National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

NL-339 (r. 88/04) c

Canadä

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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 accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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 propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-56372-9

Canada

Guidelines for Assessing the Completeness of Protocol Conformance Test Suites

By

Teddy T. Boyce, B.Sc.

A thesis submitted to
the School of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science
November, 1988

Department of Computer Science
University of Ottawa
Ottawa, Ontario
CANADA

Teddy T. Boyce, Ottawa, Canada, 1989

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

The International Organization for Standardization has given certain guidelines as to what should be included in a protocol conformance test suite in order to make it complete. One of the suggested requirements is a set of tests relating to each protocol 'phase.' This thesis examines this aspect of completeness of the test suite. Assuming that a formal description technique such as Estelle has been used to specify the protocol, a phase is represented as an equivalence class of subpaths within the protocol specification, and a path-oriented model is developed. Based on this model, a coverage metric is then derived by which the *phase coverage* of a test suite may be measured. The method to derive a value for coverage also incorporates a procedure to give some indication as to the possible feasibility of the test sequences that make up the test suite. It is the conclusion of this thesis that total phase coverage is a requirement for completeness of conformance test suites.

Key Words: protocol specification, protocol implementation, conformance testing, coverage criterion, extended finite state machine, protocol phase.

ACKNOWLEDGEMENTS

I would like to thank my research supervisor, Dr. Robert L. Probert for his guidance and encouragement, and my colleague Shahram Akhavan for his many helpful discussions and comments regarding TTCN. I would also like to thank my wife Jacqueline for her understanding, patience and support while I was preparing this thesis. Assistance for this course of study was provided in most part by a Canadian Commonwealth Scholarship awarded under the Commonwealth Scholarship and Fellowship Plan. Some assistance was also provided by the Natural Science and Engineering Research Council of Canada and Bell-Northern Research.

DEDICATION

To my wife Jacqueline.

Table of Contents

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF TERMS AND SYMBOLS xii

1. INTRODUCTION 1
   - 1.1 BACKGROUND 1
   - 1.2 MOTIVATION 2
   - 1.3 CONTRIBUTION OF THESIS 2
   - 1.4 ORGANIZATION OF THE THESIS 3

2. PROTOCOL CONFORMANCE TESTING STANDARDS 4
   - 2.1 INTRODUCTION 4
     - 2.1.1 The OSI Reference Model 4
     - 2.1.2 Protocol Specification 7
   - 2.2 CONFORMANCE TESTING 8
     - 2.2.1 Need for Conformance Testing 8
     - 2.2.2 Objectives of Conformance Testing 8
     - 2.2.3 Abstract Test Methods 9
     - 2.2.4 Test Center Architecture 12
   - 2.3 THE CONFORMANCE TEST SUITE 13
     - 2.3.1 Test Suite Structure 13
     - 2.3.2 Test Suite Specification 14

3. TEST COVERAGE CRITERIA AND COVERAGE MEASUREMENT 17
   - 3.1 OVERVIEW OF TESTING TECHNIQUES 18
     - 3.1.1 White Box Testing 18
     - 3.1.2 Black Box Testing 18
     - 3.1.3 Grey Box and FSM Based Testing 19
   - 3.2 COVERAGE CRITERIA 19
     - 3.2.1 Code and Model Coverage 19

3.2.2 Fault Coverage 23
3.3 COVERAGE MEASUREMENT TECHNIQUES 24
3.3.1 Techniques for Measuring Code Coverage 24
3.3.2 Techniques For Measuring Fault Coverage 25
3.4 SUMMARY 25

4. A METHOD FOR MEASURING TEST COVERAGE BASED ON SPECIFIED SYSTEM BEHAVIOUR 27
4.1 DEVELOPMENT OF A REPRESENTATIONAL MODEL 28
4.1.1 The Modified Breadth First Numbering Scheme 28
4.1.2 Basic Properties of Paths in the Module 32
4.1.3 Characteristics of DBMs 35
4.1.4 Identification of Phases 37
4.1.5 Construction and Representation of Phases 40
4.1.6 Joining Phases Together 41
4.1.7 Relationship Between Protocol Specification and Set of DBMs and ERMs 42
4.2 CLASSIFICATION OF PATHS IN A DBM 43
4.2.1 Feasibility Model: Terminology 43
4.2.2 Algorithm to Classify Paths in a DBM 47
4.2.3 An Application of the Algorithm 49
4.3 DETERMINATION OF TEST COVERAGE 52
4.3.1 Restrictions on Paths for Generating Test Cases 52
4.3.2 A Test Coverage Metric—Phase Coverage 53
4.3.3 Test Suite Decomposition 55
4.3.4 Measurement of Phase Coverage 57

5. ASSESSMENT OF METHOD 59
5.1 COMPARISON WITH RELATED TECHNIQUES 59
5.2 EXPERIENCE WITH ISDN LAPD 62

6. CONCLUSION 66
6.1 SUMMARY OF WORK 66
6.2 DIRECTIONS FOR FUTURE WORK 67

REFERENCES 69

APPENDIX 1 73
A1.1 Phase Decomposition of LAPD (Q.921) 73

APPENDIX 2 88
A2.1 Test Suite Overview 88
A2.2 PCO and Timer Declarations 88
A2.3 Constant and Variable Declarations 89
A2.4 Abbreviation Declarations 90
A2.5 Declaration of ASPs at pco1 90
A2.4 Declaration of ASP"s at pco2 93
A2.5 Dynamic Behaviour Descriptions 94
A2.6 Constraints Part 102

APPENDIX 3 104
A3.1 FORMAL SPECIFICATION OF TEI ASSIGNMENT PHASE OF LAPD 104

List of Tables

4.1 Results of applying Algorithm CLASSIFY to path $\{t_1, t_3\}$ of figure 4.5 51

4.2 Results of applying Algorithm CLASSIFY to path $\{t_1, t_2, t_3\}$ of figure 4.5 51

5.1 Paths through phases corresponding to test cases in LAPD test suite 63

5.2 Feasibility of test paths in LAPD test suite 64

5.3 Phase coverage of LAPD test suite 64

A2.1 Names of Phases Referenced in Figures A2.1 and A2.2 74

List of Figures

2.1 OSI reference model architecture 5
2.2 Communication path between peer entities 6
2.3 Mapping of data units between adjacent layers 6
2.4 The four abstract test methods 11
2.5 Major hardware components of Test Center 12
2.6 Major software components of Test Center 13
2.7 Hierarchical structure of test suite 14
4.1 MBF Node numbering and Paths from S to T 29
4.2 Simple and Complex cycles 34
4.3 Grouping error related transitions into modules 36
4.4 Predicates and Transitions 43
4.5 Complete Paths from Link Establishment phase of LAPD 50
4.6 Paths in connected phases 54
4.7 Mapping from DBM to test suite 55
4.8 Overview of coverage determination 56
A1.1 Phase Dependency Diagram for Normal Behaviour Phases 73
A1.2 Phase Dependency Diagram for Error Related Phases 74
A1.3 Phase: TEI Assignment 75
A1.4 Phase: Link Establishment 76
A1.5 Phase: Data Transfer 77
A1.6 Phase: Data Transfer Timeout Recovery 77
A1.7 Phase: Link Re-establishment 78
A1.8 Phase: L3 Initiated Link Release 79
A1.9 Phase: Peer Initiated Link Release 80
A1.10 Phase: TEI Removal 81
A1.11 Phase: Deactivation/TEI unassigned 82
A1.12 Phase: Deactivation/TEI assigned 82
A1.13 Phase: T200 Timeout Recovery 83
A1.14 Phase: TEI Assignment Error Recovery 83
A1.15 Phase: Attempted Peer Initiated Re-establishment 84
A1.16 Phase: Invalid Frame Error Recovery/1 84
A1.17 Phase: Invalid Frame Error Recovery/2 85

A1.18 Phase: Unsolicited Response Error Recovery/1 85
A1.19 Phase: Unsolicited Response Error Recovery/2 86
A1.20 Phase: Unsolicited Response Error Recovery/3 86
A1.21 Phase: Sequence Error and FRMR Recovery 87

List of Terms and Symbols

ASP	Abstract Service Primitive (observed locally).
ASP"	Abstract Service Primitive (observed remotely).
C	Cycle.
C_ρ	Connected phase path complexity of protocol specification.
DBM	Directed Behaviour Module.
DD_x	Disallowed Domain set for action_dependent variable x
ERM	Error Recovery Module.
FDT	Formal Description Technique.
FSM	Finite State Machine.
IPE	Implemented Protocol Entity.
ISO	International Organization for Standardization.
IUT	Implementation Under Test.
LCSAJ	Linear Code Sequence and Jump.
m_i	Node i of DBM or ERM.
$N(m_i)$	Label number of node m_i as assigned by the Modified Breadth First numbering scheme.
$\eta(P_i)$	Number of complete paths through phase i.
Path	Sequence of transitions in a DBM or ERM with no consideration give to this sequence as part of a test case.
PCI	Protocol Control Information.
PCO	Point of Control and Observation.
PDU	Protocol Data Unit.
P_i	Phase i of protocol.
PICS	Protocol Implementation Conformance Statement.
PS	Protocol Specification.
OSI	Open Systems Interconnection.
ρ	Path in DBM.
S	Set of start nodes of DBM.
SDU	Service Data Unit.
SUT	System Under Test.
t	Transition.
T	Set of target nodes of DBM.

Trace	Sequence of transitions (inputs and outputs) in a **DBM** or **ERM** which represents a test sequence which is intended to be used as (part of) a test case.
Trace(PS)	Set of traces in *PS*.
$Trace_R(PS)$	Restricted trace set of *PS*.
Trans(PS)	Set of transitions of *PS*.
TS	Transition sequence.
TTCN	Tree and Tabular Combined Notation.

Chapter 1

1. Introduction

1.1 BACKGROUND

In the past few years, there has been an increase in the number of computer systems which are interconnected via networks, both private and public. Correspondingly, there has also been an increase in the number of requirements made on these networks, and this has led to the need to further develop protocols which help to fulfil these requirements. These communication protocols are by no means simple pieces of software. They are difficult to specify and implement.

As difficult as they are to implement, however, it is thought that the thorough testing of such complex software is an even more difficult task [Myer79]. Protocol implementations are generally subjected to conformance testing in order to ascertain the extent to which the implementation conforms to the specification from which it was derived. The achievement of this goal is, however, not an easy process. One of the biggest hurdles to be overcome is the derivation of the conformance test suite (CTS) itself. Much work has been done in this area; see for example [SaBo84, BuEc86, SiLe86, SaBo87]. By far, the most difficult task in the production of a CTS is the derivation and selection of effective, representative test cases. The aim of the test suite designer should be to optimize the test suite, i.e. to capture as many errors as possible with as few tests as are necessary, thus ensuring that the test cases are effective and leading to more economical testing. However, even though a test suite may contain redundant or irrelevant test cases, their detection and elimination rely on subjective judgement [ShHo86].

One requirement of a CTS is that it covers the protocol specification. This is further discussed in chapter 3 of this thesis. However, suffice it to say at this point that coverage is

obtained by having the proper representative selection of discriminating test cases. The determination of this 'proper selection' can prove to be very difficult, for again the determination of coverage of a test suite involves much subjective judgement [LiFr86]. It appears that much less work has been done in the determination of coverage of a CTS than in the generation of the test suites themselves [SaDa85, SiLe86].

Although a number of formal methods (some of which will be presented in chapter 3) are available for the derivation of test cases, getting the proper number of them and indeed the right combination of them for proper coverage requires subjective judgement and is a difficult problem. It is the aim of this thesis to look at the coverage of a protocol by a test suite in this light, and to suggest a method whereby the coverage could be checked and perhaps improved upon.

1.2 MOTIVATION

Because it is necessary to have confidence in the implementation of a protocol, and in the light of the difficulty of producing a 'good' test suite, it is desirable to be able to judge whether a CTS adequately covers a particular protocol specification. Techniques used to gauge the coverage of CTSs either depend on the application of reachability analysis to the protocol graph [BuEc86], producing a reachability tree against which the test suite is compared, or on simulation techniques [HeKr87] that attempt to show how many types of errors the test suite can detect. These techniques are based on 'syntactic' considerations. It is thought, however, that a technique based on 'semantic' considerations might better provide meaningful coverage data, and a certain amount of semantic information could be incorporated into the results of coverage determination. Thus it is thought that a functional approach to the problem of coverage determination is worthy of study.

1.3 CONTRIBUTION OF THESIS

This thesis presents a method of decomposing a protocol specification that is not only useful as an aid in determining the coverage of an existing candidate test suite, but also in the generation of new conformance tests for the protocol. This decomposition is based on the phases of the protocol. The concept of a phase is formalized as a useful tool for the advancement of conformance testing, and a model is developed that allows the representation of these phases and hence the decomposition of the protocol. A new metric—phase coverage—is developed as one of the goals to be satisfied by a CTS. A main thrust of this thesis is the application of this phase decomposition of the protocol to decompose the CTS. A method, employing both the decomposition of the protocol and the

test suite, is presented to determine the phase coverage of the CTS. In this way, an estimate can be made of the relative completeness of the CTS.

1.4 ORGANIZATION OF THE THESIS

Chapters 2 and 3 present the basic background material necessary for understanding the thesis and also for putting this work in perspective. Chapter 2 gives an overview of conformance testing and chapter 3 provides a summary of the more popular test coverage criteria and coverage measurement techniques currently used, together with an appraisal of shortcomings of these methods. The concept of phase coverage and its determination are developed in chapter 4, while an example of an application of the technique to an ISDN protocol is presented in chapter 5. Chapter 6 then presents the conclusions of this research, and gives some suggestions for future work.

Chapter 2

2. Protocol Conformance Testing Standards

2.1 INTRODUCTION

The International Organization for Standardization (ISO) has recognized that if many computer systems are to interwork properly, there must be a standard approach in designing the protocols which realize this interworking. For this reason, it has introduced the Open Systems Interconnection (OSI) Basic Reference Model [ISO2]. In order that these aims be achieved, it is necessary to validate the implementation of protocols against their specifications. Thus these implementations need to be tested as thoroughly as possible. This is a difficult task. Based on the ISO OSI reference model architecture, some test methods have been developed to facilitate the conformance testing of an implementation against its specification. We shall look at these shortly, but first we will review some concepts and background material needed for an appreciation of conformance testing in the following sections.

2.1.1 The OSI Reference Model

The OSI reference model was developed to provide a conceptual and functional framework to allow protocol designers to independently develop protocols needed to control and direct the exchange of information between open systems. The underlying aim therefore is to see an increase in the number of systems that are able to interwork correctly. The OSI reference model [ISO2] describes an open system as a representation of a set of one or more computers, associated software, etc. which complies with the requirements of OSI standards in its communication with other real open systems.

The architecture of the OSI reference model facilitates protocol design. This is achieved by a layered organization, each layer offering certain services to the one immediately above it, releasing the upper layer of the burden of implementing these services. According to the model, an open system is logically composed of seven layers. Figure 2.1 shows this concept. Any particular layer of the model is referred to as an (N)-layer. Within each (N)-layer, there are a number of (N)-entities which perform certain functions. An (N)-layer (in conjunction with the layers below it) provides an (N)-service to the (N+1)-entities in the layer above it. This service is provided at the boundary between the (N)-layer and the (N+1)-layer at a point called an (N)-service-access-point.

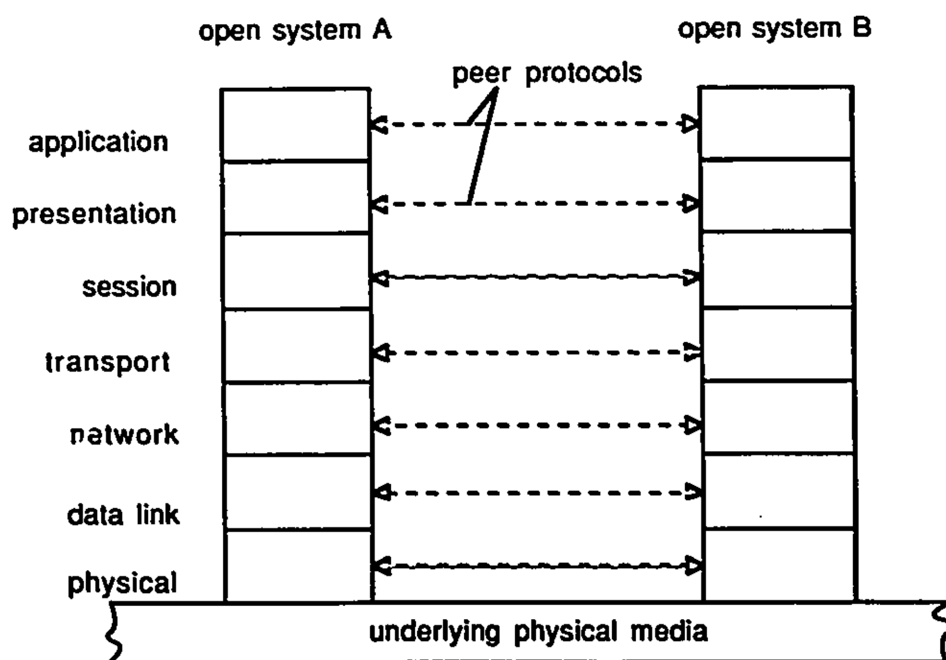


Figure 2.1: OSI reference model architecture

Entities of the same layer but in different open systems are referred to as peer entities. Peer (N)-entities transfer information between themselves via an (N–1)-connection, under the direction of a peer (N)-protocol. When an (N)-entity wishes to send data to its peer, the (N)-protocol assembles the data into one or more (N)-protocol-data-units (PDU), which consist of user data plus (N)-protocol-control-information (PCI). The (N)-PDUs are assembled into (N–1)-service-data-units (SDU) and passed to the (N–1)-layer. In the (N–1)-layer and below, the process is repeated until the physical layer is reached, within which the actual electrical data is transmitted over the physical medium to the other system. There the data moves up the (logical) layers until the corresponding (N)-

layer is reached and the correspondent (N)-entity receives the sent data. All of the activities below the (N)-layer however are transparent to the (N)-entities. Figure 2.2 shows the communication between two peer entities in different open systems, while figure 2.3 shows the relationship between PDUs moving from one layer to the next lower layer. We should note that these layers are logical only, and some manufacturers will not provide distinct access to some layers.

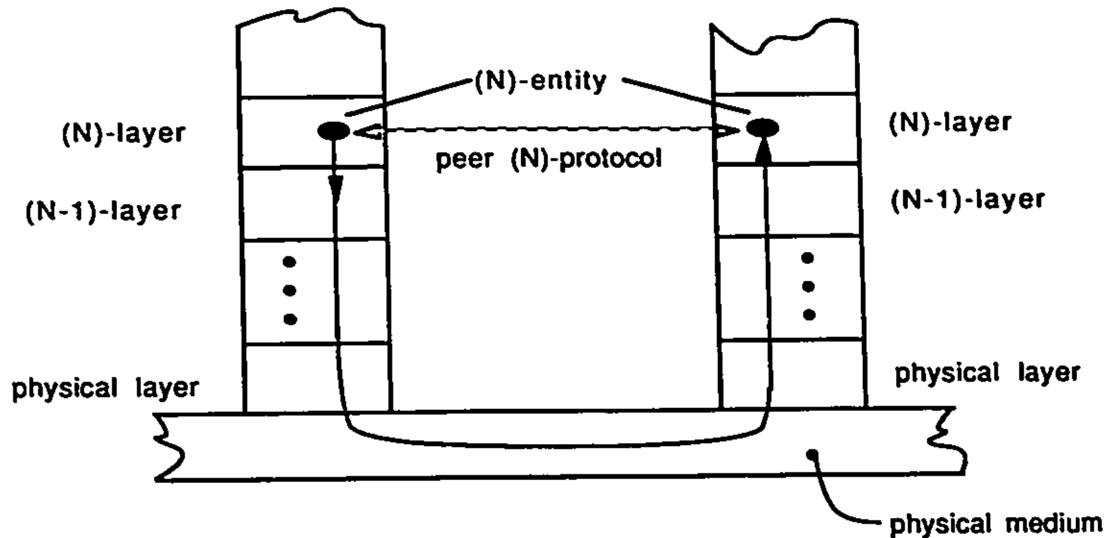


Figure 2.2: Communication path between peer entities

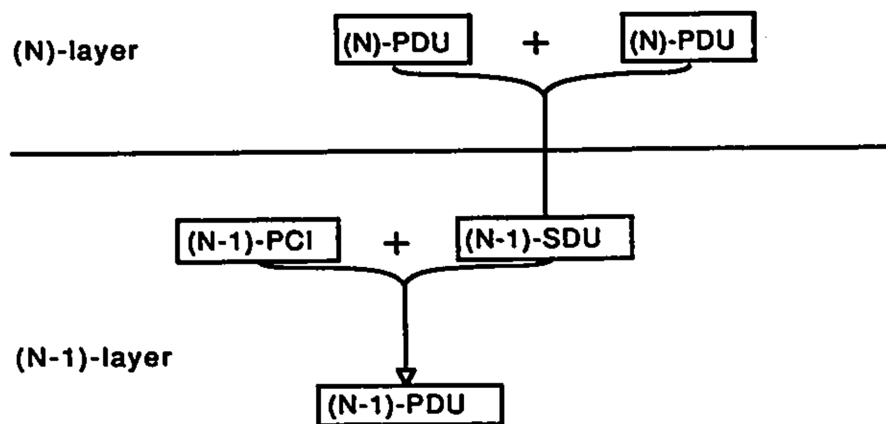


Figure 2.3: Mapping of data units between adjacent layers

2.1.2 Protocol Specification

Each layer of the OSI reference model contains entities which communicate with their peers according to a peer protocol. To ensure orderly, error-free communication, these rules can get quite complex. Before any complex protocol can be implemented, however, it is necessary to have a precise description of the protocol. This description consists of two parts, the protocol service definition and the protocol specification.

The protocol service definition is a description of the services that an entity provides to the entities in the layer above it. It also includes a description of the service primitives at the layer interfaces and constraints relating to the order in which these primitives may be executed by communicating entities. In addition, information on properties such as throughput, delay, probabilities of occurrence of certain errors, etc. is included.

The protocol specification is a description of the protocol on a less abstract level. It includes, among other things, an informal description of the protocol entities, a description of the PDU encodings and formats, and the observable behaviour of each entity on reception of user data or peer PDUs in addition to reaction to internal events such as timeouts [BoSu80].

There are many ways of describing the protocol's externally observable behaviour. Many protocols have been described in plain English. However, this may lead to ambiguities. For purposes of implementation and the derivation of test cases, it is desirable that the protocol be specified using a formal description technique (FDT). FDTs for protocol specification fall into three main groups, *viz* (1) transition models, (2) programming languages, (3) mixed models (hybrid of (1) and (2)). Two of the most popular FDTs currently in use are LOTOS [ISO7] and Estelle [ISO1]. They are being progressed by the ISO as possible standard FDTs for the specification of protocols.

LOTOS is a specification language that is derived from an extension of Milner's Calculus of Communicating Systems [Miln80]. In LOTOS, a distributed system is considered to be a process which consists of many interacting sub-processes. Each sub-process can then be viewed in the same way. Thus a LOTOS specification is a hierarchical definition of a number of processes and their interactions. These interactions are specified using certain LOTOS operators, and the specification of the processes is aided by the ability to abstractly represent value expressions and data structures using ACT ONE, a language for the specification of abstract data types [BoBr87].

Estelle is based on an extended state transition model, developed from the finite state machine formalism, and an extension of the Pascal programming language. In the extended

model, it is possible to associate variables with states and predicates with transitions. In Estelle, a system is considered to be a module, which itself can be comprised of several modules. An Estelle specification then is also hierarchical. It consists of a definition of module instances which exchange messages via links between their interaction points. An Estelle specification for one phase of the LAPD protocol [CCIT2] is given in appendix 3.

Both LOTOS and Estelle fulfil the objectives to be satisfied by an FDT as set out by the ISO, *viz* they are expressive, well defined, well structured and offer abstraction facilities. It is likely that, in the near future, we will see some version of one of these FDTs emerge as the standard language to be used in the specification of OSI protocols.

2.2 CONFORMANCE TESTING

2.2.1 Need for Conformance Testing

One of the aims of the ISO OSI is to allow more computer systems to be connected to each other, to share resources, and to exchange information among themselves. As stated earlier, the OSI reference model was developed as a framework to allow protocol designers to develop standard international specifications for OSI protocols. It is hoped that as a result of the existence of these standards, the aims of the ISO would be better realized. However, the existence of these standards by itself is no guarantee that real systems implementing protocols developed according to a standard will be able to interwork.

Inability to interwork may be the result of ambiguities in the specification, leading to different interpretations of the same standard. In addition, some functions and facilities may be optional, and some manufacturers may choose to implement only some or none of them. To make the situation worse yet, some manufacturers may implement different versions of the same standard, where perhaps the protocol may have been improved over a number of years. This is the case of X.25, for example.

It is therefore desirable that before a real system is connected to a network of open systems, the extent to which it is able to interwork with the other systems be ascertained. To do this, it is necessary to test the conformance of this new system to the protocol standard from which it was implemented. The conformance of the system is important, as this will help realize (but not ensure) the aim of the ISO OSI model, and the widespread interoperability of computer networks.

2.2.2 Objectives of Conformance Testing

Conformance testing is the testing of an implementation (called the implementation under test, or IUT) to determine to what extent the IUT conforms to the relevant standard

specification of the protocol being implemented. This includes both static (constraints) and dynamic (behavioural) conformance requirements [ISO3].

Four types of conformance tests have been identified in [ISO3], each providing a different indication of the degree of conformance of the IUT. These types are:

1. *Basic interconnection tests:* These test the main features in the relevant standard specification. The aim is to check whether there is enough conformance of the IUT to allow interconnection and subsequent interworking, without performing thorough testing.
2. *Capability tests:* These provide limited testing of the static conformance requirements (mandatory and optional features), and check that the observable capabilities of the IUT are consistent with these requirements and the protocol implementation conformance statement (PICS).
3. *Behaviour tests:* These are intended to provide as thorough testing as possible. Therefore, the full range of the dynamic conformance requirements as given in the specification should be covered.
4. *Conformance resolution tests:* These tests aim to provide a definite yes/no answer to questions pertaining to the degree of conformance of an IUT to specific requirements. These tests are useful, for example, in trying to determine the cause of a specific interoperability problem.

Depending on the degree of conformance that the tester is trying to determine, tests from all four categories may be used. It is clear, however, that behaviour tests will make up the bulk of any test suite.

2.2.3 Abstract Test Methods

In conformance testing, the IUT is considered to be a black box, that is, only the externally observable behaviour of the IUT can be used in the determination of its conformance to the specification. Based on this and also on the OSI reference model architecture, a number of abstract test methods have been proposed which take into account what outputs from the protocol entity under test can be observed and what inputs to it can be controlled.

The OSI reference model architecture assumes that the IUT be an implementation of one or more adjacent layer protocols. In the description that follows, the highest (top) layer of the IUT is referred to as N_t and the lowest (bottom) layer as N_b. Obviously, for a single-layer IUT, N_t=N_b. In the OSI standard specifications, the behaviour of a protocol entity is

described in terms of the PDUs and the abstract service primitives (ASP) that move across the layer boundaries above and below that entity. Thus for the IUT considered here, the behaviour would be described in terms of (N_t)-ASPs and (N_b–1)-ASPs, including (N_t) to (N_b)-PDUs.

