National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

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

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

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

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

AVIS

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

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

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

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

Canada

Verification of Test Cases for Protocol Conformance Testing

Kshirasagar Naik

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirement

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

May, 1992

© Kshirasagar Naik, 1992

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

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 0-315-81014-9

Canada

ABSTRACT

Verification of Test Cases for Protocol Conformance Testing

Kshirasagar Naik, Ph. D.,

Concordia University

This thesis is concerned with verifying the correctness of human designed test cases for determining the conformance of a protocol implementation with its formal specification. Correctness of a test case, like any other software systems, is established by verifying the *safety* and *liveness* properties of the test case against the *model* of a *test verification system.*

A test verification system is derived from the architectural basis of the test case by representing the Lower Tester as an extended finite state machine (EFSM), substituting the implementation under test by the EFSM model of the corresponding protocol specification, representing the Upper Tester as an EFSM model derived either from the test case or from the test management protocol depending on the test architecture, modeling the input/output service specification of the underlying service provider as an EFSM, and representing each interaction point between two EFSMs in the test architecture by two FIFO channels. Algorithms are presented to derive EFSMs from the Estelle specifications of protocols and Tree and Tabular Combined Notation (TTCN) descriptions of test cases.

The *model* of a test verification system represents the global state space of the verification system with a set of atomic predicates associated with each global state, such that a global state's atomic predicates evaluate to true in that state. An algorithm, based on the well known reachability analysis technique, is presented to derive the global state space of a test verification system.

Four types of test case safety properties are defined: transmission safety, reception safety, synchronization safety, and verdict safety. The first two types of safety properties are common to all communication systems and the latter two are specific

to protocol test systems. The liveness property of a test case is based on the idea that the test case behavior leading to a *Pass* test verdict fulfills the test *purpose.* We have proposed a set of notations to represent the safety and liveness properties as formulas in branching time temporal logic.

Test case properties are verified on the model of a test verification system by using a model checking algorithm. A methodology is also discussed to verify multiplexing test cases used in a multi-party test environment.

The verification methodology is applied to three test cases: one Remote Single-layer (RS) architecture based test case for an Association Control Service Element (ACSE) protocol, one Coordinated Single-layer (CS) architecture based single connection test case for a Class 2 transport protocol, and a CS architecture based multiplexing test case for the same Class 2 transport protocol.

ACKNOWLEDGEMENTS

I take this opportunity to thank many people and organizations for their direct and indirect technical, financial, logistic, and moral supports.

First of all, I express my deepest gratitude to my thesis supervisor Dr. Behcet Sarikaya for his valuable guidance, widespread encouragement, and instant willingness to spend time on my research work. Dr. Sarikaya's highly enthusiastic and thought provoking technical suggestions were a prominent factor in bringing this research to its present shape in a quite reasonable amount of time.

I am thankful to Prof. Phoivos Ziogas, the late Chairman of our Department, for his moral and logistic supports at some of the critical moments in the course of my work.

My most sincere thanks are due to the Canadian Commonwealth Fellowship Agency for providing me all kinds of financial supports for five years. I also gratefully acknowledge a partial financial support from the Natural Sciences and Engineering Research Council of Canada.

I sincerely thank Anindya, Basudeb, Deb, Gurinder, Kostas, Piyu, Rajveer, Ramesh Ahooja, Shesh, and many others for their innumerable supports.

I specially thank Sylvie for inspiring and helping me in many ways, in addition to taking the pain of thoroughly reading a part of this thesis.

Last but not the least, I express my whole hearted indebtedness to my mother, elder brother and sister, and my late father whose constant inspirations during my childhood guided me towards being educated.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS x

LIST OF FIGURES xii

CHAPTER 1: INTRODUCTION 1

1.1 Protocol Engineering Life-cycle 1

1.2 Conformance Testing and Test Case Verification 4

1.2.1 Conformance Testing 4

1.2.2 Test Case Verification 5

1.3 Previous Work on Test Case Verification 6

1.4 Model Checking 7

1.5 Original Contributions of this Thesis 9

1.6 Organization of the Thesis 10

CHAPTER 2: EXTENDED FINITE STATE MACHINE MODELS 14

2.1 Extended Finite State Machine 15

2.2 EFSM Model of an Estelle Specification 16

2.2.1 Estelle Specification 16

2.2.2 Translation of an Estelle Specification to an EFSM 27

2.3 EFSM Model of a TTCN Test Case 41

2.3.1 TTCN Specification 41

2.3.1.1 Overview 41

2.3.1.2 Declarations 42

2.3.1.3 Constraints 42

2.3.1.4 Dynamic Behavior 56

2.3.2 Translation of a TTCN Test Case to an EFSM 61

2.4 Input/Output Diagrams 67

2.4.1 I/O Diagram Representation of Events in ASN.1 69

2.4.2 I/O Diagram Representation of Events in Estelle 71

2.4.3 I/O Diagram Representation of Events in TTCN 72

2.5 Complexity Analysis 74

2.5.1 Complexity for a TTCN Test Case 74

2.5.2 Complexity for an Estelle Specification 74

CHAPTER 3: TEST VERIFICATION SYSTEMS (TVS) 77

3.1 Test Architectures 78

3.1.1 Local Single-layer Test Architecture 79

3.1.2 Distributed Single-layer Test Architecture 80

3.1.3 Coordinated Single-layer Test Architecture 81

3.1.4 Remote Single-layer Test Architecture 82

3.1.5 Comparison of LS, DS, CS, and RS architectures 83

3.2 Test Verification System 84

3.3 Model Generation 88

3.3.1 Reachability Analysis Algorithm 88

3.3.1.1 Predicate Evaluation 91

3.3.1.2 Perturbation 92

3.3.2 Model Generation Algorithm 96

3.4 Complexity Analysis 99

CHAPTER 4: MODEL CHECKING FOR TVS 101

4.1 Temporal Logic 101

4.2 Safety Properties 104

4.3 Liveness Property 106

4.4 Verification of Test Case Properties 109

4.4.1 Evaluation of *Until* Operator 110

4.4.2 Evaluation of $\mapsto$ Operator 111

4.4.3 Evaluation of *SEQ* Operator 112

4.4.4 Model Checking Algorithm 113

4.4.5 Complexity Analysis 117

CHAPTER 5: RS TEST CASE VERIFICATION 117

5.1 ACSE Specification 118
5.2 Test Case 126
5.3 Service Provider 129
5.4 Global State Space 129
5.5 Verification of Test Case Properties 132
5.5.1 Safety Properties of the Test Case 132
5.5.2 Liveness Property of the Test Case 133
5.6 Generation of an Upper Tester 133
CHAPTER 6: CS TEST CASE VERIFICATION 135
6.1 Transport Protocol, TMP, and Service Provider 136
6.1.1 Class 2 Transport Protocol Specification 136
6.1.2 Service Provider 139
6.1.3 Test Management Protocol 140
6.2 Single Connection Test Case 144
6.2.1 EFSM Model of the Test Case 144
6.2.2 Model Generation 148
6.2.3 Verification of Safety and Liveness Properties 150
CHAPTER 7: VERIFICATION OF PARALLEL TEST CASES 154
7.1 Parallel Test Architecture 154
7.2 Parallel Test Case Verification 156
7.3 Example of Verifying a Parallel Test Case 159
7.3.1 EFSM Model of the Test Case 160
7.3.2 Test Verification System for the Parallel Test Case 165
7.3.3 Model Generation 166
7.3.4 Verification of Safety and Liveness Properties 167
CHAPTER 8: CONCLUSIONS AND FUTURE RESEARCH 173
8.1 Conclusions 173
8.2 Future Research 179

8.2.1 Implementation of the Verification System 179
8.2.2 Verifying Invalid Behavior Test Cases 180
8.2.3 Piecewise Test Verification 181
8.2.4 Test Generation Using Model Checking Approach 182
REFERENCES 183
APPENDIX 1 189
APPENDIX 2 192
APPENDIX 3 214
APPENDIX 4 226

LIST OF ABBREVIATIONS

ACSE Association Control Service Element
ADT Abstract Data Type
AK Acknowledgement
ASE Application Service Element
ASN.1 Abstract Syntax Notation 1
ASP Abstract Service Primitive
ATS Abstract Test Suite
BNF Backus-Naur Form
BTL Branching Time Logic
CASE Common Application Service Element
CC Connect Confirm
CCITT International Consultative Committee for Telephone and Telegraph
CCS Calculus of Communicating Systems
CM Coordination Messages
CP Coordination Point
CR Connection Request
CS Coordinated Single-layer
DC Disconnect Confirm
DR Disconnect Request
DS Distributed Single-layer
DT Data TPDU
EFSM Extended Finite State Machine
FDT Formal Description Technique
FIFO First In First Out
FTAM File Transfer and Access Management
IOD Input/Output Diagram

ISO International Organization for Standardization

IUT Implementation Under Test

LOTOS LOTOS

LS Local Single-layer

LT Lower Tester

MHS Message Handling System

MTC Main Test Component

OSI Open Systems Interconnection

OTH OTHERWISE

PLT Parallel Lower Tester

PUT Parallel Upper Tester

PCO Point of Control and Observation

PDU Protocol Data Unit

PICS Protocol Implementation Conformance Statement

PIXIT Protocol Implementation eXtra Information for Testing

PTC Parallel Test Compoment

RS Remote Single-layer

S-EFSM Specification EFSM

SDL System Description Language

T-EFSM Test EFSM

TMP Test Management Protocol

TMPDU Test Management Protocol Data Unit

TPDU Transport Protocol Data Unit

TTCN Tree and Tabular Combined Notation

TVS Test Verification System

UT Upper Tester

relop relational operator

List of Figures

Figure		Page
Figure 1.1	An Outline of Test Case Verification by Model Checking	11
Figure 2.1	A modular specification	17
Figure 2.2	A hierarchical task structure	18
Figure 2.3	A hierarchy of tasks created after an initialize statement	22
Figure 2.4	Task structure before the initialize statements	40
Figure 2.5	Task structure after the attach statements	40
Figure 2.6	(a) Declaration of PDU_A	44
	(b) declaration of constraint C0	45
Figure 2.7	Declaration of constraint C1	45
Figure 2.8	An example of a parameterized constraint	46
Figure 2.9	An instantiation of the constraint in Fig. 2.8	46
Figure 2.10	An example of static constraint chaining	47
Figure 2.11	An example of dynamic constraint chaining	49
Figure 2.12	An example of an ASN.1 constraint	53
Figure 2.13	Test case dynamic behavior table	57
Figure 2.14	(a) A test case	61
	(b) a test case in tree notation	61
	(c) a structured test case containing a test step	61
	(d) a structured test case in tree notation	61
Figure 2.15	(a) Structure of an internal node	69
	(b) structure of a leaf node	69
Figure 2.16	An example of representing a CHOICE construct as an I/O diagram	70

Figure		Page

Figure 2.17 An example of representing a SEQUENCE construct as an I/O diagram 70

Figure 2.18 An Estelle ConnectReq as an I/O Diagram 72

Figure 2.19 A CONreq ASP declaration in tabular notation 73

Figure 2.20 I/O Diagram representation of the CONreq ASP in Fig. 2.19 .. 73

Figure 3.1 The Local Single-layer Test Architecture 79

Figure 3.2 The Distributed Single-layer Test Architecture 80

Figure 3.3 The Coordinated Single-layer Test Architecture 82

Figure 3.4 The Remote Single-layer Test Architecture 83

Figure 3.5 Test Verification Systems corresponding to LS, DS, CS, and RS architectures 87

Figure 4.1 An example to compute *last(f)* 108

Figure 4.2 Tree representation of a formula f = (AU(NOT X)(OR YZ)) 115

Figure 5.1 RS Test Architecture 117

Figure 5.2 Test Verification System for the RS Architecture 118

Figure 5.3 A test case dynamic behavior 127

Figure 5.4 USP-EFSM, a simple model of the Presentation Service used by an ACSE protocol 129

Figure 5.5 The correct version of the test case in Fig. 5.3 130

Figure 5.6 Structure of the global state space of the TVS in Fig. 5.2 131

Figure 5.7 IOD representation of A_ASCreq message 132

Figure 5.8 Dynamic generation of UT behavior 134

Figure		Page
Figure 6.1	Coordinated Single-layer (CS) test architecture	135
Figure 6.2	The Test Verification System for the CS architecture	135
Figure 6.3	Class 2 Transport Protocol Specification	137
Figure 6.4	Underlying Service Provider of the Transport Protocol	138
Figure 6.5	A part of the Test Management Protocol	142
Figure 6.6	A CS architecture based single connection test case	145
Figure 6.7	EFSM representation of the test case in Fig. 6.6	146
Figure 6.8	Structure of the global state space	148
Figure 6.9	Representation of an event in a queue as an I/O diagram	149
Figure 7.1	A Multi-party Test Architecture	155
Figure 7.2	A Test Verification System for parallel test architectures	157
Figure 7.3	A CS architecture based multiple connection test case	162
Figure 7.4	EFSM Model of the test case in Fig. 7.3	163
Figure 7.5	A TVS for the test case in Fig. 7.3	165
Figure 7.6	A TVS for first connection in Fig. 7.3	166
Figure 7.7	A TVS for second connection in Fig. 7.3	166
Figure 7.8	Structure of the global state space	167

CHAPTER 1

INTRODUCTION

The role of computer networks in interconnecting a set of geographically distributed computer systems for collecting, processing, and exchanging information has been vital to the technological and administrative developments in the present day society. Communication protocols are key to the operation of a network of computer systems. Therefore, there is significant ongoing research and development in methodologies for specification, verification, and conformance testing of data communication protocols.

The International Organization for Standardization (ISO) and the International Consultative Committee for Telephones and Telegraph (CCITT) are the two world bodies working towards the standardization of protocol specifications, data type definitions for information exchange, and test case specifications using Formal Description Techniques (FDT). Examples of standardized formal description techniques are LOTOS [IS8807], Estelle [IS9074], and SDL [Z100] for specifying protocols, Abstract Syntax Notation One (ASN.1) [IS8824] for specifying the format of data exchanged between two computing entities, and the Tree and Tabular Combined Notation (TTCN) [ISO 9646] for test case specifications.

1.1 Protocol Engineering Life-cycle

The four steps in the protocol development process, called the *Protocol Engineering Life-cycle*, are design of a protocol specification, validation and verification of the specification, generation of an implementation from the specification, and conformance testing of the implementation [NASA 92a].

Producing a formal specification of a protocol is a human design process and it largely involves intuition. Since a human design process is error-prone, the

resulting protocol specification needs to be *validated* and *verified.* On one hand, protocol validation refers to the detection of some well known types of errors in the specification, such as unspecified reception errors, blocking receptions errors, deadlock errors, livelock errors, and synchronization errors [ZAFI 80]. On the other hand, protocol verification refers to whether the protocol specification meets the required service specifications [SAB 88].

The first automated protocol validation technique is *duologue-matrix analysis* [ZAFI 78, WEZA 78]. A duologue matrix is a notation to represent interaction sequences between two interacting graphs. This technique addresses the validation of a protocol between a pair of asynchronous processes by defining a number of fundamental rules of protocol behavior, which are incorporated in a validation function that can be applied to an interaction sequence to determine whether the sequence contains errors. By applying the validation function to all possible interaction sequences defined by a protocol, design errors in the protocol can be detected. The limitations of the duologue-matrix analysis technique are that the interacting processes must return together to their initial states after a finite number of interaction steps and that the theory only addresses two-process protocols.

The most widely studied protocol validation technique is *reachability analysis* [WEST 78], which consists of two steps. In the first step, a global state space is generated from the multiprocess finite state description of a protocol using a state perturbation technique. In the second step, the global state space is analyzed to detect the presence of some well-defined errors such as reception errors, deadlock errors, livelock errors, and channel overflow errors. Some other state space exploration methods controlling the size of the global state space can be found in [GOYU 84, VUHU 87, GOHA 85, LU 86, HOLZ 87, WEST 87, ZHBO 87]. A state space exploration technique for protocol validation is particularly attractive because it very easily lends itself to complete automation requiring only a formal definition of a

protocol in state diagram notation.

Over the last decade, a number of protocol verification techniques have been reported in the literature. For protocols specified in programming languages, program verification techniques are used. One can check that a protocol provides a desired service specified using first-order logic or a first-order version of temporal logic. However, program verification usually requires some insightful proofs that are hard to automate [HAOW 83]. Protocol verification using symbolic execution [BRJO 78] is a technique combining program verification and reachability analysis. It has the advantage that a protocol is interpreted by the verifier in essentially the same way as in reality. Therefore, it is possible to verify protocol behavior affected by the actual contents of messages rather than merely by its arrival.

In [SAB 88] an algorithmic procedure has been presented to verify whether a protocol provides a desired service. In this methodology, a protocol is described as a collection of synchronously interacting processes similar to the Communicating Sequential Processes (CSP) language [HOAR 78]. The safety behavior of a protocol is described using Finite State Machines (FSM), whereas the liveness behavior is described using propositional temporal logic. For checking the safety part of the specification, the FSM descriptions of all the processes are composed using a reachability procedure [ZAFI 83] and for the liveness property, the verification methodology checks whether the protocol gets into a set of states in which it will not provide the intended service.

A model checking approach based on branching time temporal logic is another technique used in protocol verification [FRV 86, CES 86]. In a later part of this chapter, we present an outline of the model checking approach.

For any practical use of a protocol specification, an executable implementation must be generated from the specification. Like the production of a protocol specification, generation of an implementation from a specification is also a human design

process and is error-prone. A practical way of determining whether an implementation behaves as stated in its specification in representative instances of communications, that is, whether the implementation conforms with its specification, is to test the implementation with a set of test cases. The process of testing an implementation to determine its conformance with the specification is known as *conformance testing* [ISO 9646]. From a software engineering view point, conformance testing generates confidence in the implementation.

1.2 Conformance Testing and Test Case Verification

1.2.1 Conformance Testing

The process of conformance testing starts with the test design phase in which an abstract test suite is developed. A test suite is a collection of test cases, such that each test case is used to test one protocol feature. There are two main approaches to developing a test suite:

- human design of test suites,
- semi-automatic design of test suites based on formal protocol specifications.

Conventionally, a test suite is designed by a test designer or a team of designers having expertise in the protocol standard as well as in the conformance testing framework and methodology [ISO 9646]. Such a design is a complicated and an error-prone process. Since software testing is a practical way of building confidence in software systems, a lot of research is being carried out on automatic test generation techniques. However, software testing in general and protocol testing in particular have not yet reached a state where a readily usable test suite can be automatically generated from a specification. Presently, therefore, test suites are manually designed [NCC 88, PTT 90] and many techniques on automatic test generation are being explored [UYDA 86, SILE 89, DSU 90].

The behavior of a protocol providing a set of communication functions is marked by sequences of events appearing at its service access points and composed of input events to the protocol and the responses of the protocol in the form of output events. From the conformance point of view, the purpose of a test case is to check whether the implementation satisfies the specification requirements with respect to a protocol function. A test case does this checking by applying sequences of input events, allowed by the specification, to the implementation and comparing the responses from the implementation with the expected events stated in the protocol specification. At the end of the testing process, a test case assigns a test verdict indicating whether the implementation passes or fails the test, or behaves in an inconclusive manner. If the observed behavior of the implementation is allowed by the protocol specification and the test purpose is satisfied, then the test case assigns a *Pass* verdict to the implementation. If the behavior of the implementation is not allowed by the protocol specification, then the test case assigns a *Fail* verdict. However, if the behavior of the implementation is allowed by the specification, but the purpose of testing is not satisfied, then the test case assigns an *Inconclusive* verdict.

1.2.2 Test Case Verification

Large size, nondeterministic events, combination of a large number of parameters, and communications among the modules of a protocol specification make the behavior of a communication protocol very complex and difficult to comprehend. Thus, manually designing a set of test cases to test an implementation of the specification becomes a nontrivial task. Those human designed test cases may contain several design errors. If erroneous test cases are used to test an implementation, the result of the testing process cannot be relied upon to decide the conformance of an implementation with its specification. Thus, it is essential to have a methodology to verify the correctness of test cases against the corresponding protocol specification.

The following are the issues contributing to the difficulties in verifying test cases: First, in general, test cases and protocols are specified in different formal description techniques, which use different notations to define data types and different behavioral semantics of their operations. For example, according to ISO's standardization framework, test cases are specified in TTCN, whereas a protocol can be specified in LOTOS, Estelle, or SDL.

Second, there are architectural differences between a protocol specification and the test cases. For example, because of the layered nature of OSI protocols, a protocol specification is written in such a way that it interacts with its user and service provider through two well-defined service access points, whereas depending on the test architecture [ISO 9646], a test case can interact with an implementation using only one service access point or two service access points one of which may be away from the implementation.

Third, the correctness of a Pass test verdict is directly coupled with the test purpose being satisfied.

Fourth, the correctness of a test behavior depends not only on the protocol specification, but also on the Test Management Protocol in some test architectures and on the service provider used as a communication medium between a test entity and the implementation under test.

Fifth, testing the multiple connection capability of an implementation involves parallelism in a test case, which adds an extra dimension to the test case verification problem.

Therefore, verifying a test case by comparing the behavior of the test case with that of a protocol specification is not a straightforward task.

1.3 Previous Work on Test Case Verification

The idea of test case verification is relatively new and there is very little published

work on this topic. In the following, we summarize some early approaches to test case verification.

In the first approach [NASA 90b], first both the LOTOS specification of a protocol and the TTCN specification of a test case are translated into a common semantic model, an Extended Finite State Machine (EFSM), with first-in first-out (FIFO) queues modeling the communication mechanism between the test case and the protocol specification. Next, an interleaved symbolic execution mechanism is used to compare the behavior of the test case EFSM with the protocol specification EFSM. A limitation of this approach is that only some static aspects of a test case can be compared with the behavior of a protocol specification and dynamic aspects due to timeouts cannot be verified.

In the second approach [DUBO 90], the verification process consists of two steps. First, a TTCN test case is translated into a LOTOS specification to eliminate the language differences. Second, a Test and Trace Analysis (TETRA) tool, based on a LOTOS interpreter [LOGR 88], takes the LOTOS specification of the test case and that of the protocol specification as inputs and computes their parallel composition by tracing the executable paths in the two specifications. The concerns expressed about the tool are its large space and long verification time requirements. Moreover, this technique does not address the architectural difference between a test case and a protocol specification.

1.4 Model Checking

In the traditional approach to verification of concurrent systems, the proof that a concurrent system meets its specification is constructed by hand using various axioms and inference rules in a deductive system [HAIL 82, OWLA 82]. Theorem provers are not of much help in verifying concurrent systems, because the task of proof construction is in general quite tedious.

It is argued that proof construction is unnecessary in the case of finite state concurrent systems [CES 86], and can be replaced by a *model checking* approach which mechanically determines if a system satisfies a property expressed in branching time temporal logic. Therefore, from an automation point of view, the model checking approach to system verification has gained diverse interests in the last decade [FRV 86, CES 86].

The basic approach to verifying whether a concurrent system satisfies a property by using the model checking concept contains the following three steps.

- The concurrent system is expressed as a finite state machine and a set of propositions evaluating to true is attached to each state. Such a finite state machine with a set of propositions attached to each state is called a *model* of the concurrent system.
- The properties to be verified on the concurrent system are expressed as formulas in branching time temporal logic.
- An algorithm, called a *model checker*, is used to verify whether the model satisfies the properties expressed as temporal formulas.

There are two kinds of properties one usually wants a concurrent system to satisfy:

- *Safety properties*, which state that something bad never happens—that is, the system never enters an unacceptable state.
- *Liveness properties*, which state that something good eventually does happen—that is, the system eventually enters a desirable state.

Some well known examples of safety properties of concurrent systems are *partial correctness*, *absence of deadlock*, and *mutual exclusion*. A liveness property that has received a lot of formal treatment is *program termination*. However, program termination is not a good thing to happen to every computing system. For example, an operating system should never terminate (*crash*). For such systems, other kinds of liveness properties are important, for example:

- Each request for service will eventually be answered.
- A message will eventually reach its destination.
- A process will eventually enter its critical section.

The nature of safety and liveness properties of a system depends on the nature of the computing system. Therefore, to verify the correctness of test cases one must define safety and liveness properties applicable to test systems.

1.5 Original Contributions of this Thesis

The contribution of this thesis is to develop a methodology to verify human designed test cases used to check the conformance of a protocol implementation with its formal specification. In the following, we outline a set of individual contributions which, when combined together, give rise to a test verification methodology.

In order to be able to compare the behavior of a test case with that of a protocol specification, the first barrier is due to the use of different languages in specifying protocols and test cases. Therefore, to eliminate their syntactic and semantic differences, a common representation notation, called an Extended Finite State Machine (EFSM), is defined and algorithms are presented to translate protocols specified in Estelle and test cases specified in TTCN into EFSMs.

The notion of *control and observation* is crucial to black-box testing. Since every test case is based on some test architecture, which defines the control and observation capability of a test case, it is important to take the architectural basis of a test case into account while verifying the test case. Therefore, we define the notion of a *Test Verification System* to account for the interconnection structure among the entities of a test architecture through service provider, points of control and observations, and other interaction points.

For a test verification system, we define a verification *model* representing the global behavior of the verification system and present an algorithm, based on reachability analysis, to generate a global state space from a test verification system.

To form a basis for verifying the correctness of a test case, we identify four classes of test case safety properties and one liveness property, and express them as formulas in branching time temporal logic. The four classes of safety properties are transmission safety, reception safety, synchronization safety, and verdict safety. The safety properties are designed to reflect some general characteristics of event-based communications and some particular characteristics of protocol testing.

The liveness property is designed to establish a relation between the *test purpose* and the *Pass verdict*. Presently, a test purpose is expressed in a natural language, which makes it difficult to verify whether the test case satisfies the test purpose. Therefore, we propose a notation in temporal logic to express what we call basic test purposes, which can be combined to construct larger test purposes. The liveness property is designed to verify that a test case behavior satisfying the test purpose ends in a Pass verdict.

We present a model checking algorithm to verify the test case properties on the model derived from the test verification system.

Our test verification methodology is applicable to all the ISO/CCITT test case and protocol specification languages such as TTCN, LOTOS, Estelle, and SDL and to all test architectures.

1.6 Organization of the Thesis

A pictorial description of the model checking methodology used to verify test cases is shown in Fig. 1.1. The thesis is organized according to the modular structure of the verification methodology.

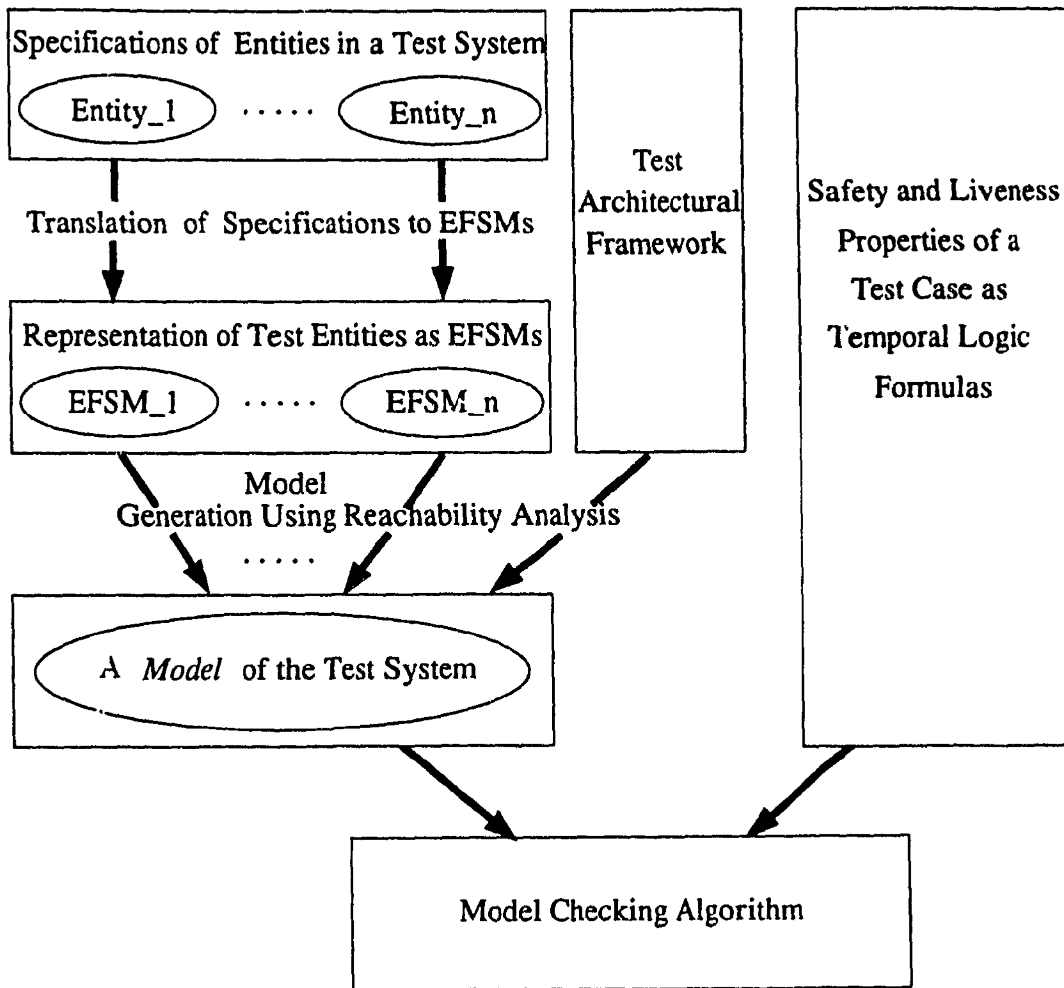


Figure 1.1 An Outline of Test Case Verification by Model Checking.

Chapter 2 contains four sections. In the first part, we define an Extended Finite State Machine (EFSM), which serves as a common representation notation for all the operational entities, such as the test case entities, the protocol specification, the Test Management Protocol, and the service provider, in a test verification system. In the second part, the syntax and semantics of Estelle constructs and an algorithm to translate an Estelle specification to an EFSM are presented. In the third part, we present a detailed description of the TTCN test specification language and an algorithm to translate a TTCN test case to an EFSM. In the fourth part, a notation called Input-Output Diagram (IOD) is defined to represent the Abstract Service Primitives (ASP)

and Protocol Data Units (PDU) exchanged among the EFSMs in a test verification system. The IOD representations of ASPs and PDUs are used to refer to the values of the fields contained in them while evaluating a boolean condition involving those fields.

Chapter 3 contains three sections. In the first part, we discuss various test architectures used in conformance testing. In the second part, we define the notion of a test verification system and derive a test verification system from each of the test architectures. In the third part, we present an algorithm to generate a state space representing the global behavior of a test verification system and present a methodology to attach a set of propositions with each global state.

Chapter 4 is devoted to model checking and consists of four parts. In the first part, we present the branching time temporal logic formalism. In the second part, the safety and liveness properties of a test case are expressed as branching time temporal logic formulas. In the third part, we model basic test purposes as temporal formulas and present a rule to compose basic test purposes into larger test purposes and express the liveness property as a temporal formula. A model checking algorithm, to verify test case properties on the model generated from a test verification system, is presented in the fourth part of Chapter 4.

In Chapter 5, we illustrate the verification of a test case based on the Remote Single-layer test architecture. The first three sections describe an Association Control Service Element (ACSE) protocol, a test case, and a service provider. A global state space is generated in the fourth section. Properties of the test case are verified in the fifth section. Generation of an Upper Tester is discussed in the last section.

Chapter 6 contains three sections. In the first part, a Class 2 Transport Protocol, a service provider of a transport protocol entity, and a Test Management Protocol used in testing a Class 2 Transport Protocol implementation are presented. In the second part, an example of verifying a test case, designed to test a Class 2 Transport Protocol

implementation in the Coordinated Single-layer test architecture, is presented.

In Chapter 7, a methodology to verify multiple connection test cases is outlined. This chapter contains three parts. In the first part, a parallel test architecture is discussed. In the second part, verification of a multiple connection test case is formalized using the verification technique for a single connection test case. An example of the verification of a multiple connection test case is presented in the third part.

In Chapter 8, contributions of the research described in this thesis are summarized. In addition, possible extensions of the verification methodology are discussed.

CHAPTER 2

EXTENDED FINITE STATE MACHINE MODELS

The use of Formal Description Techniques (FDTs) in the specification of communication protocols has received much attention, since such techniques allow a more systematic approach for protocol verification, validation, implementation, and testing compared to the traditional use of natural languages in protocol specifications. The three FDTs presently considered in this area are Estelle [IS9074], LOTOS [IS8807], and SDL [Z100]. The Tree and Tabular Combined Notation (TTCN) [ISO 9646] is the ISO's language for specifying a test suite for testing an implementation's conformance with its specification.

Estelle is based on a finite state machine model, which is extended by Pascal data types, expressions, and statements. The Estelle specification of a protocol may consist of a large number of interconnected finite state machine modules, which communicate among themselves through FIFO channels.

LOTOS, a process algebraic specification language, is a combination of Milner's Calculus of Communicating Systems (CCS) [MILN 80] formalism for behavior description and Abstract Data Types (ADTs) [EHMA 85] for data description. A set of *composition rules* is used to derive larger specifications from the primitive notions of *events* and *processes.*

SDL, like Estelle, is also based on an extended finite state machine model. It is largely oriented towards a graphical representation. Abstract data types are used to define data in an SDL specification.

In the TTCN test specification language, *constrained events* and *subtrees* constitute the building blocks in the design of the behavior part of a test case. Data in a test case are described using both a *tabular* notation and the Abstract Syntax Notation 1 (ASN.1) [IS8824].

For verifying a TTCN test case against a protocol specified in one of the FDTs, it is essential to remove the syntactic and semantic barriers between the test case and the protocol specification. Therefore, in this chapter we define an *Extended Finite State Machine (EFSM)*, a common notation for representing test cases and protocol specifications and present systematic procedures to translate them to the common notation. In this thesis, the scope of test case verification is limited to the Estelle specification language. In Section 2.1, we define an EFSM. A brief overview of Estelle features and rules to translate an Estelle specification to an EFSM are presented in Section 2.2. In Section 2.3, we discuss the TTCN specification language and present an algorithm to translate a TTCN test case to an EFSM. In Section 2.4, we define *I/O diagrams*, which provide a notation for information exchange among the EFSMs.

2.1 Extended Finite State Machine (EFSM)

We define a communicating EFSM to be a 9–tuple,

$$F = <S, S_t, V, R, s_{init}, Z, h_0, C_I, C_O>,$$

where S is a finite set of states, $S_t = \{(s, x) | s \in S \text{ and } x \text{ is a tag value}\}$ is a tagged set of states, V = {v1, v2,..., vn} is a finite set of data variables of types {t1, t2,..., tn}, respectively, R is a finite set of transitions to be defined in the following, $s_{init} \in S$ is the initial state, $Z \subseteq S$ is a set of final states, h_0 is a set of assignment functions initializing some variables in V, C_I is a set of input FIFO channels, and C_O is a set of output FIFO channels.

The set V is expressed as $V_1 \cup V_2$, where V_1 contains the variables locally used in the EFSM and V_2 contains the variables used for communication with other EFSMs. The variables in V_1 are of the conventional types such as integer, boolean, octetstring, etc., whereas the variables in V_2 are of types $\{IOD_1, IOD_2, ..., IOD_m\}$, where each Input/Output Diagram (IOD) type is defined to hold a structured value to be sent to or received from another EFSM. The IOD types are defined in detail in Section 2.4.

A transition in an EFSM is a 6-tuple, $\tau = <s, s', a, e, h, n>$, where s is the *from* state and s' is the *to* state of the transition, a is the *action* or *event* clause causing the transition to fire, e is the *enabling predicate* that must evaluate to *true* for the transition to fire, h is a set of value assignments to a subset of V, and n is the priority number of the transition in a set of choice transitions with the same *from* state.

An event in a transition can be one from the set {input, output, internal}, where the input and output events are known as *external* events and occur at some well defined interaction points through which two EFSMs communicate. An external event is characterized by three parameters: *interaction point, direction* ("?" denotes an input and "!" denotes an output), and the value (message) passed in the event. No channel or direction is associated with an internal event. For example, the event *L!msg* means a message *msg* is output to an interaction point *L*.

2.2 EFSM Model of an Estelle Specification

This section contains two parts. In the first part, we introduce the Estelle specification language and in the second part, we give rules to translate an Estelle specification to an EFSM.

2.2.1 Estelle Specification

Estelle [BUDE 87] is based on a finite state machine extended with Pascal data types and statements to describe actions. A number of notions generally applicable to distributed systems such as module, task, process, synchronous/asynchronous communication, interaction point, channel, nondeterminism, parallelism, etc. are found in Estelle. In the following, we explain each of them.

(A) Tasks: A distributed system specified in Estelle is viewed as a collection of communicating components called *module instances* or *tasks*. Each task has a number of input/output access points called *interaction points*, which are classified into two kinds: *external* and *internal*. For example, referring to Fig. 2.1, the distributed

system A consists of two concurrent tasks B and C. Task B is further refined into three concurrent tasks {D, E, F} and task C is refined into tasks {G, H}. The interaction points {1, 2,..., 13} are external, while points {14, 15, 16} are internal.

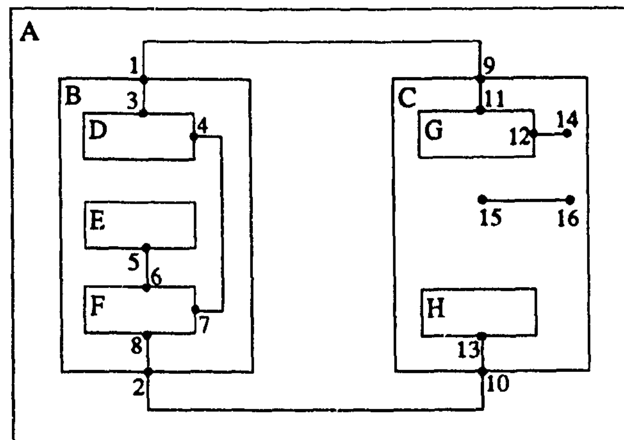


Figure 2.1 A modular specification in Estelle.

The behavior of a task is described in terms of a nondeterministic communicating extended finite state transition system. A task is *active* if its specification includes at least one transition; otherwise it is *inactive*. A task may have one of the following *class* attributes: *systemprocess*, *systemactivity*, *process* and *activity*. The tasks with systemprocess or systemactivity attributes are called *system* tasks. The class attributes of a collection of tasks represented in a structured manner in a specification determine the nature of parallelism in the specification as will be explained in the following.

(B) Structuring of Tasks: In a multitasking Estelle specification of a distributed system, two kinds of structuring principles are used: *hierarchical structure* and *communication structure*. The specification refinement principle in software engineering gives rise to a hierarchical structure of tasks as illustrated in Fig. 2.2, where the parent/child relationship is represented by edges and the root of the tree is the main task representing the specified system.

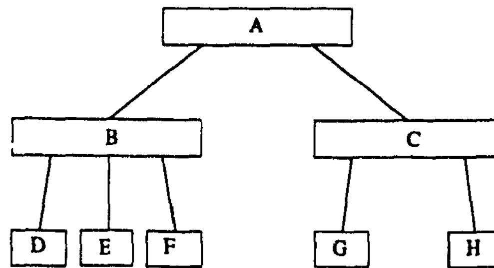


Figure 2.2 A hierarchical task structure.

Five attributing principles are observed within a hierarchy of tasks. (i) Every active task must be attributed. (ii) System tasks cannot be nested within an attributed task. (iii) Tasks with *process* or *activity* attributes must be nested within a system task. (iv) Tasks with *process* or *systemprocess* attributes may be substructured only into tasks attributed either *process* or *activity*. (v) Tasks with *activity* or *systemactivity* may be substructured only into tasks attributed *activity*.

The interconnection of interaction points for communication among the tasks in a specification gives rise to a *communication structure* within a specification. The elements of this structure are represented graphically, as shown in Fig. 2.1, by line segments binding the tasks' interaction points. There are two kinds of bindings of interaction points: *attached* and *connected*. When an external interaction point of a task is bound to an external interaction point of its parent task, these two interaction points are said to be *attached*. Two bound interaction points are said to be *connected* if both are external interaction points of two sibling tasks, or one is an internal interaction point of a task and the other is an external interaction point of one of its children tasks, or both are internal interaction points of the same task. Three rules are observed while establishing a communication link between two tasks. (i) An interaction point may be connected to at most one interaction point and it cannot be connected to itself. (ii) An external interaction point of a task may be attached to at most one external interaction point of its parent task and to at most one external interaction point of its children tasks. (iii) An external interaction point of a task

attached to an external interaction point of its parent task cannot be simultaneously connected.

(C) Communication: In Estelle, there are two kinds of intertask communication mechanisms: (i) message exchange and (ii) restricted sharing of variables. Tasks may exchange messages, called *interactions*. A task can send an interaction to another task through a previously established communication link between two interaction points of the tasks. An interaction received by a task at its interaction point is appended to an unbounded FIFO queue associated with this interaction point. The FIFO queue may either exclusively belong to this single interaction point or be shared with other interaction points of a task. A FIFO queue is called an *individual queue* if it exclusively belongs to a single interaction point and is called a *common queue* if it is shared with other interaction points of a task. Certain variables, declared as *exported* by the children tasks, can be shared between a task and its parent task. The simultaneous access to those variables by both a parent and a child is excluded because the execution of the parent's actions always has priority.

