal{W}\}$. The required test sequence is an arbitrary concatenation of the test subsequences for all the transitions in the SPEC.

The W-method assures complete fault coverage [Cho78]. Let us compare this method with the C-method. When $n' \leq n$, $\mathcal{W}$ is nothing but C. In this case the W-method applies every sequence in C at a given state s_i for identifying it in the IUT. However, the C-method only applies an IS set $\mathcal{V}_i$ - a subset of C - for identifying s_i in the IUT. This is the key difference between the C-method and the W-method. This difference and the reliable reset capability together make the W-method detect any fault in the IUT, while the C-method does not have such a capability.

Lemma 3.3 provides the 1-fault resolution capability of the W-method.

Lemma 3.3 *If an IUT has at most one fault, then the W-method always locates the fault to within $n + l_c$ transitions, where l_c is the length of a longest characterizing sequence generated in the W-method.*

Proof

Suppose the IUT fails while testing the transition $t = (s_i, s_j; a/o)$ using a sequence from $TEST(s_i, s_j; a/o)$. More specifically let $r\ P_i\ t\ Walk(j, C)$, for some $C \in \mathcal{C}$ be the sequence in $TEST(s_i, s_j; a/o)$ for which the IUT produces an output sequence different from the expected one. Since the IUT is known to be in the initial state after applying 'r', it follows that either the fault is at t, or in any of the transitions in the path P_i or in $Walk(j, C)$. Thus we conclude that the W-method has the 1-fault resolution capability of level $n + l$, if the IUT has at most n states.

□

The 1-fault resolution capability level of the W-method can be rewritten as $1 + d + l_c$ where d is the depth of the state cover tree. Therefore by minimizing the depth of the tree one can improve the 1-fault resolution capability. It is worth mentioning that the reliable reset capability assumption on the SPEC and the IUT has a significant role in diagnosing the fault to within $n + l_c$ transitions. It is known that every minimal machine with n states has a CS set $\mathcal{C}$ of cardinality at most $n - 1$, where the maximal length of a characterizing sequence in $\mathcal{C}$ is $n - 1$ [Koh78]. In such a case the level of 1-fault resolution capability reduces to $2n - 1$. Further localization of the fault could be achieved by applying additional test sequences to the IUT.

3.1.4 Transition Tour Method

The transition tour method (*T-method*, in short) [NT81] assumes that the SPEC is completely specified. The test sequence is generated based on a minimal transition

tour which traverses each transition in the SPEC at least once. Here the test subsequence for a transition is simply its input. This method generates a test sequence of the shortest length among all the methods discussed in this chapter. The T-method neither has a state verification phase nor does it verify the intermediate states in the IUT as it traverses the transitions. Hence the method does not have the capability of detecting transfer faults [DSU90b, SL89]. For the same reason this method cannot diagnose faults in the IUT with transfer faults even if it certifies the IUT as faulty. In the worst case, the 1-fault resolution capability level of the T-method is $|E|$, where $|E|$ is the number of transitions in the SPEC.

3.1.5 UIS-Based Methods

We first analyze the fault detection and diagnosis capabilities of the U-method described in Section 2.2.1. Although the fault coverage of this method is better than the T-method, it does not provide complete fault coverage [CVI89]. Let U_i be an UIS of state s_i, $1 \leq i \leq n$. Let $U(i)$ denote the walk from s_i with the input sequence U_i. That is, $U(i) = Walk(i, U_i)$. As defined in Section 2.1, the set $\mathcal{U} = \{U_1, U_2, \ldots, U_n\}$ is referred to as an UIS set. Note that it is also a CS set for the SPEC. Thus, this method is similar to the C-method. However, while the C-method confirms the starting state of the transition before testing the transition, the U-method does not. Also the U-method does not have a state verification phase. Hence we conclude that the fault coverage of this method cannot be better than the D-, C-, or W-methods. We observe that the fault resolution capability of this method is affected due to the following reasons. While applying the UISs , or transferring the IUT from one state to another state using transfer sequences, it might traverse transitions which have not yet been tested. As the method does not have a state verification phase, UIS of a state in the SPEC may not be an UIS of the corresponding state in the IUT.

For example, consider the SPEC and an IUT of an abstract protocol shown in Figure 3.1. The reset transitions are not shown explicitly in the figure.

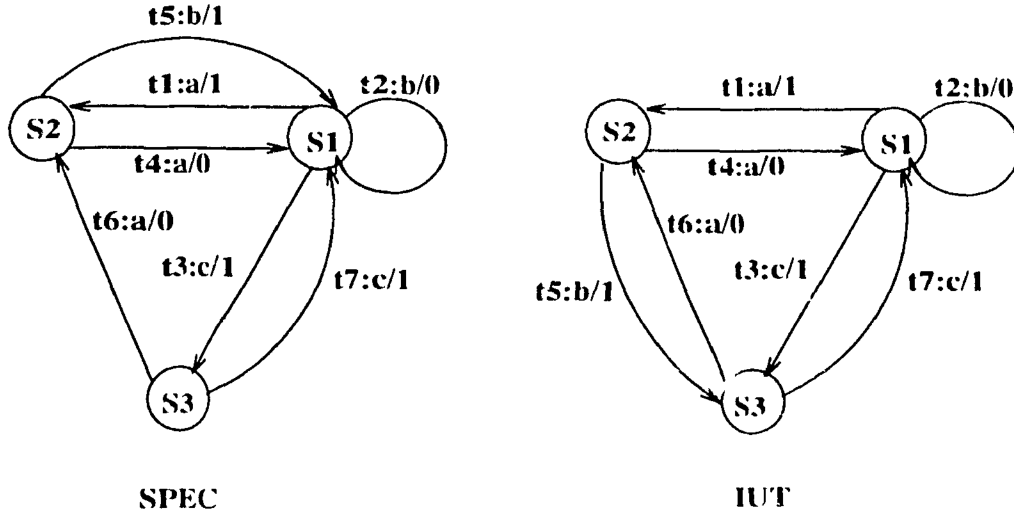


Figure 3.4: An Example for the U-method

The UISs of the states s_1, s_2, s_3 are $U_1 = b$, $U_2 = b$, and $U_3 = cb$ respectively. An optimal test sequence generated using the U-method is the concatenation of the following subsequences in that order (from left to right).

$\mathrm{TEST}(s_1, s_3; c/1) = t3 @ U(3)$	$\mathrm{TEST}(s_1, s_1; b/0) = t2 @ U(1)$
$\mathrm{TEST}(s_1, s_2, a/1) = t1 @ U(2)$	$\mathrm{TEST}(s_3, s_1, c/1) = \mathrm{T}(1,3) @ t7 @ U(1)$
$\mathrm{TEST}(s_2, s_1; b/1) = \mathrm{T}(1,2) @ t5 @ U(1)$	$\mathrm{TEST}(s_2, s_1, a/0) = \mathrm{T}(1,2) @ t4 @ U(1)$
$\mathrm{TEST}(s_3, s_2, a/0) = \mathrm{T}(1,3) @ t6 @ U(2)$	$\mathrm{TEST}(s_1, s_1; r/-) = r @ U(1)$
$\mathrm{TEST}(s_2, s_1; r/-) = \mathrm{T}(1,2) @ r @ U(1)$	$\mathrm{TEST}(s_3, s_1, r/-) = \mathrm{T}(1,3) @ r @ U(1)$
T(1, 1)	

Here, $\mathrm{T}(1,2) = t1$, $\mathrm{T}(1,3) = t3$, and $\mathrm{T}(1,1) = r$. The test sequence(TS) and the expected output sequence (OS) are given by

TS= *ccbbbabccbabbaabcabrbarbcrbr*

OS= 1100011110110100101-01-01-0-.

Though the test gives the fail verdict to the IUT as intended, it does not localize the fault. One could be misled to conclude that the fault is in the transition corresponding

to $(s_3, s_1; c/1)$ of the SPEC since the IUT fails while testing this transition with the sequence along TEST$(s_3, s_1; c/1)$. However, it is the transition $(s_2, s_1; b/1)$ of the SPEC that has a tail state fault in the IUT. Thus we conclude that the U-method has 1-fault resolution capability of level only $|E|$.

Since the traversal of transitions and UISs in the MU-method as well as our GU-method which are presented in Section 2.2.2 and Section 2.5.1, respectively, is similar to that of the U-method, the fault detection and the 1-fault diagnosis capability of these methods are similar to the U-method.

Improved UIS Method

As its name suggests, the improved UIS method (in short, the Uv-method) is an improvement over the original UIS-based method as described in [SD88]. The improvement is suggested by Chan *et al* [CVI89]. The Uv-method assumes that both the SPEC and the IUT are completely specified and strongly connected. Each state in the SPEC is assumed to have an UIS. Let $\mathcal{U} = \{U_1, U_2, \ldots, U_n\}$ be an UIS set of the SPEC, where U_i is an UIS of s_i, $1 \leq i \leq n$. It further assumes that the SPEC and the IUT have the reset capability. The method consists of two phases: UIS verification phase and transition testing phase. In the UIS verification phase, the method checks whether the selected UISs of the states of the SPEC are also UISs of the corresponding states in the IUT. This could be done by verifying the output sequence produced by the IUT for every UIS in $\mathcal{U}$ when the IUT is in s_i, for $i = 1, 2, \ldots n$. Though Chan *et al* have suggested the need for verifying the UISs in the IUT, they do not provide any method for achieving this requirement. In the transition testing phase, all transitions are tested as in the UIS-based method as described in [SD88]. Here, $TEST(s_i, s_j; a/o) = r\ P(i)\ t\ U(j)$ is the subsequence for testing the transition $(s_i, s_j; a/o)$, where $P(i)$ is a shortest path from s_1 to s_i and $U(j)$ is the walk from s_j with input sequence U_j.

The Uv-method is known to have complete fault coverage [CVI89]. In the

following theorem we analyze the 1-fault resolution capability of this method.

Lemma 3.4 *Suppose that the IUTs have at most one fault. Then the Uv-method has 1-fault resolution capability of level $n + l_u$, where n is the number of states in the SPEC and l_u is the length of a longest sequence in the UIS set $\{U_1, U_2, \ldots, U_n\}$ used by the method. Here U_i is the UIS of the state s_i, $i = 1, 2, \ldots, n$. Suppose that the successful completion of the first phase ensures that U_i is also an UIS of the state in the IUT corresponding to s_i for each state s_i in the SPEC, then the Uv-method locates the fault exactly.*

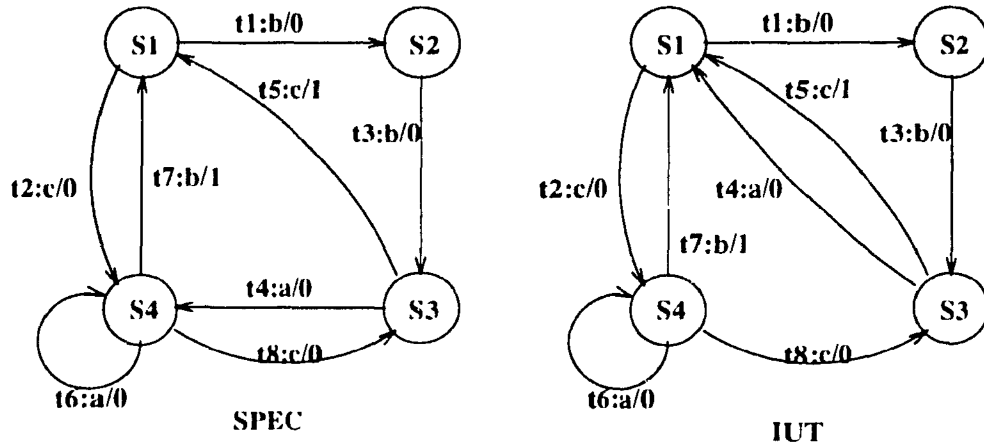
Proof:

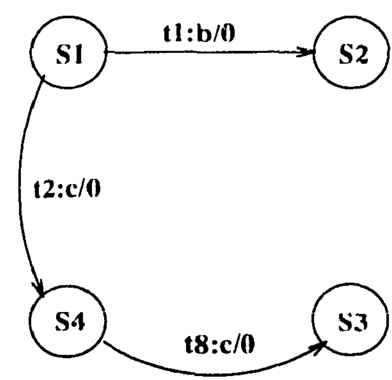
The proof of the first part of this lemma is omitted since it is similar to that of Lemma 3.3.

We now show that the Uv-method has the capability of locating the fault exactly if the successful completion of the first phase implies that U_i is also an UIS of the state in the IUT corresponding to s_i for each state s_i in the SPEC. Assume that the second phase fails while testing tranisiton $t = (s_i, s_j; a/o)$ with the sequence $r\ P(i)\ t\ U(j)$. Here $P(i)$ is a shortest path from s_1 to s_i. Note that $P(i)$ is used in the first phase while verifying if U_i is an UIS of the state in the IUT which is reached when input sequence along $P(i)$ is applied to the IUT at its initial state. Recall that $U(j)$ is the walk at s_j with the input sequence U_j. We claim that $P(i)$ puts the IUT at the state corresponding to s_i. For, if $P(i)$ puts the IUT in some state which does not correspond to s_i then, as per the first phase and our assumption, U_i is an UIS of a state which does not correspond to s_i. This is a contradiction. It follows that t is the unique faulty transition.

□

It is known that the Uv-method has complete fault coverage. We observe that the complete fault coverage property cannot be guaranteed if one uses the rural posterson optimization technique as described in [ADLU88] in the transition testing phase





State cover tree for the SPEC

Figure 3.5: Protocol passed by the Uv-method

of the Uv-method. Consider the SPEC and an IUT of an abstract protocol shown in Figure 3.5. The reset transitions are not shown explicitly in the figure. The transition $(s_3, s_4; a/0)$ has a transfer fault in the IUT. Observe that for both the SPEC and its IUT, $U_1 = cb, U_2 = ba, U_3 = c$, and $U_4 = b$ are UISs of the states s_1, s_2, s_3, and s_4, respectively. If the paths along the state cover tree T shown in Figure 3.5 and the reset transitions only are used as transfer sequences in the UIS verification phase, then the IUT passes this phase. The test sequence generated in the transition testing phase is the concatenation of the test subsequences for the transitions and the transfer sequences, as given below.

$TEST(s_1, s_2; b/0) @ T(1,3) @ TEST(s_3, s_4; a/0) @ T(1,3) @ TEST(s_3, s_1; c/1) @$
$T(1,4) @ TEST(s_4, s_1; b/1) @ T(1,4) @ TEST(s_4, s_4; a/0) @ T(1,4) @$
$TEST(s_4, s_3; c/0) @ TEST(s_1, s_4; c/0) @ T(1,2) @ TEST(s_2, s_3; b/0)$.

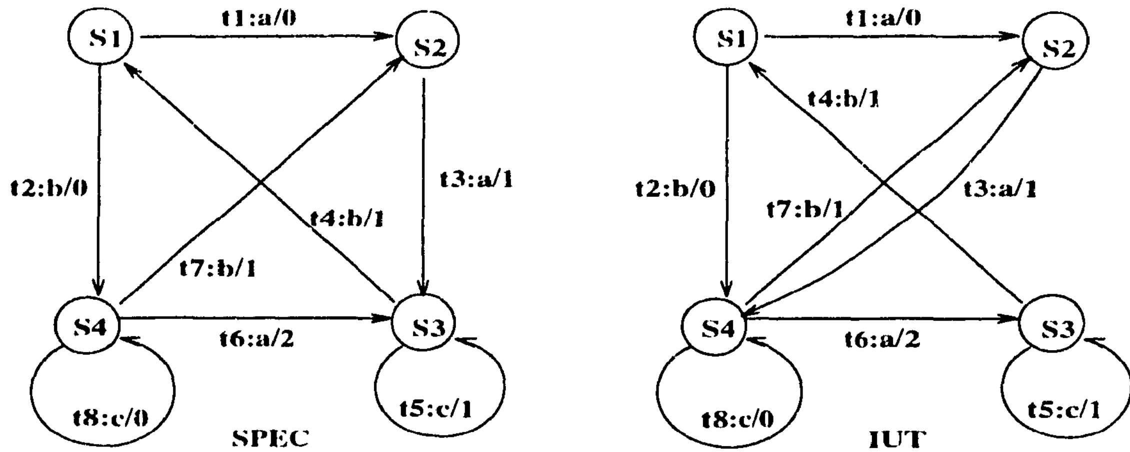
Here, $T(1,4) = t2, T(1,3) = t8, T(1,3) = t2\ t8$, and $T(1,2) = t1$. The resulting test sequence (TS) which passes the faulty IUT and the expected output sequence (OS) are given below.

TS=*bbacabccccbcbcbcabccccbbbc*.

OS=00000100101010100100101001.

The Uv-method is similar to the C-method. However, in the first phase, while the C-method confirms the state by applying sequences from the IS set of each state at that state, the Uv-method applies all the UISs from an UIS set. Unlike the C-method the Uv-method assumes that the SPEC and the IUT have the reset capability.

In the following we demonstrate that the 1-fault resolution capability of the Uv-method will be affected if one uses the rural postperson optimization technique in the transition testing phase. As in the U-method, the Uv-method may use some transitions for putting the IUT in the starting state of the transition under test. Such



(reset transitions are not shown explicitly)

Figure 3.6: An Example for the Uv-method

transitions constitute a preamble for the transition under test. This preamble may contain faulty transitions which are yet to be tested. Also, some of the transitions which constitute the UIS of the tail state of the transition under test may be faulty. In other words, even if the IUT passes the first phase, the state of the SPEC after applying an UIS, say, U_i at state s_i need not correspond to the state of the IUT after applying U_i at the state of the IUT corresponding to s_i. Therefore, if one uses the rural postperson optimization technique in the transition testing phase of the Uv method then its level of 1-fault resolution capability becomes $|E|$. For example, consider the SPEC and the IUT of an abstract protocol shown in Figure 3.6. The reset transitions are not shown explicitly in the figure. $U_1 = a$, $U_2 = a$, $U_3 = c$, and $U_4 = c$ are the UISs of s_1, s_2, s_3, and s_4 in the SPEC as well as the IUT.

Observe that the UIS of each state in the SPEC is also an UIS of the corresponding state in the IUT. It can be easily seen that if one uses the transitions $(s_1, s_2; a/0), (s_1, s_4; b/0)$, and $(s_4, s_3; a/2)$ and reset transitions to reach different states for verifying the UIS, then the IUT passes the UIS verification phase successfully. It is a simple exercise to check that an optimal test sequence generated in transition

testing phase is the concatenation of the following subsequences in that order (from left to right). Note that $U(i) = Walk(i, U_i)$, $1 \leq i \leq 4$.

$\text{TEST}(s_1, s_4; b/0) = t2@U(4)$	$\text{TEST}(s_4, s_1, c/0) = t8@U(4)$
$\text{TEST}(s_4, s_3; a/2) = t6@U(3)$	$\text{TEST}(s_3, s_3, c/1) = t5@U(3)$
$\text{TEST}(s_1, s_2; a/0) = \text{T}(3,1)@t1@U(2)$	$\text{TEST}(s_3, s_1; b/1) = t4@U(1)$
$\text{TEST}(s_2, s_3; a/1) = t3@U(3)$	$\text{TEST}(s_4, s_2; b/1) = \text{T}(3,4)@t7@U(2)$

T(3, 1).

Here, T(3, 1) = $t4$, and T(3, 4) = $t4$ $t2$. The resulting test sequence (TS) and the expected output sequence (OS) are given below

TS=$bcccacccbaabaacbbbab$.

OS=00002111101101110111.

Though the IUT fails while testing the transition corresponding to $t4 = (s_3, s_1; b/1)$, this transition is fault-free in the IUT. Actually, the transition $t3 = (s_2, s_3; a/1)$ of the SPEC has a transfer fault in the IUT. In order to make the illustration simple the test subsequences corresponding to the reset transitions are omitted. We would like to note that one can modify the SPEC and the IUT to be completely specified using the completeness assumption and the above result is still valid.

3.1.6 Wp-method

The Wp-method introduced by Fujiwara *et al* [FBK+91] is based on the W-method. The Wp-method assumes that the SPEC is strongly connected, minimal, and completely specified. The SPEC and the IUT have reset capability and same input set. It further assumes a finite upper bound, say, n' on the number of states of the IUT. Let $\mathcal{C}$ be a CS set of the SPEC. Let $\mathcal{V}_i \subseteq \mathcal{C}$ be an IS set of the state s_i in the SPEC for each state s_i. As discussed in the W-method, $\mathcal{W} = I[n' - n]\ \mathcal{C}$ is a CS set of correct implementations which have at most n' states. Clearly, in such correct implementations, $\mathcal{W}_i = I[n' - n]\ \mathcal{V}_i$ is the IS set of the states corresponding to each state s_i of the SPEC.

The Wp-method consists of two phases: state verification phase and transition testing phase. The first phase is mainly to verify whether W is a CS set of the IUT. This is done by applying every sequence in W at each state in the IUT. Paths in a state cover tree, say T, and the reset transitions are used for putting the IUT in each state for applying sequences from W. As a result, all the transitions in the state cover tree T are also tested by the end of this phase.

It is claimed in [FBK+91] that if an IUT passes the state verification phase, then

C1: all transitions in T are implemented correctly in the IUT.

C2: W_i is an IS set for the states in the IUT corresponding to the state s_i of the SPEC.

In the transition testing phase all the transitions of the SPEC which are not in the state cover tree T are tested. Transitions in T are not tested here, as it is done in the first phase itself. Reset transitions and paths in a state cover tree, say T, are used for putting the IUT in the starting state of the transition under test. In order to confirm the tail state of a given transition, say, $(s_i, s_j; a/o) \in E - T$ in the IUT, each sequence from the IS set W_j is applied at the tail state. Here, E denotes the set of all transitions in the SPEC.

Clearly, the Wp-method is an improvement over the W-method. Note that the aim of the first phase of the Wp-method is to assure that W_j is an IS set of the state in the IUT corresponding to s_j, for $j = 1, 2, \ldots n$. As a result in the second phase the tail state of the transition $t = (s_i, s_j; a/o)$ in the IUT is confirmed by applying the sequences from W_j in the current state after traversing t, instead of applying the whole CS set W. This is the main difference between the W-method and the Wp-method. Thus the length of the test sequence generated by the Wp-method is always less than or equal to the one generated by the W-method. The Wp-method also assures complete fault coverage [FBK+91].

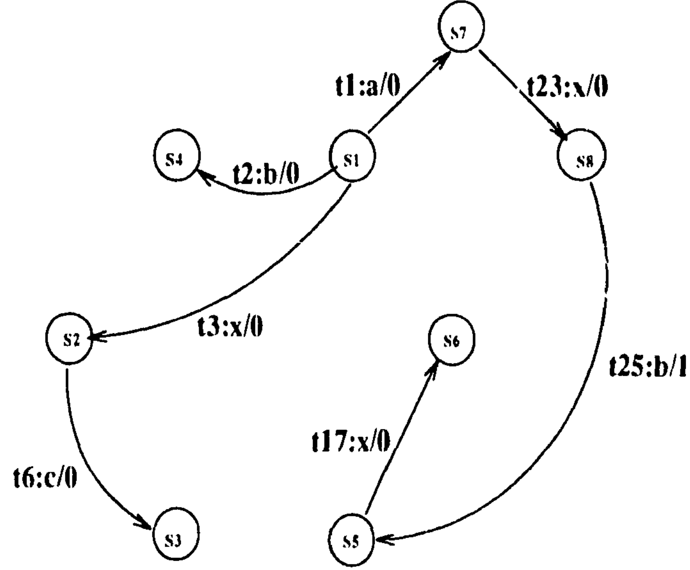


Figure 3.7: State cover tree of the SPEC given in Figure 3.1(a)

We now analyze the 1-fault resolution capability of this method. We observe that claims C1 and C2 need not always be true for IUTs passing the state verification phase successfully. We demonstrate this on an abstract protocol whose SPEC and IUT are given in Figure 3.1. Reset transitions are not shown explicitly in this figure. Here, the SPEC and the IUT have the same number of states, i.e., $n' = n$. Clearly, $C = \{axbxx\}$ is a CS set of the SPEC. Since $m = n$, we have $W = C$, and $W_i = W$, for $1 \leq i \leq 8$. The state cover tree T used in testing is shown in Figure 3.7. With state cover tree T and the CS set W the IUT passes the state verification phase. In the transition testing phase, transition $t9 = (s_3, s_5; b/0)$ is tested using the input sequence along $TEST(s_3, s_5; b/0)$. This test subsequence, its expected output and the output given by the IUT are given below:

$TEST(s_3, s_5; b/0)$	$= r\ P_3\ t9\ Walk(5, axbxx)$
Input sequence	$= r\ x\ c\ b\ a\ x\ b\ x\ x$
Expected output sequence	= - 0 0 0 1 2 1 0 1
Observed output sequence	= - 0 0 0 2 0 0 0 0

Using the claim C1 one could conclude that the transition $t9 = (s_3, s_5; b/0)$ has

a transfer fault, and its faulty tail state in the IUT is s_4. However, this transition is fault-free in the IUT. The actual fault is at the transition $t3 = (s_1, s_2; x/0)$. Though the IUT passes the first phase, the state cover tree transition $(s_1, s_2; x/0)$ has a transfer fault in the IUT. Also, though $W_2 = W_3 = W$ is an IS set for states s_2 and s_3 in the IUT, the output sequences obtained while applying W on these states are, respectively, different from the output sequences obtained while applying W at s_2 and s_3 of the SPEC. That is, the claim C2 is not valid for this IUT. In fact the responses for W at s_2 and s_3 of the IUT is the permutation of the respective responses in the SPEC. The level of 1-fault resolution capability of the Wp-method is presented in the following lemma.

Lemma 3.5 *Suppose that the IUTs have at most one fault. Then the Wp-method has 1-fault resolution capability of level $n + l_c$, where n is the number of states in the SPEC and l_c is the length of a longest sequence in $\mathcal{C}(= \mathcal{W})$. Suppose that the successful completion of the first phase ensures that the IS set $\mathcal{V}_i(= \mathcal{W}_i)$ is also an IS set of the state in the IUT corresponding to s_i for each state s_i in the SPEC, then the Wp-method locates the fault exactly.*

Proof:

The proof of the first part of this lemma is similar to that of Lemma 3.3. Suppose that the IUT fails in the first phase while verifying the CS set $\mathcal{W}$ at the state, say s_i. More specifically, if the output sequence corresponding to the input sequence along the walk $r\ P_i\ Walk(i, W)$ for some $W \in \mathcal{W}$ is not the same as the expected one, then the fault can be in any of the transitions in $P_i\ Walk(i, W)$.

On the other hand, if the IUT fails in the second phase, while testing a transition, say, $t = (s_i, s_j; a/o)$, with a sequence, say, $r\ P_i\ t\ Walk(j, W)$ for some $W \in \mathcal{W}_j$, then any of the transitions in $P_i\ t\ Walk(j, W)$ could be faulty.

In both the cases, the set of such transitions within which the faulty one lies has cardinality at most $n + l_c$. Thus the Wp-method has 1-fault resolution capability of level $n + l_c$.

Suppose that the successful completion of the first phase implies that the IS set $\mathcal{V}_i (= \mathcal{W}_i)$ is also an IS set of the state in the IUT corresponding to s_i for each state s_i in the SPEC. It is easy to prove that all the transitions in the state cover tree T are also fault free provided each state s_i in the SPEC is selected for applying C in a breadth first fashion along T. Suppose phase 2 fails while testing the transition, say, $t = (s_i, s_j; a/o)$, with the sequence along $r\ P_i\ t\ Walk(t, W)$ for some $W \in \mathcal{W}_j$. As the transitions along P_i are known to be correctly implemented, there would not be any mismatch if t is fault free. Therefore t is the unique faulty transition.

□

Although the Wp-method, in general, provides a shorter test sequence than the W-method, its fault resolution capability is the same as that of the W-method. The 1-fault resolution capability of the Wp-method can be improved by minimizing the depth of the state cover tree. This is also the case with the W-method. As noted in the W-method, if one considers a CS set C of cardinality $n-1$ in which each characterizing sequence is of length at most $n-1$, it can be deduced that Wp-method has 1-fault resolution capability of level $2n-1$. Moreover, the fault can be located exactly if the success of the first phase implies that the IS set of a state in the SPEC is also an IS set of the corresponding state of the IUT and if the IUT fails in the second phase. The Wp-method is improved in Chapter 4 so that the two properties (Claims C1 and C2) claimed in [FBK+91] for this method hold on the successful completion of the first phase of the proposed methods.

3.2 Comparison of Test Sequence Generation Methods

In this section we compare test sequence generation methods based on their fault detection and diagnosis capabilities, and the length of the test sequences they generate.

3.2.1 Fault Coverage and Diagnosis

Among all the methods discussed in Section 3.1, only the D-, the W-, the Uv, and the Wp- methods provide complete fault coverage. Since the T-method does not verify the intermediate states as it traverses the transitions, the fault coverage of this method is less than that of all state identification based methods, those which use state identification sequences. Among the state identification based methods, the fault coverages of the C- and the U-methods are less than that of the state verification based methods, namely the D-, the W-, the Uv- and the Wp- methods. Comparing the C- and U- methods, the C-method has at least the same fault coverage as the U-method, since the C-method checks the tail states of transfer sequences while the U-method does not. A partial ordering among the fault coverages of the test sequence generation methods is summarized below. Here, $F(X)$ denotes the fault coverage of the method X.

$$F(T) < F(U) \leq F(C) < F(D) = F(W) = F(Uv) = F(Wp)$$

In general, the W-, the Uv- and the Wp-methods have the best 1-fault resolution capabilities among the known methods. Consider the D-, the Uv- and the Wp-methods. They have a separate state verification phase. Suppose that these methods somehow guarantee that the sequences used for identifying the states are also the identifying sequences of the corresponding states in the IUT on successful completion of the state verification phase. From Lemma 3.1, 3.4 and 3.5, we know that (i) the D-method localizes the fault within $l_d + n - 1$ and (ii) the Uv- and the Wp-methods localizes the fault exactly. Here, l_d is the length of the distinguishing sequence used in the D-method. The known upper bound for l_d is $(n-1)n^n$. Therefore, if $FRL(X)$ denote the worst case bound on the level of 1-fault resolution capability of the method X under the condition stated above, where X is one of D-, Uv-, and Wp- methods, then

$$FRL(D) > FRL(Uv) = FRL(Wp).$$

3.2.2 Length of Test Sequences Generated

In general, the lengths of test sequences generated using the T-method are always less than those of the sequences generated using state identification based methods. The test sequences are even longer if the sequences used for identifying the states are also verified in the IUT. In order to compare the lengths, we assume that the same UIS set is used in both the U and Uv methods. The CS set for each of C, W and Wp methods is the UIS set selected for the U-method. It is also assumed that the number of states of the IUT is no more than the the number of states of the SPEC. Suppose $L(X)$ denote the length of a test sequence generated in method X. Then, the order among the lengths of test sequences generated using the various methods is as given below.

$$L(T) < L(U) \leq L(Uv),\ L(U) \leq L(Wp) \leq L(W), and\ L(U) \leq L(C).$$

As noted in [Ura92], the upper bound on the length of the test sequences generated in the D-method is the longest among all the methods discussed in this chapter. However, in practice it is the least among the sequences generated using the W-, Wp-, Uv-, and C-methods.

3.3 Summary

In this chapter, we have surveyed different methods for generating test sequences for testing communication protocols based on the FSM model. Fault coverage of each method, if available in the literature, is also reviewed. We have demonstrated that complete fault coverage cannot be guaranteed if one applies the rural postperson optimization technique in the transition testing phase of the Uv-method. We have also shown that the C-method does not have complete fault coverage. With a single fault assumption, all the FSM based methods are formally analyzed to see if they can be used for diagnosing the fault in an implementation. The levels of 1-fault resolution

capabilities derived for these methods demonstrate that the W-, the Uv- and the Wp-methods are the best ones for diagnosing the fault to within a few transitions. A comparison of the test sequence generation methods is made based on the fault coverage, 1-fault diagnosis capability and the length of test sequences they generate.

In all the test sequence generation methods discussed in this chapter, the test subsequence for testing a transition is, in general, the concatenation of a preamble, input of the transition under test, state identification sequence of the tail state of the transition under test, and a postamble. For the purpose of optimization, in some of the methods certain subsequences may be overlapped or omitted . Thus a test sequence generation method will have better fault resolution capability if the method confirms the correctness of the following transitions prior to the testing of a given transition:

(i) Transitions in the preamble of the transition under test.

(ii) Transitions along the sequence for identifying the tail state of the transition under test.

(iii) Transitions in the postamble of the transition under test.

As we have pointed out, test sequence generation methods considered in this chapter meet this requirement only partially. Attempts to meet these requirements lead to new test generation methods with improved 1-fault resolution capabilities. These methods are discussed in the following chapter.

The work presented in this chapter has been reported in [RDT93, RDTa].

Chapter 4

FAULT DIAGNOSIS METHODS

As pointed out in Chapter 3, some of the desirable properties for fault diagnosis do not always hold for the existing test case generation methods. In this chapter, we propose two test sequence generation methods for diagnosing faults with respect to the fault model defined in Section 1.2.1. The specification and an implementation are represented as FSMs, denoted by SPEC and IUT, respectively, which accept the same set of inputs. We assume that the SPEC is strongly connected and minimal. The completeness assumption is used for treating the given SPEC and the IUT as being completely specified. We further assume that the SPEC has a cyclic UIS - an UIS for a state whose corresponding walk at the state is a cycle - for the initial state which is also a cyclic UIS for the initial state of the IUT. In our opinion, this is easier to achieve in practice than the usual reset capability. For example, the SPEC and the IUT have a self-loop transition with "state/initial" label for their initial state only. We assume that the IUT has at most one output or transfer fault and that the IUT is in the initial state before the testing commences. Note that the number of states in the IUT is the same as the number of states in the SPEC.

Both our diagnosis methods are based on the Wp-method [FBK+91]. Our first method is called the **UIDD-method** and it uses an UIS set (refer to Section 2.1) for identifying the tail states of transitions. This method provides superior 1-fault

diagnosis capability than all the UIS-based test generation methods analyzed in Chapter 3. Also by incorporating the rural postperson optimization technique of Aho *et al* [ADLU88] and the heuristic algorithms for the generalized RPP developed in Section 2.4, the UIDD-method minimizes the length of the test sequence generated. This method requires an UIS set and a state cover tree satisfying the **Tree UIS Set Disjoint (TUISD) property** (defined later). It should be noted that there may exist some protocols which may not have an UIS set and a state cover tree with the required property. As reported in Section 4.6, all the real life protocols we have analyzed so far do satisfy this requirement.

Our second diagnosis method, called the CSDD-method, uses a CS set as defined in Section 3.1.2, instead of an UIS set as in the first method. While there may exist some minimum SPEC's without an UIS set, all such SPEC's do have a CS set. This method provides better 1-fault resolution capability than the Wp-method by carefully selecting the CS set and the state cover tree.

Our UIDD-method is presented in Section 4.1. In Section 4.2, we present an algorithm for computing an UIS set and a state cover tree with the TUISD property. This method is illustrated in Section 4.3. The CSDD-method is described in Section 4.4. An approach for exactly locating the fault or improving the fault resolution capability of the above methods is presented in Section 4.5. Unless otherwise stated, a tour in this chapter usually refers to one starting and ending at the initial state.