An abstract test method is described by identifying the points closest to the IUT at which control and observation can be exercised. These are called points of control and observation (PCO). An upper tester (PCO at the upper boundary of the IUT) and a lower tester (PCO at the lower boundary of the IUT) are associated with the abstract test methods. Cooperation and synchronization between upper and lower tester must be maintained, and this is normally provided through Test Coordination Procedures. Four categories of abstract test methods have been defined, one local and three external. In the external test methods, the lower tester is located remotely from the system under test (SUT) which contains the IUT, and connected to it by a telecommunications link. A distinction is made between ASPs which can be observed locally (represented by ASPs) and those which must be observed remotely (represented by ASP"s). We will now look briefly at these categories of test methods.

The *local test method* requires direct or indirect access to both the lower and upper boundaries of the IUT and a mapping between specified ASPs and their realization within the SUT. This allows control and observation of (N_t)-ASPs, (N_b–1)-ASPs and (N_t) to (N_b)-PDUs. The upper and lower testers are required to achieve control and observation of the effects of the specified ASPs and test coordination procedures, and are both assumed to be an integral part of the SUT [ISO3].

The *distributed test method* is much like the local test method except that the lower tester is remotely located from the IUT. Thus this method requires direct or indirect access only to the upper boundary of the IUT. At this boundary, the upper tester is required to achieve control and observation of the effects of the (N_t)-ASPs. It is also required to realize the test coordination procedures. The (N_b–1)-ASP"s and (N_t) to (N_b)-PDUs are controlled and observed remotely by the lower tester.

In the *coordinated test method*, no assumption is made regarding the upper boundary of the IUT. Therefore it is not assumed that the (N_t)-ASPs can be controlled and observed. The test coordination procedures are specified explicitly in a standardized test management (TM) protocol, which the upper tester is required to implement. (N_b–1)-ASP"s, (N_t) to (N_b)-PDUs, along with TM-PDUs are controlled and observed over the telecommunications link.

The *remote test method* allows for the case where it is not possible to have a point of control and observation at the upper boundary of the IUT. Therefore in reality there is no upper tester and no (N_t)-ASPs to be controlled and observed. No assumptions are made regarding the feasibility or realization of test coordination procedures, even though the SUT may itself be required to fulfil some of these requirements. This is represented in figure 2.4(d) by dashed lines. The (N_b-1)-ASP"s, with the (N_t) to (N_b)-PDUs are controlled and observed as for the other external test methods. Even in this method, it is still possible by means of a second control link to advise an SUT operator of expected upper layer responses and of requirements for the initiation of SUT actions via operator service requests.

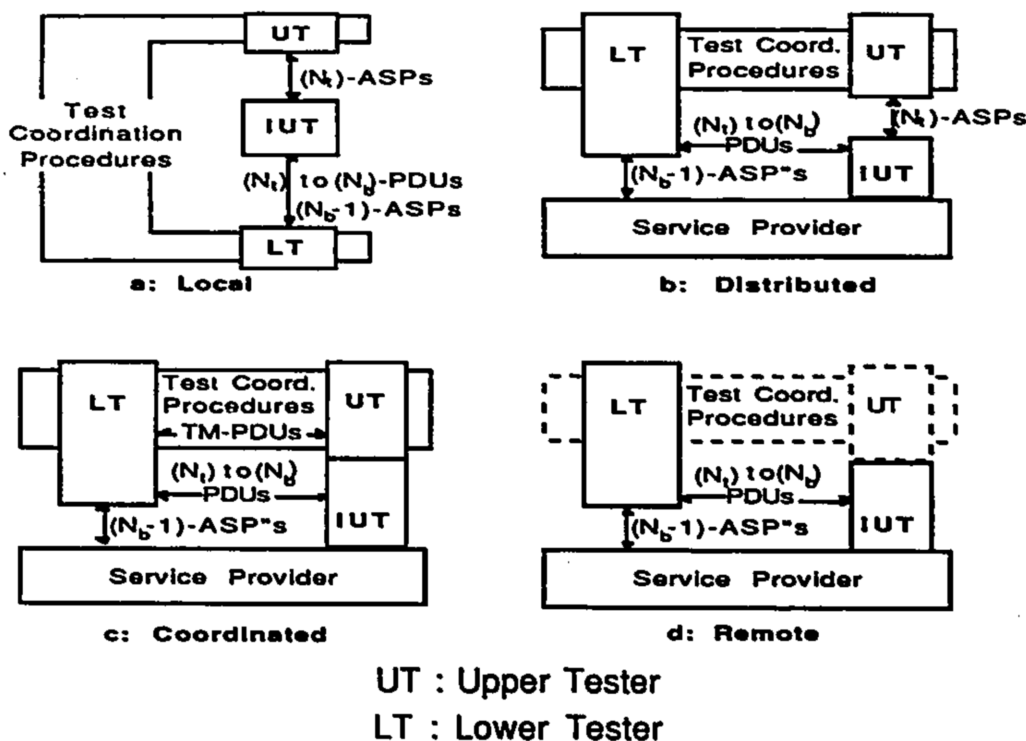


Figure 2.4: The four abstract test methods

Figure 2.4 illustrates the concepts of these four categories of test methods. For each of these four methods, there are single-layer, multi-layer and embedded variants. In the single-layer variant, the IUT is considered to be a single layer in the OSI reference model. In the multi-layer variant, the IUT is an implementation of a set of adjacent layers which are to be tested as a whole. In the embedded variant, the IUT is also an implementation of a set

of adjacent layer protocols, but each layer is tested separately, to the degree that this can realistically be achieved.

2.2.4 Test Center Architecture

Protocol testing is becoming largely automated, and centers employing automated test systems are gaining acceptance [Kanu86]. The realization of the test methods described above is achieved through a combination of hardware and software. As found in [Kanu86] and [ShHo86], the typical hardware configuration of such a center might be as shown in figure 2.5. The major components consist of a computer which would be connected to a data base containing the conformance test suite, and would be loaded with a test driver to execute the test cases and produce reports. The computer is connected to a protocol emulator, which handles the encoding/decoding, etc. of data sent to and received from the SUT. The SUT itself is connected to the protocol emulator via a telecommunications link.

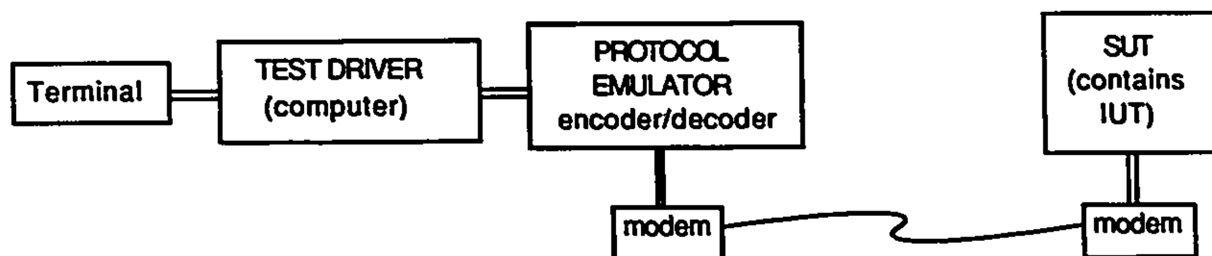


Figure 2.5: Major hardware components of Test Center

The major software components of a test center and their relationship to each other are depicted in figure 2.6. Based on [ShHo86], the functions of these might be as follows.

The *Data Base* contains all the selected test cases available for that IUT.

The *Session Handler* allows the test services customer to state which test cases are to be actually executed, and in addition to define a parameter file containing the parameters of the test cases.

The *Test Driver* controls the execution of the selected test cases.

The *Protocol Handler* formats the frames sent to the IUT and decodes the frames received. In addition, it transmits copies of all frames sent to or received from the IUT to the Test Logger.

The *Test Logger* records all messages sent to and received from the IUT.

The *Test Analyzer* determines whether a test is passed or failed by comparing the response of the IUT with the expected response as given in the test case.

The *Report Generator* prints the results of a test session. The results are a combination of the outputs of the Test Logger and Test Analyzer.

Test centers are rapidly evolving, and the above discussion serves only to give a basic representation of their architecture. As time goes on, they will become much more sophisticated and complex. Moreover, test centers will need to adhere to the guidelines specified in draft standard DP9646—Parts 4 and 5 [ISO5, ISO6].

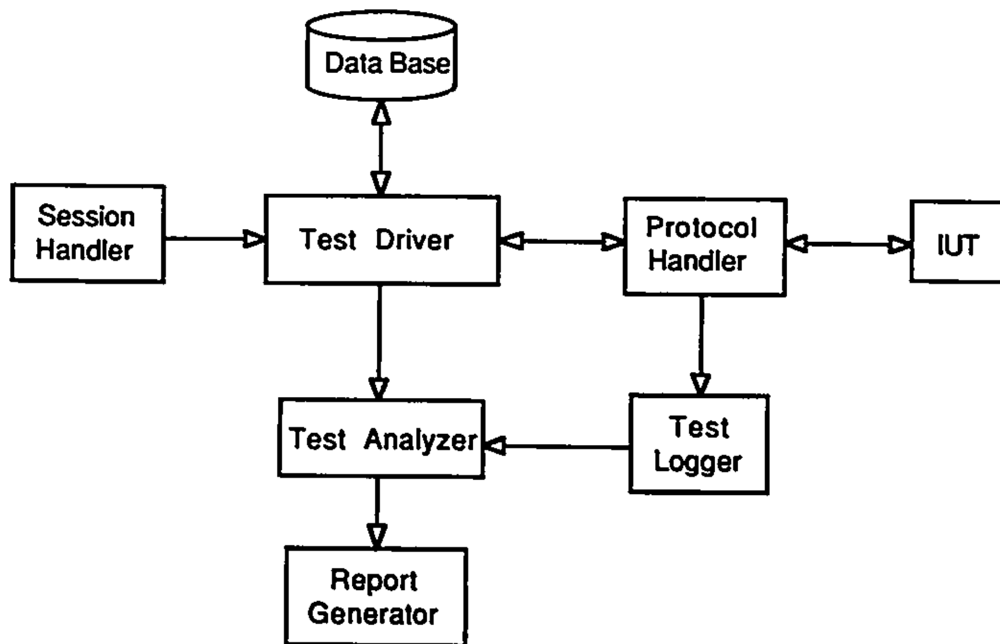


Figure 2.6: Major software components of Test Center

2.3 THE CONFORMANCE TEST SUITE

2.3.1 Test Suite Structure

A conformance test suite can be considered as a hierarchical structure, as shown in figure 2.7. At the uppermost level, the test suite is made up of a number of test groups. A test group is a set of related test cases which can themselves be broken down into test steps and test events. It should be noted however, that a test case can be in more than one test group. A conformance test suite then is a complete set of test groups necessary to perform conformance testing, along with the information needed to determine the order in which they would be executed when in executable form [ISO3].

For each test case in the test suite, an associated purpose should be explicitly specified. Test cases in a particular test group are normally related according to their purposes. Two broad classes of test groups must be present in a test suite. They are concerned with capability testing and behaviour testing. The behaviour test group may be further divided into sub-groups dealing with valid behaviour, syntactically invalid behaviour, and inopportune behaviour. In addition, there may be groups focusing on PDUs sent to and received from the IUT (involving encoding variations, timer variations, individual parameter variations and combinations of parameter variations), mandatory features, optional features which were implemented, each protocol phase, timer variations, and PDU/parameter variations.

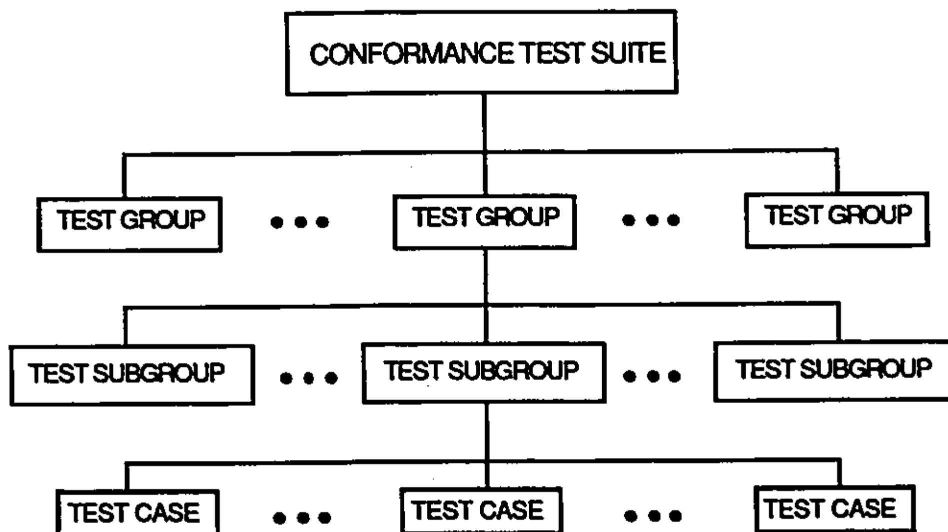


Figure 2.7: Hierarchical structure of test suite

In the above categories, similar test sub-groups may occur in more than one higher level test group. The test designer must therefore work to ensure as little redundancy as possible in his conformance test suite.

2.3.2 Test Suite Specification

A conformance test suite can be specified at many different levels of detail. The level of detail of the test suite specification is directly related to the level of detail of specification of the test cases that compose the test suite. There are three levels of detail that are currently

considered in the standards [ISO3]: *generic, abstract* and *executable*. These terms apply both to the test suite as well as to the test cases.

A *generic test case* can be considered as a refinement of its test purpose. A generic test suite will consist of one generic test case for each identified test purpose. A generic test case consists of three parts, *viz* the preamble, the test body and the postamble. These are defined as follows:

- The preamble is a textual description of the path from a stable state to the initial state in which the test body starts. The initial state includes the protocol state as well as information about the state of the SUT.
- The test body is an adaptive sequence of test events (i.e. a path) which allows a verdict of 'pass,' 'fail' or 'inconclusive' to be assigned to the outcome of the test. The assignment of these verdicts is directly related to the test purpose.
- The postamble is a textual description of the actions necessary to take the system to a stable state after the test body is run, and to verify that the response to the test body is appropriate.

An *abstract test case* is derived from a generic test case. It is specified in terms of a particular abstract test method, and gives a more precise specification of the sequence of events described in the generic test case. In addition, it adds (using a specialized notation) the sequences of events necessary to achieve the preamble and postamble. Since an abstract test case is based on a particular test method, many abstract test cases can be derived from a single generic test case.

An *executable test case* is derived from an abstract test case, and consists of all necessary data in such a form as to allow it to be run on a real IUT. Subsequently, a specific subset of tests is *selected* as appropriate to that particular SUT.

There are many different ways of specifying a test suite. The ISO is currently working on a test specification language for specifying both generic and abstract test suites. The language, called Tree and Tabular Combined Notation, or TTCN [ISO4], is based on a tree notation, but is displayed in tabular form. This tabular form is referred to as TTCN-GR. TTCN is also used in a machine processable form, called TTCN-MP.

A test suite specified in TTCN has four parts. These occur in the following order: (1) test suite overview, (2) declarations, (3) dynamic part, and (4) constraints part. These are all presented as sets of tables.

The test suite overview contains information which aids in the understanding of the test suite. This information includes a list of the standards on which the protocol and test suite are based, a reference to the PICS and a statement of the abstract test method(s) to

which the test suite applies. In addition, the suite overview also contains an index to the test suite and a list of test purposes, organized according to the structure of the test suite.

The declarations part describes the test events (i.e. ASPs and timer events) and the PCOs at which they can be controlled and observed. In addition, global constants and variables, PDUs and all parameters (of ASPs, PDUs and the test suite), along with any abbreviations used in the test suite are defined in this section.

The dynamic part contains the test cases and thus makes up the bulk of the test suite. The main part of each test case is a behaviour description, which is expressed mainly in terms of ASPs and timer events along with the PCOs at which they occur. The tree notation is used to present the behaviour description, which is really an enumeration of the possible sequences of observable events. A verdict of 'pass,' 'fail' or 'inconclusive' is assigned to each complete path (set of events) of the test case.

The constraints part specifies the values and coding of the parameters of ASPs and PDUs used in the dynamic part. An example of each of the four parts of a test suite is given in appendix 2.

TTCN is currently undergoing minor changes at the DIS (Draft International Standard) level, and it is likely that it will become a widespread standard for the specification of protocol test suites. There is substantial industrial and academic involvement in the development of TTCN, and test suite designers are encouraged to write their tests using this notation.

Conformance testing is a vitally important part of protocol development and implementation. It is a complex undertaking, requiring considerable effort and interaction on the part of those involved in the various aspects and phases of the conformance testing process. The derivation of effective, discriminating tests is certainly one of the more difficult tasks in conformance testing, and is of primary interest to us in this thesis. We shall review some of the more common methods of deriving such tests, and give some idea of their relative effectiveness in the following chapter.

Chapter 3

3. Test Coverage Criteria And Coverage Measurement

Given a protocol specification, it is a difficult task to derive effective test cases and build a good conformance test suite. Despite the difficulty, there are still many techniques that are potentially useful for deriving test cases [SiLe86]. No matter what method is used to derive test cases however, there is one important question that the test suite designer will want to have answered, *viz*, does his test suite contain enough 'good' test cases to allow him (and others) to have confidence in any protocol implementation which is tested using his test suite (and if so, can this confidence be quantified)? This is by no means an easy question to answer [HeKr87]. In order that this question can be answered however, we first need to establish a goal that must be achieved by the test cases of the test suite. We call this goal a *test coverage criterion*. Every test case should be based on some coverage criterion. A test suite may be designed around a single test coverage criterion, but generally it is designed around several.

In relation to the entire test suite, we make the following definition:

Definition 3.1: A test suite is said to *cover* a protocol with respect to a coverage criterion if, for that particular protocol, the test suite satisfies the criterion.

In most cases, the criterion is evident from the context, and we will simply say "the test suite covers the protocol." Test design strategies can be based on test coverage criteria and the representational model of the protocol. For example, if the protocol is specified using an FSM model, a test case derivation strategy may be based on firing the transitions

of the FSM in some particular order. In this light, we will now briefly review some test derivation methods used in software testing and their application to protocol conformance testing. In addition, we will present and discuss various theoretical protocol coverage metrics, particularly with a view of their applicability to real protocols.

3.1 OVERVIEW OF TESTING TECHNIQUES

Communications protocols are generally implemented via software. As such, general software testing techniques can be adapted to protocol testing. As discussed in [Myer79, Pro82a], there are three basic approaches to testing, *viz*, *black box*, *white box* and *grey box* testing. Practically all testing strategies fit into one of these areas, and we will discuss each separately.

3.1.1 White Box Testing

White box testing involves the careful study of the program code in order to determine the flow of control through the program. A two-step process is then used to design test cases. First, distinct paths (as determined by the flow of control) through the program are determined, and then an effort is made to find data which will drive the program along each such path. For this reason, it is sometimes called logic driven or structured testing. This is a very difficult task, as many paths turn out to be infeasible, that is, no data exist which can ever cause the program execution to proceed along such a path. In fact, Woodward *et al* [WoHe80] have found that beyond a certain length (in terms of LCSAJs), the total number of paths which can be identified in a program is one or two orders of magnitude greater than the number of feasible paths.

3.1.2 Black Box Testing

When the actual program code of the implementation under test (IUT) is not known, black box testing (also called functional testing) is used to design the test cases. A detailed specification, relating the observable behaviour of the IUT to input stimuli is needed. In deriving test data, the test designer does not take into account the internal structure, but rather the input/output relationships as stated in the specification. Techniques of black box testing can have application to protocol conformance testing, since the protocol implementation is treated as a black box.

3.1.3 Grey Box and FSM Based Testing

In order to get the most effective test cases, Myers [Myer79] suggests that both preceding methods be used. Some techniques have been developed [Sabo87, Ural87] which take advantage of both these methods. We call these types of techniques grey box [Pro82a] testing techniques, since both the specification and code-based testing techniques are used to generate the test cases. Grey box techniques are especially applicable to FSM models, since there are well defined paths through the graph, and white box methods can be adapted to produce test data to cover those paths.

3.2 COVERAGE CRITERIA

Test coverage criteria fall into two broad classes. The first is based on trying to cover paths through either the program code or the specification of the IUT. The second is based on trying to write test cases to expose certain specific errors which might exist in the implementation. A protocol conformance test suite (CTS) will generally have test cases which are derived using techniques based on criteria from each class.

3.2.1 Code and Model Coverage

Test cases based on criteria in this class aim at causing certain paths within the program or model representation to be executed or traversed, or to cover the input domain as given in the specification. Several criteria and related test derivation techniques of this class are discussed in the following subsections.

Criteria used in White Box Testing

The criteria in this section are all based on an intimate knowledge of the program code. The general objective is to cover certain paths (or subpaths) through the program. All of the following methods are just variations on this theme.

Total Path Coverage:

Total path coverage or *exhaustive path coverage* has the goal of executing every path through the program, as defined by the flow of control. This is an ideal, but not very practical metric. In all but very trivial programs, the number of control paths may be prohibitively large, making such testing impractical. In addition, as the size of the program grows, the number of infeasible paths also tends to increase. In fact, total path coverage is generally considered [Myer79, How80, LiFr87] to be not only impractical, but also infeasible.

Branch Coverage:

A minimal coverage requirement is branch coverage [Myer79, How80, PrMy87]. Here the requirement is that the test cases cause every branch in the program to be traversed at least once. This means that every decision node must have a *true* and *false* outcome at least once. This criterion assumes that the program source code is available, and since in protocol conformance testing this is usually not the case, this technique has practically no application in such testing. However, it may be possible to extend the concept to graphical representations of the protocol (as in transition coverage, discussed later).

Multiple Condition Coverage:

Branch coverage, as described above, considers only two-way branches. In addition, no consideration is given to the individual condition elements within a decision node. Even further, the combination of different truth values of separate conditions ought to be considered. Multiple condition coverage [Myer79] takes all these considerations into account. It requires that every possible combination of condition outcome in each decision node be tested at least once. Multiple condition coverage is a much more stringent criterion than branch coverage, but it requires correspondingly greater effort to find the test data. Like other white box techniques, this one has almost no direct application in protocol conformance testing.

<u>*Criteria used in Black Box Testing*</u>

In black box testing, the test cases are derived using only the specification of the IUT. That is, the relationship between the input and the observable output is used to generate the test data. In fact, any information which is derived solely from the specification may be used to help generate the test data. We will now briefly discuss some criteria for black box testing.

Exhaustive Input Testing:

Exhaustive input testing is an ideal approach in which the tester would write a test case for every possible input situation. Theoretically, all errors in a program could be found by generating test cases for all possible inputs [Myer79]. In all but very trivial programs however, this is impractical, as the following example shows.

Consider a simple program that has an input domain of [1, 2, 3]. Then the tester need only write three test cases for valid input. However, if the input domain were [1..3], the

tester is now faced with an infinite input domain, implying an infinite number of test cases for valid behaviour. In addition, there is an infinite number of invalid input tests. Thus exhaustive input testing is infeasible [LiFr87].

Selective Input/Output Testing:

A more realistic approach is to restrict the values chosen from the input domain to be used as test data. Myers [Myer79] describes several such techniques.

Equivalence partitioning is the technique of partitioning the input domain into a finite number of equivalence classes. A single test case is then developed for each equivalence class (both valid and invalid). In *boundary-value analysis*, values on the edges of the equivalence class are used in the test cases. In addition test cases are also generated by considering the output range. *Cause-effect graphing* forces the test designer to consider how combinations of different inputs affect each other and the output. This will hopefully lead to the production of more effective test cases.

Since in conformance testing the protocol implementation is treated as a black box, the above three methods may be used to complement other test derivation techniques where applicable. However, no substantial use of these methods has been reported in the protocol testing literature.

<u>Criteria Derived From Grey Box Testing</u>

The discussion in this section is based on the application of white box coverage criteria to models that are derived from the specification of the protocol. However, test data are still derived only from the observable input/output relationships given in the specification. In fact, many grey box coverage criteria have been proposed for protocol test design. We consider only FSM based models here. See [Meer86] for test design based on other models.

Transition Tour:

When the specification of the protocol is given as some form of an FSM, the branch coverage criterion of white box testing can be extended to this model. In this case, the criterion is to derive a minimal set of test cases to ensure that every transition in the FSM is fired at least once. This will produce a transition tour [SaBo84] of the FSM, and is generally considered to be a fairly weak and impractical criterion [LiFr87, HeKr87]. For example, transition tours may represent paths that may seldom or never arise in practice,

and therefore do not provide a reasonable test of normal protocol behaviour. Moreover, the test purpose of a transition tour test is not directly useful for detecting and correcting errors.

Distinguishing and Unique I/O Sequences:

Other techniques of generating test sequences for protocols specified as FSMs exist, and are considered to be better than the transition tour criterion. These criteria are based on the ability to produce test sequences which are able to identify states of the protocol uniquely.

A distinguishing sequence (DS) [Henn64] is a sequence of transitions which, when applied to an FSM supposedly in state m_i, can be used to determine not only if the FSM is in state m_i, but also which state it is in if it is not in state m_i. The input portion of the DS is the same for all states of the protocol, and not every FSM will have a DS. The unique input/output (UIO) sequence [SaDa85] can also be used to to identify a state m_i. However, the input portion of the UIO sequence for state m_i can only used to determine whether or not the protocol is in state m_i, (i.e. it can not tell which state the protocol is in if it is not in state m_i). The test coverage criterion based on these techniques is that each state of the protocol be visited. The state is then verified using either the DS or UIO sequence for that state. Optimization techniques have been applied in conjunction with the DS and UIO sequences in order to find the best order of touring the FSM graph. An example of this can be found in the application of the Chinese Postman algorithm to the FSM specification of the protocol in [AhDa88].

Data Flow Considerations:

Criteria based on the flow of data have also been recently proposed for protocols specified using the extended finite state machine (EFSM) formalism. The given specification is used to produce a digraph G which identifies the associations between definitions and usages of the variables employed in the specification [Ural87]. Each variable occurrence is classified as being a definition, computational use (c-use) or predicate use (p-use), and the associations between definitions and usages are represented in terms of dcu and dpu sets. This information is then used to derive test sequences to cover all definition and usage (du) pairs, with the restriction that the sequences represent complete executable paths through G.

Other work in this area has also been done by Sarikaya *et al* [SaBo87] who have developed a metric based on data flow as well as control flow coverage.

Other Criteria:

The concept of testing paths through the FSM rather than single transitions is an extension of path testing from white box techniques. By extension also, we know that total path coverage in large FSMs would be quite infeasible. The next chapter develops a criterion called *phase coverage* in which we try to break the protocol specification into functional units which are small enough to allow each path (with certain restrictions) in each unit to be tested. A similar technique is hinted at in [How80] in regards to program code and further developed in [How86].

3.2.2 Fault Coverage

Instead of writing test cases to cover the input domain of a program or branches or paths in the software, an alternative approach is to write test cases with the intention of detecting certain specific kinds of faults or errors [How86]. This technique may be considered as an extension of black box testing, and has led to some test derivation methods for protocol conformance testing.

Fault Models

A fault model is an error type, of which the implementation can have several instances [Lala85]. For example, if we consider an FSM, a fault model may be "the next-state function always gives the incorrect next-state." Then an implementation would have an instance of this error for each transition.

If the protocol is specified as an FSM, then there are three classes of implementation errors that can be recognized. These are (1) errors in the output function (operation errors), (2) errors in the next-state function (transfer errors), and (3) errors in the number of states. All operation errors can be detected by firing all transitions and observing the output. This turns out to be the same as a transition tour. Transfer errors and errors in the number of states can be detected by first transferring to a particular state and verifying that state, then firing a chosen transition and verifying the state reached. The W-method [Chow78], the DS-method [Gone70] and the UIO-method [SaDa85] mentioned earlier have been developed to facilitate the derivation of test cases based on these criteria. Extended finite state machines (EFSM) pose greater problems, because these can also have errors in a transition predicate.