(D) Parallelism: Two kinds of parallelism between tasks can be expressed in Estelle: (i) asynchronous parallelism and (ii) synchronous parallelism. Parallelism and nondeterminism in Estelle are described using the notion of *computation steps*. A computation step begins by a selection of a set of transitions among those ready-to-fire by the subsystem's tasks with at most one transition per task. Then, the selected transitions are executed in parallel and, when all of them have completed, the next computation step begins. In that sense, the relative speed of tasks within a subsystem can be synchronized and the resulting parallelism is synchronous. If the selected set consists of exactly one transition for every computation step, then we have purely deterministic behavior within a subsystem. However, the selection of a set of transitions for synchronous or nondeterministic execution within one computation step of a subsystem depends on the parent/children priority principle and on the

way the subsystem's tasks are attributed. If a systemprocess is substructured into only processes, then all ready-to-fire transitions, with at most one transition per task, that are not in the ancestor/dependent conflict are selected, whereas in a task with systemactivity attribute, only one of them is selected. The intermediate selections, between the above two extremes are possible due to the fact that a systemprocess task may be substructured in both processes and activities.

(E) Channels and Interaction Points: In an Estelle specification, *channels* are abstract objects whose definitions specify sets of interactions. A particular interaction point has a precisely defined set of interactions that can be respectively sent and received through this point. For example, the following is a channel definition:

channel CHANNEL_ID (ROLE1, ROLE2);

by ROLE1:

    m1; ...; mN;

by ROLE2:

    n1; ...; nK

where m1, ..., mN, n1, ..., nK are interaction declarations. Each interaction declaration consists of a name and possibly some typed parameters such as "REQUEST(x: integer; y: boolean)". Now, an interaction point p1 may be declared as "p1: CHANNEL_ID(ROLE1)" and another interaction point p2 as "p2: CHANNEL_ID(ROLE2)." Two interaction points both referring to the same channel and different role identifiers play opposite roles, whereas if they refer to the same channel and the same role identifier, then they are said to play the same roles. Two interaction points that are connected must play opposite roles since the exchange of interactions takes place between them. Two interaction points that are attached must play the same role since the aim of attaching them is to replace one of them by the other. Finally, to specify whether the interaction point p1 does or does not share its queue with other interaction points, Estelle allows to specify "p1: CHANNEL_ID(ROLE1)

common queue" or "p1: CHANNEL_ID(ROLE1) *individual queue*."

(F) Modules: A *module* is specified by a pair of *header* and *body* definitions as shown below. A is the name of the header with systemprocess attribute and n is a formal parameter. It may be noted that a common queue is shared by the interaction points p1 and p2.

```
module A systemprocess (n: integer);
        ip  p: T(S) individual queue;
        ip p1: U(S) common queue;
        ip p2: W(K) common queue;
        export X, Y: integer; Z: boolean
end
```

At least one *module body* definition is declared for each module header definition. For example, in the following body definition, B is associated with header A.

body B for A; "body definition" **end**

A body definition is composed of three parts: declaration, initialization, and transition.

(i) Declaration Part: The declaration part of a body definition contains the usual Pascal declarations and declaration of objects specific to Estelle such as *channels, modules, module variables, states* and *state-sets*, and *internal interaction points*. A body definition may contain declarations of other modules giving rise to a tree structure. The definition "**modvar** X, Y, Z: A1" means that X, Y, and Z are variables of the module type specified by the module header A1. The internal behavior of each module is defined in terms of a state automaton whose control states are defined by enumeration of their names such as "**state**: IDLE, WAIT, OPEN, CLOSED." The **state** variable may assume only the values enumerated by the definition of the above form.

(ii) Initialization Part: The initialization part of a module body, indicated by the keyword **initialize**, specifies the values of some variables of the module with which

every newly created instance of this module begins its execution. Local variables and the control variable **state** may have their values assigned. Also, some module variables may be initialized which means that the module's children tasks can be created. To initialize Pascal variables, Pascal statements are used and to initialize the state variable to a control state, for example IDLE, "state to IDLE" is used. The following initialization part of a module (A, B) creates three module instances referenced by the module variables X, Y, and Z, respectively.

```
initialize
    begin
    init X with B1;
    init Y with B2;
    init Z with B1;
    connect X.p1 to Y.p2;
    connect Y.p1 to Z.p2;
    attach p to X.p';
    end
```

The concrete hierarchy of tasks of Fig. 2.3 corresponds to the hierarchical structure of the above module definitions. The initialization also establishes links between appropriate interaction points of the three newly created tasks.

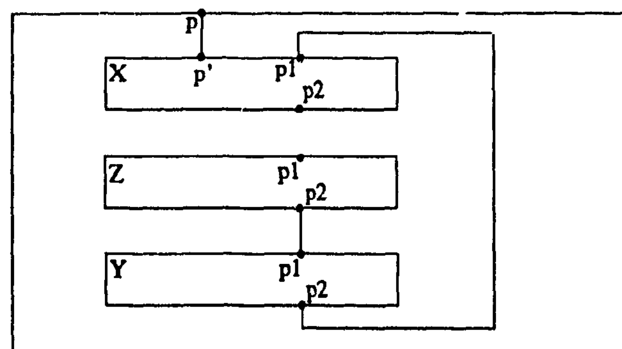


Figure 2.3 A hierarchy of tasks created after an **initialize** statement.

(iii) Transition Part: The transition part of a module describes its behavior. Each transition begins with the keyword **trans**. A *simple* transition is characterized by the fact that there is exactly one **begin ... end** block associated with each keyword **trans**. A *nested* transition is a shorthand notation for a collection of simple transitions. Each simple transition declaration is composed of two parts: *transition condition* and *transition action*. The transition condition is composed of one or more clauses of the following kinds:

i. from-clause (**from** *A1*,..., *An*, where *Ai* is a control state or state-set identifier);
ii. when-clause (**when** *p.m*, where *p* is an interaction point and *m* is an interaction);
iii. provided-clause (**provided** *B*, where *B* is a boolean expression);
iv. priority-clause (**priority** *n*, where *n* is a non-negative constant);
v. delay-clause (**delay**(*E1*, *E2*), where *E1* and *E2* are non-negative integer expressions representing the lower and upper limits of the delay, respectively).

The when-clause and the delay-clause are mutually exclusive in a transition definition. Transitions with a when-clause in their conditions are called *input* transitions and those without a when-clause are called *spontaneous* transitions. A transition is said to be *enabled* in a task state if the "from", "when", and "provided" clauses, if present in the transition condition, are satisfied in the state. A transition is said to be firable/executable if:

i. it is enabled in the state, and if it is a delay transition, with delay clause "delay(*E1*, *E2*)", then it must have remained enabled for at least *E1* time units, and
ii. it has the highest priority among transitions satisfying the previous requirement, where *higher priority* corresponds to *smaller non-negative integer*.

The *transition action* is composed of (i) a to-clause (**to** *A*, where *A* is a control state identifier) and (ii) a **begin ... end** block containing a sequence of extended Pascal statements. The to-clause specifies the next control state, which will be attained by

a task once the transition is fired. If the to-clause is omitted, then the next state is the same as the current state.

The Pascal extensions consist of additional statements that make it possible to create and destroy module instances, to create and destroy bindings between interaction points, and to send interactions. The create statements include init, connect, and detach. These have already been explained. The statements for the destroy operations are (i) *release X*, (ii) *disconnect p*, (iii) *disconnect X*, and (iv) *detach p*. The "release X" statement destroys the task referenced by the module variable X and all its descendant tasks, the "disconnect *p*" statement disconnects the interaction point *p* from the interaction point to which it was connected, the "disconnect X" disconnects all the interaction points of the children task referenced by X, and the "detach *p*" statement detaches the interaction point *p* from the interaction point it was attached to. The ability of a task to execute these statements within a transition gives the possibility to dynamically change the hierarchical tree structure of tasks as well as the communication links between them.

There is also a special statement *output*, which allows a task to send an interaction via a specified interaction point. For example, the statement "output *p1.m*" sends the interaction *m* via the interaction point *p1*. If *p1* and *p2* are the two end-points of a communication link, then the "output *p1.m*" statement leads to appending interaction *m* in the queue associated with the interaction point *p2*.

Any-clause: A transition in which an *any*-clause occurs is a shorthand for a set of transitions without any-clauses. The syntax of an any-clause is shown below:
any-clause := **any** domain-list **do**, where
domain-list:= ID-LIST: ORDINAL-TYPE,.., ID-LIST: ORDINAL-TYPE.
Each of these transition declarations contains an expanded transition resulting from the original expanded transition by replacing each applied occurrence of each ID in the ID-LIST of the any-clause. There shall be one simple, expanded transition for

each vector of values of ORDINAL-TYPE(s) contained in the any-clause. The length of each of these vectors is given by the number of identifiers in the ID-LIST of the domain-list of the any-clause.

(G) Specification Module: All modules in an Estelle specification are embodied in a unique module called *specification* which is defined as follows:

```
specification SPEC-NAME [system-class]
    [default-option]
    [time-option]
    "body-definition"
end.
```

where the parts in square brackets are optional. The system-class attribute is either *systemprocess* or *systemactivity*, and the default-option is either *individual queue* or *common queue*. The time-option defines the unit of time, millisecond, second, *etc.* to be used in a delay-clause.

(H) Global Situations: The semantics of Estelle is operational, that is, a *next-state* relation is defined over the set of the system *global states* which are called *global situations.* The next-state relation, also called the next-situation relation, specifies all possible situations that may be directly derived from a given situation. The overall behavior of an Estelle specification is characterized by the set of all sequences of global situations which can be generated from an *initial* situation of the specification. Each global situation of an Estelle transition system is composed of current information on:

i. the hierarchical structure of tasks within the specified system *SP*, the communication structure of tasks denoted by the bindings established between their interaction points, and the local state of each task. All this information is included in a *global instantaneous description* of *SP* denoted by *gid(SP).*

ii. the transitions that are in parallel execution within each subsystem; the set of

these transitions for *i*th subsystem is denoted by Ai (i = 1,..., n, where n is the number of subsystems).

Each global situation is denoted by: *sit* = (*gid(SP)*; A1,..., Ai,..., An). The global situation is said to be *initial* if the *gid(SP)* is initial and all sets Ai are empty. The *gid(SP)* is initial if it results from the initialization part of a specification *SP*. The next-situation relation defines the successive situations of an arbitrary current situation: (*gid(SP)*; A1,..., Ai,..., An). It is defined in the following manner. For every i = 1, 2,..., n,

i. if, in the current situation, Ai is empty, then (*gid(SP)*; A1,..., *AS(gid(SP)/i)*,..., An) is a next situation, where *AS(gid(SP)/i)* is the set of transitions selected for execution by the ith subsystem.
ii. if, in the current situation, Ai is nonempty, then for each transition *t* of Ai, (*t(gid(SP))*; A1,..., Ai – {t},..., An) is a next situation, that is, the new *gid(SP)* results from execution of *t* and *t* is removed from the set Ai.

The above two steps applied to the initial global situation define all possible sequences of global situations in the specification. The execution of a transition *t* of a task may cause a change in the task's local state. In particular, it may create a new child task and/or a new communication link. The transition may also output, that is, it may send an interaction which is put into the FIFO queue of another task. All these changes are expressed by *t(gid(SP))*. The selection of transitions to be executed within one computation step, by an ith subsystem, that is, the choice of the set *AS(gid(SP)/i)* is guided by the following: (i) the principle of parent/children priority, and (ii) the tasks' attributes. The rule applied to a task within a subsystem can be formulated as follows:

- if the task has a ready-to-fire transition, then this one will be selected,
- otherwise, depending on whether the task is attributed *process (systemprocess)* or *activity (systemactivity)*, respectively, all or one chosen nondeterministically of these ready-to-fire transitions offered by its children tasks will be selected.

2.2.2 Translation of an Estelle Specification to an EFSM

In the EFSM construction process, the notation *var(set of expressions)* means the set of variables used in the "set of expressions", and the function $Final(r,s)$ returns a transition r' which is identical to r except that the "to" state of r' is set to s.

Algorithm 2.1: The EFSM construction algorithm is divided into two phases.

(A) **First Phase:** In the first phase, we derive an EFSM from the **begin...end** block in an Estelle transition using the following three steps.

a. Process the functions.

b. Derive an EFSM for each procedure, function, and the main "begin ... end" block in the transition.

c. Resolve the procedure/function call references in the EFSMs derived in step b to obtain an EFSM for an Estelle transition.

(B) **Second Phase:** The second phase of the algorithm consists of the following three steps:

a. Update the EFSM obtained from the **begin...end** block of a transition to incorporate the Estelle transition-specific information such as **from, to, when,** and **provided.** We call such an updated EFSM a **transition-EFSM**, because it represents the EFSM for a complete Estelle transition. The collection of all the transition-EFSMs corresponding to the transitions in a module is called a **module-EFSM**.

b. Create a substructure by processing an *init* statement.

c. Combine the module-EFSMs for all the sub-modules in an Estelle specification to obtain an EFSM for the entire specification.

In the following, we explain each of the above steps in detail.

(A) **First Phase:**

a. Function Processing

In an Estelle specification, a function can be used in two ways:

i. as an expression on the righthand side of a simple assignment statement, such as y = *F-name(Plist)*, where *F-name* is a function name and *Plist* is a list of parameters that does not contain another function.

ii. as a parameter in a procedure/function call, like *P-name(F-name(Plist1), Plist2)*, where *P-name* is a procedure name.

Processing a function means transforming a function reference of the second type to the first type by introducing a new variable. For example, the statement *P-name(F-name(Plist1), Plist2)* is rewritten as a sequence of two statements, where *new1* is a new variable:

new1 = F-name(Plist1);

P-name(new1, Plist2).

b. Deriving an EFSM from a transition

The following algorithmic steps are used to systematically derive an EFSM from a function, a procedure, and the main **begin ... end** block in an Estelle transition. Each step of the algorithm is based on an Estelle construct that can be identified in the syntax analysis phase of an Estelle compiler.

1. compound-statement = "begin" statement-sequence "end"

The EFSM for this compound statement is the EFSM obtained from the "statement-sequence."

2. statement-sequence = statement1; statement-sequence2

Let

$F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ and

$F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >$ be the EFSMs for **statement1** and **statement-sequence2**, respectively. Then choose

$F = < S, S_t, V, R, s, Z, h, C_I, C_O >$, where

$S = (S_1 \cup S_2) - Z_1$,

$S_t = S_{t1} \cup S_{t2}$, $V = V_1 \cup V_2$,

$R = ((R_1 \cup R_2) - \{r_{1f}\}) \cup \{r_{2f}\}$, such that $(r_{1f} \in R_1) \wedge (To(r_{1f}) \in Z_1) \wedge$ $(r_{2f} = Final(r_{1f}, s_2))$,

$s = s_1$, $Z = Z_2$, $h = h_1 o h_2$.

3. Assignment statement

(a) label: X = Expression { no function call }

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$S_t = \{< s_1, L : label >\}, V = var(Expression)$,

$R = \{< s_1, s_2, i, [T], [X = Expression], 1 >\}$, where T stands for "true" and i denotes an internal event,

$s = s_1$, $Z = \{s_2\}$, $h = \epsilon$.

(b) label: X = Fname(Plist) { function call }

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$,

$S_t = \{< s_1, F : Fname(Plist) : label >\}, V = var(Plist)$,

$R = \{< s_1, s_2, +F, [T], [], 1 >\}$, where the +F event denotes that the transition is to be replaced by an EFSM corresponding to the declaration of the function *Fname* found as a tag of the *from* state of the transition,

$s = s_1$, $Z = \{s_2\}$, $h = \epsilon$.

4. Procedure call

label: Pname(Plist) { procedure call }

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$S_t = \{< s_1, P : Pname(Plist) : label >\}, V = var(Plist)$,

$R = \{< s_1, s_2, +P, [T], [], 1 >\}$, where +P denotes that the transition is to be replaced by an EFSM corresponding to the declaration of the procedure *Pname* found as a tag of the *from* state of the transition,

$s = s_1, \; Z = \{s_2\}, \; h = \; \epsilon.$

5. while-statement = "while" boolexpression "do" statement

Let

$F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ be the EFSM for the statement part.

Choose

$F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ as the new EFSM, where

$S = S_1 \cup \{s, s_2\}$, where s and s_2 are two new states,

$S_t = S_{t1}$,

$V = V_1 \cup var(boolexpression)$,

$R = R_1 \cup R_f \cup \{r_1, r_2\}$,

$R_f = \{< s_f, s, i, T, [], 1 > | s_f \in Z_1\}$,

$r_1 = < s, s_1, i, [boolexpression], [], 1 >$,

$r_2 = < s, s_2, i, [not(boolexpression)], [], 1 >$,

$Z = \{s_2\}, \; h = h_1.$

6. repeat-statement = "repeat" statement-sequence "until" boolexpression

Let

$F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ be the EFSM for the statement-sequence part. Choose

$F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ as the new EFSM, where

$S = S_1 \cup \{s_2\}$, where s_2 is a new state,

$S = S_{t1}$,

$V = V_1 \cup var(boolexpression)$,

$R = R_1 \cup R_f \cup R_b$

$R_f = \{< s_f, s_2, i, [boolexpression], [], 1 > | s_f \in Z_1\}$,

$R_b = \{< s_f, s_1, i, [not(boolexpression)], [], 1 > | s_f \in Z_1\}$,

$s = s_1$,

$Z = \{s_2\}, h = h_1$.

7. if-statement = "if" boolexpression "then" statemen1 "else" statement2

Let $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ and

$F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >$ be the EFSMs for statement1 and statement2, respectively. Choose

$F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ as the new EFSM, where

$S = S_1 \cup S_2 \cup \{s\}$, where s is a new state,

$S_t = S_{t1} \cup S_{t2}$,

$V = V_1 \cup V_2 \cup var(boolexpression)$,

$R = R_1 \cup R_2 \cup \{r_1, r_2\}$,

$r_1 =< s, s_1, i, [boolexpression], [], 1 >$,

$r_2 =< s, s_2, i, [not(boolexpression)], [], 1 >$,

$Z = Z_1 \cup Z_2, h = h_1 \cup h_2$.

8. for-statement = "for" controlvar ":=" initval ("to"|"downto") finalval "do" statement

Let $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ be the EFSM for the statement part. Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ as the new EFSM, where

$S = S_1 \cup \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$S_t = S_{t1}$,

$V = V_1 \cup \{controlvar\} \cup var(initval) \cup var(finalval)$,

$R = R_1 \cup \{r_1\} \cup R_f \cup R_b$,

$r_1 = < s, s_1, i, T, [controlvar = initval], 1 >$,

$R_f = \{< s_f, s_2, i, [not(controlvar \leq finalval)], [], 1 > |s_f \in Z_1\}$,

$R_b = \{< s_f, s_1, i, [controlvar \leq finalval], [], 1 > |s_f \in Z_1\}$,

$Z = \{s_2\}$, $h = h_1$.

9. case-statement =

"case" caseindex "of"

case constantlist1: statement-1

:

case constantlistn: statement-n

"end"

Let $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ thru $F_n = < S_n, S_{tn}, V_n, R_n, s_n, Z_n, h_n, C_{In}, C_{On} >$ be the EFSMs for statement-1 thru statement–n, respectively. Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ as the new EFSM, where

$S = S_1 \cup ... \cup S_n \cup \{s\}$, where s is a new state,

$S_t = S_{t1} \cup ... \cup S_{tn}$,

$V = V_1 \cup ... \cup V_n \cup \{caseindex\}$,

$R = R_1 \cup ... \cup R_n \cup \{r_1, ..., r_n\}$,

$r_1 = < s, s_1, i, [p_1], [], 1 >$, where p_1 is $(caseindex = CCL11) \vee ... \vee (caseindex = CCL1m)$ such that $caseconstantlist1 = \{CCL11, ..., CCL1m\}$,

$r_n = < s, s_n, i, [p_n], [], 1 >$, where p_n is $(caseindex = CCLn1) \vee ... \vee (caseindex = CCLnn)$ such that $caseconstantlistn = \{CCLn1, ..., CCLnn\}$,

$Z = Z_1 \cup ... \cup Z_n$, $h = h_1 \cup ... \cup h_n$, $C_I = C_{I1} \cup ... \cup C_{In}$, $C_O = C_{O1} \cup ... \cup C_{On}$.

10. output-statement = output channel.expression

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$V = var(expression)$,

$S_t = \{\}$,

$R = \{< s_1, s_2, channel!expression, [T], [], 1 >\}$,

$s = s_1$, $Z = \{s_2\}$, $h = \epsilon$.

11. Procedure call

Py(val1, ..., valn)

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$S_t = \{< s_1, P : +Py(val1, ..., valn) >\}$,

$V = var(val1, ..., valn)$,

$R = \{< s_1, s_2, +Py, [T], [], 1 >\}$,

$s = s_1$, $Z = \{s_2\}$, $h = \epsilon$.

12. Function call

new1 = Fy(val1, ..., valn)

Choose $F = < S, S_t, V, R, s, Z, h, C_I, C_O >$ such that

$S = \{s_1, s_2\}$, where s_1 and s_2 are two new states,

$S_t = \{< s_1, F : +Fy(val1, ..., valn) >\}$,

$V = var(val1, ..., valn)$,

$R = \{< s_1, s_2, +Fy, [T], [new1 = Fy(val1, ..., valn)], 1 >\}$,

$s = s_1,\ Z = \{s_2\}, h = \epsilon$.

c. Resolving procedure/function calls

1. Resolving procedure calls

Let Px be a procedure/function/begin...end block containing a call to the procedure Py and let F_1 and F_y be the EFSMs corresponding to Px and Py, respectively, such that $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ and $F_y = < S_y, S_{ty}, V_y, R_y, s_y, Z_y, h_y, C_{Iy}, C_{Oy} >$. Let $r_x = < j, s', +Py, T, [], 1 >$ be the transition representing the procedure call in Px and the extension of the state j be $< j, P : +Py(Vlist) >$, where *Vlist = {val1, ..., valn}*, and let the first transition in F_y be given by $r_y = < s_y, s_{1y}, +Py, T, [], 1 >$ with the extension of s_y as $< s_y, P : +Py(Plist) >$, where *Plist = {pm1, ..., pmn}*. To take care of the value assignments to formal parameters in the execution of a procedure call, we modify F_y to get $F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >$, where

$S_2 = S_y$,

$S_{t2} = S_y - \{< s_y, P : +Py(Plist) >\}$,

$V = var(val1, ..., valn)$,

$R_2 = R_y - \{r_y\} \cup \{r_1\}$,

$r_1 = < s_y, s_{1y}, i, [T], [pm1 = val1, ..., pmn = valn], 1 >$,

$s_2 = s_y,\ Z_2 = Z_y,\ h_2 = \epsilon$.

Next, we replace the transition r_x in F_1 by the EFSM F_2.

2. Resolving function calls

Let Fx be a procedure/function/begin...end block containing a call to the function Fy and let F_1 and F_y be the EFSMs corresponding to Fx and Fy, respectively, such that $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ and $F_y = < S_y, S_{ty}, V_y, R_y, s_y, Z_y, h_y, C_{Iy}, C_{Oy} >$. Let $r_x = < j, s', +Fy, T, [new1 = Fy(Vlist)], 1 >$ be the transition representing the procedure call in Fx and the extension of the state j be $< j, F : +Fy(Vlist) >$, where *Vlist = {val1, ..., valn}*, and let the first transition in F_y be given by $r_y = < s_y, s_{1y}, +Fy, T, [], 1 >$ with the extension of s_y as $< s_y, F : +Fy(Plist) >$, where *Plist = {pm1, ..., pmn}*. To take care of the value assignments to formal parameters in the execution of a function call, we modify F_y to get $F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >$, where

$$
\begin{aligned}
S_2 &= S_y \cup \{s_f\}, \\
S_{t2} &= S_y - \{< s_y, F : +Fy(Plist) >\}, \\
V &= var(val1, ..., valn), \\
R_2 &= R_y - \{r_y\} \cup \{r_1\} \cup R_f, \\
r_1 &= < s_y, s_{1y}, i, [T], [pm1 = val1, ..., pmn = valn], 1 >, \\
R_f &= \{< s, s_f, i, [T], [new1 = Fy], 1 > | s \in Z_y\}, \\
s_2 &= s_y, \; Z_2 = \{s_f\}, \; h_2 = \epsilon.
\end{aligned}
$$

Next, we replace the transition r_x in F_1 by the EFSM F_2.

B. Phase 2

a. Obtaining a transition-EFSM

The structure of a transition is as follows:

```
trans

    priority constant_value

    from s2

    to sf

    provided predicate

    when ip_id.event

    begin
        (transition_block)

    end
```

Let the EFSM representing the **begin...end** block in a transition be represented by $F_y = < S_y, S_{ty}, V_y, R_y, s_y, Z_y, h_y, C_{Iy}, C_{Oy} >$. We update F_y to incorporate the **from, to, provided** and **when** semantics and obtain a new EFSM $F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >$, where

$S_2 = S_y \cup \{s_2, s_f\}$, where s_2 and s_f are two new states,

$S_{t2} = S_y$,

$V_2 = V_y \cup var(predicate) \cup var(event)$,

$R_2 = R_y \cup \{r_1\} \cup R_f$,

$r_1 = < s_2, s_y, ip_id?event, [predicate], [], 1 >$,

$R_f = \{< s, s_f, i, T, [new1 = Fy], 1 > | s \in Z_y\}$,

$Z_2 = \{s_f\},\ h_2 = h_y$.

In the absence of a **when** clause, transition r_1 represents an internal transition in the EFSM and is represented as $< s_2, s_y, i, [predicate], [], 1 >$. Similarly, if a transition block does not contain a **when** clause, but contains a **delay** clause, then $r_1 = < s_2, s_y, i, [predicate], [], 1 >$.

b. Creation of a Substructure

The *initialization* part of an Estelle module contains a sequence of *init, connect,* and *attach* statements in a **begin ... end** block. Since the init statements create task instances of modules and the connect and attach statements define an interconnection structure among those newly created tasks, the initialization part of a module, in principle, creates an interconnected substructure of modules. In the following, we state the step of the EFSM construction algorithm corresponding to an *init* statement. An init statement is of the following form:

"init" module-variable "with" body-identifier [(parameter-list)].

Let $F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >$ be an instance of the EFSM representing a module identified by the body-identifier. Choose $F = < S, S_t, V, R, s, Z, h, C_{I1}, C_{O1} >$ as the new EFSM, where

$S = S_1 \cup \{s_1\}$, where s_1 is a new state,

$S_t = S_{t1}$,

$V = V_1$,

$R = R_1 \cup \{r_1\}$,

$r_1 = < s, s_1, i, T, [p_1 = a_1, ..., p_n = a_n], 1 >$, where $\{p_1, ..., p_n\}$ are the formal parameters of the module denoted by the body-identifier and $\{a_1, ..., a_n\}$ constitute the parameter-list,

$Z = Z_1,\ h = h_1$.

c. Combining Module-EFSMs

Modules in an Estelle specification communicate by generating interactions in output statements of their begin...end blocks. It is assumed that in the referred modules, transitions are defined corresponding to the internal interactions. Such a transition type is called an internal transition type. Also, we assume rendezvous type communication, where the first module ready to execute the interaction must

wait for the second module to be ready. To obtain single-module specifications, internal transition types should be eliminated. This elimination can be done by body replacements of the internal transition types in all other transitions containing an output statement referring to the internal interaction. Symbolic replacements for parameter values of the input interaction are also done. Because of the atomicity property of Estelle transitions, module combining is done using the following five steps.

1. Modify each transition-EFSM containing transitions corresponding to statements that make outputs to internal channels.
2. Execute the following three steps iteratively.
3. If a transition contains:
   a. an **init** mod-var **with** body-identifier, then create a new instance task of the module identified by the body-identifier.
   b. a **release/terminate** X, then delete the task instance identified by X.
   c. **attach/detach/connect/disconnect** statements, then update the communication structure among the tasks in the specification.
4. Eliminate intermodule interactions in the internal transition-EFSM.
5. Eliminate intermodule interactions in the transitions representing WHEN clauses in transition-EFSMs.

Here we explain step-1. Let r0(s1, s2) and r1(s2, s3) be transitions in a transition EFSM such that r0 contains an output statement in a transition-EFSM and r2(s4, s5) be another transition such that s5 is a final state in the transition-EFSM. Then we introduce another state s6, update transition r2(s4, s5) as r2(s4, s6), introduce a transition r1(s6, s5) corresponding to the transition r1(s6, s5), delete r1(s2, s3), and adjust transition r0(s1, s2) to be r0(s1, s3).

A static method for combining modules is presented in [SABO 86]. However, the behavior of a distributed system, in general, can be dynamic, that is, tasks and their

communication links can be created and destroyed dynamically. The dynamic module combination technique outlined in the above stated five steps uses the static module merging technique [SABO 86] between two instants within which the behavior of the specification is static as explained below.

The behavior of a specification becomes dynamic if a transition containing init/release/connect/attach/disconnect/detach is executed. We denote such a transition as a *dynamic transition*. Between the execution of two successive dynamic transitions, the behavior of a specification is static. Therefore, from a dynamic analysis point of view, we can use the static module merging technique, stated in steps 4 and 5 above, in combination with step 3 to dynamically configure the structure of a specification.

Because of the Estelle constructs such as init, release, connect, attach, disconnect, and detach statements, the interconnection structure of tasks in a specification is dynamically updated while merging the modules. In the module merging technique, each transition in the currently available tasks is scanned to see whether it makes an output or it receives an input or it changes the task structure through one of the operations belonging to the set {init, release, connect, attach, disconnect, detach}. For example, while merging the modules, let there be three tasks as shown in Fig. 2.4, where tasks X, Y, and Z have module bodies A, B, and C, respectively. Four transitions, which dynamically update the task structure of the specification, are shown in task Z. The transition <init Z1 **with** C1> creates a task Z1 with a body C1, the transition <**init** Z2 **with** C2> creates a task Z2 with a body C2, the transition <**attach** Z.p2 **with** Z1.p4> establishes a link between the interaction point p2 of task Z with the interaction point p4 of task Z1, and the transition <**attach** Z.p3 **with** Z2.p5> establishes a link between the interaction point p3 of task Z with the interaction point p5 of task Z2. The task structure after the interpretation of the four transitions in task Z is shown in Fig. 2.5.

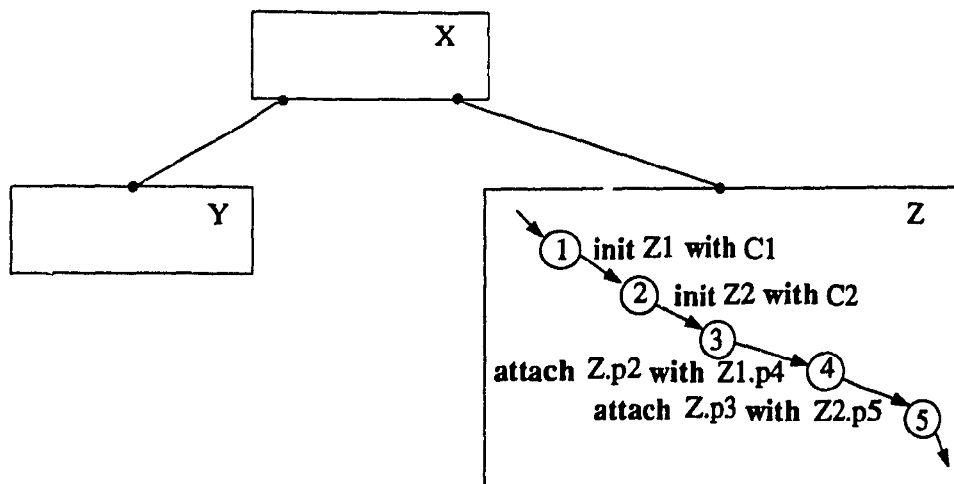


Figure 2.4 Task structure before the **init** statements.

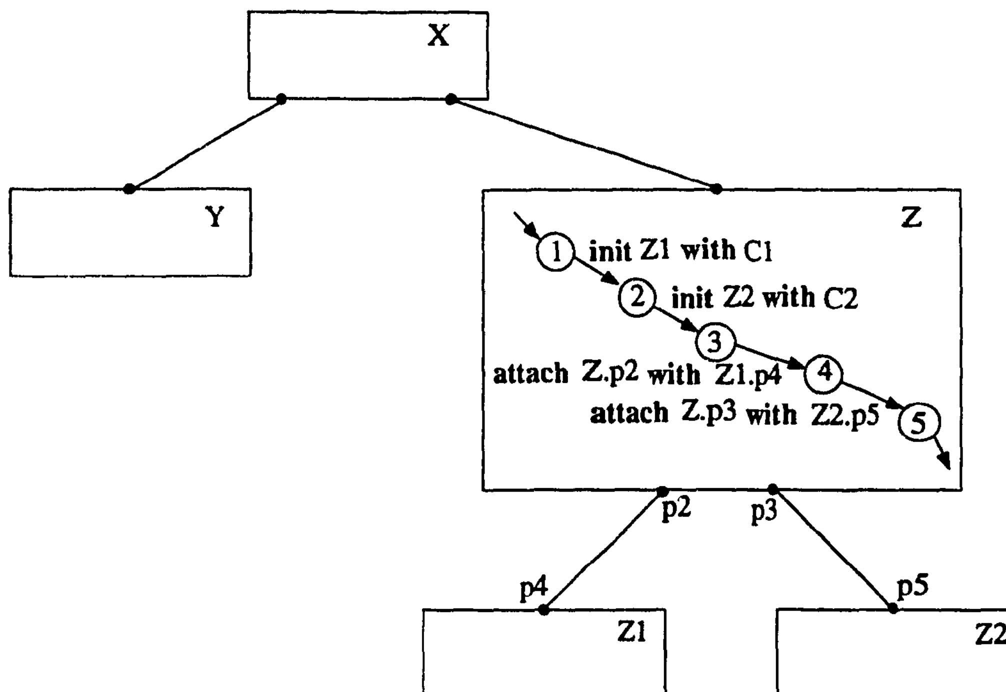


Figure 2.5 Task structure after the **attach** statements.

An advantage of keeping the dynamic constructs {init, release, connect, attach, disconnect, detach} as transitions in a transition-EFSM is that transitions in the tasks referred to in the dynamic constructs need not be interpreted, while merging the modules, before the tasks are actually required to be created and after the tasks are

terminated. That is, while merging modules at any instance of time, transitions of only those tasks which are active at that time are considered. Otherwise, if tasks are statically created by referring to the dynamic constructs, then transitions of some of the tasks are considered for module merging even though those transitions are not supposed to be active and transitions of some tasks are considered which are expected to be terminated. Therefore, we keep the dynamic constructs as transitions in transition-EFSMs and interpret them while merging the modules.

2.3 EFSM Model of a TTCN Test Case

This Section contains two parts. In the first part, we describe the test specification language TTCN. In the second part, we describe a systematic methodology to translate a TTCN test case to an EFSM [NASA 90a].

2.3.1 TTCN Specification

TTCN [ISO 9646] is a semiformal notation for use in the specification of OSI test suites. A good tutorial on TTCN can be found in [SAWI 92]. A TTCN specification contains four parts: overview, declarations, constraints, and dynamic behavior. In the following, we explain each part of a TTCN test case.

2.3.1.1 Overview

The overview section gives a broad and informal description of the test suite. Information is included in the overview on the name of the test suite, references to the Protocol Implementation Conformance Statement (PICS) and Protocol Implementation eXtra Information for Testing (PIXIT) proformas, indication of the test method to which the test suite applies, how the PICS and PIXIT are used in test case selection and parameterization, and the structural organization of the test cases in the suite.

2.3.1.2 Declarations

The purpose of the declaration section is to describe the set of test events and all components used in the test suite. There are two kinds of events: *Abstract Service Primitives (ASPs)*, which occur at the Points of Control and Observations (PCOs) used by the Lower and Upper Testers, and *timer events*. Other test suite components are test suite parameters, global constants, test suite variables, test case variables, PCOs, ASP parameters, and data types including the Protocol Data Units (PDUs) and their parameters.

2.3.1.3 Constraints

In TTCN, a constraint is defined by specifying a value for each PDU field and each ASP parameter. Each field entry in the field name column in a constraint declaration table should have been declared in the relevant ASP/PDU declaration; values assigned to each field shall be of the type specified in the ASP/PDU declaration. In the case of ASPs/PDUs that are sent, an explicit value shall be used. In the case of received ASPs/PDUs, the value may be an explicit value, a range of values, or a list of values. In the case of numeric and enumerated types, a relational operator may be used. Test suite and global constants that have been declared in the declarations part of the test suite may be included in the constraints part. In the following, we discuss the interpretations and various aspects of constraints.

(A) Interpretation of Constraints: The following rules are observed with respect to directions of the ASPs/PDUs:

(a) send ASPs/PDUs: the values specified in a constraint are sent in the ASP/PDU that is encoded according to this constraint;

(b) receive ASPs/PDUs: the values specified in a constraint must be received in the ASP/PDU that is decoded according to this constraint.

From the above descriptions of constraints, it is clear that constraints are associated with data values in two ways: (i) assignment of values to the fields in a send event, and (ii) satisfiability of boolean conditions on the values of fields in a receive event. In the following, we explain these interpretations in detail.

(i) Constraint on a Send Event: A send event is rewritten by augmenting the set of assignment operations associated with the event by another set of assignment operations derived from the constraint in the event and dropping the constraint in the new event without changing the semantics of the constraints. For example, in the send event

```
L!PDU_A(PDU_A.F3:=2, PDU_A.F4:="A") PDU_A[C0]
```

PDU_A is encoded according to the constraint C0; both PDU_A and C0 are described in Fig. 2.6(a) and Fig. 2.6(b), respectively.

PDU Type Declaration		
PDU Name: PDU_A PCO Type: LOWER Comment:		
Field Name	Field Type	Comments
F1	HEX	
F2	BOOLEAN	
F3	INTEGER	
F4	IA5STRING	
Detailed Comments:		

Fig. 2.6 (a) Declaration of PDU_A

PDU Constraint Declaration		
Constraint Name: C0 PDU Type : PDU_A Derivation Path : Comments :		
Field Name	Field Value	Comments
F1	'FF'H	
F2	T	
Detailed Comments:		

(b)

Figure 2.6 (b) Declaration of constraint C0.

The set of assignment operations in the above example is {PDU_A.F3 := 2, PDU_A.F4 := "A"}. The constraint C0 specifies the values of two fields of PDU_A: F1 and F2. According to the semantics of a constraint on a send event, the set of assignment operations that can be derived from the constraint C0 is {PDU_A.F1 := 'FF'H, PDU_A.F2 := T}. Therefore, the given send event line can be rewritten as follows.

```
L!PDU_A(PDU_A.F1:='FF'H,PDU_A.F2:=T,PDU_A.F3:=2,PDU_A.F4:="A")
```

(ii) Constraint on a Receive Event: We replace a constraint on a receive event by a set of boolean conditions checking those values, that is, we can rewrite a receive event by augmenting the set of boolean conditions associated with the event with another set of boolean conditions derived from the constraint on the receive event and dropping the constraint from the event line without changing the semantics of the constraints on the event. For example, in the receive event

```
L?PDU_A [PDU_A.F3 = N]   PDU_A[C1]
```

PDU_A is decoded according to the constraint C1 which is described in Fig. 2.7.

PDU Constraint Declaration		
Constraint Name: C1 PDU Type : PDU_A Derivation Path : Comments :		
Field Name	Field Value	Comments
F1	'10'H	
F2	FALSE	
Detailed Comments:		

Figure 2.7 Declaration of constraint C1.

The set of boolean conditions in the above event line is {PDU_A.F3 = N}, where N is some test case integer variable. The constraint C1 specifies values for two fields of PDU_A: F1 and F2. According to the semantics of a constraint on a receive event, the set of boolean conditions that can be derived from the constraint C1 is {PDU_A.F1 = '10'H, PDU_A.F2 = FALSE}. Therefore, the given receive event line can be rewritten as follows.

```
L?PDU_A [PDU_A.F1='10'H][PDU_A.F2=FALSE][PDU_A.F3=N]
```

(B) Parameterized Constraints: Constraint values may be parameterized. The constraint name in a constraint table is followed by a parameter list and the parameterized fields shall have these parameters as values. In the dynamic behavior section, while referring to a constraint, the constraint name is followed by a list of values, which are assigned to the parameters in the constraint reference. An example of a parameterized constraint is shown in Fig. 2.8.

PDU Constraint Declaration		
Constraint Name: C0(P1: HEX, P2: IA5STRING) PDU Type : PDU_A Derivation Path : Comments :		
Field Name	Field Value	Comment
F1	P1	
F2	'FF'H	
F3	'00'H	
F4	P2	
Detailed Comments:		

Figure 2.8 An example of a parameterized constraint.

PDU Constraint Declaration		
Constraint Name: C0I PDU Type : PDU_A Derivation Path : Comments :		
Field Name	Field Value	Comment
F1	'0'H	
F2	'FF'H	
F3	'00'H	
F4	"Hello"	
Detailed Comments:		

Figure 2.9 An instance of the constraint in Fig. 2.8.

A constraint reference PDU_A[C0('0'H, "Hello")] in an event line can be replaced by PDU_A[C0I], where C0I is an instance of C0 with the values P1 and P2 substituted by '0'H and "Hello", respectively; the tabular form of C0I is shown in Fig. 2.9.

ASP Constraint Declaration		
Constraint Name: N_DATAreq_With_T_CON_Class4_1 ASP Type : N_DATArequest Derivation Path : Comments : TCON_Class4_1 is a PDU constraint (i.e., static chaining is used.)		
Parameter Name	Parameter Value	Comments
CallingNetworkAddress	TS_Par3	
CalledNetworkAddress	TS_Par4	
ConnectionIdentifier	'CID'	
Data	TCON_Class4_1	
Detailed Comments:		

(a)

PDU Constraint Declaration		
Constraint Name: TCON_Class4_1 PDU Type : T_CONNECT1 Derivation Path : Comments :		
Field Name	Field Value	Comments
Source	TS_Par1	
Destination	TS_Par2	
T_Class	4	
UserData	"testing"	
Detailed Comments:		

(b)

Figure 2.10 An example of static constraint chaining.