4.1 UIDD-method

Apart from the assumptions stated above, in the UIDD-method, we assume that the SPEC has an UIS set. An UIS set $\mathcal{U}$ is used for identifying the states. Let $\mathcal{U} = \{U_1, U_2, \ldots, U_n\}$. Here U_1 is a cyclic UIS of the initial states of the SPEC and the IUT, and U_j is an UIS of the state s_j of the SPEC, where $2 \leq j \leq n$. Recall that **Walk(j, inseq)** denotes the walk taken by the SPEC when the input sequence

inseq is applied at the state s_j. A transition is called a $\mathcal{U}$**-transition** if it belongs to $Walk(j, U_j)$ for some $j, 1 \leq j \leq n$. Let $E(U)$ denote the set of all $\mathcal{U}$-transitions. The UIDD-method uses a state cover tree T to reach different states from the initial state. A transition is called a **T-transition** if it is on the state cover tree T. Let $E(T)$ denote the set of all T-transitions. As noted in the previous chapter, for better fault resolution it is necessary that we test all the T-transitions before using them for testing other transitions. It is also necessary to verify if the UISs are also the UISs of the corresponding states in the IUT. Meeting these two requirements is difficult as T-transition testing requires verified UISs and UIS verification requires tested T-transitions. This could be achieved if T and $\mathcal{U}$ have a special property known as the **Tree UIS Set Disjoint (TUISD) property** defined below.

> For each transition $t = (s_i, s_j: a/o)$ in T, the label a/o does not occur in the UIO-sequence corresponding to the UIS U_j or the walk from s_k with the input sequence U_j does not contain t, for all $k, 1 \leq k \leq n$ and $k \neq j$.

We have analyzed a number of protocols (reported later) and found that they have a T and an $\mathcal{U}$ with the TUISD property. However, there may exist some protocols which do not have such T and $\mathcal{U}$.

The UIDD-method is described in the algorithm UI_DD. This algorithm first invokes the procedure *set_uis* (to be discussed) for computing a state cover tree T and an UIS set $\mathcal{U}$ with the TUISD property. The algorithm then invokes the procedure *UIgen_seq* for generating the required test sequence. In this section, unless specifically mentioned, by an UIS we mean an UIS from $\mathcal{U}$. Let P_i denote the unique path in T from s_1 to s_i, for $1 \leq i \leq n$. Let $\mathcal{Q} = \{Q_i \mid 1 \leq i \leq n\}$, where Q_i denotes a shortest path in the FSM from s_i to s_1, for $1 \leq i \leq n$. P_i and Q_i are also known as the **preamble** and the **postamble** for the state s_i, respectively, for $1 \leq i \leq n$. Let E_q be the set of edges which denote the postambles in $\mathcal{Q}$. That is, $E_q = \{(s_i, s_1: label(Q_i)) | Q_i \in \mathcal{Q}\}$.

The procedure *UIgen_seq* consists of two phases. The purpose of the first phase is to test all the transitions in T as well as to verify if the UISs are actually UISs of the corresponding state in the IUT. Let $\mathcal{U}' \subseteq \mathcal{U}$ be obtained from $\mathcal{U}$ by removing all the duplicate UISs and those UISs which are prefixes of other sequences in $\mathcal{U}$. Each transition in T is selected for testing in a breadth-first fashion. Test sequence for $t = (s_i, s_j; a) \in T$ consists of a set of tours, one tour for each UIS $U \in \mathcal{U}'$. The tour with respect to U is obtained by concatenating $P_i.t.\ Walk(j, U).\ Q_{Tail(j,U)}$, and $Walk(1, U_1)$ in that order. Recall that the function **Tail(j, inseq)** accepts the index of the state s_j, $1 \leq j \leq n$ and an input sequence *inseq* and returns the index of the tail state of $Walk(j, inseq)$. Observe that the cyclic UIS U_1 is used for confirming the initial state. It is established later in this section (Theorem 4.1) that the test sequence generated in the above scheme is also sufficient for verifying if the UIS of any state in the SPEC is also an UIS of the corresponding state in the IUT.

The second phase consists of six steps. In the first step, we generate a tour for testing the correctness of the set of postambles ($\mathcal{Q}$) in the IUT. Step 2 through Step 3 and Step 4 through Step 5 are two alternate ways of testing all the non-T-transitions. In the first alternative, in Step 2 each of the $\mathcal{U}$-transitions is tested individually using a tour consisting of the preamble to reach the transition, the transition, the UIS of its tail state and the postamble from the tail state of the UIS. Transitions which are neither in T nor in the UISs are tested in Step 3. Let E_c be the set of test edges for the set of all non-T- non-$\mathcal{U}$-transitions. As defined in Section 2.1, a test edge for a transition $t = (s_i, s_j; a/o)$ is an edge from s_i to $Tail(j, U_j)$ with label $a@U_j$. Let $G'' = (S, E_c \cup E(T) \cup E(U) \cup E_q)$. If possible, the RPT optimization technique of Aho *et al* [ADLU88] is used for computing an RPT T4 of G'' with respect to E_c. Clearly T4 tests all the transitions in E_c. Otherwise, this step computes three approximate tours (T5, T6 and T7) using the three heuristic algorithms for the RPP developed in Section 2.4. It also computes another tour (T8) in which each of the transitions is tested exactly similar to the $\mathcal{U}$-transitions in Step 2. Among T5, T6, T7 and T8, the

one with the minimum length is then chosen as the tour T4. Note that T4 may not start and end at the initial state of the SPEC. Therefore, T4 is augmented with the preamble and the postamble of a state in T4 so that T4 starts and ends at the initial state of the SPEC.

In the second alternative for testing the non-T-transitions, the tail states of the UISs are first confirmed in Step 4. This is done by reaching each state (s_i) using the preamble and then applying the UIS U_j followed by the UIS of $Tail(j, U_j)$. The initial state is reached by traversing an appropriate postamble. A tour(T10) for testing all the non-T-transitions is derived in Step 5. Step 5 is similar to Step 3. While the set E_c in Step 3 contains an edge for each of the non-T non-$\mathcal{U}$-transitions only, the same set in Step 5 contains an edge for each of the non-T-transitions. Clearly, either of the tours T3@T4 and T9@T10 can be used for testing the non-T-transitions. We choose the one with the minimum length. In the algorithm, $T15$ denotes this tour. The tours T1, T2 and T15 are used in Step 6 to generate the final test sequence.

Algorithm *U_DD*(SPEC)
  set_uis(SPEC,T,$\mathcal{U}$);
  Ugen_seq(SPEC,T,$\mathcal{U}$);
end *U_DD*

procedure *Ugen_seq*(SPEC,T,$\mathcal{U}$)
Phase I {UIS verification and T-transition testing}
  Obtain $\mathcal{U}'$ from $\mathcal{U}$ by deleting duplicate sequences and
  removing sequences which are prefixes of other sequences;
  T1.= $\emptyset$;
  for each $U \in \mathcal{U}'$ **do**
    T1 := T1@Walk(1,U)@$Q_{Tail(1,U)}$@Walk(1,U_1).
  for each $t = (s_i, s_j : a/o) \in T$ selected in breadth-first order **do**
    for each $U \in \mathcal{U}'$ **do**
      T1 := T1@P_i@t@$Walk(j, U)$@$Q_{Tail(j,U)}$@$Walk(1, U_1)$;
Phase II

Step 1 {Postamble checking}

  T2 :=∅;

  for k := 1 **to** n **do**

    T2 := T2@P_k@Q_k@$Walk(1, U_1)$,

Step 2 {$\mathcal{U}$-transition testing}

  T3 := ∅,

  for each $\mathcal{U}$-transition $t = (s_i, s_j; a/o) \notin T$ **do**

    T3 := T3@P_i@t@$Walk(j, U_j)$@$Q_{Tail(j,U_j)}$;

Step 3 {Testing the remaining transitions}

  Compute $E_c = \{(s_i, s_p; a@U_j) \mid (s_i, s_j; a) \in E - (E(T) \cup E(\mathcal{U})) \wedge Tail(j, U_j) = p\}$;

  Let $G' = (S, E_c \cup E(T) \cup E(\mathcal{U}) \cup E_q)$,

  Compute a rural symmetric augmentation G_1 of G'

  with respect to E_c;

  if (G_1 is weakly connected) **then**

    Compute an Euler tour T4 of G_1;

  else begin

    app_rpt(G', E_c, T5);

    mst_rpt(G', E_c, T6);

    symm_rpt(G', E_c, T7);

    T8:=∅,

    for each transition $t = (s_i, s_j; a) \in E - (E(T) \cup E(\mathcal{U}))$ **do**

      T8 := T8 @ P_i@t@$Walk(j, U_j)$@$Q_{Tail(j,U_j)}$;

    Let T4 be a tour with the minimum length in {T5, T6, T7, T8};

  end

  Let s_h be a state in T4 such that P_h@Q_h is the shortest among all the states in T4;

  Rotate T4 such that it starts and ends at s_h;

  T4 := P_h@T4@Q_h

Step 4 {Confirmation of the tail states of the UISs }

  T9 := ∅;

  for j := 2 **to** n **do**

    T9 := T9@P_j@$Walk(j, U_j)$@$Walk(Tail(j, U_j), U_{Tail(j,U_j)})$@$Q_{Tail(Tail(j,U_j)\ U_{Tail(j,U_j)})}$,

Step 5 {Testing all non-T-transitions }

  Compute $E_c = \{(s_i, s_p; a@U_j) \mid (s_i, s_j; a) \in E - E(T) \wedge Tail(j, U_j) = p\}$;

  Let $G' = (S, E_c \cup E(T) \cup E_q)$;

Compute a rural symmetric augmentation G_1 of G'

with respect to E_c;

if (G_1 is weakly connected) **then**

  Compute an Euler tour T10 of G_1;

else begin

  app_rpt(G', E_c, T11);

  mst_rpt(G', E_c, T12);

  symm_rpt(G', E_c, T13);

  T14:=$\emptyset$;

  for each transition $t = (s_i, s_j; a) \in E - E(T)$ **do**

    T14 := T14 $@ P_i @ t @ Walk(j, U_j) @ Q_{Tail(j, U_j)}$;

  Let T10 be a tour with the minimum length in {T11, T12, T13, T14};

end

Let s_h be a state in T10 such that $P_h @ Q_h$ is the shortest among all the states in T10.

Rotate T10 such that it starts and ends at s_h;

T10 := $P_h @$T10$@ Q_h$

Step 6 {Compute the final test sequence}

Let T15 be a tour with the minimum length in {T3@T4 , T9@T10};

Let Γ := T1@T2@T15;

Generate the test sequence by concatenating the inputs along Γ;

end *UIgen_seq*

We will shortly give the procedure *set_uis* for finding a state cover tree T and an UIS set $\mathcal{U}$ satisfying the TUISD property. Under the single fault assumption, we shall now establish that the successful completion of Phase I of the procedure *UIgen_seq* will guarantee a fault-free state cover tree and a verified set of UISs for the IUT.

Theorem 4.1 *Suppose that an IUT has at most 1 fault and it passes Phase I of UIgen_seq successfully then the following are true:*

1. *The state cover tree T obtained from the SPEC is fault-free in the IUT.*

2. *The UIS of each state of the SPEC from the set $\mathcal{U}$ is also an UIS of the corresponding state in the IUT.*

Proof

Let s_i' denote the state in the IUT corresponding to s_i, $1 \leq i \leq n$. There is no ambiguity in the notation with respect to our fault model (which includes only transfer faults and output faults). First note that U_1 is a cyclic UIS of s_1 and s_1'. We will prove the first part by induction on the level number of the state cover tree T.

We claim that all transitions of level 1 are fault-free in the IUT. Let $(s_1, s_i; a/o)$ be a T-transition of level 1. Suppose it has an output fault in the IUT. Then when we apply the input sequence along the tour $t@Walk(i,U)@Q_{Tail(i,U)}@Walk(1,U_1)$, for some $U \in \mathcal{U}'$, which is generated in the first phase, we will get an output mismatch when the input a of t is applied. Therefore the IUT will fail in Phase I. Suppose the transition has a transfer fault in the IUT. Let its new tail state be s_j', where $j \neq i$. In other words, the corresponding transition in the IUT is $(s_1', s_j'; a/o)$. Let us observe the behavior of the IUT for the sequence $t@Walk(i,U_i)@Q_{Tail(i,U_i)}Walk(1,U_1)$. The fact that the IUT is in s_1' before applying the sequence along the above tour is confirmed by applying $Walk(1,U_1)$, a postfix of the tour preceding this walk. When the input sequence along the walk $Walk(i,U_i)$ is applied at s_j', the IUT may or may not traverse the faulty transition t. If it traverses t, then the output sequence will be different from the expected one since a/o does not occur in U_i due to the TUISD property. On the other hand, if it does not traverse t then due to the single fault assumption all the transitions traversed are fault free. Therefore, the output observed will be different from the expected one as per $Walk(i,U_i)$. In either case, the IUT fails in the first phase. Assuming all the T-transitions up to level l are fault-free, we have to prove that all the T-transitions of level $l+1$ are also fault-free. Let $(s_i, s_j; a/o)$ be the current T-transition under test of level $l+1$. Since all the transitions up to level l are fault-free, we can reach s_i' by traversing the fault-free path P_i from s_1'. It is easy to see that Phase I fails if the IUT has an output fault in $(s_i, s_j; a/o)$. Suppose $(s_i, s_j; a/o)$ has a transfer fault and let its new tail state in the IUT be $s_k', k \neq j$. Let us observe the behavior of the IUT for the tour $P_i@t@Walk(j,U_j)@Walk(1,U_1)$.

Clearly $P_i@t$ puts the IUT in the state s'_k. When the input sequence along the walk $Walk(j, U_j)$ is applied at s'_k, the IUT may or may not traverse the faulty transition t. If it traverses t then the output sequence will be different from the expected one due to the TUISD property. On the other hand if it does not traverse t then due to the single fault assumption all the transitions traversed are fault free. Therefore, the output observed will be different from the expected one as per $Walk(j, U_j)$. This completes the induction.

Observe that the initial state the IUT reaches after applying any postamble sequence is always confirmed in the first phase by applying the cyclic UIS U_1 of the initial state. Also only the T-transitions are used to reach the states in order to apply the UISs. Therefore, if Phase I is successful then this phase correctly reaches every state in the IUT in order to apply each sequence in $\mathcal{U}$. In other words, if Phase I is successful, then the UIS of each state of the SPEC from the set $\mathcal{U}$ is also an UIS of the corresponding state in the IUT.

□

The following theorem establishes the 1-fault resolution capability of the UIDD-method.

Theorem 4.2 *Suppose that an IUT has at most one fault, then the UIDD-method detailed in the algorithm* UI_DD *has 1-fault resolution capability of level $n + l_u$ where n and l_u are the number of states and the length of a longest UIS in $\mathcal{U}$, respectively.*

Proof

Suppose that the IUT fails in Phase I while applying the sequence along the following tour for the transition $t = (s_i, s_j; a/o)$.

$$P_i@t@Walk(j, U)@Q_{Tail(j,U)}@Walk(1, U_1), \qquad U \in \mathcal{U}'.$$

Since the T-transitions are traversed in breadth-first fashion, as in the proof of the last theorem, it follows from the TUISD property of T and $\mathcal{U}$ and the single fault

assumption that none of the transitions in P_i is faulty. If an output o' is observed instead of o while applying the input a of the T-transition $(s_i, s_j; a/o)$, then the T-transition has an output fault. Otherwise, either t has a transfer fault or a transition in the walk $Walk(j, U)@Q_{Tail(j,U)}$ is faulty. Thus the fault can be diagnosed within $n + l_u$ transitions.

Assume that the IUT fails in Step 1 of Phase II while applying the test subsequence $P_k@Q_k@Walk(1, U_1)$, for some $k, 2 \leq k \leq n$. As per Theorem 4.1, all the transitions in P_k are fault free. Therefore, the faulty transition lies in the path Q_k. In this case, the fault is located within $n - 1$ transitions. If the IUT fails while applying the sequence generated in the second step of Phase II for the $\mathcal{U}$-transition $t = (s_i, s_j; a/o)$, then let the sequence be on the tour $P_i@t@Walk(j, U_j)@Q_{Tail(j,U_j)}$. We know that none of the transitions in P_i is faulty. If the mismatch is at t, then t has an output fault; otherwise, t has a transfer fault. Suppose that the mismatch occurs while applying a subsequence of $T4$ (computed at the end of Step 3) along the test edge $(s_i, s_p; a@U_j) \in E_c$ for the transition $t = (s_i, s_j; a/o)$. At this stage all the T-transitions, all the $\mathcal{U}$-transitions and all the postambles in $\mathcal{Q}$ are known to be fault free. Therefore, t is the only faulty transition.

Suppose that the mismatch occurs while applying the subsequence in $T9$. for confirming the tail state of $Walk(j, U_j), 2 \leq j \leq n$ in the IUT. Let the subsequence correspond to the tour described below.

$$P_j@Walk(j, U_j)@Walk(Tail(j, U_j), U_{Tail(j,U_j)})@Q_{Tail(Tail(j,U_j),U_{Tail(j,U_j)})}.$$

From Theorem 4.1, it follows that the faulty transition is in $Walk(j, U_j)$. Therefore, the fault is located within l_u transitions. It is easy to verify that the fault can be located exactly if the mismatch occurs while applying any subsequence in the tour $T10$ which is computed at the end of Step 5. Thus we conclude that the UIDD-method locates the fault within $n + l_u$ transitions.

□

If the SPEC and the IUT have the reliable reset capability, then the SPEC and the IUT are not required to have a cyclic UIS at their initial states. However, like the other states in the SPEC, its initial state has an UIS. For this case, the algorithm *UI_DD* needs only minor changes as described below. Suppose that we denote all the reset transitions simply by 'r', the input interaction of the reset transitions. Then, we have to replace every occurrence of Q_i, for all i, $1 \leq i \leq n$, by the symbol r. The walk $Walk(1, U_1)$ postfixed to various tours in the algorithm has to be deleted. The first step in Phase II has to be removed. The following corollary is a direct consequence of these modifications in the algorithm.

Corollary 4.2.1 *Suppose that the SPEC and the IUT satisfy all the conditions required for the method except the cyclic UIS requirement for the initial states. Also, suppose that both the SPEC and the IUT have the reset capability. Then, the UIDD-method locates the fault within $1 + l_u$ transitions.*

□

Observe that the 1-fault diagnosis capability of the UIDD-method is superior to those provided by the UIS-based methods analyzed in Section 3.1.5. As noted by Sabnani *et al* [SD88] $l_u \leq 5$ for most of the known protocols. Therefore, from the above corollary we can deduce that for most of the known protocols with reliable reset capability, the UIDD-method localizes the fault within six transitions.

In the next section, we consider the problem of extracting a state cover tree T and an UIS set $\mathcal{U}$ with the TUISD property.

4.2 State Cover Tree and UIS Set Computation

We know that a state cover tree T and an UIS set $\mathcal{U}$ with TUISD property play key roles in our UIDD-method. In this section, we present an algorithm to find them. Our algorithm is referred to as *set_uis*. This algorithm uses the procedure *find_muis* which

accepts a state, say s_i, and a set (TT) of incoming transitions at s_i and it finds an UIS U_i of length at most $2n^2$ for s_i such that U_i either does not contain the label of at least one transition in TT or there exists a transition in TT such that it does not belong to the walk from any state other than s_i with U_i as the input sequence, provided such an U_i exists. Note that U_i satisfies that part of the TUISD requirement for s_i with respect to at least one transition in TT. Otherwise, it finds a set of UISs of length at most $2n^2$ such that each of the UISs either does not contain the label of at least one incoming transition at s_i or there exists an incoming transition, say t, at s_i such that t does not belong to the walk from any state other than s_i with the UIS as the input sequence. These UISs are used in the later part of the procedure. Thus *find_muis* is invoked exactly once for each state. This procedure is similar to the one by Sabnani and Dahbura [SD88]. While the latter computes an UIO-sequence of length at most $2n^2$ for a given state, our algorithm computes an UIS with special property for a given state. With minor modifications, *find_muis* can be used for computing multiple UISs for a given state.

The algorithm *set_uis* constructs the state cover tree in a breadth-first fashion. In the algorithm, SS is the set of all states already covered by the partial state cover tree T constructed thus far. $NS \subseteq SS$ is the set of all states in SS which are yet to be scanned for possible growth of T. The algorithm starts with an arbitrary cyclic UIS for s_1 and by initializing SS and NS to the singleton set $\{s_1\}$. Initially, T contains no transition. At a given step, a state, say s_k, from NS is chosen and deleted from this set; each state, say s_i, from $S-SS$ which has an incoming transition from s_k is considered for possible extension of T. Let TT be the set of all transitions from states in NS to s_i. If the procedure *find_muis* has already been invoked for s_i, then the set of UISs computed for s_i in that invocation are searched for an UIS which satisfies the TUISD requirement for s_i with respect to a transition in TT. Otherwise, the procedure *find_muis* is invoked with suitable parameters. Suppose that the procedure is successful in finding an UIS, say U_i, which satisfies the TUISD requirement for s_i with respect to a transition, say t, in TT. Then, U_i and t are added to $\mathcal{U}$ and T, respectively. Also, the state s_i is added to both SS and NS. The above step is repeated until T becomes a state cover tree or $NS = \emptyset$. The latter case simply implies that the

SPEC does not have an UIS set having UISs of length at most $2n^2$ and a state cover tree satisfying the TUISD property. We shall now present the formal description of the algorithm. Assume that the function **In**(s_i) (**Out**(s_i)) returns the set of all incoming (outgoing) transitions at the state s_i. Let **Input (Output)** be a function which returns the Input (Output) part of a given input-output sequence. Recall that the functions **Inseq, Outseq** and **IOseq** accept a walk and return the input sequence, output sequence, and the input-output sequence of the walk, respectively. The algorithm uses the following record structures and variables.

```
const
    N=...; { the number of states }
    MAX=...; { some large number }
    MAX2=..., { maximum number of incoming transitions at any state }
type
    u_rec = record
        length: integer;
        useq: sequence of input,
        intrans: set of transitions;
    end u_rec;
    pat_rec = record
        pattern: sequence of input-output;
        end : state_type;
        htset : set of htstate_rec;
        intrans: set of transitions;
    end pat_rec,
    htstate_rec = record { treated as an ordered pair of states }
        head : state_type;
        tail : state_type;
    end htstate_rec;
var
    u_tab: array[1..N, 1..MAX2] of u_rec;
    nucol: array[1..N] of integer;
            { nucol[i] = # of columns filled in the ith row of u_tab }
    OP, P: array[1..MAX] of pat_rec;
    searched: array[1..N] of boolean;
```

```
    coveredtr: array[1..N] of set of trans;
```

Following is the description of the algorithm **sct_uis**

```
Algorithm sct_uis(SPEC, U, T);
begin
   for i := 1 to N do begin
      searched[i] := false;
      nucol[i] := 0;
   end;
   SS := {s1};
   NS := {s1};
   Obtain a cyclic UIS U1 for s1 using the algorithm given in [SD88];
   for each si ∈ S do coveredtr[i] := ∅;
   repeat
      delete a state sk from NS.
      for each si ∈ S−SS do
         if (∃ a transition t' from sk to si such that t' ∉ coveredtr[i])
         then begin
            TT := {t | t ∈ In(si) − coveredtr[i] ∧ head(t) ∈ NS ∪ {sk} };
            if (searched[i]) then begin
               if (∃ j such that 1 ≤ j ≤ nucol[i] ∧ TT ∩ u_tab[i,j].intrans ≠ ∅)
               then begin
                  Choose t ∈ TT ∩ u_tab[i,j].intrans;
                  Ui := u_tab[i,j].useq;
                  Add Ui to U;
                  Add t to T;
                  Add si to SS; Add si to NS;
               end
            end
            else begin
               find_muis(si, TT, Ui, t, u_tab, found);
               searched[i]:= true;
               if (found) then begin
```

```
                Add U_i to 𝒰;
                Add t to T;
                Add s_t to SS; Add s_t to NS;
            end
        end
        coveredtr[i] := coveredtr[i] ∪ TT,
    end { if (∃ a transition t'... }
until(SS=S or NS=∅)
end scl_uis
```

Following is the formal description of the procedure *find_muis*.

```
procedure find_muis(s_i, TT, uis, t, u_tab, found);
begin
    uis := ∅;
    L := 1; j := 0;
    havepattern := false;
    for each ot ∈ Out(s_i) do begin
        first := true;
        for each it ∈ In(s_i) do begin
            if (Label(it) ≠ Label(ot) or
                it ∉ Walk(j, Inseq(ot)) for all j, 1 ≤ j ≤ n, j ≠ i) then begin
                if (first) then begin
                    j := j + 1;
                    OP[j].pattern := Label(ot);
                    OP[j].end := s_Tail(ot);
                    first := false,
                    havepattern := true
                end;
                Add it to OP[j].intrans
            end
        end;
        if (havepattern) then begin
            havepattern := false;
            Let SP := { (ff, tt) | ∃ t ∈ E s.t. t=(ff,tt;Label(ot)) };
```

```
        OP[j].htset := SP − { (s_i, s_{Tail(ot)}) },
        if (OP[j].htset = ∅) then begin
          if (OP[j].intrans ∩ TT ≠ ∅) then begin
            uis := Input(OP[j].pattern);
            Choose t ∈ OP[j] intrans ∩ TT ,
            found := true
            return
          end
          else begin
            for each t ∈ OP[j].intrans do begin
              avail := false;
              for k.= 1, nucol[i] do
                if (t ∈ u_tab[i,k].intrans then
                  avail := true,
              if (not avail) then begin
                p := nucol[i] := nucol[i] + 1;
                u_tab[i,p].length := L·
                u_tab[i,p] useq := Input(OP[j].pattern);
                u_tab[i,p].intrans := OP[j].intrans
              end
            end
          end
        end { if (OP[j].htset = ∅)..... }
      end { if (havepattern) ... }
    end { for each ot ∈ Out(s) .... }
    OND := j,
    while(uis = ∅ and L < 2N^2) do begin
      L := L + 1;
      ND := 0;
      for i := 1 to OND do begin
        for each ot ∈ Out(OP[i].end) do begin
          first := true;
          for each it ∈ OP[i].intrans do begin
            if (Label(it) ≠ Label(ot) or it ∉ Walk(j, Input(OP[i].pattern)@Inseq(ot))
```

```
        ∀ j, 1 ≤ j ≤ n, j ≠ i) then begin
        if (first) then begin
            first := false;
            ND := ND +1;
            P[ND].pattern := OP[i].pattern ∘ Label(ot);
            P[ND].end := s_Tail(ot);
            P[ND].htset := ∅
        end;
        Add it to P[ND].intrans
    end
end { for each it ... }
for each two-state tuple (from, to) ∈ OP[i].htset do
    for each transition ot2 ∈ Out(to) do
        if (Label(ot2) = Label(ot)) then
            P[ND].htset := P[ND].htset ∪ { (from, s_Tail(ot2)) };
if (P[ND].htset = ∅) then
    if (P[ND].intrans ∩ TT ≠ ∅) then begin
        uis := Input(P[ND].pattern);
        Choose t ∈ P[ND].intrans ∩ TT;
        found := true;
        return
    end
    else begin
        for each t ∈ OP[j].intrans do begin
            avail := false;
            for k:= 1, nucol[i] do
                if (t ∈ u_tab[i,k].intrans) then
                    avail := true;
            if (not avail) then begin
                p := nucol[i] := nucol[i] + 1;
                u_tab[i,p].length := L;
                u_tab[i,p].useq := Input(OP[ND].pattern);
                u_tab[i,p].intrans := OP[ND].intrans
            end
```

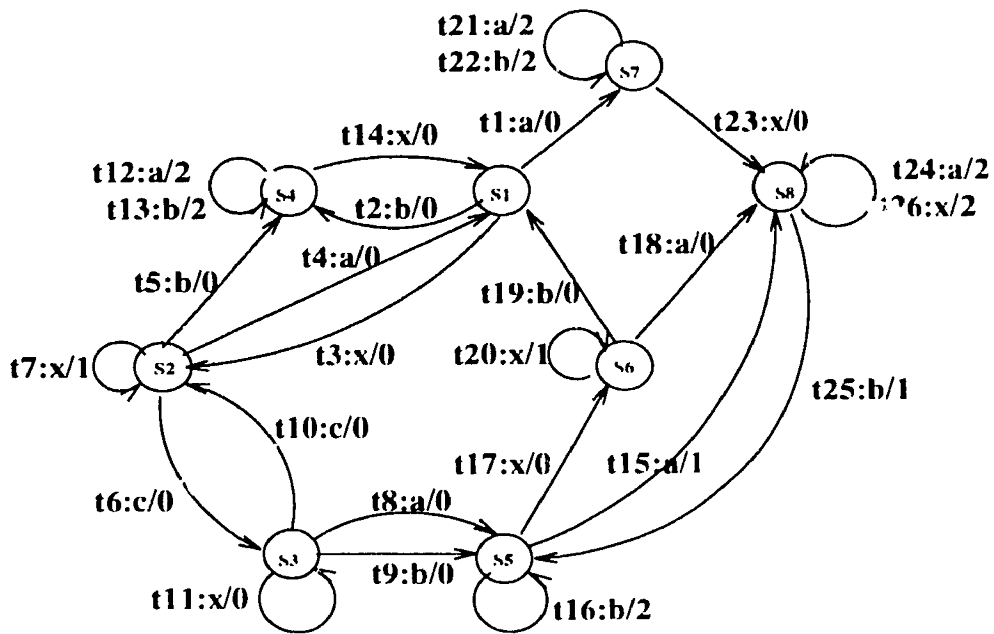
State s_i	UIS U_i	State s_i	UIS U_i
s_1	$t1t21t22$	s_2	$t7t5t13$
s_3	$t8t15$	s_4	$t12t14t1$
s_5	$t15$	s_6	$t18t24$
s_7	$t23t25$	s_8	$t26$

Table 4.1: UISs selected by *set_uis* for SPEC of Figure 4.1(a)

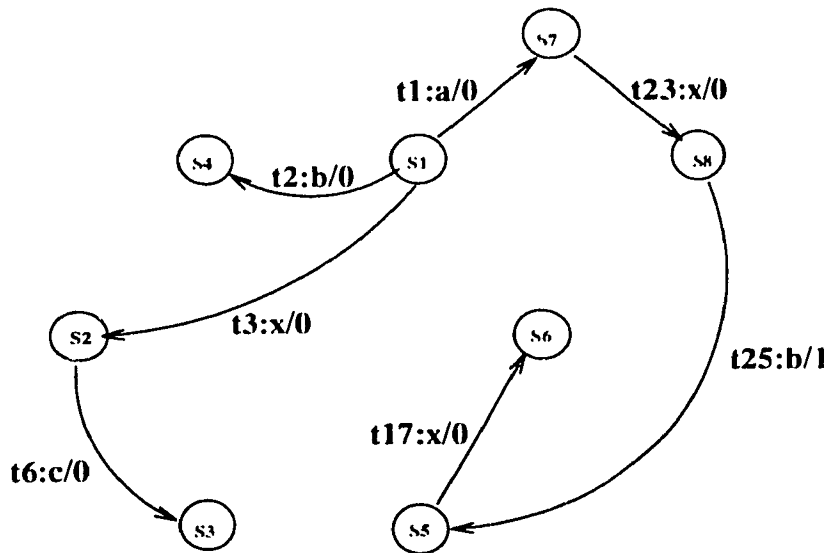
```
                end
             end
          end { for each of  . }
       end { for i = 1 to OND . }
       OND := ND.
       for i = 1 to ND do begin
          OP[i].pattern := P[i].pattern;
          OP[i].end := P[i].end
          OP[i].htset := P[i].htset;
          OP[i].intrans := P[i].intrans;
       end
    end { while(uis = ∅     }
    if(L > 2N^2) then
       found := false
    else found := true
end find_muis
```

4.3 An Illustration of the UIDD-method

We illustrate the UIDD-method by generating a test sequence for the SPEC as shown in Figure 4.1(a). We assume that the SPEC has the reset capability. As before, we express the UISs as well as the test sequence in terms of the transitions. By applying the algorithm of Sabnani and Dahbura [SD88], we get $t1t21t22$ as an UIS for s_1. The



(a) SPEC



(b) State cover tree of the SPEC

Figure 4.1: SPEC and a state cover tree of an abstract protocol

State	Subtour
s_1	$t1t21t22rt3t5t13rt1t23t24r$
s_4	$t2t12t12t13rt2t14t2t13rt2t12t14t1r$
s_2	$t3t4t1t22rt3t7t5t13rt3t4t3t4r$
s_3	$t3t6t8t15t25rt3t6t11t9t16rt3t6t8t17t18r$
s_7	$t1t21t21t22rt1t23t25t16rt1t21t23t24r$
s_8	$t1t23t24t24t25rt1t23t26t25t16rt1t23t24t26t24r$
s_5	$t1t23t25t15t24t25rt1t23t25t17t19t2rt1t23t25t15t26t24r$
s_6	$t1t23t25t17t18t24t25rt1t23t25t17t20t19t2rt1t23t25t17t18t26t24r$

Table 4.2: Test subtours generated in the first phase of DD-method

$\mathcal{U}$-transition	Subtour	$\mathcal{U}$-transition	Subtour r
t5	t3 t5 t12 t14 t1 r	t7	t3 t7 t7 t5 t13 r
t8	t3 t6 t8 t15 r	t12	t2 t12 t12 t14 t1 r
t13	t2 t13 t12 t14 t1 r	t14	t2 t14 t1 t21 t22 r
t15	t1 t23 t25 t15 t26 r	t18	t1 t23 t25 t17 t18 t26 r
t21	t1 t21 t23 t25 r	t22	t1 t22 t23 t25 r
t24	t1 t23 t24 t26 r	t26	t1 t23 t26 t26 r

Table 4.3: Subtours for testing $\mathcal{U}$-transitions

algorithm *set_uis* produces the state cover tree as shown in Figure 4.1(b). The set $\mathcal{U}$ of UISs selected by *set_uis* is tabulated in Table 4.1. Note that U_1 is not cyclic since the SPEC and the IUT are assumed to have the reliable reset capability. The above state cover tree and the UIS set satisfy the TUISD property.

The tour $T1$ generated in the first phase of *UIgen_seq* is shown in Table 4.2. The tour is subdivided into a set of subtours for testing the incoming T-transition as well as the UIS set at different states. Only sequences from the set $\mathcal{U}' = \{aab, xbb, axa\}$ are applied at each state. The length of the resulting tour $T1$ is 138. Also, the postamble set Q in the entire algorithm is replaced by the reset transition denoted by its input symbol 'r'. The test tour $T3$ generated in the second step of this phase for testing all the $\mathcal{U}$-transitions is shown in Table 4.3 as a number of subtours at s_1. The

State	Subtour	State	Subtour
s_1	t1 t21 t22 t23 t25 r	s_2	t3 t7 t5 t13 t12 t14 t1 r
s_3	t3 t6 t8 t15 t26 r	s_4	t2 t12 t14 t1 t23 t25 r
s_5	t1 t23 t25 t15 t26 r	s_6	t1 t23 t25 t17 t18 t24 t26 r
s_7	t1 t23 t25 t15 r	s_8	t1 t23 t26 t26 r

Table 4.4: Subtours for testing the tail states of the UISs

length of the tour is 68. Clearly, $\{t4, t9, t10, t11, t16, t19, t20\}$ is the set of transitions which are neither in the state cover tree nor in any of the UISs. Transitions in this set are tested in the third step. The graph obtained through the rural symmetric augmentation performed in Step 3 is weakly connected. The resulting tour $T4$ of length 38 for testing the transitions in the above set is given below.

t3 $\underline{\text{t4t1t21t22}}$ r t3 t6 $\underline{\text{t10t7t5t13}}$ r t3 t6 $\underline{\text{t11t8t15}}$ t25
$\underline{\text{t16t15}}$ t25 t17 $\underline{\text{t20t18t24}}$ t25 t17 $\underline{\text{t19t1t21t22}}$ r t3 t6 $\underline{\text{t9t15}}$ r

Underlined segments in this tour as well as in other tours correspond to the test subsequences for their leftmost transitions. For example, the underlined segment $\underline{\text{t4t1t21t22}}$ is for testing the transition t4.