Error Guessing

This technique, mentioned in [Myer79] is based mainly on the experience of the tester and a thorough knowledge of the specification. The technique requires the test designer to enumerate (by guessing) the kinds of errors that may be present in the implementation. The coverage criterion is then to have written enough test cases to find these errors if they are present. It is of course not likely that a test designer could guess all the errors in an implementation, but this technique may be useful in enhancing an already existing test suite.

3.3 COVERAGE MEASUREMENT TECHNIQUES

The techniques and coverage criteria that are used to derive test cases often determine the methods that are used to measure the coverage of the test suite. Different techniques exist for test suites based on program/model coverage and suites based on fault coverage. Of course, given enough information about the protocol specification, any measurement technique may be applied to any test suite, regardless of how it was derived. It may even be useful, in fact, to apply more than one coverage measure to a test suite, as this may lead to a more accurate assessment of its completeness.

3.3.1 Techniques for Measuring Code Coverage

When the source code is directly available to the tester, it is fairly easy to ascertain which statements, branches or paths in the program have been exercised by the test cases. The information can be obtained by instrumenting the program with probes [Pro82b, Huan78]. This procedure can be automated, making the generation of useful coverage statistics easier. Not only can these statistics show which statements and paths have been executed, but also how many times they have been executed in a given test run. The test designer can then use these statistics as a guide in deriving additional test cases to exercise statements which were not previously exercised.

In cases where only the specification is available, the above technique is clearly not useful. However, if the specification is in the form of an FSM (or other directed graph model), then perturbation analysis [Zafi80] or reachability analysis [BoSu80] will produce a reachability tree of the specification. As noted in [BuEc86], this tree will contain all allowable input/output interaction sequences and therefore each test case ought to be a path (or subpath) of this tree. The process of generating the reachability tree and of comparing

test cases in the test suite with paths in the tree ought to be automated, as it is certainly a tedious manual process. The problem of determining which paths in the tree ought to be treated as meaningful, desirable test cases, however, remains to be solved.

3.3.2 Techniques for Measuring Fault Coverage

Hengeveld and Kroon [HeKr87] describe a technique where, given certain fault models, the fault coverage of a test suite can be determined by performing simulations for every instance of error defined by each fault model. The number of simulations for a given fault model would be equal to the number of possible wrong implementations (say *N*) for that model. The normative fault coverage for that fault model would then be the number of times an error of that type is detected, divided by *N*. For any given implementation, the total number of simulations which would have to be done is prohibitively large, so statistical approaches are used instead.

For implementations based on the FSM model, machines with certain faults can be generated randomly [SaDa85, SiLe86], and then run against the test data. If no error is detected, the random machine is checked to see if it is equivalent to the original specification FSM. If it is not equivalent, then the test case is deemed to have failed.

When the program code of the IUT is available, a similar technique can be applied. Known as mutation testing [DeMi78], the procedure is to change a small section of the code and run the program against the test data. A test case will have failed if it were designed to detect the error in question, but does not.

3.4 SUMMARY

In this chapter we described several test coverage criteria. However, not all of these are applicable to protocol conformance testing. White box coverage criteria and associated test derivation techniques are generally not applicable because the program code of the IUT is normally not available. FSM based techniques however have wide application in protocol testing, and all practical black box strategies can be used to enhance the test suite in order to provide better coverage.

The coverage measurement techniques currently in use are either based on path analysis of the protocol graph or statistical simulations. To date, reports on in-depth studies to determine the completeness of test suites with regards to real protocols have not appeared in the literature.

Finally, it can be noted that even though this chapter surveyed many test coverage criteria and their measurement, and that these might appear to be rather formal, it is

currently thought that the general selection of any criterion for coverage measurement of a test suite is a highly subjective process [ShHo86, LiFr87].

In the next chapter, we present and illustrate our method. In chapter 5 we will evaluate our method *vis a vis* all the related methods and metrics discussed in this chapter.

Chapter 4

4. A Method For Measuring Test Coverage Based On Specified System Behaviour

As suggested in chapter 3, it is often difficult to determine when to stop testing [LiFr86, HeKr87], or put another way, to determine when there are enough test cases in a particular test suite. Sometimes, the decision to include one particular test case rather than some other test case in a test suite is based on the tester's intuition and knowledge [ShHo86]. In addition, due to time and cost constraints, most test suites are limited in size, and thus some test cases must be left out. It is important then, to be able to gauge whether a protocol test suite contains enough sufficiently discriminating and representative test cases to cover the protocol. We say that a test suite covers a particular protocol if for that protocol the test suite satisfies some predefined coverage requirement.

In order to obtain a reasonable solution to this problem, a coverage metric must be determined which is suitable for the representation of the protocol specification under consideration. Any proposed metric should strike a balance between cost (its complexity in terms of number of tests required to be performed) and the amount of confidence it affords. The coverage criterion proposed in this chapter is based on path coverage [Myer79]. It does not consider the program code of the protocol, but rather the paths between the nodes in a state-oriented representation of the specification. The finite state machine representation of Estelle [ISO1] facilitates this approach. The boundary-interior method proposed by Howden [Howd75] is also adapted and applied to loops on the paths. This method is discussed more fully in section 4.3.

This thesis does not give a method to detect when we have too many test cases in a test suite, but rather whether we have too few. It seeks to present a method to determine

whether certain required test cases are missing. It should be noted that the method presented in this chapter is intended to be applicable to an existing test suite, with no regards to whether it has been executed. Therefore, the method can also be used in the design phase to help ensure a more complete test suite. The method proposed in this chapter applies only to explicitly specified system behaviour. Therefore tests for unspecified behaviour and error recovery from unspecified inputs are not considered. Of course, these types of tests ought to be included in any test suite. This technique is intended to be applicable to protocols that are specified in Estelle. However, any state/transition based protocol specification can be used with this method.

In our discussion, an Estelle specification of a module will represent a protocol entity. When this entity is implemented, it is called an Implemented Protocol Entity (IPE). An IUT may contain several IPEs and the IUT is contained in the system under test (SUT).

4.1 DEVELOPMENT OF A REPRESENTATIONAL MODEL

As mentioned in chapter 3, test design strategies, coverage criteria and the representational model of the protocol are closely related. In later parts of this section, the concept of a *phase* is developed in relation to the determination of the coverage of a test suite. Before we proceed, we need a model to represent first, the meaningful collection of paths which comprise a phase and secondly, the behaviour which is characteristic of a phase. The model is based on the extended finite state machine (EFSM) formalism [ISO1]. One of the ways in which some characteristics of a phase can be shown is by the directedness of the paths in the graphical representation of the phase. In order to assist in the development of some of the properties of the phase, especially of the component transitions and paths, it is necessary to devise a method whereby the nodes of the phase can be numbered. The usefulness and application of this numbering will be shown after an algorithm to perform the numbering has been presented. The method used to accomplish the numbering will now be described.

4.1.1 The Modified Breadth First Numbering Scheme

Given an EFSM, consider two sets of nodes, S and T, representing a set of start and target nodes respectively. Further, consider a path ρ from any $m_s \in S$ to any $m_t \in T$, where ρ contains n nodes, $n \geq 1$. Denote the intermediate nodes by m_i, $2 \leq i \leq n-1$, $n > 2$, and let $m_s = m_1$ and $m_t = m_n$. The algorithm which follows is used to label the nodes of the EFSM graph so that the edges (transitions) of any such path described above can be characterized

with respect to the degree of progression on that path (as will be explained in the next section). The following notes are given as an aid in understanding the algorithm.

The number that is given to the node in this numbering scheme is called a **label number**. This is distinguished from the **level number** (distance from the start node) associated with the node due to the breadth first expansion. For example, start nodes will have level number zero. All nodes reachable by one transition from a start node will have level number 1, but will have different (unique) label numbers assigned by the algorithm. The label number at node m_i is represented by N(m_i).

The process of *expanding* a node *NE* is that of labelling all the nodes O_i with which the output transitions from *NE* are incident. Whenever a node is labeled, it is *marked* to show that it must be expanded, and it is assigned a level number that is one more than its parent. A marked node will become unmarked when all of its output nodes have been labeled.

Nodes of a lower level number are expanded before nodes of a higher level number (thus ensuring breadth first expansion).

In selecting the next node to be expanded, the one chosen is the lowest level marked node that has the greatest number of outputs to unlabeled nodes. If more than one such node exists, the node with the smallest label number is chosen.

A node which is being labeled is considered to have only one parent at that time. It is the node currently being expanded.

Cycles on paths in a *phase* (to be defined later) are of special interest to us. For our purpose we note:

Definition 4.1: A *cyclical path (or cycle)* is a path which starts at some node m_i and, after k transitions, $k \geq 1$, ends also at m_i. A *trivial loop* is a special case of a cyclical path which consists of only one transition.

$\rho 1 = \{(1,2), (2,3), (3,5), (5,7)\}$
$\rho 2 = \{(1,2), (2,4), (4,6), (6,7)\}$

Figure 4.1: MBF Node numbering and Paths from *S* to *T*

Once the nodes of a cycle have been labeled, the algorithm does not attempt to renumber it. This is because in one instance, nodes that may be part of a cycle with respect

to one path may, in another instance, be part of the acyclic path with respect to another path. To clarify this point, look at figure 4.1 and consider two paths from node 1 to node 7; $\rho1$={(1,2), (2,3), (3,5), (5,7)} and $\rho2$={(1,2), (2,4), (4,6), (6,7)}. It is easy to see that in path $\rho1$ while node 6 is in a cycle (e.g. {(3,6), (6,2), (2,3)}) attached to that path, in path $\rho2$, node 6 is actually in the acyclic path from node 1 to node 7.

The Numbering Algorithm

1. If there are p start nodes, then label them first, 1 to p inclusive, and mark them (for expansion). Assign level number of zero to all start nodes. If there are n nodes in the graph (module) and there are q target nodes, then reserve the numbers from n-q+1 to n inclusive for these nodes. The target nodes are not labeled until all other nodes in the module have been labeled.
2. While there are marked nodes

   Select a marked node (NE) to be expanded.

   2.1. For each output node O_i from NE

   2.1.1. If O_i is already labeled then do nothing.

   2.1.2. If O_i is a target node then do nothing.

   2.1.3. If O_i is not already labeled then

   2.1.3.1. Label O_i with the next available higher number.

   2.1.3.2. Mark O_i and assign it a level number which is one greater than that of NE.

   2.1.3.3. If O_i has outputs to labeled nodes X_i, which do not have outputs to labeled nodes, then exchange N(O_i) with the smallest of N(X_i).

   If node O_i now has a smaller label number than its parent (NE), then exchange N(O_i) with N(NE).

   2.1.3.4. If O_i has outputs to labeled nodes X_i and X_i has outputs to labeled nodes Y_i then

   2.1.3.4.1. For each output from O_i to a node X_i,

   If NE is not reachable from O_i and N(O_i) is bigger than any N(X_i) then

   2.1.3.4.1.1. Follow the path of labeled nodes from the above O_i, exchanging the label number at node m_i with the label number of its predecessor m_{i-1} whenever N(m_i) <

$N(m_{i-1})$. At a branch, choose the node with the smallest label number.

2.1.3.4.1.2. Stop when the last labeled node is reached or if $N(m_i)>N(m_{i-1})$.

2.1.3.4.1.3. Remember the above m_{i-1} and the path taken from O_i to m_{i-1}. Sort the node label numbers on the path from O_i to m_{i-1} so that O_i has the smallest number and m_{i-1} has the largest.

2.1.3.4.1.4. If $N(O_i)<N(NE)$ then exchange $N(O_i)$ with $N(NE)$.

2.2. Unmark *NE*.

3. Assign label numbers $n-q+1, n-q+2, ..., n$ to the target nodes in any order. ∎

Node Numbering Lemma

Lemma 4.1: The numbering of the nodes by the above algorithm ensures that on a single acyclic path from $m_s \in S$ to $m_t \in T$, a subpath from node m_i to $m_t \in T$ contains fewer edges than a subpath from node m_j to $m_t \in T$ if $N(m_i)>N(m_j)$, i.e. the label numbers are monotonic increasing along that path.

Proof:

1. By step 1 of the above algorithm, the start node will have the lowest node number of all nodes in any path.
2. By step 1 and step 3 of the above algorithm, the target node will have the highest node number of all nodes in any path.
3. Because node numbers are allocated in ascending order, a node X which is labeled before a node Y will be given a lower node number than node Y.
4. Because of the breadth first numbering scheme, nodes at a lower level are labeled before nodes at a higher level.
5. On any direct (acyclic) path from $m_s \in S$ to $m_t \in T$, if because of (3) and (4) above a subpath is found in which the nodes are labeled in non-ascending order, then if there are only two nodes in that subpath, step 2.1.3.3 of the algorithm will reverse the order of the label numbers of the nodes. If there are more than two nodes in that subpath, then steps 2.1.3.4.1.1 to 2.1.3.4.1.3 will sort the label numbers so that the nodes are labeled in ascending order, i.e. $N(m_{i-1})<N(m_i)$. ∎

Figure 4.1 shows a module whose nodes have been labeled using the MBF scheme, and two paths from $m_s \in S$ to $m_t \in T$; ρ1={(1,2), (2,3), (3,5), (5,7)} and ρ2={(1,2), (2,4), (4,6), (6,7)}. Other paths from m_s to m_t exist, of course. The MBF numbering of the nodes confirms what is claimed in lemma 4.1 concerning both paths from $m_s \in S$ to $m_t \in T$, that is, if two nodes m_i and m_j are on a path and have label numbers $N(m_i)$ and $N(m_j)$, where $N(m_i)<N(m_j)$, then a transition from m_i to m_j is guaranteed to take the IPE closer to the target node $m_t \in T$ along that path. The properties which are acquired by the transitions as a result of this numbering of the module are further discussed and exploited in the following sections.

For the remainder of this thesis, a start node m_s will be understood to be a member of set S, and a target node m_t will be understood to be a member of set T.

4.1.2 Basic Properties of Paths in the Module

We can now use the MBF numbering of the nodes of the EFSM to give some properties of the paths in the module. In the discussion which follows, in order to save space, we will use statements like "the IPE progresses from node m_i ..." to mean "the IPE progresses from the state represented by node m_i ..." Whenever nodes are mentioned in relation to IPEs, it should be understood that the nodes represent states.

Definition 4.2: A ***progressive transition with respect to the path from m_s to m_t*** is a transition such that when it is fired, the IPE goes from a node m_j to node m_k, $N(m_j)<N(m_k)$; $1 \le j,k \le n$ and the subpath from m_k to m_t does not contain any node which appears previously in this path.

Intuitively, a progressive transition will advance the IPE towards the target node. As an example, if we consider path ρ_1 of figure 4.1, then transition (2,3) in that path is a progressive transition.

In relating nodes, transitions and cycles to paths within a module, we will use the following terms.

A node is ***in*** the path from m_s to m_t if it is the initial or final node of any progressive transition which is a component of the path from m_s to m_t. A node is not in the path otherwise.

A **transition** is said to be ***in*** the path from m_s to m_t if it is a component of the path and it is progressive. A transition is said to be ***incident with*** the path from m_s to m_t if its initial node or final node (but not both) is in the path from m_s to m_t. A transition is also said to be incident with the path from m_s to m_t if (1) it makes up part of a cycle, (2) its initial node *and* final node are in the path and (3) it is not progressive.

A **cycle** is said to be ***attached to*** the path from m_s to m_t if any node of the cycle is in the path.

Definition 4.3: A ***regressive transition on the path from m_s to m_t*** is a transition from node m_i to m_j such that;

1. there exists a path of p transitions, $p \geq 0$, from m_j to some node m_k (i.e. m_j may or may not be the same node as m_k),
2. m_k is in the path from m_s to m_t and m_k appears before m_i, $m_i \neq m_k$.

Intuitively, a regressive transition will take the IPE away from the target node. If we look at path ρ_2 of figure 4.1, then we see that transition (6,2) incedent with that path is a regressive transition.

Definition 4.4: A ***non-progressive transition with respect to the path from m_s to m_t*** is a transition which forms part of a cycle attached to that path, and is not regressive.

Intuitively, with respect to the path from m_s to m_t, a non-progressive transition on this path will not cause the IPE to regress from the target node, nor will it help the IPE to advance towards the target node again once 'ground has been lost' to a regressive transition. With respect to path ρ_1 of figure 4.1, transition (2,4) is non-progressive. Note the distinctions among progressive, non-progressive and regressive transitions. The differences will become more clear when we examine the different types of cycles to be found attached to a path.

If we consider cycles attached to the path from m_s to m_t, we can identify two types, *viz*;

1). **Simple Cycles:** These are cycles which start and end at some node m_i and, ***with respect to the path under consideration***, it is possible to enter and exit the cycle only at

this node m_i, i.e. m_i is the only node of the cycle which is in the acyclic path from m_s to m_t.

2). **Complex Cycles:** These are cycles which start and end at some node m_i, which (*again with respect to the path under consideration*) must be entered at node m_i and exited at node m_j, $m_i \neq m_j$, and where m_i, m_j and all the nodes (possibly zero) between m_i and m_j which are part of the cycle are in the acyclic path from m_s to m_t.

Intuitively, simple cycles have only one node in the path from m_s to m_t, whereas complex cycles have two or more nodes in the path. Figure 4.2 shows the difference between these two types of cycles. If we look at path ρ_2 of figure 4.1 again, we see the complex cycle C_1={(2,4), (4,6), (6,2)} attached to that path.

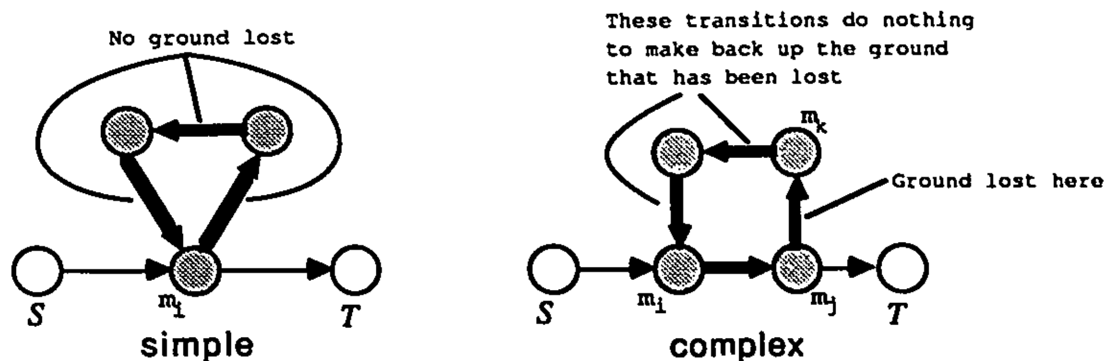


Figure 4.2: Simple and Complex cycles

Lemma 4.2: The first transition of a complex cycle leaving the path from m_s to m_t is always regressive.

Proof: The proof follows from definition 4.3 and the definition of a complex cycle.

Simple cycles are made up entirely of non-progressive transitions. Complex cycles are made up of a combination of progressive, regressive and non-progressive transitions. Suppose m_i is the node of the complex cycle that is nearest m_s on the path from m_s to m_t, and m_j is the node of the cycle that is nearest m_t, and there is a transition from m_j to m_k (m_k may be the same node as m_i) in the cycle (see figure 4.2). Then all the transitions which take the system from node m_i to m_j are progressive. The transition from node m_j to m_k is regressive, and if they exist, all the transitions from node m_k to node m_i are non-progressive. Along path ρ_2 of figure 4.1, complex cycle C_1 is entered at node 2 and exited

at node 6. Transitions (2,4) and (4,6) are progressive, while transition (6,2) is regressive. There are no non-progressive transitions in this cycle.

We can now define a module which incorporates all of the above properties. This module will represent the basic element of our model.

Definition 4.5: A submodule specification which has an identified set of start nodes S and set of target nodes T, and in which every path from $m_s \in S$ to $m_t \in T$ is composed only of progressive and non-progressive transitions is called a *directed behaviour module* (**DBM**).

4.1.3 Characteristics of DBMs

We will now give some properties of directed behaviour modules.

Lemma 4.3: A **DBM** cannot contain any complex cycles.

Proof: By definition 4.5, there are only progressive and non-progressive transitions in a **DBM**. However, by lemma 4.2, a complex cycle always contains a regressive transition. Hence there can be no complex cycles in the **DBM**. ∎

Theorem 4.1: Inside any primary path from m_s to m_t in a **DBM**, each fired transition either (1) advances the IPE towards the target node or (2) takes the IPE no nearer to or no further from the target node.

Proof: From definition 4.5, a **DBM** consists only of progressive and non-progressive transitions. By definition 4.2 and lemma 4.1, whenever a progressive transition is fired, the number of transitions which remain to be fired before the target node is reached is reduced (by at least one). By lemma 4.3, there can be no complex cycles in a **DBM**, therefore, the non-progressive transitions in the **DBM** must be, by definition 4.4, components of simple cycles. But simple cycles have only one node in the path from m_s to m_t. Thus if a cycle is entered at say node m_i, then node m_i will be the first node in the primary path to which the IPE will return, and the result is the same as if a trivial loop had been executed at node m_i. ∎

There is a purpose associated with each **DBM**. It is derived from the syntactic properties of each path in the **DBM**; being directed from the start node to the target node.

The purpose is to reach the target node, starting from the start node. Thus with each specification for a **DBM**, there is this naturally associated semantic information, i.e. the intention of reaching the target node. A question for future consideration is whether this semantic information can be reflected in the test cases derived from each **DBM**.

Once a node in a path through a **DBM** has been visited, it is not visited again except in the case where that node is part of a simple cycle. However our study has revealed that many types of errors are associated with transitions which would be considered to be regressive if included in a **DBM**, and in fact, there may be some states in the specification that are visited only when an error or some other 'abnormal' condition (e.g. timeout) occurs. All states which are linked by related transitions (i.e. transitions associated with the same type of error) of the above type are combined into a single module. Thus for a given protocol, many of these modules may exist, each module representing some specific kind of error activity. Figure 4.3 shows how this combination would be done. We call the resulting module an error/recovery module (**ERM**). Unlike a **DBM**, an **ERM** has no designated start or target nodes. Therefore the concept of progressive, non-progressive and regressive transitions does not apply to **ERMs**. In fact, any node in **ERM** that has an outgoing transition shows that error behaviour associated with that transition can start at that node. Thus every transition represents some specified error activity. This implies a strategy of single transition at a time testing rather than specific path testing as implied by the **DBM** representation.

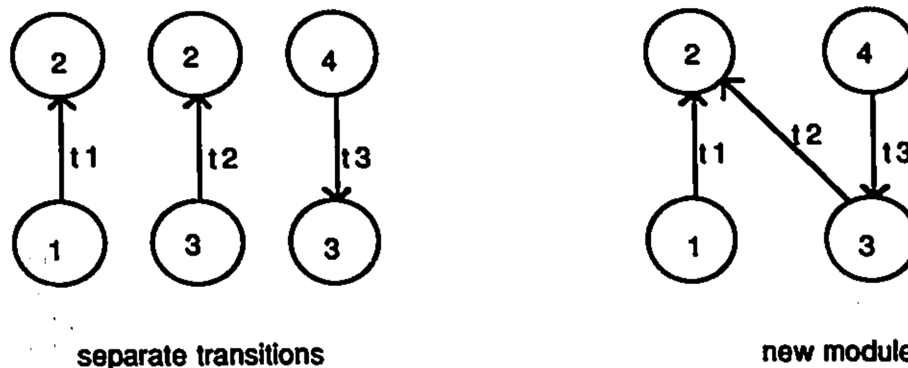


Figure 4.3: Grouping error related transitions into a module

The **DBMs** will be used to represent *normal behaviour phases* of the protocol (a concept which will be developed in the following section), while the **ERMs** represent the error and recovery phases. The protocol will enter the error/recovery phase from any other phase whenever it receives some erroneous input.

4.1.4 Identification of Phases

A careful study of the protocol specification will allow us to decompose the protocol's behaviour into mechanisms for achieving a set of goals. The identification of these goals will be based on the knowledge and intuition of the tester. As a basis for this section, we consider the protocol to be composed of entities, each of which has separate functions to perform. Alternatively, we can consider the protocol itself as one entity which has numerous subfunctions to perform. Some of these subfunctions may be related. For example, the sending of a data frame and the reception of a corresponding acknowledgement are related subfunctions. With careful study, we can determine whether related sub-functions can be combined into a single function. The performance of a (sub)function leads to the achievement of a subgoal. The discussion that follows next will be based on these concepts.

Let ρ_1 and ρ_2 be two subpaths in the protocol specification and suppose that there exists some mechanism for determining the start and target nodes of each path. We define a *phase grouping* **R**, to be binary relation on paths in the protocol specification as follows:

> **Definition 4.6:** $\rho_1 \mathbf{R} \rho_2$ iff for some goal G, the actions performed when the transitions of each path are executed are intended to and actually achieve the intended goal G.

For our purposes, we will define an *action* to be represented by the tuple $(m_i, m_j, L_{ij}, (v_1, ..., v_n))$ where m_i and m_j are two nodes in the specification, L_{ij} is the label on edge (m_i, m_j) and $(v_1, ..., v_n)$ represents the values of the context variables $v_1, ..., v_n$ after the transition from m_i to m_j. In this way, an action can be associated with a single transition.

Lemma 4.4: Relation **R** is an equivalence relation.

Proof: Let ρ_1 be a path in the protocol specification. Then $\rho_1 \mathbf{R} \rho_1$ since there is only one set of actions being considered, and one set of actions will achieve one goal (*Reflexive*).

Suppose we have two paths, ρ_1 and ρ_2, and let $\rho_1 \mathbf{R} \rho_2$. Since the actions performed by ρ_1 achieves the same goal as the actions achieved by ρ_2, then obviously the actions

performed by ρ_2 will achieve the same goal as those performed by ρ_1 and hence $\rho_2 \mathbf{R} \rho_1$ (*Symmetric*).

Let ρ_1, ρ_2 and ρ_3 be three paths in the protocol specification. Then if the actions performed by ρ_1 achieves the same goal as the actions performed by ρ_2, and the actions performed by ρ_2 achieves the same goal as the actions performed by ρ_3, then it follows the actions performed by ρ_1 achieves the same goal as the actions performed by ρ_3. Hence we have $\rho_1 \mathbf{R} \rho_2 \wedge \rho_2 \mathbf{R} \rho_3 \Rightarrow \rho_1 \mathbf{R} \rho_3$ (*Transitive*). ∎

Since **R** is an equivalence relation on the paths in the protocol specification, it will partition this set of paths into a number of equivalence classes.

Theorem 4.2: Relation **R** partitions the paths of the specification into a single class which contains (possibly) more than one path and a number of classes which contain only one path.

Proof: For any particular goal *G*, the actions performed by the execution of any path either achieves the goal, or they do not. Thus we have two general classes of paths. For any given goal, this will always be true. If now the above equivalence relation is defined on the paths, those paths which achieve the goal will be grouped into a single equivalence class. If there is only one path in the specification which achieves the goal, then this will be the only path in this class. Any path which does not achieve the goal will not be related to any other path, and hence will be placed into a class by itself. ∎

For each goal *G*, there will be one equivalence class which will be of interest to us, *viz*, the one in which the actions associated with the paths achieve the goal *G*. It will be the only class which contains (possibly) more than one path. If we consider a number of different sub-goals, then each application of **R** will produce one 'interesting' equivalence class.

We can see then, that for each set of actions relating to a specific sub-goal, there is an associated set of paths within the protocol specification. We call this set, i.e. each 'interesting' equivalence class, a *phase*. The sub-goal achieved by executing any path in a phase is considered to be the goal of the phase.

For each phase, it must be determined what state that phase can start in. Some phases can start in more than one state. The end state of the phase will be any state that the protocol entity will be in when it has performed a particular set of actions and has achieved the goal

of the phase. We also call this the *goal state* of the phase. The goal state of one phase often represents the start state of a logically successive phase. However, this does not hold for all phases. For each phase, we must determine separately which states it can start in.