(C) Chaining of Constraints: Constraints may be chained by referencing a constraint as the value of a parameter or field in another constraint. For example, the value of the Data parameter of a Network Data Request abstract service primitive

could be a reference to a Transport Connect Request PDU. Constraints can be chained in one of the following two ways:

i. *static chaining*, where an ASP parameter value or PDU field value in a constraint is an explicit reference to another constraint,

ii. *dynamic chaining*, where an ASP parameter value or PDU field value in a constraint is a formal parameter of the constraint. When such a constraint is referenced from a dynamic behavior, the corresponding actual parameter to the constraint is a reference to another constraint.

Chaining is useful in specifying constraints for nested PDUs in ASPs. An example of static constraint chaining is shown in Fig.2.10. An ASP constraint on an N_DATArequest service primitive is shown in Fig. 2.10(a). This constraint contains a parameter "Data" whose value, in fact, refers to another constraint "TCON_Class4_1", which is defined in Fig.2.10(b).

An example of dynamic constraint chaining is shown in Fig. 2.11. An ASP constraint on an N_DATArequest service primitive is shown in Fig.2.11(a). This constraint contains a parameter "Data" whose value, in fact, refers to a parameterized constraint name "A_Constraint." The constraint "N_DATAreq_With_T_CON" is dynamic in the sense that its value depends on the value of "A_constraint." For example, with two different values "TCON_Class4_1", shown in Fig.2.10(b), and "TCON_Class4_2", shown in Fig.2.11(b), of "A_Constraint", the constraint "N_DATAreq_With_T_CON(TCON_Class4_1)" is different from the constraint "N_DATAreq_With_T_CON(TCON_Class4_2)." In a dynamic constraint, since the actual parameter is a constraint name, which can itself be parameterized, it is possible to express an arbitrary depth of nesting of PDUs.

ASP Constraint Declaration		
Constraint Name: N_DATAreq_With_T_CON(A_Constraint: T_CONNECT) ASP Type : N_DATArequest Derivation Path : Comments : A_Constraint is a PDU constraint (i.e, dynamic chaining is used.)		
Parameter Name	Parameter Value	Comments
CallingNetworkAddress	TS_Par3	
CalledNetworkAddress	TS_Par4	
ConnectionIdentifier	'CID'	
Data	A_Constraint	
Detailed Comments: Different values of A_Constraint give rise to different instances of N_DATAreq_With_T_CON constraint resulting in a dynamic constraint.		

(a)

PDU Constraint Declaration		
Constraint Name: TCON_Class4_2 PDU Type : T_CONNECT2 Derivation Path : Comments :		
Field Name	Field Value	Comments
T_Address	WrongAddress	
T_Class	4	
UserData	"one, two, three"	
Detailed Comments:		

(b)

Figure 2.11 An example of dynamic constraint chaining.

Here we explain a methodology to derive a single constraint from a constraint named X that refers to a derivation path C1.C2.C3...Cn and defines a table containing value inheritance. This is done in two steps: (i) derive a single constraint from the chain of constraints in the derivation path C1.C2.C3...Cn and (ii) compose the

constraint values specified in the table of X with the single constraint derived in the first step.

(i) First step: Let C1.C2.C3...Cn be a chain of constraints in a derivation path. Then, a combined constraint C is generated from the chain C1.C2.C3...Cn by successively combining two consecutive constraints Cj.Ck at a time, starting with C1.C2, as stated below. In the set notation, let FCj = {(fi, vi)| fi is a field in Cj and vi is the value of fi in Ck} and FCk = {(fi, vi)| fi is a field in Ck and vi is the value of fi in Ck}.

```
procedure constraint_chain_combine(C1.C2....Cn){
   FCj = φ
   for i = 1, n
      do {
            Cm = constraint_combine(FCj, FCi)
            FCj = Cm
         }
   return(C = Cm)
}

procedure constraint_combine(FCj, FCk) {
   Initially, the set Cm is empty.
   for each element (fi, vi) ∈ FCj, do {
      for any v, if (fi, v) ∈ FCk, then Cm = Cm ∪ {(fi, v)}
                          else Cm = Cm ∪ {(fi, vi)};
      FCj = FCj − {(fi, vi)}
      FCk = FCk − {(fi, v)}
   }
   Cm = Cm ∪ FCk
   Return(Cm)
}
```

(ii) Second step: Let FC = {(fi, vi)| fi is a field in C and vi is the value of fi in C, where C is the set computed in step (i)}. Let FX = {(fx, vx)| fx is a field in the constraint table X and vx is the value of fx in X}. Then, the single total constraint representing X is given by TX, where TX = **constraint_combine**(FC, FX).

(D) Base Constraints and Modified Constraints

For every PDU type declaration, at least one *base constraint* is specified. A base constraint specifies a set of base, or default, values for every field defined in the PDU declaration. There may be any number of base constraints for any particular PDU.

When a subsequent constraint, called a *modified constraint*, is defined for a PDU, any fields not respecified in the modified constraint default to the values specified in the base constraint. The name of the modified constraint is a unique identifier. The name of the base constraint that is to be modified is indicated in the *derivation path* entry in the constraint header. This entry is left blank for a base constraint. A modified constraint can itself be modified. In such a case the derivation path indicates the concatenation of the names of the base and previously modified constraints, separated by dots. The rules for building a modified constraint from a base constraint are as follows.

i. If a parameter or field and its corresponding value are not specified in the constraint, then the value in the parent constraint shall be used, that is, the value is *inherited.*

ii. If a parameter or field and its corresponding value are specified in the constraint, then the specified value shall replace the inherited value.

If a base constraint is defined to have a formal parameter list, then the following rules apply to all modified constraints derived from that base constraint, whether or not they are derived in one or several modification steps.

i. The modified constraint shall have the same parameter list as the base constraint. In particular, there shall be no parameters omitted from or added to this list.

ii. The formal parameter list shall follow the constraint name for every modified constraint.

iii. Parameterized ASP parameters or PDU fields shall not be modified or explicitly modified.

(E) ASN.1 Modular Method: The ASN.1 Modular Method is a mechanism for specifying constraints on PDU fields or ASP parameters defined in the declarations part using ASN.1. The syntax and some of the semantics of ASN.1 constraints are different from those of the TTCN constraints. The constraints specified using the ASN.1 Modular Method can be used both for specifying PDUs/ASPs that are to be sent, or for specifying patterns against which an incoming PDU or ASP may be matched. For specifying constraints to be matched against received events, the ASN.1 Modular Method provides the following features.

i. The use of the "?" character as an element acts as a wildcard that must match against any single value, whereas the use of the "*" character as an element acts as a wildcard that will match against zero or more values.

ii. If a field is specified as OPTIONAL, then this indicates that the corresponding field in the incoming value may or may not be present.

iii. If a field is specified as DEFAULT, then it means that the following value shall be assumed if the field is missing from the incoming value.

iv. If a field is specified as CHOICE, then it means that the field in the incoming value may take any one of the values specified by the CHOICE.

v. The ANY type.

For specifying values for events to be sent, none of the above five features are used; this is to ensure that events to be sent are fully specified. Some of the above features are explained in the following.

OPTIONAL: In a receive constraint, if a field is specified with a value and an OPTIONAL attribute, then it means if the specified field is present in the incoming event, then the corresponding value must be equal to the value specified in the constraint. However, if the field is absent in the received event, then the value of the field in the constraint is irrelevant.

CHOICE: Whenever an ASP/PDU field has a CHOICE attribute, a set of alternative values is associated with the field. The field in the incoming value may take any of the values specified in the set of values. This naturally gives rise to a disjunction of boolean conditions checking the equality of each value in the CHOICE with the value of the corresponding field in the receive event.

DEFAULT: A DEFAULT attribute of a field implies that the value associated with the field shall be assumed if the field is missing from the incoming event. That is, a DEFAULT attribute does not impose a constraint on a receive event.

PUT and GET Semantics: The use of PUT and GET in a constraint reference in the dynamic behavior section calls for extra semantics in the constraint reference. The keyword GET is used in a constraint reference associated with a receive event. An example of an ASN.1 constraint is given in Fig. 2.12.

ASN.1 PDU Constraint Declaration
Constraint Name: ThePDU(x, y) PDU Type : APDU Derivation Path : Comments :
Constraint Value

SEQUENCE {
INTEGER [10] [LI] x,
BOOLEAN [ID] [LI] y
}

Detailed Comments:

Figure 2.12 An example of an ASN.1 constraint.

The following is an example of a constraint reference with the keyword GET in a receive event.

```
S_SAP?S_DATAind<UserData~APDU> ThePDU(GET N,GET STATUS)
```

This constraint means that the incoming integer and boolean values are assigned to N and STATUS. In other words, the above event line has the following semantics.

```
S_SAP?S_DATAind<UserData~APDU>(N:=APDU.x)(STATUS:=APDU.y)
```

The other keyword that can be associated with a parameter in a constraint reference is PUT. An example of an event line using PUT is

```
S_SAP?S_DATAind<UserData~APDU>ThePDU(PUT N,PUT STATUS)
```

which means that the event matches if the incoming S_DATAind event has a UserData field whose value is the correct encoding of the value obtained by substituting the formal parameters x and y in ThePDU by the values of the actual variables N and STATUS, respectively. Therefore, the above event can be rewritten as:

```
S_SAP?S_DATAind<UserData~APDU> [APDU.x=N][APDU.y=STATUS].
```

(F) Constraint Processing: In the following, we present procedures to transform a constraint on a receive event to a conjunction of boolean conditions and a constraint on a send event to a set of assignment functions. A TTCN event line *El* with a constraint is represented as:

El := N Label E (h) [B] C {Comment},

where *N* is the event line number, *Label* is a label identifier, *E* is an event, *h* is a set of assignment functions, *B* is a boolean condition, and *C* is a constraint on *E*.

A constraint *C* is viewed as a set of triplets, such as, $C = \{\langle f, a, Val\rangle \mid f$ is a field in E, $a \in$ {OPTIONAL, CHOICE, DEFAULT, ANY, NULL} is an attribute, and *Val* is a value or a set of values {*val1, val2, ..., valn*}}.

Since a constraint is defined only for an external input/output event, for the purpose of processing a constraint, we define two boolean functions *input(E)* and *output(E)*, such that *input(E)* and *output(E)* evaluate to true if *E* is an input event and an output event, respectively. A constraint processing procedure *Constraint_processing*,

which takes *El* as input and returns an event line *El'* without *C*, is presented below.

```
Procedure Constraint_processing(El) {
    /* El := N Label E (h) [B] C {Comment} */
    If input(E) then do {
        Bl = input_processing(E, C)
        return(El' := N Label E (h) [B AND Bl] {Comment})
    }
    else if output(E) then do {
        hl = output_processing(E, C)
        return(El' := N Label E (h, hl) [B] {Comment})
    }
}
```

The two procedures *input_processing(E, C)* and *output_processing(E, C)* are defined in the following.

```
Procedure input_processing(E, C) {
    /* C = {<f, a, Val>} */
    Bl = True
    for each <f, a, Val> ∈ C, do {
        If (Val == "?") then
            Bl = [Bl ∧ [[E.f == ANY] ∧ [E.f ≠ NULL]]]
        else if (a == OPTIONAL) then
            Bl = [Bl ∧ [[E.f == Val] ∨ [E.f == NULL]]]
        else if (a == DEFAULT) then do
                { if [E.f == NULL] then E.f = Val
                  else Bl = [Bl ∧ [E.f == Val]]
                }
        else if (a == CHOICE) then
```

```
        Bl = [Bl ∧ [[E.f == val1] ∨ ... ∨ [E.f == valn]]]
      else if (a == ANY) then
        Bl = [Bl ∧ [E.f == ANY]]
    }
    return (Bl)
}
Procedure output_processing(E, C) {
    hl = φ
    for each <f, a, Val> ∈ C, do {
        hl = hl ∪ {E.f = Val}
    }
    return (hl)
}
```

Note that in the *output_processing(E, C)* procedure, the attributes {OPTIONAL, CHOICE, DEFAULT, ANY, NULL} and a wildcard value "?" of fields of an event are not considered, because none of these features are used in a send event to ensure that events to be sent are fully specified.

2.3.1.4 Dynamic Behavior

The dynamic behavior table for a test case contains the specification of the combinations of sequences of test events that are deemed possible by the test suite specifier. Such a table is provided in a format shown in Fig. 2.13. The first column is for numbering the lines on the table. In the second column, a *label* name can be associated with each behavior line. The third column contains the behavior description. Constraint references are stated in column four. A test verdict can be attached to a behavior line in column five. A description of the behavior line can be put in column six in the form of a comment. In the following, we explain the dynamic behavior table in detail.

Test Case Dynamic Behavior					
Test Case Name : *TestCaseIdentifier* **Group** : *TestGroupReference* **Purpose** : *FreeText* **Default** : *[DefaultsReference]* **Comments** : *[FreeText]*					
Nr.	**Label**	**Behavior Description**	**Constraints Ref.**	**Verdict**	**Comments**
1 *2* . . . . . *n*	. . *[Label]*	. . *StatementLine*	*[Constraint Reference]*	*[Verdict]*	*[FreeText]*
Detailed Comments: *FreeText*					

Figure 2.13 Test case dynamic behavior table.

(A) Purpose: A test *purpose* is a natural language description of the protocol functionality of an implementation that the test case is supposed to test for conformance.

(B) Behavior Description: The behavior description column of a dynamic b .avior table contains the specifications of TTCN statements that are deemed possible by the test suite specifier. Each TTCN statement is shown on a separate statement line. The statements can be related to one another in two ways: (i) as sequences of statements and (ii) as alternative statements. Sequences of statements are represented one statement line after the other, each new statement being indented once from left to right, with respect to its predecessor as shown below:

```
EVENT_A
  CONSTRUCT_B
    EVENT_C.
```

Statements at the same level of indentation and belonging to the same predecessor node represent the possible alternatives that may occur at that time as shown below:

CONSTRUCT_A1

STATEMENT_A2

EVENT_A3.

Whether a statement can be evaluated successfully depends on various conditions associated with the statement line. These conditions are not mutually exclusive, that is, it is possible that at any given point in time more than one statement line could be evaluated successfully. Since statement lines are evaluated in the order of their appearance in the set of alternatives, the first statement with a fulfilled condition will be successful.

Each behavior description table contains at least one behavior tree. For unambiguously referring to a tree, each tree has a unique name. The first tree in a behavior description is called the *root* tree. All the trees except the root may be parameterized.

The tree notation allows the specification of test events initiated by the test entities (SEND and IMPLICIT SEND events), test events received by the test entities (RECEIVE, OTHERWISE, and TIMEOUT), constructs (GOTO, ATTACH, and REPEAT), and pseudo-events comprising of boolean expressions, assignments, and timer operations. These are collectively known as statements. Test events can be accompanied by boolean expressions (qualifiers), assignments, and timer operations. Boolean expressions, assignments, and timer operations can also stand alone, in which case they are called pseudo-events. A send symbol is denoted by "!" and a receive symbol is denoted by "?". Every send or receive event takes place at a Point of Control and Observation (PCO), which is used as a prefix to a send/receive symbol.

(i) Receive Event: A receive event line evaluates successfully if an incoming ASP/PDU on the same specified PCO matches the event line. A match occurs if the following conditions are fulfilled:

i. The incoming ASP/PDU is valid according to the ASP/PDU type definition referred to by the event name on the event line. In particular, all parameters

and/or field values must be of the defined types.

ii. The ASP/PDU matches the constraint reference on the event line.

iii. In cases where a boolean expression is specified on the event line, the boolean expression must evaluate to true. The boolean expression may contain references to ASP/PDU parameters.

The incoming event is removed from the PCO queue only when it successfully matches a receive event line.

(ii) Send Event: A send event line with a qualifier is successful if the expression in the qualifier evaluates to true. Unqualified send events are always successful. An outgoing ASP/PDU that results from a send event is constructed as follows:

i. The values of the ASP parameters and PDU fields are set as specified in the constraint referenced on the event line.

ii. Any direct assignments to ASP parameters or PDU fields on the event line supersede the corresponding values specified in the constraint.

(iii) Implicit Send Event: In the Remote Test [ISO 9646] architecture, to be discussed in Chapter 3, although there is no explicit PCO above the Implementation Under Test (IUT), it is necessary to have a means of specifying, at a given point in the description of the behavior of the Lower Tester, that the IUT should be made to initiate a particular ASP/PDU. For this purpose, the implicit send event is defined, with the following syntax: < IUT!ASP/PDU >. There is no specification of what is done to the IUT to trigger the specified ASP/PDU.

(iv) TIMEOUT: The TIMEOUT event allows expiration of a timer to be checked in a test case. When a timer expires, a TIMEOUT event is placed into a timeout list. A TIMEOUT is not associated with any PCO. TIMEOUTs are used in a test behavior to avoid any indefinite wait due to a permanent communication failure or due to a faulty implementation that never generates an expected output.

(v) **OTHERWISE:** Since a faulty implementation can generate any unexpected events whose types and values are not known at the time of designing a test case, TTCN provides a facility in the form of OTHERWISE to trap those events as shown in the following:

PCO1? A

PCO1? B

PCO1? OTHERWISE.

If the alternative events before an OTHERWISE are not successful and there is an event at the PCO referenced in the OTHERWISE statement, then the OTHERWISE event becomes successful. Due to the significance of ordering of alternatives, incoming events which are alternatives following an unconditional OTHERWISE on the same PCO will never match. An OTHERWISE event may also be used together with qualifiers and/or assignments. If a qualifier is used, this boolean expression becomes an additional condition for accepting any incoming event. If an assignment is used, the assignment will take place only if all conditions for matching the OTHERWISE are satisfied. An example of a conditional OTHERWISE is shown below:

PCO1? A

PCO2? B [X = 2]

PCO2? OTHERWISE [X <> 2] (Reason := "X not equal 2").

(vi) **Test Step:** A test case description, shown in Fig. 2.14(a), is structured as a tree, as shown in Fig. 2.14(b), with test events as nodes in the tree and verdict assignments as its leaves. A test case may also be structured by using *test steps*, as shown in Fig. 2.14(c), where +STEP is a test step. The structured tree description of the structured test case in Fig. 2.14(c) is shown in Fig. 2.14(d).

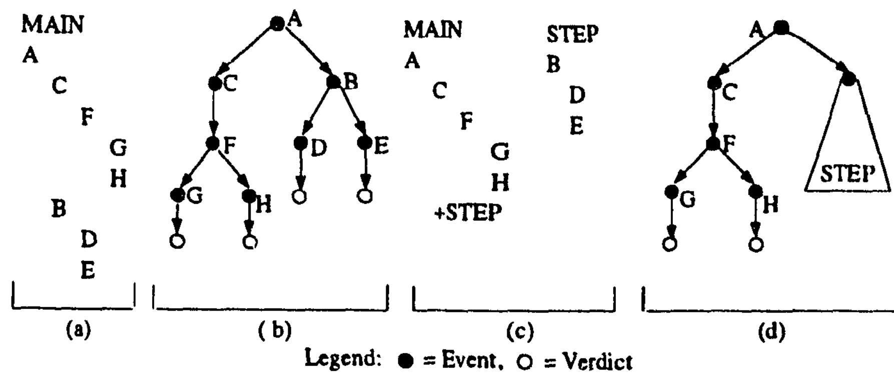


Figure 2.14 : (a) A test case, (b) a test case in tree notation, (c) a structured test case containing a test step, and (d) a structured test case in tree notation.

(C) Default Behavior: A test case specifies alternative behavior for every possible event. It often happens that in a test behavior tree every sequence of alternatives ends in the same behavior. Therefore, this common behavior is factored out as a *default* behavior to the tree.

2.3.2 Translation of a TTCN Test Case to an EFSM

Now we present the rules for translating a TTCN test case to an EFSM. To assist us in the EFSM construction process, we introduce an auxiliary event *empty* that follows every *leaf* event in a test case tree. In the following, we assume that $F' = < S', S'_t, V', R', s'_0, Z', h'_0, C'_I, C'_O >$ and $F'' = < S'', S''_t, V'', R'', s''_0, Z'', h''_0, C''_I, C''_O >$ are EFSMs for subtrees A' and A'', respectively. Let s_0 be a control state absent in both S' and S''. If $f : V1 \rightarrow V2$ and $g : V2 \rightarrow V3$ then their composition $g \circ f : D1 \rightarrow D3$ is defined by $g \circ f(x) = g(f(x))$. Let $\varepsilon : V \rightarrow D$ be an arbitrary but fixed function, where D is a set of constants.

Let r = $< s, s', a, p, f, n, l, V_d >$ be a transition. Then we define the following functions on r.

i) From(r) = s, the *from* clause of r,

ii) To(r) = s', the *to* clause of r,

iii) R(s) = {r = $< s, s', a, p, f, n, l, V_d >$| From(r) = s},

iv) A(s, s') = a, the *action* clause of a transition from s to s',

v) Pr(s, s') = n, the *priority* clause of a transition from s to s',

vi) $\Theta(s, s') = f$, the function clause of a transition from s to s'.

Algorithm 2.2: The translation rules in the algorithm are applied bottom-up starting with the *empty* event. The algorithm operates in five phases. In the first phase, we apply some replacement and rewrite rules to simplify a test case. In the second phase, we construct an EFSM for the main tree and an EFSM for each subtree. In the third phase, tree attachments are resolved. In the fourth phase, each transition is updated by dropping the label clause and making the verdict clause a tag of the *to* state of the transition. In the fifth phase, the ASP/PDU events are represented as I/O diagrams. The translation rules are as follows.

(1) First Phase: In this phase, all the default behaviors are expanded and each REPEAT statement is replaced by a combination of a GOTO and a boolean condition. The rules for expanding defaults and replacing REPEATs are discussed in [ISO 9646]. Constraint references are eliminated by transforming a constraint on a send event to a set of assignment functions and that on a receive event to a boolean expression as explained in Section 2.3.1.

(2) Second Phase: In this phase, we present rules for deriving an EFSM from a test tree; the resulting EFSM may contain references to other trees, which are resolved in the third phase.

Base Cases

- **The Dummy Event**

empty

Corresponding to an **empty** event, the EFSM consists of just one state and all other fields have null values. Therefore, choose $F =< \{s_0\}, \phi, \phi, \phi, s_0, \{s_0\}, \varepsilon, \phi, \phi >$.

- **GOTO Label**

Here we create a new state s_0 and a tag $(s_0, Label)$ with all other fields in the EFSM equated to null. Subsequently, the tag $(s_0, Label)$ is used to resolve the reference to the **Label** occurring at a predecessor statement of the GOTO.

Inductive Steps

- **Tree attachment**

+X$(t_1, ..., t_m)$ **L**

A'

Here we create a new state s_0 and add a transition from s_0 to the start state in F', the EFSM for A'; the event in the added transition is +X which will be removed while resolving the tree attachment in the third phase. There may be a GOTO in A' jumping to the tree attachment above. Therefore, we resolve all such GOTOs jumping to the label L by adding a transition with an internal event from a state with an extension +L to s_0. This tree attachment will be resolved in the third phase of the algorithm. Thus, choose

F = $< S' \cup \{s_0\}, S_t, V' \cup f_p(X), R, s_0, Z', \epsilon, \phi, \phi >$ with

$R = R' \cup \{< s_0, s_0', +X, T, \epsilon, 1, L, V_d >\}$

$\cup \{< s, s_0, i, true, \phi, 1, \phi, \phi > |(s, +L) \in S_t'\}$

$S_t = S_t' \cup \{(s_0, +X)\} - \{(s, +L)\}$,

$f_p(X) = < v_1, ..., v_m >$, where $f_p(X)$ denotes the formal parameters of **X**.

- **A send event**

g!t [p] (f) L V_d

A'

Here we create a new state s_0 and add a transition from s_0 F' to the start state in the EFSM for A'. The added transition has the event **t**, predicate **p**, verdict V_d,

and function f. Moreover, for every GOTO in A' to the label L, we add a transition with an internal event from the state with an extension +L to s_0. Therefore, choose

$$F = S' \cup \{s_0\}, S_t, V', R, s_0, Z', \phi, \phi, \phi > \text{ with}$$
$$R = R' \cup \{< s_0, s'_0, g!t, p, f, 1, L, V_d >\}$$
$$\cup \{< s, s_0, i, true, \phi, 1, \phi, \phi > |(s, +L) \in S'_t\}$$
$$S_t = S'_t - \{(s, +L)\}.$$

•**A receive event**: This case is similar to a send event case.

•**A pseudo-event**

[p] (f) L V_d

A'

This pseudo-event is translated to an internal event. Thus, choose

$$F = < S' \cup \{s_0\}, S_t, V', R, j_0, Z', \phi, \phi, \phi >, \text{ where}$$
$$R = R' \cup \{< s_0, s'_0, i, p, f, 1, L, V_d >\}$$
$$\cup \{< s, s_0, i, true, \phi, 1, \phi, \phi > |(s, +L) \in S'_t\}$$
$$S_t = S'_t - \{(s, +L)\}.$$

• **Timer events**

?TIMEOUT L V_d

A'

Because a TIMEOUT event does not occur at any particular PCO, it is translated to a transition without an internal event. Thus, choose

$$F = < S' \cup \{s_0\}, S_t, V', R, s_0, Z', \phi, \phi, \phi > \text{ with}$$
$$R = R' \cup \{< s, s'_0, i, true, \phi, 1, L, V_d >\}$$
$$\cup \{< s, s_0, i, true, \phi, 1, \phi, \phi > |(s, +L) \in S'_t\}$$
$$S_t = S'_t - \{(s, +L)\}.$$

• **Timer operations**

START Tid Tval / CANCEL Tid / READTIMER Tid L V_d

A'

Since these pseudo-events do not occur at any particular PCO, they are translated to internal events. Thus, choose

F = $< S' \cup \{s_0\}, S_t, V', R, s_0, Z', \phi, \phi, \phi >$ with
$R = R' \cup \{< s, s'_0, i, true, \phi, 1, L, V_d >\}$

$\cup \{< s, s_0, i, true, \phi, 1, \phi, \phi > |(s, +L) \in S'_t\}$
$S_t = S'_t - \{(s, +L)\}$.

Corresponding to a timer pseudo-event, we create a new state s_0, establish a transition from s_0 to the initial state of F', and resolve any references to the label L in a GOTO in A'.

- **A set of alternative events**

  A'

  A''

  :

  $A^{n'}$

We create a new state s_0 and add transitions from s_0 to all the states to which there are transitions from s'_0, s''_0,..., $s^{n'}_0$. All the extensions of s'_0, s''_0, ..., $s^{n'}_0$ are made the extensions of s_0. Therefore, choose

F = $< S, S_t, V, R, j_0, Z', \phi, \phi, \phi >$ with

$S = (S' - \{s'_0\}) \cup ... \cup (S^{n'} - s^{n'}_0) \cup s_0$,

$V = V' \cup V'' \cup ... \cup V^{n'}$,

$S_t = \{(s_0, +X) | + X \in \{S'_t(s'_0) \cup ... \cup S_t{}^{n'}(s^{n'}_0)\}\} \cup S_t\text{-}1$,

$S_t\text{-}1 = (S'_t - \{(s'_0, +X)| + X \in S'_t(s'_0)\}) \cup ... \cup (S^{n'}_t - \{(s^{n'}_0, +X)| + X \in S^{n'}_t(s^{n'}_0)\})$,

$R = \{< s_0, s, a, p, f, n, L, V_d > | < s_1, s, a, p, f, n, L, V_d > \in R_1\} \cup (R' \cup ... \cup R^{n'}) - R_1$,

$R_1 = \{R'(j'_0) \cup ... \cup R^{n'}(s^{n'}_0)\}$, $s_1 \in \{s'_0, s''_0, ..., s^{n'}_0\}$

$R = (R - \{r\}) \cup \{r'\}$

$r' = < s_{any}, s_0, i, true, \phi, 1, \phi, \phi >$ is derived from

$r = < s_{any}, s, i, true, \phi, 1, \phi, \phi > | s \in \{s'_0, ..., s_0^{n'}\}$.

There may be a transition leading to the start state of any EFSM $F', F'', ..., F^{n'}$; this transition may be due to a GOTO in the corresponding EFSM. Since the new start state is s_0 with the transitions adjusted, the GOTO must lead to s_0. In a set of alternative events, we associate a *priority* number with each event; the first event from the top has a priority 1, the second event has a priority 2, and so on. The priority number only indicates the order of checking the occurrence of an event in the execution model of TTCN. While formulating a transition for the first event of each of $A', A'', ..., A^{n'}$, we associated a priority of 1 with each of those events. In the above set of transitions for obtaining the EFSMs $F', F'', ..., F^{n'}$, we have not adjusted the priority of the transitions from s_0. The priority of the transitions from s_0 are adjusted as follows. Make the priority in the transition from s_0 to a state in F' equal to 1, the priority from s_0 to a state in F'' equal to 2, and so on. Note that from s_0 there is only one transition to each of the EFSMs $F', F'', ..., F^{n'}$.

- **OTHERWISE event**

  g?OTHERWISE [p] L V_d

  A'

Here we create a new state s_0 and add a transition from s_0 to the start state in the EFSM for A' with an event name ?OTHERWISE. Therefore, choose

$$F = < S' \cup \{s_0\}, S_t, V', R, s_0, Z', \phi, \phi, \phi > \text{ with}$$
$$R = R' \cup \{< s_0, s'_0, g?OTHERWISE, [p], \{\}, n, L, V_d >\}$$
$$\cup \{< s, s_0, i, true, \phi, n, \phi, \phi > | (s, +L) \in S'_t\}$$
$$S_t = S'_t - \{(s, +L)\}.$$

(3) Third Phase: In this phase, all the tree attachments are resolved. Let T_x be a tree containing an attach $+T_y$ and let F' and F'' be the EFSMs corresponding to T_x and T_y, respectively. Let,

$$F' = < S', S'_t, V', R', s'_0, Z', h'_0, C'_I, C'_O >,$$

$F'' = < S'', S_t'', V'', R'', s_0'', Z'', h_0'', C_I'', C_O'' >$ and

$(j, +T_y) \in S_t'$, where S_t' is the extension set in F'.

Define PF = $\{s | (r \in R'') \wedge (From(r) = s) \wedge (To(r) = s') \wedge (R''(s') = \phi)\}$. In the EFSM for T_y, PF denotes the set of all states from which there are transitions to *terminating* states.

Define Z = $\{s | R''(s) = \phi\}$, the set of *terminating* states in the EFSM for T_y. Then,

$$R = R' \cup \{< j, s', a, p, f, n, L, V_d > | < j'', j', a, p, f_1, n, L, V_d > \in R''\}$$
$$\cup R_1 - \{R'(s) | A(s, j') = +T_y\}$$
$$R_1 = \{< s'', s', a, p, f, n, L, V_d > | (< s', s_f, a, p, f_1, n, L, V_d > \in R'') \wedge (j'' \in PF) \wedge (s_f \in Z) \wedge (A(j, s') = +T_y)\}$$
$$S_t = S_t' - \{(s, +T_y)\}, \; f = f_1 \circ \Theta(j, s') | A(j, s') = +T_y.$$

Tree attachments are done according to the guidelines in [ISO 9646]. In a tree attachment, parameters may be passed; this parameter passing is expressed as a function and is associated with the transitions constructed while translating a tree attachment.

(4) Fourth Phase: In the fourth phase each transition r = $< s, s', a, p, f, n, l, V_d >$ is transformed to a transition $< s, s', a, p, f, n >$ by dropping the *label* clause and making the verdict clause V_d a tag of the state s' in the form of (s', V_d).

(5) Fifth Phase: In the fifth phase, each ASP/PDU event is represented as an I/O diagram as discussed in the following section.

2.4 Input/Output Diagrams

In the layered OSI communication architecture [ISO 7498], two protocol entities communicate through the exchange of *events*, called Abstract Service Primitives (ASPs) and Protocol Data Units (PDUs), at the service boundary between them. If data in two communicating entities are defined using different data definition techniques, then it is not possible to interpret a received event in a communicating entity. It is rather natural than an exception to use different data definition techniques

while specifying a communication protocol in LOTOS/Estelle/SDL, a test case in TTCN, and a Test Management Protocol in a semi-formal manner using structured data types.

Therefore, it is important that the syntax of and the naming conventions used in the events are interpreted in a unified manner. In this section we present a notation, called **Input/Output Diagrams (IOD)**, which is used to represent an event exchanged between two communicating entities. The concept of an I/O diagram was first introduced by Jackson in the context of structured programming [JACK 83].

The I/O diagram notation is selected as a means of representing a communication event because of its ability to represent a variety of attributes of the parameters in the event, such as *tree structure* of ASP parameters representing *composite* data types, grouping of parameters using *sequence* and *set* semantics, *choice* of a parameter in a set of alternatives, *repetitive* nature of the same parameter, *optional/mandatory* presence of some parameters in an event, and *default* values of some parameters.

To represent a communication event as an I/O diagram, we define two primitive building blocks and a notation to combine them to describe a complete ASP/PDU. An I/O diagram takes a tree structure using two types of nodes: *internal node* and *leaf node*. The root node of a tree is also treated as an internal node. An internal node, shown in Fig. 2.15(a), contains three fields: *name, type,* and *tag*, which can take possible values {optional, mandatory, default, choice, set, sequence}. The *name* and *type* fields represent the name and type of a parameter field. The *tag* field represents various attributes, as stated above, of the data field. A leaf node has only one field to contain the value of a type stated in its parent internal node.

An event parameter can be either a *primitive* type or a *composite* type. Examples of a primitive type are integer type, boolean type, bit string type, etc. A composite type may contain more than one primitive types or a combination of primitive and composite types. Therefore, an internal node representing a primitive type has only

one child successor node which is a leaf node, whereas an internal node representing a composite type has at least two internal successor nodes or more than one successor leaf node.

Figure 2.15 (a) Structure of an internal node and (b) structure of a leaf node.

2.4.1 I/O Diagram Representation of Events in ASN.1 Notation

The Abstract Syntax Notation One (ASN.1) is a data definition language widely used in defining the ASPs and PDUs in protocol specifications. To construct composite data types from primitive types, ASN.1 provides a set of operators **CHOICE, SEQUENCE, SEQUENCE OF, SET,** and **SET OF** with **OPTIONAL** and **DEFAULT** attributes. In the following, we illustrate the representation of each composite data type constructed using the above operators.

CHOICE: A CHOICE operator is represented as a tree structure in which the root node denotes the type of the composite object and each child node has a *choice* tag. The leaf nodes contain the values of the individual primitive types. For example, if a data *asd* is of type *Assoc_src_diag* defined as

```
Assoc_src_diag ::= CHOICE
  { s_user INTEGER {null(0), ...,
                 called_AE_invalid_not_recognized(10)}
    s_provider INTEGER {null(0), no_reason_given (1)}
  },
```

then the corresponding I/O diagram can be represented as shown in Fig. 2.16.

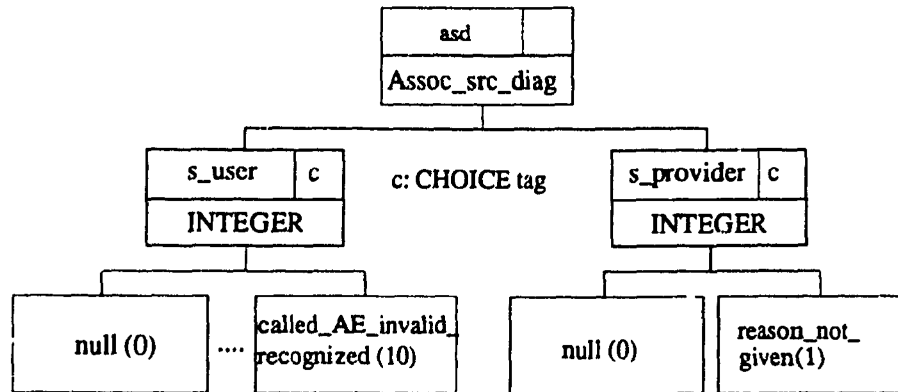


Figure 2.16 An example of representing a CHOICE construct as an I/O diagram.

SEQUENCE: In the following, we give an example of representing a SEQUENCE construct with OPTIONAL and DEFAULT attributes. If *aare* is an ASP of type *AARE_apdu* defined as

```
AARE_apdu ::= SEQUENCE
 {protocol_version BITSTRING {version1(0)}
                                    DEFAULT version1
   result_src_diag  Assoc_src_diag
   responding_AP_title  AP_title OPTIONAL
 },
```

then the corresponding I/O diagram can be represented as shown in Fig. 2.17.

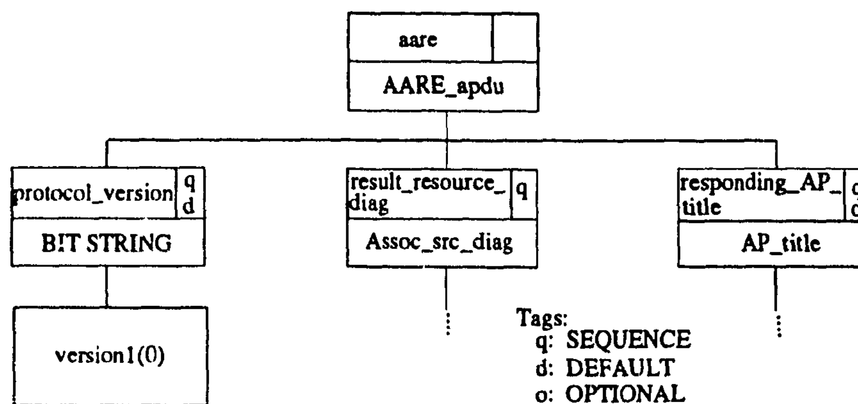


Figure 2.17 An example of representing a SEQUENCE construct as an I/O diagram.

All the children nodes belonging to the same parent node and having a SEQUENCE tag denoted by a q form a SEQUENCE from left to right in the I/O diagram.

SET: The I/O diagram corresponding to a composite data type formed by a SET operator is similar to that for a SEQUENCE operator except that an s-tag denoting a SET is used in place of a q-tag. Semantically, all the child nodes belonging to the same parent node and having an s-tag form a set with no consequence of their ordering.

SEQUENCE OF: The SEQUENCE OF construct is similar to a SEQUENCE construct except that the types in a SEQUENCE construct may be different from one another, whereas they are all of the same type in a SEQUENCE OF construct.

SET OF: The SET OF construct is similar to a SET construct except that the types in a SET construct could be different from one another, whereas they are all of the same type in a SET OF construct.

2.4.2 I/O Diagram Representation of Events in Estelle

In an Estelle specification of a protocol in a layered architecture, each channel conventionally has two roles: *service user* and *service provider*. Therefore, a channel definition contains two sets of external events, such that one set contains the events generated by the *service user* and the other set contains events generated by the *service provider*. In Fig. 2.18, we present an I/O diagram corresponding to the *ConnectReq* event in the following channel definition between a transport protocol entity and a network protocol entity.

```
channel {definition for the N-layer SAP}
        N_SapDef(User_role, Provider_role);
        by User_role:
           ConnectReq(ToAddress:   AddressType;
                      FromAddress: AddressType;
                      Qos:         QualityType;
                      Data:        DataType);
```

```
     :
     ExDataReq(Data:           DataType);
  by Provider_role:
     ConnectInd(ToAddress:     AddressType;
                FromAddress:   AddressType:
                Qos:           QualityType;
                Data:          DataType);
     :
     ExDataInd(Data:           DataType);
```

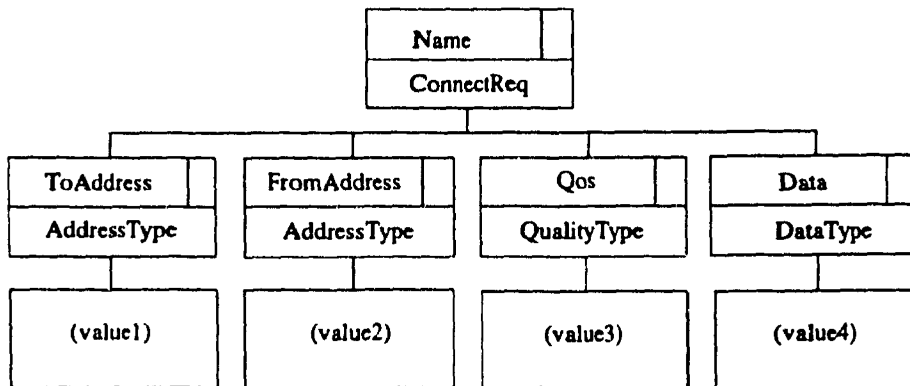


Figure 2.18 An Estelle ConnectReq event as an I/O diagram.

2.4.3 I/O Diagram Representation of Events in TTCN

ASPs and PDUs constitute events in a TTCN test case specification. There are two ways of specifying a TTCN event: *ASN.1* definition and *tabular* definition. A TTCN event specified as an ASN.1 type can be expressed as an I/O diagram as discussed in Section 2.4.1. In the following, we discuss the presentation of a TTCN event specified in a tabular notation.

For each table representing a TTCN event, an I/O diagram is generated. The name and type of the event are used to construct the root of the I/O diagram and each parameter/field in the table is used to construct the second level of internal nodes. The value of each parameter/field is used to construct a leaf node. If a tabular event contains a parameter type that is defined as another table, then an I/O diagram

representing the parameter type is attached to the parent I/O diagram. In this way, I/O diagrams of arbitrary depths can be constructed.

For example, the ASP CONreq defined in Fig. 2.19 can be represented as an I/O diagram shown in Fig. 2.20. While sending the CONreq event, the value fields value1, value2, and value3 are assigned with proper values by the sender.