Transitions which are not covered by the state cover tree can also be tested using the tour generated in the fourth and the fifth steps. The tour $T9$ of Step 4 consists of the set of subtours for confirming the tail states of the UISs. The subtours are shown in Table 4.4. The length of $T9$ is 50. As the graph induced by the test edges for the above transitions is weakly connected, the graph obtained through the rural symmetric augmentation is also weakly connected. The resulting test tour $T10$ is shown in Table 4.5; the length of $T10$ is 84. As the length of the tour $T3T4$ is shorter than that of the tour $T9T10$, the former is chosen as the tour for testing all the transitions. The length of this tour is 107. The final test sequence is obtained by concatenating the inputs along the transitions in the tour $T1T3T4$. Thus the total length of the test sequence is 244.

t2 t14t1t21t22 r t3 t7t7t5t13 t13t12t14t1 r t3 t4t1t21t22 r t3 t5t12t14t1 r t3 t6 t10t7t5t13 t12t12t14t1 t21t23t25 t15t26 t25 t17 t18t26 t25 t17 t20t18t24 t24t26 t26t26 t25 t17 t19t1t21t22 t22t23t25 t16t15 r t3 t6 t8t15 r t3 t6 t11t8t15 r t3 t6 t9t15 r

Table 4.5: Tour for testing all non-T-transitions

State Symbol	Actual State	State Symbol	Actual State	State Symbol	Actual State
s_1	CLOSED	s_2	CALLED	s_3	CALLING
s_4	REF_WAIT	s_5	CR_RCVD	s_6	CR_SENT
s_7	ACK_WAIT	s_8	CLOSING	s_9	C_ACK_WAIT
s_{10}	G_CLOSING	s_{11}	ESTAB	s_{12}	G_CLOSING_P
s_{13}	GCLS2	s_{14}	GCLS1	s_{15}	GCLS3

Table 4.6: Explanation of state symbol notations in transport protocol

As the maximum length of any UIS in $\mathcal{U}$ is 3, the test sequence generated localizes any output fault or transfer fault in any of the T-transitions within 4 transitions. Any such fault in other transitions is located exactly by the above test sequence.

We have applied the UIDD-method on a subset of the class 4 transport protocol, NBS TP4 developed by the National Bureau of Standards [Nat83]. In [SL89] Sidhu *et al* have analyzed this subset for studying different formal methods of test sequence generation based on the FSM model. The protocol has 15 states and 61 core transitions. The SPEC of the protocol as specified in [SL89] is shown in Figure 4.2. The relation between the state symbols we have used and the actual states as expressed in [SL89] is given in Table 4.6. As shown in Figure 4.2, each transition is given a distinct number. The corresponding labels are given in Table 4.7. Details of the protocol and the abbreviations of input-output symbols can be obtained from the paper by Sidhu *et al* [SL89]. We assume that the SPEC has the reliable reset capability; the reset transitions are however not shown in the figure. In the tours and the test cases the reset transitions are simply expressed by the symbol 'r'. Also, with the completeness assumption, the protocol is considered to be completely specified.

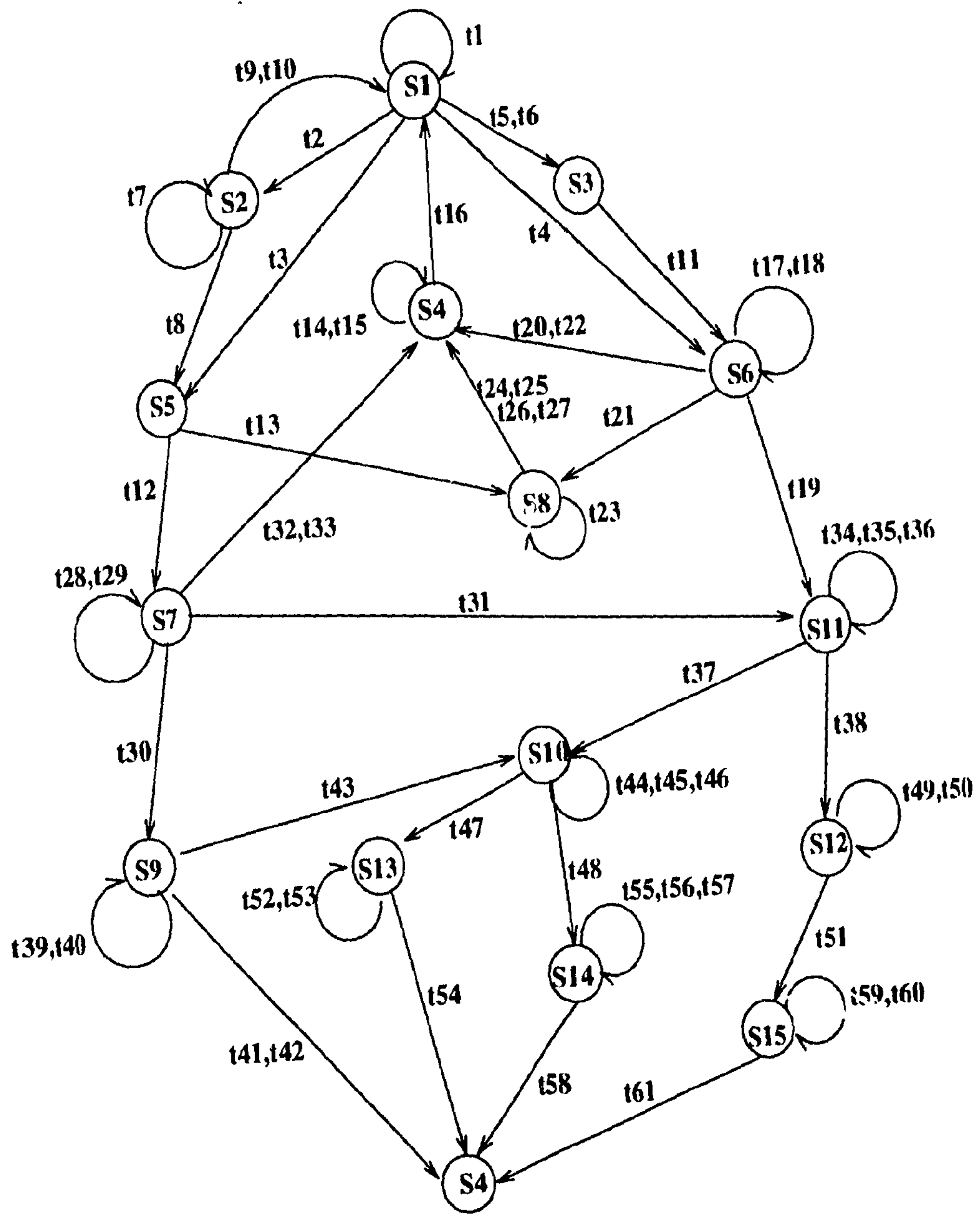


Figure 4.2: A SPEC of a subset of NBS TP4 transport protocol

Trans.	Label	Trans.	Label
t1	N_CR'/DR	t2	S_ct/N_cr S_I_NCT
t3	N_CR/ci	t4	U_cq/CR S_IT
t5	N_di/Null	t6	U_cq'/N_cq
t7	N_CR'/DR	t8	N_CR/ci
t9	S_ct/N_dr	t10	N_di/Null
t11	N_CC/CR S_IT	t12	U_cr/CC S_RTT
t13	U_dr/DR S_TT	t14	N_GR''/AK C_RT S_RT
t15	N_DR/DC S_IT C_RT	t16	S_rt/N_dr
t17	S_it/CR S_IT	t18	S_it'/S_GT
t19	N_CC/AK S_IAT cc C_IT C_GT	t20	N_DR/DC dc C_AT S_RT
t21	N_CC''/DR dc S_TT C_IT C_GT	t22	S_gt/dc S_RT
t23	S_tt/DR S_TT	t24	S_gt/S_RT
t25	N_DC/C_AT S_RT	t26	N_DR/C_AT S_RT
t27	N_di/C_AT S_RT	t28	S_xt'/S_GT
t29	S_xt/cc S_RTT	t30	U_kr/GR
t31	N_AK'/Null	t32	S_gt/dc S_RT
t33	N_DR/DC dc C_AT S_RT	t34	N_AK'/S_IAT C_IAT
t35	N_GR''/C_IAT S_IAT AK	t36	N_GR'/S_IAT C_IAT
t37	U_kr/GR	t38	N_GR/C_IAT S_IAT AK kc
t39	S_xt'/S_GT	t40	S_xt/cc S_RTT
t41	S_gt/dc S_RT	t42	N_DR/DC dc C_AT S_RT
t43	N_AK'/C_RTT C_GT S_IAT S_FCT S_WT	t44	N_GR''/C_IAT S_IAT AK
t45	N_GR'/S_IAT C_IAT	t46	N_AK'/S_IAT C_IAT
t47	N_GR/C_IAT S_IAT AK	t48	N_AK/S_IAT C_IAT
t49	N_AK'/S_IAT C_IAT	t50	N_GR''/C_IAT S_IAT AK
t51	U_kr/GR	t52	N_GR''/C_IAT S_IAT AK
t53	N_AK'/S_IAT C_IAT	t54	N_AK/C_AT S_RT kc
t55	N_GR'/S_IAT C_IAT	t56	N_GR''/C_IAT S_IAT AK
t57	N_AK'/S_IAT C_IAT	t58	N_GR/AK kc C_AT S_RT
t59	N_GR''/C_IAT S_IAT AK	t60	N_AK'/S_IAT C_IAT
t61	N_AK/C_AT S_RT		

Table 4.7: Labels of the transitions of NBS-TP4 transport protocol

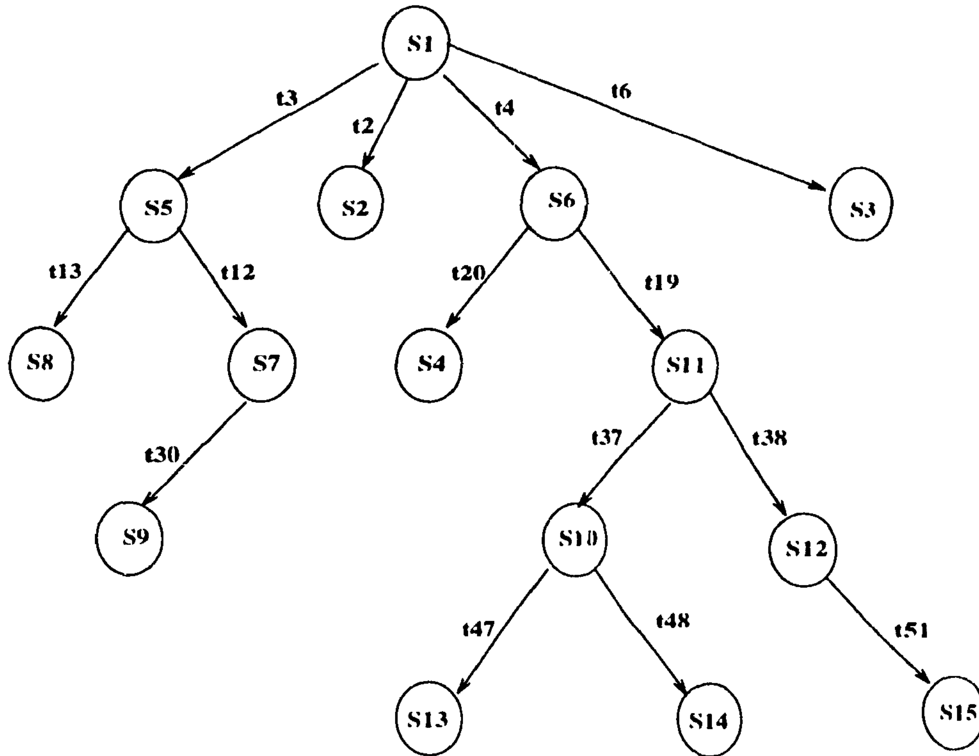


Figure 4.3: A state cover tree of the transport protocol

Again, the non-core transitions which are added due to the completeness requirement are not shown in the figure. By applying the procedure *set_uis* we obtain the state cover tree T and the UIS set $\mathcal{U}$ of UISs as shown in Figure 4.3 and in Table 4.8, respectively. The first phase of the procedure *UIgen_seq* for test case generation results in a tour $T1$ of length 960 transitions. As specified in the procedure, $T1$ consists of

State	UIS	State	UIS	State	UIS
s_1	t4	s_2	t9	s_3	t11
s_4	t15	s_5	t12	s_6	t19
s_7	t31	s_8	t26	s_9	t43
s_{10}	t48	s_{11}	t38	s_{12}	t51 t61
s_{13}	t54	s_{14}	t58	s_{15}	t61

Table 4.8: UISs generated for the transport protocol

Trans	Subtour	Trans	Subtour
t9	t2 t9 t4 r	t11	t6 t11 t19 r
t15	t4 t20 t15 t15 r	t26	t3 t13 t26 t15 r
t31	t3 t12 t31 t38 r	t43	t3 t12 t30 t43 t48 r
t54	t4 t19 t37 t47 t54 t15 r	t58	t4 19 t37 t48 t58 t15 r
t61	t4 t19 t38 t51 t61 t15 r		

Table 4.9: Subtours for testing the $\mathcal{U}$-transitions

a number of subtours given in the following set with the usual notation.

$$\{P_i \circ Walk(i, U_k) \circ r \mid 1 \leq i, k \leq 15\}.$$

{t9, t11, t15, t26, t31, t43, t54, t58, t61 } is the set of $\mathcal{U}$-transitions which are not covered by the state cover tree. A collection of subtours which constitute the tour $T3$ required for testing the transitions in this set is obtained in the second step of Phase II and is shown in Table 4.9. Note that the length of $T3$ is 50. As listed below there are 38 core transitions which are neither in the state cover tree nor in any of the UISs.

t1, t5, t7, t8, t10, t14, t16, t17, t18, t21, t22, t23, t24, t25,
t27, t28, t29, t32, t33, t34, t35, t36, t39, t40, t41, t42,
t44, t45, t46, t49, t50, t52, t53, t55, t56, t57, t59, t60.

A tour for testing these transitions is generated in the third step of Phase II. It is easy to see that $G'[E_c]$ is weakly connected, where G' is the graph constructed in this step and E_c is the set of test edges for the transitions listed above. Therefore, the rural symmetric augmentation G_1 computed in this step is eulerian. An euler tour $T4$ of length 139 obtained from G_1 which is also the tour for testing the above transitions is shown in Table 4.10. By applying Step 4 and Step 5 of the second phase, we compute an alternate tour for testing the non-T-transitions. As this tour is longer than the tour which consists of the tours $T3$ and $T4$ computed in Step 2 and Step 3, the latter becomes the final tour for testing the non-T-transitions. Thus, the concatenation

t2 t7t9 t2 t8t12 t29t31 t34t38 t49t51t61 r t2 t10t4 t22t15 r t5t11 t21t26 r
t1t4 t18t19 t35t38 t50t51t61 r t3 t13 t27t15 r t3 t13 t23t26 r t3 t13 t24t15
r t3 t13 t25t15 r t4 t19 t38 t51 t59t61 r t4 t19 t36t38 t51 t60t61 r t4
t19 t37 t47 t52t54 r t3 t12 t30 t41t15 r t3 t12 t30 t42t15 r t3 t12 t30 t39t43
t46t48 t55t58 r t3 t12 t30 t40t43 t45t48 t56t58 r t3 t12 t28t31 t37 t44t48
t57t58 t14t15 r t3 t12 t33t15 t16t4 t17t19 t37 t47 t53t54 r t3 t12 t32t15 r

Table 4.10: An optimal tour for testing non-T- non-$\mathcal{U}$-transitions

of the inputs along the tour $T1T3T4$ becomes the required test sequence for testing the transport protocol. This test sequence has 1149 inputs. Observe that the test sequence generated localizes any output fault or transfer fault within 3 transitions, if the fault is at a T-transition; otherwise it locates the fault exactly.

4.4 CSDD-method

As indicated in the introduction of this chapter, we can generalize the approach presented in the UIDD-method to develop a diagnosis method which uses a characterizing set for generating the test sequence. In this section, we briefly sketch the method, henceforth referred to as the CSDD-method. While the UIDD-method requires a UIS set and a state cover tree with TUISD property, this method needs a characterizing set, say $\mathcal{W}$ and a state cover tree, say T, with what is known as the **Tree Characterizing Set Disjoint (TCSD) property**. Let $\mathcal{V}_i \subseteq \mathcal{W}$ be the identifying sequence set (IS set) of s_i, $1 \leq i \leq n$. Let t be the incoming T-transition at s_i. T and $\mathcal{W}$ are said to have the **TCSD property** if for each state s_i, $1 \leq i \leq n$, at least one of the following is true.

(a) The label of t does not occur in the input-output sequence along the walk $Walk(i, V)$, for all $V \in \mathcal{V}_i$.

(b) For each state s_j, $1 \leq j \leq n$, $j \neq i$, there exists at least one $W \in \mathcal{W}$ such that $Outseq(Walk(i, W)) \neq Outseq(Walk(j, W))$ and $t \notin Walk(j, W)$.

Note that, as per our assumption, there exists a $U_1 \in \mathcal{W}$ such that U_1 is a cyclic sequence which is a UIS for the initial states of the SPEC and the IUT. Before discussing the CSDD-method, we first present an algorithm for computing a state cover tree which satisfies the TCSD property for a given characterizing set. In the algorithm, *InTr(i,j)* denotes a set of incoming transitions at the state s_i, where $1 \leq i, j \leq n$.

procedure *compute_set*(SPEC,$\mathcal{W}$, T);
begin
  for i := 2 **to** n **do begin**
    InTr(i,i) := $\bigcap_{V \in \mathcal{V}_i} \{e \in In(s_i) \mid Label(e) \notin IOseq(Walk(i, V))\}$;
    for j := 1 **to** n **do**
      if (i $\neq$ j) **then**
        InTr(i,j) := $\{e \in In(s_i) \mid \exists\, W \in \mathcal{W}$
          $\wedge\, Outseq(Walk(i, W)) \neq Outseq(Walk(j, W)) \wedge e \notin Walk(j, W)\}$;
    $InTr(i, i) := InTr(i, i) \bigcup (\bigcap_{1 \leq j \leq n, j \neq i} InTr(i, j))$;
  end;
  Let EE := $\{InTr(i, i) \mid 2 \leq i \leq n\} \cup Out(s_1)$;
  Construct G = (S, EE),
  if (G has a path from s_1 to every other state in S) **then**
    Compute a directed spanning tree T rooted at s_1
  else
    writeln(" No state cover tree for $\mathcal{W}$");
end *compute_set*.

The algorithm finds a state cover tree which satisfies the TCSD property in conjunction with the CS set $\mathcal{W}$, whenever such a tree exists. An alternate CS set has to be chosen if the given CS set does not have the required state cover tree for $\mathcal{W}$.

The CSDD-method of test sequence generation is also based on the Wp-method. It consists of two phases. As defined earlier, let P_i denote the unique path in T from

s_1 to s_i. Let $\mathcal{Q} = \{Q_i \mid 1 \leq i \leq n\}$, where Q_i is a shortest path from s_i to s_1. The first phase is to test the T-transitions as well as to verify if the identifying set of each state of the SPEC is an identifying set of the corresponding state in the IUT. In order to apply a sequence from $\mathcal{W}$ at a given state, P_i is used to reach the state from the initial state; the postamble in $\mathcal{Q}$ is used for reaching the initial state from the state after applying W. As in the Wp-method, all the non-T-transitions are tested in the second phase. For confirming the tail state of any non-T-transitions only those sequences which form the identifying set of the state are applied, instead of applying all the sequence in $\mathcal{W}$. A detailed description of the algorithm is given below.

Algorithm *CS_DD*(SPEC, $\mathcal{W}$)
compute_set(SPEC,T,$\mathcal{W}$);
CSgen_seq(SPEC,T,$\mathcal{W}$);
end *CS_DD*.

procedure *CSgen_seq*(SPEC,T,$\mathcal{W}$)
Phase I {IS verification and T-transition testing}
T1:= $\emptyset$;
for each W $\in \mathcal{W}$ **do**
T1 := T1@Walk(1,W)@$Q_{Tail(1,W)}$@Walk(1,U_1);
for each $t = (s_i, s_j; a/o) \in T$ {select $t \in T$ in BF-order} **do**
for each $W \in \mathcal{W}$ **do**
T1 := T1@P_i@t@$Walk(j, W)$@$Q_{Tail(j,W)}$@$Walk(1, U_1)$;
Phase II {Testing all non-T-transitions }
T2 := $\emptyset$;
for each transition $t = (s_i, s_j; a/o) \in E - T$ **do**
for each $V \in \mathcal{V}_j$ **do**
T2 := T3@P_i@t@$Walk(j, V) Q_{Tail(j,V)}$@$Walk(1, U_1)$;
Let Γ := T1@T2;
Generate the test sequence by concatenating inputs along Γ;
end *CSgen_seq*

We next present results analogous to those for the UIDD-method described in the previous section. For the sake of completeness we present these results with proofs.

Theorem 4.3 *Suppose that an IUT has at most 1 fault and it passes Phase I of CSgen_seq successfully then the following are true:*

1. *The state cover tree T obtained from the SPEC is fault-free in the IUT.*

2. *The IS set $\mathcal{V}_i \subseteq \mathcal{W}$ of s_i of the SPEC is also an IS set of the corresponding state in the IUT.*

Proof

First note that U_1 is a cyclic UIS of the initial state of both SPEC and the IUT. Let s_i' be the state in the IUT corresponding to the state s_i of the SPEC. We will prove the first part by induction on the level number of the state cover tree T. We claim that all transitions of level 1 are fault-free in the IUT. Let $t = (s_1, s_i; a/o)$ be a T-transition of level 1. Suppose it has an output fault in the IUT. Then when we apply the input a, we will get some output $o' \neq o$. Therefore the IUT will fail in Phase I. Suppose the transition has a transfer fault in the IUT. Let its new tail state be s_j', where $j \neq i$. In other words, the corresponding transition in the IUT is $(s_1', s_j'; a/o)$. If we consider that the label a/o does not occur in any of the CS in $\mathcal{V}_i$ and if the subwalk $Walk(i, V)$ of the tour $t@Walk(i, V)@Q_{Tail(i,V)}@Walk(1, U_1)$ in the IUT, for some $V \in \mathcal{V}_i$ such that $Outseq(Walk(i, V) \neq Outseq(Walk(j, V)$, traverses the faulty transition t, then we will observe o while applying the input symbol a as a part of the sequence V. This is a mismatch as the walk $Walk(i, V)$ does not have the label a/o. On the other hand, if the above subwalk does not traverses the unique faulty transition in the IUT, then also a mismatch occurs as the observed output sequence $(Outseq(Walk(j, V))$ is not the same as the expected output sequence $(Outseq(Walk(j, V))$ as we traverse the subwalk in the IUT. If we consider that there exists a $W \in \mathcal{W}$ such that $Outseq(Walk(i, W)) \neq Outseq(Walk(j, W))$ and $t \notin Walk(j, W)$, then a mismatch

will be observed while applying the tour $t@Walk(i,W)@Q_{Tail(i,W)}@Walk(1,U_1)$ to the IUT. Thus, the IUT would not have passed the first phase. Observe that the fact that the IUT is in the initial state before applying this walk is confirmed by applying $Walk(1,U_1)$, a postfix of the tour preceding this walk. Assuming all the T-transitions up to level l are fault-free, we have to prove that all the T-transitions of level $l+1$ are also fault-free. Let $(s_i, s_j; a/o)$ be the current T-transition under test of level $l+1$. Since all the transitions up to level l are fault-free, we can reach s_i' by traversing the fault-free path P_i from s_1'. It is easy to see that Phase I fails if $(s_i, s_j; a/o)$ has an output fault in the IUT. Suppose $(s_i, s_j; a/o)$ has a transfer fault. Then, using the argument similar to the one used in the initial step of the induction, we can prove that the IUT will fail in the first phase. This completes the induction.

Observe that the initial state the IUT reaches after applying any postamble sequence is always confirmed in the first phase by applying the cyclic UIS U_1 of the initial state. Also only the T-transitions are used to reach the states in order to apply the CSs. Therefore, if Phase I is successful, then this phase correctly reaches every state in the IUT in order to apply each sequence in $\mathcal{W}$. In other words, if Phase I is successful, then the IS set of each state of the SPEC from the set $\mathcal{W}$ is also an IS set of the corresponding state in the IUT.

□

Theorem 4.4 *The CSDD-method diagnoses the fault within $n+l_c$ transitions, where n and l_c are the number of states and the length of a longest CS in the set $\mathcal{W}$, respectively.*

Proof

Suppose that a mismatch occurs while applying the following tour for testing the T-transition $t = (s_i, s_j; a/o)$.

$$P_i@t@Walk(j,W)@Q_{Tail(j,W)}@Walk(1,U_1), \quad W \in \mathcal{W}.$$

Since the IUT has produced the expected output for all the tours selected for every transition in P_i in breadth-first order, as in the proof of the previous theorem, we can see that all the transitions in P_i are fault free. Also, the IUT is confirmed to be in the initial state before applying this tour. Therefore, one of the transitions in the walk $t@Walk(j, W)@Q_{Tail(j,W)}$ is faulty. On the other hand, suppose that the observed output is different from the expected one while applying the following tour for testing the non-T-transition $t = (s_i, s_j; a/o)$.

$$P_i@t@Walk(j, V)@Q_{Tail(j,V)}@Walk(1, U_1), \quad V \in \mathcal{V}_j.$$

Then, we infer that t is the unique faulty transition in the IUT. Thus the CSDD-method has the 1-fault resolution capability of level $n + l_c$, where n and l_c are as described in the theorem.

□

The following corollary is easily derivable from the above theorem.

Corollary 4.4.1 *Suppose that the SPEC as well as the IUT has the reset capability then the fault can be located within* $1 + l_c$ *transitions.*

□

The Wp-method assumes the availability of the reset transition from every state in the SPEC (IUT) to its initial state. This is mainly to put the IUT in its initial state between any two successive tours of the IUT for testing transitions. This is achieved in this method by confirming the initial state by a known cyclic UIS of the initial states of the SPEC and the IUT. This requirement is less restrictive than the reliable reset capability requirement. Instead of using arbitrary state cover tree and a CS set as the Wp-method does, our method uses those which satisfy the TCSD requirement. As a result, the CSDD-method satisfies assertions stated in Theorem 4.3 whereas they need not always hold for the Wp-method, as shown in the previous chapter (Section 3.1.6). The CSDD-method achieves better 1-fault resolution capability than the Wp-method for protocols with reliable reset capability.

4.5 Fault Localization

As we know, the level of resolution of the fault is an important factor in any diagnosis method. In this section we present an approach for improving the level of 1-fault resolution capability of the UIDD-method and the CSDD-method by using some additional test sequences. We first discuss this approach with respect to the UIDD-method. A set of transitions is called a **fault resolution set** if it contains a faulty transition. It is known that if the IUT fails in the first phase of $UIgen_seq$ then the fault can be located within $n + l_u$ transitions. Different candidate sets are possible for F depending on the instant at which the IUT fails. Suppose the IUT fails while testing the T-transition $t = (s_i, s_j; a/o)$ with the subsequence along the following subtour, where $1 \leq k \leq n$ and $U \in \mathcal{U}' \subseteq \mathcal{U}$.

$$P_i @ t @ Walk(j, U) @ Q_{Tail(j,U)} Walk(1, U_1).$$

Then, F is taken as the set containing the transition $(s_i, s_j; a/o)$ and the transitions in $Walk(j, U) @ Q_{Tail(j,U)}$. On the other hand if the IUT fails while applying the test subsequence along the subtour $P_k @ Q_k @ Walk(1, U_1)$ which is generated in the first step of the second phase, then consider F to be the set of transitions along Q_k. Assume F to be the set of transitions along the walk $t @ Walk(j, U_j)$ if the IUT fails while testing the $\mathcal{U}$-transition $t = (s_i, s_j; a/o)$ with the input sequence along the tour $P_i @ t @ Walk(j, U_j) @ Q_{Tail(j,U_j)}$ which is obtained in the second step of Phase II. Suppose that the mismatch between the observed and the expected output sequences occur while applying the input sequence along the tour for confirming the tail state of the walk $Walk(j, U_j)$. Clearly, the required tour as described below is generated in the fourth step of Phase II.

$$P_j @ Walk(j, U_j) @ Walk(Tail(j, U_j), U_{Tail(j,U_j)}) @ Q_{Tail(Tail(j,U_j),U_{Tail(j,U_j)})}.$$

In this case, we take the set of transitions along the walk $Walk(j, U_j)$ as the set F. From the proof of Theorem 4.2, it is easy to see that in all the cases F is a fault resolution set. The faulty transition in F can be further localized by repeatedly applying

UIgen_seq using different state cover trees and UIS sets with TUISD property such that the sequences generated in the first phase of *UIgen_seq* do not involve at least one transition from F. The procedure *UIlocalize_fault* for generating a test sequence with improved fault resolution is given below. In this algorithm, $|F| > 1$ initially.

```
procedure UIlocalize_fault(F,SPEC,IUT);
   Let each transition in F be unmarked;
   while (|F| > 1 and F has an unmarked transition ) do
   begin
      Choose and mark an unmarked transition e in F;
      Choose Q such that none of the postambles contains e;
      set_uis(SPEC−e, T_e, U_e);
      if (set_uis is successful) then
      begin
         UIgen_seq(SPEC, T_e, U_e);
         Let F_e be the resulting fault resolution set,
         F := F ∩ F_e;
      end
   end
end UIlocalize_fault.
```

At each iteration of the **while** loop an unmarked transition e in F is marked and, if possible, a state cover tree T_e and a UIS set $\mathcal{U}_e$ are selected such that they satisfy the TUISD property and that neither T_e nor any walk $Walk(j, U_k)$ $1 \leq j, k \leq n$ contains the transition e. Each postamble in Q is chosen in such a way that it does not traverse e. If such Q, T_e and U_e are found then the IUT is tested with the corresponding test sequence and a fault resolution set F_e is obtained as above with respect to this test sequence. From the proof of Theorem 4.2, it is easy to see that either the fault is located exactly or $e \notin F_e$. In the latter case, the number of transitions in the fault resolution set is reduced by at least one since $|F \cap F_e| < |F|$ before the execution of the statement $F := F \cap F_e$. Thus using *UIlocalize_fault* the fault resolution can be improved significantly or the fault can be located exactly.

The same approach can in fact be used for improving the 1-fault resolution capability of the CSDD-method. It is easy to see that F is the set of all transitions in the walk $t@Walk(j, W)@Q_{Tail(j,W)}$ if the IUT fails in the first phase while testing the T-transition $t = (s_i, s_j; a/o)$ with the following tour where $W \in \mathcal{W}$.

$$P_i@t@Walk(j, W)@Q_{Tail(j,W)}@Walk(1, U_1).$$

We know that the fault is located exactly if the IUT fails in the second phase.

The procedure for fault localization is presented below without any further explanation since it is very similar to the previous procedure.

procedure *CSlocalize_fault*(F,SPEC,IUT);

Let each transition in F be unmarked;

while ($|F| > 1$ and F has an unmarked transition) **do**

begin

Choose and mark an unmarked transition e in F;

Choose Q such that none of the postambles contains e;

Choose a CS set $\mathcal{W}_e$ such that $e \notin Walk(i, W) \forall i\ 1 \leq i \leq n$ and $\forall W \in \mathcal{W}_e$;

compute_set(SPEC-e, T_e, $\mathcal{W}_e$),

if (*compute_set* is successful) **then**

begin

CSgen_seq(SPEC, T_e, $\mathcal{W}_e$);

Let F_e be the resulting fault resolution set;

$F := F \cap F_e$;

end

end

end *CSlocalize_fault*.

Observe that the above procedure improves the 1-fault resolution capability by generating additional test sequences. Clearly, the algorithm *UIlocalize_fault* can be improved by considering the original state cover tree T and the original UIS-set $\mathcal{U}$ in the selection of T_e and $\mathcal{U}_e$. Similarly, the algorithm *CSlocalize_fault* can also be improved.

4.6 Summary

In this chapter, we have presented two new diagnosis methods based on the Wp-method. The first method called the UIDD-method uses an UIS set and a state cover tree with a special property known as the TUISD-property. The UIDD-method has a superior 1-fault diagnosis capability. Length of the test sequence is minimized by applying the RPT-algorithm of Aho *et al* [ADLU88] and the heuristic algorithms for the general RPP (developed in Section 2.4) at appropriate places. An adaptive approach is proposed for further improving the 1-fault resolution capability of the method by generating additional test sequences.

Another method, known as the CSDD-method, uses a CS set and a state cover tree with what is called the TCSD-property. The 1-fault resolution capability of the CSDD-method is an improvement over the Wp-method by upto n transitions, where n is the number of states in the SPEC. Further localization of fault is also possible as in the case of the UIDD-method.

Our UIDD-method (CSDD-method) achieves good 1-fault resolution capability on a SPEC if the SPEC has a state cover tree, and an UIS set (CS set) with the TUISD property (TCSD property). Interestingly such a tree and a UIS set (CS set) exist for the simplified NBS TP4 transport protocol which we have used for illustrating the UIDD-method. We have also found that a few other protocols such as the ISDN-BRI-D-Channel signaling protocol (network interface side, originating end) [DSU90a], an FSM representation of a simplified transport protocol studied in [Boc90], and the alternating bit protocol [SD88] satisfy the required conditions for the UIDD-method as well as the CSDD-method.

Some of the results presented in this chapter have been reported in [RDT93, RDTb].

Chapter 5

TEST CASE GENERATION FROM THE EFSM MODEL

In this chapter, we propose a new approach for generating a quadratic polynomial size set of feasible test cases which adequately tests both the control flow and the data flow aspects of a protocol specified as an EFSM. Each test case in this approach corresponds to a tour which starts and ends at the initial state of the protocol. Our control flow coverage criterion is based on the W-method [Cho78] for the FSM model. In the FSM model, this criterion requires the selection of a Characterizing Sequence set (CS set) [Koh78, Cho78] and a set of t st tours such that for each transition and each characterizing sequence, the latter set has at least one tour which covers the transition followed by the characterizing sequence. For the data flow coverage, we extend the "all-uses" criterion proposed in [RW85] for testing computer programs to what is called a **def-use-ob criterion**. We shall see that this new criterion is required due to the so called black-box approach of protocol testing and it enhances the observability of the def-use associations. In the worst case, the order of the set of tours which satisfies the def-use-ob criterion is only quadratic in terms of the number of transitions in the protocol. Another important requirement of our method is to consider the feasibility of the tours during their generation itself. In other words, we do

not intend to first generate a set of tours which satisfies the required coverage criteria and then check for their feasibility. The latter strategy, in general, results in discarding a large number of infeasible tours, which in turn affects the coverage criteria. Except for the combined testing method of Miller and Paul [MP92], all the existing methods have taken the latter strategy of checking for the feasibility of the tours after their selection. The combined testing method uses a backtracking technique for generating feasible test cases. As discussed in Section 1.3.3, this technique does not handle the feasibility issue effectively while joining different types of test subsequences into a single feasible sequence.