In order to determine these start and goal states, the goal of each phase must be clearly recognized. The determination of goals (and subsequently the start states and goal states) of phases can be considered a behavioural decomposition of the protocol. As observed in [LiFr86], this decomposition is quite subjective. Whereas the determination of the start states and goal states of the corresponding phases may not be as subjective as the determination of the goals themselves, the process is still rather informal. In general, an examination of the protocol specification (after having determined the goal of the phase) will allow us to select these states. We are now in a position to give a definition of a phase of a protocol.

> **Definition 4.7:** A *phase* of a protocol is an equivalence class of paths which begin at some identified start state(s) and end at some identified goal state(s), and which represent the performance of a set of actions specifically intended to achieve a particular goal.

Two broad classes of phases have been identified in this research. These are normal behaviour phases and error/recovery phases. The normal behaviour phases are those phases of the protocol that represent valid behaviour sequences and are derived from the valid interaction sequences as given in the specification. The error/recovery phases represent those interactions which cause or are caused by some error condition as specified in the specification, and timer expiry. These interactions may include the actions taken to bring the system to a stable state after an error or timeout has occurred.

There is an ordering associated with the phases of the protocol (see for example figure A1.1 showing the phase decomposition of LAPD). It is clear that some phases must be performed before other phases (e.g. **Link Establishment** phase must be performed before **Data Transfer** phase). Thus we can talk about a 'previous' or 'next' phase. A particular phase of the protocol is identified as the 'starting phase.' This phase will have the idle state of the protocol as its start state, and the protocol must always go through this phase when it is first started up or reset.

4.1.5 Construction and Representation of Phases

Having identified all start states and goal states, we simply find every path from each start state to each goal state that is made up only of progressive and non-progressive transitions. Not all these paths may be relevant to the phase under consideration, and the paths should be examined to eliminate irrelevant ones. At transitions that can be fired by more than one input, care must be taken to select the input that is meaningful for the phase under consideration. This clearly requires a good understanding of the protocol. These paths are grouped together into a module. When all the paths are grouped together, there ought to be no complex cycles (which contain regressive transitions) in the module. Paths can be selected in this way since there is a goal associated with each phase and actions that are performed during a phase are intended to help achieve that goal (and hence take the IPE to the goal state). Regressive transitions correspond to actions which are in fact part of some other phase, since these actions take the system away from the goal state, and do not help achieve the goal.

Sometimes however, it is not obvious which states a phase will start or end in just by examining the protocol specification. In this case, the actions that are considered necessary in order for the goal to be achieved are traced through the protocol. The transitions which would be fired by this process are linked into paths. The results obtained by using this method indicate that the paths will tend to terminate at very few states (one or two in most cases), but sometimes, the starting points may not be localized at a few states. This method was used for example to determine the **Deactivation** phases of LAPD.

It is better to use a combination of these two techniques to help ensure that some paths which ought to be in a particular phase are not missed. Where possible, the first method should be applied before the second method, the second method serving only to find any paths which might have been overlooked by the first method.

A phase may be determined in such a way that it has more than one start or goal state. Except in the case where the goal state is also the start state, there will be no transitions from the goal state to any other state within the phase. However, a start state may have transitions entering it from other states within the same phase.

The concept of a phase does not include the ability to perform some action. For example, a given protocol in some phase P_1 may be in a state m_i where it is possible to perform some action of phase P_2, but this does not mean that the protocol is in phase P_2. We say that the protocol enters phase P_2 only when it starts executing a set of actions specific to phase P_2.

Theorem 4.3: The normal behaviour phases of the protocol can be represented by a **DBM**.

Proof: There is a 1–1 correspondence between the goal associated with a phase and the goal of reaching the target node of the **DBM**. They are both achieved by following a directed path from the start state/start node to the goal state/target node of the phase/**DBM** respectively. Thus the start states of the phase can be represented by the start nodes of the **DBM** and the goal states can be represented by the target nodes.

The necessity for the actions of the phase to advance the IPE towards the goal state (to allow the goal to be achieved) allows only progressive and non-progressive transitions to be components of phases. By theorem 4.1, the transitions of a **DBM** either advance the IPE towards the target node or take it no further from the target node, thus fulfilling the requirements for only progressive and non-progressive transitions in phases. ∎

4.1.6 Joining Phases Together

Attaching the phases together is a relatively simple matter. Because these phases were derived from an original (total) protocol specification, the following method will always yield correct submodules (according to the original protocol specification).

The phase diagram showing the ordering of the execution of the phases may be useful in this exercise, but is not necessary. It will however show at a glance which phases can be connected together. To attach phase P_1 to phase P_2, all the states which they have in common must first be identified. Two phases cannot be joined to each other unless they have at least one state in common. If P_1 and P_2 have states s_1 to s_n in common, then the procedure for attachment is as follows:

1). All the transitions which enter s_1 in P_1 are made to enter s_1 in P_2.
2). All the transitions which leave s_1 in P_1 are made to leave s_1 in P_2.
3). Eliminate s_1 from P_1.
4). Repeat steps 1) to 3) for states s_2 to s_n.

Since we are not creating any new paths, but rather rebuilding old ones, the resulting module will be correct. It can be noted however, that some paths which were separately feasible in the two phases may form one infeasible path when the phases are joined.

Similarly, two paths which were separately infeasible may become feasible. This will be discussed further in a following section.

The above procedure can be applied equally well to joining the modules which result when two phases are joined together.

4.1.7 Relationship Between Protocol Specification and Set of DBMs and ERMs

We will now summarize what has been said up to this point in relation to phases of the protocol and the protocol specification as a whole.

Let $\cup$ represent the joining or union of modules as described above, and $\cap$ be the set intersection with reference to the transitions comprising a module M. A module M represents a phase of the protocol, and is either represented by a **DBM** or **ERM**. Let PS represent a protocol specification and *Trans(PS)* be the set of transitions which compose the specification.

With respect to transitions, DBMs and ERMs are mutually disjoint. Formally;

Theorem 4.4: If $t \in M \in \{\mathbf{DBM}\}$ then

$$M \cap \{ \{ \cup_i \mathbf{DBM}_i - M \} \cup \{ \cup_j \mathbf{ERM}_j \} \} = \varnothing$$

and if $t \in M \in \{\mathbf{ERM}\}$ then

$$M \cap \{ \{ \cup_i \mathbf{DBM}_i \} \cup \{ \cup_j \mathbf{ERM}_j - M \} \} = \varnothing$$

i.e. t is in no other module but M.

Proof: From definitions 4.6 and 4.7 and from theorem 4.2, we know that a phase is an equivalence class of paths, and equivalence classes are disjoint. Furthermore, phases are made up of specific actions. The actions of one particular phase are not found in any other phase, and for any given action, there is a single corresponding transaction. ∎

Theorem 4.5: Every transition of the original specification is in some module M, i.e.

$$\forall t \in Trans(PS)\ \exists\ M \in \{ \cup_i \mathbf{DBM}_i \} \cup \{ \cup_j \mathbf{ERM}_j \} \text{ such that } t \in M.$$

Proof: Consider how phases are determined. We start with a complete specification and from it derive a number of disjoint phases. Therefore, the union of all the phases represents the original specification, i.e,

$$(\cup_i\{\mathbf{DBM}_i\})\cup(\cup_j\{\mathbf{ERM}_j\}) \equiv PS. \quad (1)$$

Now a phase represents a specific set of actions, therefore the union of all the phases represents the totality of all the actions of the protocol. But for every action, there is a corresponding transition t. Therefore, identity (1) above represents the totality of all transitions in the protocol specification. Hence the result. ∎

In the foregoing sections, we developed the mechanisms and model that allow us to decompose any state/transition oriented protocol specification into a number of disjoint phases. We will now proceed to take a closer look at the paths in these phases, as this will be necessary in order to allow this decomposition to be useful in the conformance testing process.

4.2 CLASSIFICATION OF PATHS IN A DBM

In any given specification of a protocol, not all the paths are necessarily executable. This holds true even at the level of phases and their **DBM** representations. Since it is desirable that the test cases (actually paths through phases—and hence paths through the protocol) in a test suite be executable, it will be useful to have some idea of the feasibility of these test cases (paths) before execution of the test suite is attempted. In this section, we attempt to classify the paths in a given **DBM** with respect to feasibility. First we present the main concepts and definitions necessary to characterize feasibility.

4.2.1 Feasibility Model: Terminology

Predicates Associated with Transitions

If we consider a transition between two nodes of a **DBM**, then it may be that the transition should be fired only if some condition is met. This case is normally shown by the conjunction of that condition with the input (if any) at that transition. The condition in this case is called an ***enabling predicate*** and the transition is fired only if the predicate is true. Figure 4.4 shows a predicate ($x<5$) at transition t_1. In this case, transition t_1 can be fired only if $x<5$.

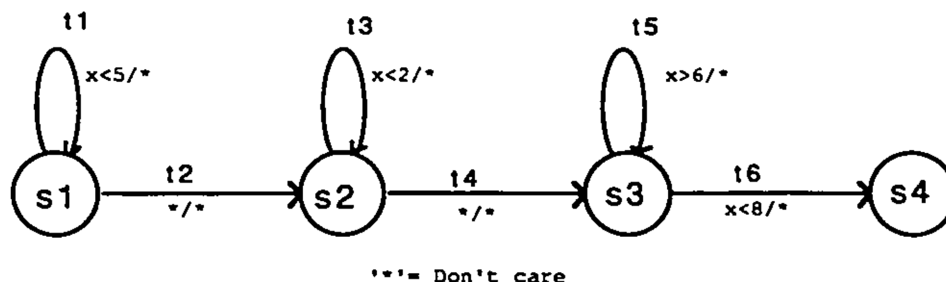


Figure 4.4: Predicates and Transitions

Infeasibility

Consider a protocol specification. A transition which starts at node m_i and terminates at node m_j represents some action which is possible at node m_i, depending perhaps on the truth value of some predicate associated with that transition. If there is no predicate associated with that transition, then that transition is always firable. This is always true, except in the case of a nonexecutable transition, i.e. the transition in question can never be fired. Assuming that there are no transitions in that path which can never be fired, then the only thing that can cause a given path to be non-executable is the falseness of a predicate in the path. We can now therefore state what it means for a path to be feasible.

Definition 4.8: A path ρ from m_s to m_t is *feasible* if it is possible for the IPE to progress from node m_s to node m_t by firing only those transitions which compose the given path ρ.

For a given **DBM** then, we can determine a priori three sets (categories) of paths;

1). A set of feasible paths.

2). A set of infeasible paths.

3). A set of possibly infeasible paths. Paths in this set could turn out to be infeasible on the expanded protocol level, but are feasible on the isolated phase (**DBM**) level.

For a given specification, any (but not all) of these sets may be empty, of course.

We now give some definitions which will be useful in the following sections;

Definition 4.9: A variable of a predicate is said to be *input_dependent* if its value can be changed by the input (e.g. different parameter value) associated with some transition.

Definition 4.10: A variable of a predicate is said to be *action_dependent* if its value can be changed only as a result of the actions associated with some transition.

There is a class of variables which is neither input_dependent nor action_dependent, but whose value depends upon the state of the system under test. We call these *system-state_dependent* variables. An example of such a variable is the Able_to_Establish variable used in LAPD. System-state_dependent variables can change value as a result of actions which occur outside of the IPE under consideration and its peer, and the test designer does not have direct control over the values of these variables. The occurrence of such a variable in a predicate at a transition therefore introduces an element of uncertainty on the firability of the transition, and causes the path to be possibly infeasible.

Definition 4.11: A predicate is said to be input_dependent (action_dependent, system-state_dependent) if it is composed only of input_dependent (action_dependent, system-state_dependent, respectively) variables. A predicate is said to be *mixed* if it is composed of any combination of input_dependent, action_dependent and system-state_dependent variables.

We note therefore that input_dependent predicates can always be made true by the appropriate (combination of) input values and hence are relatively mutually independent. Thus a path which contains only input_dependent predicates can (theoretically) be made feasible.

On the other hand, action_dependent and mixed predicates will only be true when their component action_dependent variables have been explicitly set to the appropriate values. In addition, suppose we consider two transitions t_1 and t_2 in some given path ρ and suppose that there are predicates p_1 and p_2 which have certain action_dependent variables in common associated with t_1 and t_2 respectively. Now if the values of the action_dependent variables in common between p_1 and p_2 are not changed between t_1 and t_2, then the truth of p_2 will be related to the truth of p_1. In figure 4.4 for example, if x is an action_dependent variable, and we consider the subpath $\{t_1, t_2, t_4, t_5\}$, then if t_1 is firable, then t_5 is not, since the value of x is not changed between transitions t_1 and t_5. Thus action_dependent and mixed predicates in a path may not generally be mutually independent. This observation leads to the development of what we might call *disallowed*

domain sets for action_dependent variables. We will further develop this notion in the following subsection.

Disallowed Domain Sets

A disallowed domain set for variable x, represented by DD_x, will be all the values which x can not have. Note that DD_x has meaning only when associated with a transition within a particular given path. We will now show basically the three actions which can be performed on the disallowed domain set for a particular variable. These are *creation, update* and *deletion.*

If an action_dependent variable is used in some predicate associated with a given transition and that variable has neither been defined (i.e. explicitly set to a particular value) nor used at any point in the path prior to the transition under consideration, then a disallowed domain set is *created* for that variable. The disallowed domain set will contain all the values which are outside the range of values required by the variable in the predicate. As an example, consider figure 4.4 again. The predicate at transition t_1 contains the first occurrence of the action_dependent variable x. Thus we create the disallowed domain set for x at this point, given that the predicate is '$x<5$.' We then have that

$$DD_x = \{x \mid x \geq 5\}.$$

If an action_dependent variable is used in some predicate before it is explicitly defined, but already has a disallowed domain set associated with it, then it may be necessary to *update* the disallowed domain set for that variable. The set is only updated if the new range of values required by the variable is a superset of the already existing disallowed domain set. Thus for example in figure 4.4, if we consider the subpath $\{t_1, t_2, t_3\}$ where the predicate at transition t_3 is '$x<2$,' and the new set of values required for this is a superset of the already existing set, we must change that set to

$$DD_x = \{x \mid x \geq 2\}.$$

On the other hand, if we consider the path $\{t_1, t_2, t_4, t_6\}$ where the predicate at transition t_6 is '$x<8$,' this would require a disallowed domain set of $\{x \mid x \geq 8\}$ which is not a superset of the one already existing. Therefore , the DD_x set is not updated at transition t_6.

A disallowed domain set for a particular variable is *deleted* whenever one exists for that variable and at some transition in a path under consideration, the variable is defined, i.e. it is explicitly set to a particular value.

4.2.2 Algorithm to Classify Paths in a DBM

Following is the development of an algorithm that can be used to categorize the *complete* paths in a given **DBM**, a priori. A path ρ in a **DBM** is said to be ***complete*** if it begins at the start node $m_s \in S$ of the given **DBM** and finishes at the target node $m_t \in T$ of the same **DBM**. In some cases, the algorithm can conclude without examining the entire path, that the path is infeasible. In other cases, e.g. if the path is feasible, the algorithm must examine the entire path. It is assumed that input_dependent variables can always be set to the values needed to make an associated predicate evaluate true.

Overview of Algorithm

Algorithm CLASSIFY

1). Given a path ρ, begin at the start node m_s.

2). For each transition in the path, determine if that transition is firable:

a). **If** the transition is not firable,
then declare the path to be infeasible.

b). **If** the transition is firable,
then go on to the next node.

3). Repeat step 2 until the target node is reached or until the path is found to be infeasible.∎

At the termination of the algorithm, the given path will be placed in one of the categories given earlier, i.e. feasible, infeasible, or possibly infeasible.

Detailed Algorithm

The following is an expansion of the above algorithm. Note that when considering a transition from any node m_i in the path, it is assumed that all the transitions in the subpath from m_s to m_i have already been found to be firable or possibly firable.

Algorithm CLASSIFY

Given a path ρ from $m_s \in S$ to $m_t \in T$

1). At the start node m_s, set (or assume) all predicates in the path ρ true. Intuitively, this means that the variables associated with these predicates can be assumed to have the appropriate values.

2). Try to trace the execution of the path ρ from the start node m_s to the target node m_t, by considering the transition from each node m_i to the successive node m_j.

3). If there are no predicates associated with any transition in the subpath from m_s to m_i,
then the subpath is feasible, **else**

4). If there are *only* input_dependent predicates in the subpath from m_s to m_i, and there is a predicate at the transition from m_{i-1} to m_i,
then that subpath (m_s to m_i) can be considered to be feasible when the predicate under consideration is made true by the input (parameter) values.

5). If there is a system_state_dependent predicate at the transition from m_{i-1} to m_i,
then the subpath from m_s to m_i will be possibly infeasible. For the remainder of the path m_i to m_t, if no predicate evaluates false, then the whole path will be characterized as being possibly infeasible.

6). If there are action_dependent or mixed predicates in the path,
then in tracing the execution of the path from m_s to m_t, we will keep track of the values of the action_dependent variables associated with these predicates. Disallowed domain sets must be created and updated for those action_dependent variables whose values are not explicitly set before the variables are used. In considering the transition from node m_{i-1} to m_i:

a). If the values of the variables associated with an action_dependent predicate at that transition are explicitly set such that the predicate evaluates true,
then the subpath from m_s to m_i is feasible.

b). If there are no system_state_dependent variables in the predicate, and the values of the action_dependent variables associated with a mixed predicate at the transition are explicitly set such that the predicate evaluates true when the input_dependent variables associated with that predicate have appropriate values (remember that input_dependent variables can always be set to a particular value),
then the subpath from m_s to m_i is feasible.

c). If the values of any action_dependent variables associated with an action_dependent or mixed predicate at the transition are explicitly set such that the predicate evaluates false,
then the path from m_s to m_t is infeasible.

d). If the values of the action_dependent variables associated with an action_dependent or mixed predicate are not explicitly set at any point in the

path before the transition under consideration such that the predicate under consideration evaluates to false,

then consider the disallowed domain sets for the variables concerned:

If any action_dependent variable of the predicate must have a value which is in the disallowed domain for that variable in order for the predicate to be true,

then the subpath from m_s to m_i is infeasible (and hence the path from m_s to m_t),

otherwise the subpath from m_s to m_i is considered as being possibly infeasible. This is so because one or more of the action_dependent variables might have been assigned some value outside the current module. The variables may then still have these values when the path under consideration is being executed, and these values may cause the predicate in question to evaluate false. For the remainder of the path from m_i to m_t, if no action_dependent or mixed predicate in the subpath evaluates false, then the result of this substep is applied to the complete path from m_s to m_t, i.e. the path is categorized as being possibly infeasible.

e). **If** there is a system-state_dependent variable in a predicate at the transition from m_{i-1} to m_i,

then the subpath from m_s to m_i will be possibly infeasible if no other other condition in the predicate causes it to evaluate false. For the remainder of the path from m_i to m_t, if no predicate evaluates false, then the whole path will be categorized as being possibly infeasible.

7). Repeat steps 3 to 6 until target node m_t is reached or until the path is declared infeasible.▮

In summary, the algorithm will place the given path ρ in one three categories. If any predicate in the path is determined to be false, then the complete path is considered to be infeasible. If any action_dependent variable causes a predicate value to be indeterminable, as in step 6d, then the subpath is categorized as being possibly infeasible. In all other cases, the subpath is considered to be feasible.

4.2.3 An Application of the Algorithm

The following two tables show the expected results of applying the above algorithm to a pair of complete paths in a **DBM**. Those paths, shown in figure 4.5, are taken from

the **Link Establishment** phase of LAPD. To assist the reader, we make the following comments:

- Only the steps of the algorithm that are applicable to the transitions being investigated are shown in the table.
- 'F' is an input_dependent variable, therefore any value needed to make the predicate true can be provided by an input parameter.
- 'able_to_estab' is a system-state_dependent variable. Its value is therefore not directly controllable in any interaction.
- 'L3_initiated,' 'V_S' and 'V_A' are action_dependent variables whose values must be set explicitly before any definite true/false judgement can be made about the truth-value of the predicate.

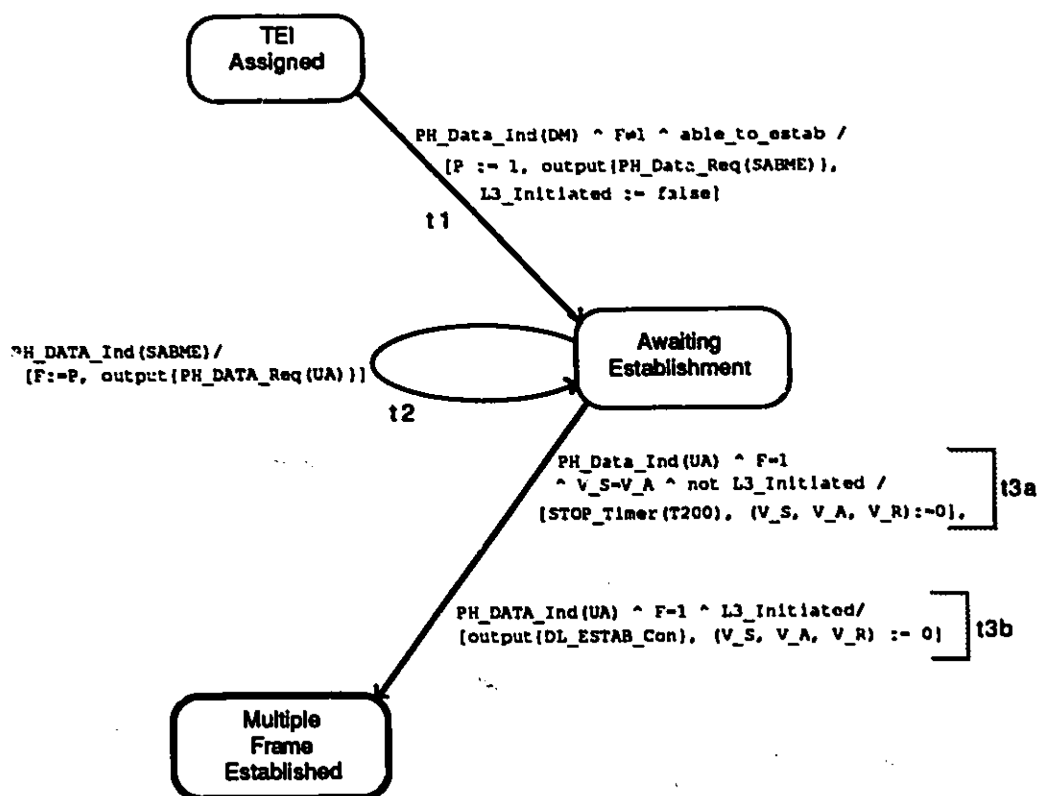


Figure 4.5: Complete Paths from Link Establishment Phase of LAPD

Applicable Step	Subpath	Transition Investigated	Predicate	Action on Variable	Verdict on Subpath
6c	$\{t_1\}$	t_1	F can be made ≠ 1 by input parameter; value of able_to_estab is unknown.	L3_Initated := false	possibly infeasible (able_to_estab is system-state dependent)
6d	$\{t_1, t_{3a}\}$	t_{3a}	F can be made = 1; not (L3_Initiated) is true. V_S and V_A values not set.	V_S := 0 V_A := 0 V_R := 0	possibly infeasible (values of V_A, V_S unknown)
Verdict on complete path $\{t_1, t_{3a}\}$ = "possibly infeasible"					

Table 4.1: Results of applying Algorithm CLASSIFY to path $\{t_1, t_{3a}\}$ of figure 4.5

Applicable Step	Subpath	Transition Investigated	Predicate	Action on Variable	Verdict on Subpath
6c	$\{t_1\}$	t_1	F can be made ≠ 1 by input parameter; value of able_to_estab is unknown.	L3_Initated := false	possibly infeasible (able_to_estab is system-state dependent)
3	$\{t_1, t_2\}$	t_2	none	F := P	possibly infeasible
6d	$\{t_1, t_2, t_{3b}\}$	t_{3b}	F can be made = 1; L3_Initiated is false.	V_S := 0 V_A := 0 V_R := 0	infeasible (since L3_Init is false)
Verdict on complete path $\{t_1, t_2, t_{3b}\}$ = "infeasible"					

Table 4.2: Results of applying Algorithm CLASSIFY to path $\{t_1, t_2, t_{3b}\}$ of figure 4.5

The above example shows one of the main steps in the determination of coverage of a test suite as presented in this thesis. In the following section, we will see where this algorithm is applied, and how the results are used.

4.3 DETERMINATION OF TEST COVERAGE

We will now present the sequence of steps that are proposed as a method of determining the coverage completeness of a protocol conformance test suite. It should be emphasized again at this point that this procedure is applied before any attempted execution of the test suite. In this section we will fully define our proposed coverage metric, and present a method to determine whether a given test suite satisfies this metric. Before we give the details of this however, we need some supporting terms and definitions.

Let $<t_1,t_2>$ represent the event 'transition t_1 is fired, then transition t_2 is fired.' Then a transition sequence represented by *TS* is defined to be the event $<t_1,t_2,...,t_{n-1},t_n>$ where $n \geq 1$. A null sequence is one in which no transition is fired, represented by $< >$.

Definition 4.12: An *actual trace* is a transition sequence which has been executed (single-valued test case with instantiated inputs and outputs) in a given protocol implementation.

Definition 4.13: A *potential trace* is a transition sequence which has not been executed but which is intended to be executed as (part of) a test case in a given protocol implementation.

Definition 4.14: The *trace set* of a given protocol specification, represented by *Trace(PS),* is the set of all traces which can be obtained by traversing all paths in the protocol. (This set is possibly infinite and of course may contain infeasible sequences in the set of potential traces).

4.3.1 Restrictions on Paths for Generating Test Cases

From the point of view of coverage, a test suite would be complete if it contained a test case for each trace in *Trace(PS)*. However, because this set may be infinite, this is not a reasonable goal. To get a realistic test suite, we must restrict the paths we consider in some way. For our study, we restrict our paths in two ways:

1. The number of repetitions of any trivial loop in the path is restricted to at most three particular values, *viz*, one (1), some finite constant $k>1$ (discussed below), and $k+1$.
2. The path must either:
   a. Start at the start node $m_s \in S$ of some **DBM** and finish at the target node $m_t \in T$ of some (not necessarily the same) **DBM**. Thus we consider complete (or combinations of complete) paths through the set of **DBMs**.
   b. Start at the start node of a **DBM** and finish at any node in an **ERM** which represents the state reached on completion of a set of actions that cause an error condition. In this case, the path through the **DBM** may be partial (i.e. not complete).

There is little loss in generality in making the above assumptions, because they represent most cases that are desirable to test. Restriction (1) says that if we traverse a loop at all, we should be able to traverse it at least once. We should also be able to traverse it k times without error where the selection of k depends on information extracted from the protocol specification. For example, in the multiple frame mode of operation of LAPD, we can send up to 128 I_frames without receiving an acknowledgement. Thus in the case where we have a loop that involves sending information, we might select k to be 128. If we try to traverse the loop more than k times, this should produce an error on the first such attempt (hence $k+1$). In all cases of attempted traversal over k times, a similar error should be signalled as that for $k+1$; thus, the $k+1$ case represents all cases $> k$. This is analogous to standard boundary-interior testing techniques [How75].

Restriction (2) is needed simply because of the approach taken in this thesis. Each test path should represent a functionally meaningful interaction. Such an interaction is of course represented by a complete path through a **DBM**, or by some (perhaps partial) path through a **DBM** followed by a path in an **ERM**.

4.3.2 A Test Coverage Metric—Phase Coverage

It would be desirable, from the standpoint of coverage completeness, that every path through the protocol specification could be tested. However, it is well known that this is not practical [LiFr86, Howd80]. A less demanding but effective coverage requirement is now proposed.