ASP Type Definition		
ASP Name : CONreq (T-CONNECTrequest) **PCO Type :** TSAP **Comments :**		
Parameter Name	**Parameter Type**	**Comments**
Cda (Called Address)	CDA	... of upper tester
Cga (Calling Address)	CGA	... of lower tester
Qos (Quality of Service)	QOS	
Detailed Comments: Service primitive to be sent at Transport service access point.		

Figure 2.19 A CONreq ASP declaration in tabular notation.

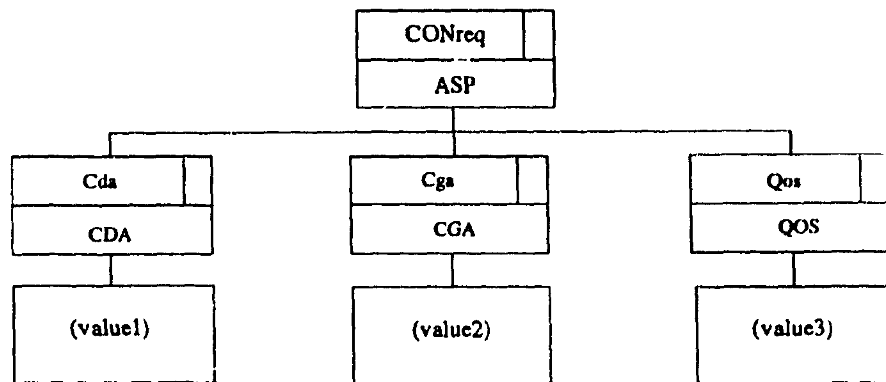


Figure 2.20 I/O diagram representation of the CONreq ASP in Fig. 2.19.

2.5 Complexity Analysis

While analyzing the complexity of an EFSM generated from a formal protocol specification or a test case, it is essential to take the syntax and semantics of the various constructs in the corresponding specification languages. In the following, we analyze the complexity of the EFSMs in terms of the order of transitions generated from TTCN and Estelle specifications.

2.5.1 Complexity for a TTCN Test Case

In the algorithm for translating a TTCN test case to an EFSM, initially, each TTCN event is translated to a transition and each tree behavior is translated to an EFSM. That is, a tree behavior with n event lines is translated to an EFSM with n transitions. The number of states depends on the sequential and alternative behavior in the tree. If there are N trees in a test case with $n_1, n_2, ..., n_N$ event lines, then the second phase of Algorithm 2.1 generates N EFSMs with $n_1, n_2, ..., n_N$ transitions, respectively.

In a test case T, for each attachment of a tree T_i generating an EFSM M_i, the algorithm creates an instance of M_i. That is, if there are k_i attachments of T_i, then k_i instances of M_i are present in the EFSM M representing the entire test case T. Therefore, in a test case specification with N trees with n_i event lines in tree T_i and k_i attachments of T_i, the total number of transitions in M is of the order $\sum_{i=1}^{N} k_i.n_i$.

2.5.2 Complexity for an Estelle Specification

Deriving the total number of transitions from an Estelle specification is more cumbersome. The difference between an Estelle transition (E-transition) and a transition defined in Section 2.1, called a simple transition (S-transition), is in terms of granularity of control-flow and computation. A S-transition is simple in the sense that the function clause of the transition is a sequence of assignment functions, whereas an E-transition is more complex in the sense that its transition block represented

by a sequence of Pascal statements enclosed in **begin...end**, may contain all types of control-flow statements and additional statements for creating, destroying, and interconnecting task modules.

To compute the complexity of transitions generated from an Estelle specification, it is necessary to take the following into account.

- Generation of transition-EFSMs from E-transitions to create a module-EFSM for each module.
- Creation of instances of modules.

To generate a transition-EFSM from each Estelle transition, here we explain the complexity of generating transitions from each Estelle construct. An S-transition, or simply transition, is created from each assignment, output, and procedure call statement. For a "while condition do statement" block, three transitions in addition to the EFSM for the "statement" block are generated. For a repeat statement block, two transitions in addition to the EFSM for the statement sequence of the repeat block are generated. An "if ... else" block generates two transitions in addition to the EFSMs for the statement blocks corresponding to the "if" part and the "else" part. A "for" statement block, in addition to the transitions in the statement part, generates one transition plus twice the number of final states in the EFSM corresponding to the "statement" part. A "case" statement with n choices generates n transitions in addition to the EFSM corresponding to each statement block associated with each choice in the case statement.

A complete Estelle transition generates as many transitions as there are in the EFSM corresponding to its **begin...end** block plus as many transitions as there are final states in the same EFSM plus one transition to take care of its input event.

While combining the modules, the "create" statements are taken into account. A specification with N modules generates N module-EFSMs. Therefore, in a specification with N modules with n_i transitions in module-EFSM M_i with k_i

instances, the order of the transitions in the EFSM generated from the specification is $O(\prod_{i=1}^{N} k_i.n_i)$. The product is due to the concurrency among the task modules in an Estelle specification.

CHAPTER 3

TEST VERIFICATION SYSTEM

To test a protocol implementation for conformance with its specification, a test case sends a sequence of input events to the implementation and observes the responses of the implementation at well-defined points known as Points of Control and Observation (PCOs). In the context of OSI's layered protocol configuration, a test system can be designed with the ability to control and observe test events at two PCOs: one at the upper and the other at the lower service boundary of the Implementation Under Test (IUT). However, practical considerations affect the PCOs in the following two ways.

- Control and observation of events at the lower service boundary may be done at a point away from the implementation.
- The test case may not control and observe test events at the upper service boundary in an active manner, thereby delegating the control and observation functions to a passive test entity.

The placement and number of PCOs in a test system give rise to the notion of *test architectures* in the ISO conformance testing framework [ISO 9646].

To verify the properties of a test case against a reference protocol specification, it is essential to study the relationship of the test case to the reference protocol specification in terms of the test architectural framework.

The organization of this chapter is as follows. In the first section, we discuss and compare various test architectures. The process of deriving a test verification system from a given test architecture is presented in the second section. Algorithms to generate a global state space from a test verification system and a model for the test verification system from the global state space are presented in the third section. The fourth section contains complexity analysis of the algorithms.

3.1 Test Architectures

Depending on the proximity of the PCOs to an IUT, there are two broad classes of test architectures: *local* and *external.* Local test architectures are characterized by control and observation being specified in terms of events occurring at the layer boundaries immediately above and below the IUT. External test architectures, on the other hand, are characterized by the control and observation of test events taking place externally from the IUT, on the other side of the underlying service provider from the IUT. The local test architectures are only applicable to in-house testing by developers of implementations, whereas the need for external test architectures arises due to the fact that conformance testing can also be the responsibility of users or national/international organizations that undertake testing activities in a centralized or distributed manner. The external test architectures have the advantage of closely resembling a realistic communication environment.

In external test architectures, stimulation and observation functions are distributed between local and external sites. The testing entity on the remote site is called the Lower Tester (LT). It controls and observes ASPs below the IUT. Locally there is no control and observation point on the (*N-1*) service boundary while testing an (*N*)-entity implementation. However, direct or indirect control and observation of the ASPs above the IUT are done through the use of an Upper Tester (UT).

The test architectures come in three variant forms: *single-layer*, *multi-layer*, and *embedded.* Single-layer architectures are designed for testing one layer of a protocol implementation at a time, without depending on the functionalities of the layer under test. Multi-layer architectures are designed for testing a multi-layer implementation as a whole. Embedded architectures are designed for testing a single layer implementation within a multi-layer implementation stack, using the knowledge of what protocols are implemented in the layers above the layer being tested.

In practice, an implementation of a protocol specification can support both single

connection and simultaneous multiple connections. Therefore, test cases must be designed to test both single and multiple connection capabilities of an implementation.

In this thesis, we study the verification of only the single-layer test architecture based single and multiple connection test cases. In the following, we present the structural details of four single connection single-layer test architectures. Verification issues related to parallel test cases with multiple connection establishment capabilities are discussed in Chapter 7.

3.1.1 Local Single-layer Test Architecture

The Local Single-layer (LS) test architecture, shown in Fig. 3.1, defines the points of control and observation as being at the service boundaries above and below the *(N)*-entity under test. The test events are specified in terms of *(N)*-ASPs above the IUT and *(N-1)*-ASPs and *(N)*-PDUs below the IUT. In an abstract sense, a Lower Tester is considered to observe and control the *(N-1)*-ASPs and *(N)*-PDUs while an Upper Tester observes and controls the *(N)*-ASPs.

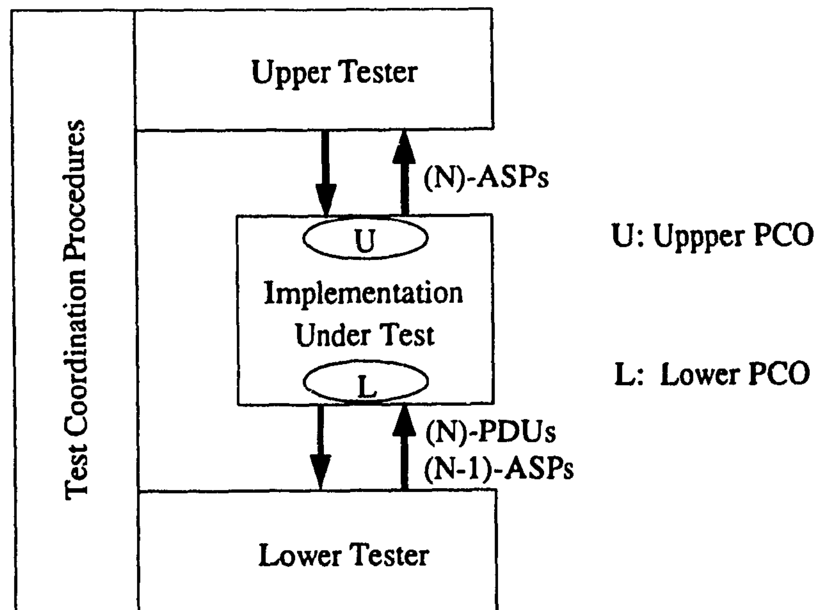


Figure 3.1 The Local Single-layer Test Architecture.

An LS architecture based test case, in principle, can take two kinds of structures. On one hand, a test case behavior can consist of two separate behaviors representing upper tester behavior and lower tester behavior, such that both the behaviors run concurrently, with a test coordination procedure synchronizing their activities. On the other hand, due to the local nature of testing, both the upper and lower tester behavior can be combined into a single behavior without requiring any test coordination procedure.

Because of the fact that the tester has full control and observation of all the external access points of the implementation under test, the LS architecture has a complete error detection ability.

3.1.2 Distributed Single-layer Test Architecture

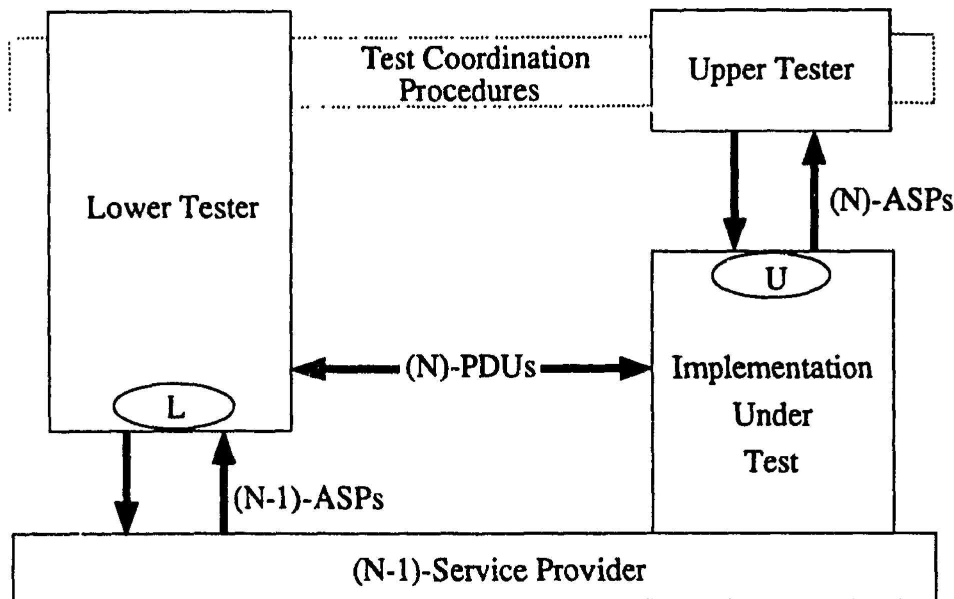


Figure 3.2 The Distributed Single-layer Test Architecture.

The Distributed Single-layer (DS) test architecture, shown in Fig. 3.2, defines the points of control and observation as being at the service boundaries above the (*N*)-entity under test and at the opposite side of the (*N-1*)-Service Provider from the

(*N*)-entity under test. The test events are specified in terms of (*N*)-ASPs above the IUT and (*N-1*)-ASPs and (*N*)-PDUs remotely.

In DS architecture, direct control and observation of (N)-ASPs are done with an UT and control and observation of events at the lower service boundary of the IUT are done in an indirect manner by a LT through the use of an underlying service provider. A given test case is applied by both LT and UT running independently.

It has been observed that some DS architecture based test cases may result in *synchronization* problems between the LT and the UT [SABO 84]. The synchronization problem arises when, in a given test case, the LT (UT) is expected to send an ASP/PDU while in the previous step the LT (UT) had not received/sent any ASP/PDU. From a test design point of view, the synchronization problem is avoided by adding one or more test events from the point where the problem arises.

Because of the concurrent executions of the LT and the UT, there is a need to coordinate the actions of the two test entities in order to achieve the objective of a test case. Such a coordination can be done by *test coordination procedures*. There are various ways of designing the test coordination: defining a test management protocol between the LT and the UT [ISO 9646], integrating the coordination into individual tests, which are executed by the UT and the LT [BOCE 83], and using an *astride responder architecture* where the UT design is simplified by establishing an extra (N)-connection and implementing the UT functions on the LT site [RACA 85].

3.1.3 Coordinated Single-layer Test Architecture

The Coordinated Single-layer (CS) test architecture, shown in Fig. 3.3, is an enhanced version of the DS method, using a standardized upper tester and the definition of a Test Management Protocol (TMP) to realize the test coordination procedures between the UT and LT. There is only one point of control and observation at the opposite side of the (N-1)-Service Provider from the (N)-entity under test. Test

events are specified in terms of (N-1)-ASPs, (N)-PDUs, and Test Management PDUs (TMPDUs).

The Lower Tester in CS architecture can control and observe the behavior of an implementation at its upper service boundary by deterministically controlling the activities of the UT through test management PDUs.

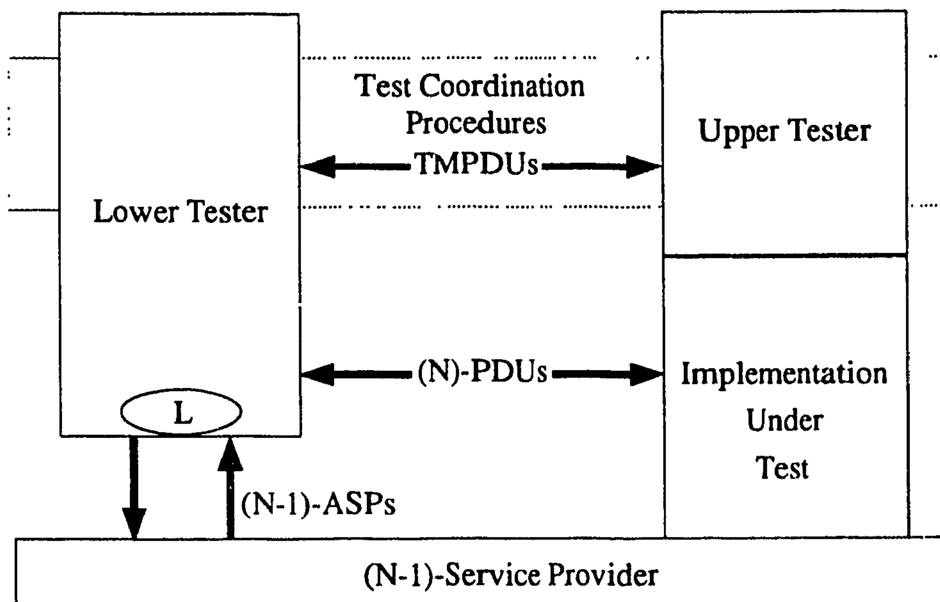


Figure 3.3 The Coordinated Single-layer Test Architecture.

3.1.4 Remote Single-layer Test Architecture

The Remote Single-layer (RS) test architecture, shown in Fig. 3.4, defines the point of control and observation as being on the opposite side of the (N-1)-Service Provider from the (N)-entity under test. The test events are specified in terms of the (N-1)-ASPs and (N)-PDUs remotely. In the RS architecture, there is no well-defined upper tester as denoted by the dotted box in Fig. 3.4.

Because of the absence of the UT, no expected input/output behavior at the upper boundary of the IUT can be specified as a part of a test case. TTCN provides a facility, called an *implicit send event*, to specify an event to be sent by the IUT to

the LT. The implicit test entity UT, shown by the dotted box in Fig. 3.4, has the role of providing stimuli to the IUT such that the IUT generates the events stated in the implicit send events at the appropriate instants of test execution.

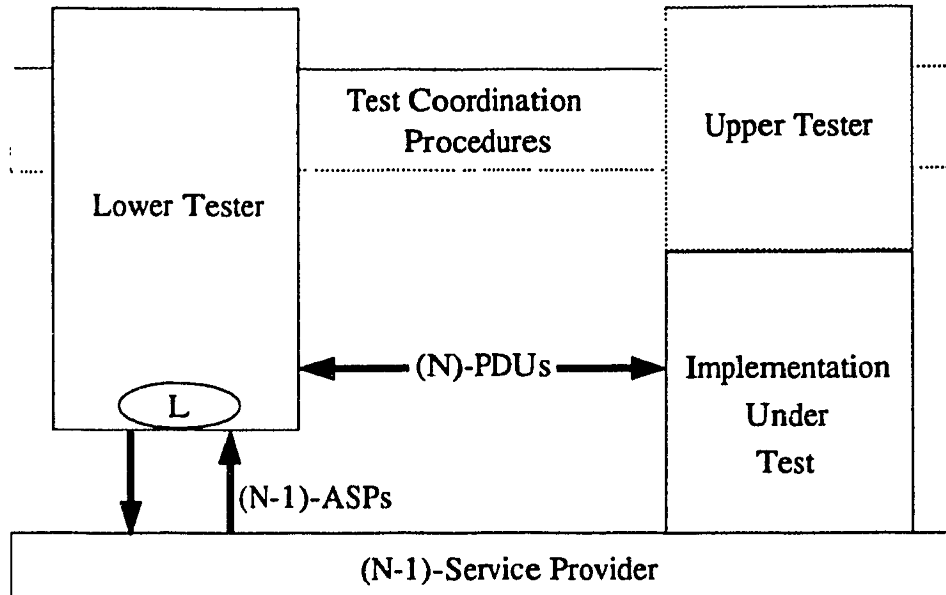


Figure 3.4 The Remote Single-layer Test Architecture.

3.1.5 Comparison of LS, DS, CS, and RS Architectures

The effect of the structural relationship among the test entities, the control and observation points, and the implementation under test on the error detection ability of a test architecture has been studied in [SARI 89].

In the LS architecture, due to the fact that the tester has full control and observations of all the external access points of an IUT, the LS architecture has a complete error detection capability. The restrictions in this architecture are the impossibility of exhaustive testing and the nondeterminism in protocol specifications. The second restriction means that it is not possible, for example, to deterministically force a transport protocol implementation to reduce credit or a file transfer, access,

and management protocol implementation to invoke recovery procedures from disk crashes.

The error detection ability of DS architecture is reduced compared to the LS architecture, since the (*N-1*)-ASPs are controlled and observed externally over a (*N-1*)-service of the underlying service provider. The external control and observation procedure has the disadvantage that certain inputs cannot be applied to an IUT. For example, the effect of a network reset on a transport protocol implementation cannot be tested since a network reset cannot be generated. Compared with other external architectures, the DS architecture has improved error detection ability due to the fact that a part of a test case runs as the UT.

A CS architecture based test case consists of only one behavior, that is, the LT behavior. Since observation and control of the activities at the upper service boundary of the IUT are indirectly done through the use of TMPDUs flowing between the UT and the LT through a possibly faulty IUT, the error detection ability of CS architecture is further reduced compared to that of DS architecture.

The RS architecture offers the most limited error detection ability because of the fact that activities at the upper service boundary of an IUT cannot be controlled and observed.

3.2 Test Verification System

In this section, we define a generic test verification system that is applicable to single connection test cases. Conceptually, a test case, designed to test implementations of a protocol specification in a given test architecture, is verified for correctness by comparing the behavior of the test case with the behavior of the protocol specification in the same architectural framework. Incorporation of the test architectural framework in the verification process is essential in the sense that it is the test architecture that defines the behavioral relationship of a test case with an implementation. Formally, a *Test Verification System (TVS)* is defined as follows.

Definition 3.1: A *Test Verification System (TVS)* is defined to be a 5-tuple, $TVS = < \Sigma, \Omega, P, \Psi, C >$, where Σ is an EFSM corresponding to the Lower Tester (LT-EFSM), Ω is an EFSM corresponding to the underlying service provider (USP-EFSM), P is an EFSM corresponding to the protocol specification (S-EFSM), Ψ is an EFSM corresponding to the Upper Tester (UT-EFSM), and C is a set of *channel functions* defining the interconnections among Σ, Ω, Ψ, and P.

A channel function *channel(EFSM1, EFSM2)* denotes that *EFSM1* outputs messages to the *channel*, which are received by *EFSM2*. In general, we denote a test EFSM by *T EFSM* while referring to either the Lower Tester or the Upper Tester EFSM. Corresponding to a test architecture, we derive a test verification system as follows.

i Replace the IUT module by the EFSM representation of the corresponding protocol specification.

ii Replace the LT module by the EFSM representation of the lower tester part of the test case. An EFSM can be generated from the TTCN specification of a test case using Algorithm 2.2.

iii For a given test architecture, replace the UT module in the following manner.

  a For LS and DS test architectures, replace the UT module by the EFSM representation of the TTCN specification of the upper tester part of the test case.

  b For CS architecture, replace the UT module by the EFSM representation of the Test Management Protocol (TMP) specification.

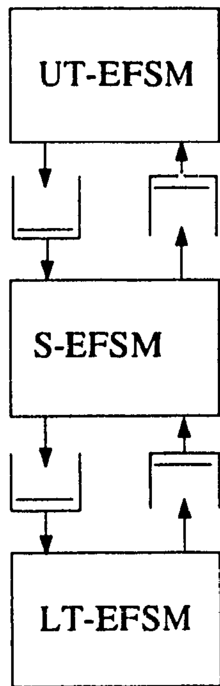
  c For RS architecture, the behavior of the UT module is dynamically generated during the model generation process. Initially, the UT-EFSM consists of only one state s_0 and one transition $r = < s_0, s_0, U?OTH, true, \{\}, 1 >$. Implicit send events in conjunction with the behavior of the S-EFSM are used to update the UT-EFSM while generating a global state space from the

verification system.

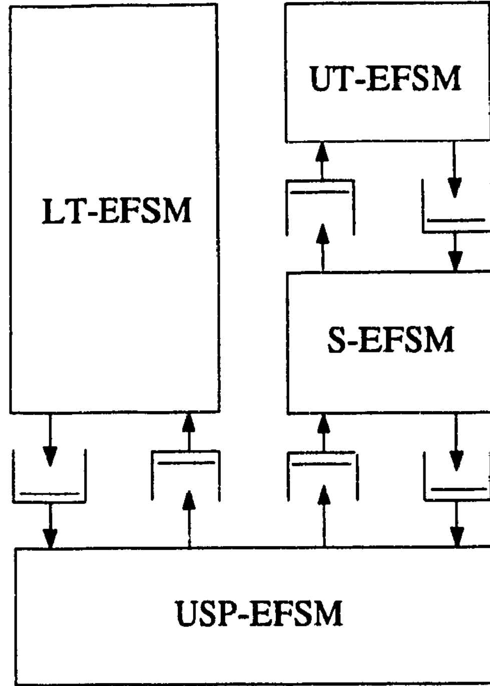
iv Replace the underlying service provider module by its EFSM representation, that is, by its input/output behavior.

v Replace each interaction point and PCO between two modules in the test architecture by two unidirectional FIFO channels.

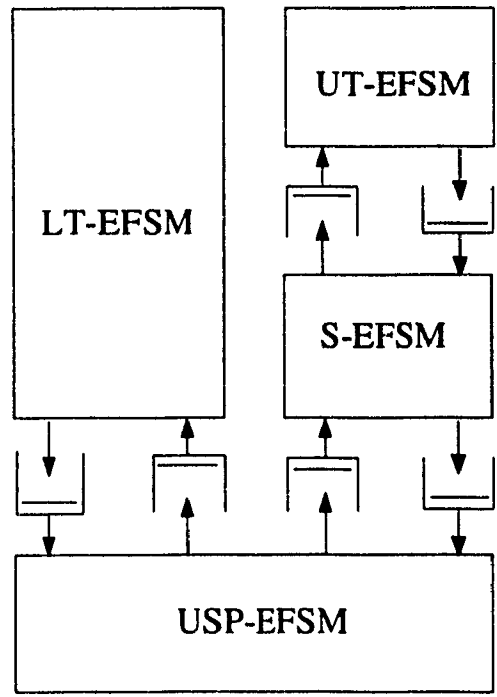
Test verification systems corresponding to the four basic test architectures, LS, DS, CS, and RS, are shown in Fig. 3.5 and a methodology to verify test cases belonging to those architectures is presented in Chapter 4. A parallel test architecture and verification of parallel test cases are discussed in Chapter 7.



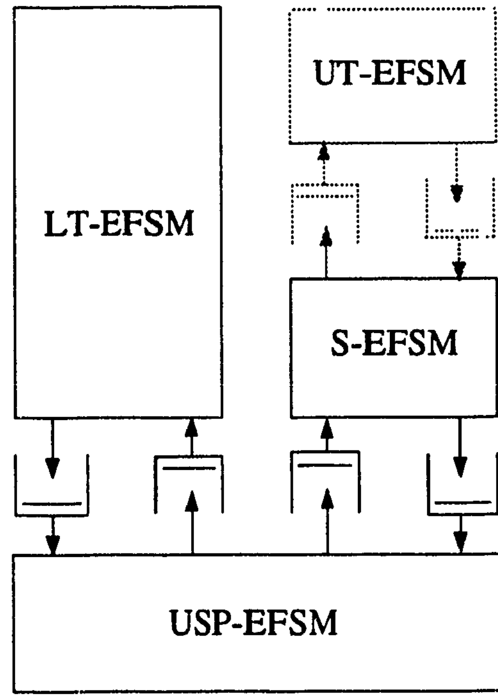
(a) TVS for LS architecture



(b) TVS for DS architecture



(c) TVS for CS architecture



(d) TVS for RS architecture

Figure 3.5 Test Verification Systems corresponding to LS, DS, CS, and RS architectures.

3.3 Model Generation

In this section, we generate a model for a test verification system. A *model* for a test verification system consists of the global state space of the verification system with a set of atomic predicates associated with each state. The model generation process consists of the following two steps.

- First, a global state space is generated from a test verification system by using a reachability analysis algorithm.
- Second, a model is generated from the global state space by associating a set of atomic predicates with each state such that the predicates hold in the state.

In the following, we discuss the issues related to both the steps in the model generation process.

3.3.1 Reachability Analysis Algorithm

We generate a global state space representing the composed behavior of a test verification system by using a reachability analysis algorithm. The reachability analysis algorithm is based on the well-known concept of *state perturbation* [WEST 78] using all the *executable transitions* in the component machines' present states. A transition in the present state of a machine is said to be executable if the *enabling condition* of the transition *evaluates* to true.

Therefore, before presenting the global state space generation algorithm, we define a global state, a procedure to compute all the executable transitions in a global state, a procedure to evaluate a transition's enabling predicate, and a global state perturbation process to generate all the reachable global states from a given state.

Definition 3.2: The global state s of a test verification system $TVS = <\Sigma, \Omega, P, \Psi, C>$ is defined as a 6-tuple $<\Sigma_s, \Omega_s, P_s, \Psi_s, C_s, \Pi \cup \upsilon>$, where Σ_s, Ω_s, P_s, and Ψ_s represent the present states of Σ, Ω, P, and Ψ, respectively, and C_s is a set of states consisting of the present states of each channel in C; Π is the set containing

values of all the variables of the EFSMs in the TVS, and v is a verdict variable assumed to be unique.

In a global state space, the states are connected by global transitions. A global transition in the transition space R is a four-tuple <s, s', E, e> where s and s' are present and next states, respectively, E is the event that triggers the transition, and e is an enabling predicate.

The initial global state s_0 is defined as follows:

$<s_{init}(\Sigma), s_{init}(\Omega), s_{init}(P), s_{init}(\Psi), C_{empty}, init(\Pi) \cup v = Null>$,

where $s_{init}(\Sigma)$ is the initial state of Σ, $s_{init}(\Omega)$ is the initial state of Ω, $s_{init}(P)$ is the initial state of the specification entity P, $s_{init}(\Psi)$ is the initial state of Ψ, C_{empty} denotes all the channels in C to be empty, and $init(\Pi) = h_0(\Sigma) \cup h_0(\Omega) \cup h_0(P) \cup h_0(\Psi)$, where the function h_0 denotes initial assignments to the variables in the corresponding EFSM. Notationally, the present state of an EFSM, M, is denoted by the function notation *ps(M)*.

Definition 3.3: The set of executable transitions *XT(s)*, occurring in the present global state $s = < \Sigma_s, \Omega_s, P_s, \Psi_s, C_s, \Pi \cup v >$ of a $TVS = < \Sigma, \Omega, P, \Psi, C >$ is given as follows:$XT(s) = XTP(s) \cup XTUSP(s) \cup XTLT(s) \cup XTUT(s)$, where

$$XTP(s) = \{r = < From, To, E, e, h, n > |(from(r) = P_s) \wedge$$
$$((int(E) \wedge eval(s, \phi, e)) \vee$$
$$(ext(E) \wedge (dir(E) =!) \wedge eval(s, \phi, e)) \vee$$
$$(ext(E) \wedge (dir(E) =?) \wedge (msg(E) = head(channel(E)) \wedge$$
$$eval(s, head(channel(E)), e))))$$
$$\}.$$

$$XTUSP(s) = \{r = < From, To, E, e, h, n > |(from(r) = \Omega_s) \wedge$$
$$((int(E) \wedge eval(s, \phi, e)) \vee$$
$$(ext(E) \wedge (dir(E) =!) \wedge eval(s, \phi, e)) \vee$$
$$(ext(E) \wedge (dir(E) =?) \wedge (msg(E) = head(channel(E)) \wedge$$

$eval(s, head(channel(E)), \epsilon))))$

}.

Initially, $XTLT(s) := \phi$, R = {r| from(r) = $ps(\Sigma)$}, init_priority := 0, $Flag(c_j) :=$ False $\forall c_j \in C$.

While $R \neq \phi$ begin {

init_priority := init_priority + 1

for r ∈ R | (priority(r) = init_priority) do {

if $((int(E) \wedge eval(s, \phi, e))$ then

$XTLT(s) := XTLT(s) \cup \{r\}, R := R - \{r\}$

if $(ext(E) \wedge (dir(E) =!) \wedge eval(s, \phi, e))$ then

$XTLT(s) := XTLT(s) \cup \{r\}, R := R - \{r\}$

if $(ext(E) \wedge (dir(E) =?) \wedge (msg(E) == head(channel(E))) \wedge$

$(flag(channel(E) = False) \wedge eval(s, head(channel(E)), e)))$

then

$\{XTLT(s) := XTLT(s) \cup \{r\}, R := R - \{r\},$

$Flag(channel(E)) := True\}$

if $(ext(E) \wedge (dir(E) =?) \wedge (msg(E) == OTH) \wedge$

$(content(channel(E)) \neq \phi) \wedge (flag(channel(E) = False) \wedge$

$eval(s, head(channel(E)), e)))$ then

$\{XTLT(s) := XTLT(s) \cup \{r\}, R := R - \{r\},$

$Flag(channel(E)) := True\}$

}

}

$XTUT(s)$ is computed the same way as $XTLT(s)$ is computed.

Here TTCN *TIMEOUT* events translated as *internal (timeout)* transitions in the test EFSMs are treated qualitatively by assuming that a timeout can occur any time without referring to its quantitative value. The advantage of such a treatment is that all possible effects of the timeout transitions can be studied. Also, TTCN *OTHERWISE* events translated as *OTH* transitions in T-EFSMs are selected only if none of the events of higher priority alternatives match with the first event in the channel.

3.3.1.1 Predicate Evaluation

The procedure *eval* used in computing the set of executable transitions in a global state space is defined here. This procedure decides on the truth value of the enabling predicate e of a transition given a global state s and the IOD at the head of the channel.

Assume that e is expressed in conjunctive normal form such that each operand of a logical operator in e is either a boolean variable or an expression of the form op_1 *relop* op_2, which we call an elementary predicate, and *relop* is a relational operator. It is also assumed that the IOD has non-symbolic values assigned at all leaf nodes. These assumptions are essential for the procedure *eval* to always return a true or false value.

procedure eval(s, IOD, e) {

/* An IOD is a tree structured representation of an instance of an ASP or a PDU event in a FIFO channel. */

1. Execute step 2 for each elementary predicate in e. If any of the returned values are false, then exit the procedure with a false value. If all the returned values are true, then exit the procedure with a true value.
2. Evaluation of a relational operator consists of the following three cases which return a boolean result:

   a. op1 **relop** op2, where both op1 and op2 are local variables: Evaluate op1 **relop** op2 and return the boolean result.

b. op1 **relop** op2, where op1 is a field of the IOD and op2 is a constant: Traverse the tree-structured IOD and extract the value of op1, evaluate "op1 **relop** op2", and return the boolean result.

c. op1 **relop** op2, where op1 is a field of the IOD and op2 is a local variable: Traverse the tree-structured data IOD and extract the value of op1. Since op2 can be an expression in the local variables, first compute the value of op2, then compute "op1 **relop** op2", and return the boolean result.

}

3.3.1.2 Perturbation

Given the present global state and a transition, the perturbation function computes the next global state. If the transition is an implicit send in an RS architecture then the perturbation function also updates the UT-EFSM by calling the *UT_gen* procedure. In the following definition, the procedure *map_to_IOD(Event)* is assumed to transform the *Event* of type ASP or PDU to an IOD.

Definition 3.4: Let $s = < \Sigma_s, \Omega_s, P_s, \Psi_s, \{c_1, .., c_i, ..., c_n\}, \Pi \cup v >$ be the present global state of a test verification system $TVS = < \Sigma, \Omega, P, \Psi, C >$. Let XT(s) be the set of executable transitions in s. Then the perturbation of s by a transition r = <From, To, E, e, h, n> $\in$ XT(s) is written as *pert(s, r)* $= s_n$, where s_n is computed as follows.

if $((From = \Sigma_s) \wedge int(E))$ then

$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, .., c_i, .., .., c_n\}, \Pi' \cup v' >$, where

$\Pi' := h(\Pi)$ and $v' :=$ *new_verdict(v, verdict(To))*

if $((From = \Sigma_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i))) \vee$

$((msg(E) = OTH) \wedge (c_i \neq \phi))$, then

$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, .., tail(c_i), .., c_n\}, \Pi' \cup v' >$

if $((From = \Sigma_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$ then

$s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, .., c_{i_new}, .., c_n\}, \Pi' \cup v' >$, where

$c_{i_new} := append(c_i, map_to_IOD(msg(E)))$

if $((From = \Sigma_s) \wedge (dir(E) = !) \wedge (channel(E) = IUT))$ then

{ $s_n := < To, \Omega_s, P_s, \Psi_s, \{c_1, .., c_i.., c_n\}, \Pi' \cup v' >$, s.t. $c_i(\Psi, P) \in C$

Call UT_gen(s, r)}

if $((From = P_s) \wedge int(E))$, then

$s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, .., c_i, .., c_n\}, \Pi' \cup v' >$

if $((From = P_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i)))$,

then $s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, .., tail(c_i), .., c_n\}, \Pi' \cup v' >$.

if $((From = P_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then

$s_n := < \Sigma_s, \Omega_s, To, \Psi_s, \{c_1, .., c_{i_new}, .., c_n\}, \Pi' \cup v' >$, where

$c_{i_new} := append(c_i, map_to_IOD(msg(E)))$.

if $((From = \Omega_s) \wedge int(E))$, then

$s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, .., c_i, .., c_n\}, \Pi' \cup v' >$

if $((From = \Omega_s) \wedge (dir(E) =?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i)))$,

then $s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, .., tail(c_i), .., c_n\}, \Pi' \cup v' >$.

if $((From = \Omega_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then

$s_n := < \Sigma_s, To, P_s, \Psi_s, \{c_1, .., c_{i_new}, .., c_n\}, \Pi' \cup v' >$, where

$c_{i_new} := append(c_i, msg(E))$.

if $((From = \Psi_s) \wedge int(E))$, then

$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, .., c_i, .., .., c_n\}, \Pi' \cup v' >$, where

$\Pi' := h(\Pi)$ and $v' = new_verdict(v, verdict(To))$

if $(((From = \Psi_s) \wedge (dir(E) = ?) \wedge (channel(E) = c_i) \wedge (msg(E) = head(c_i))) \vee$

$((msg(E) = OTH) \wedge (c_i \neq \phi)))$, then

$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, .., tail(c_i), .., c_n\}, \Pi' \cup v' >$

if $((From = \Psi_s) \wedge (dir(E) = !) \wedge (channel(E) = c_i))$, then

$s_n := < \Sigma_s, \Omega_s, P_s, To, \{c_1, .., c_{i_new}, .., c_n\}, \Pi' \cup v' >$, where

$c_{i_new} := append(c_i, map_to_IOD(msg(E)))$.

In the TTCN specification of a test case, a test designer may associate intermediate test verdicts with expected receive events from an implementation and a final test verdict may be assigned on termination of a test case behavior. Depending on the expected responses of an implementation, different verdicts – Pass, Fail, or Inconclusive – may be assigned to receive events in a sequence of test behavior. The resultant test verdict at any point in a sequence of test behavior depends on the previous verdict and the current verdict. Therefore, to compute a resultant verdict in a state given the previous verdict and the present verdict, in the following we define a procedure *new_verdict(ov, pv)*, where *ov* is the old verdict or the previous verdict and *pv* is the present verdict.

```
procedure new_verdict(ov, pv) {
  if (ov == "none" then return(pv)
  else{
       if (ov == Fail) or (pv == Fail) then return(Fail)
       else if (ov == Pass) and (pv == Inconclusive) then return(Inconclusive)
       else if (ov == Inconc.) and (pv == Inconc.) then return(Inconclusive)
       else if (ov == Pass) and (pv == Pass) then return(Pass)
     }
}
```

The RS test architecture does not have an explicitly defined upper tester. However, while executing an RS architecture based test case, it is required to specify some behavior at the upper service boundary of the IUT. Therefore, during the global state space generation process, we dynamically generate the desired behavior of the upper tester in an incremental manner by calling the following *UT_gen()* procedure.

procedure *UT_gen(s, r_1)*{

Let $\Psi = < S, \{\}, V, R, a_0, \{a_j\}, h.C_I, C_O >$,

$s = < \Sigma_s, \Omega_s, P_s, \Psi_s, \{c_1, .., c_i, .., c_n\}, \Pi \cup v >$, and

$r_1 = < \Sigma_s, s_1, E_1, e_1, h_1, n_1 >$

1. Explore all paths from P_s until a transition $r_3 =< s_3, s'_3, E_3, e_3, h_3, n_3 >$ is encountered such that the event E_3 matches E_1.

   Let $r_2 = < s_2, s'_2, E_2, e_2, h_2, n_2 >$ be a transition on the path from P_s to s_3 such that $((dir(E_2) =?) \wedge (channel(E_2) = channel(\Psi, P)))$.

2. Generate:

   $E_n = channel(E_2)!event(E_2)$

   h_n = a set of assignments to the fields of E_n such that e_2 is satisfied and all transitions up to and including r_3 can be fired.

3. Update the UT behavior by creating a state a_{j+1} and two transitions

   $r' = < a_j, a_{j+1}, E_n, true, h_n, 1 >$, $r'' = < a_{j+1}, a_{j+1}, OTH, true, \phi, 1 >$

   $\Psi = < S \cup \{a_{j+1}\}, \phi, V \cup var(h_n), R \cup \{r', r''\}, a_0, \{a_{j+1}\}, h, C_I, C_O >$

}

In the following, we present a state space generation algorithm based on the traditional reachability analysis algorithm extended to EFSMs [WEST 78]. Our algorithm handles some special characteristics of test specifications such as nondeterminism, OTHERWISE events, and verdict computation.

Algorithm 3.1

Input: A set of EFSMs, a set of communication channels, and the capacities of the channels.

Output: A global state space.

S1. Define a set of global states S and a set of global transitions R. Initially, S contains only the initial global state s_{init} and $R = \phi$.

S2. Find a member $s \in S$ of the set of global states whose perturbations have not been determined. If no such member exists, then terminate.

S3. Calculate the set of executable transitions $XT(s)$ in state s using Definition 3.4.

S4. Compute S_p, a set of global states by perturbing s. Initially $S_p = \phi$.

$\forall r = < From, To, E, e, h, n > \in XT(s), do$

{ $S_p = S_p \cup \{s'\}$, where $s' = pert(s, r)$;

$P_r(s) = \phi$; /* $P_r(s)$ is the set of predicates being true in state s. */

$R = R \cup \{< s, s', E, e, h, n >\}$;

}

S5. If S_p is an empty set, report s as a terminal state in the global state space.