In order to achieve the control flow coverage criterion, we define a special type of UIS, known as the **Context Independent Unique Sequence (CIUS)**. We shall discuss the importance of CIUS in Section 5.2.1 where we establish the criterion. We present an algorithm for computing a CIUS for a given state and select one CIUS for each state. The set of CIUSs, one for each state, becomes the required CS set. For tracking the data flow information, we define a new type of data flow graph for a given transition and a walk which contains this transition. From this data flow graph one can determine the set of def-use pairs covered by this walk for the def-use-ob criterion. We design various procedures for manipulating this graph. We finally present our main algorithm which generates a set of test tours which adequately covers the required control flow and the data flow criteria. This algorithm uses a breadth-first approach for computing the test tours.

This chapter is organized as follows. The EFSM model is introduced in Section 5.1. In Section 5.2, we present the test case selection criteria and the data flow graph. The algorithm for computing a CIUS for a given state is described in Section 5.3. Algorithms for manipulating the data flow graphs are developed in Section 5.4. The breadth-first approach for feasible test tour generation is the topic of Section 5.5. We illustrate the proposed approach in Section 5.6 by selecting test tours for a major module in a transport protocol. A restricted case of the feasibility

problem encountered in the CIUS computation algorithm and the test tour selection algorithm is addressed in Section 5.7. We then conclude this chapter in the final section.

5.1 The EFSM Model

The EFSM model presented in this chapter is inspired from [UY91]. An EFSM M is a 6-tuple $M = (S, s_1, I, O, T, V)$, where S, I, O, T, V are a nonempty set of states, a nonempty set of input interactions, a nonempty set of formal output interactions, a nonempty set of transitions, and a set of variables, respectively. These sets are mutually disjoint. Let $S = \{s_j \mid 1 \leq j \leq n\}$; s_1 is called the **initial state** of the EFSM. Each member of I is expressed as $ip?i(parlist)$, where ip denotes an interaction point [BD87] where the interaction of type i occurs with a list of input interaction parameters $parlist$, which is disjoint from V. Each member of O is expressed as $ip!o(outlist)$, where ip denotes an interaction point where the interaction of type o occurs with a formal list of parameters, $outlist$. Each parameter in $outlist$ can be replaced by a suitable variable from V, an input interaction parameter, or a constant. The interaction thus obtained from a formal output interaction is referred to as an **output interaction** or an **output statement**. We will assume that the variables in V and the input interaction parameters can be of types integer, real, boolean, character, and character string only. Each element $t \in T$ is a 5-tuple $t = (source, dest, input, pred, compute_block)$. Here, $source$ and $dest$ are the states in S representing the starting state and the tail state of t, respectively. $input$ is either an input interaction from I or empty. $pred$ is a Pascal-like predicate expressed in terms of the variables in V, the parameters of the input interaction $input$ and some constants. The $compute_block$ is a computation block which consists of Pascal-like assignment statements and output statements. While the left side of an assignment

statement can have only a variable, the expression in the right side can have the input interaction parameter from the input interaction of the transition, variables and constants, of course with suitable operators. A component of a transition can also be represented by postfixing the transition with a period followed by the name of the component. For example $t.pred$ represents the predicate component of the transition t. Note that, unlike a variable, the scope of a parameter in an input interaction of a transition is restricted to the transition only. Let m denote the number of transitions in M. We will assume that $m \geq n$.

As defined in Section 1.2, a walk is a sequence of transitions where the tail state of every transition, except the last transition, is the starting state of its successive transition. The starting state of the first transition and the tail state of the last transition in a walk are called the **starting state** and the **tail state** of the walk, respectively. Recall that a wa' ,aid to be **closed** if the tail state of the walk is also the starting state of the walk. A closed walk which starts and ends at the initial state is referred to as a **tour**.

A transition in M with empty input interaction is called a **spontaneous transition**. A spontaneous transition is referred to as a **silent transition** if its computation block does not have any output statement. A **silent walk** is a walk which has only silent transitions.

A **context** of M is the set $\{(var, val) \mid var \in V$ and val is a value of var from its domain $\}$. A **valid context** of a state in M is a context which is established when M's execution proceeds along a walk from the initial state to the given state.

Let $t = (source, dest, input, pred, compute_block)$ be a non-spontaneous transition in M. t is said to be **executable** if (i) M is in the state $t.source$, (ii) there is an input interaction of type i at the interaction point ip, where $t.input = ip?i(parlist)$, and (iii) the valid context of the state and the values of the input interaction parameters in $parlist$ are such that the predicate $t.pred$ evaluates to $true$. If t =

($source, dest, input, pred, compute_block$), where $input = \emptyset$, is a spontaneous transition, then t is **executable** if (i) M is in the state $t.source$ and (ii) the valid context of the state is such that $t.pred$ evaluates to $true$. When a transition is executed, all the statements in its computation block get executed sequentially and the machine goes to the destination state of the transition after the execution.

A walk W in M is said to be **executable** if all the transitions in W are executable sequentially, starting from the beginning of the walk.

A walk W in M can be executed symbolically [Cla76] by assuming distinct symbolic values for the local variables at the beginning of W as well as distinct symbolic values for the input interaction parameters along W. Note that the same input interaction parameter occurring in two different transitions has to be treated differently. Therefore, different symbols will be taken for each such occurrence. A symbolically executed walk is also called a **(symbolically) interpreted walk.**

Let W be a symbolically interpreted walk. Clearly the conjunction of the predicates along W is also interpreted and is expressed in terms of the initial symbolic values for the local variables and the symbolic values for the input interaction parameters. W is said to be **satisfiable** if the conjunction of the interpreted predicates is satisfiable.

Note that a walk which is executable is always satisfiable. However, its converse is not true. This is because none of the possible values for the variables which made W satisfiable may be a valid context at the starting state of the walk. That is, these values are not 'settable' by any of the executable walks from the initial state to the starting state of W.

A specification described as an EFSM as defined above is also known as a normal form specification [SB86]. In their paper [SB86] Sarikaya and Bochmann have presented an algorithm for transforming an Estelle module into a normal form specification. The **priority** clause in an Estelle module can be eliminated using the approach proposed by Chun and Amer in [CA91]. The transition with **delay** clause

can be treated as a spontaneous transition in the EFSM. This is due to the difficulty in evaluating the relative speeds of the implementation of a protocol and the testing unit [CA91]. In [LL91] Lee and Lee have proposed a method for transforming protocols specified as a system of communicating Estelle modules into a control flow graph which is similar to the EFSM. In [UW93] Ural and Williams provide a mapping for transforming an SDL process or procedure into an equivalent flow graph. The flow graph is similar to the EFSM model studied in this thesis. Methods are also available in the literature [Kar88, Tri92] for transforming protocols specified in SDL or LOTOS into labeled transition systems which are similar to the EFSM. Thus, the proposed test case generation method based on the EFSM model can be applied for generating test cases for protocols which are specified in Estelle, LOTOS, or SDL.

The EFSM representation of the specification, henceforth referred to as SPEC, and the EFSM representation of the implementation, denoted as IUT, of the protocol are assumed to be deterministic. That is, for a given valid context of a state, there exists at most one executable outgoing transition from that state. We assume that the interaction points in the SPEC and the IUT are controllable and observable. In other words, we assume that we can directly apply/observe interactions at the interaction points of the SPEC and the IUT.

An EFSM M is said to be **completely specified** if it always accepts any input interaction defined for the EFSM. We assume that the SPEC and the IUT are completely specified. An arbitrary EFSM M can be transformed into a completely specified one using what is called a **completeness transformation** described next. Given a valid context of a state and an instantiated input interaction, suppose that M does not have an executable outgoing non-spontaneous transition at the state for the given valid context and the input interaction, and that M does not have an outgoing spontaneous transition at the state such that it is executable for the given valid context, then according to the completeness transformation a self-loop transition is added at the state such that it is executable for the given context and the input

interaction. The newly added transitions are called **non-core transitions** and they do not have computation blocks.

It is assumed that for every transition in the SPEC, the SPEC has an executable walk from the initial state to the starting state of the transition such that the transition is executable for the resulting valid context. Similarly, we assume that the initial state is always reachable from any state with a given valid context. We further assume that the SPEC has no satisfiable silent closed walk of length greater than K, for a fixed integer $K \geq 1$. Observe that all the variables in our EFSM model are assumed to be of types integer, real, boolean, character and character string only. That is, we do not consider dynamic data structures.

5.1.1 An Example

As an example of a SPEC, let us consider a major module (*AP-module* in [Boc90]) of a simplified version of a class 2 transport protocol [IS8073]. This module participates in connection establishment, data transfer, end-to-end flow control, and segmentation. It has the interaction point labeled U connected to the transport service access point and another interaction point labeled N connected to a mapping module. Here, we represent the EFSM $SPEC = (S, s_1, I, O, T, V)$ of this module. We would like to note that the $SPEC$ is obtained from the *AP-module* by eliminating a few non-determinisms in certain transitions starting from the data transfer state. This EFSM is used throughout this chapter for illustrating various points. Let $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The following is the set of input interactions (I):

{U?TCONreq(dest_add, prop_opt), U?TCONresp(accpt_opt),
U?TDISreq, U?TDATreq(Udata, EoSDU), U?U_READY(cr),
N?TrCR(peer_add, opt_ind, cr), N?TrCC(opt_ind, cr),
N?TrDR(disc_reason, switch), N?TrDT(send_sq, Ndata, EoTSDU),
N?TrAK(XpSsq, cr), N?ready, N?terminated, N?TrDC }

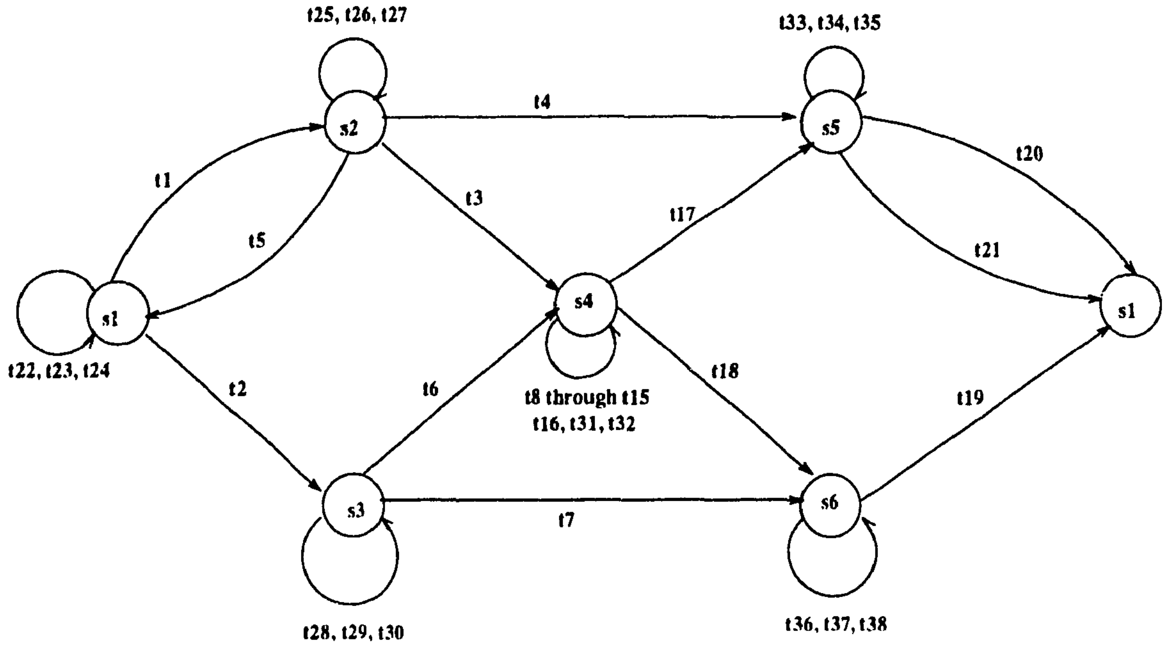


Figure 5.1: An EFSM for the AP-module in the Class 2 transport protocol

The set O of output interactions is given below:

{U!TCONconf(opt), U!TCONind(peer_add, opt), U!TDISind(msg),
U!TDATAind(data, EoTSDU), U!error, U!READY, U!TDISconf,
N!TrCR(dest_add, opt, credit), N!TrDR(reason, switch),
N!terminated, N!TrCC(opt, credit), N!TrDT(sq_no, data, EoSDU),
N!TrAK(sq_no, credit), N!error, N!TrDC}

V={*opt, R_credit, S_credit, TRsq, TSsq* }. All the variables in V are of integer type. The transitions are shown in Figure 5.1, Table 5.1 , Table 5.2 and Table 5.3. The state s_1 is repeated in the figure merely for convenience. Note that the transitions $t1$ to $t21$ are the only core transitions. $t22$ through $t38$ are some of the non-core transitions. While the non-core transitions do not have the computation block, transitions $t22$ through $t38$ do not have predicates either.

Tr.	Input	Predicate	Compute-block
t1	U?TCONreq(dst_add, *prop_opt*)		opt:= prop_opt; R_credit := 0; N!TrCR(dst_add,opt,R_credit)
t2	N?TrCR(peer_add, *opt_ind, cr*)		opt := opt_ind; S_credit := cr; R_credit := 0; U!TCONind(peer_add, opt)
t3	N?TrCC(opt_ind,cr)	opt_ind $\leq$ opt	TRsq:=0;TSsq:=0; opt := opt_ind; S_credit := cr; U!TCONconf(opt)
t4	N?TrCC(opt_ind, cr)	opt_ind > opt	U!TDISind(' procedure error'), N!TrDR('procedure error', false)
t5	N?TrDR(disc_reason, *switch*)		U!TDISind(disc_reason); N!terminated
t6	U?TCONresp(accpt_opt)	accpt_opt $\leq$ opt	opt := accpt_opt; TRsq := 0, TSsq := 0; N!TrCC(opt, R_credit)
t7	U?TDISreq		N!TrDR('User initiated' , true)
t8	U?TDATreq(Udata, *EoSDU*)	S_credit > 0	S_credit := S_credit−1; N!TrDT(TSsq, Udata, EoSDU); TSsq := $(TSsq + 1) mod 128$;
t9	N?TrDT(send_sq, Ndata, *EoTSDU*)	R_credit $\neq 0 \wedge$ send_sq = TRsq	TRsq := $(TRsq + 1) mod\ 128$; R_credit := R_credit − 1; U!TDATAind(Ndata, EoTSDU); N!TrAK(TRsq, R_credit)
t10	N?TrDT(send_sq, Ndata, *EoTSDU*)	R_credit = $0 \vee$ *send_sq* $\neq$ *TRsq*	N!error; U!error
t11	U?U_READY(cr)		R_credit := R_credit+cr, N!TrAK(TRsq, R_credit)

Table 5.1: Core transitions in the transport protocol

Tr	Input	Predicate	Compute-block
t12	N?TrAK(XpSsq, cr)	TSsq≥XpSsq ∧ $cr + XpSsq - TSsq \geq 0$ ∧ $cr + XpSsq - TSsq \leq 15$	S_credit := $cr + XpSsq - TSsq$
t13	N?TrAK(XpSsq, cr)	TSsq≥XpSsq ∧ ($cr + XpSsq - TSsq < 0$ ∨ $cr + XpSsq - TSsq > 15$)	U!error; N!error
t14	N?TrAK(XpSsq, cr)	TSsq<XpSsq ∧ $cr + XpSsq - TSsq - 128 \geq 0$ ∧ $cr + XpSsq - TSsq - 128 \leq 15$	S_credit := $cr + XpSsq - TSsq - 128$
t15	N?TrAK(XpSsq, cr)	TSsq<XpSsq ∧ ($cr + XpSsq - TSsq - 128 < 0$ ∨ $cr + XpSsq - TSsq - 128 > 15$)	U!error, N!error
t16	N?ready	S_credit > 0	U!READY
t17	U?TDISreq		N!TrDR('User initiated', *false*)
t18	N?TrDR(disc_reason, *switch*)		U!TDISind(disc_reason), N!TrDC
t19	N?terminated		U!TDISconf
t20	N?TrDC		N!terminated; U!TDISconf
t21	N?TrDR(disc_reason, *switch*)		N!terminated

Table 5.2: Core transitions in the transport protocol (Contd.)

Transitions	Input
t25, t28, t31, t33, t36	U?TCONreq(dest_add, prop_opt)
t23, t26, t34, t38	U?TDISreq
t22, t29, t37	N?TrDR(disc_reason, switch)
t24, t27, t30, t32, t35	N?terminated

Table 5.3: Non-core transitions in the transport protocol

5.1.2 Unique Input Sequence

An **input sequence** in an EFSM is a sequence of input interactions, where each input interaction, as we know, consists of the name of an interaction point, name of the interaction type, and a set of input interaction parameters. An input sequence is said to be **instantiated** if all the parameters in the sequence are properly instantiated with values. Given an instantiated input sequence X, a state s_i and a valid context C at s_i, $Ewalk(i, X, C)$ denotes the unique walk traversed when X is applied to the SPEC which is currently at s_i with the context C.

A **test sequence** is a sequence of input interactions and output interactions. A sequence of zero or more output interactions between two successive input interactions in a test sequence is the sequence to be observed after applying the preceding input interaction to an EFSM and before applying the succeeding one.

The sequence of input and output interactions along a satisfiable walk W is denoted as **Trace(W)**, known as the **trace of the walk** W. The sequence of input (output) interactions along a walk W is denoted by **Inseq(W) (Outseq(W))**. $Trace(W)$ and $Outseq(W)$ are actually obtained by symbolically executing W for some symbolic values for the variables at the starting state of W and for the symbolic values of the input interaction parameters along W. It is easy to see that these sequences are expressed in terms of the symbolic values and constants. Suppose that the actual value of a symbol is known, then the corresponding sequences can be obtained from the above sequences by replacing the symbol by the value throughout the sequences.

Two input interactions are said to be **distinguishable** if: (i) they occur at two different interaction points or (ii) their interaction types are different. We say that two output interactions are **distinguishable** if at least one of the following is true: (i) they occur at two different interaction points, (ii) their interaction types are different, and (iii) if the parameters in a given position in both interactions are constants then they are different.

For example, the output interactions N!TrDR('procedure error', false) and N!TrDR('procedure error', true) are distinguishable. However, N!TrDT(TSsq, Udata, EoSDU) and N!TrDT(TRsq, Udata, EoSDU) are not distinguishable.

An input interaction is obviously distinguishable from an output interaction. The total number of input and output interactions - each occurrence of an interaction is counted - in a sequence is called the **length** of the sequence. Let S_1 and S_2 be two sequences of input and/or output interactions. Assume that they are of the same length. In order to check for distinguishability of the two sequences, starting from the first position the interactions in S_1 and S_2 are checked position-wise. S_1 and S_2 are said to be **distinguishable** if the interactions in at least one position in S_1 and S_2 are distinguishable. Otherwise, they are said to be **indistinguishable**. Two sequences of different lengths are always distinguishable.

Let W be an executable walk at s_j. Let U be an instantiation of $Inseq(W)$. We define U as an **Unique Input Sequence (UIS)** of s_j if $Trace(W)$ is distinguishable from $Trace(W')$, for any satisfiable walk W' at state s_k, for $k = 1, 2, \ldots, n, k \neq j$. In this case, W is called an **UIS walk** for U. Note that it is enough to check only those walks W' with $Inseq(W) = Inseq(W')$.

5.1.3 Control Flow Fault Model

We assume the the IUT has the same number of states as the SPEC. Let s_j' denote the IUT state corresponding to s_j, $1 \leq j \leq n$.

A transition t from s_j to s_k in the SPEC is said to have a **simple control flow fault** in the IUT if the corresponding transition from s_j' terminates at some state s_p', where $p \neq k$.

A transition t from s_j to s_k in the SPEC is said to have **output interaction type fault** in the IUT, if the sequence of output interaction types of the corresponding transition in the IUT is different from that of t.

We define the **control flow fault coverage** of a test case generation method

for the EFSM model as the percentage of faulty implementations the method detects from the set of all implementations with any number of simple control flow faults and/or output interaction type faults, only.

The data-oriented fault model for the EFSM will be discussed in Section 5.5.2.

5.2 Test Case Selection Criteria

In this section we define the coverage criteria for the control flow and the data flow.

5.2.1 Control Flow Coverage

Our test generation scheme for the control flow coverage is similar to the W-method [Cho78]. We would like to apply an UIS of every state at the tail state of the transition under test. A careful selection of the UISs is required since an arbitrary UIS for each state may not be sufficient to generate test cases. For example, let U be an UIS for s_j, and let W be the UIS walk of U. Let t be an incoming transition at s_j and . be the starting state of t. In order to test t, we need to compute an executable preamble walk P_t from s_1 to s_i and associate values for the input interaction parameters along P_t and t such that $P_t\ t\ W$ is executable. For a given W, it is in general difficult to find a P_t so that the walk $P_t\ t\ W$ is executable.

A walk from a state is said to be **context independent** if the predicate of every transition along the walk, duly interpreted, is independent of the symbolic values of the local variables at the starting of the walk. Observe that every context independent satisfiable walk is executable.

We introduce a special type of UIS, called **Context Independent Unique Sequence (CIUS)**. Let U_i be an UIS of s_i and let $U(i)$ be the corresponding UIS walk at s_i. U_i is said to be a **CIUS** of s_i if $U(i)$ is context independent and executable.

Note that all the local variables used in the predicate of each transition in $U(i)$ are defined within $U(i)$ prior to their use. In other words, the predicates along $U(i)$

are independent of any valid context at s_i. Therefore, $U(i)$ can be postfixed to any executable walk from the initial state to s_i and the resulting walk is also executable. This property reduces the complexity of computing feasible test cases for the control flow coverage.

In the next section, we present an algorithm for computing a CIUS for a state. Note that a CIUS is simply a sequence of input interactions and is always instantiated. The test subsequence which corresponds to the application of a CIUS at a state is the trace along a walk which is executed when the CIUS is applied at that state. Thus, in this sequence, each input interaction of the CIUS will be followed by a sequence of zero or more output interactions to be observed before applying the next input interaction. This test subsequence, of course, depends on the state where the CIUS is applied and the valid context of the state if the CIUS is not for this state.

A CIUS is said to be **cyclic** if the corresponding CIUS walk is a cycle. We assume that the SPEC has a cyclic CIUS U_1 for its initial state, which is also a cyclic CIUS for the initial state of the IUT. This, in our view, is easy to realize in practice. For example, the SPEC can be designed and the IUT implemented such that their initial states have a self-loop with an unique input interaction and an output interaction. Observe that this is similar to the "status/state" self-loop transition proposed in [DSU90b]. However, we need such a transition only for the initial states of the machines. Let U_i be a CIUS for the state s_i, $2 \leq i \leq n$. Let $\mathcal{U} = \{U_i \mid 1 \leq i \leq n\}$. We call $\mathcal{U}$ as a **CIUS set.** Our control flow coverage criterion, called the **trans-CIUS-set criterion** is to select a set $\mathcal{T}$ of executable tours such that for each transition t in the SPEC and for each $U_i \in \mathcal{U}$, $\mathcal{T}$ has a tour which traverses t followed by U_i. An executable walk from the initial state to the starting state of a transition t is called a **preamble walk for** t if Wt is also executable. In Section 5.5.1, we present an algorithm for generating a set of feasible test tours for covering this control flow criterion as well as the data flow coverage criterion which is established in the following.

5.2.2 Data Flow Coverage

The data flow testing is basically to check if the implementation has the right flow of information as its execution proceeds. A hierarchy of data flow coverage criteria has been proposed in [RW85] for testing computer programs. It is proved that the "all-uses" criterion is superior to those criteria, for instance the "all-defs" criterion, which can be satisfied by a polynomial order set of test cases [Wey84, RW85]. Ural and Williams [UW93] have recently used the all-uses criterion for generating test cases for protocols specified in SDL. Due to the black-box approach of protocol testing, the set of test cases which satisfies the all-uses criterion may not have observability. For the data flow coverage, we extend the all-uses criterion to what is called a **def-use-ob criterion**. An useful property of the def-use-ob criterion is that the set of test cases selected as per this criterion facilitates the tester to observe every def-use association in the protocol. The observable extension is similar to the one proposed for the IO-def-chain criterion [UY91].

We introduce some definitions before formally defining the def-use-ob criterion. A parameter v occurring in the input interaction of a transition t is referred to as a **def** and is denoted by $t.I.v$. Similarly, a variable v in the left side of an assignment statement at the location c in the computation block of a transition t is also said to be a **def** and it is denoted by $t.c.v$. The use of a variable or input interaction parameter v in the predicate of a transition t is called a **p-use** and is denoted by $t.P.v$. The variable/input interaction parameter v used in the right side of an assignment statement at the location $c1$ in the computation block of a transition t is referred to as a **c-use** and is denoted by $t.c1.v$. Similarly, the variable/input interaction parameter v appearing as a parameter in the output interaction at the location $c2$ in the computation block of a transition t is referred to as a **c-use** and it is denoted by $t.c2.v$. By an **use**, we refer to a p-use, a c-use or a o-use.

A **def-use pair** D with respect to a variable/parameter v is an ordered pair of def and use of v such that there exists a walk in the SPEC which satisfies the

following: (i) the first transition in the walk is the one where v is defined (i.e., where the def occurs) and the last transition of the walk is the one where v is used (i.e., where the use occurs) and (ii) v is not redefined in the walk between the location where it is originally defined and the location where it is used. Such a walk is called a **def-clear** walk for D. Note that the walk could be a single transition.

Let $\mathcal{D}$ be the set of all def-use pairs for all the variables and input interaction parameters. The same input interaction parameter occurring in two different transitions are treated as distinct. The def-use pairs in $\mathcal{D}$ can be classified into five types as follows.

type 1: An input parameter v is defined in the input interaction of a transition t_1 and is used in the predicate of the same transition. Such a pair is denoted by $(t_1.I, t_1.P)v$.

type 2: An input parameter v is defined in the input interaction of a transition t_1 and is used in an output statement c_2 in the computation block of the same transition. Such a pair is denoted by $(t_1.I, t_1.c_2)v$.

type 3: An input parameter v is defined in the input interaction of a transition t_1 and is used in an assignment statement c_3 in the computation block of the same transition. Such a pair is denoted by $(t_1.I, t_1.c_3)v$.

type 4: A variable v is defined in an assignment statement c_1 in the computation block of a transition t_1 and is used in the predicate of another transition t_2. Such a pair is denoted by $(t_1.c_1, t_2.P)v$.

type 5: A variable v is defined in statement c_1 in the computation block of a transition t_1 and is used in statement c_2 in the computation block of a transition t_2. Such a pair is denoted by $(t_1.c_1, t_2.c_2)v$. Note that c_2 can either be an assignment statement or an output statement.

The def-use pairs of types 1 and 2 are not considered here because, as we shall see later, they are covered for the def-use-ob criterion by our trans-CIUS-set criterion itself. We assume that the set $\mathcal{D}$ of all def-use pairs of types 3, 4 and 5 in the SPEC is known. Note that, apart from the usual def-use pairs where the def and the use belong to different transitions, $\mathcal{D}$ also includes the def-use pairs within a transition. A minor modification of the algorithm presented in [CZ93] would suffice to obtain $\mathcal{D}$. This modification is to consider the def-use pairs within a transition.

Let l (l') be a location in transition t (t') where a variable/parameter v (v') is defined (used). Suppose that $X = D_1 D_2 \ldots D_k$, where $k \geq 1$, is a sequence of def use pairs such that (i) D_i is a def-use pair for variable v_i, $i = 1, 2, \ldots, k$, (ii) $v_1 = v$ and $v_k = v'$ and the source of D_1 is $t.l$ and the destination of D_k is $t'.l'$, (iii) the use part of D_i is for defining v_{i+1}, where $i = 1, 2, \ldots, k-1$ and (iv) if $k = 1$, then $v = v'$. Then, X is called an **information flow chain** from the definition of v at the location l of transition t to the use of v' at the location l' of transition t'. Further, if a walk W has a subwalk W' with t and t' as the first and the last transition such that W' can be expressed as $W' = W'_1 @ W'_2 @ \ldots @ W'_k$, where W'_i is a def-clear walk for D_i, for $i = 1, 2, \ldots, k$, then, we say that X is an **information flow chain along** W.

Our **def-use-ob criterion** requires the selection of a set of executable tours such that for each feasible def-use pair $D \in \mathcal{D}$, the set has at least one tour, say T, satisfying the following conditions.

(a) If the use part in D is an o-use, then T contains a def-clear walk for D.

(b) If the use part in D is a p-use, then T contains a def-clear walk $W1$ for D followed by the CIUS walk $U(j)$, where s_j is the tail state of $W1$.

(c) If the use part in D is a c-use, then T contains a walk $W2$ followed by a walk $W3$, where $W2$ is a def-clear walk for D and $W3$ has an information flow chain from the variable which is defined at the location where the variable for D is c-used to a location where a variable is either o-used or p-used. Moreover, if

the information flow chain terminates in a p-use variable, then, in T, $W3$ is followed by the CIUS walk $U(p)$, where s_p is the tail state of $W3$.

Condition (a) takes care of the def-use association for all the def-use pairs in which the use part is an o-use. If the use part of D is a p-use, then apart from meeting the def-use association, by applying the CIUS of s_j, condition (b) enables the tester to check if the predicate of the transition where the p-use occurs evaluates to **true** as expected. On the other hand, if the use part of D is a c-use, then condition (c) enables the tester to observe the effect of the value computed. Actually, this value flows through other intermediate variables along T until it is used in an output statement or in a predicate of a transition. In addition, the correct evaluation of the predicate is ensured by T as in condition (b).

An executable walk W starting from the initial state is called a **preamble walk for** D if it satisfies conditions (a), (b) and (c) where T is replaced by W.

We know that, as per the trans-CIUS-set criterion, each transition followed by the CIUS of the tail state of the transition will be covered by at least one tour. Clearly, this tour also covers all the def-use pairs of types 1 and 2 for the def-use-ob criterion.

We define a new type of Data Flow Graph (DFG) to represent the data flow information on a particular executable walk starting from the initial state. Given a transition t in such a walk W, the data flow graph for t with respect to W contains the data flow information along W for all the input interaction parameters and local variables defined in t. Suppose $\mathcal{D}_1 \subseteq \mathcal{D}$ such that each def-use pair in $\mathcal{D}_1$ has its def part in t. Then, this graph is useful in computing the subset of $\mathcal{D}_1$, for which this walk is a preamble walk, except possibly for the CIUS walk extension.

The data flow graph has four types of nodes: i-node, c-node, p-node and o-node.

- An **i-node** is labeled as (t, I, v) and it corresponds to the definition of the parameter v in the input interaction of the transition t.

- A **c-node** is labeled as (t, c, v) and it corresponds to the definition of the variable v in the assignment statement c of the transition t.
- A **p-node** is labeled as (t, P) and it indicates that the node corresponds to the predicate of the transition t.
- A **o-node** is labeled as (t, c) and it simply denotes that it corresponds to the output statement c in the computation block of the transition t.

The **data flow graph** for the transition t with respect to the walk W which contains t is denoted by DFG$[t, W]$. DFG$[t, W]$ has a connected subgraph G for each definition of a variable or a input interaction parameter, say v, in t. G is a directed graph with a unique designated node, called the **root node** which identifies the definition of v. If we consider the nodes to be in three different levels with the root node as the unique node in the first level, then the edges are always from the node of a given level to a node in the next higher level. It is always the case that the root node of G is either an i-node (t, I, v) or a c-node (t, c, v), which represents the definition of the input interaction parameter v or the local variable v, respectively. Nodes in level 2 correspond to the direct use of v in statements/predicates in W. In other words, the pair consisting of the root node and a level 2 node is in fact a def-use pair for which the part of W is a def-clear walk for this pair. The root node is connected to all the nodes of level 2. It is easy to see that a node in level 2 can be a c-node, p-node, or a o-node. The level 2 p-nodes and o-nodes do not have any outgoing edges. Similar to a level 2 node, a level 3 node is a c-node, p-node, or a o-node such that there exists a data flow along W from an assignment statement corresponding to at least one c-node in level 2 to a predicate, assignment statement, or an output statement corresponding to this level 3 node. A c-node in level 2 is connected to a level 3 node if there exists an information flow chain along W from the level 2 node to the level 3 node. For notational convenience, we also denote a node at a given level by attaching the level number as a subscript to the label of the node. For example, a c-node

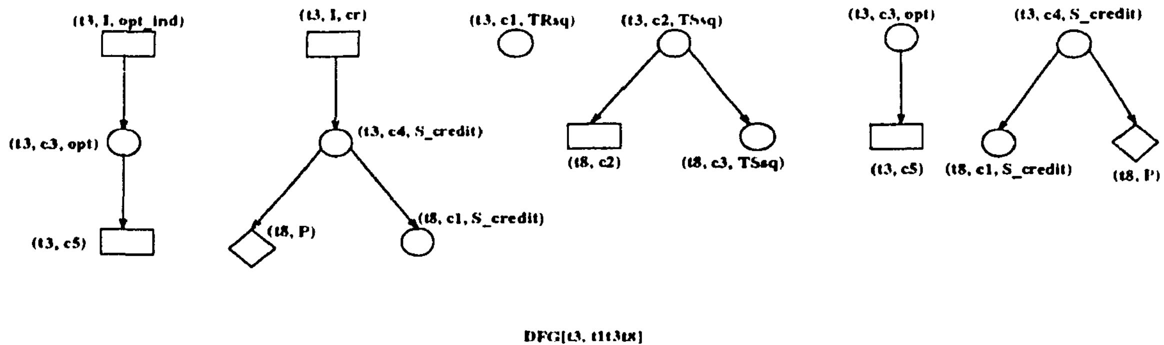


Figure 5.2: A data flow graph for $t3$ with respect to the walk $t1t3t8$

(t, c, v) at level 3 is also denoted by $(t, c, v)_3$. In the graphical representation of a data flow graph, rectangles represent i-nodes as well as o-nodes, whereas the circles and diamonds represent c-nodes and p-nodes, respectively.

Figure 5.2 shows the data flow graph DFG$[t3, t1t3t8]$, for the transition $t3$ in the walk $t1t3t8$ of the EFSM given in Figure 5.1. The second subgraph in this data flow graph corresponds to the definition of the input interaction parameter cr. Hence the root node of the subgraph is the i-node $(t3, I, cr)$. Since cr is directly used in the definition of the variable S_credit at the fourth computation statement in $t3$, this subgraph has an edge from $(t3, I, cr)$ to the level 2 c-node $(t3, c4, S_credit)$. Similarly the edges from $(t3, c4, S_credit)$ to the level 3 nodes $(t8, P)$ and $(t8, c1, S_credit)$ indicate that the variable S_credit defined in $t3.c4$ is p-used at the predicate of transition $t8$ and c-used in the definition of S_credit at the first statement in the computation block of $t8$, respectively.

Algorithms for constructing and manipulating a data flow graph are presented in Section 5.4. In the following, we establish that the trans-CIUS-set and the def-use-ob criteria together require only a set of test tours of quadratic order.