Let $\eta(P_i)$ be the number of paths in a phase, i.e. the number of complete paths (with restrictions on loops as given above) in the **DBM** or **ERM** representing the phase. Then we define the *connected-phase complexity* C_p, of the protocol to be the sum of the number

of paths in each phase of the protocol. More precisely, if the protocol specification is decomposed into n phases, then

$$C_\rho = \sum_{i=1}^{n} \eta(P_i).$$

Note that C_ρ is the path complexity of the union of all the DBMs and ERMs as described in section 4.1.6, and that C_ρ is much less than the path complexity of the original protocol specification. This complexity gives us a measure of the minimum number of test cases required to provide complete phase coverage of the protocol. It is defined as the sum of the number of distinct complete paths through each phase because each distinct path can be represented by a unique test case with a distinct purpose, and in protocol testing we take the view that each identified test purpose should be represented by a unique associated test case.

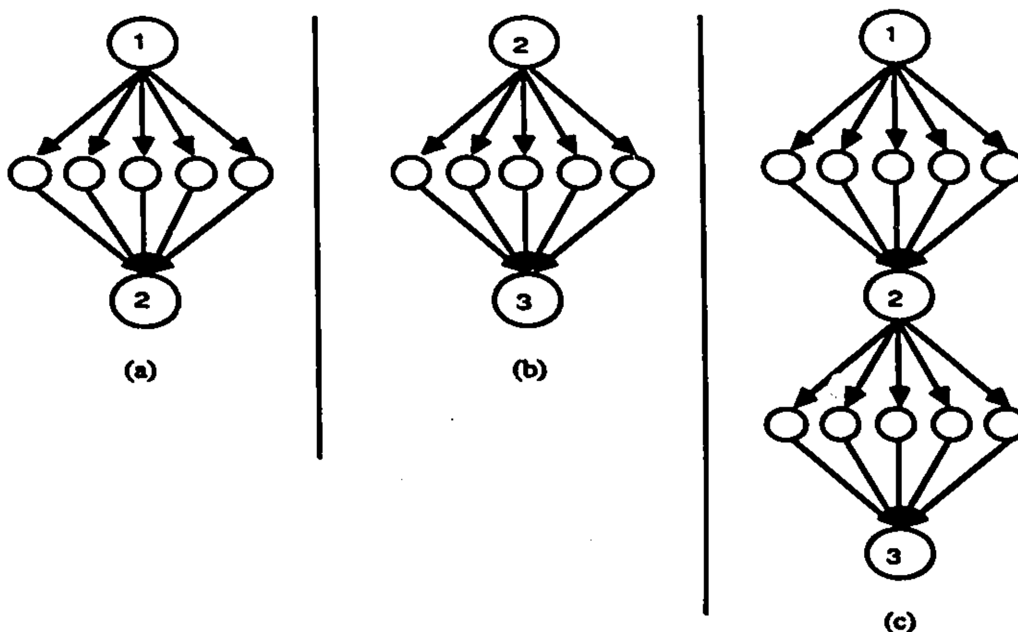


Figure 4.6: Paths in connected phases

If we look at figure 4.6, we see that there are 5 paths in each of (a) and (b), but there are 25 from node 1 to node 3 in (c). If figure 4.6(a) and (b) represented phases of a protocol, then we would require that every path in (a) be covered, and similarly for (b). The total number of paths to be covered is then 10. Even when the two phases are joined as in figure 4.6(c), the minimum number of paths to be covered would still be 10, rather than

25. This is an important point. Based on our metric, there is no path explosion when phases are joined. Thus decomposing the protocol into phases allows us to select a manageable number of paths to be tested. For the whole protocol specification, the total number of paths considered would be C_P. The minimum coverage criterion then is that for each normal behaviour phase, every complete path (with restrictions as given previously) in the phase should be traversed by a unique test case, and in the case of error/recovery phases, every transaction or (ordered) set of transactions corresponding to an error condition should be fired by a test case. When this criterion is satisfied by a test suite, that test suite is said to have achieved ***phase coverage*** of the protocol. This coverage criterion, while not as comprehensive as total path coverage, is more comprehensive than transition testing in the sense that it covers functional paths rather than functionally unrelated transitions.

4.3.3 Test Suite Decomposition

Recall that phases are represented by DBMs and ERMs. In this section we want to show that the set of DBMs and ERMs can be used as an aid in decomposing the test suite into a number of trace subsets in order to help determine the *phase coverage* of the test suite. From any DBM representation of the phase, we can obtain (1) purpose of the module and (2) capabilities and some constraints. These properties will be used in the manner illustrated in figure 4.7, and we will expand on that shortly. The general idea of the method of determining phase coverage is shown in figure 4.8.

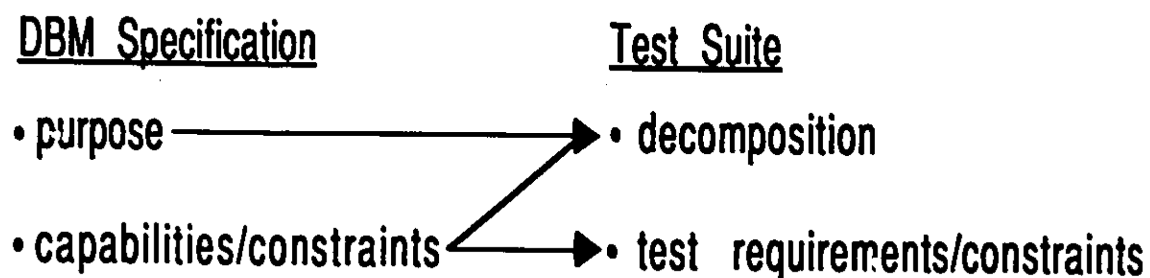


Figure 4.7: Mapping from DBM to test suite

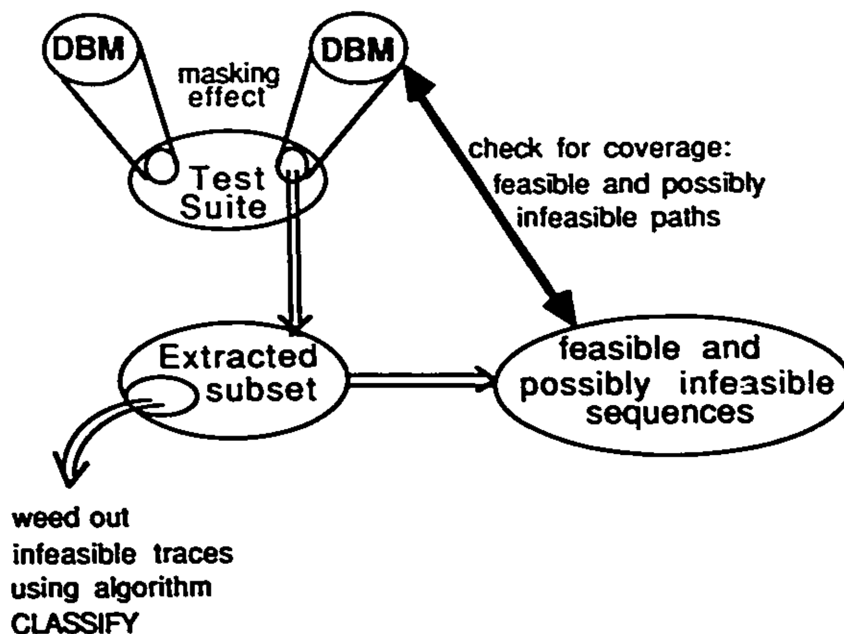


Figure 4.8: Overview of coverage determination

As mentioned above, we can extract a purpose, capabilities and constraints from the specification of a DBM. The purpose of a DBM bears a one-to-one relationship with the goal of its corresponding phase. The purpose associated with each DBM is to reach the set of target nodes T, starting from the set of start nodes S. There are constraints embedded in the specification of the DBM in that the paths are all directed from S to T and that we are restricted not only as to the number of times a loop is traversed but also in the kind of loop (or cycle) that is allowed on the path. Complex cycles are not found in DBM specifications; only simple cycles are found (in fact, the presence of complex cycles would indicate the presence of subpaths from another phase). The fact that there are certain constraints associated with the paths in a DBM means that we can only select certain test sequences from the test suite to be applied to the phase. Test sequences with integral complex cycles for instance can not be selected even though part or all of the path from some $m_s \in S$ to $m_t \in T$ may be included in the transition sequence of the test. Thus we have some test requirements that are imposed by the structure of the DBM itself. In addition, when the purposes of test groups within the test suite are stated, this information can be used to help select the DBMs which ought to be associated with the test groups.

This information can now be used to help select trace subsets from (in other words, to decompose) the test suite. We examine the test suite and extract from it all the test

sequences which appear to cover every complete path in the **DBM**. For our purposes, we may note that:

> **Definition 4.15:** A path is said to be *covered* by a test sequence if all the transitions of the path are sequentially fired when the test sequence is executed.

> **Definition 4.16:** A **DBM** is said to be *covered* by a set of test sequences if every complete path in the **DBM** is covered by that set of test sequences.

Thus each **DBM** or **ERM** can be used as a '*mask*' to extract certain test cases from the test suite (see figure 4.8). We can decompose the test suite in this way since no single complete path is in more than one **DBM** or **ERM**. This assures us that the potential traces (test sequences) which are extracted for the **DBM** under consideration can not be (and hopefully were not intended to be) applicable to some other **DBM**. Note that the sequences are selected based on the considerations given to cycles in the previous paragraph, and they all start at the start node $m_s \in S$ and finish at the target node $m_t \in T$. Thus , using this masking effect of the **DBMs** and **ERMs**, we have a clear way to decompose the test suite. Remembering that **DBMs** and **ERMs** represent phases, we see that we have decomposed the test suite in accordance with the phases of the protocol.

4.3.4 Measurement of Phase Coverage

Not every potential trace which is extracted from the test suite may be executable, i.e. the path which would have to be traversed on execution may be infeasible. The next step is to apply algorithm CLASSIFY to every potential trace in the set and eliminate any traces which are classified as infeasible. We retain the set of traces which are found to be possibly infeasible however, since we are not sure of their feasibility and they may turn out to be feasible when appended to the end of some other potential trace of some other **DBM** (the case where one phase must be executed before another). Together with the set of feasible potential traces, we now have a new set that is made up of two types of traces (feasible and possibly infeasible).

The next step is to match this new set of traces to the set of paths in the **DBM**. The paths are considered under the restrictions given in section 4.3.1, namely that repetition of loops is restricted to a finite number and the path must be complete. Bearing these considerations in mind, ideally every complete path should be covered at least once by a potential trace. Using this as a standard of measurement then, we can count the number of

paths that are not covered, and the number that are. We also make a distinction between paths that are covered by feasible potential traces and paths that are covered by possibly infeasible potential traces. The following definitions are now in order.

Definition 4.17: Paths that are covered by feasible potential traces are said to be strongly covered and the corresponding test sequences are said to provide *strong coverage.*

Definition 4.18: Paths that are covered by possibly infeasible potential traces are said to be weakly covered and the corresponding test sequences are said to provide *weak coverage.*

If every complete path in a phase is covered at least once by feasible potential traces alone, then we achieve 100% strong coverage. If every complete path is covered at least once by possibly infeasible potential traces alone, then 100% weak coverage is achieved. This distinction is made between strong and weak coverage because it is important for the test designer to know that some of his test cases are possibly infeasible. His aim should be to have as high a percentage of strong coverage and as low a percentage of weak coverage as possible. Since he knows which test cases are possibly infeasible, in his test plan he can try to append such test cases to the end of a test case (from a previous phase) in which the uninstantiated action_dependent variable is assigned a value. We should note at this point however that a detailed discussion on test case design methodology based on a phase decomposition of the protocol is beyond the scope of this thesis.

As an example, suppose that there are twenty distinct paths identified in a DBM (phase) and that we have found fifteen feasible potential traces and two possibly infeasible potential traces in the test suite which cover seventeen of these paths. Then we can say that for this phase, the test suite achieves 75% strong coverage, 10% weak coverage, or simply 85% combined phase coverage. Even though we have defined these terms (i.e., % strong and weak coverage) with respect to phases, we can also use them to describe how a test suite covers the entire protocol. Notice how this typed, structural definition of coverage is quite different from some other previous definitions which take into consideration the number of errors discovered during testing [HeKr87].

Chapter 5

5. Assessment of Method

5.1 COMPARISON WITH RELATED TECHNIQUES

In chapter 3, we briefly discussed some test coverage criteria and applicable coverage measurement techniques developed over the past few years. Now we will assess the method presented in the previous chapter in relation to those applicable techniques presented in chapter 3, as well as some recent techniques for decomposing protocol specifications.

To aid in this assessment, recall that a broad overview of the method presented in chapter 4 would be:

1. Determine the phase decomposition of the protocol specification.
2. Cover every feasible path through each phase by a test sequence (including single tests for each transition or set of transitions representing single errors in error/recovery phases).

In [Howd86], Howden gives the essential basis for this method. With regards to program code, he suggests that functions be first identified, and that only those paths which correspond to meaningful, readily identifiable requirements, design or program functions be tested. Rather than considering code, however, we extend this idea to the protocol specification. When the protocol is specified in an FDT language such as Estelle, this is not very difficult. Functional units of the protocol specification are determined. We call these units phases of the protocol, and the objective is then to test every complete path through each phase. This is stronger than the transition tour criteria, since if every path in each phase is covered, the every transition is also covered. The DS and UIO sequences discussed earlier are usually applied in association with 'single transition at a time' testing,

and are therefore not generally applicable to our method. However, they can be used to verify that the goal state is reached at the end of a complete path.

One of our objectives was to create 'semantic phases,' i.e., phases which had associated with them and their structure some information which allowed them to be identified as performing some readily identifiable function. To do this, we consider a functional decomposition of the protocol, based on purposes and intended accomplishments. This method obviously requires subjective judgement, but any method which attempts to reconstruct what might have been the original design goals will require some subjective judgement.

When the protocol is specified in the form of a directed graph (e.g. FSM), there exist few methods which do not require any subjective judgement in their application for finding decompositions of the protocol. Chow et al [ChGo85] give a complex definition for a 'phase,' and any substructure within the protocol graph which fits this definition is considered to be a 'phase.' Choi et al [ChMi86] describe a method where they start with the smallest 'structured partition' of the protocol graph (essentially, each node is considered to be a separate subgraph) and from that form larger partitions. This process stops when all the subgraphs have the required properties (*viz*, no unspecified receptions, no nonexecutable interactions, no state deadlocks). For these two methods, it is not clear beforehand whether the resulting subgraphs (which I will refer to as 'syntactic phases') will correspond to meaningful functional units of the protocol. After these syntactic phases are determined, one must then find a correspondence between them and meaningful functional units of the protocol (a subjective process in itself). For large protocols, these methods are very tedious (but can be automated). Moreover, not every protocol is decomposable using these methods [BrZa83, ChGo85, ChMi86].

The semantic phases of this thesis are similar to the control phases mentioned by Sarikaya et al [SaBo87], who have also developed a method of decomposing a protocol based on data flow considerations. Their method consists of first deriving a Data Flow Graph (DFG) from a normal form Estelle specification of the protocol. The DFG is then decomposed into elementary blocks. Here, a block consists of all the nodes participating in a distinct flow of data between certain types of nodes. These elementary blocks are then combined into larger functional blocks. Interaction with the test designer is usually necessary to determine which elementary blocks should be merged. Sometimes the resulting functional blocks coincide with control, or semantic phases, as presented in chapter 3. Whereas the method in [SaBo87] is based on the flow of data, the method presented in this chapter is is based on the control flow, with particular attention paid to

identifying original protocol functional units. A careful study of the protocol specification using the method presented will allow these units to be identified. The method presented in this chapter also emphasizes the directed nature of control paths in a phase, and in fact uses this as a basis for the structural composition of the phase. This is unique among the methods presented.

There is also quite a difference in the way the semantic phases of this chapter are covered and the way the functional blocks of [SaBo87] are covered. The method of this chapter suggests a path based strategy, where a complete path through a phase is individually tested (and thus a phase is executed from beginning to end). The ordering of the phases is also considered in our testing process. On the other hand, the method of [SaBo87] is based on the transition tour of the protocol graph. A subtour (sequence of transitions from the idle state of the protocol to the idle state) is selected and executed. All the functional blocks which are partially covered by this subtour are noted. The aim of this method is to have each block tested using all subtours of the block.

A number of advantages of the method of determining the phase coverage of a test suite as presented in this thesis have been identified. These are mainly the following:

- The selection of paths which comprise a phase is a largely mechanical process, implying that automation might be partially possible, so long as the protocol specification is available in a machine readable form.
- Some indication is given as to whether or not test sequences in the test suite are feasible with respect to paths in the phases.
- Since test suites can be presented in machine processable form (e.g. TTCN–MP), and the **DBM** and **ERM** representation of phases can easily be made machine processable also, we are thus brought closer to the point where it will be possible to automatically determine the completeness of the test suite.
- The machine processable **DBMs** and **ERMs** could be used in the design phase of test suite development as an aid in the automated generation of functional and specified error tests. These tests make up a major part of a conformance test suite.

The last two points require a straightforward extension of this work, i.e. designing appropriate data types for **DBMs** and **ERMs** that would allow these to be presented in machine processable form. This includes simple structures which would allow the set of start and target nodes to be identified and defined along with all other nodes of the **DBM/ERM**, in addition to some link or pointer structure which would allow the transitions between nodes to be represented.

Our method however is not without its disadvantages; in particular:

- It does not take into consideration the general interphase complexity of normal behaviour phases in determining the connected phase complexity of the protocol.
- The construction of phases requires the use of subjective judgement in the determination of goals. Thus different phase decompositions of the same protocol may exist. Similar disadvantages exists for related approaches.
- The production of a phase decomposition requires a good understanding of the protocol. Any manual derivation of phases will thus tend to be error prone.

Whether or not our method will be useful to anyone as a means of test suite assessment will depend on how far, in their opinion, these advantages outweigh the disadvantages. Our method provides better coverage than that provided by current techniques, but the cost-effectiveness of this method needs to be investigated. Based on present industry concerns with productivity, our method appears to be quite attractive.

5.2 EXPERIENCE WITH ISDN LAPD

In this chapter, we present an example using a LAPD basic interconnection test suite (given in appendix 2) developed specifically for the purpose of demonstrating the application of the algorithm and method of determining coverage which were presented in chapter 4. Appendix 1 gives a phase decomposition of the LAPD protocol, along with the **DBMs** and **ERMs** that were derived in the process.

In considering test coverage of a test group, the purpose and scope of application of the tests must be taken into account. For example, for a basic interconnection test group, we want to set up the link, transfer data and release the link. Therefore the coverage of the basic interconnection test group is not measured with respect to timer recovery or error recovery. However, it could contribute to the overall test suite coverage of such unintended phases.

In considering paths through a phase, all possible permutations of transitions are taken into account. However, in order to simplify this example, we assume that loops on any complete path through a phase are executed at most once for any traversal of that path.

As noted above, the first thing that we need to do is determine the purpose of the test group. This will allow us to choose the phases (and hence which **DBMs** and **ERMs**) of the protocol specification to consider. For the LAPD basic interconnection tests, the purpose is to establish a connection, send some data, receive acknowledgement and release the link. The phases corresponding to these actions are **TEI Assignment**, followed by

Link Establishment, Data Transfer and **L3 Initiated Link Release** or **Peer Initiated Link Release**, possibly followed by **TEI Removal** in that order.

Next we consider the dynamic behaviour part of the test suite in conjunction with the DBMs of the phases selected above. Each table in the dynamic behaviour part represents a subpath through the protocol. Since phase coverage is measured as a correspondence between complete paths through phases and traces in the test suite, the test body in each such table should represent at least a complete path through a phase. The traces in the dynamic part are compared to the paths in the phases and any traces that are found in the tables but do not have corresponding paths in a phase are noted and discarded (i.e. not considered further). In our example, the traces (test cases), paths and their corresponding phases are listed below in table 5.1. Note that test t2 has no corresponding path in any phase, indicating that this coverage method does not look at all possible test cases. The reason why t2 is not found in any one phase is because the activities involved in this test case are based on the interaction of the management entities (not specified in this example) and this results in a test case that spans both **TEI Assignment** and **Data Transfer** phases. The path through the phase is expressed as a sequence of edges (ordered pairs of node numbers assigned according to the MBF numbering scheme).

Test Case (Identifier)	Phase	Path
t1	TEI Assignment	{(1,4)}
t2	-	-
t3	TEI Assignment	{(1,2),(2,4)}
t4	TEI Assignment	{(1,3), (3,5)}
t5	Link Establishment	{(1,1), (1,3)}
t6	Data Transfer	{(1,1), (1,1)}
t7	Peer Init. Link Release	{(1,4)}
t8	L3 Init. Link Release	{(1,3), (3,4)}
t9	TEI Removal	{(4,8)}
t10	TEI Removal	{(3,8)}

Table 5.1: Paths through phases corresponding to test cases in LAPD test suite

Our next step is to use algorithm Classify to take a closer look at these paths. To do this we need to consider the variables associated with predicates in the paths as described in chapter 3. The results of this application are shown in table 5.2.

Table 5.2 shows that of the nine potential traces to which we can relate a single phase, one is possibly infeasible and eight are feasible.

For **TEI Assignment** phase, our test suite has three feasible traces. The **DBM** representing that phase (see figure A1.3, appendix 1) shows that there are six complete paths in that phase. Thus we have 3/6×100 = 50% strong coverage. Results for other phases are derived similarly, and are shown in table 5.3.

To help give some meaning and perspective to the values in table 5.3, let us consider the overall situation. For the purpose of basic interconnection testing we take a restricted view of the protocol (i.e., we do not consider error/recovery transitions), and C_ρ can be taken to equal to 129 (the sum of the number of complete paths in each applicable phase). Nine applicable test cases have been given, eight of which provide strong coverage the remaining one weak coverage. For the entire test suite then, we have $(8/C_\rho)\times 100 = 6.2\%$ strong coverage and $(1/C_\rho)\times 100 = 0.8\%$ weak coverage. This gives a total of 7.0% combined phase coverage. Even for basic interconnection testing, this is clearly quite small.

Test Case	Phase	Verdict
t1	TEI Assignment	feasible
t2	-	-
t3	TEI Assignment	feasible
t4	TEI Assignment	feasible
t5	Link Establishment	feasible
t6	Data Transfer	possibly infeasible
t7	Peer Initiated Link Release	feasible
t8	L3 Initiated Link Release	feasible
t9	TEI Removal	feasible
t10	TEI Removal	feasible

Table 5.2: Feasibility of test paths in LAPD test suite

Phase	No. of Paths	No. of Tests	% Weak Coverage	% Strong Coverage
TEI Assignment	6	3	0	50
Link Establish.	45	1	0	2.2
Data Transfer	5	1	20	0
Peer Init. L/R	35	1	0	2.9
L3 Init. L/R	34	1	0	2.9
TEI Removal	4(applicable)	2	0	50

Table 5.3: Phase coverage of LAPD test suite

With the information provided by the above analysis, it is now a straightforward matter to examine the phases, i.e. the set of **DBMs** and **ERMs** and determine which paths (for **DBMs**) and set of transitions (for **ERMs**) have not been covered. For every such path or set of transitions, a new test case can be written, thus increasing the phase coverage of the test suite, and enhancing its completeness. In our example, in order to achieve a modest 50% phase coverage, 65 (i.e. approximately half of C_p) applicable test cases would have to be provided. Thus we would have to add 56 more test cases to our test suite of appendix 2 to achieve this.

This kind of analysis is certainly useful, as it will show any imbalance in the distribution of test cases over the phases. The addition of test cases can then be made so as to insure that each phase is equally well tested, or is in fact tested to any extent that the test designer desires. Thus, phase coverage appears to be a reasonable test case coverage requirement. Further investigation with other real protocols is therefore justified.

Chapter 6

6. Conclusion

6.1 SUMMARY OF WORK

The intention of this thesis was to investigate a method whereby some judgement could be made concerning the completeness of a given protocol conformance test suite (CTS). While there are certainly many approaches to this problem, the direction taken in this thesis was prompted by the ISO's recommendation [ISO3] that tests relating to each protocol phase be included in a CTS. This recommendation was extended however, in that we not only consider the inclusion of tests for each protocol phase, but rather that a phase decomposition of the protocol specification could be used as a source of all specified behaviour tests in the CTS. This in turn means that a phase decomposition of the protocol specification can be used to check the coverage completeness of the CTS as far as specified behaviour is concerned. No effort was made in this thesis to consider tests for unspecified behaviour. However, it is generally accepted that in order to obtain a more complete CTS, valid behaviour tests should be augmented with a set of defensive behaviour tests. This thesis focused only on valid (i.e. specified) behaviour of the protocol and looked at how a phase decomposition of the protocol specification could be used to determine the extent to which the CTS covered the specified behaviour of the protocol.

To obtain a phase decomposition of a protocol specification, we first tried to formalize the notion of a phase. This led to the development of a model that could be used as a representational basis for phases. From this development, normal behaviour phases were represented by **DBMs** and error/recovery phases by **ERMs**. Having laid this foundation, a technique for obtaining the phases of the protocol was described. The application of this technique assumed that the protocol was specified using some formal

state/transition based technique, e.g. Estelle. However, even if this were not the case, a precise English specification of the protocol could be transformed into an FSM from which the phases could be derived.

Before the phases were applied in an effort to determine the coverage of the CTS, an attempt was made to classify the complete paths in a **DBM** in terms of their executability (i.e. feasibility). This was done because the coverage of the CTS was determined in terms of the correspondence between test cases in the test suite and complete paths in the **DBMs**, and therefore, some idea of the feasibility of the test cases could be obtained before execution of the CTS is attempted. Algorithm 'CLASSIFY' was developed as an aid in determining this feasibility, and categorized the paths (hence test cases) as either 'feasible,' 'infeasible' or 'possibly infeasible.'

One of the main concepts developed in the thesis was the phase coverage of the CTS, where the aim was to have every (feasible) path in each identified phase covered by at least one test case. The concept was further refined to include the notion of strong and weak coverage, depending on whether the test cases covering the phase correspond to feasible or possibly infeasible paths in the **DBM**. We then presented a method whereby the phase coverage could be measured, based on a partitioning (decomposition) of the test suite that is directly related to the phase decomposition of the protocol specification.

Some advantages of the scheme presented were identified, including the fact that the state/transition based representation of the phase is easily machine processable and part of the derivation of the phase is a basic mechanical process. However, the method is not without its drawbacks, of which one of the greatest is the necessity of using subjective judgement in the determination of the goals of the phases. Nonetheless, the method appears to be effective for estimating test coverage, and accordingly could be applied in test design incrementally as a guideline for the derivation of tests from the protocol specification [Boyc88].

6.2 DIRECTIONS FOR FUTURE WORK

The development of the test coverage methodology presented in this thesis has identified some areas which require additional work. These include:

- A need for more systematic, formal methods for decomposing protocol specifications. This would allow certain test related techniques (e.g. the determination of coverage as presented in this thesis) to be applied without the need to rely on subjective judgement.

- The concept of a phase as defined in this thesis has associated with it certain semantic information, *viz*, the accomplishment of a certain goal as indicated by the common target node(s) of the paths of the **DBM**. The explicit representation of this semantic information in the test cases themselves is a subject for future research.

- The coverage criterion could be extended from the present case of requiring at least one path to each **ERM** state, to a case that considers all possible paths from each logically preceding **DBM** to an **ERM** state. The feasibility of this approach is also a subject for further study.

- A machine processable representation of phases (i.e. some form of **DBM** or **ERM**) may be utilized in the *automatic* determination of the phase coverage of an already existing test suite. The mechanical nature of the process presented implies that this may be possible.