S6. $\forall s \in S_p$ do {

if $channeloverflow(s)$ then mark s "perturbed" and $S = S \cup \{s\}$

else if $s \notin S$ then mark s "unperturbed" and $S = S \cup \{s\}$

}

S7. Go to step S2.

3.3.2 Model Generation Algorithm

A model for a TVS is generated from the global state space of the TVS by associating a set of atomic predicates with each global state. Therefore, we first identify the types of predicates and then present an algorithm to systematically associate a set of predicates with each global state.

We identify five types of atomic predicates to be associated with each global state: *state predicate*, *variable predicates*, *event predicates*, *PCO predicates*, and *verdict predicate*. Identification of the predicate types is guided by the test properties to be verified. In the following, we explain the five types of predicates.

(i) **State predicate:** The state predicate INIT is associated with the initial state.

(ii) **Variable predicates:** These are assertions about the values of the variables in the global state space. These assertions arise from the enabling conditions of the transitions in the component EFSMs of a test verification system.

(iii) **Event predicates:** These characterize the possibility or the actual execution of specified events. There are two types of event predicates: AT and AFTER used with different parameters. The predicate *AT(Treceive(Channel, Event))* is true in a state s if there is a *test case* transition in state s such that the *Event* is received from the *Channel*. Similarly, *AT(Sreceive(Channel, Event))* is true in a state s if there is a *protocol* transition such that the *Event* is received from the *Channel*. *AFTER(Treceive(Channel, Event))* is true in a state s' if there is a transition to the state s' such that the *Event* is received from the *Channel* as a result of firing the transition. A similar explanation holds for *AFTER(Sreceive(Channel, Event))*.

(iv) **PCO predicates:** The direction of events in a TTCN test case is with respect to the points of control and observation (PCO). We define a set of assertions about the PCOs and the input/output directions of events occurring at the PCOs: LOWER, UPPER, INPUT, OUTPUT, LOWER_OUTPUT, UPPER_OUTPUT, INTERNAL, and NULL. The predicate LOWER (UPPER) is true in a state if a transition fires in the state with an external event occurring at the Lower (Upper) PCO. The predicate INPUT (OUTPUT) is true in a state if a transition fires in the state with an external input (output) occurring at one of the two PCOs. If the external event is output at the Lower (Upper) PCO then LOWER_OUTPUT (UPPER_OUTPUT) is true. If an internal event occurs in a state, then INTERNAL is true. If a transition containing neither an external nor an internal event, but containing some assignment functions occurs in a state, then the predicate NULL is associated with that state.

(v) **Verdict predicate:** This is an assertion about the test verdict and is one of the following three: (Verdict == Pass), (Verdict == Inconclusive), and (Verdict == Fail).

In the following, we present an algorithm to associate a set of predicates with

each global state of a test verification system. Such a predicated global state space is denoted as a *model* for the verification system.

Algorithm 3.2

Input: The state set S and the transition set R of the global state space generated by Algorithm 3.1.

Output: Predicated S.

S1. Initialization: $\forall s \in S, P_r(s) = \phi$, where P_r assigns a set of predicates to s.

S2. $\forall r = < s, s', E, e, h, n > \in R$ do {

$P_r(s) = P_r(s) \cup Pper(r)$

$P_r(s') = P_r(s') \cup Pgen(r)$

}

The functions *Pper* and *Pgen* are defined to compute the set of atomic predicates evaluated to true in a global state. The function $Pper(r)$ associates a set of predicates with a state *s* when *s* is perturbed by the executable transition *r*. Similarly, the function $Pgen(r)$ associates a set of atomic predicates with the state s' when s' is generated by perturbing the state *s* using the transition *r*.

```
Pper(r){/* r =< s, s', E, e, h, n > */
  if (ext(E) ∧ (dir(E) == !)) then
     { if (pco(E) == L) then temp = {LOWER_OUTPUT, LOWER};
       else if (pco(E) == U) then temp = {UPPER_OUTPUT, UPPER};
     }
  else if (ext(E) ∧ (dir(E) == ?)) then
     { if (pco(E) == L) then temp = {LOWER};
       else if (pco(E) == U) then temp = {UPPER};
     }
  else if (E == i) then temp = {INTERNAL};
```

```
    else temp = {NULL};
    return (temp ∪ {e})
}
Pgen(r){/* r = < s, s', E, e, h, n > */
    if (ext(E) and r is a test transition), then
        { if (dir(E) == ?) then temp = {AFTER(Treceive(pco(E), message(E)))};
          else if (dir(E) == !) then temp = {AFTER(Tsend(pco(E), message(E)))};
        }
    else if (ext(E) and r is a protocol specification transition), then
        { if (dir(E) == ?) then temp = {AFTER(Sreceive(pco(E), message(E)))};
          else if (dir(E) == !) then temp = {AFTER(Ssend(pco(E), message(E)))};
        }
    else temp = { };
    return (temp)
}
```

3.4 Complexity Analysis

In this Section, we analyze the complexities of the global state space and model generation algorithms, Algorithms 3.1 and 3.2, respectively. Let there be n EFSMs,

$$F_1 = < S_1, S_{t1}, V_1, R_1, s_1, Z_1, h_1, C_{I1}, C_{O1} >,$$

$$F_2 = < S_2, S_{t2}, V_2, R_2, s_2, Z_2, h_2, C_{I2}, C_{O2} >,$$

$$\ldots$$

$$F_n = < S_n, S_{tn}, V_n, R_n, s_n, Z_n, h_n, C_{In}, C_{On} >$$

and m channels $C_1, C_2, ..., C_m$ interconnecting them in a test verification system.

Since a global state consists of the states of the individual EFSMs and the channel states, to determine the complexity of the global state space, it is required to compute

the set of states each channel can be in. The size of the state set of channel C_i is computed as follows. Let the capacity of channel C_i be C_{ki} and let there be T_i types of messages that can be present in channel C_i. Then the number of states $S(C_i)$ of channel C_i is given as:

$$
\begin{aligned}
S(C_i) &= T_i^0 + T_i^1 + T_i^2 + ... + T_i^{C_{ki}} \\
&= \frac{T_i^{C_{ki}+1} - 1}{T_i - 1}
\end{aligned}
$$

Therefore, the state complexity of the global state space S, denoted by *SComplexity(S)*, generated from the n EFSMs $F_1, F_2, ..., F_n$ and m channels $C_1, C_2, ..., C_m$ is given as follows:

$$SComplexity(S) = (\prod_{j=1}^{n} S_j . \prod_{i=1}^{m} (\frac{T_i^{C_{ki}+1} - 1}{T_i - 1})),$$

which represents the product of all EFSM states and channel states.

The complexity of the data space, denoted by *DComplexity(S)*, attached to the global state space S depends on the data spaces of the individual EFSMs, the set of predicates attached to each global state, and the complexity of the global state space. The following expression represents the data space complexity:

$DComplexity(S) = (V_1 \cup V_2 \cup ... \cup V_n \cup P_r(S)) \times SComplexity(S),$

where $P_r(S)$ is the set of predicates associated with all the states in S.

The complexity of the model generation algorithm is linear in the size of the global state space and is equal to the space complexity *SComplexity(S)*.

CHAPTER 4

MODEL CHECKING FOR TVS

To verify the correctness of a system, one must verify that the system satisfies its *safety* and *liveness* properties. Safety properties state that something *bad* never happens and liveness properties state that something *good* eventually does happen.

This chapter contains four sections. The branching time temporal logic formalism is presented in the first section. In the second section, we define the safety properties of a test case. In the third section, we define a set of notations to formally express a test purpose as a temporal formula and a notation to express the liveness property of a test case. Algorithms to evaluate temporal formulas using various temporal operators on a model representing a test verification system are presented in the fourth section.

4.1 Temporal Logic

Temporal logic has been of particular interest to the designers of both hardware and software specifications for more than a decade in the form of verifying some well-defined properties of specifications [BOCH 82, LAMP 83]. With the widespread use of communication protocols and the subsequent global effort in standardizing the formal specifications of protocols, temporal logic is also used in verifying protocol specifications [SAB 88].

Temporal logics are extensions of the propositional logic, which include certain types of assertions about the future. A temporal formula is constructed using propositional variables, the conventional logical operators, and a set of temporal operators. The set of temporal operators in a temporal logic is defined based on the structure of time on which the temporal formalism is based. There are two main classes of temporal logics: *linear time* [LAMP 80] and *branching time* [BPM 83]. The linear time temporal logic considers time to be a linear sequence and the branching

time approach adopts a tree structured time allowing some instants to have more than a single successor instant. Both types of temporal logics are used in the verification of communication protocols [FRV 86, CES 86, SAB 88].

In the following we explain the two basic operators in linear time logic.

$\Box$ – meaning "now and for ever";

$\Diamond$ – meaning "now or sometime in the future".

The following are some examples of temporal assertions using these operators:

$(x > 0)$ - "the variable x is positive now";

$\Box(x > 0)$ – "x is positive now and forever";

$\Diamond(x > 0)$ – "x is positive now or will be positive some time in the future".

Whether to use the linear time logic or the branching time logic is pragmatically based on the types of systems and properties one wishes to formalize and study [BPM 83]. Since the global behavior of a test system containing multiple nondeterministic protocol and test entities takes a tree structure rather than a sequential structure, we use branching time temporal logic for studying the properties of a test system. In the following, we present the syntax and semantics of branching time logic.

Definition 4.1: Let AP be a set of atomic propositions. A BTL structure is defined as a 5–tuple: $M = < S, V, R, P_r, s_{init} >$, where

- S is a finite set of states.
- V is a finite set of variables.
- R is a set of transitions among the elements of S.
- $P_r : S \longrightarrow 2^{AP}$ assigns to each state the set of atomic predicates evaluating to true in that state.
- $s_{init} \in S$ is the initial state.

Definition 4.2: Using the propositional logic operators $\neg$, $\wedge$, and $\vee$ and the branching time temporal operator *Until (U)*, the formulas of a BTL structure are defined as follows.

i. Every atomic proposition $f \in AP$ is a BTL formula.

ii. If f and g are BTL formulas, then so are $\neg f$, $f \wedge g$, $f \vee g$, $A[f\ U\ g]$, and $E[f\ U\ g]$.

Definition 4.3: A *path* is a sequence of states $(s_0,\ s_1,\ s_2,\ ...)$ such that $\forall i[(s_i, s_{i+1}) \in R]$. The state s_0 need not be the initial state of a BTL structure.

We use a standard notation to express the truth value of a formula f in a BTL structure M: $(M,\ s_0 \models f)$ means that the temporal formula f holds at state s_0 in structure M [CES 86]. When the structure M is understood, we simply write $s_0 \models f$.

Definition 4.4: The semantics of the temporal operators used to construct BTL formulas are defined using the $\models$ notation as follows:

i. $s_0 \models p$ iff $p \in P_r(s_0)$.

ii. $s_0 \models \neg f$ iff $not(s_0 \models f)$.

iii. $s_0 \models f \wedge g$ iff $s_0 \models f$ and $s_0 \models g$.

iv. $s_0 \models f \vee g$ iff $s_0 \models f$ or $s_0 \models g$.

v. $s_0 \models A[f\ U\ g]$ iff for all paths $(s_0, s_1, ...)$ starting with s_0,

$$\exists i[(i \geq 0) \wedge (s_i \models g) \wedge (\forall j[0 \leq j < i \rightarrow (s_j \models f)])].$$

vi. $s_0 \models E[f\ U\ g]$ iff for some path $(s_0, s_1, ...)$ starting with s_0,

$$\exists i[(i \geq 0) \wedge (s_i \models g) \wedge (\forall j[0 \leq j < i \rightarrow (s_i \models f)])].$$

The following abbreviations are also used in writing BTL formulas:

$AF(f) \equiv A[True\ U\ f]$ means that f holds in the future along every path from s_0; that is f is *inevitable.*

$EF(f) \equiv E[True\ U\ f]$ means that there is some path from s_0 that leads to a state at which f holds; that is, f *potentially* holds.

$EG(f) \equiv \neg AF(\neg f)$ means that there is some path from s_0 on which f holds at every state.

$AG(f) \equiv \neg EF(\neg f)$ means that f holds at every state on every path from s_0; that is f holds *globally.*

$(f_1 \mapsto f_2) \equiv AG(f_1 \rightarrow AF(f_2))$ (read "f_1 leads to f_2") means that for any time at which f_1 is true, f_2 must be true then or at some later time.

4.2 Safety Properties

Based on the idea that nothing bad happens during a testing process, we group the safety properties of a test case into four distinct classes: *transmission safety, reception safety, synchronization safety,* and *verdict safety.* Each type of safety property is defined below.

Transmission safety: There are two transmission safety properties corresponding to transmission of events by the test case and transmission of events by the protocol specification.

1. $INIT \models AG(AFTER(Tsend(Q,E)) \mapsto AFTER(Sreceive(Q,E)))$ and
2. $INIT \models AG(AFTER(Ssend(Q,E)) \mapsto AFTER(Treceive(Q,E)))$

In the above formulas, Q is any communication channel between the test system and the protocol specification and E stands for an event. Intuitively, the first property states that every event sent by the test case must eventually be accepted by the protocol specification. That means the test case does not generate any event that is unacceptable to the protocol. The second property states that every event generated by the protocol specification during the testing process must eventually be accepted by the test case, i.e., the test case is ready to receive any event generated by the protocol. Satisfaction of these two properties ensures that there is no blocking reception error [ZAFI 80] in the test system.

Reception safety: There are two reception safety properties corresponding to reception of events by the test case and reception of events by the protocol specification.

1. $INIT \models AG(AT(Treceive(Q_s,E_s)) \mapsto AFTER(Treceive(Q_i,E_i)) \vee$
$AFTER(Tinternal))$

2. $INIT \models AG(AT(Sreceive(Q_s, E_s)) \mapsto AFTER(Sreceive(Q_s, E_s)) \vee$
$AFTER(Pinternal))$

In the above formulas, $AT(Treceive(Q_s, E_s)) \equiv AT(Treceive(Q_1, E_1)) \vee$

$\vdots$

$AT(Treceive(Q_n, E_n))$.

The predicate $AT(Sreceive(Q_s, E_s))$ is defined in a similar way. *Tinternal* and *Pinternal* are internal events in the test case and in the protocol specification, respectively.

Intuitively, the first (second) reception property states that when control reaches a state in the test case (protocol specification) where there is a set of alternative receive events, one receive event must be enabled or an internal event must occur in that state. Satisfaction of these two properties ensures that the test case is not deadlocked [ZAFI 80] with the protocol specification. The internal event may be due to a timeout.

Synchronization safety: This safety property is a special characteristic of protocol testing and is not found in conventional program testing. The issue of synchronization in a test case, which is an event timing problem, was first studied in [SABO 84] using deterministic FSM models of a protocol specification and a test case. This problem arises when the test system interacts with the protocol through two PCOs. Conceptually, a test case faces a synchronization problem at one of the PCOs if an output test event is preceded by a sequence of events consisting of internal protocol events and/or a test event occurring at the other PCO. The synchronization problem in a test system using the nondeterministic EFSM models of protocol specification and test case has been studied in [NASA 92b]. We express the synchronization safety properties using the *until (U)* operator as follows.

1. For every state s in the BTL structure,

   $s \models \neg E[f_1\ U\ f_2]$, where $f_1 \equiv UPPER \vee INTERNAL \vee NULL$

   $f_2 \equiv LOWER_OUTPUT$.

2. For every state s in the BTL structure,

$$s \models \neg E[f_1 \; U \; f_2], \text{ where } f_1 \equiv LOWER \vee INTERNAL \vee NULL$$
$$f_2 \equiv UPPER_OUTPUT.$$

The first (second) synchronization safety property is for the Lower (Upper) PCO.

Verdict safety: Intuitively, during the testing process a test case must not assign a *Fail* verdict to any behavior allowed by the protocol specification. Therefore, a BTL structure representing the composed behavior of a test and a protocol specification must not contain any state with the (Verdict = Fail) predicate true. Therefore, the verdict safety property is formulated as follows.

$$INIT \models AG(\neg(Verdict = Fail))$$

4.3 Liveness Property

While testing a protocol implementation using a test case, if the implementation fulfills the *test purpose*, then the test case assigns a *Pass* verdict. Therefore, the test case behavior satisfying the test purpose must end with a Pass verdict. Satisfaction of the test purpose is a good thing that must happen in a test case. Thus, a test case satisfying the liveness property means that the test behavior satisfies the test purpose and eventually assigns a Pass verdict. The liveness property is formally stated as follows.

$$INIT \models (f_1 \mapsto (Verdict = Pass)),$$

where f_1 is a temporal formula representing the test purpose.

We identify the following five primitive test purposes from which more complex test purposes can be derived by using a composition rule. The test purposes are expressed as temporal formulas using the $\mapsto$ operator defined in Section 4.1.

i. Direct Response

$$(p_s \wedge AFTER(Tsend(Q_i, E_i)) \wedge (t = T)) \mapsto$$
$$(p_r \wedge AFTER(Treceive(Q_j, E_j)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the test system sends an event E_i to the protocol through the channel Q_i at time T with the predicate p_s true, then it receives an event E_j from the protocol through the channel Q_j during an interval of length T_0 such that the predicate p_r is true.

ii. Timed Response

$$(p_s \wedge (t = T)) \mapsto (p_r \wedge AFTER(Treceive(Q_j, E_j)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the predicate p_s is true in the test system at time T and the test system waits without doing anything, then it receives an event E_j from the protocol through the channel Q_j during an interval of length T_0 such that the predicate p_r is true. This test purpose can be used to specify a conformance requirement in which the protocol outputs an event after a timeout.

iii. No Response to External Input

$$(p_s \wedge AFTER(Tsend(Q_i, E_i)) \wedge (t = T)) \mapsto$$
$$(p_s \wedge \neg AFTER(Treceive(ANY_channel, ANY_event)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the test system sends an event E_i to the protocol through the channel Q_i at time T with the predicate p_s true and waits an interval T_0, then no event is received from the protocol specification through any of the channels during the same period. This purpose is useful to model a conformance requirement in which the protocol ignores an invalid/inopportune event and does not output any event.

iv. No Response in an Interval

$$(p_s \wedge (t = T)) \mapsto$$
$$(p_s \wedge \neg AFTER(Treceive(ANY_channel, ANY_event)) \wedge (T < t \leq T + T_0))$$

This purpose states that if the predicate p_s is true in the test system at time T and the test system waits for an interval T_0, then no event is received from the protocol specification through any of the channels during the same period. This purpose is

useful to model a conformance requirement in which a timer does not prematurely expire in the protocol.

v. Eventual Response

$$(p_s \wedge (t = T)) \mapsto (p_r \wedge AFTER(Treceive(Q_i, E_i)) \wedge (T < t < \infty))$$

Intuitively, this purpose states that if the predicate p_s is true in the test system at time T and the test system waits indefinitely without doing anything, then it eventually receives an event E_j from the protocol through the channel Q_j such that the predicate p_r is true. This test purpose can be used to specify a conformance requirement in which the protocol nondeterministically outputs an event. Though this purpose looks like a special case of the *Timed Response* test purpose, conceptually they are different.

We define *SEQ*, a *sequential* operator for composing primitive test purposes into larger test purposes using the following syntax where f_1 and f_2 are two test purposes. $s_0 \models (f_1\ SEQ\ f_2)$ iff $s_0 \models f_1$ and $\forall s_i \in last(f_1)\ \exists s_j \in reachable(s_i)$ such that $s_j \models f_2$. Intuitively, $(f_1\ SEQ\ f_2)$ means f_2 is satisfied after f_1.

The function *reachable(s)* returns a set of states that can be reached from s.

The function *last(f)* is the set of all the terminating states of the finite partial paths from s_0, over which f is evaluated. In the expression $s_0 \models f$, the formula f is evaluated over a set of paths starting with s_0.

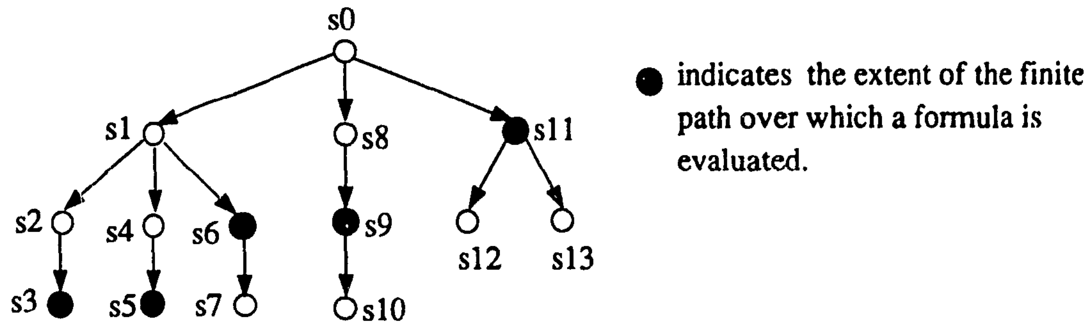


Figure 4.1 An example to compute *last(f)*.

For example, referring to Fig. 4.1, let a formula f be evaluated over all the finite paths whose extents are indicated by the bold states. Then *last(f)* = $\{s_3, s_5, s_6, s_9, s_{11}\}$.

4.4 Verification of Test Case Properties

Safety and liveness properties of a test case are verified by evaluating, also called *model checking*, the corresponding temporal formulas. The formulas used to express the test case properties contain expressions of the form: $A[f_1 \ U \ f_2]$, $f_1 \mapsto f_2$, and $f_1 \ SEQ \ f_2$. The length of a formula is an important parameter in the termination of a formula evaluation algorithm. Hence, in the following, we define *length(f)*.

Definition 4.5: Let f' be the prefix representation of a formula f. Then, the *length* of a formula f, denoted by *length(f)*, is determined by counting the total number of operands and operators in f'.

Example: Let f = *A[(NOT X) U (Y OR Z)]*. Then f' = *AU((NOT X)(OR Y Z))*. Thus, *length(f)* = 6.

Assume that we wish to evaluate whether a formula f is true in a BTL structure $M = (S, V, R, P_r, s_{init})$. The model checking algorithm operates in stages: the first stage processes all subformulas of f of length 1, the second stage processes all subformulas of length 2, and so on. At the end of the ith stage, each state will be labeled with the set of all subformulas of length less than or equal to i that are true in the state. The notation *label(s)* denotes this set for state s. Upon termination of the algorithm at the end of stage n = *length(f)*, for all states s, $M, s \models f_j$ iff $f_j \in$ label(s) for all subformulas f_j of f.

The following primitives are used to manipulate formulas and access the labels associated with states:

- *arg1(f)* and *arg2(f)* give the first and second arguments of a two-argument temporal operator; thus if f is $A[f_1 \ U \ f_2]$, then arg1(f) = f_1 and arg2(f) = f_2.
- *labeled(s, f)* returns true (false) if state s is (is not) labeled with formula f.

- *add_label(s, f)* adds formula f to the current label of state s.
- *marked[1:nstates]* is used to indicate which states have been visited by the search algorithm.
- *stacked(s)* indicates whether state s is currently on the stack ST.

In the following, we present algorithms to evaluate operators $U, \mapsto$, and SEQ.

4.4.1 Evaluation of *Until* Operator

In the following, we explain how a formula $f = A[f_1 \; U \; f_2]$ involving the *until* operator is evaluated [CES 86]. The recursive procedure *au(f, s, b)*, given below, performs the search for formula f starting from state s. When *au* terminates, the boolean result parameter b will be set to true if $s \models f$, otherwise it is set to false.

Algorithm 4.1

Input: A BTL structure, s, and $f = A[f_1 \; U \; f_2]$.

Output: True or false.

procedure *au(f, s, b)*

begin

  if marked(s) **then**

    begin

      if labeled(s, f) **then**

        begin b := *true;* **return end;**

      else b := *false;* **return**

    end

{Mark state s as visited. Let $f = A[f_1 \; U \; f_2]$. If f_2 is true at s, f is true at s; so label s with f and return true. If f_1 is not true at s, then f is not true at s; so return false. }

  marked(s) := true;

  if labeled(s, *arg2(f)*) **then**

```
        begin add_label(s, f); b := true; return end
    else if ¬labeled(s, arg1(f)) then
        begin b := false; return end;
```

{Now we know that f_1 is true at s and that f_2 is not. Check to see if f is true at all successor states of s. If there is any successor state $s1$ at which f is false, then f is false at s also; hence remove s from the stack and return false. If f is true for all successor states, then f is true at s; so remove s from the stack, label s with f, and return true.}

```
        push(s, ST);
        for all s1 ∈ successors(s) do
            begin
                au(f, s1, b1);
                if ¬ b1 then
                    begin pop(ST); b := false; return end
            end;
        pop(ST); add_label(s, f); b := true; return
    end {of procedure au}
```

4.4.2 Evaluation of "implies" Operator

Algorithm 4.2

Input: A BTL structure, s, and $f = (f_1 \mapsto f_2)$.

Output: True or False.

S1. Let the BTL structure be denoted by $< S, V, R, P_r, s_init >$. Define G, a set of states, initially containing ϕ and a boolean variable *Result* that holds the result of evaluating the formula f. Go to step S2.

S2. If $((f_1 \notin P_r(s)) \vee (f_2 \in P_r(s)))$ then go to Step S4

else **begin**

$G \leftarrow successors(s)$;

if $(G = \phi)$ then go to step S5;

else **begin** mark(s) := true; Go to step S3 **end**

end

S3. If $(G = \phi)$ then go to step S4

else for any $g \in G$ such that ¬mark(g) do

begin

If $f_2 \in P(g)$ then **begin**

$G \leftarrow (G - \{g\})$;

mark(g) := true; go to step S3

end

else if $(successors(g) = \phi)$ then go to S5.

else **begin**

$G \leftarrow (G \cup successors(g) - \{g\})$;

mark(g) := true; Go to step S3

end

end

S4. *Result := true;* Stop.

S5. *Result := false;* Stop.

4.4.3 Evaluation of SEQ Operator

Algorithm 4.3

Input: A BTL structure, s, and $f = (f_1 \; SEQ \; f_2)$.

Output: True or false.

S1. If $\neg(s \models f_1)$ then go to step S4

Else **begin**

compute $S_l = last(f_1)$;

If $S_l = \phi$ then go to S4 else go to step S2

end

S2. If $S_l = \phi$ then go to S5

else for any $s_i \in S_l$ **begin**

compute $S_r = reachable(s_i)$;

$S_l \leftarrow (S_l - \{s_i\})$;

Go to step S3

end

S3. If $S_r = \phi$ then go to S4

else for each $s_j \in S_r$ **begin**

If $s_j \models f_2$ then go to step S2

else **begin** $S_r \leftarrow S_r - \{s_j\}$; go to step S3 **end**

end

S4. *Result := False;* Stop.

S5. *Result := True;* Stop.

4.4.4 Model Checking Algorithm

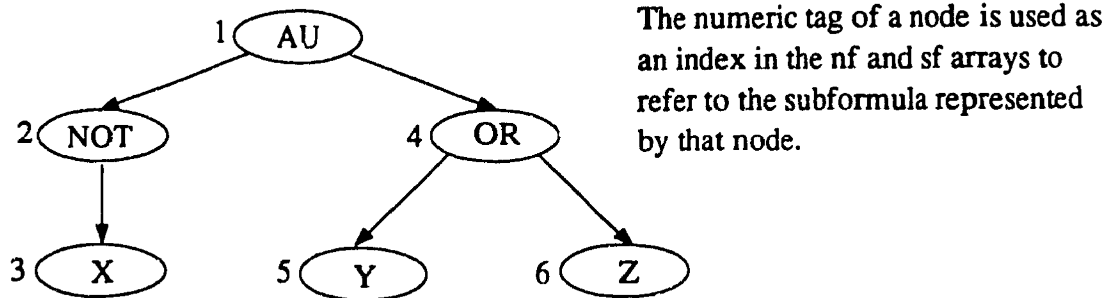
In this section we explain how to evaluate a BTL formula f with arbitrary nesting of subformulas. Conceptually, a temporal formula can be viewed as a tree with the internal nodes consisting of logical operators $\neg, \vee, \wedge$ and temporal operators $U, \mapsto, SEQ$, and the leaf nodes consisting of atomic predicates. Therefore, evaluating a temporal formula means evaluating all the subformulas in the temporal formula tree in a bottom-up manner. In order to represent a temporal formula as a tree consisting of its subformulas and operators, we need a scheme to number the subformulas and a data structure to store the subformulas. Therefore, in the following, we first present a numbering scheme followed by the descriptions of two arrays to refer to the subformulas of a temporal formula.

Assume that f is written in prefix notation. Then, *length(f)* denotes the total number of operands and operators in f. *length(f)* is used to number the subformulas of f in the following manner.

1. Assume that formula f is assigned the integer i.
2. If f is unary, *i.e.* $f = (\text{op}(f_1))$, then assign the integer $(i+1)$ through $(i+length(f_1))$ to the subformulas of f_1.
3. If f is binary, *i.e.* $f = (\text{op}(f_1 f_2))$, then assign the integers from $(i+1)$ through $(i + length(f_1))$ to the subformulas of f_1 and $(i + length(f_1))$ through $(i + length(f_1) + length(f_2))$ to the subformulas of f_2.

Thus, in one pass through f, we can build two arrays *nf*[1: length(*f*)] and *sf*[1: length(*f*)], where *nf*[*i*] is the *i*th subformula of f in the above numbering scheme and *sf*[*i*] is the list of the numbers assigned to the immediate subformulas of the *i*th formula. For example, the formula f = *(AU(NOT X)(OR Y Z))* can be represented by the tree structure shown in Fig. 4.2 with the contents of the arrays *nf* and *sf* given below:

nf[1]	*(AU (NOT X)(OR Y Z))*	*sf*[1]	(2 4)
nf[2]	(NOT X)	*sf*[2]	(3)
nf[3]	X	*sf*[3]	(nil)
nf[4]	*(OR Y Z)*	*sf*[4]	(5 6)
nf[5]	Y	*sf*[5]	(nil)
nf[6]	Z	*sf*[6]	(nil)



The numeric tag of a node is used as an index in the nf and sf arrays to refer to the subformula represented by that node.

Figure 4.2 Tree representation of a formula *f = (AU(NOT X)(OR Y Z))*.

Using the model generated from a test verification system in Chapter 3 and the safety and liveness properties stated in this chapter, we are in a position to apply the model checking algorithm of [CES 86]. The following algorithm uses Algorithms 4.1, 4.2, and 4.3 stated above to verify test case properties.

Algorithm 4.4.

Input: BTL structure M and a test case property as a temporal formula *f*.

Output: Correctness of *f*.

S1. Compute *length(f)*, the total number of subformulas and operators in *f*.

S2. Build two arrays *nf*[1:*length(f)*] and *sf*[1:*length(f)*] where *nf*[i] is the *i*th subformula of *f*, *sf*[*i*] is the list of the numbers assigned to the immediate subformulas of the *i*th formula. Essentially these two arrays maintain the tree structure describing the formula *f*.

S3. Define a bit array L[*s*] of size *length(f)* for each global state s such that L[*s*][*i*] is set to true if the subformula *nf*[*i*] holds in *s*.

S4. /* Successively apply the state labeling algorithm label_graph to *f*. */

for *fi* := *length(f)* **step** – 1 **until** 1 **do**

label_graph(*nf*[*fi*]).

S5. If f is a synchronization property then f holds if L[*s*][1] is true $\forall s \in S$

else the property f holds if L[s_0][1] is true.

Procedure *label_graph(f)*

begin

{main operator of f is AU}

execute Algorithm 4.1 for f

{main operator of f is $\mapsto$}

execute Algorithm 4.2 for f

{main operator of f is SEQ}

execute Algorithm 4.3 for f

end

4.4.5 Complexity Analysis

Let $M = (S, V, R, P_r, s_{init})$ be a BTL structure and f be a formula to be evaluated on M. To evaluate a formula f, Algorithm 4.4 calls the procedure *label_graph()* $length(f)$ times. Then, *label_graph()* uses Algorithm 4.1, Algorithm 4.2, and Algorithm 4.3 to evaluate operators $U, \mapsto$, and SEQ, respectively. To evaluate an operator, each of the Algorithms 4.1, 4.2, and 4.3 take time of the order $O(card(S)+card(R))$. Therefore, the time complexity of evaluating f on M, denoted by *TComplexity(f, M)*, is given by the following expression:

$TComplexity(f, M) = O(length(f) \times (card(S) + card(R)))$,

where $card(A)$ stands for the cardinality of a set A.

CHAPTER 5

RS TEST CASE VERIFICATION

In this chapter, we apply the verification technique to a RS architecture based test case chosen from a test suite developed at the Dutch PTT Research [PTT 90] to test an Association Control Service Element (ACSE) protocol implementation. The RS architecture and its corresponding test verification system are shown in Figs. 5.1 and 5.2, respectively.

This chapter contains five sections. In the first section, we briefly describe the ACSE protocol. The dynamic behavior table of the test case to be verified is discussed in the second section. An outline of the underlying service provider of the ACSE protocol entity is presented in the third section. In the fourth section, the global behavior of the test verification system is discussed. Finally, we state and verify the safety and liveness properties of the test case.

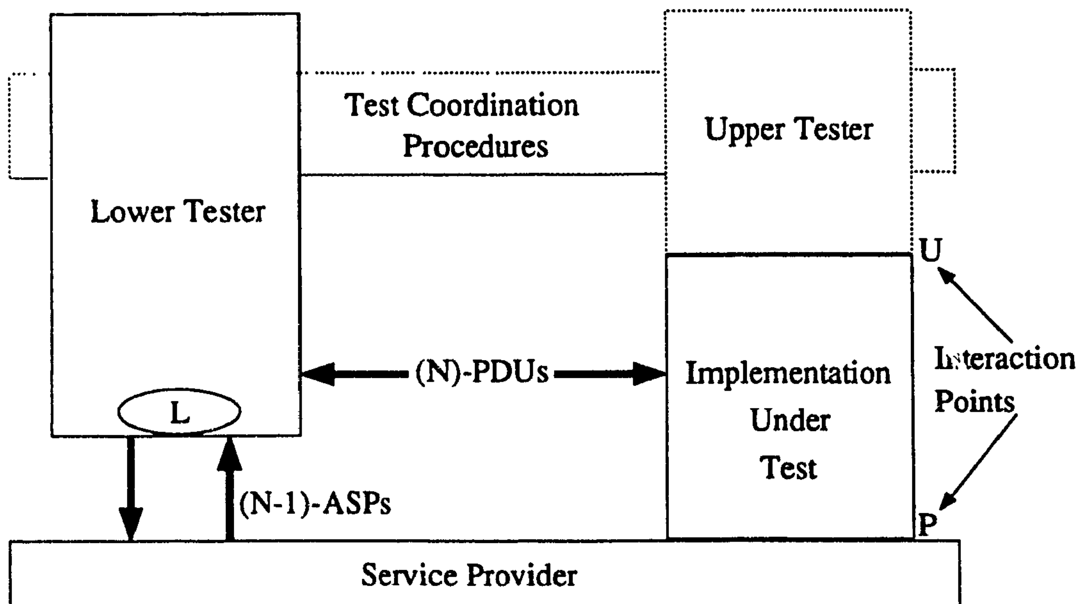


Figure 5.1 RS Test Architecture.

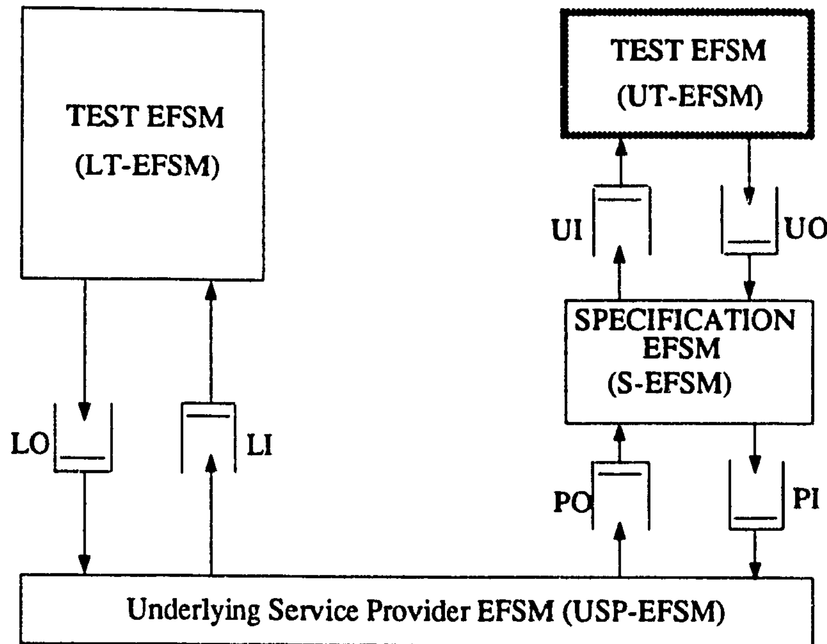


Figure 5.2 Test Verification System for the RS Architecture.

5.1. ACSE Specification

ACSE [ISO 8650] is a protocol at the application layer, i.e., the *7th.* layer of the OSI protocol stack. A number of Application Service Elements (ASEs), such as the File Transfer and Access Management (FTAM), Message Handling System (MHS), *etc.*, use the Common ASE (CASE) ACSE protocol for various services: confirmed establishment (A-ASSOCIATE) and disconnection (A-RELEASE) of associations, non-confirmed user initiated abortion of associations (A-ABORT), and provider initiated abortion of associations (A-P-ABORT). For a particular association, the ACSE services operate either in the *normal* mode or in the *X.410–1984* mode. The mode of operation is determined by the "Mode" parameter in the A-ASSOCIATE request primitive.

The state table model of the ACSE protocol is referred to as the Association Control Protocol Machine (ACPM). The ACPM communicates with its service-user by means of the ACSE service primitives and with its presentation service-provider

by means of the presentation services. The ACPM is activated by the receipt of input events from its user and service-provider. The input events from its user are request and response primitives and the input events from its presentation service-provider are presentation indication and confirm primitives. The ACPM responds to input events by issuing output events to its presentation service-provider and ACSE service-user. A new invocation of an ACPM is employed upon the receipt of an A-ASSOCIATE request primitive or a P-CONNECT indication primitive.

The ACSE protocol consists of the following three procedures:

i. association establishment;
ii. normal release of an association; and
iii. abnormal release of an association.

(i) **Association Establishment:** The association establishment procedure is used to establish an association between two Application Elements (AE). The association establishment procedure uses the A-ASSOCIATE-REQUEST (AARQ) and the A-ASSOCIATE-RESPONSE (AARE) Application PDUs (APDUs). This procedure is activated by the following events:

i. an A-ASSOCIATE request primitive from the requester;
ii. an AARQ APDU as user data in a P-CONNECT indication primitive from its service provider;
iii. an A-ASSOCIATE response primitive from the acceptor; and
iv. a P-CONNECT confirm primitive, which may or may not contain an AARE APDU.

The requesting ACPM forms an AARQ APDU from parameter values of the A-ASSOCIATE request primitive and issues a P-CONNECT request primitive. The User Data parameter of the P-CONNECT request primitive contains the AARQ APDU. Then, the requesting ACPM waits for a primitive from the presentation service-

provider and does not accept any other primitive from the requestor other than an A-ABORT request primitive.

The accepting ACPM receives an AARQ APDU from its peer as user data in a P-CONNECT indication primitive. The ACPM determines if the AARQ APDU is acceptable. If the AARQ APDU is not acceptable, a protocol error results, the association establishment procedure is disrupted, and an A-ASSOCIATE indication primitive is not issued. If the ACPM does not support a common protocol version, then it forms an AARE APDU with Protocol Version, Application Context Name, Result field with the value "rejected (permanent)", and Result Source-Diagnostic field with the values "ACSE service-provider" and "no common ACSE version." Then the AARE APDU is sent as user data in a P-CONNECT response primitive with a result parameter that has the value "user rejection." The receiving ACPM does not issue an A-ASSOCIATE indication primitive and the association is not established.

However, if the P-CONNECT indication primitive and its AARQ APDU are acceptable, the receiving ACPM issues an A-ASSOCIATE indication primitive to the acceptor. The A-ASSOCIATE indication primitive parameters are derived from the AARQ APDU and the P-CONNECT indication primitive. The ACPM waits for a primitive from the acceptor.

When the accepting ACPM receives the A-ASSOCIATE response primitive, the Result parameter specifies whether the service-user has accepted or rejected the association. The ACPM forms an AARE APDU using the A-ASSOCIATE response primitive parameters. The ACPM sets the Result Source-Diagnostic field to "ACSE service-user" and the value derived from the Diagnostic parameter of the response primitive. The AARE APDU is sent as the User Data parameter on the P-CONNECT response primitive.

If the acceptor accepts the association request, the Result parameter on the related P-CONNECT response primitive specifies "acceptance," and the result field of the

outgoing AARE APDU specifies "accepted." Then the association is said to be established on the accepting ACPM's side.

If the acceptor rejected the association request, the Result parameter on the P-CONNECT response primitive specifies "user-rejection," the Result field of the AARE APDU contains the appropriate rejection value, and the association is not established.

When the requesting ACPM receives a P-CONNECT confirm primitive, the following situations are possible:

i. the association has been accepted;

ii. the accepting ACPM or the acceptor has rejected the association; or

iii. the presentation service-provider has rejected the related presentation connection.

If the association has been accepted, the requesting ACPM issues an A-ASSOCIATE confirm primitive to the requestor derived from parameters from the P-CONNECT confirm primitive and the AARE APDU. The A-ASSOCIATE confirm primitive Result parameter specifies "accepted." Then the connection is said to be established on the initiating ACPM's side also.