Theorem 5.1 *The order of the set of test tours required to satisfy the trans-CIUS-set and the def-use-ob criteria together is quadratic in the number of transitions in the EFSM SPEC.*

Proof:

Let n denote the number of states in the SPEC. We know that the trans-CIUS-set criterion requires the selection of a set $\mathcal{T}$ of executable tours such that for each transition t in the SPEC, and for each $U_i \in \mathcal{U}$, $\mathcal{T}$ has a tour which traverses t followed by the walk along U_i. Thus, this criterion requires a maximum of n tours for each transition. Therefore, the maximum number of tours required to satisfy this criterion is only $m \times n$, where m is the number of transitions.

Let P_{max} and C_{max} denote maximum number of input interaction parameters in any input interaction and the maximum number of statements in the computation block of any transition, respectively. Thus the maximum number of definitions in the SPEC is only $m(P_{max} + C_{max})$. Suppose C_u is the maximum number of times a variable or an interaction parameter is used in a predicate, or in a computation statement. Then, the maximum number of def-use pairs in $\mathcal{D}$ is only $m^2(P_{max} + C_{max})(C_{max} + 1)C_u$. In order to satisfy the def-use-ob criterion, we need only one tour for each def-use pair in $\mathcal{D}$. Therefore, the maximum number of tours to satisfy the def-use-ob criterion is $O(m^2)$.

Since $m \geq n$, from the above we conclude that the order of the set of test tours required to satisfy the trans-CIUS-set and the def-use-ob criteria together is quadratic in the number of transitions.

□

In the following section we develop an algorithm for computing a CIUS of a given state.

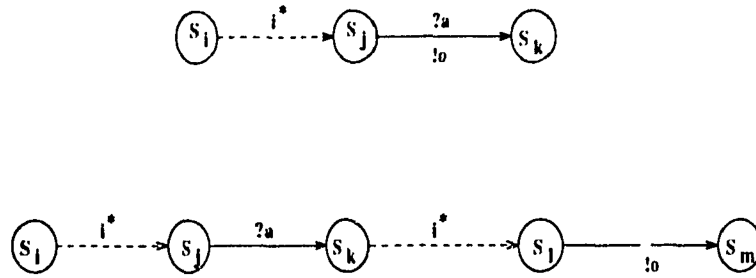


Figure 5.3: Different walks with the same behavior sequence a/o

5.3 CIUS Computation Algorithm

The presence of contexts and predicates in the transitions makes the EFSM model more powerful than the FSM model. Unlike a transition in an FSM, the one in an EFSM need not have both an input and an output. Due to these enhancements and the special characteristics of the CIUS, the existing algorithm [SD88] for computing UIO-sequence is not suitable for the CIUS computation. For instance, in order to find whether a state in an FSM produces an output o when an input a is applied, it is enough to check if the state has any outgoing transition with the label a/o. But in an EFSM, any of the two types of feasible walks from the state, say s_i, as shown in Figure 5.3, may have the same trace; therefore their presence has to be analyzed. In the figure, the dashed edges with label i^* denote walks consisting of a finite number (possibly zero) of silent transitions. $?a$ and $!o$ indicate the input interaction a and the output interaction o, respectively. Note that these interactions do not have any parameter. In this section, we develop an algorithm for computing a CIUS of a given state in the EFSM SPEC. In order to guarantee the termination of the algorithm, as stated earlier, we assume that the SPEC has no satisfiable silent closed walk of length more than K for some integer $K \geq 1$.

Let $W1$ and $W2$ be two satisfiable walks in the SPEC. Let t be the last transition in $W1$. We say that $Trace(W1)$ **subsumes** $Trace(W2)$ if there exists a sequence OS of zero or more output interactions at the end of t such that $Trace(W1)$ and $Trace(W2)@OS$ are indistinguishable. OS is called the **surplus sequence** in $W1$

with respect to $W2$. By a **null walk** at a state , we refer to an empty walk, a walk without any transition, starting and ending at that state. We first present a higher level description of the proposed algorithm.

Input: An EFSM SPEC and a state s_k in SPEC

Output: If it exists, a CIUS for s_k such that its underlying walk W is of length $\leq 2n^2$ and W has only non-silent transitions.

Step 0 { Initialization }

(i) $Wset$:= { the null walk at s_k }.

(ii) $OWset$:= { the null walk at s_j | $1 \leq j \leq n, j \neq k$ }.

(iii) $L := 0$.

Step 1 { Iterative step }

(i) repeat (a) to (c) until $L \geq 2n^2$:

(a) $L := L + 1$.

(b) Set $TWset$ and $TOWset$ to the empty set.

(c) Do Step 2.

(ii) Stop.

Step 2 { Compute TWset, the set of all context independent executable walks of length L at s_k. Also, compute TOWset, the set of all satisfiable walks from the other states such that the trace of each walk subsumes the trace of a walk in TWset. }

(i) Do Step 2.1 for each walk $W \in Wset$ and for each non-silent outgoing transition t in the SPEC at the tail state of W.

(ii) Copy $TWset$ to $Wset$.

(iii) Copy $TOWset$ to $OWset$.

Step 2.1 { For a given walk W and a transition t, if Wt is a context independent executable walk, then find the set of all satisfiable walks from states other than s_k such that the trace of each walk subsumes the trace of Wt. If the latter set is empty, then $Inseq(Wt)$ is the required CIUS of s_k.}

(i) If $W^{\prime} t$ is a context independent executable walk then do the following in that order.

(a) Let W'' be the walk $W^{\prime} t$.

(b) Add W'' to *TWset*.

(c) Initialize *NOWset* to the empty set.

(d) Do Step 2.1.1 for each walk W_1 in *OWset*.

(e) If *NOWset* is empty then Declare *Inseq(W' t)* as the CIUS and stop.

Step 2.1.1

(i) If there exists a satisfiable extended walk W_2 in the SPEC for W_1 such that $Trace(W_2)$ subsumes $Trace(W'')$ then do the following:

(a) Add all such satisfiable extended walks of W_1 to *TOWset*.

(b) Add all such satisfiable extended walks of W_1 to *NOWset*.

This algorithm computes a CIUS of the state s_k in the SPEC such that the walk from s_k which corresponds to the CIUS is of length at most $2n^2$. For better controllability and observability while testing, these walks are allowed to have only non-silent transitions. However, the algorithm can easily be adapted to compute CIUS walks with silent transitions. We would like to note that there may exist a state which does not have a CIUS, in general, and the CIUS with this length and non-silent transition restriction, in particular.

At the beginning of the ith iteration of Step 1, *Wset* contains the set of all context independent executable walks of length $(i-1)$ which starts from s_k. At the same instant, *OWset* contains the set of all satisfiable walks from all the states other than s_k such that the trace of every walk in *OWset* subsumes the trace of a walk in *Wset*. In Step 0, *Wset* is initialized with the null walk at s_k and *OWset* contains the null walk at s_j, for all $j, 1 \leq j \leq n, j \neq k$. Step 1 is the iterative step which is repeated at most $2n^2$ times.

When the ith iteration of Step 1 invokes Step 2, the latter step computes a set

($TWset$) of context independent executable walks of length i which start from s_k. This is done by checking the executability of the walk obtained from each walk W in $Wset$ by appending each non-silent outgoing transition from the tail state of W to W. Step 2 also computes the set ($TOWset$) of all satisfiable walks from any state other than s_k such that the trace along a walk in this set subsumes the trace along some walk in $TWset$. Step 2 does these computations by repeatedly calling Step 2.1 which in turn invokes Step 2.1.1 many times. $TWset$ and $TOWset$ become $Wset$ and $OWset$, respectively, for the $(i+1)$th iteration. Step 2.1 and Step 2.1.1 are explained below.

Given a walk $W \in Wset$ and a non-silent outgoing transition t from the tail state of W, if the walk $W' = W\ t$ is executable and context independent, then Step 2.1 adds this walk to $TWset$. For each walk $W_1 \in OWset$, Step 2.1 invokes Step 2.1.1 for computing the set of all satisfiable walk extensions of W_1 such that the traces of the resulting walks subsume the $Trace(W')$. In $NOWset$, Step 2.1 stores the set of all satisfiable walks from any state other than s_k such that the trace along a walk in this set subsumes the trace along a walk $W' \in TWset$. If $NOWset$ is empty, then the trace along W' is clearly a CIUS of s_k. And in this case the algorithm terminates. If $NOWset$ is not empty for all the $2n^2$ iterations, then the algorithm terminates without finding a CIUS for s_k.

Given a walk $W_1 \in OWset$, and a walk $W' \in TWset$, Step 2.1.1 computes the set of all satisfiable walk extensions of W_1 such that the traces of the resulting walks subsume $Trace(W')$. The extended walks are added to $TOWset$ as well as $NOWset$.

We shall now present the formal description of a more detailed algorithm. Explanations for the various steps in the algorithm are given after the formal description. It uses the function **Prefix(X,Y)**, where X and Y are sequences of output interactions. It returns **true** if a prefix of Y is indistinguishable from X. In this case, we say the X is an **indistinguishable prefix** of Y. Otherwise, it returns **false**. The

function **DelPrefix(X,Y)** returns the sequence obtained from Y by deleting the prefix of Y which is indistinguishable from X provided Prefix(X,Y) is true. Throughout this algorithm '@' is used as a concatenation operator.

Algorithm *ComputeCIUS* (SPEC: EFSM, s_k: state)

{ A CIUS for the state s_k in the SPEC is computed such that the
CIUS walk W is of length at most $2n^2$ and W has no silent transition,
whenever such a CIUS exists. OUTS will hold the CIUS. }

const

MAX1 := ... ;

MAX2 := ... ;

MAX3 := ... ;

type

outlist = sequence of output interactions;

walk1 = **record**

trlist: list of transitions;

pred: predicate; { interpreted predicate }

dest: state; { tail state of the walk}

Sval: V → SymExp;

{ SymExp is the set of all symbolic expressions }

{ Sval(v) is the symbolic expression for the variable v
at the end of a walk of type walk1 }

end;

walk2 = **record**

trlist: list of transitions;

pred: predicate ; { interpreted predicates }

source: state; { starting state }

dest: state; { tail state }

Sval: V → SymExp;

```
        { SymExp is the set of all symbolic expressions }
        surpol:outlist;
{ sequence of surplus ouput interactions at the last transition in the walk }
    end;
var
    walk, twalk: array[1..MAX1] of walk1 ; { walks from state s_k }
    owalk, towalk: array[1..MAX1, 1..MAX2] of walk2 ;
        { walks from other states }
    NW, NTW: integer;
    NOW, NTOW: array[1..MAX3] of integer;
    i,j,L: integer;
    OUTS:sequence of input interactions;
    temp: walk2;
    a : input interaction;
    go, go1, sol: outlist;
    Ipred: predicate; { interpreted predicate }
begin
    NW := 1; NOW[1] := n-1; OUTS := ∅;
    with walk[1] do begin
        dest := s_k;
        trlist := ∅; pred := ∅
        for each variable v ∈ V do
            Sval(v) := a unique symbolic value for v;
    end
    for j := 1 to k-1 do
        with owalk[1,j] do begin
            source := s_j; dest := s_j;
            trlist := ∅; pred := ∅;
```

```
            surpol := ∅
        for each variable v ∈ V do
            Sval(v) := a unique symbolic value for v;
        end
    for j := k to n−1 do
        with owalk[1,j] do begin
            source := s_{j+1}; dest := s_{j+1};
            trlist := ∅; pred := ∅;
            surpol := ∅
        end
    L := 0;
    while (OUTS= ∅ and L < 2n^2) do begin
        L := L+1;
        NTW := 0;
        for i := 1 to NW do
            for each  non-silent t ∈ out(walk[i].dest) do begin
                Obtain Ipred by interpreting t.pred as per the symbolic values
                in walk[i].Sval and a unique symbolic value for each input
                parameter.. in the input interaction of t, if applicable;
                if (Ipred is independent of the symbolic values of the variables
                    at s_k and (walk[i].pred ∧ Ipred is satisfiable))
                then begin
                    NTW := NTW + 1;
                    with twalk[NTW] do begin
                        trlist := walk[i].trlist @ t;
                        pred := walk[i].pred @ (∧ Ipred);
                        Obtain Sval from walk[i].Sval by symbolically and
                        sequentially executing the statements in t ;
```

```
    dest := t.dest;
end;
NTOW[NTW]:=0; a := t.input;
go:= sequence of symbolically interpreted output
interactions in t;
for j := 1 to NOW[i] do begin
    if (a = ∅ or owalk[i,j].surpol = ∅) then
    { otherwise no extension }
        if (owalk[i,j].surpol = ∅) then
            ExtendWalk(a, go, owalk[i,j])
        else if Prefix(go, owalk[i,j].surpol) then begin
                { observe that a=∅ }
            NTOW[NTW] := NTOW[NTW] + 1;
            with towalk[NTW, NTOW[NTW]] do begin
                trlist := owalk[i,j].trlist;
                pred := owalk[i,j].pred;
                source := owalk[i,j].source;
                dest := owalk[i,j].dest;
                surpol := DelPrefix(go, owalk[i,j].surpol);
            { surpol is obtained by removing the prefix
            of owalk[i,j].surpol, which is indistinguishable
                    from go, from owalk[i,j].surpol }
            end
        end
        { a = ∅ at this point }
        else if Prefix(owalk[i,j].surpol, go) then begin
            go1 := DelPrefix(owalk[i,j].surpol, go) ;
            with temp do begin
```

```
                        trlist := owalk[i,j].trlist;
                        pred := owalk[i,j].pred;
                        source := owalk[i,j].source;
                        dest := owalk[i,j].dest;
                        surpol := ∅
                    end
                    ExtendWalk(∅, gol, temp)
                end
            end { for j = 1... }
            if (NTOW[NTW] = 0) then begin
                { Trace(twalk[NTW]) is unique }
                OUTS:= Inseq(twalk[NTW]);
                exit;
            end
        end { if }
    end { for each t } { for i := 1 to NW }
    NW := NTW;
    for i := 1 to NW do begin
        walk[i] := twalk[i];
        NOW[i] := NTOW[i];
        for j := 1 to NOW[i] do
            owalk[i,j] := towalk[i,j]
    end
  end { while }
end.

procedure ExtendWalk(a:input; go:outlist; walk:walk2);
{ checks if walk could be extended for the given input interaction
```

a and outputlist *go*. If so it is extended }

{ *i.e.*, it checks if *walk* could be extended to a walk W1 such that *walk.surpol* followed by the trace along the extended part of W1 subsumes *a* followed by *go* }

```
var
    temp:walk2;
    osurpext, go1:outlist;
    Ipred : predicate;
begin
    if (a = ∅ and go = ∅) then begin { recursion termination }
        NTOW[NTW] := NTOW[NTW] + 1;
        towalk[NTW, NTOW[NTW]] := walk;
    end
    else for each t ∈ out(walk.dest) do begin
        Obtain Ipred from t.pred by interpreting it with the symbolic value
        of the variables as in walk.Sval and the symbolic values for the
        input interaction parameters of t;
        if(walk pred ∧ Ipred is satisfiable) then
            if (t is silent) then begin
                with temp do begin
                    trlist := walk.trlist @ t ;
                    pred := walk.pred @ (∧ Ipred);
                    surpol := walk.surpol;
                    source := walk.source;
                    dest := t.dest;
                end;
                ExtendWalk(a, go, temp)
            end
```

```
            else if (TransMove(a, go, go1, walk, t, osurpext)) then begin
                with temp do begin
                    trlist := walk.trlist @ t ;
                    pred := walk.pred @ (∧ Ipred);
                    surpol := osurpext;
                    source := walk.source;
                    dest := t.dest;
                end;
                ExtendWalk(∅, go1, temp)
            end
        end
    end;

function TransMove(a:input; go:outlist; var go1:outlist;
            ep:walk2; t:transition; var osurpext:outlist): boolean;
{ Checks if ep can be extended by t for the given input interaction a
and outputlist go. If so, TransMove is set to true and go is
adjusted to go1. Surplus output interactions in t after the extension
is returned through osurpext }
var
col, tol: outlist;
begin
    tol:= sequence of symbolically interpreted output
    interactions in t with respect to ep;
    TransMove := false;
    if (a ≠ ∅) then
        if (ep.surpol = ∅) then
            if (t.input = a ) then
```

```
                if Prefix(tol, go) then begin
                    gol := Delprefix(tol,go); osurpext:= ∅;
                    TransMove.= true;
                end
                else if Prefix(go, tol) then begin
                        osurpext := DelPrefix(go, tol); gol := ∅;
                        TransMove := true
                end;
        if (a = ∅) then
            if (t.input = ∅ )then begin
                col := ep.surpol ∘ tol;
                if Prefix(col, go) then begin
                    gol := Delprefix(col, go) ; osurpext := ∅ ;
                    TransMove := true
                end
                else if Prefix(go, col) then begin
                        osurpext := Delprefix(go, col); gol := ∅ ;
                        TransMove := true
                end
            end
    end;
```

The algorithm uses the recursive procedure *ExtendWalk* which in turn uses the function *TransMove*. At the end of the iteration l of the **while** loop in the algorithm *ComputeCIUS*, $walk[i]$, $1 \le i \le NW$, contains a context independent executable walk of length l from s_k, containing only non-silent transitions. $owalk[i,j]$, $1 \le j \le NOW[i]$, is a satisfiable walk from some state other than s_k such that $Trace(walk[i])$ is subsumed by $Trace(owalk[i,j])$. At the iteration $l+1$, every $walk[i]$, $1 \le i \le NW$,

is extended, if possible, by every non-silent transition from $walk[i].dest$, to form a context independent executable walk of length $l+1$. Let $\{twalk[p] | 1 \leq p \leq NTW\}$ be the resulting set of walks of length $l+1$. Let $twalk[p]$ be the walk obtained from $walk[i]$ by extending it with the transition t. Each walk $owalk[i,j]$, $1 \leq j \leq NOW[i]$, is also extended, if possible, by a sequence of transitions so that the resulting walk is satisfiable and the trace of each of the resulting walks subsumes $Trace(twalk[p])$. $\{towalk[p,j] | 1 \leq j \leq NTOW[p]\}$ is the set of all such satisfiable walks from states other than s_k corresponding to $twalk[p]$. This set is also computed in this iteration. If $NTOW[p] = 0$, then clearly OUTS, the input interactions along $twalk[p]$, becomes a CIUS for s_k. After considering all the out transitions from the tail states of $walk[i]$, $i = 1, 2, \ldots, NW$, if it has not found a CIUS, then the algorithm proceeds to the next iteration. Note that as per the termination condition of the **while** loop in the algorithm *ComputeCIUS* the maximum number of iterations is only $2n^2$.

In order to extend the walks $owalk[i,j]$, $1 \leq j \leq NOW[i]$, the algorithm uses the procedure *ExtendWalk* which in turn calls the function *TransMove*. As before, let $twalk[p]$ be the walk obtained from $walk[i]$ by extending it with the transition t. $owalk[i,j]$ is considered for extension only if t is a spontaneous transition or $owalk[i,j]$ has no surplus output interaction sequence with respect to $walk[i]$. For otherwise, the trace of no extension of $owalk[i,j]$ can subsume $Trace(twalk[p])$. Let $owalk[i,j]$ be the walk under consideration. Suppose that $owalk[i,j]$ has a surplus output interaction sequence (with respect to $walk[i]$). Then *ExtendWalk* is invoked to find all satisfiable walk extensions of $owalk[i,j]$ such that their traces subsume $Trace(twalk[p])$. However, suppose that t is spontaneous and $owalk[i,j]$ has a surplus output interaction sequence with respect to $walk[i]$ such that the sequence of ouput interactions in t is a prefix of this surplus sequence. Then, the indistinguishable prefix of the surplus sequence with respect to the output interactions in t is removed. $owalk[i,j]$ with the resulting surplus sequence becomes $towalk[p,k]$ for some k since the trace of this walk subsumes the trace of $twalk[p]$.

Suppose that t is spontaneous and *owalk*[i, j] has surplus output interactions such that it is an indistinguishable prefix of the output interactions in t. Then, the procedure *ExtendWalk(∅, go1, temp)* is invoked , where *go1* is the remaining sequence after removing the indistinguishable prefix of *go* with respect to the surplus output interactions in *owalk*[i,j] and temp is the walk obtained after removing the surplus output interactions in *owalk*[i,j].

We will first explain the function *TransMove*. This function accepts an input interaction (a), a list of output interactions (*go*), the walk (*cp*) to be extended, and the transition (t) to be tried for an extension. Let $S1$ be the trace of t for the symbolic values of the variables as given in *cp*. Let $X1$ be the sequence obtained by prefixing the input interaction in a to the sequence of output interactions in *go* and $X2$ be the sequence obtained by prefixing the surplus sequence in *cp* to $S1$. If $X1$ is an indistinguishable prefix of $X2$, then *TransMove* returns **true**. *TransMove* also returns the surplus output interactions in $X2$ with respect to $X1$ through *osurpext*. If $X2$ is an indistinguishable prefix of $X1$, then *TransMove* also returns **true**. In this case, the surplus output interactions in $X1$ are returned through *go1*.

The recursive procedure *ExtendWalk* accepts an input interaction (a), a list of output interactions (*go*), and a walk (*walk*). It computes all satisfiable walk extensions of *walk* such that the surplus sequence in the walk followed by the trace on the sequence of extended transitions subsumes the trace obtained by prefixing the input interaction in a to the sequence of output interactions in *go*. This is done by extending *walk* by a sequence of transitions starting from the tail state of *walk*. If a as well as *go* are empty, then the procedure declares *walk* itself as an extension and terminates. Otherwise, it considers all the outgoing transitions from the tail state of *walk* such that *walk* followed by the transition is satisfiable. Let t be one such transition. If t is silent, then t is postfixed to *walk* to form a new walk *temp*, and *ExtendWalk* is reinvoked with the same a and *go* along with the walk *temp*. However, if t is not a silent transition, then *TransMove(a, go, go1, walk, t osurpext)* is called to determine

if *walk* is extendible along t for the given a and *go*. If *TransMove* returns **true** further extensions of *walk*@t for the remaining sequence of output interactions obtained through *go*1 and *osurpext* is done by calling *ExtendWalk* with suitable parameters.

We shall now discuss the complexity and correctness of the algorithm.

Lemma 5.1 *Suppose that the EFSM SPEC has no satisfiable closed silent walk of length more than K for some integer $K \geq 1$. Then the depth of recursion of the procedure ExtendWalk during any of its invocations in ComputeCIUS is $(1+O_{max})(K+1)$, where O_{max} is the maximum number of output interactions in any transition in the SPEC. The total number of recursive instantiations of ExtendWalk for a given invocation of this procedure in ComputeCIUS is at most $(d^{out}_{max})^{(1+O_{max})(K+1)+1}$, where d^{out}_{max} is the the maximum number of outgoing transitions including self-loops at any state.*

Proof:

As the EFSM has no closed satisfiable silent walk of length more than K, it is easy to see that either the input interaction in a, or at least one output interaction in *go* will be deleted in every (K+1) successive recursive invocations of *ExtendWalk*. As a result, within $(1+|go|)(K+1)$ successive recursive invocations of ExtendWalk both a and *go* will be empty and so the last instantiation of the procedure terminates. Here, $|go|$ denotes the number of output interactions in *go*. The first part of the lemma follows from the fact that $|go| \leq O_{max}$. Since d^{out}_{max} is the maximum number of times this procedure is called at a particular invocation of *ExtendWalk* (i.e, the maximum number of children in a node in the recursion tree for *ExtendWalk*), the total number of recursive instantiations of the procedure for a given invocation of *ExtendWalk* in *ComputeCIUS* is at most $(d^{out}_{max})^{(1+O_{max})(K+1)+1}$.

□

Theorem 5.2 *Suppose that the EFSM SPEC has no satisfiable closed silent walk of length more than K for some integer $K \geq 1$. Then the algorithm ComputeCIUS takes*

at most $2(d^{out}_{max})^{2n^2+1} + (n-1)(d^{out}_{max})^{(2n^2+2)+(1+O_{max})(K+1)(2n^2+1)}$ *steps, where* n, O_{max} *and* d^{out}_{max} *are the number of states, the maximum number of output interactions in any transition in the EFSM, and the maximum number of outgoing transitions including self-loops in any state, respectively.*

Proof

Let $\theta = (d^{out}_{max})^{(1+O_{max})(K+1)}$. It is enough to analyze the time complexity of the **while** loop of the algorithm *ComputeCIUS* since it dominates every other step. In the first iteration of this loop it computes a maximum of d^{out}_{max} walks from the state s_k. In this iteration, *ExtendWalk* is called at most $(n-1)d^{out}_{max}$ times. The last **for** loop within the **while** loop is also executed at most d^{out}_{max} times. Thus, this iteration takes at most $2d^{out}_{max} + (n-1)(d^{out}_{max})^2\theta$ steps. In the second iteration, the number of walks from s_k which are considered for feasibility is at most $(d^{out}_{max})^2$. The maximum number of times *ExtendWalk* is called is $(n-1)(d^{out}_{max})^2\theta$. Also, the last **for** loop in the **while** loop is performed at most $(d^{out}_{max})^2$ times. Therefore, this iteration will take at most $2(d^{out}_{max})^2 + (n-1)(d^{out}_{max})^3\theta^2$ steps In general, the maximum number of steps taken in the ith iteration of the **while** loop is $2(d^{out}_{max})^i + (n-1)(d^{out}_{max})^{i+1}\theta^i$. As this loop is performed at most $2n^2$ times , the algorithm takes at most $2(d^{out}_{max})^{2n^2+1} + (n-1)d^{out}_{max}[d^{out}_{max}\theta]^{2n^2+1}$ steps.

□

Note that only a higher level complexity of the above algorithm is given in terms of the number of times variaous basic steps are executed. The executions of some of these steps may themselves be complex. For instance, finding whether a given walk is context independent and executable is considered to be a part of a single step in our analysis.

Theorem 5.3 *Suppose that the EFSM SPEC has at least one walk W of length at most* $2n^2$ *at* $s_k \in S$ *such that (i) W is a context independent executable walk having*

only non-silent transitions, and (ii) Trace(W) is distinguishable from the trace of any satisfiable walk from any state other than s_k. Then, the algorithm ComputeCIUS *returns the input sequence U along a shortest walk at s_k which satisfies (i) and (ii). U is a CIUS of s_k.*

Proof:

The first iteration of the **while** loop in ComputeUIS considers all the walks from s_k of unit length and checks if any of them satisfies the conditions (i) and (ii) of the theorem. Condition (ii) is in fact checked in the third **for** loop within the **while** loop by invoking the procedure *ExtendWalk*. If the algorithm finds a walk satisfying (i) and (ii), then it terminates after returning the input sequence along this walk. Otherwise, it considers all the walks from s_k of length 2 and it proceeds as above. This process is repeated until it returns an input along a walk satisfying (i) and (ii) or it has considered all the walks of length at most $2n^2$ at s_k. Thus, if M has walks satisfying conditions (i) and (ii) of the theorem, then it returns the input sequence along one such shortest walk.

Suppose that W is a walk satisfying conditions (i) and (ii). Then, (ii) implies that if W' is a walk from s_j, $j \neq k$, such that W' may be executable for some context and for Inseq(W), then $Trace(W)$ is distinguishable from $Trace(W')$. Therefore $Inseq(W)$ is a CIUS for s_k.

□

5.3.1 An Illustration

Let us find a CIUS for the state s_4 of the EFSM given in Figure 5.1. Initially, NW=1, walk[1].dest=s_4, walk[1].trlist=∅, walk[1].pred=∅, walk[1].Sval(opt)=opt_4, walk[1]. Sval(R_credit)=R_credit_4, walk[1].Sval(S_credit)=S_credit_4, walk[1].Sval(TRsq)= $TRsq_4$, walk[1].Sval(TSsq)=$TSsq_4$, NOW[1]=n-1, owalk[1,1].source=s_1, owalk[1,1]. dest=s_1, owalk[1,1].trlist=∅, owalk[1,1].pred=∅, owalk[1,1].Sval(opt)=opt_1, owalk[1,1].

Sval(R_credit)=R_credit_1, owalk[1,1].Sval(S_credit)=S_credit_1, owalk[1,1]. Sval(TRsq)=$TRsq_1$, owalk[1,1].Sval(TSsq)=$TSsq_1$. Similarly, we know the initial values for the null walks owalk[1,2], owalk[1,3], owalk[1,4], and owalk[1,5] at states s_1,s_2, s_3, s_5, s_6, respectively. Without loss of generality, assume that the transition $t17$ from s_4 is chosen as the first outgoing transition from s_4 for executing the second **for** loop within the **while** loop. As it has no associated predicate, the transition is automatically context independent and executable. Therefore, $twalk[1]$ is defined for the walk $t17$. That is, $twalk[1].trlist = t17$, $twalk[1].pred = \emptyset$, and $twalk[1].dest = s_5$. twalk[1].Sval remains the same as walk[1].Sval since there is no change of context in $t17$. Also, a and go are set to the input interaction U?TDISreq and the output interaction N!TrDR('User initiated', false), respectively. NTOW[1] is initialized with zero In the third **for** loop within the **while** loop, the walk $owalk[1,1]$ is selected first. The procedure *ExtendWalk* is invoked with the parameters a, go, and $owalk[1,1]$, in that order, to check if it can be extended to satisfiable walks whose traces subsume $Trace(twalk[1])$. The **for** loop of *ExtendWalk* is executed for every outgoing transition t from s_1. Therefore, *TransMove(a,go,go1,owalk[1,1], t, osurpext)* is invoked for every transition t from s_1. The procedure *TransMove* returns **true** only for the transition $t23$ since its trace U?TDISreq is an indistinguishable prefix of the sequence U?TDISreq N!TrDR('User initiated', false) - the input interaction in a followed by the output interaction in go. Therefore, $owalk[1,1]$ is extended to a walk $temp$ by postfixing $t23$ to $owalk[1,1]$. Note that $go1$ will hold the output interaction N!TrDR('User initiated', false) after the execution of *TransMove* for $t23$. The procedure *ExtendWalk* is invoked with the parameters, $\emptyset$, $go1$, and $temp$, in that order. *ExtendWalk*, in turn, calls *TransMove* for every outgoing transition from s_1. But at this time, each invocation of *TransMove* returns **false** since s_1 has no spontaneous transition and $go1$ has an output interaction. Therefore, there is no walk starting from s_1 whose trace subsumes $Trace(twalk[1])$. The complete execution of the third **for** loop within the **while** loop of *ComputeCIUS* yields the same result for the states s_2, s_3, s_5, and s_6.

State	CIUS	Transition Seq.
s_1	U?TCONreq(dst_add, prop_opt)@ N?TrDR(disc_reason, switch)	t1 t5
s_2	N?TrDR(disc_reason, switch)	t5
s_3	U?TDISreq	t7
s_4	U?TDISreq	t17
s_5	N?TrDR(disc_reason, switch)	t21
s_6	N?terminated	t19

Table 5.4: CIUSs for the states in the AP-module

That is, NTOW[1] remains zero after the execution of the third **for** loop within the **while** loop. Therefore U?TDISreq - the input sequence along *twalk*[1] - becomes a CIUS of s_4.

The CIUSs for the states s_2, s_3, s_4, s_5 and s_6 of the AP-module thus obtained are presented in Table 5.4. The algorithm terminates with a single iteration of the **while** loop in *ComputeCIUS* for any of the above states. Also the maximum depth of recursion for a given invocation of *ExtendWalk* in the main algorithm is only 2. Though the given algorithm is exponential, for real life protocols which have CIUSs for all the states, the algorithm is expected to terminate within a few iterations, as in the above EFSM. As per our assumption, the cyclic CIUS for s_1 as shown in the table is already known. The transitions($t1$ and $t5$) which are part of this CIUS at s_1 are also known to be fault free in the IUT. The parameters in the CIUSs have to be instantiated with certain feasible values.

We have also found that a few other protocols such as a class 0 transport protocol as specified in [UY91] and the abracadabra protocol as specified in [Tur93] have a CIUS for every state. The EFSM representation of the class 0 transport protocol has 4 states and 14 core transitions. The shortest CIUS walk for the initial state is of length 2. All other states have a CIUS walk of unit length. The EFSM representation of the abracadabra protocol has 5 states and 30 core transitions. It has a CIUS set such that the maximum length of a CIUS walk for a CIUS in this set is only 2.

5.4 Data Flow Graph Manipulation

Let W be an executable walk from the initial state of the SPEC. Let t be a transition in W. We know that the data flow graph DFG[t, W] for t with respect to W has one subgraph for each definition in t. As defined in Section 5.2.2, a subgraph of DFG[t, W] for a variable/parameter defined in t represents the information flow chains along W from the location in t where the variable/parameter is defined to variables which are used in locations in the subwalk of W starting from t. In this section, we develop different procedures for constructing and manipulating the data flow graphs.

Each walk starting from the initial state is represented by a record as described below.

```
type
    trwalk = sequence of transition;
    walk = record
        trlist: trwalk; { sequence of transitions in the walk }
        pred: predicate ;
            { conjunction of the interpreted predicates along the walk }
        dest: state; { tail state of the walk }
        def: set of variables; { set of variables and parameters defined }
        puse: set of variables;
            { set of variables and parameters used in predicates }
        Sval: def → SymExp;
{symbolic values of variables/parameters in def at the end of the walk};
        recentdef: def → (transition, 'I'/assignment_statement_no)
            { walk.recentdef(v) is an ordered pair of transition and 'I'
    signifying the input interaction of the transition or the statement no.
            of the assignment statement where v is most recently defined}
    end
```

The array DFWALK as declared below is a global variable of our *SelectTestTour* algorithm (to be discussed in Section 5.5.1) which calls the procedures described in this section.

DFWALK:**array**[1..|$\mathcal{D}$|] **of** walk

In order to simplify our explanation, we refer to an element of the array corresponding to the def-use pair D as DFWALK[D]. We design and present the procedures for the DFG manipulation in a bottom-up fashion.