- The use of phases as defined in this thesis can be extended in that machine processable versions of the **DBMs** and **ERMs** can be used in the *automatic* generation of test cases to cover the specified behaviour of the protocol.

Communication protocols are becoming more complex. As a result, it is becoming more difficult and expensive to produce CTSs and to test implementations for conformance to these protocols. Therefore, one of the aims of test suite designers is to produce as complete a test suite as possible, while keeping the size of the suite within reasonable limits. It is suggested that the use of phases in the design stage of the test suite can help achieve this goal. Whether or not phases are used as an aid in the the construction of the test suite however, we conclude that in order to help ensure the completeness of the CTS, the phase coverage achieved by the test suite should be as close to 100% as possible, i.e., phase coverage is a necessary condition for completeness.

References

[AhDa88] Aho A. V. *et al*, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," *Eighth International Symposium on Protocol Specification, Testing and Verification*, pp 75–86, June 1988.

[BoBr87] Bolobnes T., Brinksma E., "ISO Specification Language LOTOS," *Computer Networks and ISDN Systems*, Vol. 14, 1987.

[BoGo86] Bochman G.v., Gotzhan R., "Deriving Protocol Specifications from Service Specifications," *Proc. ACM SIGCOM Symposium*, pp. 148–156, 1986.

[BoSu80] Bochman G.v., Sunshine C. A., "Formal Methods in Communication Protocol Design," *IEEE Trans. on Communications*, Vol. Com-28, No. 4, pp. 624–631, April 1980.

[Boyc88] Boyce T. *et al*, "Formalization of ISDN LAPD for Conformance Testing," to appear, INFOCOM '89.

[BrZa83] Brand D., Zafiropulo P., "On communicating Finite State Machines," *J. ACM*, Vol. 30, No. 2, April 1983.

[BuEc86] Burkhardt H., Eckertt H., Giesster A., "Testing of Protocol Implementations - A Systematic Approach to Derivation of Test Sequences from Global Protocol Specifications," *Fifth International Workshop on Protocol Specification, Testing and Verification*, pp 461–481, 1986.

[CCIT1] CCITT "Recommendation Q.920, ISDN User-Network Interface Data Link Layer - General Aspects"

[CCIT2] CCITT "Recommendation Q.921, ISDN User-Network Interface Data Link Layer Specification"

[Chow78] Chow T. S., "Testing Software Design Modelled by Finite State Machines," *IEEE Trans. on Soft. Eng.*, Vol. SE-4, No. 3, pp. 178–187, May 1978.

[ChGo85] Chow C. H., Gouda M. G., Lam S. S., "A Discipline for Constructing Multi-Phase Communication Protocols," *ACM Trans. on Com. Sys.*, Vol. 3, No. 4, pp 315–343, Nov 1985.

[ChMi86] Choi T. Y., Miller R.E., "Protocol Analysis and Synthesis by Structured Partitions," *Computer Networks and ISDN Systems*, Vol. 11, No. 5, pp. 367–381, May 1986.

[DeMi78] DeMillo R. A., Lipton R. J., Sayward G. F., "Hints on Test Data Selection: Help for the Practicing Programmer," *IEEE Computer Magazine*, pp. 34–41, April 1978.

[Gone70] Gonenç G., "A Method for the Design of Fault Detection Experiments," *IEEE Trans. on Computers*, pp. 551–558, June 1970.

[Huan78] Huang J. C., "Program Instrumentation and Software Testing," *IEEE Computer*, pp. 25–32, April 1987.

[HeKr87] Hengeveld W., Kroon J., "Using Checking Sequences for OSI Session Layer Conformance Testing," *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification*, pp 435–449, May 1987.

[Henn64] Hennie F. C., "Fault Detecting Experiments for Sequential Circuits," *Proc. 5th Ann. Symp. on Switching Theory and Logical Design*, pp. 95–110, Nov. 1964.

[Howd75] Howden W., "Methodology for the Generation of Program Test Data," *IEEE Trans. on Computers*, Vol. C-24, No. 5, pp. 554–559, May 1975.

[Howd80] Howden W., "Functional Program Testing," *IEEE Trans. on Soft. Eng.*, Vol. SE-6, No. 2, pp. 162–169, March 1980.

[Howd86] Howden W., "A Functional Approach to Program Testing and Analysis," *IEEE Trans. on Soft. Eng.*, Vol. SE-12, No. 10, pp. 997–1005, Oct. 1986.

[ISO1] ISO "Estelle—A Formal Description Technique Based on an Extended State Transition Model," DP9074, Oct. 1986.

[ISO2] ISO "Information Processing Systems - Open Systems Interconnection - Basic Reference Model," DP7498, (1983).

[ISO3] ISO "OSI Conformance Testing Methodology and Framework, Part 1: General Concepts," DP9646–1, Sept. 1986 (Egham).

[ISO4] ISO "The Tree and Tabular Combined Notation," DP9646–3, July 1988 (Sweden).

[ISO5] ISO "OSI Conformance Testing Methodology and Framework, Part 4: Test Realization," DP9646–4, May 1988.

[ISO6] ISO "OSI Conformance Testing Methodology and Framework, Part 5: Requirements on Test Laboratories and Clients for the Conformance Assessment Process," DP9646–5, May 1988.

[ISO7] ISO "LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," DP8807, March 1985.

[Kanu86] Kanungo B. *et al*, "A Useful FSM Representation for Test Suite Design and Development," *Sixth International Workshop on Protocol Specification, Testing and Verification*, pp 163–176, June 1986.

[Lala85] Lala P. K. *"Fault Tolerant & Fault Testable Hardware Design,"* Prentice-Hall International Inc., London, 1985.

[LiFr86] Linn R., Fravreau J., "Automatic Generation of Test Scenario Skeletons from Protocol Specifications Written in Estelle," *Sixth International Workshop on Protocol Specification, Testing and Verification*, pp 191–202,June 1986.

[Meer86] de Meer J., "Derivation and Validation of Test Scenarios based on the Formal Specification Language Lotos," *ibid*, pp 203–216.

[Miln80] Milner R., *"A Calculus of Communicating Systems,"* Springer-Verlag, Berlin, 1980.

[Myer79] Myers G. J., *"The Art of Software Testing,"* John Wiley & Sons, Inc., 1979.

[PrMy87] Prather R. E., Myers J. P., "The Path Prefix Software Testing Strategy," *IEEE Trans. on Soft Eng.*, Vol. SE-13, No. 7, pp. 761–765, July 1987.

[Pro82a] Probert R. L., "Life-Cycle/Grey-Box Testing," *Congressus Numerantium*, Vol. 34, pp. 87–117, 1982.

[Pro82b] Probert R. L., "Optimal Insertion of Software Probes in Well-Delimited Programs," *IEEE Trans. on Soft. Eng.*, Vol SE-8, No. 1, pp.33–42, Jan. 1982.

[Rath87] Rathgeb E.P. *et al*, "Protocol Testing for the ISDN D_Channel Network Layer," *Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification*, pp 421–434, May 1987.

[SaDa85] Sabnani K., Dahbura A., "A Procedure for Generating Protocol Tests," *Computer Comm. Review*, Vol. 15, No. 4, Dec 1985.

[SaBo84] Sarikaya B., Bochmann G., "Synchronization and Specification Issues in Protocol Testing," *IEEE Trans. on Communications*, Vol. Com-32, No. 4, pp.389–395, April 1984.

[SaBo87] Sarikaya B., Bochmann G., Cerny E., "A Test Design Methodology for Protocol Testing," *IEEE Trans. on Soft. Eng.*, Vol. SE-13. No. 5, 518–531, May 1987.

[ShHo86] Sherif M. H., Hoover G., Wiederhold R.P., "X.25 Conformance Testing - A Tutorial," *IEEE Communication Magazine*, Vol-24, No. 1, pp. 16–27, Jan. 1986.

[SiLe86] Sidhu L., Leung T., "Formal Methods for Protocol Testing: A Detailed Study," TR#86–23, Iowa State Univ. Research Report.

[Ural87] Ural H., "Test Sequence Selection Bases on Static Data Flow Analysis," *Computer Communications*, Vol. 10, No. 5, pp. 234–242, Oct. 1987.

[WoHe80] Woodward M. R., Hedley D., Hennel M. A., "Experience with Path Analysis and Testing of Programs," *IEEE Trans. on Soft. Eng.*, pp. 278–286, May 1980.

[Zafi80] Zafiropulo P. *et al*, "Towards Analyzing and Synthesizing Protocols," *IEEE Trans. on Communications*, Vol. Com-28, No. 4, pp. 651–660, April 1980.

Appendix 1

A1.1 Phase Decomposition of LAPD (Q.921)

The following diagrams present the phase decomposition of LAPD. Figures A1.1 and A1.2 show a general overview of this decomposition as well as the order in which these phases may be executed. Table A1.1 lists the names of the phases as referenced in figures A1.1 and A1.2. The states of the phases are shown numbered as they would be by the MBF numbering scheme. To save space, sometimes a primitive may be shown with more than one parameter, separated by slashes, e.g. `PH_DATA_Ind(RR_ind/REJ_ind)`. When this is encountered, it is to be understood that all instances of the primitive with each parameter could occur in that situation. i.e. in the example above, either `PH_DATA_Ind(RR_ind)` or `PH_DATA_Ind(REJ_ind)` could occur. A similar thing is done in the representation of transmission of signals, .e.g. `output{x,y}` means that signal *x* is generated and then signal *y* is generated in that order, where *x* and *y* represent primitives.

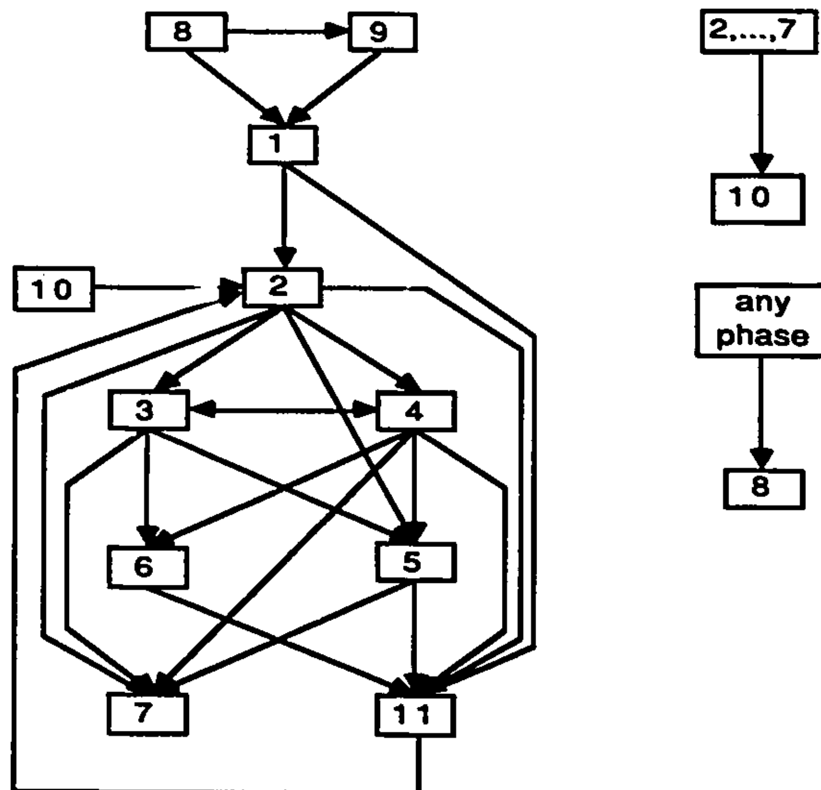


Figure A1.1: Phase Dependency Diagram for Normal Behaviour Phases

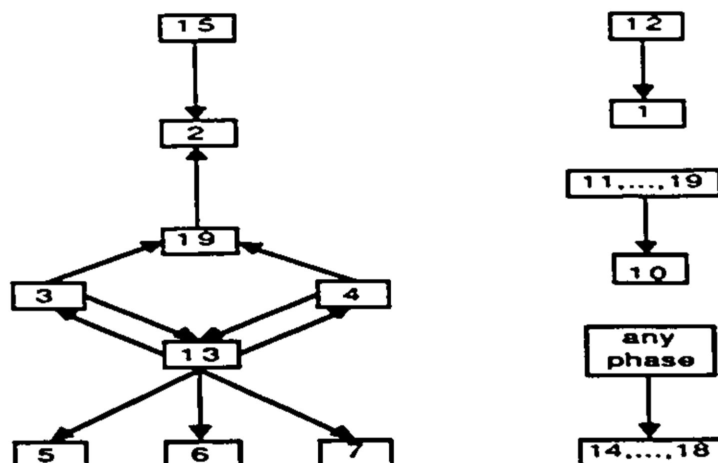


Figure A1.2: Phase Dependency Diagram for Error Related Phases

Fig.A1.1/A1.2 Ref	Phase Name
1	TEI Assignment
2	Link Establishment
3	Data Transfer
4	Data Transfer Timeout Recovery
5	Link Re-establishment
6	L3 Initiated Link Release
7	Peer Initiated Link Release
8	TEI Removal
9	Deactivation/TEI assigned
10	Deactivation/TEI unassigned
11	T200 Timeout Recovery
12	TEI Assignment Error Recovery
13	Attempted Peer Initiated Re-establishment
14	Invalid Frame Error Recovery/1
15	Invalid Frame Error Recovery/2
16	Unsolicited Response Error Recovery/1
17	Unsolicited Response Error Recovery/2
18	Unsolicited Response Error Recovery/3
19	Sequence Error and FRMR Recovery

Table A1.1: Names of Phases Referenced in Figs. A1.1 and A1.2

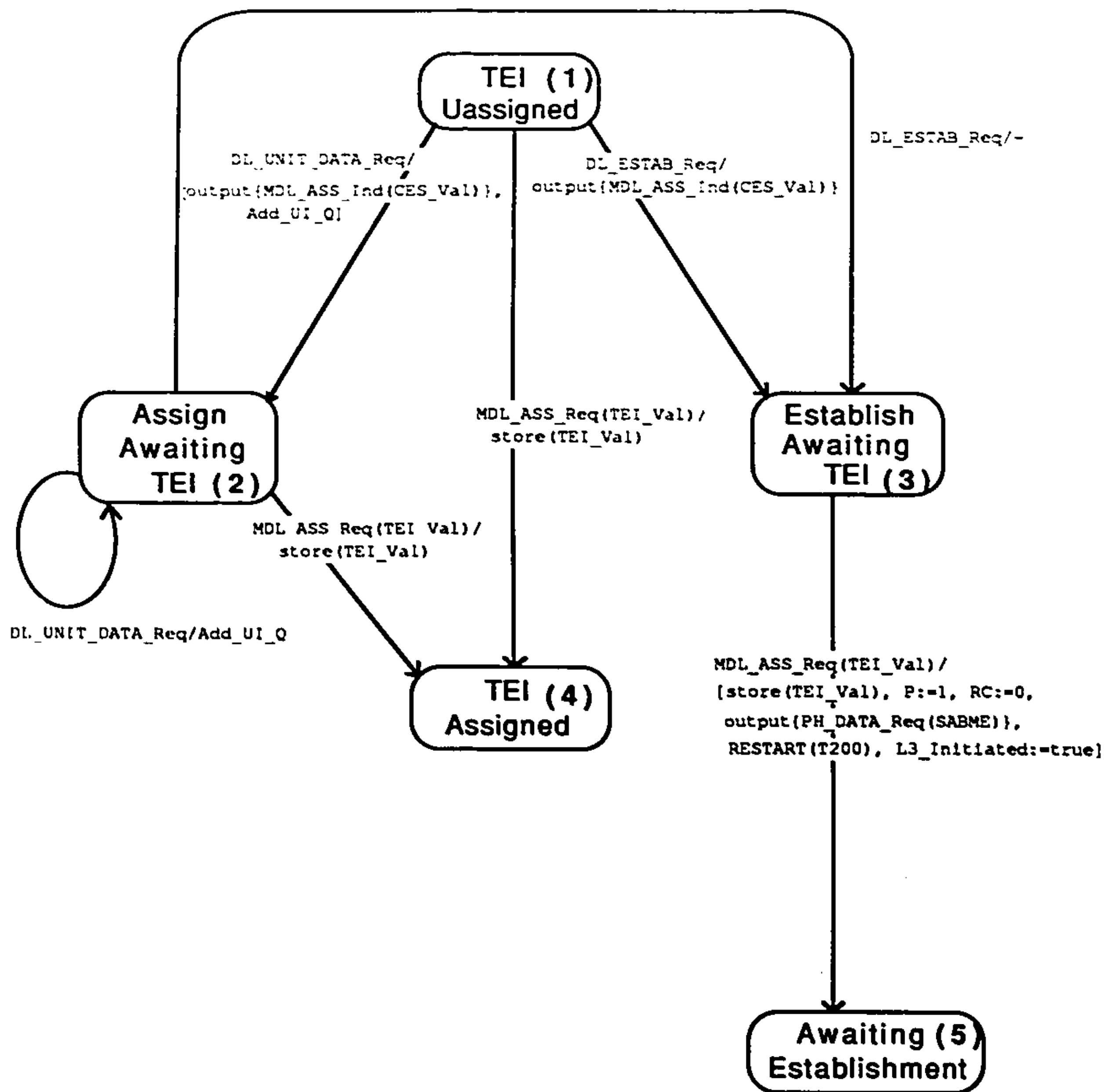
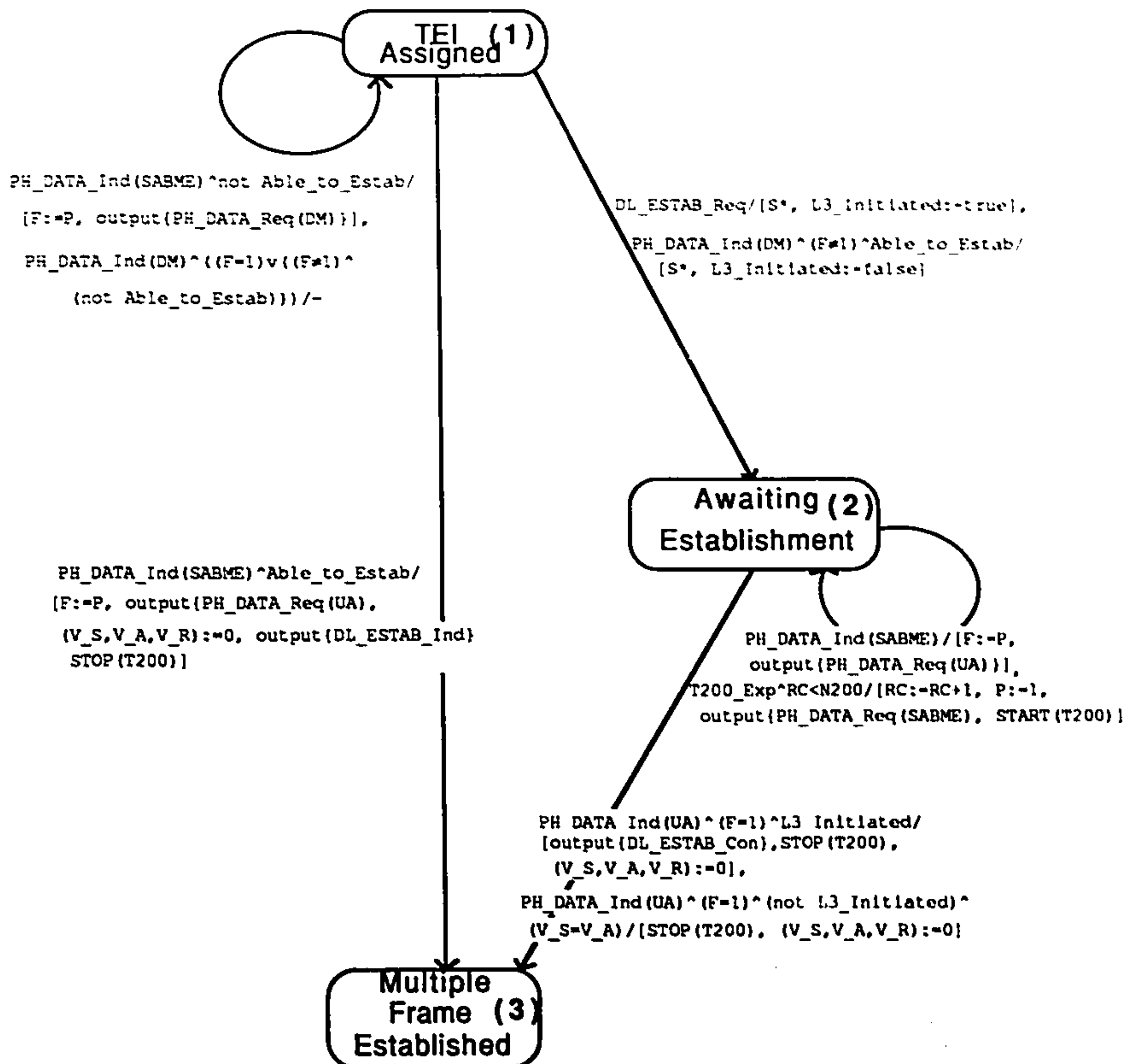