If the association has been rejected, the requesting ACPM issues an A-ASSOCIATE confirm primitive to the requestor derived from parameters of the P-CONNECT confirm primitive and the AARE APDU. The A-ASSOCIATE confirm primitive Result parameter indicates "rejected (transient)" or "rejected (permanent)" and the association is said to be not established.

If the presentation-connection was rejected by the presentation service-provider, the P-CONNECT confirm primitive Result parameter specifies "provider-rejection." In this situation, the User Data field is not used. The requesting ACPM issues an A-ASSOCIATE confirm primitive with the Result parameter indicating "rejected (permanent)." The Result Source parameter indicates "presentation service-provider" and the association is said to be not established.

Collisions and Interactions: For a given ACPM, an A-ASSOCIATE collision cannot occur, because a new invocation of an ACPM is employed upon the receipt of an A-ASSOCIATE request primitive or a P-CONNECT indication primitive. Each such invocation controls exactly one association. For a given application element, two distinct ACPMs would be involved that represent the processing for two distinct associations:

i. an ACPM that processes the initial A-ASSOCIATE request primitive that results in the sending of an AARQ as user data in a P-CONNECT request primitive; and
ii. an ACPM that processes the subsequently received AARQ APDU as user data in a P-CONNECT indication primitive.

(ii) Normal Release: This procedure is used for the normal release of an association by an AE without loss of information in transit. The normal release procedure uses the A-RELEASE-REQUEST (RLRQ) APDU and the A-RELEASE-RESPONSE (RLRE) APDU. Each of these two APDUs contains two fields: Reason and User Information. This procedure is activated by the following events:

i. an A-RELEASE request primitive from the requestor;
ii. an RLRQ APDU as user data in a P-RELEASE indication primitive;
iii. an A-RELEASE response primitive from the acceptor; or
iv. an RLRE APDU as user data in a P-RELEASE confirm primitive.

When an A-RELEASE request primitive is received, the ACPM sends an RLRQ APDU as user data in a P-RELEASE request primitive and waits for a primitive from the presentation service-provider. It does not accept any primitives from the requestor other than an A-ABORT request primitive.

When the accepting ACPM receives the RLRQ APDU as user data in a P-RELEASE indication primitive, it issues an A-RELEASE indication primitive to the acceptor. It does not accept any ACSE primitives from its service-user other than an A-RELEASE response primitive or an A-ABORT request primitive. The Result

parameter on the A-RELEASE response primitive specifies whether the acceptor accepts or rejects the release of the association. The accepting ACPM forms an RLRE APDU from the response primitive parameters. The RLRE APDU is sent as user data in a P-RELEASE response primitive according to the following rules:

i. If the acceptor accepts the release, the Result parameter of the P-RELEASE response primitive has the value "affirmative."
ii. If the acceptor rejected the release, the Result parameter of the P-RELEASE response primitive has the value "negative." In this case, the association continues to be active.

When the requesting ACPM receives a P-RELEASE confirm primitive containing an RLRE APDU from its peer, the Result parameter on the P-RELEASE confirm primitive specifies either that the acceptor agrees or disagrees that the association may be released The requesting ACPM forms an A-RELEASE confirm primitive from the RLRE APDU fields in the following manners:

i. If the Result parameter on the P-RELEASE confirm primitive specifies "affirmative," the association is said to be released.
ii. If the Result parameter on the P-RELEASE confirm primitive specifies "negative," the association continues to be active.

A-RELEASE service collision: An A-RELEASE service collision occurs when an ACPM has sent out an RLRQ APDU as the user data of a P-RELEASE request primitive and instead of receiving the expected RLRE APDU as user data in a P-RELEASE confirm primitive from its peer, it receives an RLRQ APDU as the user data of a P-RELEASE indication primitive. The ACPM issues an A-RELEASE indication primitive to its service user. The procedure then followed by an ACPM depends on whether its service-user was the association initiator or the association responder as stated in the following:

a. For the association initiator:

   i. The ACPM waits for an A-RELEASE response primitive from its service-user. When it receives the response primitive, it sends an RLRE APDU as user data in a P-RELEASE response primitive and the association continues to exist.

   ii. The ACPM waits for an RLRE from its peer as user data in a P-RELEASE confirm primitive. It does not accept any primitive from its service-user other than an A-ABORT.

   iii. When the ACPM receives the RLRE, it forms an A-RELEASE confirm primitive and sends it to its service-user. The association is said to be released.

b. For the association responder:

   i. The ACPM waits for an RLRE from its peer and does not accept a primitive from its user other than an A-ABORT request primitive.

   ii. When this ACPM receives an RLRE, it issues an A-RELEASE indication primitive and the association continues to exist.

   iii. The ACPM now waits for an A-RELEASE response primitive from its service-user. When it receives the response primitive, it forms an RLRE APDU from the response primitive's parameters. The RLRE is sent as user data in a P-RELEASE response primitive and the association is released.

Abnormal Release of an Association: The Abnormal Release procedure can be used at any time to force the abrupt release of the association by a request in either application element, by either ACPM, or by the presentation service-provider. When the abnormal release procedure is applied during an attempt to establish an association, the association is not established. The abnormal release procedure supports the A-

ABORT and A-P-ABORT services. This procedure is activated by the following events:

i. an A-ABORT request primitive from the requestor;
ii. a P-U-ABORT indication primitive;
iii. a P-P-ABORT indication primitive; or
iv. a protocol error detected by an ACPM.

When an ACPM receives an A-ABORT request primitive from its service-user, the processing that it performs depends on the version of the underlying session protocol [ISO 8327] that supports the association as specified in the following. For version 1, the ACPM simply sends a P-U-ABORT request to the service provider and the association is released. For other versions, the ACPM sends an ABRT APDU in a P-U-ABORT request primitive and the association is released.

When an ACPM receives a P-U-ABORT indication primitive, it issues an A-ABORT indication primitive to its user. When an ACPM receives a P-P-ABORT indication primitive, the ACPM issues an A-P-ABORT indication primitive to the acceptor and the association is said to be released.

Two types of ACSE protocol errors are possible:

i. for a particular ACPM state, an unexpected APDU is received; or
ii. an invalid field is encountered during the processing of an incoming APDU.

We consider an Estelle specification of ACSE. The specification has two Service Access Points *A*–also denoted as *U* in the testing environment–and *P* through which it interacts with its user and the service provider, respectively. The specification has 8 major states {ACSE_IDLE, AWAIT_AARE_APDU, AWAIT_AARE, AWAIT_RLRE_APDU, AWAIT_RLRE, ACSE_ASSOC, COLLISION_ASSOC_INITIATOR, COLLISION_ASSOC_RESPONDER} and 46 normalized transitions in a single module [ISO 8650].

By applying the transformation technique represented by Algorithm 2.1 to the ACSE specification, we obtain 54 states and 89 transitions in the protocol specification EFSM, i.e, S-EFSM. The transitions of the S-EFSM required to verify the test case are shown below.

```
S1: <ACSE_IDLE, 1, U?A_ASCreq,
    [Event.Mode = MODE_Supported],
    (P_CONreq.User_Data.Protocol_Version[VERSION1] := 1;
     :
     P_CONreq.Session_Con_Id := Event.Session_Con_Id;),1>
S2: <1, AWAIT_AARE_APDU, P!P_CONreq, [True], (),1>
S3: <AWAIT_AARE_APDU, 10, P?P_CONcnf,
     [(Event.Result = P_ACCEPTANCE) and
      (Event.User_Data.Result = ACCEPTED)],
     (ASCcnf.Appl_Cntx_Name := Event.User_Data.Appl_Cntx_Name;
      :
      ASCcnf.Session_Con_Id := Event.Session_Con_Id;),1>
S4: <10, ACSE_ASSOC, U!ASCcnf, [True], (),1>
S5: <ACSE_ASSOC, 25, U?A_RLSreq, [True],
    (P_RELreq.User_Data.reason := Event.Reason;
     P_RELreq.User_Data.User_Info := Event.User_Info;),1>
S6: <25, AWAIT_RLRE_APDU, P!P_RELreq, [True], (),1>
S7: <AWAIT_RLRE_APDU, 28, P?P_RELcnf,
     [Event.Result = P_AFFIRMATIVE],
     (A_RLScnf.Result := ACSE_AFFIRMATIVE;
      A_RLScnf.Reason := Event.User_Data.Reason;
      A_RLScnfcnf.User_Info := Event.User_Data.User_info;),1>
S8: <28, ACSE_IDLE, U!A_RLScnf, [True], (),1>
S9: <AWAIT_RLRE_APDU, 30, P?P_UABind, [true],
    (A_ABind.Abort_Source := Event.User_Data.Abort_Source;
     A_ABind.User_Info := Event.User_Data.User_info;),1>
```

5.2. Test Case

We consider a test case from the ACSE Abstract Test Suite (ATS) [PTT 90]. This test case is designed as a valid behavior test to test the normal mode of IUT-initiated association establishment. The dynamic behavior table of the test case is given in Fig. 5.3. The main behavior of the test case is written to satisfy the test purpose, which consists of three sequential activities: make the IUT initiate an association by specifying an implicit send event, accept the association, and check that the IUT

establishes the association by making the IUT release the association through the use of another implicit send. A Pass verdict is assigned if the test goes through these steps.

The test case uses a default behavior tree *DEF1* (not shown here) to treat all abnormal situations such as aborts, timeouts, and unexpected events. *DEF1* appears as an alternative to the two input events L?P_CONind and L?P_RELind in Fig. 5.3. A fail/ inconclusive verdict is assigned for these cases.

By applying Algorithm 2.2 to the test case, we obtain 54 states and 53 transitions in the LT-EFSM. In the following, we show the transitions corresponding to the main tree of the test case. State 16 has a Pass verdict tag.

Test Case Dynamic Behavior

Test Case Name: BV/NM/AE/I/1
Group :
Purpose : The IUT initiates the association.
The LT accepts the association. Check that the IUT establishes the association.
Default : DEF1
Comments:

Nr	Lab	Behavior Description	Constraints Ref.	V	Com.
		(STATE := NONE)			
		(TEST_BODY := TRUE)			
		<IUT!A_ASCreq>	A_ASCreqbase(		
		#	CONRQPDUbase)		
		START A			
		L?P_CONind	P_CONindbase(		
		#	AARQ501)		
		SESS_CON_ID_IUT := Session_connection_identifier			
		CANCEL A			
		L!P_CONrsp(STATE:=CONNECT)	P_CONrspbase(		
		# Session_connection_identifier:=SESS_CON_ID_IUT	AAREbase(		
		#	EXTCONREbase))		
		<IUT!A_RLSreq>	A_RLSreqbase(		
		#	RELRQPDUbase)		
		START A			
		L?P_RELind	P_RELindbase(		
		#	ARLRQbase_R)		
		CANCEL A			
		L!P_RELrsp	P_RELrspbase(		
		#	ARLREbase_S(		
		#	EXTRELREbase))		
		(STATE := NONE)			
		[STATE = NONE]		(P)	

Figure 5.3 A test case dynamic behavior taken from [PTT 90], Ref. BV/NM/AE/I/1

```
LT_1: <1,2,Null,True,(STATE:=NONE,TEST_BODY:=TRUE),1>
LT_2: <2,3,IUT!A_ASCreq,True,
       (A_ASCreq.Mode:=Mode_normal,
        A_ASCreq.Appl_context_name:=Appl_context_name_FTAM,
          :
        A_ASCreq.User_data:=CONRQPDUbase),1>
LT_3: <3,4,START A,True,(),1,T>
LT_4: <4,5,L?P_CONind,[P_CONind.User_data=AARQ501],(),1>
LT_5: <5,6,Null,True,
        (SESS_CON_ID_IUT:=Sess_conn_identifier),1>
LT_6: <6,7,CANCEL A,True,1>
LT_7: <7,8,L!P_CONrsp,True,
       (STATE:=CONNECT,Sess_conn_identifier:=SESS_CON_ID_IUT,
        P_CONrsp.Resp_pres_addr:=TSP_pres_addr_tester,
        :
        P_CONrsp.Result:=Result_accept),1>
LT_8: <8,9,Null,True,(STATE:=CONNECT),1>
LT_9: <9,10,IUT!A_RELreq,True,(A_RLSreq.User_data:=relrqpdu,
A_RLSreq.reason:=Release_request_reason_norm),1>
LT_10: <10,11,START A,True,(),1,T>
LT_11: <11,12,L?P_RELind,
         [P_RELind.User_data=ARLRQbase_R],(),1>
LT_12: <12,13,CANCEL A,True,(),1>
LT_13: <13,14,L!P_RELrsp,True,
        (P_RELrsp.User_data:=ARLREbase_S(EXTRELREbase),
        P_RELrsp.Result:=result_affirmative),1>
LT_14: <14,15,Null,True,(STATE:=NONE),1>
LT_15: <15,16,Null,[STATE = NONE],(),1>
LT_16: <4, 21,?Timeout(A), True, (), 2>
LT_17: <21,22,i, [STATE <> NONE], (), 1>
LT_18: <21,24,i, [STATE = NONE], (), 2>
LT_19: <22,23,L!P_UABreq(STATE:=NONE), (),1>
LT_20: <11, 25,?Timeout(A), True, (), 2>
LT_21: <25,26,i, [STATE <> NONE], (), 1>
LT_22: <25,28,i, [STATE = NONE], (), 2>
LT_23: <26,27,L!P_UABreq(STATE:=NONE), (),1>
```

5.3. Service Provider

In an external test architecture, such as the RS architecture, the LT test entity communicates with the implementation under test through a *service provider*. To provide connection association/release services to its users, an ACSE protocol entity uses the services of a *Presentation Layer* protocol. Therefore, to verify RS architecture based test cases against an ACSE protocol specification, it is essential to have the EFSM description of the Presentation Layer in terms of input/output events as viewed by its users. In the following, we present a state machine of a presentation protocol entity that communicates with the LT-EFSM through an interaction point L and with the protocol specification EFSM (S-EFSM) through interaction point P. This machine, shown in Fig. 5.4, contains of 11 states and 21 transitions and maps the abstract service primitives between the LT-EFSM and the S-EFSM.

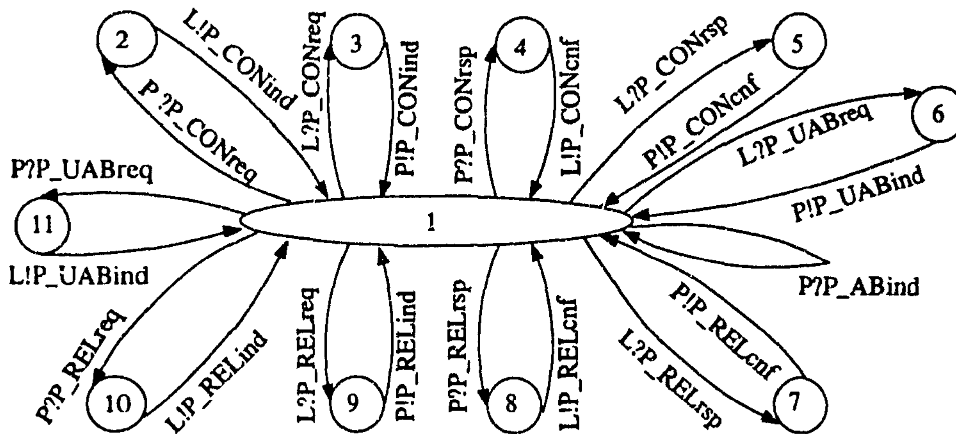


Figure 5.4 USP-EFSM, a simple model of the Presentation Service used by an ACSE protocol.

5.4. Global State Space

In the first step of the verification methodology, a model is generated from the test verification system. While applying the global state space generation algorithm, presented in Chapter 3, we detected an error in the test case. After selecting the

LT_2 transition containing an implicit send event <IUT!A_ASCreq> in the LT-EFSM to perturb a global state, it was detected that it was not possible for the S-EFSM to output an A_ASCreq event to the USP-EFSM. The correct implicit send event is <IUT!P_CONreq>. Similarly, the implicit send event <IUT!A_RELreq> in transition LT_9 was found to be incorrect. The correct event is <IUT!P_RELreq>. The corrected version of the test case is shown in Fig. 5.5. After these two modifications to the test case, we generated a global state space containing 45 states and 44 transitions.

Test Case Dynamic Behavior

Test Case Name: BV/NM/AE/I/1
Group :
Purpose : The IUT initiates the association.
The LT accepts the association. Check that the IUT establishes the association.
Default : DEF1
Comments:

Nr	Lab	Behavior Description	Constraints Ref.	V	Com.
		(STATE := NONE)			
		(TEST_BODY := TRUE)			
		<IUT!P_CONreq>	A_ASCreqbase(		
		#	CONRQPDUbase)		
		START A			
		L?P_CONind	P_CONindbase(		
		#	AARQ501)		
		SESS_CON_ID_IUT := Session_connection_identifier			
		CANCEL A			
		L!P_CONrsp(STATE:=CONNECT)	P_CONrspbase(		
		# Session_connection_identifier:=SESS_CON_ID_IUT	AAREbase(		
		#	EXTCONREbase))		
		<IUT!P_RELreq>	A_RLSreqbase(		
		#	RELRQPDUbase)		
		START A			
		L?P_RELind	P_RELindbase(		
		#	ARLRQbase_R)		
		CANCEL A			
		L!P_RELrsp	P_RELrspbase(		
		#	ARLREbase_S(		
		#	EXTRELREbase))		
		(STATE := NONE)			
		[STATE = NONE]		(P)	

Figure 5.5 The corrected version of the test case in Fig. 5.3.

Then, by applying the model generation algorithm in Chapter 3, we associated a set of atomic propositions with each global state. The resulting model for the test verification system is given in **Appendix 1.**

The graphical representation of the state space is shown in Fig. 5.6. The path from state *g1* to *g32* represents the behavior of the verification system corresponding to the main behavior test tree given in Fig. 5.3. The path from state *g6* to *g36* and the path from state *g21* to *g45* are due to timeouts in the default behavior of the test case.

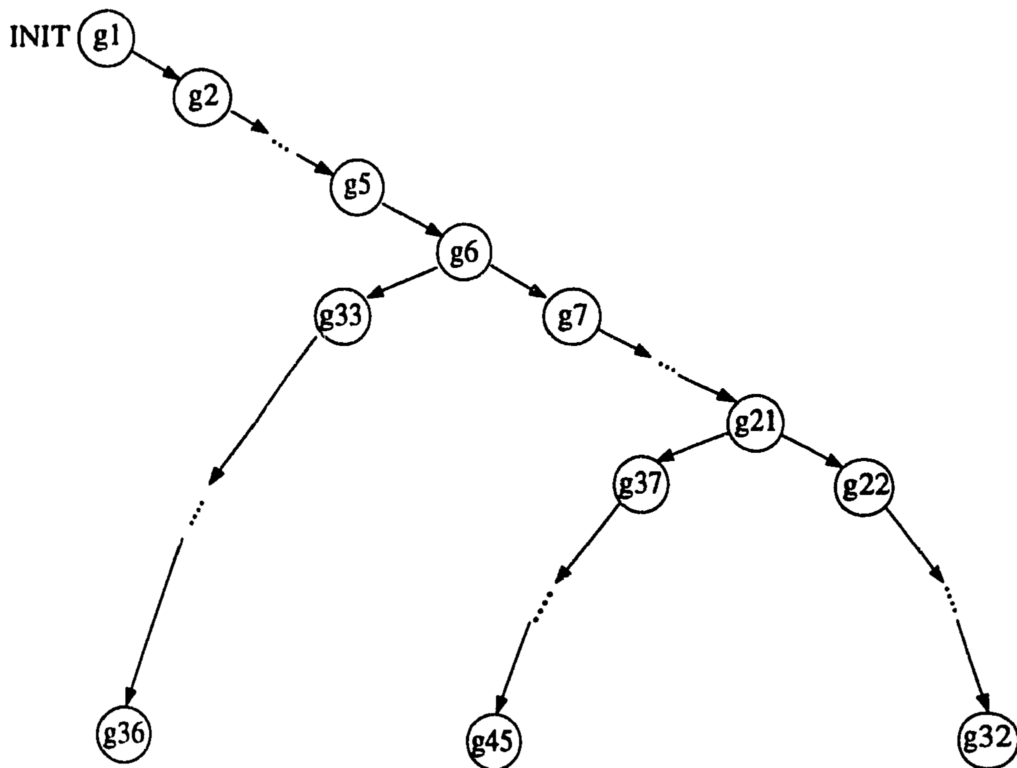


Figure 5.6 Structure of the global state space of the TVS in Fig. 5.2.

The message A_ASCreq generated as a part of the UT after the first implicit send event in the test case is represented as a tree structure, shown in Fig. 5.7, and is put in the channel AO. As an example we evaluate the predicate [Event.Mode = MODE_Supported] , which is the enabling condition of transition S1 of the S-EFSM. Here "Event" A_ASCreq is received by the specification from the channel AO.

To evaluate the predicate, the *eval* routine first extracts the value of "Event.Mode" (in this case, it is 1) from the composite data A_ASCreq by traversing IOD until the field "Mode" is encountered. Then the value of "Event.Mode" is compared with the variable "MODE_Supported" which is also equal to 1. Thus the predicate [Event.Mode = MODE_Supported] evaluates to true.

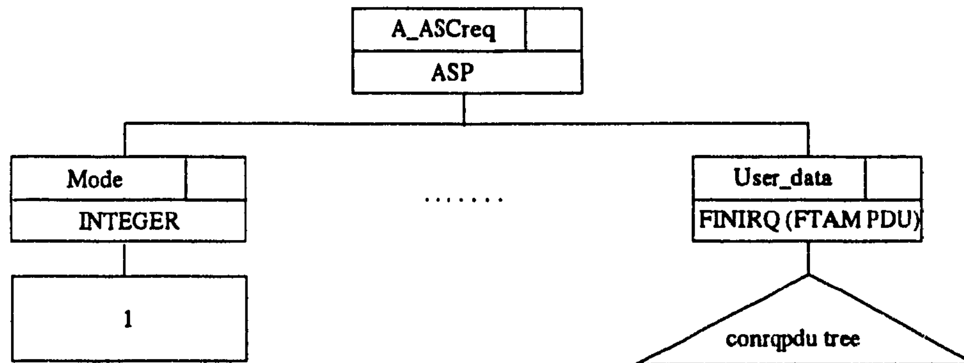


Figure 5.7 IOD representation of A_ASCreq message.

5.5 Verification of Test Case Properties

In this Section, we show the detailed verification of a transmission safety property, and the formalization and verification of the liveness property.

5.5.1 Safety Properties of the Test Case

Here we explain the satisfaction of a transmission safety property. The transmission safety property

$$INIT \models AG(AFTER(Tsend(U, ASCreq)) \mapsto AFTER(Sreceive(U, ASCreq))),$$

is satisfied, because the predicate AFTER(Tsend(U, A_ASCreq)) holds in state *g3* and the property AFTER(Sreceive(U, A_ASCreq)) holds in state *g5*. Thus, AFTER(Sreceive(U, A_ASCreq)) eventually holds on every path reachable from *g3*. The test case under consideration satisfies all the safety properties.

5.5.2 Liveness Property of the Test Case

To verify the liveness property of the test case, we first express the test purpose as a temporal formula. As a part of the test case, the test purpose is specified in a natural language as follows.

i. *The IUT initiates the association.*

ii. *The LT accepts the association.*

iii. *Check that the IUT established the association.*

In the following, each part of the test purpose is expressed as a temporal formula:

i. $AFTER(Treceive(L, P_CONind))$

ii. $AFTER(Tsend(L, P_CONrsp))$

iii. $AFTER(Tsend(U, A_RELreq)) \mapsto AFTER(Treceive(L, P_RELind))$.

These basic test purposes can be composed using the SEQ operator to give rise to a formula for the entire test purpose as follows:

$$f_1 = (AFTER(Treceive(L, P_CONind)) \text{ SEQ } AFTER(Tsend(L, P_CONrsp)) \text{ SEQ } AFTER(Tsend(U, A_RELreq)) \mapsto AFTER(Treceive(L, P_RELind))).$$

Then the liveness property of the test case is stated as $INIT \models (f_1 \mapsto (Verdict = Pass))$, which is satisfied by the sequence of global states from *g1* to *g32* and (Verdict = Pass) holds at *g32*.

5.6 Generation of an Upper Tester

It may be recalled from Chapter 3 that there is no explicit Upper Tester in an RS architecture. While generating a state space from a test verification system, the

state space generation algorithm in Chapter 3 dynamically generates the behavior of an Upper Tester. The initial configuration of such an Upper Tester is shown in Fig. 5.8(a). It consist of one state, denoted by 1, and one transition <1, 1, A?OTHERWISE, T, (), 2> to accept any event from the protocol specification. During the state space exploration process, as the algorithm encounters the first implicit send event <IUT!P_CONreq>, state 2 and transitions <1, 2, A!A_ASCreq, T, (), 1> and <2, 2, A?OTHERWISE, T, (), 2> are added to the initial UT behavior resulting in another partial UT behavior as shown in Fig. 5.8(b). When the algorithm encounters the second implicit send event <IUT!P_RELreq>, state 3 and transitions <2, 3, A!A_RELreq, T, (), 1> and <3, 3, A?OTHERWISE, T, (), 2> are added to the partial UT behavior in Fig. 5.8(b) to generate an UT behavior as shown in Fig. 5.8(c). The UT behavior shown in Fig. 5.8(c) is the final UT behavior since there are no more implicit send events in the test case.

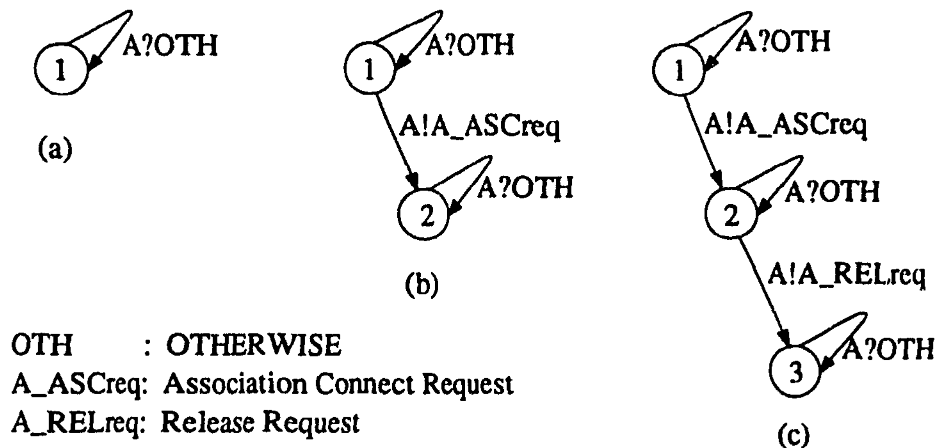
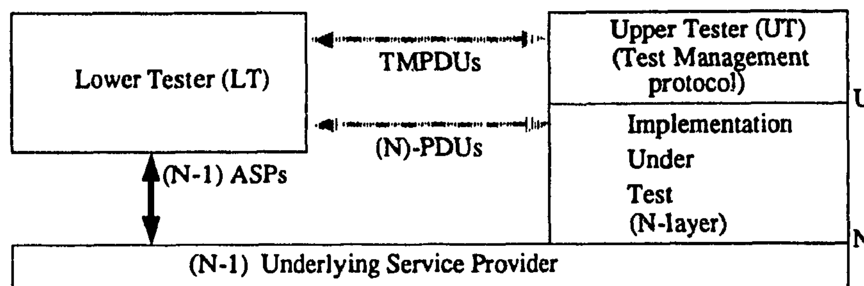


Figure 5.8 Dynamic generation of UT behavior.

CHAPTER 6

CS TEST CASE VERIFICATION

In this chapter, we present an example of verifying a single connection test case chosen from a Coordinated Single-layer (CS) architecture based test suite [NCC 88] designed to test a Class 2 transport protocol implementation. The CS architecture and its corresponding test verification system are shown in Figs. 6.1 and 6.2, respectively.



U: Interaction Point between the TMP and the Implementation
N: Interaction Point between the Implementation and the Service Provider

Figure 6.1 Coordinated Single-layer (CS) test architecture.

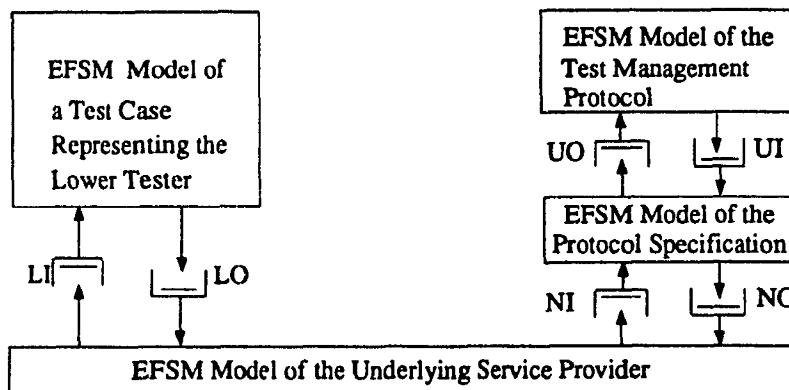


Figure 6.2 The Test Verification System for the CS architecture.

We first present the specifications of the transport protocol, the Test Management Protocol (TMP), and the service provider of the transport protocol. Then the test case is explained and verified.

6.1 Transport Protocol, TMP, and Service Provider

In this section, we describe a Class 2 transport protocol specification [NCC 88], a test case for testing an implementation of the protocol, the underlying service provider of the protocol, and a Test Management Protocol (TMP) used in testing an implementation in the CS architecture.

6.1.1 Class 2 Transport Protocol Specification

A nondeterministic EFSM model of a Class 2 transport protocol specification [BOCH 89] is shown in Fig. 6.3. State 1 is both the initial and final state of the EFSM and represents a *closed connection.* State 10 corresponds to an *open connection* state. On one hand, if the connection establishment procedure is initiated by the user of the transport entity, then the EFSM moves from state 1 to state 10 through the state sequence {1, 2, 3, 4, 10}. In this case, if the peer entity refuses to establish a connection, then the EFSM goes back to the initial state through the sequence {1, 2, 3, 19, 20, 21, 1}. On the other hand, if the connection is established by the peer entity, then the EFSM moves from state 1 to state 10 through the sequence {1, 5, 6, 7, 10}. In this case, a request for connection establishment can be refused by the protocol EFSM such that the EFSM goes back to the initial state through the state sequence {1, 5, 1}. However, if the request for connection establishment is refused by the user of the protocol entity, then the protocol EFSM goes back to the initial state through the sequence {1, 5, 6, 8, 1}. There are two internal transitions in the EFSM. The first internal transition from state 10 to 11 models the effect of the environment on the protocol specification leading to a disconnection of the transport connection along the state sequence {10, 11, 12, 9, 1}. The second internal transition from state 10 to 18 models the acknowledgement (AK) transmission policies including timeouts.

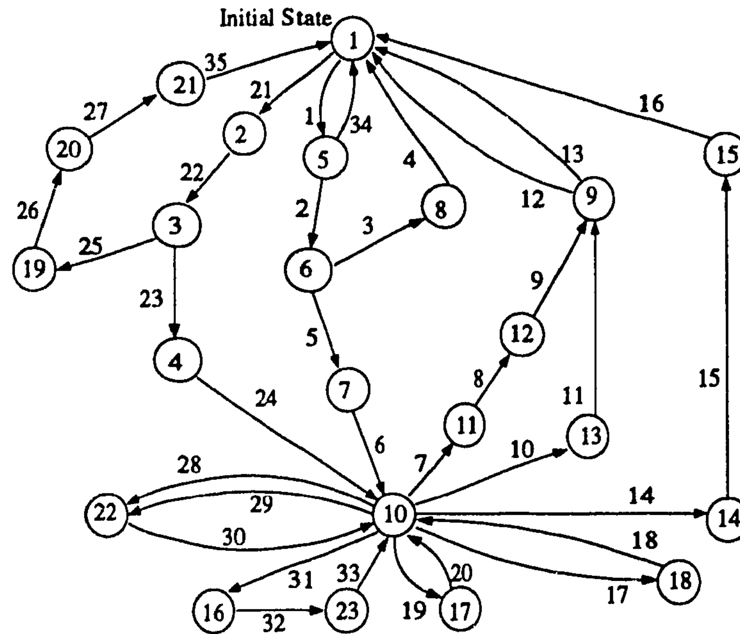


Figure 6.3 Class 2 Transport Protocol Specification.

The transitions of the EFSM shown in Fig. 6.3 are given below.

```
Following variables are updated during the example state
exploration process.
opt := ""          /* negotiation option */

PRseq :=0          /* receive sequence number */

PRcredit := 2      /* receive credit          */

SPEC_1:   <1,5,N?NDTind(CR),T,
           {opt::=CR.Exp_option, PScredit:=CR.credit},1>
SPEC_2:   <5,6,U!TCONind(CR.Called_addr,
           CR.Calling_addr,opt,CR.Qos,CR.User_data),
           T,{},1>
SPEC_3:   <6,8,U?TDISreq,T,{},1>
SPEC_4:   <8,1,N!NDTreq(DR(TDISreq.Reason,
           IDISreq.User_data)),T,{},1>
SPEC_5:   <6,7,U?TCONresp,[TCONresp.Exp_option<=opt],
           {opt:=TCONresp.Exp_option,PRseq:=PSseq:=0},1>
SPEC_6:   <7,10,N!NDTreq(CC(TCONresp.Calling_addr,TCONresp.Qos,
            opt,PRcredit,TCONresp.User_data)),T,{},1>
SPEC_7:   <10,11,i,T,{},1>
```

```
SPEC_8:  <11,12,U!TDISind(Reason, User_data),
          T,{User_data:=NULL},1>
SPEC_9:  <12,9,N!NDTreq(DR(Reason,User_data)),
          T,{User_data:=NULL},1>
SPEC_10: <10,13,U?TDISreq,T,{},1>
SPEC_11: <13,9,N!NDTreq(DR(TDISreq.Reason,
          TDISreq.User_data)),T,{},1>
SPEC_12: <9,1,N?NDTind(DC),T,{},1>
SPEC_13: <9,1,N?NDTind(DR),T,{},1>
SPEC_14: <10,14,N?NDTind(DR),T,{},1>
SPEC_15: <14,15,U!TDISind(DR.Reason, DR.User_data),T,{},1>
SPEC_16: <15,1,N!NDTreq(DC),T,{},1>
SPEC_17: <10,18,i,T,{},1>
SPEC_18: <18,10,N!NDTreq(AK(PRseq,PRcredit)),
          T,{},1>
SPEC_19: <10,17,N?NDTind(DT),
          [PRcredit <> 0 & DT.Seq=PRseq,{},1>
SPEC_20: <17,10,U!TDATAind(DT.User_data,DT.EOT),T,
           {PRseq:=(PRseq+1)mod 128,PRcredit:=PRcredit-1},1>
SPEC_21: <1,2,U?TCONreq, T, {opt:=TCONreq.proposed_options},1>
SPEC_22: <2,3,N!NDTreq(CR(TCONreq.Called_addr,
          TCONreq.Calling_addr,opt,PRcredit)),
          T,{},1>
SPEC_23: <3,4,N?NDTind(CC),[CC.Exp_option <= opt],
          {opt:=CC.Exp_option,PRseq:=0,PSseq:=0,
           S_credit:=CC.credit_value},1>
SPEC_24: <4,10,U!TCONconf(CC.Calling_addr,
          CC.Exp_option,CC.Qos,CC.User_data),
          T, {},1>
SPEC_25: <3,19,N?NDTind(CC),
          [not(CC.Exp_option <= opt)], T, {},1>
SPEC_26: <19,20,N!NDTreq(DR(procedure_error,
          User_data)), T, {User_data:=NULL}, 1>
SPEC_27: <20,21,U!TDISind(procedure_error,
          User_data), T, {User_data:=NULL}, 1>
SPEC_28: <10, 22, N?NDTind(AK),
          [TSseq < AK.expected_send_sequence],
          {new_credit:=AK.credit_value+
          AK.expected_send_sequence - (TSseq+128)},1>
SPEC_29: <10,22,N?NDTind(AK),
          [PSseq>=AK.expected_send_sequence],
        {new_credit:=AK.credit_value+
         AK.expected_send_sequence - PSseq},1>
SPEC_30: <22,10, i, T, {PScredit:=new_credit}, 1>
SPEC_31: <10, 16, U?TDATAreq,
```

```
          [PScredit > 0],{PScredit:=PScredit - 1},1>
SPEC_32: <16, 23, N!NDTreq(DT(TSseq,
         TDATAreq.TS_user_data,TDATAreq.EOT)),
         T,{},1>
SPEC_33: <23, 10, i, T,
          {PSseq:=(PSseq+1) mod 128},1>
SPEC_34: <5,1,N!NDTreq(DR(Reason,User_data)),
         [opt not supported],{User_data:=Null},2>
SPEC_35: <21,1,N?NDTind(DC),T,{},1>
```

6.1.2 Service Provider

The transport protocol, in order to provide the desired service to its user, uses the services of a network layer. That is, the network layer acts as the Underlying Service Provider in the CS test architecture. A simplified EFSM view of the service provider is shown in Fig. 6.4. The service provider EFSM has 9 states. In the test verification methodology, the basic function of the service provider is to transport a *request* primitive from the Lower Tester (protocol specification) to the protocol specification (Lower Tester) in the form of an *indication* primitive.

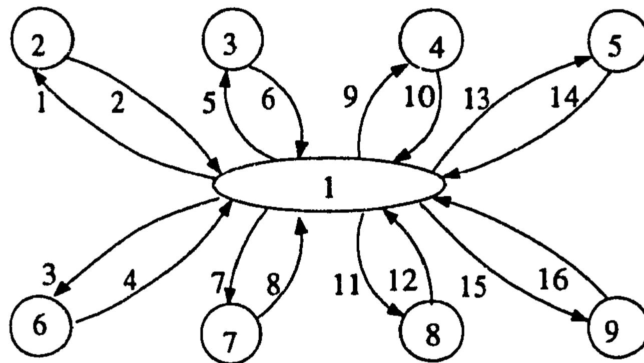


Figure 6.4 Underlying Service Provider of the Transport Protocol.

The transitions of the EFSM shown in Fig. 6.4 are given below.

```
USP_1:  <1,2,L?NCONreq(x),T,{},1>
USP_2:  <2,1,N!NCONind(x),T,{},1>
USP_3:  <1,6,N?NCONreq(x),T,{},1>
USP_4:  <6,1,L!NCONind(x),T,{},1>
USP_5:  <1,3,L?NCONresp(x),T,{},1>
USP_6:  <3,1,N!NCONconf(x),T,{},1>
USP_7:  <1,7,N?NCONresp(x),T,{},1>
USP_8:  <7,1,L!NCONconf(x),T,{},1>
USP_9:  <1,4,L?NDTreq(x),T,{},1>
USP_10: <4,1,N!NDTind(x),T,{},1>
USP_11: <1,8,N?NDTreq(x),T,{},1>
USP_12: <8,1,L!NDTind(x),T,{},1>
USP_13: <1,5,L?NDISreq(x),T,{},1>
USP_14: <5,1,N!NDISind(x),T,{},1>
USP_15: <1,9,N?NDISreq(x),T,{},1>
USP_16: <9,1,L!NDISind(x),T,{},1>
```

6.1.3 Test Management Protocol (TMP)

A TMP is required to function as a test responder (Upper Tester) while testing an implementation in the CS architecture. Since the TMP entity in the CS architecture interacts with the upper service boundary of an implementation to be tested, every protocol specification must have a corresponding TMP. In this section, we explain a TMP [NCC 88] corresponding to a transport protocol. We developed an EFSM model of the Test Management Protocol from its informal tabular description. There are 35 states and 447 transitions in the TMP EFSM given in **Appendix 2**. For the Lower Tester to be able to control the events occurring at the upper service access point of the implementation, the TMP entity must be controlled in a deterministic manner by the Lower Tester through the use of Test Management Protocol Data Units (TMPDU). The TMP contains three types of internal variables: *counts*, *modes*, and *stored items*.

There are 38 count variables C1 through C38. The counts monitor the traffic of transport service primitives across the interface between the TMP and the transport protocol entity. Counts are assigned to each category of service primitive in each direction across the interface, with normal and out of context primitives being counted

separately. In addition to the transport service primitives, counts also exist for various totals. Data and expedited data octets received by the TMP entity are also counted. For example, C1 counts normal T-CONNECT indications, C2 counts normal T-CONNECT confirms, C8 counts out of context T-CONNECT indications, etc.

The behavior of the TMP is controlled by the Lower Tester through a set of 25 mode parameters M1 thru M25. Mode parameters are used to define the series of actions which make up the response to a *defined event*, or to define the parameters for the *Count Limit Event*, or for generating data to be sent in a T-DATA/ T-EXPEDITED-DATA request service primitive. For example, M1 defines the action to be taken in response to a normal T-CONNECT indication, M2 defines the action to be taken in response to a T-CONNECT confirm, etc. Each mode parameter can take values from the set {A0, A1, ..., A15}.

The defined events mentioned above are simply some form of stimulus to which the TMP makes a response. Examples of the defined events are the START internal event, receipt of T-CONNECT indication, receipt of T-CONNECT confirm, Count Limit Event, etc. In response to these events, the appropriate mode parameters specify a sequence of primitives. For example, if the TMP receives a T-CONNECT indication in the IDLE state, then it moves to the IUT_WFTRESP_M1 state and takes an action depending on the value of the mode parameter M1. For instance, if the value of M1 is A1 then the TMP issues a T-DISCONNECT request and moves to the IDLE state, if the value of M1 is A4 then the TMP issues a T-CONresp and moves to the OPEN state, and so on.