Our first procedure *PredExtendGraph* is for processing a predicate in a given transition. The procedure accepts a walk $W2$, a transition $t2$, where $t2$ is the last transition in $W2$, and a partial subgraph G of a data flow graph for a transition $t3 \neq t2$ in $W2$ with respect to $W2$. Let G correspond to a variable/parameter u defined at $t3$. Note that G is partial since it does not have the data flow information corresponding to the use of the variables in $t2$. *PredExtendGraph* extends the graph G if the value of u is eventually used in the predicate of $t2$. The procedure is given below.

```
procedure PredExtendGraph(G:graph; t2:transition; W2:walk);
begin
   inlevel2 := false; inlevel3 := false;
   Let (t1,x1,u) be the root node of G; { x1 = 'I' or assignment stmt. no. }
   for each variable v used in t2.pred do begin
      Let (t, c) = W2.recentdef(v);
      if ((t,c,v) is the root node of G) then begin { (t,c,v)= t1,x1,u) }
         if (not inlevel2) then begin
            Create a p-node (t2,P) at level 2 in G; inlevel2 := true;
         end;
```

```
        Add an edge from (t1,x1,u)_1 to (t2,P)_2 in G;
        if (D = (t1.x1, t2.P)(u)∈ 𝒟 is not yet covered) then begin
            Mark D as covered;
            Obtain DFWALK[D] by appending U(j) to W2,
                where s_j= t2.dest & U(j) is the CIUS walk for U_j;
        end
    end;
    if ((t,c,v) is a node at level 2 in G) then begin
        if (not inlevel3) then begin
            Create a p-node (t2,P) at level 3 in G; inlevel3 := true;
        end;
        Add an edge from (t,c,v)_2 to (t2,P)_3 in G;
        if (D = (t1.x1, t.c)(u)∈ 𝒟 is not yet covered) then begin
            Mark D as covered;
            Obtain DFWALK[D] by appending U(j) to W2,
                where s_j= t2.dest & U(j) is the CIUS walk for U_j;
        end
    end;
    if ((t,c,v) is a node at level 3 in G) then begin
        if (not inlevel3) then begin
            Create a p-node (t2,P) at level 3 in G; inlevel3 := true;
        end;
        for each incoming edge e to (t, c, v) do begin
            Let (t', c', v')_2 be the starting node of e;
            Add an edge from (t', c', v')_2 to (t2,P)_3 in G;
            if (D = (t1.x1, t'.c')(u)∈ 𝒟 is not yet covered) then begin
                Mark D as covered;
                Obtain DFWALK[D] by appending U(j) to W2,
```

where s_j= t2.dest & U(j) is the CIUS walk for U_j;

end

end

end

end { **for each** variable v }

end { **PredExtendGraph** }

Let $(t1, x1, u)$ be the root node of G. Here, $x1$ is either 'I' which denotes the input interaction or a computation statement number in $t1$. Let *pred* be the predicate of $t2$ and v be a variable used in *pred*. Let $t.c$ be the location in $W2$ where v is most recently defined, where t is a transition in $W2$ and c is the statement number of the assignment statement in which v is defined. If $v = u$ and $t1.x1$ is the same as $t.c$, then a p-node $(t2, P)$ is added as a level 2 node in G, provided $(t2, P)_2$ is not already present, and an edge from $(t1, x1, u)_1$ to $(t2, P)_2$ is also added. Let W be the walk obtained by suffixing $W2$ with the walk $U(j)$ from s_j. Recall that $U(j)$ is the CIUS walk for U_j. Clearly, W is a preamble walk for covering the def-use pair $D = (t1.x1, t2.P)(u)$. Note that the CIUS walk extension is required for observing the p-use, as discussed earlier. If D is not already covered, then it is marked covered and W is stored in DFWALK[D]. Suppose that (t, c, v) is a level 2 node in G, then a p-node $(t2, P)$ is added at level 3 in G, provided $(t2, P)_2$ is not already present, and an edge from $(t, c, v)_2$ to $(t2, P)_3$ is added to G. If $D = (t1.x1, t.c)(u)$ is not already covered, then it is marked covered and W is stored in DFWALK[D] since it is a preamble walk for D. However, if (t, c, v) is a node in the third level of G, then a p-node $(t2, P)$ is added at the same level, provided $(t2, P)_3$ is not already present. An edge is added to $(t2, P)_3$ from the starting vertex $(t', c', v')_2$ of each incoming edge of $(t, c, v)_3$. Also, if the def-use pair $D = (t1.x1, t'.c')(u)$ is not yet covered, then it is marked as covered and W is stored in DFWALK[D].

Our next procedure is *StmtExtendGraph*. This procedure accepts a walk $W2$,

the transition $t2$ which is the last transition in $W2$, an assignment statement $c2$ in $t2$, and a partial subgraph G of DFG[$t3$, $W2$], for some transition $t3$ in $W2$. It extends G with respect to the uses of variables/parameters in $c2$ of $t2$. The detailed description of the procedure is given below. Explanation of the procedure is omitted since it is very similar to *PredExtendGraph*. Observe that, unlike *PredExtendGraph*, *StmtExtendGraph* does not check if $W2$ covers any def-use pair for the def-use ob criterion.

procedure *StmtExtendGraph*(G:graph; t2:transition; c2:statement; W2:walk);
begin
  Let c2: w := exp be the statement;
  inlevel2 := **false**; inlevel3 := **false**;
  for each variable or parameter v used in exp **do begin**
  Let (t, x) = W2.recentdef(v); { x = 'I' or computation statement no. }
    if ((t, x,v) is the root node of G) **then begin**
      if (**not** inlevel2) **then begin**
        Create a c-node (t2,c2,w) at level 2 in G; inlevel2 := **true**;
      end
      Add an edge from $(t,x,v)_1$ to $(t2,c2,w)_2$ in G;
    end;
    if ((t, x,v) is a c-node at level 2 in G) **then begin**
            { x has to be a computation stmt. no }
      if (**not** inlevel3) **then begin**
        Create a c-node (t2,c2,w) at level 3 in G; inlevel3 := **true**;
      end
      Add an edge from $(t, x,v)_2$ to $(t2,c2,w)_3$ in G;
    end;
    if ((t, x,v) is a c-node at level 3 in G) **then begin**
            { x has to be a computation stmt no. }

```
            if (not inlevel3) then begin
                Create a c-node (t2,c2,w) at level 3 in G; inlevel3 := true;
            end
            for each incoming edge e to (t, x,v)3 do begin
                Let (t', c', v')2 be the starting node of e;
                Add an edge from (t', c', v')2 to (t2,c2,w)3 in G;
            end
        end
    end { for each variable or parameter v }
end {StmtExtendGraph }
```

The procedure *OutputExtendGraph* is also similar to *PredExtendGraph*. It accepts a walk $W2$, a transition $t2$ which is the last transition in $W2$, the statement number of an output statement in $t2$, and a partial subgraph of a DFG of some transition $t3$ in $W2$. It extends G by adding nodes and edges necessary to represent the information flow chains along $W2$ from the variable/parameter which corresponds to the root node of G to the variables used in the output statement. This procedure also checks if $W2$ is a preamble walk for a def-use pair along $W2$ where the definition corresponds to the root node of G. The formal description of *OutputExtendGraph* is as follows.

```
procedure OutputExtendGraph(G:graph; t2:transition; c2:statement; W2:walk);
begin
    inlevel2 := false; inlevel3 := false;
    Let (t1,x1,u) be the root node of G; { x1 = 'I' or computation stmt. no. }
    Let c2: ip.oi(parlist);
    for each variable v used in parlist do begin
        Let (t, c) = W2.recentdef(v);
        if ((t,c,v) is the root node of G) then begin
```

```
    if (not inlevel2) then begin
        Create a o-node (t2,c2) at level 2 in G; inlevel2 := true;
    end;
    Add an edge from (t,c,v)_1 to (t2,c2)_2 in G;
    if (D = (t1.x1, t2.c2)(u) ∈𝒟 and it is not yet covered)
    then begin
        Mark D as covered; DFWALK[D] := W2;
    end
end;
if ((t,c,v) is a node at level 2 in G) then begin
    if (not inlevel3) then begin
        Create a o-node (t2,c2) at level 3 in G; inlevel3 := true;
    end;
    Add an edge from (t,c,v)_2 to (t2,c2)_3 in G;
    if (D = (t1.x1, t.c)(u) ∈𝒟 and it is not yet covered)
    then begin
        Mark D as covered; DFWALK[D] := W2;
    end
end;
if ((t,c,v) is a node at level 3 in G) then begin
    if (not inlevel3) then begin
        Create a o-node (t2,c2) at level 3 in G; inlevel3 := true;
    end;
    for each incoming edge e to (t, c, v) do begin
        Let (t',c',v')_2 be the starting node of e;
        Add an edge from (t',c',v')_2 to (t2,c2)_3 in G;
        if (D = (t1.x1, t'.c')(u) ∈𝒟 and it is not yet covered)
        then begin
```

Mark D as covered; DFWALK[D] := W2;

end

end

end

end { **for each** variable v }

end { *OutputExtendGraph* }

We shall now describe procedure *ExtendDFG*. This procedure accepts a walk $W1$, a transition $t1$ in $W1$, and a transition $t2$ which starts from the tail state of $W1$ and computes DFG[$t1, W1\ t2$], the data flow graph for $t1$ with respect to the walk $W1\ t2$. *ExtendDFG* achieves this by extending the already known data flow graph DFG[$t1, W1$] as per the data flows along $W1\ t2$ from the variables/parameters defined in $t1$ to the variables used in the predicates and the statements in $t2$. Let $W2 = W1\ t2$. Let us assume that the set of def-use pairs in $\mathcal{D}$ which are yet to be covered is known at the starting of the procedure. After copying DFG[$t1, W1$] into DFG[$t1, W2$], it manipulates each subgraph in DFG[$t1, W2$] with respect to the variables used in the predicate of $t2$. It calls the procedure *PredExtendGraph* for this purpose. It then sequentially selects every statement in the computation block of $t2$, and updates every subgraph in DFG[$t1, W2$] by considering all the variables/parameters used in the statement. If it is an assignment statement, then *ExtendDFG* calls the procedure *StmtExtendGraph*; otherwise it invokes *OutputExtendGraph* for updating a given subgraph. The formal description is given below.

procedure *ExtendDFG*(t1:transition;W1:walk;t2:transition);

{ DFG[t1,W] → DFG[t1,W2] where W2.trlist = W1.trlist@t2 }

{ Mark the def-use associations which are covered by DFG[t1,W2] }

begin

Let W2 be the walk obtained by appending t2 to the walk W1;

```
    DFG[t1,W1] := DFG[t1,W2];
    for each subgraph G in DFG[t1,W2] do
        PredExtendGraph(G, t2, W2);
    { Sequentially process the statements in the compute-block of t2 }
    for each statement c2 in the compute-block of t2 do
        for each subgraph G in DFG[t1,W2] do
            if (c2 is an assignment statement) then
                StmtExtendGraph(G, t2, c2, W2)
            else OutputExtendGraph(G, t2, c2, W2);
end; { ExtendDFG }
```

Our final procedure for DFG manipulation is *ConstructDFG* for constructing DFG[*t*, *t*] for every transition *t* in SPEC. The procedure is described below. It is very similar to *ExtendDFG*. All the DFG manipulation procedures are illustrated in Section 5.6 while generating test tours for the SPEC given in Figure 5.1.

```
procedure ConstructDFG(t:transition);
begin
    DFG[t, t] := ∅;
    if (t is not a spontaneous transition) then begin
        Let ip.i(parlist) be the input interaction;
        for each parameter v in parlist do
            create an i-node (t,I,v) as a root node in DFG[t, t];
    end
    {Sequentially process the statements in the computation block of t }
    for each statement c in the compute-block of t do begin
        for each subgraph G in DFG[t, t] do
            if (c is an assignment statement) then
```

```
            StmtExtendGraph(G, t, c, t)
        else OutputExtendGraph(G, t, c, t);
    if (c is an assignment statement) then begin
        Let c: w := exp be the statement;
        Create a c-node (t,c,w) as a root node in DFG[t, t]
    end
  end
end { ConstructDFG }
```

In the above procedures for manipulating data flow graphs we assume that the addition of edges in a DFG is done such that there are no duplicate edges between any given pair of nodes.

5.5 Automatic Test Case Generation

5.5.1 The Two-Phase Algorithm

We have already established the trans-CIUS-set criterion for the control flow testing and the def-use-ob criterion for data flow testing. Here, we develop an algorithm for generating a set of test tours for covering the above criteria for a given EFSM SPEC. The algorithm has two phases. The first phase constructs a preamble walk W_1 for every transition in the SPEC. Recall that an executable walk W starting from the initial state is called a preamble walk for t if Wt is also executable. This phase also computes preamble walks for the feasible def-use pairs in $\mathcal{D}$. Refer to Section 5.2.2 for the definition of a preamble walk of a def-use pair.

In the second phase, all preambles computed in the first phase are completed into executable tours. These tours are in fact the required set of tours for the coverage criteria. The step-wise description of the first phase of the algorithm is given below.

Phase I

Input: EFSM SPEC, CIUS-set $\mathcal{U} = \{U_j \mid 1 \leq j \leq n\}$, Def-use pairs set $\mathcal{D}$. A fixed positive integer K_1.

Output: UFset: set of preamble walks for the coverage criteria.

<u>Step 0</u> { Data flow graphs initialization }

(i) Construct the data flow graph of each transition with respect to itself.

<u>Step 1</u> { null walk initialization }

(i) Let P be a null walk at s_1; Let $\mathcal{P} = \{P\}$.

<u>Step 2</u> { ith iteration of this step computes the set of all executable walks of length i starting from s_1. They are computed from the executable walks of length $i-1$ computed in the previous iteration. This step marks all transitions and def-use pairs covered by the new walks.}

(i) Let $\mathcal{T} = \emptyset$.

(ii) Do Step 2.1 for each $P \in \mathcal{P}$ and for each outgoing transition t from the tail state of P.

(iii) If all the transitions in SPEC are covered for control flow and all the def-use pairs in $\mathcal{D}$ are covered for data flow or the number of iterations of Step 2 exceeds K_1, a fixed positive integer, then proceed to Step 3.

(iv) Consider $\mathcal{T}$ as $\mathcal{P}$ and repeat Step 2.

<u>Step 3</u> { For every transition t, and for every CIUS, postfix t followed by the walk along the CIUS to the preamble walk. Also collect the resulting walks for the transitions as well as the preamble walks for the def-use pairs into $UFset$. }

(i) Let both $CFset$ and $DFset$ to be the empty set.

(ii) For each transition t covered by Step 2 and for each CIUS U_k, $1 \leq k \leq n$, add $W @ t @ Ewalk(j, U_k, C)$ to CFset, where W is the preamble walk computed for t, s_j is the tail state of t and C is the context after executing $W @ t$.

(iii) For each def-use pair $D \in \mathcal{D}$ covered by Step 2, add the preamble walk for D computed in Step 2 to $DFset$.

(iv) Let $UFset = CFset \cup DFset$. Delete each walk $W \in UFset$ such that W is a prefix of some other walk in $UFset$.

(v) Stop.

Step 2.1

(i) Let $Q = P\ t$.

(ii) If Q is executable and t is not yet covered for control flow then mark t as covered and take P as the preamble walk for t.

(iii) If Q is executable and either t is not a self-loop or t has at least one assignment statement in its computation block then add Q to $\mathcal{T}$.

(iv) If Q is executable then do Step 2.1.1.

Step 2.1.1

(i) For each $t' \in P$, (a)compute DFG$[t', Q]$ from DFG$[t', P]$, (b) Mark all the def-use pairs covered by Q, and (c) Construct an appropriate preamble walk for each such pair.

(ii) Consider DFG$[t, t]$ to be DFG$[t, Q]$.

The first phase starts by constructing DFG$[t, t]$, for every transition t in SPEC. As we shall see in the detailed algorithm, this can be done using the procedure

ConstructDFG. In Step 1, the set $\mathcal{P}$ is initialized with the singleton set containing the null walk at s_1. Starting from the initial state, Step 2 traverses SPEC in a breadth-first fashion, in order to compute the preambles for each transition in SPEC and for each feasible def use pair in $\mathcal{D}$. At the starting of the kth iteration of Step 2, $k \geq 1$, $\mathcal{P}$ consists of the set of all executable walks of length $k-1$ which start from the initial state. At this instant, the data flow graphs DFG$[t, W]$, for all $W \in \mathcal{P}$, and for all transitions t in W, are also known. The kth iteration of this step computes the set of all executable walks of length k by extending the walks in $\mathcal{P}$ by single transitions. The executability of the extended walk is checked only with respect to the last transition since the rest of the walk is known to be executable at this point. This reduces the complexity of the feasibility problem to a great extent. We shall return to this feasibility problem in Section 5.7.

For each walk $P \in \mathcal{P}$ and for each transition t from the tail state of P, Step 2.1 checks if the walk Q obtained by postfixing t to P is executable. If so, then Q is added to $\mathcal{T}$ provided either t leads to a state other than the tail state of P or some context is set by t. Also, if Q is executable and if t is not yet covered, then t is marked as covered and P becomes the preamble walk for t. When Q is executable, Step 2.1 uses Step 2.1.1 for computing the data flow graphs pertaining to Q, for determining the def-use pairs in $\mathcal{D}$ covered by Q, and for selecting a preamble walk for every def-use pair covered by Q. As given in the detailed description of the algorithm, Step 2.1.1 can be achieved using the procedure *ExtendDFG* which extends DFG$[t', P]$ to DFG$[t', Q]$, for all t' in P.

Step 2 is repeated until the preambles for all transitions in SPEC are computed and all def-use pairs in $\mathcal{D}$ are covered or the number of iterations of Step 2 exceeds a fixed positive integer K_1. K_1 depends on the SPEC. It has to be chosen in such a way that the preambles for all the transitions are computed in K_1 iterations of Step 2. Recall that, for every transition, the SPEC is assumed to have a feasible walk from the initial state such that the transition is executable for the resulting context.

Therefore, the preamble for all the transitions are computable in a finite number of iterations of Step 2. Observe that some of the def-use pairs in $\mathcal{D}$ may not be feasible. Also, the problem of finding whether a given pair is feasible or not is undecidable. If $\mathcal{D}$ has some infeasible pairs, then this phase terminates after K_1 iterations of Step 2.

We know that, for each transition in SPEC, the trans-CIUS-set criterion requires the traversal of the transition followed by each CIUS $U_j, 1 \leq j \leq n$. Suppose that W is a preamble walk for t computed in Step 2. Then, for each CIUS U_k, $1 \leq k \leq n$, Step 3 computes a walk which extends Wt by postfixing it with $Ewalk(k, U_k, C)$, where C is the context set by Wt. Note that $Ewalk(k, U_k, C)$ is the unique walk executable from s_k for the given context C and the instantiated input sequence U_k. These walks are computed for every transition covered in Step 2 and they are stored in the set $CFset$. Similarly, the set of all walks computed in Step 2 for covering the def-use pairs are stored in the set called $DFset$. Note that the walks for the def-use pairs covered in Step 0 are not added to DFset since their corresponding walks are single transitions. Therefore they are automatically covered by the trans-CIUS-set criterion. In order to minimize the number of test tours, duplicate walks and walks which are prefixes of other walks in $CFset \cup DFset$ are removed; $UFset$ is the resulting set of walks.

Phase II described below is essentially for completing each walk in $UFset$ into an executable tour.

Phase II

Input: The EFSM SPEC, UFset returned by Phase I

Output: UFTourset, a set of tours for the selection criteria

Step 1 { Initialization }

(i) Let P be a null walk at s_1; Let $\mathcal{P} = \{P\}$.

(ii) Let $UFTourset$ be the empty set.

Step 2 { ith iteration of this step computes the set $\mathcal{T}$ of all satisfiable walks of length i ending at s_1. The set of all preambles in *UFset*, which are executable in conjunction with a walk in $\mathcal{T}$ which starts at the tail state of the preambles, are declared to be covered by the tour obtained by prefixing the preamble to the walk. }

(i) Let $\mathcal{T}$ be the empty set.

(ii) Do Step 2.1 for each $P \in \mathcal{P}$ and for each transition t starting from a state other than s_1 and ending at the starting state of P.

(iii) If all the walks in *UFset* are covered, then stop.

(iv) Consider $\mathcal{T}$ as $\mathcal{P}$ and repeat Step 2.

Step 2.1

(i) Let $Q = t\ P$.

(ii) If Q is satisfiable, then add Q to $\mathcal{T}$.

(iii) Do Step 2.1.1 for each walk W in *UFset* such that $W\ Q$ is a tour provided Q is satisfiable.

Step 2.1.1

(i) If $W\ Q$ is executable then Add $W\ Q\ Walk(1, U_1)$ to *UFTourset* and mark W as covered.

Starting from the initial state (s_1), Phase II traverses the transitions, in the reverse direction, in a breadth-first fashion and complete the walks in *UFset* into executable tours. The first iteration of the second step starts after initializing the set $\mathcal{P}$ to the singleton set containing the null walk starting at the initial state and initializing *UFTourset* to the empty set. At the starting of the kth iteration of Step 2, $k \geq 1$, $\mathcal{P}$ contains the set of all satisfiable walks of length $k-1$ such that each walk starts at some state and ends at s_1. At the kth iteration of this step, Phase II computes

the set $\mathcal{T}$ of all satisfiable walks of length k using walks in $\mathcal{P}$. For extending each $P \in \mathcal{P}$, it considers every incoming transition t at the starting state of P such that t does not start from the initial state s_1. Step 2 invokes Step 2.1 for computing the set of all preambles in *UFwalk* which can be completed into executable tours by postfixing them with $t\ P$. This step first determines if the walk Q obtained by prefixing t to P is satisfiable. If so, then Q is added to $\mathcal{T}$, which was initialized to the empty set at the starting of the current iteration of Step 2. Also, if Q is found to be satisfiable, then for each uncovered walk $W \in UFset$ which ends at the starting state of Q, this step also checks if the tour $W\ Q$ is executable. Step 2.1 uses Step 2.1.1 for this purpose. If Step 2.1.1 finds that $W\ Q$ is executable, then W is marked as covered and the tour $W\ Q\ Walk(1, U_1)$ is added to *UFTourset*. Note that $Walk(1, U_1)$ is postfixed to WQ in order to confirm the initial state. $W\ Q\ Walk(1, U_1)$ is the required test tour for all the def-use pairs for which a prefix of W is a preamble, and it is also a member of the set of required tours for covering any transition for which W is a preamble. Step 2 is repeated until all the walks in *UFset* are covered. $\mathcal{T}$ at the end of the current iteration is considered as $\mathcal{P}$ for the next iteration. Note that Phase II successfully completes the walks in *UFset* into tours and terminates in a finite number of steps since the initial state is reachable from every other state with every valid context.

For the sake of completeness, a formal description of the algorithm is provided below.

Algorithm *SelectTestTour* (M:EFSM, $\mathcal{U}$:CIUS-set; $\mathcal{D}$:all-use-set, K_1:integer);
{ It computes a set of test tours for covering
the data flow and control flow in the EFSM, M }
const
N = ... { number of states }
type
trwalk = sequence of transition;
walk = **record**

```
        trlist: trwalk: { sequence of transitions in the walk }
        pred: predicate ;
            { conjunction of the interpreted predicates along the walk }
        dest: state; { tail state of the walk }
        def: set of variables; { set of variables and parameters defined }
        puse: set of variables;
            { set of variables and parameters used in predicates }
        Sval: def → SymExp;
{ symbolic values of variables/parameters in def at the end of the walk } ;
        recentdef: def → (transition, 'I'/assignment_statement_no)
            { walk.recentdef(v) is an ordered pair of transitio: and 'I'
    signifying the input interaction of the transition or the statement no.
            of the assignment statement where v is most recently defined}
    end;
var
t: transition; p: integer;
CFPRE: array[1..|T|] of walk; { an element of this array
        corresponding to transition t is referred to as CFPRE[t]
DFWALK:array[1..|D|] of walk; { an element of this array
        corresponding to the def-use pair D is referred to as DFWALK[D]
CFWALK:array[1..|T|, 1..N] of walk; { an element of this array corresponding
        to transition t and for the state s_j is referred to as CFWALK[t,j]
UFWALK, UFTOUR: set of walk;
begin
    { PHASE I }
    for each t ∈ T do
        ContructDFG(t);
    Delete all the def-use pairs in D which are covered by single transitions;
```

{ def-use-ob criterion for these pairs will be satisfied by the
trans-CIUS-set criterion }

ComputePreamble;

for each transition t in the SPEC **do**

  for p := 1 **to** N **do begin**

    CFWALK[t, p] := CFPRE[t]@t@Ewalk(t.dest, U_p , C), where
    C is the context set after executing CFPRE[t]@t;

    UFWALK := UFWALK ∪ CFWALK[t, p]

  end;

for each covered def-use pair D in $\mathcal{D}$ do

  UFWALK := UFWALK ∪ DFWALK[D];

Delete each W∈UFWALK s.t. W is a subwalk of
  some other walk in UFWALK;

{ PHASE II }

ConstructTour;

end.

procedure *ComputePreamble*

var

  $\mathcal{P}$, $\mathcal{T}$: set of walk;

  P, Q:walk;

  t', t:transition;

  v: variable;

begin

  with P **do begin**

    dest := s_1; pred := ∅; trlist := ∅;

    def := ∅; puse:= ∅; Sval := ∅;

  end;

```
L:=0; 𝒫 := {P};
repeat
    𝒯 := ∅;
    for each P ∈ 𝒫 do
        for each out transition t from P.dest do
            if (every variable used in t is defined in P@t prior to its use)
                    then begin
                Interpret t.pred with respect to P.Sval and t.input;
                if (P.pred ∧ t.pred is satisfiable) then begin
                    if (t is not yet covered) then begin
                        Mark t as covered;
                        CFPRE[t] := P
                    end
                    Compute Q.Sval by updating P.Sval
                    with respect to the symbolic execution of t;
                    with Q do begin
                        trlist := P.trlist@t; pred := P.pred@(∧t.pred);
                        def := P.def ∪ t.def; puse := P.puse ∪ t.puse;
                        dest := t.dest;
                    end
                    Compute Q.recentdef(v) ∀ v ∈ Q.def;
                    if (t is not a self-loop or t has at least one assignment
                        statement in its computation block) then
                        Add Q to 𝒯;
                    for eact t' in P.trlist do
                        ExtendDFG(t', P, t);
                    DFG[t, Q] := DFG[t, t];
                end
```

```
            end;
        𝒫 := 𝒯; L := L + 1;
    until((all transitions in T are covered and
        (all def-use pairs in 𝒟 are covered)) or (L > K₁))
end;

procedure ContructTour;
var
    𝒫, 𝒯: set of walk;
    ℱ: set of trwalk;
    U, V: trwalk;
    P, Q, W: walk;
    t, v: transition;
begin
    for each W ∈ UFWALK do
        if W is a tour then
            Delete W from UFWALK and add W to UFTOUR;
    with P do begin
        dest := s₁; pred := ∅; trlist := ∅;
        def := ∅; puse := ∅; Sval := ∅
    end;
    𝒫 := {P};
    repeat
        𝒯 := ∅;
        for each P ∈ 𝒫 do
            for each incoming transition t at P.source s.t. t.source ≠ s₁
            do begin
                Compute t.Sval by symbolically executing t for some
                initial symbols for the local variables and input parameters ;
```

```
Symbolically interpret P.Sval with respect to t.Sval ;
infsw := false;
if ( ∃ F ∈ 𝓕 such that F = P.trlist @ t ) then
    infsw := true;
Interpret P.pred with respect to t.Sval;
if (infsw or t.pred ∧ P.pred is satisfiable) then begin
    if (infsw ) then begin
        Delete F from 𝓕; infsw := false
    end
    { Q is the walk obtained by prefixing t to P }
    Compute Q.Sval by updating P.Sval
    with respect to t.Sval;
    with Q do begin
        trlist := t @ P.trlist; pred := t.pred @ (∧ P.pred);
        def := P.def ∪ t.def; puse := P.puse ∪ t.puse;
    end
    Add Q to 𝒯;
    for each W ∈ UFWALK do
        if (W.dest = Q.source) then begin
            Interpret Q.pred with respect to W.Sval;
            if (every variable in Q.puse is defined
                    in W or in Q prior to its use) then
                if (W.pred ∧ Q.pred is satisfiable)
                        then begin
                    for each v in W.trlist do begin
                        Let W.trlist = U@v@V;
                        if (∄F ∈ 𝓕 such that F.trlist =
                            v@V@Q.trlist) then
```

```
                        Add v@V@Q.trlist to 𝓕;
                    end { for each v }
                    if (W ∈ UFWALK) then begin
                        Let T = W.trlist @Q.trlist;
                        Postfix U(1) to T;
                        Add T to UFTOUR;
                        Delete W from UFWALK;
                    end
                end { if (W.pred ∧ Q.pred }
            end { if (W.dest = Q.source) }
        end { if (infsw or t.pred ∧ P.pred }
    end { for each incoming.. }
    𝒫 := 𝒯;
  until (UFWALK = ∅)
end; { ConstructTour }
```

In the following lemma, we establish the time and the space complexities as well as the correctness of *ComputePreamble*.

Lemma 5.2 *The time and the space complexities of ComputePreamble are* $O((d^{out}_{max})^{K_1+1})$ *steps and* $O((d^{out}_{max})^{K_1})$ *units, respectively, where* d^{out}_{max} *is the maximum number of outgoing transitions including self-loops at any state in the SPEC and* K_1 *is the maximum number of iterations of the* **repeat...until** *loop of the procedure ComputePreamble. ComputePreamble successfully computes a preamble walk for those transitions in the SPEC which have at least one preamble walk of length at most* K_1*. It also computes a preamble walk for every feasible def-use pair in* $\mathcal{D}$ *which has at least one preamble walk of length at most* K_1 *excluding their CIUS subwalk extension.*

Proof:

Note that the time complexity of the **repeat...until** iterative loop of *ComputePreamble* is proportional to $|\mathcal{P}|$ multiplied by d^{out}_{max}, where $|\mathcal{P}|$ is the cardinality of $\mathcal{P}$. At the start of the ith iteration $|\mathcal{P}|$ is at most $(d^{out}_{max})^{i-1}$, $i \geq 1$. Since the **repeat...until** structure is executed at most K_1 times, *ComputePreamble* takes at most $O((d^{out}_{max})^{K_1+1})$ steps.

Let P_{max} and C_{max} denote the maximum number of input interaction parameters in any input interaction and the maximum number of statements (both the assignment statements and the output statements) in the computation block of any transition in SPEC, respectively. The maximum number of subgraphs in a data flow graph is $P_{max} + C_{max}$. Since K_1 is the maximum length of a walk in $\mathcal{P}$ generated at any iteration in the first phase, each variable/parameter can be used in at most $O(K_1(C_{max}))$ statements/predicates along any walk. In other words, the number of nodes at either the second level or the third level of any subgraph in a data flow graph is $O(K_1(C_{max}))$. As the edges in the data flow graphs are only from a node of given level to the nodes in the next-higher level, the total number of edges in a subgraph is at most $O((K_1(C_{max}))^2)$. Thus, the memory requirement of a data flow graph is $O((P_{max}+C_{max})(K_1)^2(C_{max})^2)$. We know that there are at most K_1 data flow graphs for any walk, and $(d^{out}_{max})^{K_1}$ walks in any given iteration of the **repeat...until** loop of *ComputePreamble*. Hence the space requirement for storing the data flow graphs of a given iteration is at most $(d^{out}_{max})^{K_1} K_1 O((P_{max} + C_{max})(K_1)^2(C_{max})^2)$. Also at any given iteration, $|\mathcal{P}|$ is at most $(d^{out}_{max})^{K_1}$. Hence, it follows that the space complexity of *ComputePreamble* is $O((d^{out}_{max})^{K_1})$.

The remaining part of the lemma directly follows from the fact that *ComputePreamble* searches the set of all executable walks of length at most K_1.

□

Lemma 5.3 *Let d^{in}_{max} denote the maximum number of incoming transitions including the self-loops at any state in the SPEC. Suppose that the* **repeat...until** *loop of*

ComputeTour is executed K_2 times. Then the second phase of the test case generation algorithm takes $O((d^{in}_{max})^{K_2+1})$ steps and it requires $O((d^{in}_{max})^{K_2})$ units of memory. It completes all the preambles computed in ComputePreamble into feasible tours.

Proof:

The time and space complexities proof of this lemma is similar to that of Lemma 5.2 and is therefore omitted. Note that K_2 is finite as the EFSM has an executable walk from any state with a given valid context to the initial state. From the termination condition of the **repeat** loop in *ComputeTour*, it is clear that it completes all the preambles computed in ComputePreamble into feasible tours.

□

We summarize the time and space complexities and correctness of the *SelectTest-Tour* algorithm in the following theorem.

Theorem 5.4 *Let K_2 (K_1) be the number of times (maximum number of times) the* **repeat...until** *loop of ComputeTour (ComputePreamble) is executed. The time complexity of the algorithm SelectTestTour is $O((d^{out}_{max})^{K_1+1} + (d^{in}_{max})^{K_2+1})$ steps, where d^{in}_{max} (d^{out}_{max}) denotes the maximum number of incoming (outgoing) transitions including the self-loops at any state in the SPEC. The algorithm also requires $O((d^{out}_{max})^{K_1} + (d^{in}_{max})^{K_2})$ units of memory. It successfully computes an executable tour for those transitions in the SPEC which have at least one preamble walk of length at most K_1. The algorithm computes an executable tour for every feasible def-use pair in $\mathcal{D}$ which have at least one preamble walk of length at most K_1 excluding their CIUS subwalk extension.*

□

Corollary 5.4.1 *For a suitable value of $K_1, 1 \leq K_1 < \infty$, SelectTestTour successfully computes a set of tours such that (i) the set satisfies the trans-CIUS-set criterion, and (ii) the set satisfies the def-use-ob criterion if $\mathcal{D}$ has only feasible def-use pairs.*

Proof:

We know that every transition in the SPEC is reachable from the initial state. Therefore, in a finite number of iterations of the **repeat..until** loop of *ComputePreamble*, a preamble walk can be computed for every transition in the SPEC. Similarly, if $\mathcal{D}$ has only feasible def-use pairs, then a preamble walk for every def-use pair in $\mathcal{D}$ can be computed in a finite number of iterations of the **repeat..until** loop of *ComputePreamble*. The rest of the proof follows from the above theorem.

□

Note that only a higher level complexity of the above algorithm is given in terms of the number of times variaous basic steps are executed. The executions of some of these steps may themselves be complex. For instance, finding whether a given walk is satisfiable is considered to be a part of a single step in our analysis.

5.5.2 Fault Coverage

It is known that some of the FSM-based test sequence generation methods achieve complete fault coverage capability by including the verification of the state identification sequences in the IUT. In the EFSM model, in order to establish that an input sequence is an UIS of a state in the IUT, one has to show that for any valid context of the IUT at that state, the output sequence produced by the IUT while applying the input sequence is different from the output sequence obtained by applying the input sequence at any other state with every valid context. Due to the black-box approach of testing, it is, in general, difficult to achieve this UIS verification requirement. For each incoming transition at a state s_i, our test case generation scheme in *SelectTestTour* generates one feasible tour for applying the CIUS U_i at s_i to see if it provides the expected output, and a tour for applying the CIUS U_j of the state s_j, $j = 1, 2, \ldots, n, j \neq i$ at s_i to check if it produces the output different from the

one obtained when U_j is applied at s_i. Further, these tours can be exercised for different data in their feasible domain. Thus the *SelectTestTour* algorithm establishes the CIUS verification requirement partially, while the existing EFSM based test generation methods do not consider this issue. In addition, the test tours selected are all feasible and for a suitable value for K_1, they satisfy the control flow criterion. Therefore, the control flow fault coverage of this method is the same or better than those guaranteed by the existing EFSM based methods.

We shall now discuss the data-oriented fault model for the EFSM and point out how the data-oriented faults can be detected using the test tours generated with our algorithm.

Let $ip?i(parlist)$ be the input interaction at the transition t. Let v be a parameter at the position p in *parlist*. Transition t is said to have **input interaction parameter fault** with respect to the parameter v at the position p in *parlist* if the corresponding transition in the IUT is identical to t but for the parameter in the pth position in the *parlist*. The corresponding parameter in the IUT is actually different from v.

Let $c : v := exp_1\ op\ exp_2$ be a computation statement in the transition t. Here, c, v, and op are the statement reference number, variable, and an arithmetic operator, respectively. exp_1 and exp_2 are arithmetic expressions. We say that transition t has an **arithmetic operator fault** for the operator op (in its position) at statement c, if $c' : v := exp_1\ op'\ exp_2$ is the corresponding computation statement of the transition corresponding to t in the IUT, where $op \neq op'$.

Suppose that $c : v := exp$ is a computation statement in the transition t, where c, v, and exp are the statement reference number, a variable, and an arithmetic expression, respectively. Transition t is said to have a **variable definition fault** with respect to the statement c if $c' : w := exp$ is the corresponding statement in the transition in the IUT corresponding to t, and w is different from v.

Let $c : u := exp(...v...)$ be a computation statement, where c is a statement

reference number, u and v are variables, and v is referred in the arithmetic expression exp. Transition t is said to have a **variable reference fault** with respect to v (in its position) at the statement c, if $c' : u := exp(...w...)$ is the corresponding statement in the transition in the IUT corresponding to t and w is different from v.