Figure A1.3: Phase: TEI Assignment

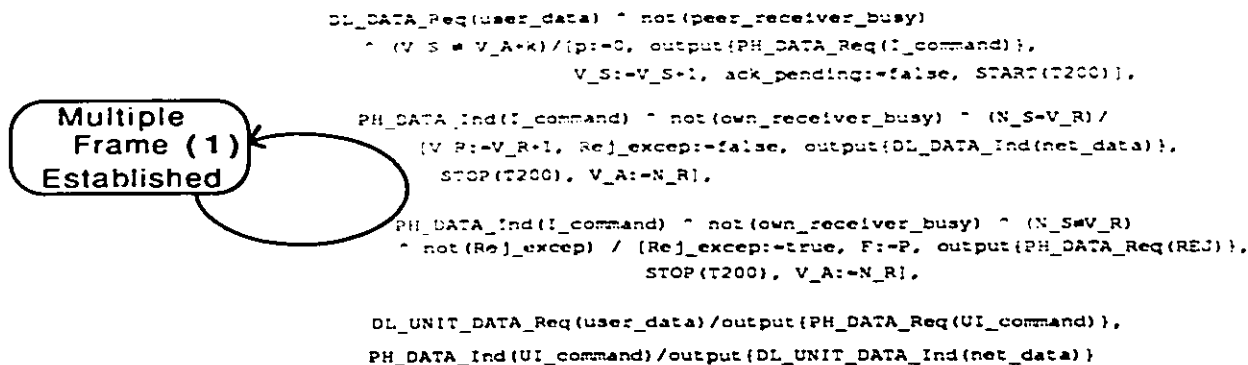


S* = {RC:=0, P:=1, output{PH_DATA_Req(SABME)}, RESTART(T200)}

Start State: {1}

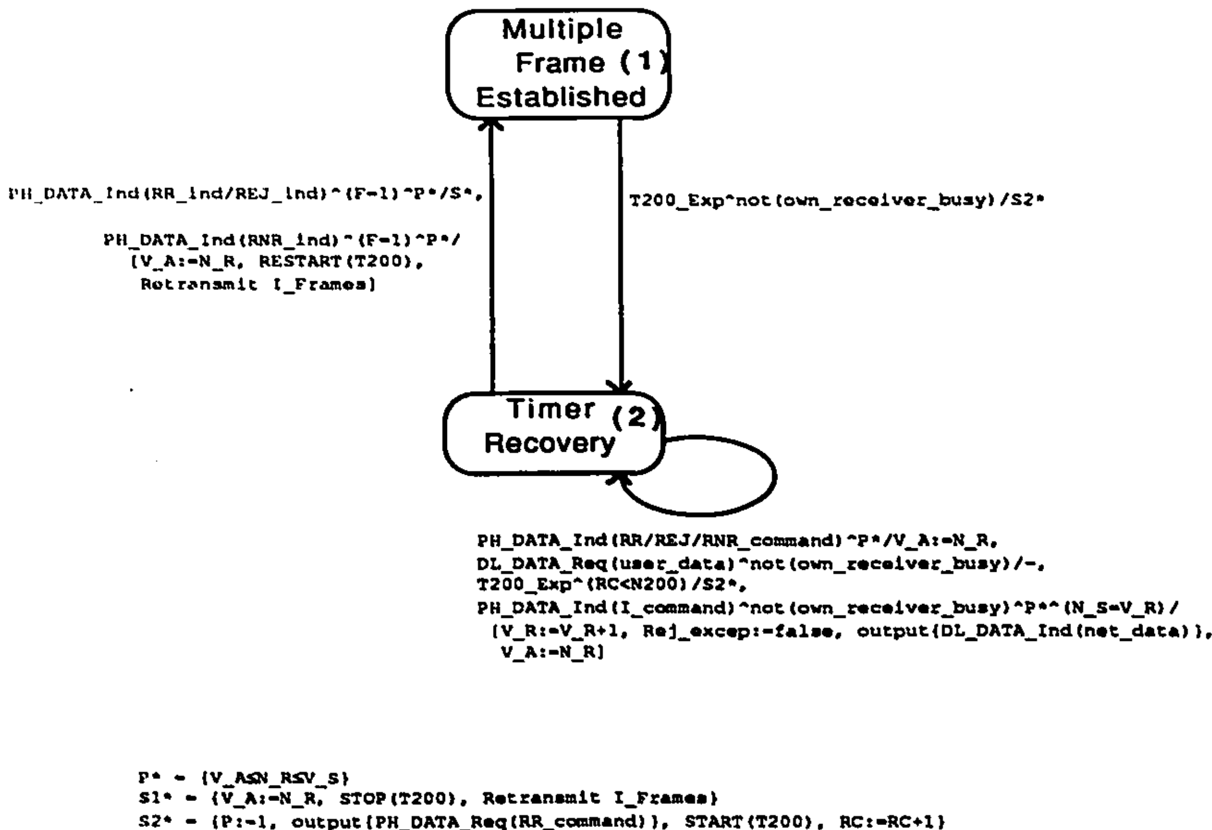
Goal State: {3}

Figure A1.4: Phase: Link Establishment



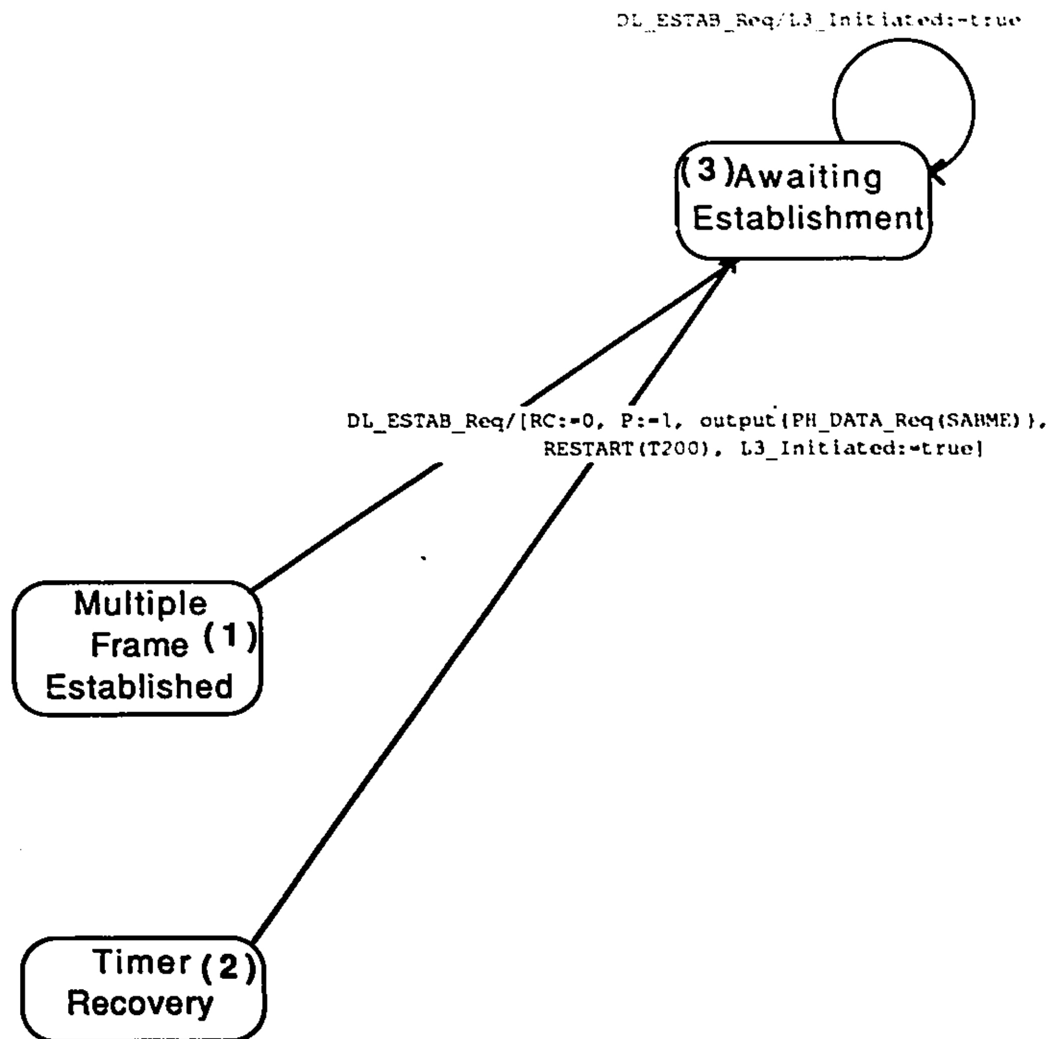
Start State: {1}
Goal State: {1}

Figure A1.5: Phase: Data Transfer



Start State: {1}
Goal State: {1}

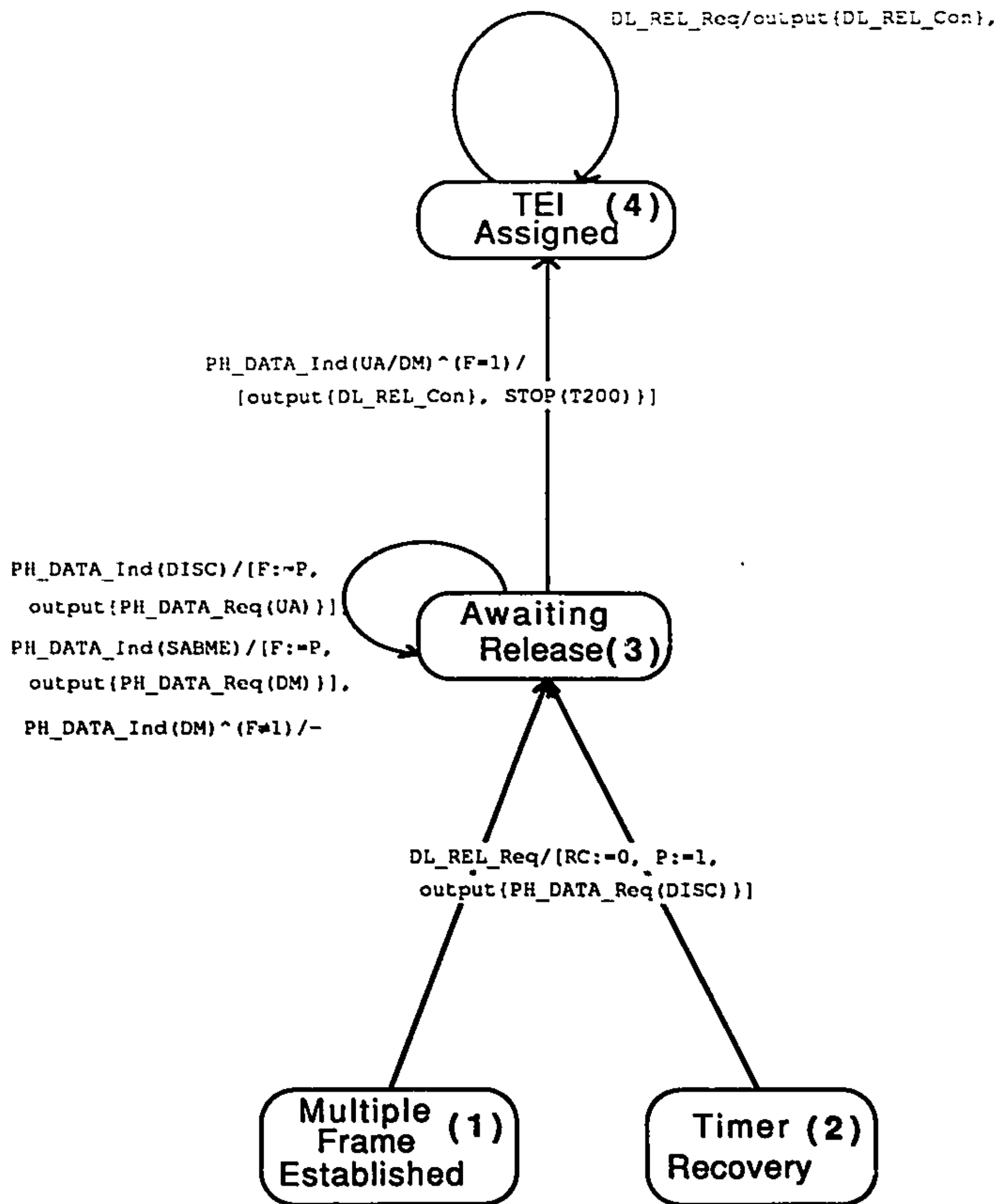
Figure A1.6: Phase: Data Transfer Timeout Recovery



S1* = {F:=P, output(PH_DATA_Req(UA), MDL_ERROR_Ind(F)), STOP(T200), (V_S, V_A, V_R):=0}
S2* = {F:=P, output(PH_DATA_Req(UA), MDL_ERROR_Ind(F), DL_ESTAB_Ind), STOP(T200), (V_S, V_A, V_R):=0}

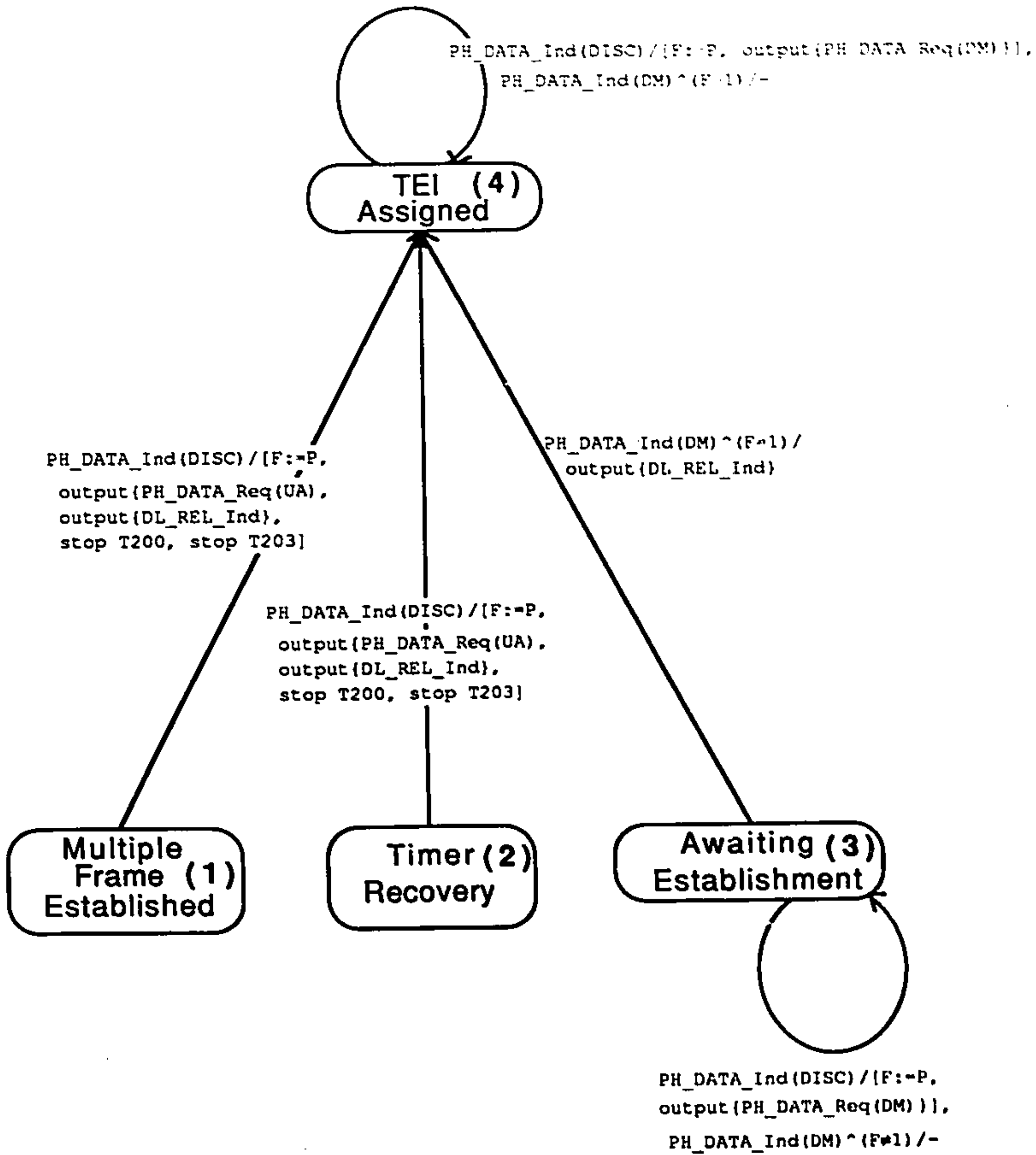
Start States: {1, 2, 3}
Goal State: {3}

Figure A1.7: Phase: Link Re-establishment



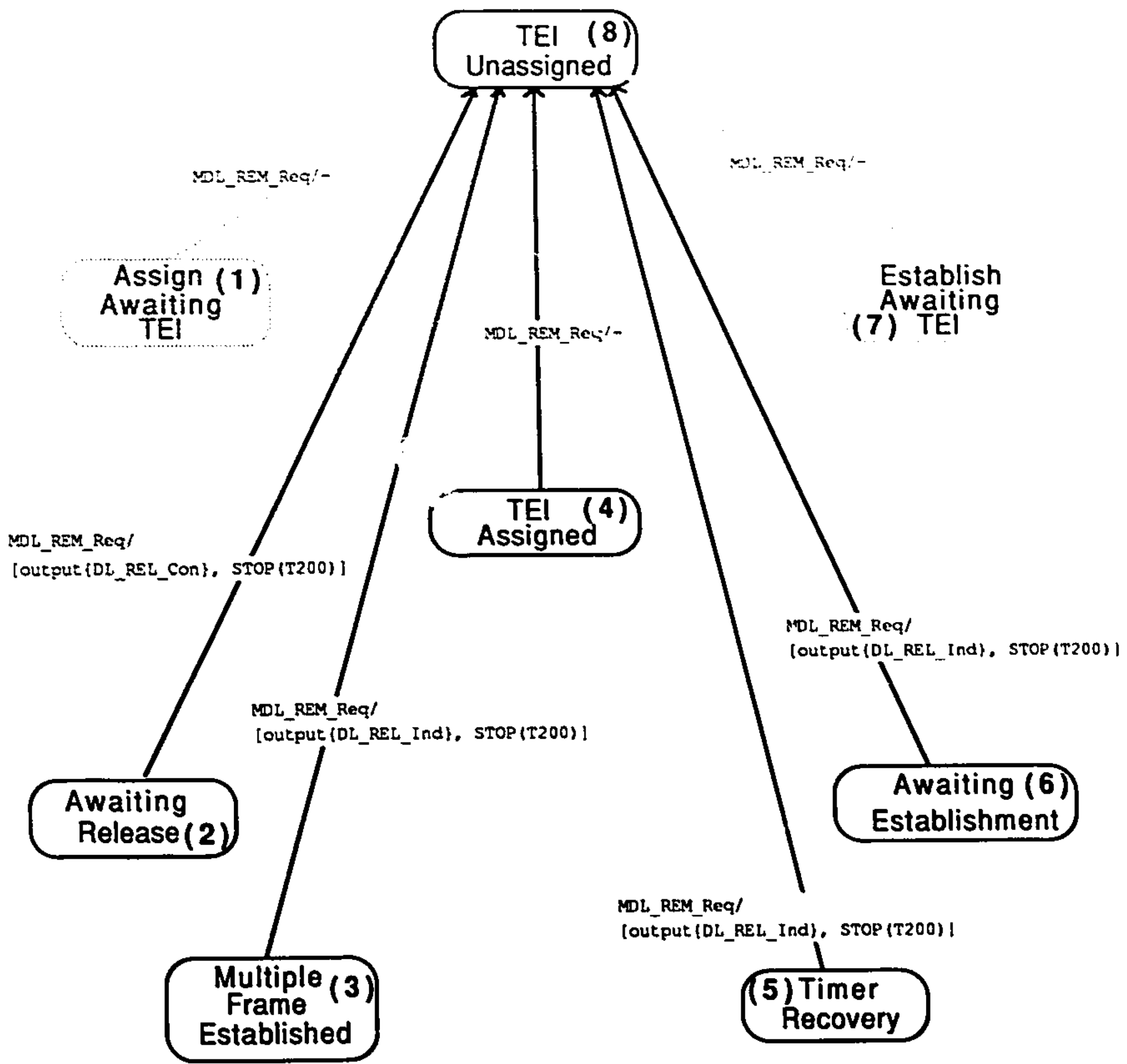
Start States: {1, 2}
Goal State: {4}

Figure A1.8: Phase: L3 Initiated Link Release



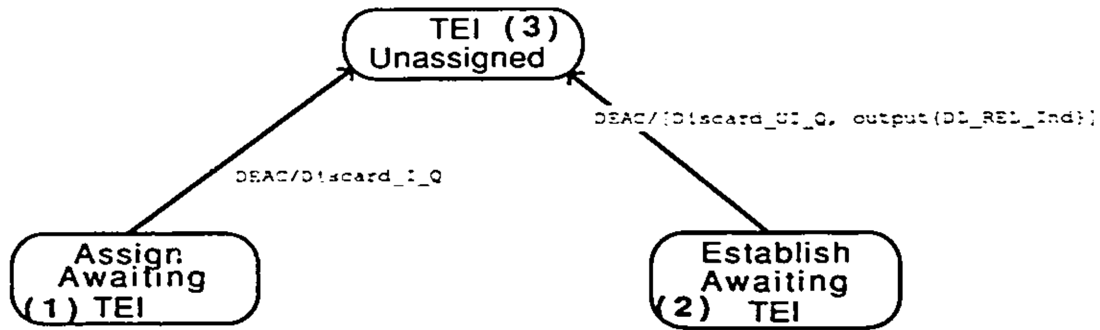
Start States: {1, 2, 3}
Goal State: {4}

Figure A1.9 :Phase: Peer Initiated Link Release



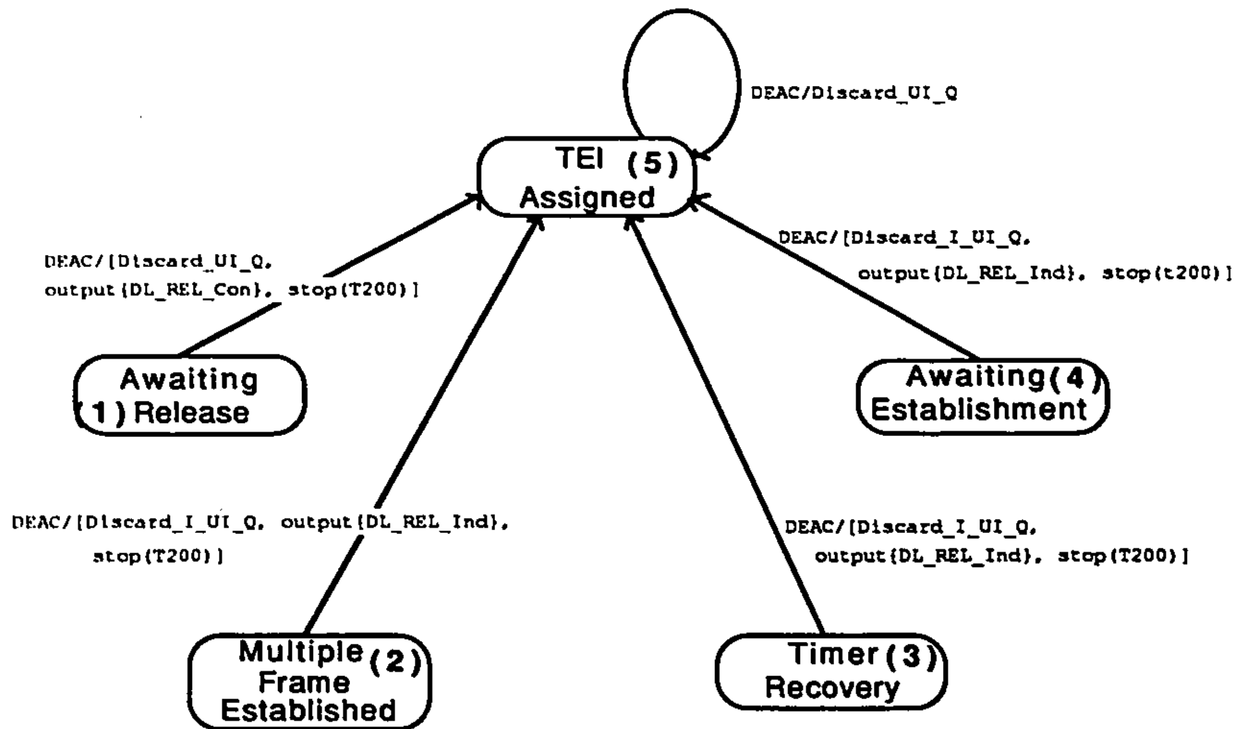
Start States: {1, 2, 3, 4, 5, 6, 7}
Goal State: {8}

Figure A1.10: Phase: TEI Removal



Start States: {1, 2}
Goal State: {3}

Figure A1.11: Phase: Deactivation/TEI unassigned



Start States: {1, 2, 3, 4}
Goal State: {5}

Figure A1.12: Phase: Deactivation/TEI assigned

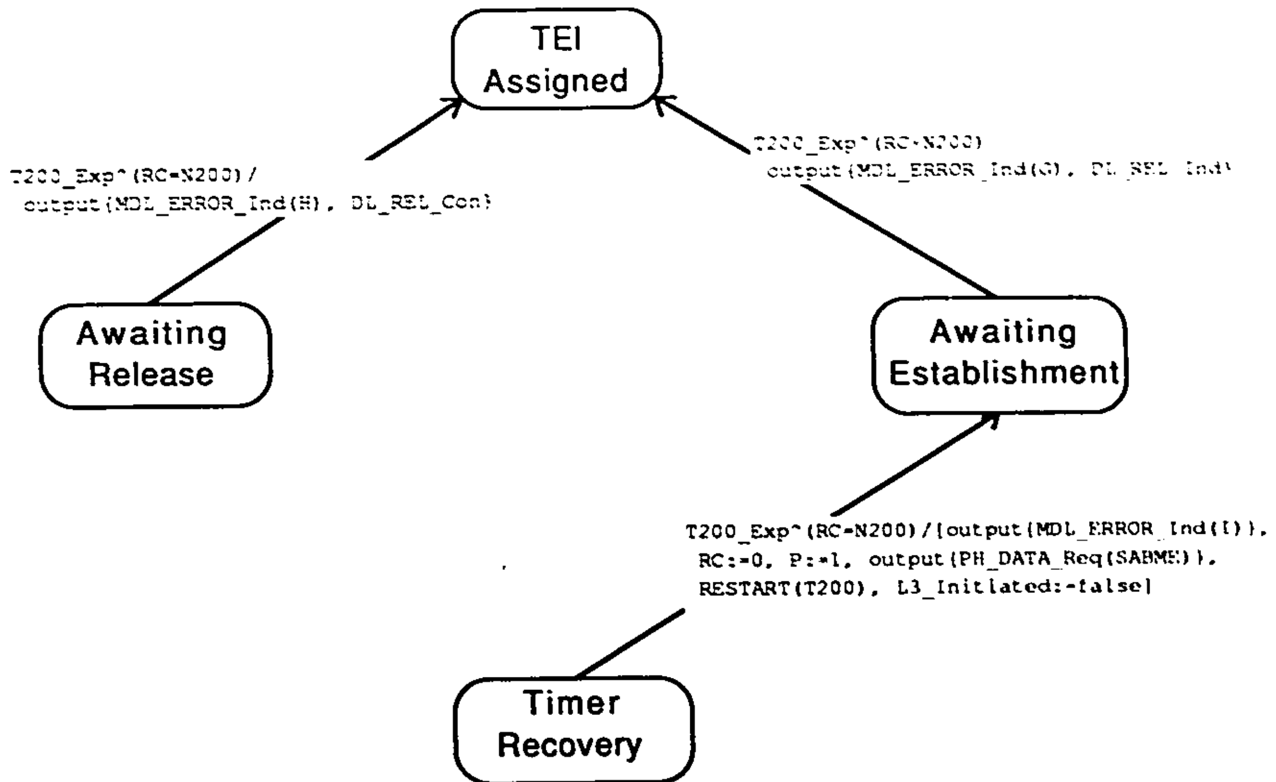


Figure A1.13: Phase: T200 Timeout Recovery

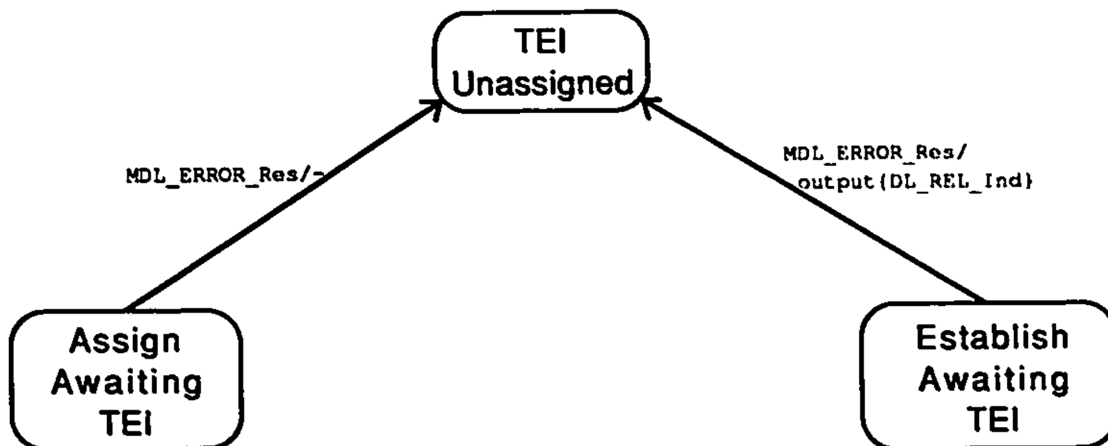
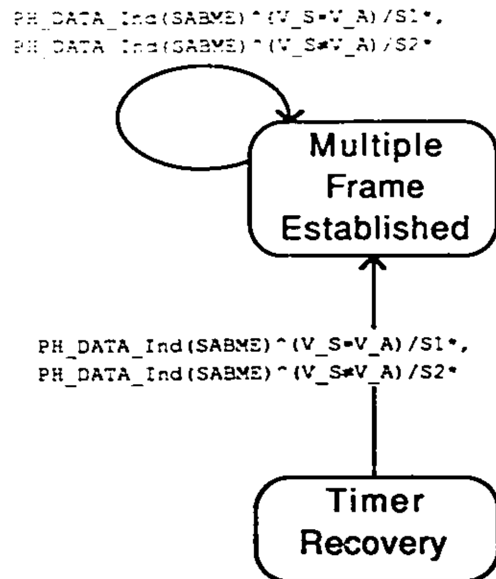
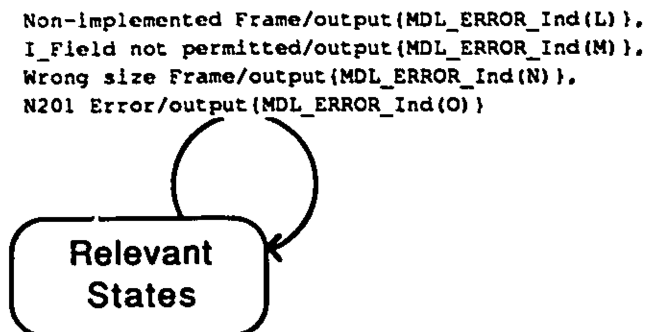


Figure A1.14: Phase: TEI Assignment Error Recovery



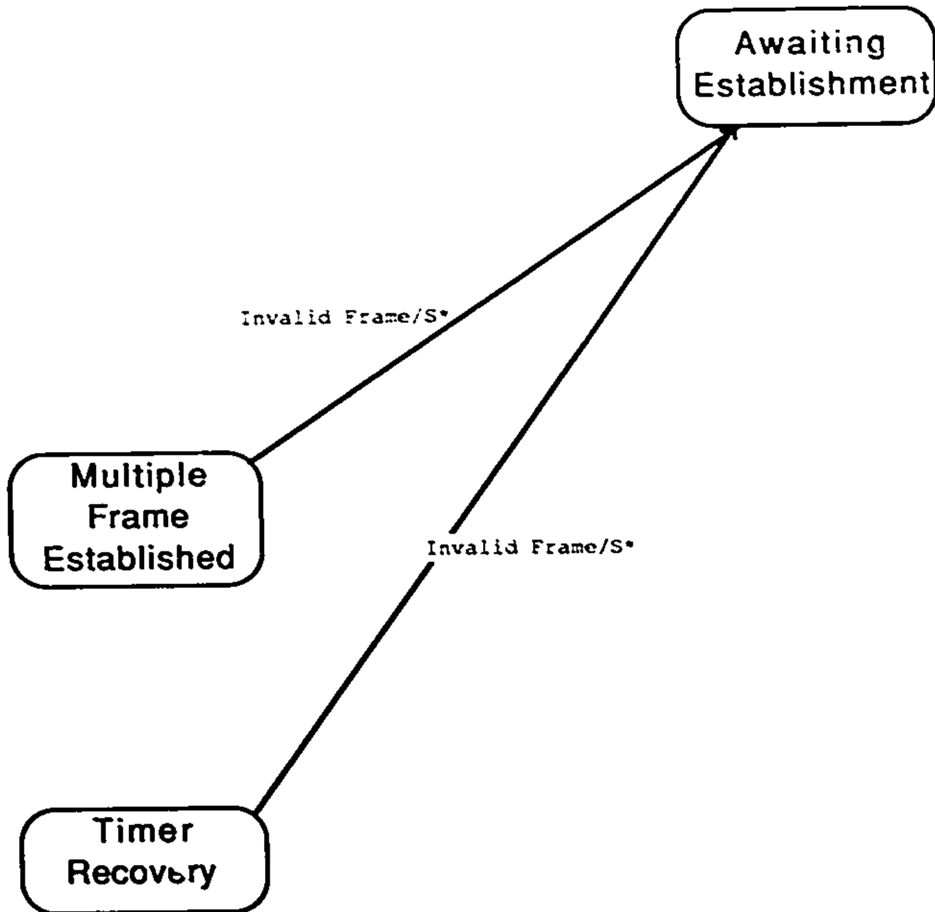
S1* = (F:=P, output(PH_DATA_Req(UA), MDL_ERROR_Ind(F)), STOP(T200), (V_S, V_A, V_R):=0)
S2* = (F:=P, output(PH_DATA_Req(UA), MDL_ERROR_Ind(F), DL_ESTAB_Ind), STOP(T200), (V_S, V_A, V_R):=0)

Figure A1.15: Phase: Attempted Peer Initiated Re-establishment Error



Relevant States: TEI Assigned, Awaiting Establishment, Awaiting Release.