A *Count Limit Event (CLE)* is an *internal* event generated in the TMP when the count variable identifier specified by the mode parameter M13 attains the value specified in M14.

The 28 stored items S1 thru S28 consist of additional variables, which include the last received parameters from incoming service primitives and the values supplied

as parameters to outgoing primitives. For example, the stored items (S5, S6, S7, S8) are used with a T-CONNECT response such that S5 contains "Quality of Service", S6 contains "Responding Address", S7 contains "Expedited Data Option", and S8 contains "TS User Data". If a T-CONNECT indication is received by the TMP, then the "Called Address" is saved in S16, the "Calling Address" is saved in S17, the "Expedited Data Option" is saved in S18, the "Quality of Service" is saved in S19, and the "TS User Data" is saved in S20.

The TMP's internal variables can be controlled and monitored by the Lower Tester through the Test Management Protocol Data Units (TMPDUs). There are 23 TMPDUs, which can be classified into two groups: *command* and *reply*. TMPDU1 through TMPDU15 constitute command TMPDUs and TMPDU8r through TMPDU15r constitute the reply TMPDUs. The Lower Tester can issue a command to the TMP to take some action by sending a command TMPDU and the TMP can send a reply to the Lower Tester through a reply TMPDU. There are three categories of commands: those which set internal variables, those which request relay of internal information, and those which cause the TMP to take some other specific actions such as sending T-DATA requests by starting a data source.

We show the EFSM description of the part of the TMP that is used in the verification of a test case in Fig. 6.5.

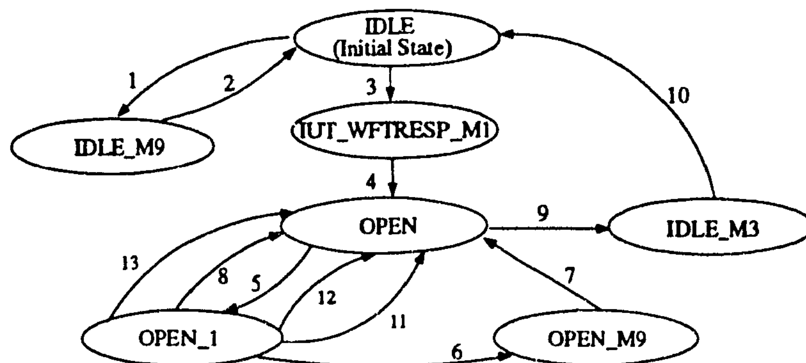


Figure 6.5 A part of the Test Management Protocol.

The transitions of the above TMP are given below.

```
TMP_1:  <IDLE, IDLE_M9, Internal_START, [T],
         {C1:=0,C2:= 0,..,C38:=0,M1:=A4,M2:=A0,..,
          M10:=A0,M11:=A5,M12:=A5,M13:=A0,..,M25:=A0,
          S1:=..,S2:=..,.., S5:="1",S6:="you",
          S7:="No",S8:="any_data",..S28},1>
TMP_2:  <IDLE_M9, IDLE, Null,   [M9=A0], {},1>
TMP_3:  <IDLE, IUT_WFTRESP_M1, U?TCONind, [T],
         {S16:=TCONind.Called_address,
          S17:=TCONind.Calling_address,
          S18:=TCONind.Exp_data_option,
          S19:=TCONind.Qos,
          S20:=TCONind.TSuser_data,i[illegible]1, C19)},1>
TMP_4:  <IUT_WFTRESP_M1, OPEN,
         U!TCONresp(S5,S6,S7,S8),[M1=A4],
         {inc(C24,C37)},1>
TMP_5:  <OPEN, OPEN_1, U?TDTind, [P_TMPDU],
         {inc(C4) per octet, inc(C5,C19)}>
TMP_6:  <OPEN_1, OPEN_M9, Null, [TMPDU1 in TDTind],
         {M1:=TDTind.TMPDU1.M1,
          M2:=TDTind.TMPDU1.M2, ...,
          M25:=TDTind.TMPDU1.M25},1>
TMP_7:  <OPEN_M9, OPEN, Null,[M9=A0], {},1>
TMP_8:  <OPEN_1, OPEN, Null, [TMPDU4 in TDTind],
         {S5:=TDTind.TMPDU4.S5,S6:=TDTind.TMPDU4.S6,
         S7:=TDTind.TMPDU4.S7,S8:=TDTind.TMPDU4.S8},4>
TMP_9:  <OPEN,IDLE_M3,U?TDISind,[T],
         {S14:=TDISind.Reason,
          S15:=TDISind.TSuser_data,inc(C3,C10)},1>
TMP_10: <IDLE_M3, IDLE, Null,[M3=A0], {},1>
TMP_11: <OPEN_1, OPEN, Null, [TMPDU3 in TDTind],
         {S1:=TDTind.TMPDU3.S1},3>
TMP_12: <OPEN_1, OPEN, Null, [TMPDU5 in TDTind],
        {S9:=TDTind.TMPDU5.S9,
         S10:=TDTind.TMPDU5.S10, S11:=TDTind.TMPDU5.S11,
         S12:=TDTind.TMPDU5.S12,
         S13:=TDTind.TMPDU5.S13},5>
TMP_13: <OPEN_1,OPEN,U!TDTreq(TSuser_data),[TMPDU8 in TDTind],
         {TSuser_data:=HERALD||"8"||"25"||M1||...
          ...||M25||TRAILER},8>
```

6.2 Single Connection Test Case

In this section, we give an example of verifying a CS architecture based test case. First, the TTCN specification of the test case and its EFSM description are presented. Second, a global state space is generated from a test verification system using the transport protocol specification, the underlying service provider, the TMP described in Section 6.1, and the EFSM representation of the test case. Third, the safety and liveness properties of the test case are specified and verified.

6.2.1 EFSM Model of the Test Case

For verification purpose we select a test case from a human designed CS architecture based test suite developed at the National Computing Center, UK [NCC 88]. The main dynamic behavior of the test case is shown in Fig. 6.6. For the sake of clarity, the subtrees of the test case, identified by a "+" sign in front of them, are not shown here. The functionality of the test case consists of two distinct phases: *preamble* and *test body*. The preamble part consists of five steps. In the first step, the test case establishes a transport connection between the LT and the TMP. In the second step, the LT sends a command TMPDU (TMPDU1) directing the TMP to set the default values of its counters, mode parameters, and stored items. In the third step, the LT sends a command TMPDU (TMPDU4) directing the TMP to set the value of the stored item S8 which is used as the "User_data" parameter in a T-CONNECT response primitive. The "User_data" parameter in a T-CONNECT response primitive becomes the "User_data" parameter in a Connect Confirm Transport Protocol Data Unit (CC TPDU). In the fourth step, the LT waits for acknowledgements of the previously sent two DT TPDUs containing the TMPDUs from the implementation. In the fifth step, the LT disconnects the transport connection. The objective of the preamble is to set S8 to a known value VAL.

Test Case Dynamic Behavior					
Identifier: ABCT2URA00 Group: Purpose: LT sends CR, receives CC with VAL octets of user data. Default: Def1 Comments :					
Nr.	Lab.	Behavior Description	Constraint	Verdict	Comment
		Test (VAL)			
		preamble			
		+ LT_con			Step1: Establish a transport connection.
		L!TMP1()			Step2: Send TMPDU1 in a DT TPDU.
		L!TMP4 (S8 := VAL)			Step3: Send TMPDU4 in a DT TPDU.
		+ Wait_for_ak			Step4: Wait for acknowledgements.
		+ LT_dis			Step5: Release the transport connection.
		body			
		+ preamble			
		L!CR	CR1		Initiate a connection establishment.
		Start (A, no_response)			Start a no_response timer.
		L?CC [user_data = VAL]	CC1	Pass	Connection is established.
		Cancel (A)			
		+ PO1/Postamble			Release the connection.
		?Timeout (A)		Fail	Implementation not responding
Comments: This test relies on the ability to control user data.					

Figure 6.6 A CS architecture based single connection test case taken from [NCC 88], Ref. ABCT2URA00.

In the *test body*, the LT initiates the establishment of a transport connection with the TMP by sending a Connection Request (CR) TPDU and waiting for a CC TPDU. If the LT receives a CC TPDU with VAL as the "User_data" parameter, then the LT assigns a Pass test verdict and releases the transport connection. Otherwise, if the LT receives a CC with the "User_data" value different from VAL or it receives a different TPDU, or a timeout occurs, then the LT assigns a Fail test verdict and terminates its operations.

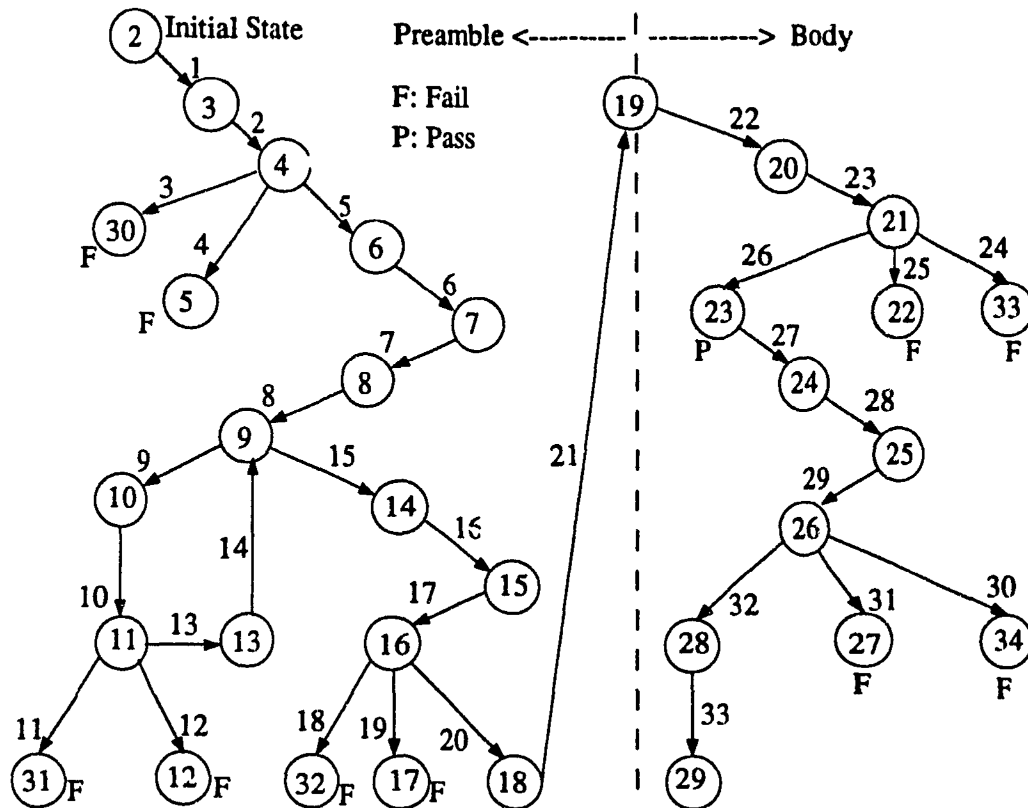


Figure 6.7 EFSM representation of the test case in Fig. 6.6

The transitions of the above test case are given below.

```
Following are the verdict TAGS of the states:
verdict(5) = verdict(12) = verdict(17) = verdict(22) = Fail

verdict(30) = verdict(31) = verdict(32) = Fail
verdict(33) = verdict(34) = Fail verdict(23) = Pass

Initialization:

VAL := "test_data"
/* Parameters of a TCONresp primitive.*/
TS5 := 1             /* Quality of service*/
TS6 := "you"         /* Called_address*/
TS7 := No            /* Expedited data option*/
TS8 := VAL           /* User_data*/
Called_address := "you"
Calling_address := "me"
```

```
Exp_option := "No"
Qos := "1"
User_data0 := "any_data"
seqrecak := 0
seqsendt := 0
```

Transitions of the LT EFSM:

```
LT_1:  <2,3,L!NDTreq(CR(Called_addr,Calling_addr,
        Exp_option,Qos,User_data0)),T,{},1>
LT_2:  <3,4,Start(A,no_response),T,{},1>
LT_3:  <4,30,L?OTH, T, {}, 3>
LT_4:  <4,5,?Timeout(A),T,{},2>
LT_5:  <4,6,L?NDTind(CC),T,{},1>
LT_6:  <6,7,Cancel(A),T,{},1>
LT_7:  <7,8,L!NDTreq(DT(User_data1,EOT)),
        {M1:=A4,M2:=A0,..,M10:=A0,M11:=A5,M12:=A5,
         M13:=A0,...,M25:=A0,
         User_data1:=TMP1(M1,M2,..,M25),
         EOT:=True, seqsendt:=0},1>
LT_8:  <8,9,L!NDTreq(DT(User_data2,EOT)),
        {User_data2:=TMP4(TS5,TS6,TS7,TS8),
         EOT:=True, seqsendt:=1},1>
LT_9:  <9,10,Null,[seqrecak < seqsendt],{}, 1>
LT_10: <10,11,Start(B,wait_ak),T,{},1>
LT_11: <11,31,L?OTH, T, {}, 3>
LT_12: <11,12,?Timeout(B),T,{},2>
LT_13: <11,13,L?NDTind(AK),T,{seqrecak:=AK.seqno},1>

LT_14: <13,9,Cancel(B),T,{},1>
LT_15: <9,14,Null,[seqrecak >= seqsendt],{},1>
LT_16: <14,15,L!NDTreq(DR(Reason, User_data3)),T,
        {Reason:="normal_disconnect", User_data3:=Null},1>
LT_17: <15,16,Start(A,wait_dc),T,{},1>
LT_18: <16,32,L?OTH, T, {}, 3>
LT_19: <16,17,?Timeout(A),T,{},2>
LT_20: <16,18,L?NDTind(DC),T,{},1>
LT_21: <18,19,Cancel(A),T,{},1>
LT_22: <19,20,L!NDTreq(CR(Called_addr,Calling_addr,
        Exp_option,Qos,User_data0)),T,{},1>
LT_23: <20,21,Start(A,no_response),T,{},1>
LT_24: <21,33,L?OTH, T, {}, 3>
LT_25: <21,22,?Timeout(A),T,{},2>
LT_26: <21,23,L?NDTind(CC),[CC.User_data=VAL],{},1>
LT_27: <23,24,Cancel(A),T,{},1>
```

```
LT_28: <24,25,L!NDTreq(DR(Reason, User_data3)),T,
       {Reason:="normal_disconnect", User_data3:=Null},1>
LT_29: <25,26,Start(A,wait_dc),T,{},1>
LT_30: <26,34,L?OTH, T, {}, 3>
LT_31: <26,27,?Timeout(A),T,{},2>
LT_32: <26,28,L?NDTind(DC),T,{},1>
LT_33: <28,29,Cancel(A),T,{},1>
```

6.2.2 Model Generation

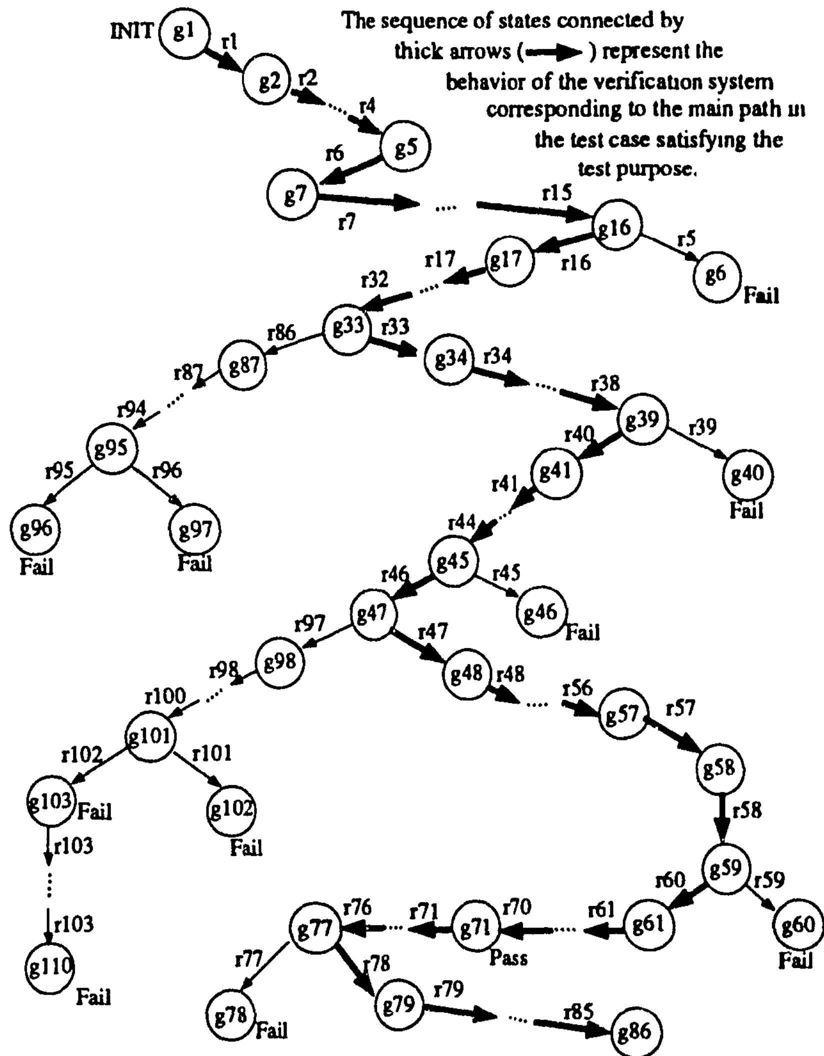


Figure 6.8 Structure of the global state space.

Using the global state space generation algorithm in Chapter 3, we generate the global state space of the test verification system, shown in Fig. 6.2, by using

the LT-EFSM in Fig. 6.7, the USP-EFSM in Fig. 6.4, the S-EFSM in Fig. 6.3, and the TMP-EFSM in Fig. 6.5. The global state space contains 110 states and 109 transitions. We show the entire global state space in EFSM notation and only partially in graphical form in Fig. 6.8. The detailed state descriptions and the transitions of the global state space are given in **Appendix 3**.

Events between two EFSMs are exchanged in the form of I/O diagrams. For example, the representation of the NDTreq event sent by the test case in step 3 of the preamble is shown in Fig. 6.9. In the receiving EFSM, the I/O diagram is traversed from its root downward to extract the values of its fields while executing an assignment function or evaluating a boolean condition.

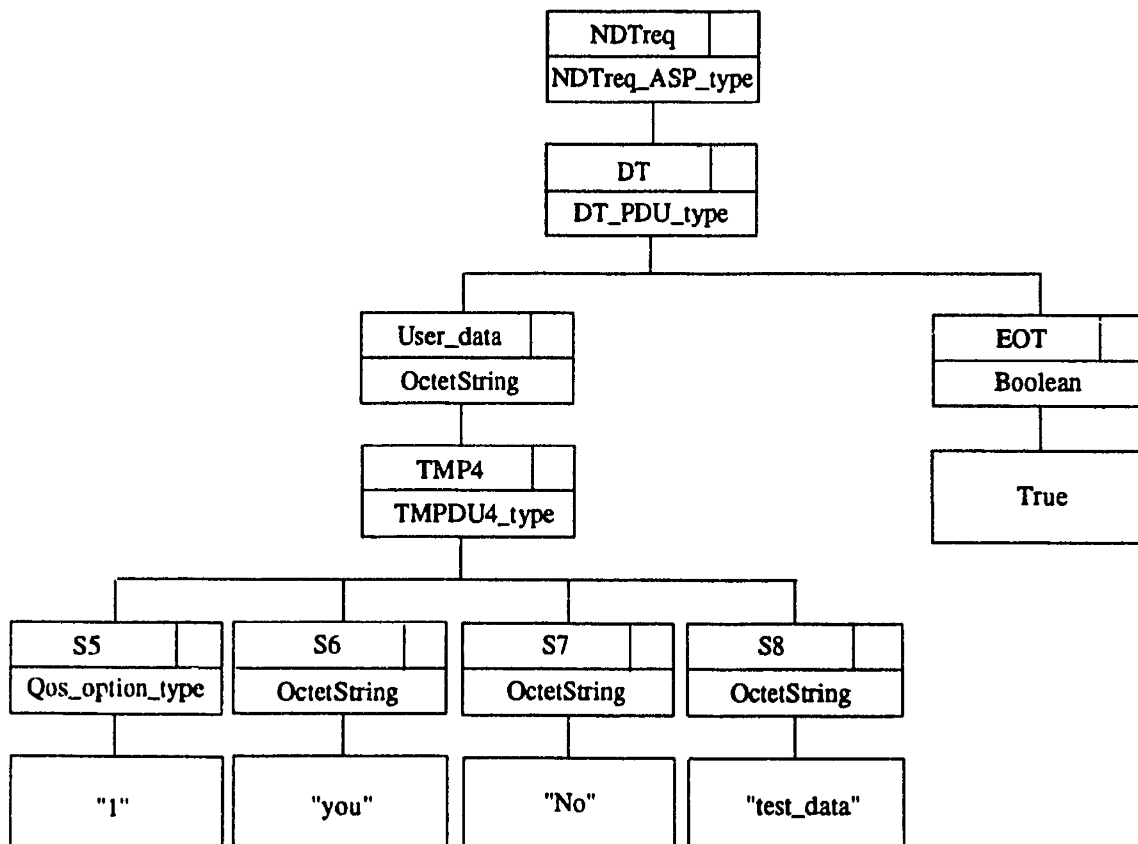


Figure 6.9 Representation of an event in a queue as an I/O diagram.

6.2.3 Verification of Safety and Liveness Properties

The global state space of the TVS, shown in Fig. 6.8, contains one initial state and 10 final states. Therefore, there are 10 paths from the initial state to the final states. Let us denote a path by a function $path(g_i, g_j)$, where g_i and g_j are the initial and a final state, respectively. The 10 paths in the global state space are denoted by $path(g_1, g_6)$, $path(g_1, g_{96})$, $path(g_1, g_{97})$, $path(g_1, g_{40})$, $path(g_1, g_{46})$, $path(g_1, g_{102})$, $path(g_1, g_{110})$, $path(g_1, g_{60})$, $path(g_1, g_{78})$, and $path(g_1, g_{86})$.

A. Safety Properties

In the given test case, there are a few transmission safety errors. Let us consider the property

$$INIT \models AG(AFTER(Ssend(N, NDTreq(CC))) \mapsto AFTER(Treceive(L, NDTind(CC)))).$$

The predicate AFTER(Ssend(N, NDTreq(CC))) holds in the global state g_{14} but the predicate AFTER(Treceive(L, NDTind(CC))) holds only in state g_{17}. Therefore, the property

$$AFTER(Ssend(N, NDTreq(CC))) \mapsto AFTER(Treceive(L, NDTind(CC)))$$

holds on all the paths except $path(g_1, g_6)$. That is, the safety property does not hold on all possible executions of the test system. This safety error arises because of a timeout in the test case as explained in the following.

From the global state g_{16}, there are two possible transitions, *r5* and *r16*, which are derived from the LT_4 and LT_5 transitions in the test case EFSM. Transition LT_4 represents a timeout event as an alternative to transition LT_5 which is a L?NDTind(CC) event. If the timeout occurs before the test case receives the NDTind(CC) event from PCO L, then the safety error would occur.

Consider another transmission safety property

$$INIT \models AG(AFTER(Ssend(N, NDTreq(AK))) \mapsto AFTER(Treceive(L, NDTind(AK)))).$$

The predicate AFTER(Ssend(N,NDTreq(AK))) holds in state g_{35}, but the predicate Treceive(L, NDTind(AK)) holds only in state g_{41}. Therefore, the formula $AFTER(Ssend(N, NDTreq(AK))) \mapsto AFTER(Treceive(L, NDTind(AK)))$ does not hold on the path $path(g_1, g_{40})$ leading to a transmission safety error in the test system. This error is also due to the timeout in the test case represented by the LT_12 transition.

The other transmission safety properties, which are not satisfied due to the test case timeout transitions LT_19, LT_25, and LT_31, are

$$INIT \models AG(AFTER(Tsend(L, NDTreq(DR))) \mapsto AFTER(Sreceive(N, NDTind(DR)))),$$

$$INIT \models AG(AFTER(Tsend(L, NDTreq(CR))) \mapsto AFTER(Sreceive(N, NDTind(CR)))), \text{ and}$$

$$INIT \models AG(AFTER(Tsend(L, NDTreq(DC))) \mapsto AFTER(Sreceive(N, NDTind(DC)))),$$

respectively. It is observed that if the timeouts in the test case are appropriately tuned, then all the above transmission safety properties are satisfied. That is, in a qualitative sense, if the timeouts are well tuned, then the global transitions *r5, r39, r45, r59*, and *r77* would be absent from the global state space and the safety properties would hold in the remainder of the state space.

Two other transmission safety properties, which do not hold in the test system, are:

$$INIT \models AG(AFTER(Ssend(N, NDTreq(DR))) \mapsto AFTER(Treceive(L, NDTind(DR)))) \text{ and}$$

$$INIT \models AG(AFTER(Ssend(N, NDTreq(AK))) \mapsto$$

$AFTER(Treceive(L, NDTind(AK))))$.

The predicate AFTER(Ssend(N,NDTreq(DR))) holds in the state g_{89}, but the predicate does not hold in any of the states following g_{89}. To find out the cause of the error, we analyze the test system in the following.

The test system arrives at global state g_{89} from state g_{33} due to the global transition sequence {*r86, r87, r88*} derived from the transition sequence {SPEC_7, SPEC_8, SPEC_9} in the protocol specification EFSM. The sequence {SPEC_7, SPEC_8, SPEC_9} means the specification nondeterministically releases an open connection by sending a TDISind primitive to its service user and a DR TPDU to its peer entity, which is the test case EFSM in this situation. The safety error denotes a design error in the test case in the sense that the test case is not ready to receive the DR TPDU from the protocol specification.

Similarly, the second transmission safety error is due to a design error in the test case in the sense that after the establishment of a transport connection, the test case is not ready to receive spontaneous acknowledgement (AK) TPDUs from the protocol specification.

B. Liveness Property

To verify the liveness property of the test case, we first express the test purpose as a temporal formula. As a part of the test case, the test purpose is specified in a natural language as follows:

i. *LT sends CR,*

ii. *LT receives CC with VAL octets of user data.*

In the following, corresponding to each part of the test purpose, we state a temporal formula:

i. *AFTER(Tsend(L, NDTreq(CR))),*

ii. *(AFTER(Treceive(L, NDTind(CC))) and (NDTind.CC.User_data = VAL)).*

We compose these basic test purposes using the SEQ operator to give rise to a formula for the entire test purpose as follows:

$f_1 = (AFTER(Tsend(L, NDTreq(CR)))\ SEQ$
$(AFTER(Treceive(L, NDTind(CC))) \wedge (NDTind.CC.User_data = VAL))).$

Then, the liveness property of the test case is stated as $INIT \models (f_1 \mapsto (verdict = Pass))$.

The liveness property is satisfied by the sequence of global states denoted by the *path([illegible], g86)*. The predicate (AFTER(Tsend(L, NDTreq(CR)))) holds in state *g58* and the predicates (AFTER(Treceive(L, NDTind(CC)))) and (NDTind.CC.User_data = VAL) hold in a subsequent state *g71* where the predicate (verdict =Pass) also holds.

All other paths do not satisfy the test purpose. The paths reachable due to the time out transitions (*r5*, *r39*, *r45*, *r59*, and *r77*) can be avoided from the global states by appropriately tuning the timeout intervals. However, the test purpose is not satisfied if the implementation behaves nondeterministically.

CHAPTER 7

VERIFICATION OF PARALLEL TEST CASES

There are three reasons to define parallelism in a test specification language such as TTCN. First, parallelism in a test specification language makes it easier to specify test behavior at several PCOs and non-sequential protocol behavior. Second, it provides language constructs to describe explicitly the cooperation of distributed components of a test architecture. Finally, parallelism in a test notation is essential for testing some aspects of an implementation such as multiple connection support capability.

In Section 7.1, we present a test architecture incorporating parallelism. A methodology for verifying a parallel test case is discussed in Section 7.2 and an example of verifying a parallel test case is given in Section 7.3.

7.1 Parallel Test Architecture

A parallel test architecture for multi-party testing is shown in Fig. 7.1 [ISO9646-3E]. Conceptually, a tester consists of a *Main Test Component (MTC)* and zero or more *Parallel Test Components (PTC)*. Test components may communicate with each other by exchanging *Coordination Messages (CM)* through *Coordination Points (CP)*. A test component may communicate with an implementation under test via points of control and observation. A test system may have any number of PCOs, but only one test component shall be connected to a particular PCO. Coordination points always connect exactly two test components. The implementation under test provides a connection between a pair of Lower Tester and Upper Tester.

The MTC is intended to fulfill the role of the Master Lower Tester. Its behavior is described in the first tree of a test case behavior description table. The MTC is responsible for:

- creating and terminating PTCs,
- managing coordination points that exist between itself and PTCs,
- receiving local test verdicts from PTCs, and
- computing the overall test verdict using the local verdicts.

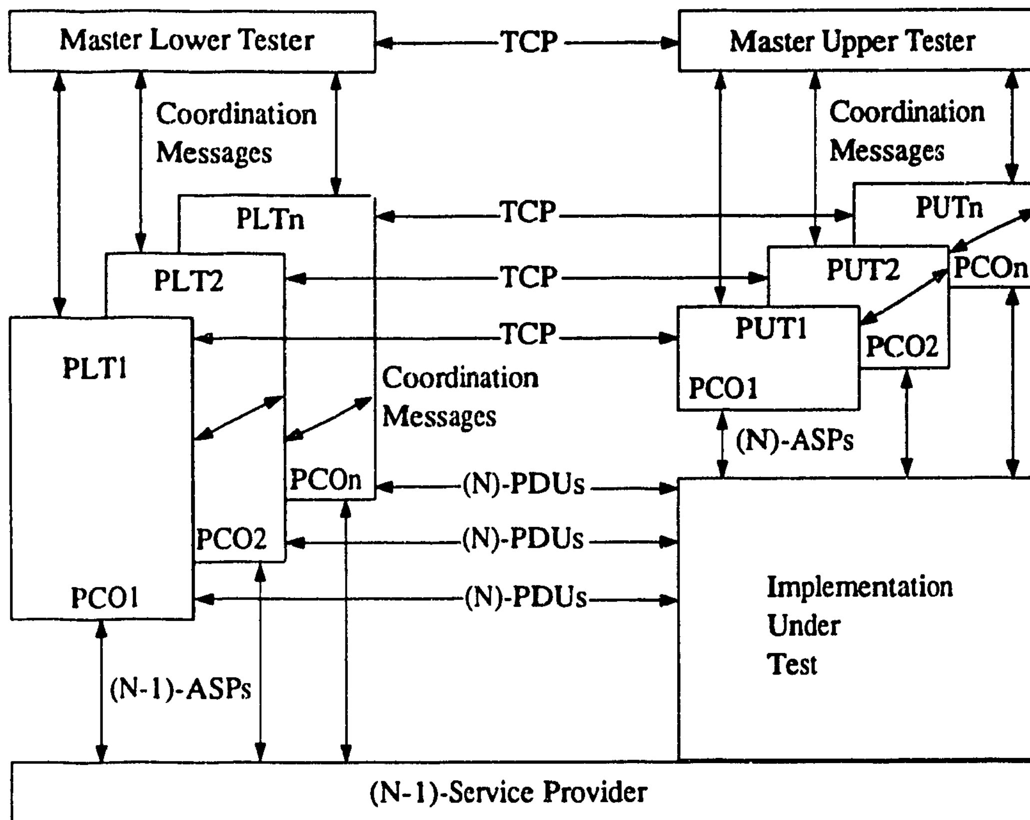


Figure 7.1 A Multi-party Test Architecture.

Parallel Test Components are intended to fulfill the role of the Parallel Lower Testers (PLT) or the Parallel Upper Testers (PUT). Their behavior are described in trees of the test case behavior description table, following the first tree describing the Main Test Component. A verdict assigned in a PTC has only local significance. Each PTC is responsible for:

- assigning its local verdict, and
- sending a message containing its local verdict to the MTC.

Multi-party testing requires multiple parallel lower testers, a master lower tester, and possibly multiple upper testers. In a multi-party test architecture, conceptually, it is possible to view the IUT as a multi-peer IUT, which handles more than one interface, that is, connection or association, at a time. A multi-peer IUT is called *homogeneous* if all sets of protocols on the different interfaces are identical, otherwise it is called *heterogeneous* [BEWI 90]. In this thesis, we consider the verification of test cases for testing a homogeneous IUT.

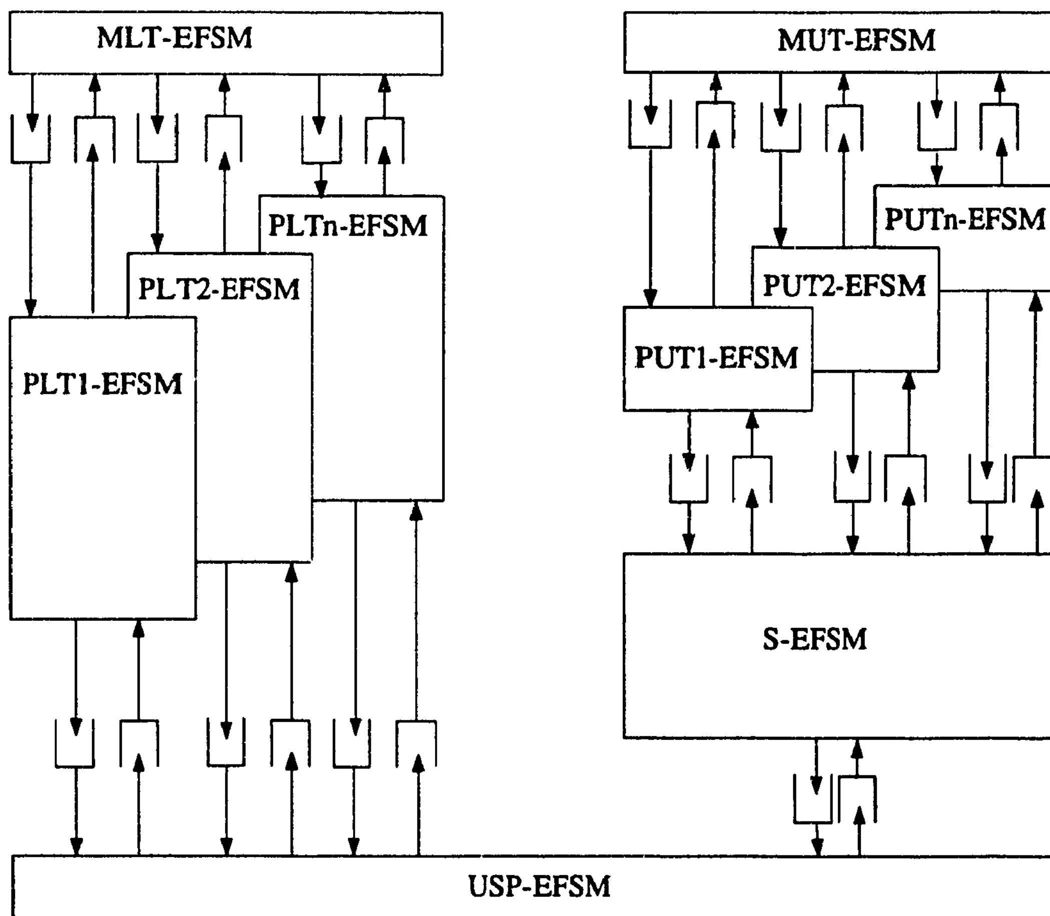
In the given multi-party test architecture, it is interesting to observe that the interconnection structure of a pair of PLT and PUT, the service provider, and the IUT resemble the DS test architecture. Since the multi-party test architecture does not impose a one-to-one correspondence between a test component - Lower or Upper - and a PCO, in principle, it should be possible to define any basic test architecture - CS, DS, RS, and LS - between a pair of PLT and PUT. However, a Test Management Protocol used in the CS architecture in the multiparty test architecture must be designed such that it communicates with the Master Upper Tester in addition to communicating with the IUT.

An advantage of using different basic test architectures between different pairs of PLT and PUT is that the resulting multi-party test architecture achieves total error detection capability in contrast to the partial and complementary error detection capabilities of the individual basic test architectures [SARI 89].

7.2 Parallel Test Case Verification

Verification of a multiplexing test case running in a parallel test architectural framework can be done by first deriving a test verification system from the test architecture. A test verification system corresponding to the parallel test architecture is given in Fig. 7.2. The test verification system is derived from the parallel test architecture by representing the Master Lower Tester (MLT), Master Upper Tester (MUT), PLTs, PUTs, and the service provider by their EFSM representations. The

IUT is replaced by the EFSM model of its protocol specification. Each PCO, CP, and interaction point is replaced by two FIFO channels. If there are n simultaneous connections being opened by the test system, then there are $(2n + 4)$ EFSMs in the verification system. The $2n$ factor is due to a pair of PLT and PUT for each of the n connections, and one EFSM for each of MLT, MUT, service provider, and the protocol specification.



Legends: MLT-EFSM = Master Lower Tester EFSM
MUT-EFSM = Master Upper Tester EFSM
PLTn-EFSM = Parallel Lower Tester-n EFSM
PUTn-EFSM = Parallel Upper Tester-n EFSM
S-EFSM = Specification EFSM
USP-EFSM = Underlying Service Provider EFSM

Figure 7.2 A Test Verification System for Parallel Test Architecture

To verify a parallel test case, one can directly use the verification technique discussed in Chapters 3 and 4, that is:

i. Derive a test verification system from a parallel test architecture as discussed above.
ii. Generate a global state space from the test verification system with a set of atomic predicates attached to each global state.
iii. Express the test case safety and liveness properties as temporal formulas.
iv. Verify the test case properties on the global state space.

However, in practice, because of the large number of EFSMs in the verification system, there is a possibility of state space explosion while generating the global state space model of the verification system. To contain the state space explosion, we adopt the following approach.

Let us represent the verification of a single connection test case t in a test verification system TVS by the function $Verify(t)|_{TVS}$. Since the verification process involves the verification of the safety properties and the verification of the liveness property, we express the verification function as follows:

$Verify(t)|_{TVS} = Safety(t)|_{TVS} \cup Liveness(t)|_{TVS}$,

where the functions $Safety(t)|_{TVS}$ and $Liveness(t)|_{TVS}$ represent the verification of the safety and the liveness properties in TVS, respectively.

Assume that a multiple connection test case T is represented as the parallel composition of n single-connection test cases, that is:

$T = t_1 \parallel t_2 \parallel \ldots \parallel t_n$.

Then, we define the verification of the multiple connection test case T in a parallel test verification system TVS^P corresponding to a parallel test architecture as follows:

$$
\begin{aligned}
Verify(T)|_{TVS^P} &= Safety(T)|_{TVS^P} \cup Liveness(T)|_{TVS^P} \\
&= (Safety(t_1)|_{TVS1} \cup ... \cup Safety(t_n)|_{TVSn}) \cup \\
&\quad (Liveness(t_1)|_{TVS1} \wedge ... \wedge Liveness(t_n)|_{TVSn})
\end{aligned}
$$

where $TVS1, ..., TVSn$ are n distinct single connection test verification systems derived from TVS^P which contains n pairs of Upper/Lower Testers. A well structured parallel test system with the following characteristics makes it possible to decompose a parallel test verification system TVS^P to a set of single-connection test verification system $\{TVS1, ..., TVSn\}$:

i. The Master Test Component communicates with the Parallel Test Components only at the beginning and end of their executions. At the beginning, the MTC creates a PTC and sends some parameters and receives the local verdict returned by a PTC at the end of execution of the PTC.

ii. The Parallel Test Components do not communicate among themselves.

Therefore, verification of a multiplexing test case can be done as follows.

i. Obtain n single connection test verification systems from the parallel architecture test system.

ii. To verify the safety properties of the multiplexing test case, verify the safety properties of the n single connections separately.

iii. To verify the liveness property of the multiplexing test case, express the test purpose as a temporal formula and verify that the test purpose temporal formula is satisfied by arriving at a Pass verdict in all the n single connection test verification systems. This is because, the test purpose of a multiplexing test case must take the local Pass verdicts of all the n connections into account.

7.3 Example of Verifying a Parallel Test Case

To test the multiplexing capability of an implementation, a test case must establish at least two connections. The CS architecture based transport test suite in [NCC 88]

contains four multiplexing test cases. In this section, we give an example of verifying one of the multiplexing test cases. First, the TTCN specification of the test case and its EFSM description are presented. Second, a global state space is generated from a test verification system using the transport protocol specification, the underlying service provider, the TMP described in Section 6.1, and the EFSM representation of the test case. Third, the verdict composition mechanism in a multiplexing test case is illustrated.

7.3.1 EFSM Model of the Test Case

The TTCN specification of a multiplexing test case chosen from the human designed test suite [NCC 88] rewritten using the extended TTCN [ISO 9646–3E] is shown in Fig. 7.3. The test case contains a *Main Test Component (MTC)* which creates two *Test Components (TC)* denoted by *TREE1* and *TREE2*. A TC returns a test verdict to the MTC after completing its execution. The MTC computes the final verdict after receiving verdicts from both the TCs. Both the TCs have identical behavior except their source and destination addresses.