If the statement where the variable reference fault occurs is an output statement, then we refer to this fault as **output parameter fault.** A **constant reference fault** can be defined similar to the variable reference fault.

Similarly, we can define predicate operator fault and predicate variable reference fault.

We believe that the data-oriented faults as defined above can be detected by thoroughly executing each tour generated by the algorithm with a number of test data from the feasible domain of the tour.

For instance, let us suppose that the computation statement $c : v := exp$ in the transition t has a variable definition fault for the variable v. Here, c is the statement reference number in the computation block of t and exp is an arithmetic expression. Let $\mathcal{T}$ be the set of all tours generated by our algorithm. Let $\mathcal{T}_c \subseteq \mathcal{T}$ be the set of tours for covering all the feasible def-use pairs for the definition of v (at c in t). Assume that $\mathcal{T}_c \neq \emptyset$. Let $D = (t.c, t'.c')v$ be a feasible def-use pair with respect to v such that $\mathcal{T}_c$ has a tour, say T_D for covering D. Here, c' is an assignment statement in the computation block of t'. We know that T_D contains a def-clear walk $W1$ for D followed by another walk $W2$ through which the effect of the "use" of v at $t'.c'$ flows until it is assigned to an output interaction parameter or to a variable in the predicate of the last transition in $W2$. In the latter case, T_D also has the walk $W3$ which corresponds to the CIUS of the tail state of $W2$. Since v did not get the intended value at $t.c$, for a suitable value for the input interaction parameters along T_D from the feasible domain for T_D, either the variable defined at c' will get a value different from the expected one or $W1$ itself will not be executed. If the latter is the case, then it is quite likely that the observed sequence of output interactions

from the IUT for the sequence of input interactions along T_D will be different from the expected one; so the presence of a fault is detected. In the former case, if the terminal "use" in $W2$ corresponds to an output interaction parameter, then for certain data values for the input interaction parameters in T_D the output obtained through the output parameter in question may be different from the expected one if the walk $W2$ is executed by the IUT; otherwise the erroneous control flow would produce the unexpected output. Similarly, the terminal "use" in $W2$ is on the predicate of the last transition in $W2$, then, unexpected transitions may be chosen while the IUT attempts to execute $W2$ and unexpected output would be observed as a result.

Though it is not addressed in this thesis, the problem of finding a set of test data for executing each tour generated by a given test case generation algorithm such that the data-oriented faults are detected is certainly an interesting research problem. We believe that the set of tours generated by our approach is a good candidate for the test data selection problem, since (i) all the tours generated are executable and (ii) it provides observability of the data flow. The fault based techniques as described in [FW93, RT93, TRC93, Mor90] would be helpful to gain more insight on the test data selection problem.

5.6 Transport Protocol Test Case Generation

We shall illustrate our test case generation algorithm on the transport protocol given in Figure 5.1. In this example, we consider only the core-transitions for the coverage criteria. The set $\mathcal{D}$ of all def-use pairs of types 3, 4 and 5 (refer Section 5.2.2) is given in the first column of Table 5.5 through Table 5.7. Let us take $K_1 = 5$. Note that the def-use pairs which correspond to input parameters in a transition being used in the predicate of the same transition as well as those pairs which correspond to input parameters of a transition being used in the output interaction of the same transition are not considered. Such pairs are automatically covered in the control flow coverage

Def-Use Pair	DFWALK	UFTOUR
(t1.I, t1.c1)prop_opt	t1	
(t1.c1, t3.P)opt	t1 t3 **t17**	
(t1.c1, t4.P)opt	t1 t4 **t21**	
(t1.c2, t9.P)R_credit	infeasible	
(t1.c2, t9.c2)R_credit	infeasible	
(t1.c2, t10.P)R_credit	t1 t3 t10 **t17**	
(t1.c2, t11.c1)R_credit	t1 t3 t11	
(t2.I, t2.c1)opt_ind	t2	
(t2.I, t2.c2)cr	t2 t6 t8 **t17**	t2 t6 t8t17t20
(t2.c1, t6.P)opt	t2 t6 **t17**	
(t2.c2, t8.P)S_credit	t2 t6 t8 **t17**	
(t2.c2, t8.c1)S_credit	t2 t6 t8 t8 **t17**	t2 t6 t8 t8t17t20
(t2.c2, t16.P)S_credit	infeasible	
(t2.c3, t6.c4)R_credit	t2 t6	
(t2.c3, t9.P)R_credit	infeasible	
(t2.c3, t9.c2)R_credit	infeasible	
(t2.c3, t10.P)R_credit	t2 t6 t10 **t17**	t2 t6 t10 t17t20
(t2.c3, t11.c1)R_credit	t2 t6 t11	t2 t6 t11t17t20
(t3.I, t3.c3)opt_ind	t1 t3	
(t3.I, t3.c4)cr	t1 t3 t8 **t17**	
(t3.c1, t9.P)TRsq	t1 t3 t11 t9 **t17**	
(t3.c1, t9.c1)TRsq	t1 t3 t11 t9 t9 **t17**	t1 t3 t11 t9 t9 t17t20
(t3.c1, t10.P)TRsq	t1 t3 t10 **t17**	
(t3.c1, t11.c2)TRsq	t1 t3 t11	
(t3.c2, t8.c2)TSsq	t1 t3 t8	
(t3.c2, t8.c3)TSsq	t1 t3 t8 t8	
(t3.c2, t12.P)TSsq	t1 t3 t12 **t17**	
(t3.c2, t12.c1)TSsq	t1 t3 t12 t8 **t17**	t1 t3 t12 t8t17t20

Note: Postfix $t1t5$ to all the tours in UFTOUR

Table 5.5: Data flow coverage in the AP-module

Def-Use Pair	DFWALK	UFTOUR
(t3.c2, t13.P)TSsq	t1 t3 t13 **t17**	
(t3.c2, t14.P)TSsq	t1 t3 t14 **t17**	
(t3.c2, t14.c1)TSsq	t1 t3 t14 t8	
(t3.c2, t15.P)TSsq	t1 t3 t15 **t17**	
(t3.c4, t8.P)S_credit	t1 t3 t8 **t17**	
(t3.c4, t8.c1)S_credit	t1 t3 t8 t8 **t17**	
(t3.c4, t16.P)S_credit	t1 t3 t16 **t17**	
(t6.I, t6.c1)accp_opt	t2 t6	
(t6.c2, t9.P)TRsq	infeasible	
(t6.c2, t9.c1)TRsq	infeasible	
(t6.c3, t8.c2)TSsq	t2 t6 t8	
(t6.c3, t8.c3)TSsq	t2 t6 t8 t8	
(t6.c3, t12.P)TSsq	t2 t6 t12 **t17**	t2 t6 t12t17t20
(t6.c3, t12.c1)TSsq	t2 t6 t12 t8 **t17**	t2 t6 t12 t8t17t20
(t6.c3, t13.P)TSsq	t2 t6 t13 **t17**	t2 t6 t13t17t20
(t6.c3, t14.P)TSsq	t2 t6 t14 **t17**	t2 t6 t14t17t20
(t6.c3, t14.c1)TSsq	t2 t6 t14 t8 **t17**	t2 t6 t14 t8t17t20
(t6.c3, t15.P)TSsq	t2 t6 t15 **t17**	t2 t6 t15t17t20
(t8.c1, t8.P)S_credit	t1 t3 t8 t8 **t17**	t1 t3 t8 t8t17t20
(t8.c1, t8.c1)S_credit	t1 t3 t8 t8 t8 **t17**	t1 t3 t8 t8 t8t17t20
(t8.c1, t16.P)S_credit	t1 t3 t8 t16 **t17**	t1 t3 t8 t16t17t20
(t8.c3, t8.c2)TSsq	t1 t3 t8 t8	
(t8.c3, t8.c3)TSsq	t1 t3 t8 t8 t8 **t17**	
(t8.c3, t12.P)TSsq	t1 t3 t8 t12 **t17**	t1 t3 t8 t12t17t20
(t8.c3, t12.c1)TSsq	t1 t3 t8 t12 t8 **t17**	t1 t3 t8 t12 t8t17t20
(t8.c3, t13.P)TSsq	t1 t3 t8 t13 **t17**	t1 t3 t8 t13t17t20
(t8.c3, t14.P)TSsq	t1 t3 t8 t14 **t17**	t1 t3 t8 t14t17t20
(t8.c3, t14.c1)TSsq	t1 t3 t8 t14 t8 **t17**	t1 t3 t8 t14 t8t17t20
(t8.c3, t15.P)TSsq	t1 t3 t8 t15 **t17**	t1 t3 t8 t15t17t20

Note: Postfix $t1t5$ to all the tours in UFTOUR

Table 5.6: Data flow coverage in the AP-module (contd.)

Def-Use Pair	DFWALK	UFTOUR
(t9.c1, t9.P)TRsq	t1 t3 t11 t9 t9 **t17**	
(t9.c1, t9.c1)TRsq	t1 t3 t11 t9 t9	
(t9.c1, t10.P)TRsq	t1 t3 t11 t9 t10 **t17**	
(t9.c2, t9.P)R_credit	t1 t3 t11 t9 t9 **t17**	
(t9.c2, t9.c2)R_credit	t1 t3 t11 t9 t9	
(t9.c2, t10.P)R_credit	t1 t3 t11 t9 t10 **t17**	t1 t3 t11 t9 t10t17t20
(t9.c2, t11.c1)R_credit	t1 t3 t11 t9 t11	t1 t3 t11 t9 t11t17t20
(t11.I, t11.c1)cr	t1 t3 t11	
(t11.c1, t11.c1)R_credit	t1 t3 t11 t11	t1 t3 t11 t11t17t20
(t11.c1, t9.P)R_credit	t1 t3 t11 t9 **t17**	t1 t3 t11 t9t17t20
(t11.c1, t9.c2)R_credit	t1 t3 t11 t9	
(t11.c1, t10.P)R_credit	t1 t3 t11 t10 **t17**	t1 t3 t11 t10t17t20
(t12.I, t12.c1)XpSsq	t1 t3 t12 t8 **t17**	
(t12.I, t12.c1)cr	t1 t3 t12 t8 **t17**	
(t12.c1, t8.P)S_credit	t1 t3 t12 t8 **t17**	
(t12.c1, t8.c1)S_credit	t1 t3 t12 t8 t8 **t17**	t1 t3 t12 t8 t8t17t20
(t12.c1, t16.P)S_credit	t1 t3 t12 t16 **t17**	t1 t3 t12 t16t17t20
(t14.I, t14.c1)XpSsq	t1 t3 t14 t8 **t17**	
(t14.I, t14.c1)cr	t1 t3 t14 t8 **t17**	
(t14.c1, t8.P)S_credit	t1 t3 t14 t8 **t17**	t1 t3 t14 t8t17t20
(t14.c1, t8.c1)S_credit	t1 t3 t14 t8 t8 **t17**	t1 t3 t14 t8 t8t17t20
(t14.c1, t16.P)S_credit	t1 t3 t14 t16 **t17**	t1 t3 t14 t16t17t20

Note: Postfix $t1t5$ to all the tours in UFTOUR

Table 5.7: Data flow coverage in the AP-module (Contd.)

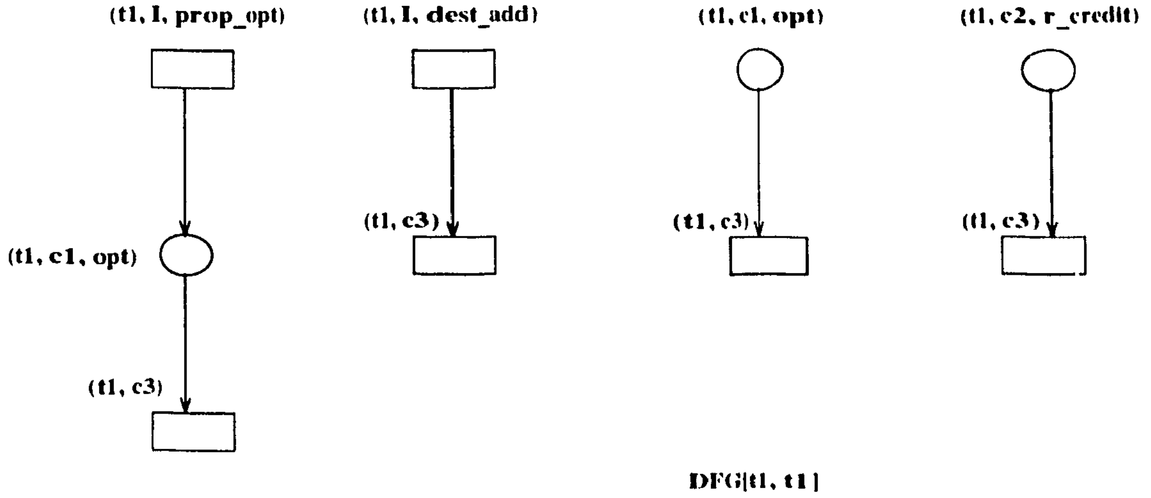


Figure 5.4: Data flow graph for the transition $t1$ in the AP-module

of the transition where the def-use association occurs. Data flow graphs for all the transitions in the AP-module are obtained using the procedure *ConstructDFG*. The data flow graph for the transition $t1$ with respect to the same transition is shown in Figure 5.4. Recall that, in the data flow graphs, rectangles represent the i-nodes as well as the o-nodes, whereas the circles and diamonds represent the c-nodes and the p-nodes, respectively. The procedure *ConstructPreamble* is illustrated by considering its fourth iteration of the **repeat..until** loop on the AP-module. At the end of the third iteration, $\mathcal{P}$ has the following walks.

$$t1t3t8,\ t1t3t11,\ t1t3t12,\ t1t3t13,\ t1t3t14,\ t1t3t15,$$
$$t1t3t17,\ t1t3t18,\ t1t4t21,\ t1t4t20,\ t2t7t19,\ t2t6t8,\ t2t6t11,$$
$$t2t6t12,\ t2t6t13,\ t2t6t14,\ t2t6t15,\ t2t6t17,\ t2t6t18$$

By the end of the third iteration, the procedure has successfully found preamble walks for the control flow coverage of all the transitions except $t9$. The procedure also maintains the data flow graph for each transition in every walk in $\mathcal{P}$. Let $W = t1t3t8$. Figure 5.5 shows the data flow graphs for the transitions in W with respect to W. The system of interpreted constraints associated with W is given below.

$$N?TrCC_3.opt_ind - U?TCONreq_1.prop_opt \leq 0 \qquad (5.1)$$

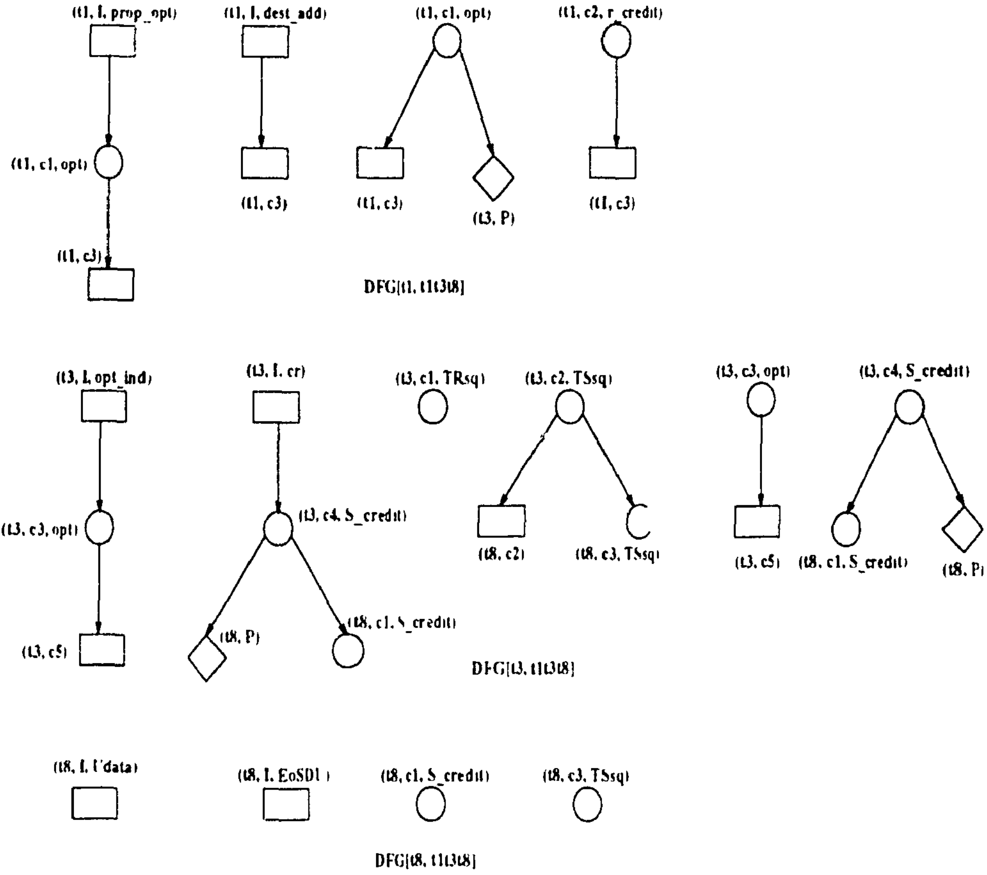


Figure 5.5: Data flow graphs for $t1, t3, t8$ with respect to the walk $t1t3t8$ of the AP-module

$$N?TrCC_3.cr > 0 \tag{5.2}$$

The subscripts of the input interactions represent the transitions in which the interactions occur. From the previous iteration it is known that the above system is feasible. In order to extend W, *ComputePreamble* considers all the transitions from s_4 (the tail state of W) one by one. Following is the interpreted predicate of $t8$, an outgoing transition at s_4, subject to the context set by W.

$$N?TrCC_3.cr - 1 > 0 \tag{5.3}$$

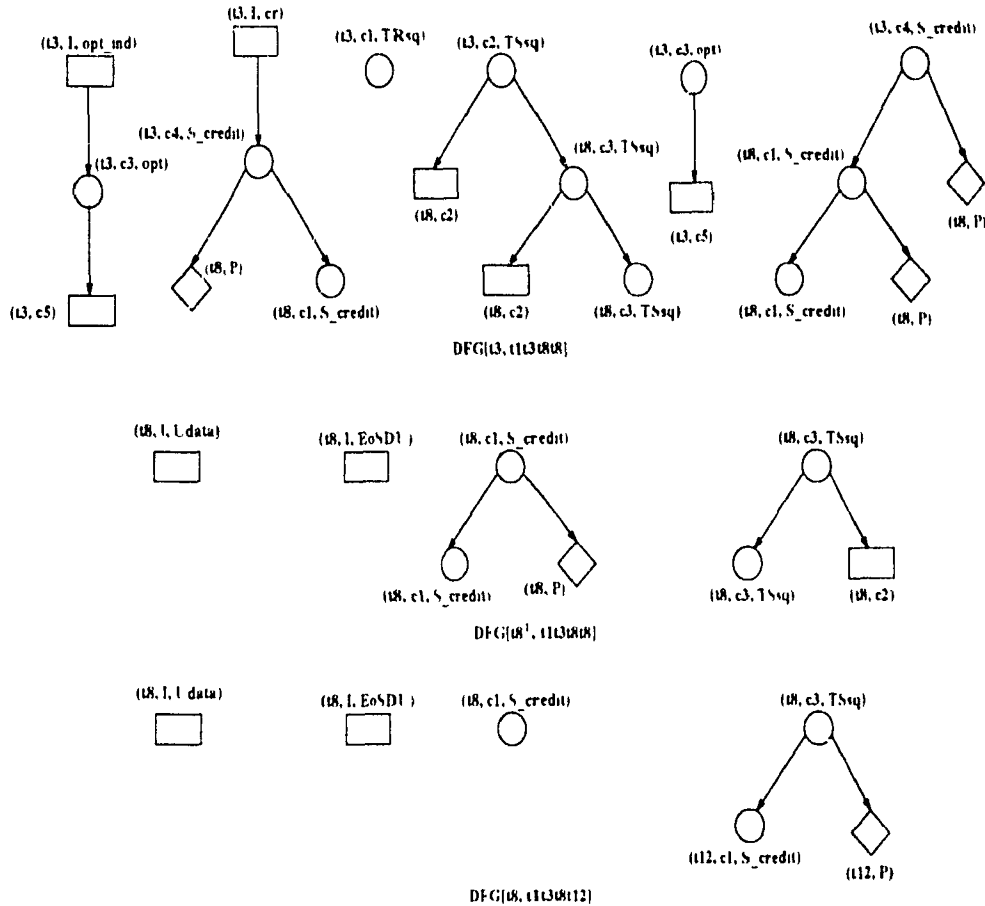


Figure 5.6: Data flow graphs for $t3$ and $t8^1$ with respect to $t1t3t8^1t8^2$ and $t8$ with respect to $t1t3t8t12$

The system of inequalities (5.1), (5.2), and (5.3) is feasible since W' is feasible, (5.1) and (5.3) are independent, (5.3) is feasible and (5.3) implies (5.2). Let $W2$ denote the extended walk $t1t3t8^1t8^2$. The superscript of a transition in a walk denotes the order of occurrence of the transition in the walk. The data flow graphs DFG[t3, W2] and DFG[t8^1, W2] obtained by extending the graphs in Figure 5.5 over the transition $t8$ are shown in Figure 5.6. Note that DFG[t1, W2] is the same as DFG[t1, W]. Observe from the figure that the walk $W2$ covers $(t3.c4, t8.c1)S_credit$, $(t3.c2, t8.c3)TSsq$, $(t8.c1, t8.P)S_credit$, and $(t8.c3, t8.c2)TSsq$ for our def-use-ob criterion.

Transition (t)	Preamble CFPRE[t]	Set of walks for testing t	Tour UFTOUR
t1		t1t5	t1t5
		t1t25	t1t25t5
		t1t26	t1t26t5
		t1t27	t1t27t5
t2		t2t7	t2t7
		t2t28	t2t28t7t19
		t2t29	t2t29t7t19
		t2t30	t2t30t7t19
t3	t1	t1t3t17	t1t3t17
		t1t3t31	t1t3t31t17t20
		t1t3t32	t1t3t32t17t20
		t1t3t18	t1t3t18t19
t4	t1	t1t4t21	t1t4t21
		t1t4t33	t1t4t33t20
		t1t4t34	t1t4t34t20
		t1t4t35	t1t4t35t20
t5	t1	t1t5t1	t1t5t1t5
		t1t5t22	t1t5t22
		t1t5t23	t1t5t23
		t1t5t24	t1t5t24
t6	t2	t2t6t17	t2t6t17t20
		t2t6t31	t2t6t31t17t20
		t2t6t32	t2t6t32t17t20
		t2t6t18	t2t6t18t19
t7	t2	t2t7t19	t2t7t19
		t2t7t36	t2t7t36t19
		t2t7t37	t2t7t37t19
		t2t7t38	t2t7t38t19
t8	t1t3	t1t3t8t17	t1t3t8t17t20
		t1t3t8t31	t1t3t8t31t17t20
		t1t3t8t32	t1t3t8t32t17t20
		t1t3t8t18	t1t3t8t18t19

Note 1: Obtain the entry for each of $t10 \ldots t15$ by replacing $t8$ in its entry
Note 2: Postfix $t1t5$ to all the tours in UFTOUR

Table 5.8: Preambles and control flow test tours for the transport protocol

Transition (t)	Preamble CFPRE[t]	Set of walks for testing t	Tour UFTOUR
t9	t1t3t11	t1t3t11t9t17	t1t3t11t9t17t20
		t1t3t11t9t31	t1t3t11t9t31t17t20
		t1t3t11t9t32	t1t3t11t9t32t17t20
		t1t3t11t9t18	t1t3t11t9t18t19
t16	t1t3	t1t3t16t17	t1t3t16t17t20
		t1t3t16t31	t1t3t16t31t17t20
		t1t3t16t32	t1t3t16t32t17t20
		t1t3t16t18	t1t3t16t18t19
t17	t1t3	t1t3t17t21	t1t3t17t21
		t1t3t17t33	t1t3t17t33t20
		t1t3t17t34	t1t3t17t34t20
		t1t3t17t35	t1t3t17t35t20
t18	t1t3	t1t3t18t19	t1t3t18t19
		t1t3t18t36	t1t3t18t36t19
		t1t3t18t37	t1t3t18t37t19
		t1t3t18t38	t1t3t18t38t19
t19	t2t7	t2t7t19t1	t2t7t19t1t5
		t2t7t19t22	t2t7t19t22
		t2t7t19t23	t2t7t19t23
		t2t7t19t24	t2t7t19t24
t20	t1t4	t1t4t20t1	t1t4t20t1t5
		t1t4t20t22	t1t4t20t22
		t1t4t20t23	t1t4t20t23
		t1t4t20t24	t1t4t20t24

Note 1: Obtain the entry for $t21$ by replacing $t20$ in its entry

Note 2: Postfix $t1t5$ to all the tours in UFTOUR

Table 5.9: Preambles and control flow test tours for the transport protocol (Contd.)

Let us consider another transition, $t9$, from s_4. The predicate for $t9$ is $R_credit \neq 0 \wedge send_sq = TRsq$. As per $W = t1t3t8$, the current value of R_credit is zero. Therefore the walk W followed by $t9$ is unexecutable. However, the walk W followed by $t10$ is executable for the current value of R_credit. W followed by $t11$ is also executable since $t11$ has no predicate. These extended walks neither cover any new def-use pair for data flow nor any new transition for control flow (the preamble ($t1t3$) for $t10$ and $t11$ is already obtained in the previous iteration). The system of constraints (5.4),(5.5) and(5.6) (given below) correspond to the interpretation of the predicate in $t12$ with respect to the walk W.

$$N?TrAK_{12}.XpSsq \leq 1 \tag{5.4}$$

$$N?TrAK_{12}.cr + N?TrAK_{12}.XpSsq \geq 1 \tag{5.5}$$

$$N?TrAK_{12}.cr + N?TrAK_{12}.XpSsq \leq 16 \tag{5.6}$$

It is easy to see that the above system is feasible. Since this system is independent from the constraints (5.1) and (5.2), the walk $t1t3t8t12$ is also executable. The data flow graph DFG[t8, t1t3t8t12] as given in Figure 5.6 shows that $t1t3t812$ is a preamble for the def-use pair $(t8.c3, t12.P)TS.sq$. Similarly, the walks $t1t3t8t13$, $t1t3t8t14$ and $t1t3t8t15$ are found to be the preamble walks for the def-use pairs $(t8.c3, t13.P)TSsq$, $(t8.c3, t14.P)TSsq$, and $(t8.c3, t15.P)TSsq$, respectively Though W followed by $t17$ is executable it does not have any additional coverage for the control or data flow criterion. The above step is repeated for all the walks in $\mathcal{P}$, before moving on to the fifth iteration. A preamble for transition $t9$ is also obtained in the fourth iteration. The preamble walks obtained for the control flow coverage of all the transitions are shown in the second column of Table 5.8 and Table 5.9. Note that only U?TCONreq(dest_add, prop_opt) - the first part (CIUS part) of the cyclic CIUS of s_1 - is applied at the tail state of the transitions.

By the fifth iteration, the procedure successfully finds the preamble walks for all the feasible def-use pairs for our def-use-ob criterion. The preamble for each feasible

def-use pair as obtained in this procedure is shown in the second column of Table 5.5 through Table 5.7. Observe that the bold faced transitions appended to a walk in the table is for confirming the tail state of the previous transition whose predicate uses the value of the variable in the def-use pair. Note that this is a requirement of the def-use-ob criterion. Also, observe from Table 5.5 through Table 5.7 that some of the def-use pairs are infeasible. The preamble walks computed in *ComputePreamble* are collected into the set UFWALK. It consists of the walks in column 2 of Table 5.5 through Table 5.7 as well as column 3 of Table 5.8 and Table 5.9 of those entries which have valid tours in their last column. It can be seen that all other walks in the tables are either duplicates or subwalks of those in UFWALK. Tours covering the walks in UFWALK are computed by the procedure *ComputeTour*. The procedure is fairly straight forward when it is applied to the AP-module. Since all the incoming transitions ($t5, t19, t20$ and $t21$) at state s_1 do not have predicates, in the first iteration, all the walks in UFWALK which terminate at the starting states (s_2, s_5 and s_6) of these transitions are completed into executable tours by concatenating the appropriate transitions from $\{t5, t19, t20, t21\}$. Among other walks, $\mathcal{P}$ has the single transition walks $t19$ and $t20$ at the end of the first iteration of *ComputeTour*. Note that the incoming transitions $t7$ and $t17$ to the starting state of $t19$ and $t20$, respectively, are predicate-free. This implies that the walks $t7t19$ and $t17t20$ are always satisfiable independent of the context by which their starting states are entered. Therefore, all the walks in UFWALK which terminate at s_3 and s_4 are augmented into executable tours by concatenating them with the postamble walks $t7t19$ and $t17t20$, respectively. In this manner, tours are successfully found for all the walks in UFWALK. Thus, the procedure *ComputeTour* terminates in the second iteration. The resulting tours are shown in the last columns of Table 5.5 through Table 5.7, Table 5.8 and Table 5.9. The tour $t1t5$, the CIUS of s_1 is finally postfixed to all the selected tours. This set of tours satisfies both the trans-UIO-set criterion for control flow testing and the def-use-ob criterion for the data flow testing.

Let us examine the fault detection capability of the generated test tours through examples. Suppose that an IUT has a simple control flow fault at the transition $t6$, which originally ends at s_4. Let the tail state of this transition in the IUT be s_2. While applying a test data along the tour $t2t6t17t20$ which is one of the tours for covering the trans-CIUS-criterion for $t6$ (refer to Table 5.8), it shows an output mismatch. Following is the the expected sequence of outputs.

$$U!TCONind(peer_add, opt)\ N!TrCC(opt, R_credit)$$
$$N!TrDR(\text{'User initiated'}, \text{false})\ N!terminated\ U!TDISconf$$

However, the IUT produces the output sequence given below.

$$U!TCONind(peer_add, opt)\ N!TrCC(opt, R_credit)\ N!terminated\ U!TDISconf$$

Therefore the fault is detected.

Suppose that the IUT has a variable definition fault at $t3.c4$ where the variable S_credit is defined. Let us assume that the default value for all the integer variables is zero. Take the def-use pair $D = (t3.c4, t8.c1)S_credit$. From Table 5.6, we see that $t1t3t8t8t17$ covers D and $T = t1t3t8t8t17t20t1t5$ is the required tour for covering D with respect to the def-use-ob criterion. Observe that for any feasible test data for T, the expected sequence along the tour is

$$U?TCONreq(dest_add, prop_opt)\ N!TrCR(dest_add, opt, R_credit)$$
$$N?TrCC(opt_ind, cr)\ U!TCONconf(opt)\ U?TDATreq(Udata, EoSDU)$$
$$N!TrDT(TSsq, Udata, EoSDU)\ U?TDATreq(Udata, EoSDU)$$
$$N!TrDT(TSsq, Udata, EoSDU)\ U?TDISreq$$
$$N!TrDR(\text{'Userinitiated'}, false)\ N?TrDC\ \ N!terminated\ U!TDISconf$$
$$U?TCONreq(dest_add, prop_opt)\ N!TrCR(dest_add, opt, R_credit)$$
$$N?TrDR(disc_reason, switch) U!TDISind(disc_reason) N!terminated$$

However, the sequence of inputs and outputs as per the IUT is

$$U?TCONreq(dest_add, prop_opt)\ N!TrCR(dest_add, opt, R_credit)$$
$$N?TrCC(opt_ind, cr)\ U!TCONconf(opt)\ U?TDATreq(Udata, EoSDU)$$
$$U?TDATreq(Udata, EoSDU)\ U?TDISreq$$
$$N!TrDR(\text{'Userinitiated'}, false)\ N?TrDC\ N!terminated\ U!TDISconf$$
$$U?TCONreq(dest_add, prop_opt)\ N!TrCR(dest_add, opt, R_credit)$$
$$N?TrDR(disc_reason, switch) U!TDISind(disc_reason) N!terminated$$

Thus, the presence of the fault in the IUT is detected.

5.7 The Feasibility Problem

In this section we propose an approach for solving a special type of the feasibility problem encountered in the algorithms for CIUS computation and test case generation. Let the predicates in the transitions along a walk starting from the initial state be expressed in terms of some constants and the input interaction parameters along the walk. The input interaction parameters in the predicates of the transitions are referred to as **decision variables**. We consider a restricted class of the feasibility problem where the predicates are linear (in terms of the decision variables). We also assume that all the decision variables are of type integer, real, or boolean. For the sake of test case generation, a boolean decision variable can be considered as an integer variable over the domain $\{0, 1\}$. Thus, every decision variable in a protocol considered in this section is either of integer or of real type only. Let W be a walk starting from the initial state (arbitrary walks in the SPEC can also be handled using the proposed approach). Let t be a transition starting from the tail state of W.

> **The feasibility problem is to find if the walk W' obtained by postfixing t with W is feasible given that W is feasible**

Assume that the predicates in all the transitions in W as well as the predicate in t are already interpreted as per W'. Consider the **Disjunctive Normal Form (DNF)**

of the conjunction of the predicates along W. It is easy to see that each product term in this form can be expressed as a system of linear equations and inequalities (henceforth referred to as a *system*). Let $C_1, C_2, \ldots, C_w$ be the family of systems for W. In addition to the equations and the inequalities of the ith product term of the DNF for W, C_i also contains constraints on the upper and the lower bounds on the decision variables. Similarly, let $D_1, D_2, \ldots, D_z$ be the family of systems for t. Clearly, each C_i, $1 \leq i \leq w$ and each D_j, $1 \leq j \leq z$ is a **mixed integer programming feasibility problem** [PS82, Chv83]. Here, we simply refer to the above problem as the **Integer Programming Problem (IPP)**. The IPP in **standard form** is to find the column vector x of size n satisfying the following constraints, where A is an $m \times n$ matrix over $\Re$, the set of all real numbers, and b, l, and u are column vectors of size m over $\Re$.

$$\begin{aligned} Ax &= b , \\ l \leq x &\leq u , \\ x_i &\in \mathcal{Z},\ 1 \leq i \leq k , \\ x_i &\in \Re,\ k+1 \leq i \leq n . \end{aligned}$$

Here, k is an integer between 1 and n and $\mathcal{Z}$ is the set of all integers. In general, if y is a matrix, then y_{ij} denotes the (i,j)th element of y. Before presenting the method for solving the feasibility problem on hand, we need to provide a method for solving the IPP. We propose an approach, henceforth referred to as GC-PET, for solving the IPP by combining the **Gomory Cut** (GC) [PS82] and the **Partial Enumeration Technique** (PET) [PR88]. Consider the following **Linear Programming (LP)** relaxation of the IPP.

$$\begin{aligned} Problem(0) \qquad Ax &= b , \\ l \leq x &\leq u , \\ x &\in \Re^n . \end{aligned}$$

We shall solve Problem(0) using the first phase of the two-phase simplex method [Chv83]. If Problem(0) is infeasible, so is the original IPP. Any integral solution of

Problem(0) is obviously a solution for the IPP. Suppose that the solution x^0 obtained for Problem(0) is not integral for some integer variables. Then, let x_p be one such variable. Let $A_I x = b_I$ be a system of Gomory cuts [PS82] for pruning some of the feasible regions of Problem(0) which do not contain any point with integer components for all the variables x_i, $1 \leq i \leq k$. Now, consider the following subproblems of Problem(0).

$$\begin{array}{llcl}
Problem(1) & Ax & = & b\,, \\
 & A_I x & = & b_I\,, \\
 & x_p & \leq & \lfloor x_p^0 \rfloor\,, \\
 & l \leq x & \leq & u\,, \\
 & x_i & \in & \mathcal{Z},\ 1 \leq i \leq k\,, \\
 & x_i & \in & \Re,\ k+1 \leq i \leq n\,.
\end{array}$$

$$\begin{array}{llcl}
Problem(2) & Ax & = & b\,, \\
 & A_I x & = & b_I\,, \\
 & x_p & \geq & \lfloor x_p^0 \rfloor + 1\,, \\
 & l \leq x & \leq & u\,, \\
 & x_i & \in & \mathcal{Z},\ 1 \leq i \leq k\,, \\
 & x_i & \in & \Re,\ k+1 \leq i \leq n\,.
\end{array}$$

Find a solution to the LP-relaxation of one of the above two problems, say Problem(1). If the resulting solution is integral, then we are done. If the LP-relaxation is infeasible, then choose the other IPP, Problem(2) and, as before, proceed solving its LP-relaxation and so on. However, if the solution to the LP-relaxation of Problem(1) has a non-integer solution to some integer variables, then split Problem(1) into Problem(3) and Problem(4), just as we split Problem(0) and start solving the LP-relaxation of one of the subproblems. This process continues recursively, as the feasible regions for the subproblems get finer and finer. This process will stop in a finite number of steps as there are only a finite number of variables and each of the

integer decision variables has explicit lower and upper bounds. This approach of solving an IPP can be used to solve any of the systems $C_i, 1 \leq i \leq w$ and $D_j, 1 \leq j \leq z$, while solving our feasibility problem at hand.