Figure A1.16: Phase: Invalid Frame Error Recovery/1



S* = {output(MDL_ERROR_Ind(L/M/N/O)), RC:=0, P:=1, output(PH_DATA_Req(SABME)), RESTART(T200), L3_initiated:=false}

Figure A1.17 Phase: Invalid Frame Error Recovery/2

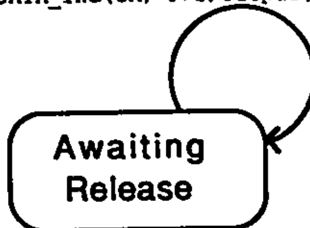


Figure A1.18 Phase: Unsolicited Response Error Recovery/1

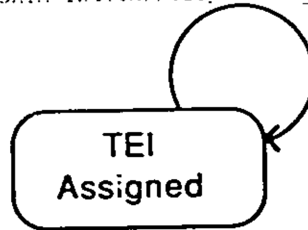
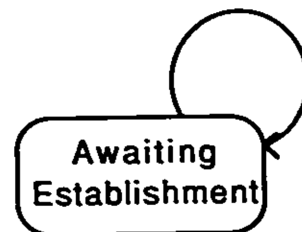
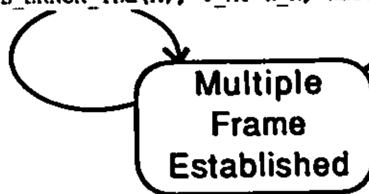


Figure A1.19 Phase: Unsolicited Response Error Recovery/2





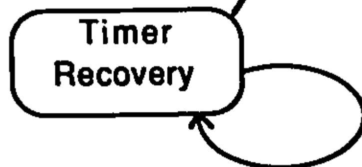
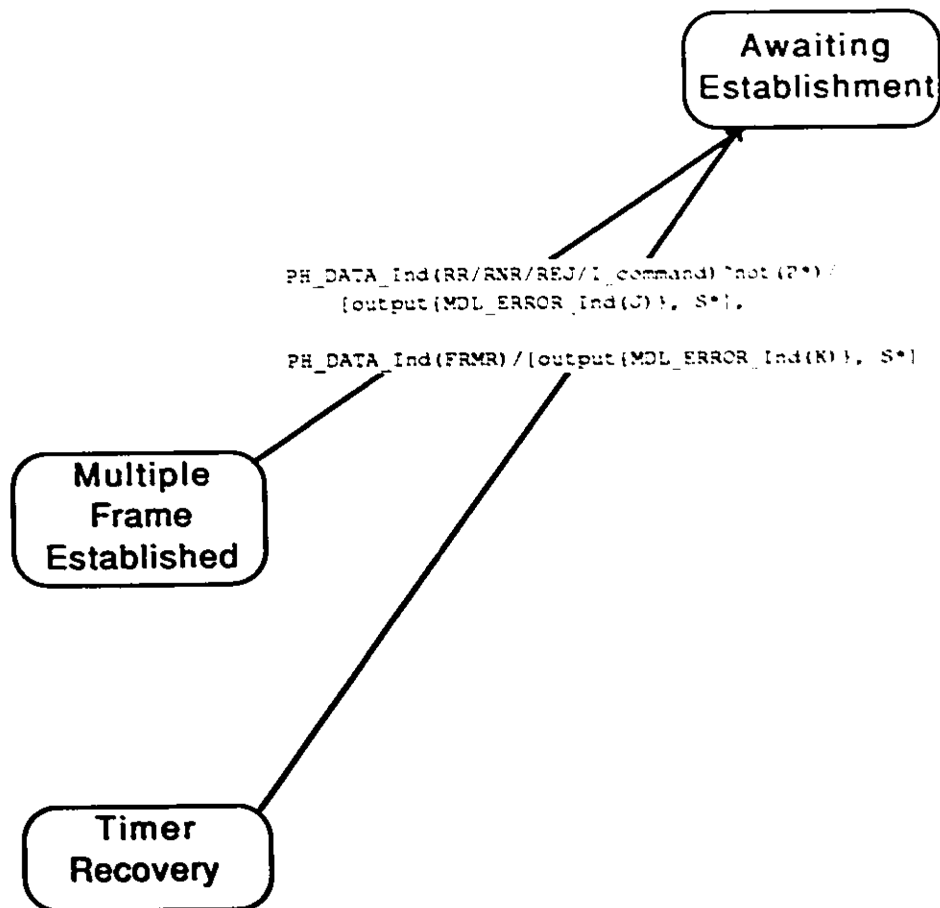


Figure A1.20 Phase: Unsolicited Response Error Recovery/3



S* = {RC:=0, P:=1, output(PH_DATA_Req(SABME)),RESTART(T200)}
P* = (V_A≤N_R≤V_S)

Figure A1.21 Phase: Sequence Error and FRMR Recovery

Appendix 2

Presented in this section is an incompletely specified TTCN test suite, i.e., this specification does not include all of the structures that a TTCN specification requires. Only the most basic information required for the application in this thesis is presented.

A2.1 Test Suite Overview

Suite Overview	
Suite Name:	LAPD
Reference to Standards:	TTCN based on ISO DP9646–3, July 1988. Test cases derived from CCIT Recommendation Q.921, May 1986.
Reference to PICS:	
Reference to PIXIT:	
How Used:	
Test Method:	The Distributed Single (DS) test method is assumed.
Comments:	The upper tester plays the role of both the Management entities and layer 3 entity. Thus pco1 defined below represents SAP's to all of these entities.The lower tester provides the services normally provided by the physical layer; pco2 is defined to represent the SAP at the boundary between this layer and layer 2.

A2.2 PCO and Timer Declarations

PCO Declaration	
PCO	**Role**
pco1	SAP at upper tester. L2-L3 and L2-Management communication
pco2	SAP at lower tester. L2-L1 communicaton

Timer Declaration		
Timer Type Name	**Duration (sec)**	**Comments**
T200	0 .. 2.5	Retransmission of data
T202	0 .. 2	Retransmission for TEI assignment request

A2.3 Constant and Variable Declarations

Global Constants			
Name	**Type**	**Value**	**Comments**
N202	integer	3	Max number of retries for TEI Assignment

Global Variables			
Name	**Type**	**Value**	**Comments**
RC	integer	0	Retransmission Counter
Able_to_Establish	boolean	true	System state—abbility to establish data link connection
Peer_Receiver_Busy	boolean		
Own_Receiver_Busy	boolean		
Reject_exception	boolean		
Ack_pending	boolean		

A2.4 Abbreviation Declarations

Abbreviation Declarations		
Abbreviation	**Expansion**	**Comments**
M_U_DR	MDL_UNIT_DATA_Req	
P_U_DR	PH_UNIT_DATA_Req	
P_DI	PH_DATA_Ind	
P_DR	PH_DATA_Req	
A_E	Able_to_Establish	
P_R_B	Peer_Receiver_Busy	
O_R_B	Own_Receiver_Busy	
R_E	Reject_exception	
A_P	Ack_pending	

A2.5 Declaration of ASPs at pco1

ASP Declaration		
ASP: MDL_UNIT_DATA_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
messsage_type	(IR, IA, ID)	
Ri	integer[0..65535]	
Ai	integer[0..127]	Value of 127 asks peer to assign any available TEI value

ASP Declaration		
ASP: MDL_UNIT_DATA_Ind	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
messsage_type	(IR, IA, ID)	IA or ID
Ri	integer[0..65535]	
Ai	integer[0..127]	Value of TEI assigned

ASP Declaration		
ASP: MDL_ASS_Req	PCO: pco1	CEId:
Service Control Information		
Parameter Name	Type	Comments
TEI_Value	integer[0 .. 127]	

ASP Declaration		
ASP: MDL_ASS_Ind	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	Type	Comments
CES_Value	integer	

ASP Declaration		
ASP: MDL_REM_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	Type	Comments
TEI_Value	integer[0 .. 127]	

ASP Declaration		
ASP: MDL_ERROR_Res	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	Type	Comments
error	generalstring	

ASP Declaration		
ASP: DL_UNIT_DATA_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	Type	Comments
data	octetstring	user data

ASP Declaration		
ASP: DL_DATA_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
data	octetstring	user data; acknowledged

ASP Declaration		
ASP: DL_DATA_Ind	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
data	octetstring	peer data; acknoweledged

ASP Declaration		
ASP: DL_ESTAB_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
address	octetstring	

ASP Declaration		
ASP: DL_REL_Req	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
address	octetstring	

ASP Declaration		
ASP: DL_REL_Ind	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	**Type**	**Comments**
address	octetstring	

ASP Declaration		
ASP: DL_REL_Con	PCO: pco1	CEId: used
Service Control Information		
Parameter Name	Type	Comments
address	octetstring	

A2.6 Declaration of ASP"s at pco2

ASP Declaration		
ASP: PH_UNIT_DATA_Req	PCO: pco2	CEId: used
Service Control Information		
Parameter Name	Type	Comments
L2_unit_data	octetstring	L3 or Management

ASP Declaration		
ASP: PH_UNIT_DATA_Ind	PCO: pco2	CEId: used
Service Control Information		
Parameter Name	Type	Comments
L2_unit_data	octetstring	peer data

ASP Declaration		
ASP: PH_DATA Req	PCO: pco2	CEId:
Service Control Information		
Parameter Name	Type	Comments
L2_ack_data	octetstring	user data and I_commands

ASP Declaration		
ASP: PH_DATA_Ind	**PCO:** pco2	**CEId:** used
Service Control Information		
Parameter Name	**Type**	**Comments**
L2_ack_data	octetstring	peer data

A2.7 Dynamic Behaviour Descriptions

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/reset **Identifier:** r1 **Purpose:** Reset IUT to idle state **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
RESET pco1 ! MDL_REM_Req + TEI_UNASS_STATE_CHK				reset IUT to TEI Unassigned state
Extended Comments: Test case passes if TEI_UNASS_STATE_CHK passes				

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/VarifyState/TEI_Unassigned				
Identifier: Chk01				
Purpose: Verify that IUT is in TEI Unassigned state				
Defaults Reference:				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_UNASS_STATE_CHK /* TEI_Unassigned check sequence */				UIO or DS for TEI Unassigned state
Extended Comments: pass if in TEI Unassigned state, fail otherwise				

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/VarifyState/TEI_Assigned				
Identifier: Chk02				
Purpose: Verify that IUT is in TEI Assigned state				
Defaults Reference:				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_ASS_STATE_CHK /* TEI_Assigned check sequence */				UIO or DS for TEI Assigned state
Extended Comments: pass if in TEI Assigned state, fail otherwise				

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/TEI-Assign/Direct **Identifier:** t1 **Purpose:** Check direct TEI assignment of non-automatic equipment **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_ASS1				
+ RESET				
pco1 ! MDL_ASS_Req				store TEI value
+ TEI_ASS_STATE_CHK				
Extended Comments: Test case passes if TEI_ASS_STATE_CHK passes				

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/TEI-Assign/Direct-automatic **Identifier:** t2 **Purpose:** To check the assignment of TEI values to automatic user equipment **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_ASS2				
+ RESET				
pco1 ! M_U_DR (START T202)	L1	M_U_DR[c1]		
pco2 ? PH_UNIT_DATA_Req				
pco2 ! PH_UNIT_DATA_Ind				
pco1 ? M_U_DI		M_U_DI[c1]		
pco1 ! MDL_ASS_Req (CANCEL T202)			pass	
? (TIMEOUT T202) [RC≤N202] (RC:=RC+1)				
# goto L1				
? (TIMEOUT T202)[RC>N202]				
pco1 ! MDL_ERROR_Res			fail	

Dynamic Behaviour

Reference: LAPD/BasicInterconnection/TEI-Assign/ViaDataTrans
Identifier: t3
Purpose: Check TEI assignment as a result of request to transfer data
Defaults Reference:

Behaviour Description	Label	Constraints Reference	Verdict	Comments
TEI_ASS3				
+ RESET				
pco1 ! DL_UNIT_DATA_Req				
pco1 ? MDL_ASS_Ind				
pco1 ! MDL_ASS_Req				store TEI value
+ TEI_ASS_STATE_CHK				
pco1 ? OTHERWISE			inconc	

Extended Comments: Test case passes if TEI_ASS_STATE_CHK passes

Dynamic Behaviour

Reference: LAPD/BasicInterconnection/TEI_Assign/ViaDataTrans
Identifier: t4
Purpose: Check TEI assignment as a result of request to transfer data
Defaults Reference:

Behaviour Description	Label	Constraints Reference	Verdict	Comments
TEI_ASS4				
+ RESET				
pco1 ! DL_ESTAB_Req				
pco1 ? MDL_ASS_Ind				
pco1 ! MDL_ASS_Req				store TEI value
pco2 ? P_DR (RC:=0)		P_DR[c1]		
# (RESUME T200)				
# (L3_Initiated:=true)			pass	
pco2 ? OTHERWISE			fail	
pco1 ? OTHERWISE			fail	

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/Link-access **Identifier:** t5 **Purpose:** To check direct establishment of communication link **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
LINK_ACCESS				
/* To TEI_Assigned state */				Test starts
pco2 ! P_DI	L1	P_DI[c1]		in TEI_As-
pco2 ? P_DR[A_E=false] goto L1		P_DR[c2]		signed
pco2 ! P_DI		P_DI[c1]		state.
pco2 ? P_DR[A_E=true]		P_DR[c3]		Lower tester.
# (P_R_B:=false, O_R_B:=false)				
# (R_E:=false, A_P:=false)				
# (V_S:= 0, V_A:= 0, V_R:= 0)				
pco1 ? DL_ESTAB_Ind (CANCELT200)			pass	Upper tester.
pco1 ? OTHERWISE			fail	

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/DataTrans/SingleFrame **Identifier:** t6 **Purpose:** To transmit a single frame and receive its acknowledgement **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
DATA_TRANS				
/* To Multiple Frame Established state */				
pco1 ! DL_DATA_Req[(P_R_B=false) AND (V_S≠V_A+k)]				
pco2 ? P_DR (V_S := V_S + 1) (A_P:=false)		P_DR[c4]		data sent
# (START T200)[T200 ≠ running]				
pco2 ! P_DR [O_R_B=false] (V_R := V_R + 1, R_E:=false)		P_DI[c2]		ack received
pco1 ? DL_DATA_Ind				
(CANCEL T200)[(V_A≤N_R≤V_S) AND (P_R_B=false)				
# AND (N_R=V_S)]			pass	
(RESUME T200)[(V_A≤N_R≤V_S) AND (P_R_B=false) AND				
# (N_R≠V_S) AND (N_R≠V_A)] (V_A := V_R)				
? (TIMEOUT T200)			inconc	

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/LinkRelease/PeerInitiated **Identifier:** t7 **Purpose:** To test peer initiated release of link **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
LINK_RELEASE1				
/* To MFE state */				
pco2 ! P_DI		P_DI[c3]		
pco2 ? P_DR		P_DR[c3]		
pco1 ? DL_REL_Ind			pass	
pco1 ? OTHERWISE			fail	

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/LinkRelease/L3Initiated **Identifier:** t8 **Purpose:** To test link release intiated by layer 3 **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
LINK_RELEASE2				
/* To MFE state */				
pco1 ! DL_REL_Req (RC:=0)				
pco2 ? P_DR		P_DR[c5]		
pco2 ! P_DI		P_DI[c4]		
pco1 ? DL_REL_Con			pass	
pco1 ? OTHERWISE			fail	

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/TEI_Rem/FromTEI_Ass **Identifier:** t9 **Purpose:** To remove TEI from TEI Assigned state **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_REM1 /* To TEI_Assigned */ pco1 ! MDL_REM_Req + TET_UNASS_STATE_CHK				Test starts in TEI Assigned state. Discard TEI
Extended Comments: Test case passes if TEI_UNASS_STATE_CHK passes				

Dynamic Behaviour				
Reference: LAPD/BasicInterconnection/TEI_Rem/FromMulFrmEst **Identifier:** t10 **Purpose:** To verify TEI removal from Multiple Frame Established state **Defaults Reference:**				
Behaviour Description	**Label**	**Constraints Reference**	**Verdict**	**Comments**
TEI_REM2 /* To MFE state */ pco1 ! MDL_REM_Req pco1 ? DL_REL_Ind pco1 ? OTHERWISE			 pass fail	Test starts in Multiple frame Established state

A2.8 Constraints Part

ASP Constraint		
ASP Name: MDL_UNIT_DATA_Req	**Constraint Name: c1**	
Field Name	**Value**	**Comments**
message_type	IR	Request TEI assignment
Ri	?	
Ai	?	Value of 127 requests any available TEI

ASP Constraint		
ASP Name: MDL_UNIT_DATA_Ind	**Constraint Name: c1**	
Field Name	**Value**	**Comments**
message_type	IA	
Ri	?	
Ai	0..126	Ai holds assigned TEI value

ASP Constraints List					
ASP Name: PH_DATA_Req					
Constraint Name	**Field Name**				**Comments**
	control_info	P_F	N_S	data_info	
c1	SABME	1	-	?	
c2	DM	P	-	?	
c3	UA	P	-	?	
c4	I_command	0	V_S	?	
c5	DISC	1	-	?	

ASP Constraints List					
ASP Name: PH_DATA_Ind					
Constraint Name	**Field Name**				**Comments**
	control_info	P_F	N_S	data_info	
c1	SABME	?	-	?	
c2	I_command	?	V_R	?	
c3	DISC	?	-	?	
c4	UA	1	-	?	

Appendix 3

A3.1 FORMAL SPECIFICATION OF TEI ASSIGNMENT PHASE OF LAPD

```
specification LAPD_TEI_Assignment;

  (* Estelle specification of the TEI Assignment phase of LAPD. This specification is
     based on ISO/DP 9074, 1986/10/15 and CCITT Q.921 *)

  const
    N201 = 260; (* default *)
    flag = 126;

    (* The following constants have been derived from Table 3/Q.921 with the
       assumption that the P/F bit is set to zero. *)

    SABME = 111;
    DM = 15;
    DISC= 67;
    UA = 99;

  type
    (* The following 'Error_Type' codes are taken from Table II-I/Q.921 *)
    Error_Type = 'A' .. 'O';

    bit_val_type = ( 0, 1 );
    ctrl_type = ( SABME, DM, DISC, UA );
    info_type = packed_array [ 0 .. N201 ] of char;
    TEI_type = 0 .. 127;
    data_type
      record
        Start_flag : integer;
        Address : integer;
```

```
        Control : ctrl_type;
        FCS : integer;
        End_flag : integer;
      end;

var
  TEI_Register, Own_TEI_Val : TEI_type;
  CES_Val : integer;
  L3_Initiated, Able_to_Establish : boolean;
  Final_Bit_Val, Poll_Bit_Val : bit_val_type;
  V_Send, V_Ack, V_Rec : integer;
  error : Error_Type;

(* Channel Definitions *)

channel SAP_to_Layer_Man ( Manage_User, Manage_Provider );
  by Manage_User:
     MDL_ASSSIGN_Indication ( CES_Val );

  by Manage_Provider:
     MDL_ASSIGN_Request ( Own_TEI_Val );
     MDL_REMOVE_Request ( Own_TEI_Val );


channel SAP_to_Connect_Man ( Manage_User, Manage_Provider );
  by Manage_User:
     MDL_ERROR_Indication ( error );

  by Manage_Provider:
     MDL_ERROR_Response ( error );


channel DL_SAP_to_L3 ( DL_User, DL_Provider );
  by DL_User:
     DL_ESTABLISH_Request;
```

```
    DL_RELEASE_Request;

  by DL_Provider:
    DL_ESTABLISH_Indication;
    DL_ESTABLISH_Confirm;
    DL_RELEASE_Indication;
    DL_RELEASE_Confirm;


channel PH_SAP_to_DL ( PH_User, PH_Provider );
  by PH_User:
    PH_DATA_Request ( User_data );

  by PH_Provider:
    PH_DATA_Indication ( Net_data )


(* Module_type Definitions *)

module Layer_Management_Type process
  ip
    s1 : SAP_to_Layer_Man ( Manage_Entity ) common queue
end;

module Connect_Management_Type process
  ip
    s2 : SAP_to_Connect_Man ( Manage_Entity ) common queue
end;

module L3_Entity_Type process
  ip
    s3 : DL_SAP_to_L3 ( DL_User ) common queue
end;

module Physical_Connect_Type process
```

```
 ip
   s4 : PH_SAP_to_DL ( PH_Provider ) common queue
end;

module DL_Entity_Type process
  (* parameter passed to module at instantiation *)
  ( Peer_TEI_Val : TEI_type );

  ip
    s1 : SAP_to_Layer_Man ( DL_Entity ) common queue;
    s2 : SAP_to_Connect_Man ( DL_Entity ) common queue;
    s3 : DL_SAP_to_L3 ( DL_Provider ) common queue;
    s4 : PH_SAP_to_DL ( PH_User ) common queue;
end;

(* Body Definitions *)

body Layer_Management_body for Layer_Management_Type;
  external;

body Connect_Management_body for Connect_Management_Type;
  external;

body Physical_Layer_body for Physical_Connect_Type;
  external;

body L3_Entity_body for L3_Entity_Type;
  external;

body DL_Entity_body for DL_Entity_Type;
  const
    N200 = 3;
    null = '';
    T200 = ...;
```

```
type
  Timer_type   = T200;
  Unit_Data_type = ...;

var
  Peer_Receiver_Busy, Own_Receiver_Busy : boolean;
  Ack_Pending, Reject_Exception : boolean;
  Retrans_Count : integer;
  Timer_T200_Expired : boolean;
  User_Data, Net_Data : data_type;

state
  TEI_UNASSIGNED, ESTABLISH_AWAITING_TEI, TEI_ASSIGNED,
  ASSIGN_AWAITING_TEI, AWAITING_ESTABLISHMENT

(* The following procedures stop, start, and restart the specified timer. They
are     provided by the implementor. A call to START a particular timer will first
result      in that timer being cancelled (reset to zero). The variable
'Timer_T200_Expired'            is set true when timer T200 expires. *)

procedure START_Timer ( Timer : Timer_type );
  primitive;

procedure STOP_Timer ( Timer : Timer_type );
  primitive;

procedure RESTART_Timer ( Timer : Timer_type );
  primitive;

function ADRRESS_Value ( ctrl_field : ctrl_type ) : integer;
  (* This function uses the value of the control field of the frame to determine
the        SAPI, whether to use Own_TEI_Val or Peer_TEI_Val, (see pg. 7 of
[CCIT2]),
  and to set the C/R (see pg. 6 of [CCIT2]) bit appropriately *)
```

```
  primitive;

procedure set_PFbit ( ctrl_var : ctrl_type;
                      bit_val : bit_val_type );
  (* This procedure sets the P/F bit to the value specified in bit_val. It is provided
     by the implementor. *)
  primitive;

function FCS_Value ( User_Data : data_type ) : integer;
  (* This function calculates the frame check sequence (FCS) of the frame. It is
     provided by the implementor *)
  primitive;

procedure format_frame ( ctrl_var : ctrl_type;
                         bit_val : bit_val_type );
  (* This procedure assembles the data_link frame to be sent to the physical
  layer *)
  begin
    User_Data.Start_flag := flag;
    User_Data.Address := ADDRESS_Value ( ctrl_var );
    User_Data.Information := null;
    User_Data.Control := ctrl_var;
    set_PFbit ( ctrl_var, bit_val );
    User_Data.FCS := FCS_Value ( User_Data )
    User_Data.End_flag := flag
  end; (* of format_frame *)

procedure store ( var TEI_Register : integer; Own_TEI_Val : integer );
  (* This procedure stores the TEI value obtained from the layer management
entity        in the TEI register. *)
  primitive;

procedure Add_UI_Q ( Unit_Data : Unit_Data_type );
  (* This procedure adds the Unit Data received to the rear of the UI queue. The
     procedure is provided by the implementor. *)
```

```
  primitive;

procedure Discard_I_Queue;
  (* This procedure empties (conceptually) the I_queue *)
  primitive;

procedure Clear_Exception_Conditions;
  begin
    Peer_Receiver_Busy := false;
    Reject_Exception := false;
    Own_Receiver_Busy := false;
    Ack_Pending := false;
  end; (* Clear_Exception_Conditions *)

procedure Establish_Data_Link;
  begin
    Clear_Exception_Conditions;
    Retrans_Count := 0;
    Poll_Bit_Val := 1;
    format_frame ( SABME, Poll_Bit_Val );
    output s4.PH_DATA_Request ( User_Data );
    RESTART_Timer ( T200 );
  end; (* Establish_Data_Link *)


(* Initialize starting state and variables for DL_Entity *)

initialize
  to TEI_UNASSIGNED

  begin
    Retrans_Count := 0;
  end; (* of initialization for this module *)
```

```
(* Transitions for the Data_Link entity *)

from TEI_UNASSIGNED
to ASSIGN_AWAITING_TEI
when s3.DL_UNIT_DATA_Request ( Unit_Data )
  begin
    output s1.MDL_ASSIGN_Indication ( CES_Val );
    Add_UI_Q ( Unit_Data );
  end;

from TEI_UNASSIGNED
to ESTABLISH_AWAITING_TEI
when s3.DL_ESTABLISH_Request
  begin
    output s1.MDL_ASSIGN_Indication ( CES_Val );
  end;

from TEI_UNASSIGNED
to TEI_ASSIGNED
when s1.MDL_ASSIGN_Request ( Own_TEI_Val )
  begin
    store ( TEI_Register, Own_TEI_Val );
  end;

from ASSIGN_AWAITING_TEI
to ESTABLISH_AWAITING_TEI
when s3.DL_ESTABLISH_Request
  begin     end;

from ASSIGN_AWAITING_TEI
to ASSIGN_AWAITING_TEI
when s3.DL_UNIT_DATA_Request ( Unit_Data )
  begin
```

```
      Add_UI_Q ( Unit_Data );
    end;

from ASSIGN_AWAITING_TEI
to TEI_ASSIGNED
when s1.MDL_ASSIGN_Request ( Own_TEI_Val )
    begin
      store ( TEI_Register, Own_TEI_Val );
    end;

from ESTABLISH_AWAITING_TEI
to AWAITING_ESTABLISHMENT
when s1.MDL_ASSIGN_Request ( Own_TEI_Val )
    begin
      store ( TEI_Register, Own_TEI_Val );
      Establish_Data_Link;
      L3_Initiated := true;
    end;

end; (* of DL_Entity_body *)


(* module variable declaration *)

modvar
    Layer_Man_Entity : Layer_Management_Type;
    Connect_Man_Entity : Connect_Management_Type;
    L3_Entity : L3_Entity_Type;
    DL_Entity : DL_Entity_Type;
    Physical_Entity : Physical_Connect_Type;


(* Initialization section for the system *)

initialize
```

```
begin
  (* module instance creation (module instantiation) *)
  init Layer_Man_Entity with Layer_Management_body;
  init Connect_Man_Entity with Connect_Management_body;
  init L3_Entity with L3_Entity_body;
  init DL_Entity with DL_Entity_body;
  init Physical_Entity with Physical_Layer_body;

  (* Bind interaction points to create channels *)
  connect Layer_Man_Entity.s1 to DL_Entity.s1;
  connect Connect_Man_Entity.s2 to DL_Entity.s2;
  connect L3_Entity.s3 to DL_Entity.s3;
  connect DL_Entity.s4 to Physical_Entity.s4;

 end; (* of initialization section *)

end. (* of specification *)
```