At the starting of the given feasibility problem, it is known that C_p is feasible for some $p, 1 \leq p \leq w$. Let us assume that C_p was solved using GC-PET. Let B be a feasible basis for the LP-relaxation of C_p. Since C_p was solved using GC-PET, B is also known. In order to find if W' is feasible, first check if any one of the systems $D_j, 1 \leq j \leq z$ is independent of C_p and is feasible. If there exists one such system, then let it be D_s for some $s, 1 \leq s \leq z$. In this case, W' is obviously feasible and the solutions for C_p and D_s together form a solution for executing W'. Also, the basis for the LP-relaxation of C_p and that of the LP-relaxation of D_s together form a basis for the LP-relaxation of the combined system of C_p and D_s for W'.

If none of the systems $D_j, 1 \leq j \leq z$, is independent of C_p and feasible, then pick a feasible system, say D_q, arbitrarily. Let E_{pq} be the system obtained by combining C_p and D_q. Using GC-PET, we need to check if the system E_{pq} is feasible. As explained below, the basis B could be used as part of the starting basis while solving the LP-relaxation for E_{pq}. We just have to add one artificial variable to X_B, the set of basic variables, for each of the constraints in D_q. We then invoke the revised dual simplex method if the resulting basis is dual feasible; if not we proceed with the revised simplex method until we obtain the optimum solution to the auxiliary problem of the first phase of the two-phase simplex method. If E_{pq} is found to be feasible, then we are done. Otherwise, other feasible systems for t have to be similarly checked for feasibility when they are combined with C_p. If C_p is not feasible with any of the feasible systems for t, then we need to replace C_p with another feasible system for W and repeat the above procedure. Note that the proposed approach for the feasibility problem achieves the reduction in the complexity of the feasibility problem since the solution obtained from a system for W is used in solving a system for $W\,t$.

We shall illustrate our approach of solving the feasibility problem using the transport protocol given in Figure 5.1. We would like to find whether the transition $t8$ is executable when it is reached using the walk $W = t1t3t8$. Following is the system associated with W.

$$\begin{aligned} x_1 - x_2 &\leq 0 , \\ x_3 &> 0 , \\ 1 \leq x_1, x_2 &\leq 5 , \\ 1 \leq x_3 &\leq 15 , \\ x_1, x_2, x_3 &\in \mathcal{Z} . \end{aligned}$$

Here, x_1, x_2 and x_3 correspond to the protocol's decision variables $N?TrCC_3.opt_ind$, $U?TCONreq_1.prop_opt$ and $N?TrCC_3.cr$, respectively. The bounds on these decision variables are normally set by the tester based on the specification of the protocol. By adding the slack variable x_4 and the surplus variable x_5, the above IPP can be expressed in the standard form as shown below.

$$\begin{aligned} IP1 \quad x_1 - x_2 + x_4 &= 0 , \\ x_3 - x_5 &= 0 , \\ 1 \leq x_1, x_2 &\leq 5 , \\ 1 \leq x_3 &\leq 15 , \\ 0 \leq x_4 &\leq 4 , \\ 1 \leq x_5 &\leq 14 , \\ x_i &\in \mathcal{Z},\ 1 \leq i \leq 5 . \end{aligned}$$

We will first check for the feasibility of the problem IP1 using GC-PET. Let LP1 denote the LP-relaxation of IP1; LP1 is obtained from IP1 by simply replacing $\mathcal{Z}$ by $\Re$ in the last constraint of IP1. LP1 is solved using the first phase of the two phase simplex method. In other words, LP1 can be solved by solving the following auxiliary

problem AP1.

$$
\begin{aligned}
AP1 \quad Maximize \qquad & -x_6 - x_7 \\
s.t. \quad x_1 - x_2 + x_4 + x_6 &= 0 , \\
x_3 - x_5 + x_7 &= 0 , \\
1 \le x_1, x_2 &\le 5 , \\
1 \le x_3 &\le 15 , \\
0 \le x_4 &\le 4 , \\
1 \le x_5 &\le 14 , \\
0 \le x_6, x_7 &\le \infty , \\
x_i &\in \Re, 1 \le i \le 7 .
\end{aligned}
$$

Here, both x_6 and x_7 are artificial variables. The initial solution for the problem as per the revised simplex method is given as $X_B = (x_6, x_7)^T = (4,0)^T$ and $X_N = (x_1, x_2, x_3, x_4, x_5)^T = (1,5,1,0,1)^T$. The basis B is the identity matrix. It is easy to see that x_1 enters the basis and x_6 leaves the basis in the first iteration. As a result, we have $X_B = (x_1, x_7)^T = (5,0)^T$, $X_N = (x_2, x_3, x_4, x_5, x_6)^T = (5,1,0,1,0)^T$. B still remains as the identity matrix. In the second iteration, x_3 and x_7 become the entering and the leaving variables, respectively. B continues to be the identity matrix. Also, the iteration produces the following solution: $X_B = (x_1, x_3)^T = (5,1)^T$, $X_N = (x_2, x_4, x_5, x_6, x_7)^T = (5,0,1,0,0)^T$. The third iteration, in fact, confirms that the above solution is optimal. As the optimal value of the objective function is zero, it can be concluded that the problem LP1 is feasible and $(x_1, x_2, x_3, x_4, x_5)^T = (5,5,1,0,1)^T$ is a solution. This also becomes a solution for the original problem IP1 since it is integral. Therefore, $W(= t1t3t8)$ is feasible.

Suppose that W' is the walk obtained by postfixing $t8$ to W. Then, our feasibility problem is to check if the walk W' is executable. Note that the constraint '$x_3 > 1$' of the transition $t8$ (after the execution of W) in isolation is feasible. We need to add this constraint to the problem IP1. Let us record the resulting problem

in standard form as IP2.

$$
\begin{aligned}
IP2 \quad x_1 - x_2 + x_4 &= 0\,, \\
x_3 - x_5 &= 0\,, \\
x_3 - x_8 &= 1\,, \\
1 \leq x_1, x_2 &\leq 5\,, \\
1 \leq x_3 &\leq 15\,, \\
0 \leq x_4 &\leq 1\,, \\
1 \leq x_5, x_8 &\leq 14\,, \\
x_i &\in \mathcal{Z}, 1 \leq i \leq 5\,, \\
x_8 &\in \mathcal{Z}\,.
\end{aligned}
$$

Note that x_8 is a surplus variable. Let LP2 be the LP-relaxation of IP2. In order to solve LP2, it is enough to solve the following auxiliary problem.

$$
\begin{aligned}
AP2 \quad Maximize \quad & -x_6 - x_7 - x_9 \\
s.t. \quad x_1 - x_2 + x_4 + x_6 &= 0\,, \\
x_3 - x_5 + x_7 &= 0\,, \\
x_3 - x_8 + x_9 &= 1\,, \\
1 \leq x_1, x_2 &\leq 5\,, \\
1 \leq x_3 &\leq 15\,, \\
0 \leq x_4 &\leq 4\,, \\
1 \leq x_5, x_8 &\leq 14\,, \\
0 \leq x_6, x_7, x_9 &\leq \infty\,, \\
x_i &\in \Re,\ 1 \leq i \leq 9\,.
\end{aligned}
$$

A starting basis for AP2 can be obtained from the optimum basis of AP1. The columns corresponding to the optimum basic variables of AP1 and the column corresponding to the artificial variable x_9 together form the starting basis for AP2. The

starting basis B and the basic feasible solution are given below.

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

$$X_B = (x_1, x_3, x_9)^T = (5, 1, 1)^T \quad \& \quad X_N = (x_2, x_4, x_5, x_6, x_7, x_8)^T = (5, 0, 1, 0, 0, 1)^T.$$

Since the above solution is feasible but not optimal, we proceed with the revised simplex method. In the first iteration, x_5 enters the basis whereas x_9 leaves the basis. The resulting solution is $X_B = (x_1, x_3, x_5)^T = (5, 2, 2)^T$. Also, we have, $X_N = (x_2, x_4, x_6, x_7, x_8, x_9)^T = (5, 0, 0, 0, 1, 0)^T$ and the basis matrix is

$$B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 0 \end{pmatrix}.$$

The second iteration confirms the optimality of the above solution. As the optimal value of the objective function is zero, we conclude that the above solution yields the feasible solution $(x_1, x_2, x_3, x_4, x_5, x_8)^T = (\,,5, 2, 0, 2, 1)^T$ for the problem LP2. As this solution for the LP-relaxation LP2 of IP2 is integral, it is also a solution for our problem IP2. Therefore, the walk $W' = t1t3t8t8$ is feasible for $N.TrCC_3.opt_ind = 5$, $U.TCONreq_1.prop_opt = 5$ and $N.TrCC_3.cr = 1$.

5.8 Summary

An approach for generating test cases for both the control flow and data flow aspects of an EFSM is discussed in this chapter. A new type of state identification sequence, namely, the Context Independent Unique Sequence, is defined and an algorithm for computing a CIUS of a given state in an EFSM is developed. The trans-CIUS-set criterion used in the test case selection is superior to the existing control flow coverage criteria for the EFSM. In order to provide observability, the "all-uses" data

flow coverage criterion is extended to what is called the def-use-ob criterion. A two-phase breadth-first search algorithm is designed for generating a set of executable test tours for covering the selected criteria. The approach is also illustrated on an EFSM module of a transport protocol. Finally, the feasibility problem encountered in the above algorithms is discussed for the restricted case where the constraints are linear in real and integer variables.

Chapter 6

SUMMARY AND PROBLEMS FOR FUTURE STUDY

In this concluding chapter, we summarize the work reported in this thesis and suggest a few problems for further study.

6.1 Summary

Conformance testing of communication protocols is essential for achieving interworking of heterogeneous systems in a computer network. Conformance testing is done by generating a set of test cases from the specification of the protocol standard and applying them on the implementations to verify if their behavior is as per the specification. Finite State Machine (FSM) and Extended Finite State Machine (EFSM) models are the ones widely used for automatic test case generation. As the quality of the test results and the efficiency of testing depend on the test cases selected, the test case generation problem has been an active area of research. This thesis is concerned with some of the important open issues in test case generation from the FSM and the EFSM models.

Fault coverage and the length of a test sequence are often conflicting factors to

be considered while generating test sequences. Ideally, one would like to minimize the length of the test sequence and maximize the fault coverage. The U-method [ADLU88] and the MU-method [SL92] minimize the length of the test sequences by successfully applying the techniques for the asymmetric Rural Postperson Problem (RPP) and the minimum cost maximum flow problem, respectively. The test sequences generated by these methods have reasonably high fault coverage [MCS93]. The methods require certain auxiliary graphs constructed from the specification FSM to be connected for an assignment of an Unique Input Sequence (UIS) for each transition. Unfortunately, not all the protocols satisfy this requirement. In Chapter 2, we have addressed this problem and presented an extension of the MU-method [SL92] so that our method can be applied to any protocol which is represented as a strongly connected FSM having at least one UIS for every state.

Our new method selects a minimum number of transitions and an assignment of UIS for them from a set of UISs such that the resulting auxiliary graph has a minimum number of connected components (If the auxiliary graph has exactly one component, then it is connected). This problem, defined as the Basic UIS Assignment Problem (BUAP), is formulated as a Maximum Cardinality Two Matroid Intersection Problem (MC2MIP) and an efficient algorithm based on the works of Lawler [Law75, Law76] and Edmonds [Edm79] is presented. In order to obtain an optimal test sequence, the problem of assigning the UISs for the remaining transitions is formulated as a maximum flow minimum cost problem which does not affect the solution for the BUAP. To handle the case when the auxiliary graph is not connected, three heuristic algorithms are proposed for the general RPP. For one of them an explicit bound on the cost of the approximate tour is established. The proposed method carefully combines these algorithms with the MU-method [SL92] for generating a test sequence for any strongly connected FSM-based protocol which has at least one UIS for each state. The fault coverage of our method is the same as that of the MU-method.

Chapter 3 and Chapter 4 consider the fault detection and the fault diagnosis

aspects of testing for protocols based on the FSM model. In Chapter 3, the existing FSM-based test case generation methods are analyzed for their fault detection and diagnosis capabilities when the implementations have at most a single output or transfer fault. It is observed that the C-method [KRK74, Koh78] does not have complete fault coverage. Some guidelines for developing a method with better fault diagnosis capability are also proposed in this chapter.

In Chapter 4, two new methods based on the Wp-method [FBK+91] are presented for diagnosing the fault in implementations with at most one output or transfer fault. Our first method, called the UIDD-method, uses an UIS set and a state cover tree which satisfy what is called the TUISD-property. An algorithm based on the one given in [SD88] is presented for finding such a state cover tree and an UIS set, whenever they exist. In order to minimize the length of the test sequence, the UIDD-method uses solutions to the RPP proposed in [ADLU88] as well as the heuristic algorithms for the general RPP presented in Chapter 2 at appropriate places. It is proved that the fault diagnosis capability of this method is better than that of all the UIS-based methods analyzed in Chapter 3. The method is illustrated by generating a test sequence for the NBS TP4 transport protocol [Nat83] as specified in [SL89]. Our second diagnosis method, namely the CSDD-method, uses a state cover tree and a CS set with a special property. It is proved that the fault diagnosis capability of this method is superior to that of the Wp-method. An adaptive technique is finally proposed in this chapter for further improving the fault diagnosis capabilities of the UIDD-method as well as the CSDD-method.

In Chapter 5, we present a methodology for generating test cases for covering both the control flow and the data flow aspects of protocols which are represented as EFSMs. In contrast to most of the existing methods, we consider the feasibility of the test cases during their generation itself. A new type of UIS, called the Context Independent Unique Sequence (CIUS), is defined and an algorithm is provided to generate a CIUS for each state. CIUSs help in reducing the complexity of test case

generation. We also define, what is called a trans-CIUS-set criterion which requires application of a CIUS of every state at the tail state of every transition. This criterion is superior to the existing control flow coverage criteria for the EFSM. In order to enhance the observability of the def-use associations, we have extended the "all-uses" data flow coverage criterion to the "def-use-ob" criterion. This criterion facilitates external observability of the def-use associations established by the implementation. In order to track the data flow coverage of different walks in an EFSM, a new type of data flow graph is defined and a number of procedures are developed to manipulate the data flow graphs.

Finally, a two-phase algorithm is presented for generating test tours for covering the trans-CIUS-set criterion as well as the def-use-ob criterion. Starting from the initial state the algorithm scans the feasible walks in a breadth-first fashion and selects them if they contribute to the coverage of the trans-CIUS-set criterion for a transition or the def-use-ob criterion for a def-use pair. In the second phase all the walks selected in the first phase are completed into feasible tours ending at the initial state. This task is also done by traversing the feasible walks ending at the initial state in a breadth-first fashion; this is achieved by traversing the walks in the reverse direction. An incremental approach based on Integer Programming techniques is discussed for solving a restricted case of the feasibility problem when the predicates are linear in terms of real and integer variables. This approach demonstrates how the breadth-first search approach in the test case generation algorithm reduces the complexity of the feasibility problem. The methodology is illustrated by generating test cases for the EFSM representation of a major module in a simplified class 2 transport protocol [IS8073] as specified in [Boc90].

6.2 Problems for Future Study

In the test case generation algorithms of Chapter 2 as well as in the UIDD-method, we have used the heuristic algorithms for the general RPP. Thus the optimality of the test sequence depends on the optimality of the solution obtained by the RPP. Therefore, designing heuristic algorithms for the RPP with an optimality bound better than the one given in this thesis deserves further study. To the best of our knowledge, *app_rpt* presented in Chapter 2 is the first heuristic for the RPP with an explicit bound on the optimality of the solution.

The fault diagnosis methods proposed in this thesis assume that the implementations have at most one output or transfer fault. Developing diagnosis methods with a multiple fault model is an important research problem. Thus one of the future research directions is to study how the approach taken in the UIDD-method and the CSDD-method can be applied for diagnosing multiple faults in an implementation.

In order to evaluate the fault detection capabilities of the proposed methodology for the test case generation from the EFSM model, the algorithms presented in Chapter 5 need to be implemented. As the EFSM model considered in this thesis is similar to the underlying model for the FDTs Estelle and SDL, such implementations can then be integrated with the existing FDT tools such as PET/DINGO [SS91b, SS91a], EDT [Bud92] and SDT [Tel93].

As indicated in [LHHT94], automatic test case generation problem for the EFSM model is difficult when a general UIO-sequence is used. Chapter 5 shows that the problem is more tractable for protocols which have a CIUS for every state. Therefore, for an EFSM, the ratio of the number of states which have CIUSs to the total number of states in the EFSM can be considered as a measure of testability. It would be interesting to see how to design a protocol such that this measure is maximum. Also, the methodology presented in Chapter 5 needs to be extended so that it generates test cases for protocols which may not have CIUSs for a few states.

A thorough analysis of the fault detection and diagnosis capabilities of the

existing EFSM-based test case generation methods is needed. Such an analysis would help in achieving a better understanding of the issues involved in the fault detection and diagnosis of the implementations which are modeled as EFSMs.

Test case generation for the communicating EFSM model is an interesting open problem in conformance testing. The methodology proposed in Chapter 5 can be investigated for its applicability in this extended model.

Bibliography

[ADLU88] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification, VIII*, pages 75–86. Elsevier Science Publishers B. V. (North-Holland), 1988.

[BB87] T. Bolognesi and Ed. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN systems*, 14:25–59, 1987.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle : A specification language for distributed systems. *Computer Networks and ISDN systems*, 14:3–23, 1987.

[BDD+91] G. v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Proc. 4th International Workshop on Protocol Test Systems*, Leidschendam, The Netherlands, October 1991.

[BDZ89] G.v. Bochmann, R. Dssouli, and J. R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Tr. Soft. Engg.*, SE-15:1347–1356, November 1989.

[BGS87] G. v. Bochmann, G. Gerber, and J-M. Serre. Semi-automatic implementation of communication protocols. *IEEE Tr. Soft. Engg.*, SE-13(9):989–1000, Sept 1987.

[Boc78] G.v. Bochmann. Finite state description of communication protocols. *Computer Netrworks*, 2:361-372, 1978.

[Boc90] G. Bochmann. Specifications of a simplified transport protocol using different formal description techniques. *Computer Networks and ISDN systems*, 18:335-377, 1989/1990.

[BPY94] G. v. Bochmann, A. Petrenko, and M. Yao. Fault coverage of tests based on finite state models. In *7th International Workshop on Protocol Test Systems, Tokyo , Japan*, November 1994.

[BU91] S. C. Boyd and H. Ural. On the complexity of generating optimal test sequences. *IEEE Tr. Soft. Engg.*, 17(9):976 978, September 1991.

[Bud92] S. Budkowski. ESTELLE Development Toolset (EDT). *Computer Networks and ISDN systems*, 23(5), 1992.

[CA91] W. Chun and P. D. Amer. Test case generation for protocols specified in Estelle. In J. Quemada, J. Manas, and E. Vazquez, editors, *Formal Description Techniques, III*, pages 191-206. Elsevier Science Publishers B. V. (North-Holland), 1991.

[CCK90] M. -S. Chen, Y. Choi, and A. Kershenbaum. Approaches utilizing segment overlap to minimize test sequences. In *Proc. 10th International Symposium on Protocol Specification, Testing and Verification*, pages 67 84, Ottawa, Canada, June 1990.

[Cho78] T. S. Chow. Testing software design modeled by finite state machine. *IEEE Tr. Soft. Engg.*, SE-4(3):178-187, March 1978.

[Chv83] V. Chvatal. *Linear Programming*. W.H. Freeman and Company, New York, USA, 1983.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Tr. Soft. Engg.*, SE-2(3):215–222, September 1976.

[CS92] A. Chung and D. Sidhu. Applications of sufficient conditions for efficient protocol test generation. In *Proc. 5th International Workshop on Protocol Test Systems*, pages 196–205, Montreal, Canada, September 1992.

[CVI89] W. Y. L. Chan, S. T. Vuong, and M. R. Ito. An improved protocol test generation procedure based on UIOs. In *ACM SIGCOMM*, pages 283–294, 1989.

[CZ93] S. T. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Proc. IEEE INFOCOM*, pages 106–114, 1993.

[DS88] A. T. Dahbura and K. K. Sabnani. An experience in estimating fault coverage of a protocol test. In *Proc. IEEE INFOCOM*, pages 71–79, March 1988.

[DSU90a] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar. Algorithmic generation of protocol conformance tests. *AT&T Technical Journal*, 69(1):101–118, January/February 1990.

[DSU90b] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar. Formal methods for generating protocol conformance test sequences. In *Proc. IEEE, Vol 78, No 8*, pages 1317–1326, 1990.

[Edm79] J. Edmonds. Matroid intersection. *Annals of discrete mathematics*, 4:39–49, 1979.

[EJ73] J. Edmonds and E. L. Johnson. Matching, Euler Tours and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1, EATCS Monographs on theoretical computer sceience, 6.* Springer-Verlag, Berlin, 1985.

[FBK+91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Tr. Soft. Engg.*, SE-17(6):591–603, June 1991.

[FGM82] A. M. Frieze, G. Galbiati, and F. Maffioli. On the worst case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[FMC93] ISO SC21 WG1 P54: Information Processing Systems - Open Systems Interconnection - Formal Methods in Conformance Testing, Working Document, June 1993.

[FS92] B. Forghani and B. Sarikaya. Semi-automatic test suite generation from Estelle. *IEE/BCS Software Engineering Journal*, 7(4):295–307, July 1992.

[FW93] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Tr. Soft. Engg.*, SE-19(10):962-975, October 1993.

[Ghe92] A. Ghedamsi. *Diagnostic tests for protocol implementations modeled by finite state machines.* PhD thesis, University of Montreal, Montreal, Canada, December 1992.

[Gil61] A. Gill. State-identification experiments in finite auotomata. *Information and control*, 4:132–154, 1961.

[Gil62] A. Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill, New York, USA, 1962.

[Gon70] G. Gonenc. A method for the design of fault detection experiments. *IEEE Tr. Computers*, C-19:551–558, June 1970.

[Hen64] F. C. Hennie. Fault detecting experiments for sequential circuits. In *Proc. 5th Annual Symposium on Switching Circuits Theory and Logical Design*, pages 95-110, Princeton, N.J., U.S.A., November 1964.

[Hoa85] C. A. R. Hoare. *Communicating sequential Processes.* Printice-Hall International, Englewood Cliffs, New Jersey, USA, 1985.

[Hog92] D. Hogrefe. OSI formal specification case study: the Inres protocol and service, Revised. Technical report, Institute for Informatics, University of Berne, May 1992.

[How87] W. E. Howden. *Functional program testing and analysis.* McGraw-Hill, 1987.

[IS7498] ISO 7498: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, 1984.

[IS8073] ISO TC97/SC6 8073: Information Processing Systems - Open Systems Interconnection - Connection Oriented Transport Protocol Specification.

[IS8807] ISO/IEC 8807: Information Processing Systems - Open Systems Interconnection - LOTOS - a Formal Description Technique Based on the Temporal Ordering of Observational Behavior, June 1988.

[IS9074] ISO/IEC 9074: Information Processing Systems - Open Systems Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model, 1987.

[IS9646] ISO/IEC 9646: Information Processing Systems - Open Systems Interconnection - Conformance Testing Methodology and Framework, 1991.

[JJ89] C. Jard and J.-M. Jezequel. A multiprocessor Estelle-to-C compiler to prototype distributed algorithm on parallel mechine. In E. Brinksma,

G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing and Verification, IX*, pages 161–174. Elsevier Science Publishers B. V. (North-Holland), 1989.

[JL87] J. Jaffar and J. -L. Lassez. Constraint logic programming. In *14th ACM Principles of Programming Languages Conference*, 1987.

[JW74] K. Jensen and N. Wirth. *PASCAL: User manual and report.* Lecture Notes in Computer Science. 18. Springer-Verlag, Berlin, 1974.

[Kar88] G. Karjoth. Implementing process algebra specifications by state machines. In *Proc. 8th International Symposium on Protocol Specification, Testing and Verification* 1988.

[KK68] I. Kohavi and Z. Kohavi. Variable-length distinguishing sequences and their application to the design of fault-detection experiments. *IEEE Tr. Computers*, 17:792–795, August 1968.

[KL67] Z. Kohavi and P. Lavallee. Design of sequential machines with fault detection capabilities. *IEEE Tr. Electronic Computers*, EC-16(4):473–484, August 1967.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[KRK74] Z. Kohavi, J. A. Rivierre, and I. Kohavi. Checking experiments for sequential machines. *Information sciences*, 7:11–28, 1974.

[Kua62] M. -K. Kuan. Graphic programming using odd or even points. *Chinese Math*, 1:273–277, 1962.

[Law75] E. L. Lawler. Matroid intersection algorithms. *Mathematical programming*, 9:31–56, 1975.

[Law76] E. L. Lawler. *Combinatorial optimization : Networks and Matroids*. Holt. Reinhart and Winston. New York. USA. 1976.

[LHHT94] X. Li. T. Higashino. M. Higuchi. and K. Taniguchi. Automatic generation of extended UIO sequences for communication protocols in an EFSM model. In *7th International Workshop on Protocol Test Systems, Tokyo. Japan*. November 1994.

[Lin89] M. T. Liu. Protocol Engineering. In M. C. Yovits. editor. *Advances in Computers*. pages 79–195. Academic Press. 1989. Vol. 29.

[LJH92] Z. Lidong. L. Jiren. and L. Huatian. A further optimization technique for conformance testing based on multiple UIO sequences. In *Proc. 5th International Workshop on Protocol Test Systems*. pages 206–211. Montreal. Canada. September 1992.

[LL91] D. Y. Lee and J. Y. Lee. A well-defined Estelle specification for the automatic test generation. *IEEE Tr. Computers*. 40(4):526–542. April 1991.

[LRK76] J. K. Lenstra and A. H. G. Rinnooy Kan. On general routing problems. *Networks*. 6(3):273–280. July 1976.

[Mac77] A. K. Mackworth. Consitency in networks of relations. *Artificial Intelligence*, 8:99–118. 1977.

[MCS93] H. Motteler. A. Chung. and D. Sidhu. Fault coverage of UIO-based methods for protocol testing. In *Proc. 6th International workshop on protocol test systems*. pages 23–35. Pau. France. September 1993.

[Mil89] A. J. R. G. Milner. *Communication and concurrency*. Addison-Wesley. Reading. Massachusetts. USA. 1989.

[Mor90] L. J. Morell. A theory of fault-based testing. *IEEE Tr. Soft. Engg.*, SE-16(8):844-857, August 1990.

[MP91] R. E. Miller and S. Paul. Generating minimal length test sequences for conformance testing of communication protocols. In *Proc. IEEE INFOCOM*, April 1991.

[MP92] R. E. Miller and S. Paul. Generating conformance test sequences for combined control and data flow of communication protocols. In *Proc. 12th International Symposium of Protocol Specification, Testing and Verification*, 1992.

[Nai92] K. Naik. *Verification of test cases for protocol conformance testing*. PhD thesis, Concordia University, Montreal, Canada, 1992.

[Nat83] National Bureau of Standards, Washington, DC. *Specification of a transport protocol for computer communicatons, vol. 3: class 4 protocol*, January 1983. Report: ICST/HNLP-83-1.

[NT81] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proc. 11th IEEE Fault Tolerent Computing Conference*, pages 238-243, 1981.

[NW88] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization*. John Wiley & Sons , New York, USA, 1988.

[Pap76] C. H. Papadimitriou. On the complexity of edge traversing. *J. ACM.*, 23(3):544–554, July 1976.

[PR88] R. G. Parker and R.L. Rardin. *Discrete Optimization*. Academic Press, San Diego, USA, 1988.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Printice-Hall, New Jersey, USA, 1982.

[Ray87] D. Rayner. OSI conformance testing. *Computer Networks and ISDN systems*, 14:79–98, 1987.

[RDTa] T. Ramalingam, A. Das, and K. Thulasiraman. Analysis of fault detection and diagnosis capabilities of test sequence selection methods based on the FSM model. To appear in Computer Communications.

[RDTb] T. Ramalingam, A. Das, and K. Thulasiraman. On testing and diagnosis of communication protocols based on the FSM model. To appear in Computer Communications.

[RDT93] T. Ramalingam, A. Das, and K. Thulasiraman. On conformance test and fault resolution of protocols based on FSM model. In S. V. Raghavan, G. v. Bochmann, and G. Pujolle, editors, *Computer Networks, Architecture and Applications*, pages 211–222. Elsevier Science Publishers B. V. (North-Holland), 1993.

[RT93] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Tr. Soft. Engg.*, SE-19(6):533–553, June 1993.

[RTD94] T. Ramalingam, K. Thulasiraman, and A. Das. A generalization of the multiple UIO method of test sequence selection for protocols represented in FSM. In *7th International Workshop on Protocol Test Systems, Tokyo, Japan*, November 1994.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Tr. Soft. Engg.*, SE-11(4):367–375, April 1985.

[Sab88] K. Sabnani. An algorithmic technique for protocol verification. *IEEE Tr. Comm.*, 36(8):924–931, August 1988.

[Sar93] B. Sarikaya. *Principles of protocols engineering and conformance testing.* Ellis Horwood, NewYork, USA, 1993.

[SB86] B. Sarikaya and G. v. Bochmann. Obtaining normal form specifications for protocols. In Csaba, Tarnay, and Szentivayni, editors, *Computer Network Usage: Recent Experiences.* Elsevier Science Publishers B. V. (North-Holland), 1986.

[SBC87] B. Sarikaya, G.v. Bochmann, and E. Cerny. A test design methodology for protocol testing. *IEEE Tr. Soft. Engg.*, SE-13(5):518-531, May 1987.

[SD85] K. K. Sabnani and A. T. Dahbura. A new technique for generating protocol tests. In *Proc. 9th Data Communication Symposium*, pages 36-43. IEEE Computer Society press, September 1985.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*, 15:285-297, 1988.

[SDL88] CCITT/SGx/WP3-1, Specification and Description Language, SDL. CCITT Recommendations Z.100, 1988.

[SF87] B. Strausser and J.P. Favreau. User guide for the NBS prototype compiler for Estelle. Technical Report ICST/SNA - 87/3, NIST, October 1987.

[SL89] D. P. Sidhu and T. -K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Tr. Soft. Engg.*, SE-15(4):413-426, 1989.

[SL92] Y. -N. Shen and F. Lombardi. On two graph algorithms for the rural chinese postman tour problem in protocol verification and validation. Technical report, Department of Computer Science, Texas A & M University, College Station, U.S.A., 1992.

[SLD89] Y. -N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. In E. Brinksma, G. Scollo, and

C. A. Vissers, editors, *Protocol Specification, Testing and Verification, IX*, pages 131–144. Elsevier Science Publishers B. V. (North-Holland), 1989.

[SLD92] Y. -N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. *IEEE Tr. Comm.*, 40(8):1282–1287, August 1992.

[SS91a] R. Sijelmassi and B. Strausser. The Distributed Implementation Generator: an overview and user guide. Technical Report NCSL/SNA-91/3, NIST, January 1991.

[SS91b] R. Sijelmassi and B. Strausser. The Portable Estelle Translator: an overview and user guide. Technical Report NCSL/SNA-91/2, NIST, January 1991.

[Sta93] W. Stallings. *Networking Standards: a guide to OSI, ISDN, LAN, and MAN.* Addison-Wesley, New York, 1993.

[SU90] M. H. Sherif and M. U. Uyar. Protocol modeling for conformance testing : Case study for the ISDN LAPD protocol. *AT&T Technical Journal*, 69(1):60–83. January/February 1990.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* Applied mathematics. SIAM, Philadelphia, USA, 1983.

[Tel93] Telelogic, Malmo, Sweden. *SDT reference manual*, 1993.

[TRC93] M. C. Thompson, D. J. Richardson, and L. A. Clarke. An information flow model of fault detection. In *Proc. International Symposium on Software Testing and Analysis*, pages 182–192, Cambridge, USA, June 1993. ACM press.

[Tri92] P. Tripathy. *A unified model for protocol test suite design.* PhD thesis, Concordia University, Montreal, Canada, 1992.

[TS92] K. Thulasiraman and M.N.S. Swamy. *Graphs:Theory and algorithms.* John Wiley & sons, New York, USA, 1992.

[Tur93] K. J. Turner, editor. *Using formal description techniques.* John Wiley & Sons, Chichester, England, 1993.

[Ura87] H. Ural. Test sequence selection based on static data flow analysis. *Computer Communications*, 10(5):234-242, October 1987.

[Ura92] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311-325, June 1992.

[UW93] H. Ural and A. Williams. Test generation by exposing control and data dependencies within system specifications in SDL. In *Proc. FORTE'93*, October 1993.

[UY91] H. Ural and B. Yang. A test sequence selection method for protocol testing. *IEEE Tr. Comm.*, 39(4):514-523, April 1991.

[VK90] S. T. Vuong and K. C. Ko. A novel approach to protocol test sequence generation. In *IEEE Global Telecomm. Conference and Exhibition*, pages 1880-1884, December 1990.

[Wey84] E. J. Weyuker. The complexity of data flow criteria for test data selection. *Information processing letters*, 19:103-109, August 1984.

[Wey90] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Tr. Soft. Engg.*, SE-16(2):121-128, February 1990.

[WL93] C.-J. Wang and M. T. Liu. Generating test cases for EFSM with given fault models. In *Proc. IEEE INFOCOM*, pages 774-781, 1993.

[YPB93] M. Yao, A. Petrenko, and G. v. Bochmann. Conformance testing of protocol machines without reset. In *Proc. 13th International Symposium on Protocol Specification, Testing and Verification*, 1993.

[Zaf78] P. Zafiropulo. Protocol validation by duologue-matrix analysis. *IEEE Tr. Comm.*, 26(8):1187–1194, August 1978.

[ZWR+80] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand. Towards analyzing and synthesizing protocols. *IEEE Tr. Comm.*, 28(4):651–661, April 1980.