Each TC contains *four* major steps. In the first step, a TC establishes a connection with an Upper Tester. In the second step, which is a data transfer step, five TMPDUs are sent to the Upper Tester. TMPDU1 initializes the Upper Tester, TMPDU3 contains a value to be assigned to the S1 stored item in the Upper Tester, TMPDU4 contains values to be assigned to the S5, S6, S7, and S8 stored items in the Upper Tester, TMPDU5 contains values for the S9, S10, S11, S12, and S13 stored items, and the command TMPDU8 requests the Upper Tester to send all the mode parameters in a response TMPDU denoted by TMPDU8r. After sending all five TMPDUs in separate five DT TPDUs, the TC waits for an AK TPDU. In the third step, the TC waits for a DT TPDU containing a TMPDU8r. If the TC receives a TMPDU8r, then it returns a Pass test verdict. In the fourth step, the TC closes the connection. The TC returns a

Fail test verdict if the connection cannot be established, an AK TPDU is not received after sending five DT TPDUs, or there is any error while releasing a connection.

To verify the test case, we first construct an LT-EFSM for each test component. Second, we generate a model for the test verification system by considering one LT-EFSM at a time. Third, the transmission, reception, and synchronization safety properties of each test component are verified. Fourth, the final test verdict assigned by the main test component is computed by combining the verdicts returned by both the test components.

In this example, since both the test components are identical, we construct only one LT-EFSM, as shown in Fig. 7.4. There are 31 states and 30 transitions in the EFSM. The test component assigns a Fail verdict if control reaches one of the states in the set {5, 12, 20, 21, 26, 27, 30, 31} and assigns a Pass verdict if control reaches the state 29.

Test Case Dynamic Behavior				
Reference: ABCT2MPA00 Identifier: CS_Example_2 Purpose: Each TC establishes a connection, transfers data with a command to make a reply, and receives a reply TMPDU. Default: Def1				
Label	**Behavior Description**	**Constr.**	**Verdict**	**Comment**
	Main Test Component			
	CREATE(PTC1, TREE1)			Create 1st. test component
	CREATE(PTC2, TREE2)			Create 2nd. test component
	CP1? result(R1:=result)			Receive result from 1st. component
	CP2? result(R2:=result)			Receive result from 2nd. component
	[(R1="PASS") AND (R2="PASS")]		Pass	If both components Pass the test
	[(R1="FAIL") OR (R2="FAIL")]		Fail	If any component Fails the test
	TREE1			**First connection**
	(set_A)			Set the address values for 1st. conn.
	+Con_Data_Disconn			Invoke the connection
	TREE2			**Second connection**
	(set_B)			Set the address values for 2nd. conn.
	+Con_Data_Disconn			Invoke the connection
	Con_Data_Disconn			
	+LT_trans_con			Establish a connection
	L!TMP1()			Set mode parameters
	L!TMP3(S1:="")			Set DR parameters
	L!TMP4(S5:="",S6:="you", S7:=1,S8:="")			Set CC parameters
	L!TMP5(S9:="me",S10:="you", S11:=1,S12:="",S13:="")			
	L!TMP8()			Request mode parameters
	+Wait_for_ak			Wait for acknowledgements
	+Wait_for_mode_params()		Pass	Wait for the response TMPDU8r
	+P01_mult/postamble			Release the connection

Figure 7.3 A CS architecture based multiple connection test case.

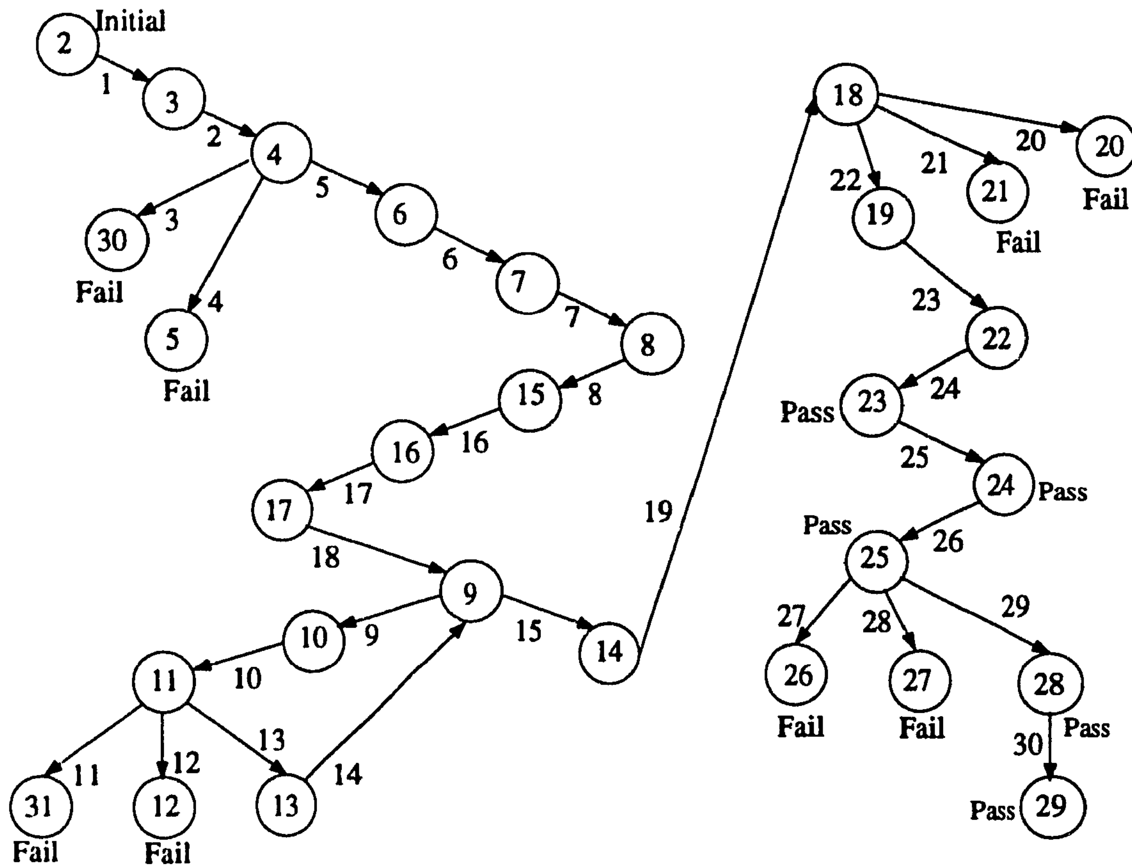


Figure 7.4 EFSM Model of the test case in Fig. 7.3.

The transitions of the above test case are shown below.

```
Following are the verdict TAGS of the states:
verdict(5)=verdict(12)=verdict(20)=verdict(21)=Fail
verdict(26)=verdict(27)=verdict(30)=verdict(31)=Fail
verdict(23)=verdict(24)=verdict(25)=Pass
verdict(28)=verdict(29) = Pass

Initialization:

TS1:= "",TS5 := 1,TS6 := "you",TS7 := No,
TS8 := "",TS9 := "me",TS10 := "you",
TS11 := "1",TS12 := "1",TS13 := "",
Called_address := "you",Calling_address := "me"
Exp_option := "No",Qos := "1",
User_data0 := "any_data",seqrecak := 0,seqsendt := 0
```

```
------------------------------------------------------------
Transitions of the LT EFSM
------------------------------------------------------------
LT_1:  <2,3,L!NDTreq(CR(Called_addr,Calling_addr,
                        Exp_option,Qos,User_data0)),
        T,{},1>
LT_2:  <3,4,Start(A,no_response),T,{},1>
LT_3:  <4,30,L?OTH, T, {}, 3>
LT_4:  <4,5,?Timeout(A),T,{},2>
LT_5:  <4,6,L?NDTind(CC),T,{},1>
LT_6:  <6,7,Cancel(A),T,{},1>
LT_7:  <7,8,L!NDTreq(DT(User_data1,EOT)),
        {M1:=A4,M2:=A0,..,M10:=A0,M11:=A5,
         M12:=A5,M13:=A0,...,M25:=A0,
         User_data1:=TMP1(M1,M2,..,M25),
         EOT:=True, seqsendt:=1},1>
LT_8:  <8,15,L!NDTreq(DT(TMP3(TS1),EOT)),
        {seqsendt:=2},1>
LT_16:  <15,16,L!NDTreq(DT(User_data2,EOT)),
         {User_data2:=TMP4(TS5,TS6,TS7,TS8),
         EOT:=True, seqsendt:=3},1>
LT_17:  <16,17,L!NDTreq(DT(User_data2,EOT)),
         {User_data2:=TMP5(TS9,TS10,TS11,TS12,TS13),
         EOT:=True, seqsendt:=4},1>
LT_18:  <17,18,L!NDTreq(DT(TMP8(),EOT)),
         {seqsendt:=4},1>
LT_9:  <9,10,Null,[seqrecak < seqsendt],{}, 1>
LT_10: <10,11,Start(B,wait_ak),T,{},1>
LT_11: <11,31,L?OTH, T, {}, 3>
LT_12: <11,12,?Timeout(B),T,{},2>
LT_13: <11,13,L?NDTind(AK),T, {seqrecak:=AK.seqno},1>
LT_14: <13,9,Cancel(B),T,{},1>
LT_15: <9,14,Null,[seqrecak >= seqsendt],{},1>
LT_19: <14,18,i,T,{},1>
LT_20: <18,20,L?OTHERWISE,T,{},2>
LT_21: <18,21,i, T, {}, 3>
LT_22: <18,19,L?NDTind(DT(TMP8)),T,{},1>
LT_23: <19,22,L!NDTreq(AK(1),T,{},1>
LT_24: <22,23,i,T,{},1>
LT_25: <23,24,L!NDTreq(DR(Reason, User_data3)),T,
        {Reason:="normal_disconnect",
        User_data3:=Null},1>
LT_26: <24,25,i,T,{},1>
LT_27: <25,26,L?OTHERWISE, T, {}, 2>
LT_28: <25,27,i,T,{},3>
```

```
LT_29: <25,28,L?NDTind(DC),T,{},1>
LT_30: <28,29,Cancel(A),T,{},1>
-----------------------------------------------------------------
```

7.3.2 Test Verification System for the Parallel Test Case

A test verification system for the multiple connection test case is shown in Fig. 7.5. The MLT-EFSM is derived from the TTCN specification of the Main Test Component in the test case, PLT1–EFSM is derived from the TTCN specification of TREE1, and PLT2–EFSM is derived from TREE2 of the test case. Both PUT1–EFSM and PUT2–EFSM are identical in behavior and correspond to the Test Management Protocol discussed in Section 6.1. The S-EFSM and the USP-EFSM are also discussed in Section 6.1.

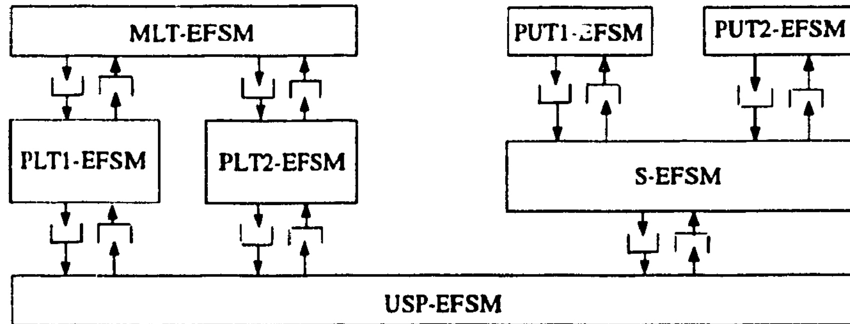


Figure 7.5 A Test Verification System for the test case in Fig. 7.3.

Generation of a global state space from the TVS shown in Fig. 7.5 would result in state space explosion. Therefore, as discussed in Section 7.2, we derive state spaces for individual connections by deriving a test verification system for each connection as shown in Figs. 7.6 and 7.7.

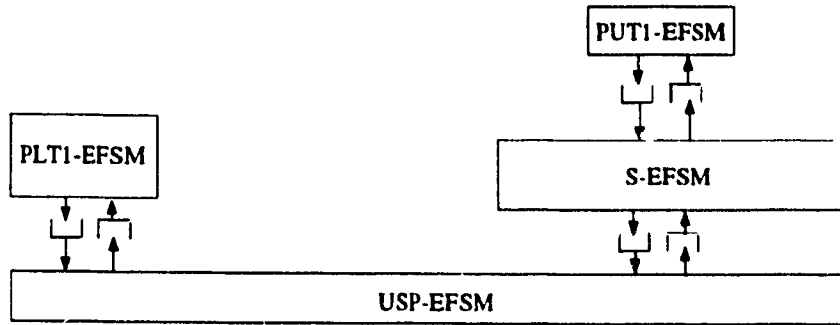


Figure 7.6 A Test Verification System for first connection in Fig. 7.3.

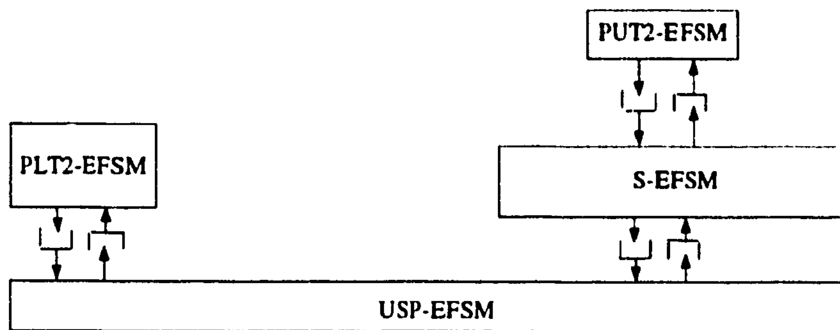


Figure 7.7 A Test Verification System for second connection in Fig. 7.3.

7.3.3 Model Generation

Using the global state space generation algorithm in Chapter 3, we generate the global state space for each of the test verification systems in Figs. 7.6 and 7.7 by using the PLT-EFSM in Fig. 7.4, the Underlying Service Provider EFSM in Fig. 6.4, the protocol specification EFSM in Fig. 6.3, and an Upper Tester EFSM derived from a Test Management Protocol in Fig. 6.5. Since both the verification systems in Figs. 7.6 and 7.7 are identical, in the following, we present the global state space for only one of them. The global state space contains 116 states and 115 transitions. We show only the structure of the global state space in Fig. 7.8. The detailed state description and the transitions of the global state space are given in **Appendix 4**.

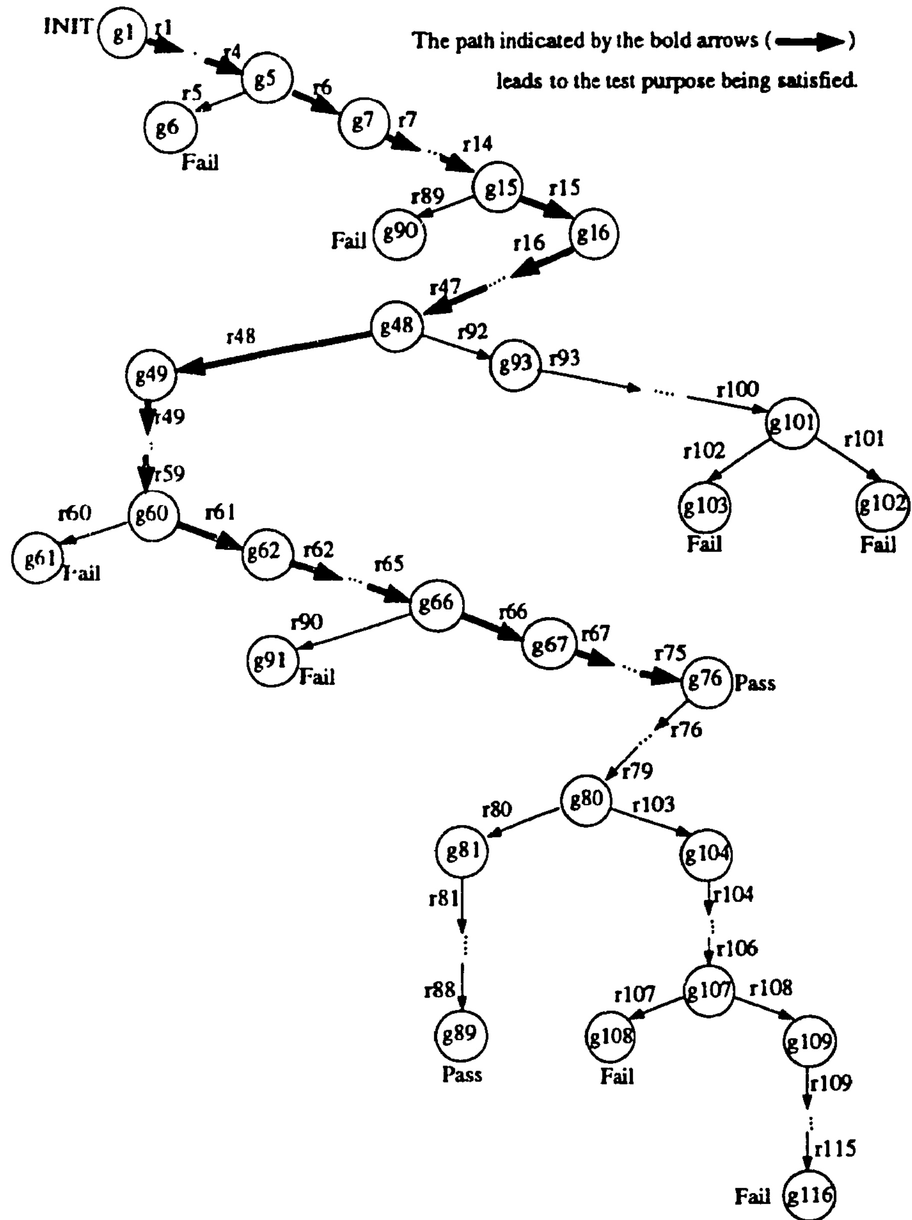


Figure 7.8 Structure of the global state space.

7.3.4 Verification of Safety and Liveness Properties

Safety Properties

Verification of the safety properties of the multi-party test case consists of verifying the safety properties of the individual test components (TC). Since in the

test case example under consideration both the test components are identical, in the following we verify the safety properties of one test component only. In each of the test components, there are a few safety errors. Let us consider the property:

$$INIT \models AG(AFTER(Ssend(N, NDTreq(DR))) \mapsto AFTER(Treceive(L, NDTind(DR)))).$$

The predicate AFTER(Ssend(N, NDTreq(DR))) holds in the global state g95 which is a descendant of state g93 in the global state space diagram in Fig. 7.8. From the global state g95, execution can proceed along two paths leading to the two global states g102 and g103. However, the predicate AFTER(Treceive(L, NDTind(DR))) does not hold in any of the states following g95. Therefore, the above stated safety property does not hold. The cause of the safety error is explained as follows.

While the test verification system is in the global state g48, the Lower Tester is in its local state 9 just after sending four DT TPDUs and is about to enter into a loop in which it expects to receive AK TPDUs from the protocol specification, the service provider is in state 1 waiting for any input events from the Lower Tester or the protocol specification, the protocol specification is in state 10 representing a connection open state, and the TMP is in state OPEN. Then the test system moves to state g93 after a nondeterministic internal transition takes the protocol EFSM from state 10 to 11. The protocol EFSM initiates the disconnection of the transport connection by sending a TDISind event to the TMP and a NDTreq(DR) event to the Lower Tester through the service provider. The protocol EFSM sends the TDISind in state 11 and moves to state 12 and then sends the NDTreq(DR) event to the service provider and moves to state 13. These two actions of the protocol EFSM takes the test system through the global state sequence {g93, g94, g95}. The TMP consumes the event TDISind event sent by the protocol EFSM and moves to the IDLE state through a sequence of internal transitions, thereby taking the test system through the state sequence {g95,

g96, g97}. When the service provider transfers the NDTreq(DR) event from the protocol EFSM to the input channel of the Lower Tester, the test system goes through the state sequence {g97, g98, g99}. After this point, the Lower Tester enters into a loop initiating a timer to receive AK TPDUs from the protocol EFSM. Since the protocol EFSM has initiated a nondeterministic termination of the connection, there are two alternative scenarios in the Lower Tester: occurrence of an OTHERWISE event and occurrence of a timeout. Both these events take the test system to states g102 and g103, respectively. Therefore, the desired predicate AFTER(Treceive(L, NDTind(DR))) is never satisfied after state g95 and hence the safety error occurs. A design deficiency in the test case in the form of its inability to handle nondeterministic termination of the transport connection causes the safety error to occur.

Next, let us consider the following safety property:

$$INIT \models AG(AFTER(Ssend(N, NDTreq(AK))) \mapsto AFTER(Treceive(L, NDTind(AK)))).$$

The predicate AFTER(Ssend(N, NDTreq(AK))) holds in global state g56, which is a descendant of state g49 and an ancestor of g60. Control can proceed along two sequences of states starting with g56: {g56, g57, g58, g59, g60, g62} and {g56, g57, g58, g59, g60, g61}. Both these paths execute the same event sequences until reaching state g60, after which point they execute different events. Global state g60 refers to the LT-EFSM's local state 11 where the LT waits for an NDTind(AK) event from the service provider. There is also the possibility of the occurrence of a timeout in state 11. Because of the occurrence of the timeout, the predicate AFTER(Treceive(L, NDTind(AK))) holds in global state g62 only, that is, it does not hold on all paths reachable from g56. Hence the safety error occurs.

The first error is due to the nondeterministic disconnection of the transport connection by the protocol specification. To avoid any safety error due to the first

cause, a test case must be designed to accept events nondeterministically generated by the protocol specification. The second "error" is due to a qualitative analysis of timers in the test case. It indicates that the timers in test cases must have sufficiently long durations. As such the conclusion is that a diagnostic message rather than an error message should be generated during the verification process.

Liveness Property

Since the liveness property of the test case relates the satisfaction of the entire test purpose with the assignment of a Pass verdict, for the test purpose to be satisfied it is imperative that the purpose of each TC is satisfied with the TC returning a Pass verdict. In the following, we explain the computation of final verdict by the main test component.

The overall test verdict assigned by the test case to an implementation depends on the verdicts assigned by each test component. Since, both the connections are independent and run in parallel, the final test verdicts are computed by taking the product of the possible set of verdicts over each connection. In the example under consideration, referring to the global state space in Fig. 7.8, each state in the set VS = {g6, g61, g89, g90, g91, g102, g103, g108, g116} returns a verdict to the test coordinator. Only the state g89 returns a Pass verdict and all other states return a Fail verdict. In this example, since both the connections are identical, we compute $(VS \times VS)$ as follows.

```
verdict(g6,X)     := Fail for all X in VS,
verdict(g61,X)    := Fail for all X in VS,
verdict(g89,g89)  := Pass,
verdict(g89,X)    := Fail for all X, except g89, in VS,
verdict(g90,X)    := Fail for all X in VS,
verdict(g91,X)    := Fail for all X in VS,
verdict(g102,X)   := Fail for all X in VS,
verdict(g103,X)   := Fail for all X in VS,
verdict(g108,X)   := Fail for all X in VS,
verdict(g116,X)   := Fail for all X in VS.
```

The test coordinator assigns a Pass verdict to the implementation if Pass verdicts are assigned over both the connections.

To formulate liveness property of the test case, we first state the informal description of the purpose of each test component and then specify its formal representation. The three steps of the purpose of a test component are that a TC

i. establishes a connection,

ii. sends data including a command in TMPDU8 to send a reply in a TMPDU8r, and

iii. receives a reply in a TMPDU8r.

The formal specifications of the above three steps are stated below.

i. $AFTER(Tsend(L, NDTreq(CR))) \mapsto AFTER(Treceive(L, NDTind(CC)))$,

ii. $AFTER(Tsend(L, NDTreq(DT(TMPDU8))))$, and

iii. $AFTER(Treceive(L, NDTreq(DT(TMPDU8r))))$.

We compose the basic purposes of a test component using the SEQ operator as follows:

$$f_1 = AFTER(Tsend(L, NDTreq(CR))) \mapsto AFTER(Treceive(L, NDTind(CC)))\ SEQ\ AFTER(Tsend(L, NDTreq(DT(TMPDU8))))\ SEQ\ AFTER(Treceive(L, NDTreq(DT(TMPDU8r)))).$$

Then the liveness property of a test component is stated as $INIT \models (f_1 \mapsto (verdict = Pass))$.

All the execution paths in the global state space, shown in Fig. 7.8, do not satisfy the liveness property. Only the path indicated by bold arrows satisfies the property.

Therefore, the entire test purpose, which is the logical **AND** of the properties of the individual test components, is only partially satisfied.

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the results contributed by this research and presents some suggestions for further research extending the scope of the test verification technique.

8.1 Conclusions

This thesis represents the beginning of a new research topic in the area of *Protocol Engineering Life-cycle.* To develop operational confidence in the conformance of a protocol implementation with its formal specification, conventionally, the implementation is tested with a collection of test cases designed to check the behavior of the implementation in representative and exceptional instances of communication. Because of the complex nature of software systems, research over the last fifteen years on generating test cases for conventional sequential software in general and communication protocols in particular has not produced any breakthrough. Though limited success has been achieved in generating test cases from simplified protocol models such as deterministic finite state machines [SILE 89, DSU 90], automatic test generation becomes more and more elusive as more and more powerful standardized specification languages [Z100, IS7094, IS8807] are used in formalizing the behavior of complex communication systems.

Since the state-of-the-art in automated test suite design is still in a nascent state, human intuition and ingenuity play a decisive role in developing a complete test suite for a protocol specification. Expectedly, like any human designed systems, the development of a test suite for a large and complex protocol is error-prone. Philosophically, since the objective of a test case is to detect any implementation errors for the purpose of judging the implementation's conformance with its specification,

such a conformance judgement is meaningful only if the test case is correct with respect to the protocol specification.

With a simple protocol model, such as a deterministic FSM, a test case consists of a sequence of input/output events and is easy to comprehend in the sense that no parameters or predicates are attached to the test events. Therefore, verification of such a test sequence is a straightforward exercise. However, protocol specifications and test cases can be more complex than a simple deterministic FSM [NASA 92b]. The complexity of a protocol specification is due to higher expressive power of the specification language [Z100, IS7094, IS8807] including nondeterminism and the complexity of a test case is due to its expressiveness in detecting both desirable and unexpected behavior of an implementation, conditionally sending and receiving test events, in using timeouts and repeating some test behavior, in checking the values of various parameters received in an event, and in assigning a test verdict with respect to the *purpose* of the test case [ISO 9646]. Additional difficulties in test verification arise due to different languages used in specifying protocols and test cases, and architectural issues in a test system.

The objective of this research is to assist test designers, in the form of providing a methodology, to verify the correctness of test cases against the formal specifications of protocols. The verification methodology consists of the following steps:

i. Eliminate the syntactic and semantic differences among the languages used to specify protocols, test cases, and test management protocols by transforming them to a common EFSM notation.
ii. Derive a test verification system from a given test architecture.
iii. Generate the global behavior of the test verification system.
iv. Identify the test case properties as safety and liveness properties expressed as formulas in branching time temporal logic.
v. Verify the test case properties on the global behavior of the test verification system

by using a model checking approach.

In the following, we summarize the results of each chapter.

In chapter 1, the role of test case verification in protocol engineering life-cycle was identified and some preliminary test case verification techniques were summarized.

In chapter 2, a common EFSM notation for representing protocol specifications and test cases was defined, the protocol specification languages Estelle and the test specification notation TTCN were introduced, and algorithms for translating Estelle and TTCN specifications to EFSMs were presented. A notation, called I/O diagram, to represent the external ASP/PDU events exchanged among the entities in a test system was also defined. The need for a common notation to represent an external ASP/PDU event arises for accessing the parameter values in a received event while evaluating a boolean condition.

In chapter 3, we outlined the four basic test architectures LS, DS, RS, and CS and summarized their comparative error detection capabilities. We also defined the notion of a test verification system and derived test verification systems from different test architectures. Architectural concept played an important role in the verification process, because a test architecture defines both the physical interconnection among the components of a test system and the logical interconnection between two test entities in the form of a test coordination/management protocol. The global behavior of a test verification system was obtained by using a reachability analysis algorithm and a set of atomic predicates was associated with each global state. The motivation for associating a set of atomic predicates with each global state was that the predicates were used while verifying the safety and liveness properties of a test case.

In chapter 4, many new concepts developed as a result of this research were presented. For verification purpose, test case properties were expressed using the well known notions of safety and liveness. We identified four types of test case safety properties: transmission safety, reception safety, synchronization safety, and

test verdict safety. Identification of these properties was based on the safety notion that *nothing bad happens*. To express the liveness property, it was essential to formally express a test purpose, because fulfillment of the test purpose is *something good that must happen* in the testing process. We presented a set of notations to represent primitive test purposes as temporal formulas and a composition operator to define complex test purposes from primitive ones. To verify the test properties expressed as temporal formulas, we presented a model checking algorithm that runs in linear time with the size of the global state space.

In chapter 5, we demonstrated the verification of an RS architecture based test case designed to test an ACSE protocol implementation. An RS architecture has the distinction of defining a test case with only the Lower Tester and no Upper Tester. In this architecture, we had to derive the behavior of an Upper Tester using information from the Lower Tester and the protocol specification. Two design errors were found in the test case in the form of wrong use of *implicit send* events to define the behavior of the nonexistent Upper Tester.

In chapter 6, we verified a CS architecture based single connection test case designed to test a Class 2 transport protocol. The special feature of a CS architecture based test case is the use of test management protocol data units in the Lower Tester to control the behavior of the Upper Tester. We found out some transmission safety errors in the test case arising due to the nondeterministic send of acknowledgement PDUs and nondeterministic disconnection of the transport connection by the protocol specification entity. It was discovered that the test case was not designed to handle nondeterministic protocol behavior. We also detected the qualitative effect of timeouts on transmission safety.

In chapter 7, we verified a CS architecture based parallel test case with two simultaneous transport connections. We found out some transmission safety errors in the test case arising due to the nondeterministic send of acknowledgement PDUs

and nondeterministic disconnection of the transport connection by the protocol specification entity. It was discovered that the test case was not designed to handle nondeterministic protocol behavior.

Here we analyze the intertwining of safety and liveness properties of a test case. In the context of conventional program verification, in order to prove that "something good eventually happens", which refers to the liveness properties, one has to show that "nothing bad happens", which refers to the safety properties, along the execution of a system [OWLA 82]. That is, if something bad happens along the execution of a system, then something good cannot happen eventually. Moreover, in connection with program verification, a liveness property is evaluated with respect to program termination. Therefore, satisfaction of the safety properties is a prerequisite for the satisfaction of the liveness properties.

However, the context of protocol testing is slightly different from executing a conventional program. The purpose of a test case may be satisfied before the termination of the execution of a test system. That is, many test cases are designed in such a way that the test cases continue to interact with the IUT even after the purposes of the test cases are satisfied. Such interactions arise in practical testing, because the entire process of evaluating the conformance of an implementation is done by a set of test cases with a view that each test case tests one protocol function of an IUT at a time. Since a protocol function need not lead to the termination of the protocol, satisfaction of the purpose of a test case may need to be evaluated much before the protocol operation is terminated and a test case continues to interact with the IUT after the satisfaction of the test purpose.

For example, the purpose of the test case, referred to by the identifier ABCT2PRE00 in the CS architecture based test suite in [NCC 88], is to "test IUT's ability to connect and then disconnect." To achieve the test purpose, the test case contains the following behavior steps. The Test Management Protocol is initialized

in such a manner that it initiates a connection request by sending a TCONreq event to the IUT. If the test case receives a CR TPDU, then a CC TPDU is sent to the IUT followed by a DT TPDU. The DT TPDU contains a TMPDU directing the TMP to initiate a disconnection. If the connection is properly established, then the IUT must send an AK TPDU after receiving the CC followed by the DT TPDUs. After receiving the TMPDU, the TMP disconnects the transport connection by sending a TDISreq event to the IUT. If the test case receives a DR TPDU from the IUT, then it is confirmed that the IUT can initiate a disconnection procedure and hence the test purpose is satisfied. However, the test case does not terminate immediately after receiving a DR TPDU, but completes the disconnection procedure.

Therefore, it is essential that the test case does not contain any safety errors before the liveness property is satisfied, but it may contain safety errors after the liveness property is satisfied.

The safety and liveness errors reflect on the quality of the test case under verification. Ideally, a good test case should not contain any safety errors and the test purpose should be satisfied on all the execution paths in the model generated from a test verification system. Because of some inherent nondeterministic actions in a protocol specification, there may be a possibility that the test purpose is not satisfied on all execution paths. If a test case satisfies the liveness property on some execution paths in spite of containing a few safety errors, it is possible to improve the quality of the test case by eliminating the safety errors. However, if the liveness property is not satisfied along any execution path in the model, it is implied that the test case is highly erroneous and is useless from the point of the test purpose.

While verifying a parallel test case, we observed that the test purpose may be partially satisfied. In the presence of nondeterministic actions in the protocol specification, a partial satisfaction of test purpose does not indicate any design errors in the test case provided there are no safety errors in it.

The question of whether to continue with the verification process after detecting a safety error does not have a simple answer. On one hand, if the causes of safety errors are independent of one another, then the verification process can be run until all the present errors are detected and then corrections can be made to the test case. On the other hand, if a safety error occurs due to a previously occurring safety error, the second error vanishes once the first error is eliminated from a redesigned test case. In such a scenario, it is desirable to stop the verification process after the first error is detected and resume the verification process once the error is eliminated. However, determining the interdependence of errors is not an easy task. Therefore, the verification process may be run like compiling a program in the sense that if the number of errors detected exceeds some predetermined number then the verification process may be stopped.

8.2 Future Research

A natural phenomenon in research is that the solution of one problem leads to many new questions and to better solutions of some other problems. Since this work represents the beginning of research on test case verification, naturally, a large number of potential future tasks remain to be done to study the test verification issues with a broader perspective. In the following, we categorize the future tasks into four areas: implementation of the verification methodology, verifying invalid behavior test cases, piecewise test verification, and the use of model checking approach in generating test cases in a different and probably better way.

8.2.1 Implementation of the Verification System

None of the steps of the test verification methodology was implemented. Applications of the verification methodology to the three RS and CS architecture based test cases were developed manually. However, to be able to verify all the test cases in a test suite, it is essential to automate the verification methodology as follows.

- To generate EFSMs from protocol specifications in Estelle, LOTOS, and SDL and test case specifications in TTCN, it is essential to write an interpreter or a compiler for each of the languages. Experience in using a LOTOS interpreter [LOGR 88], which translates a LOTOS specification to horn clauses, shows that PROLOG is a good language for fast implementation of the EFSM construction algorithms described in chapter 2.
- In the implementation of the global state space generation algorithm in Chapter 3, a crucial step is to implement the predicate evaluation procedure *eval*.
- The model checking algorithm in Chapter 4 can be implemented in a manner similar to some other model checking implementations [FRV 86, CES 86].

8.2.2 Verifying Invalid Behavior Test Cases

Ideally, the objective of a complete test suite is not only to check that an implementation with valid inputs behaves correctly as stated in its specification, but also to check that the implementation does not behave abnormally under exceptional situations. The second type of behavior checking is known as *robustness testing*. An essential element in robustness testing is the application of invalid/inopportune test events to an implementation, that is, a test designer intentionally makes a test case output invalid/inopportune events. Therefore, if special care is not taken in the verification system, then correct test cases designed to test exceptional protocol behavior would be declared by the verification methodology as erroneous.

To verify test cases designed for robustness testing, it is required to modify the global state space generation algorithm such that generation of a state space continues after ignoring the invalid/inopportune events received by the protocol specification, because an ideal implementation, conceptually, can detect and ignore invalid/inopportune events and continue to behave as if no invalid/inopportune events were received. This is the basis of designing test cases for robustness testing in many test suites [NCC 88, PTT 90].

8.2.3 Piecewise Test Verification

Generation of EFSMs from protocol and test case specifications is the first step in the test verification methodology. Our observations reveal that test case EFSMs are small in size with states and transitions in the order of 100 and the EFSM representations of some of the protocol specifications could be very large with thousands of states and transitions. In reality, a protocol specification provides several communication functions with many mandatory and negotiable optional and alternative features in terms of protocol behavior, protocol parameters, and quality of service. Therefore, a test case, in general, is designed to check one protocol function implying that only a small part of the protocol specification gets activated in the reachability analysis process. Therefore, to verify a test case against a protocol specification, it is not required to generate a large EFSM, taking a large space and long time, representing the complete behavior of the protocol specification.

Accompanying every test suite are two documents: Protocol Implementation Conformance Statement (PICS) and Protocol Implementation eXtra Information for Testing (PIXIT). Information in the PICS and PIXIT can be used to divide a large test suite T_S into a collection of smaller test groups $\{T_{S1}, T_{S2}, ..., T_{Si}, ..., T_{Sn}\}$ and a large protocol specification P_S can be transformed into a collection of smaller specifications $\{P_{S1}, P_{S2}, ..., P_{Si}, ..., P_{Sn}\}$, such that test cases in group T_{Si} are verified against a smaller specification P_{Si}. For example, all the test cases in which the Upper Tester initiates the connection establishment procedure in the test suite [NCC 88] can be grouped as T_{Si} and a smaller specification P_{Si} can be derived from a transport protocol specification by eliminating its behavior responsible for handling connection requests from its peer entity.

Incorporating the PICS and PIXIT information with a large protocol specification to generate a collection of smaller specifications requires that the PICS and PIXIT are formally specified and the protocol is specified in a *structured* manner such that if

some behaviors are eliminated from a specification, the resulting smaller specification possesses a semantically consistent behavior.

8.2.4 Test Generation Using Model Checking Approach

The current test generation techniques, such as the *transition tour method*, the *distinguishing sequence method*, the *characterizing sequence method*, and the *unique input/output method*, derive test cases from a protocol specification by *syntactically* analyzing the protocol model [SILE 89, DSU 90, TRSA 91]. After deriving a test case using one of those methods, it is a hard task to know the *purpose* of the test case. Since a protocol specification is better understood as an entity providing a set of communication functions, if no *functional purpose* is associated with a test case, it becomes increasingly difficult for a test party to know how many protocol functionalities one test suite can test. Therefore, it is essential to define a functional purpose and then derive a test case fulfilling the same. The following steps outline a *purpose-directed* test generation technique using the model checking approach.

i. Identify a test purpose T_p and express it as a temporal formula f.

ii. Generate a state space model from a protocol specification and attach a set of atomic predicates AP_{s_i} with each state s_i such that all the predicates in the set AP_{s_i} hold in state s_i.

iii. Using the model checking approach, find out a set of paths T satisfying the temporal formula f. Now T denotes a test case that can check whether an implementation satisfies the test purpose T_p.

REFERENCES

[BEWI 90] S. R. Berkhout and M. F. Witteman, "Application of multi-party test methods to the GSM mobile network system", *Proc. of the 3rd. International Workshop on Protocol Test Systems*, McLean, Virginia, October 1990.

[BOCH 89] G. v. Bochmann, "Specification of a Simplified Transport Protocol Using Different Formal Description Techniques", *Computer Networks and ISDN Systems*, 18 (1989/90), pp. 335-377.

[BOCH 82] G. v. Bochmann, "Hardware Specification with Temporal Logic: An Example", *IEEE Trans. on Computers*, Vol. C-31, March 1982, pp. 223-231.

[BOCE 83] G. v. Bochmann, E. Cerny, M. Maksud, and B. Sarikaya, "Testing Transport Protocol Implementations", in: *Proc. CIPS*, Ottawa (1983) pp. 123-129.

[BPM 83] M. Ben-Ari, A. Pnueli, and Z. Manna, "The Temporal Logic of Branching Time", *Acta Informatica* 20, 1983, pp. 207-226.

[BRJO 78] D. Brand and W. H. Joyner, Jr., "Verification of Protocols Using Symbolic Execution", *Computer Networks* 2 (1978) pp. 351-360.

[BUDE 87] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems", *Computer Networks and ISDN Systems* 14 (1987), pp. 3–23.

[CES 86] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM TOPLAS*, Vol. 8, No. 2, April 1986, pp. 244-263.

[DSU 90] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar, "Formal Methods for Generating Protocol Conformance Test Sequences", *Proceedings of the IEEE*, Vol. 78, No. 8, August 1990, pp. 1317-1326.

[DUBO 90] M. Dubuc and G. v. Bochmann, *et. al.*, "Translation from TTCN to LOTOS and the Validation of Test Cases", *Proc. of the Formal Description Technique '90 Conference (FORTE'90)*, Madrid, Nov. 1990.

[EHMA 85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification*, Springer-Verlag, Berlin, 1985.

[FRV 86] J. C. Fernandez, J. L. Richier, and J. Voiron, "Verification of Protocol Specifications Using the CESAR System", *Proc. of the 5th. International Symposium on Protocol Specification, Testing, and Verification*, 1985.

[GOHA 85] M. G. Gouda and J. -Y. Han, "Protocol Validation by Fair Progress State Exploration", *Computer Networks and ISDN Systems* 9 (1985), pp. 353–361.

[GOYU 84] M. G. Gouda and Y. T. Yu, " Protocol Validation by Maximal Progress State Exploration", *IEEE Trans. on Comm.* Vol. 32, No. 1, January 1984, pp. 94–97.

[HAIL 82] B. T. Hailpern, "Verifying Concurrent Processes Using Temporal Logic", *LNCS 129*, Springer-Verlag, New York, 1982.

[HAOW 83] B. T. Hailpern and S. S. Owicki, "Modular Verification of Computer Communication Protocols", *IEEE Trans. on Comm.*, Vol. COM-31, No. 1, Jan. 1983, pp. 56-68.

[HOAR 78] C. A. R. Hoare, "Communicating Sequential Processes", *Comm. of the ACM*, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

[HOLZ 87] G. J. Holzmann, "Automated Protocol Validation in Argos: Assertion Proving and Scatter Searching", *IEEE Trans. on Software Eng.*, Vol. SE-13, No. 6, June 1987, pp. 683-696.

[ISO 7498] Information Processing Systems - Open Systems Interconnection - Basic Reference Model, ISO 7498, 1984.

[ISO 8327] ISO 8327, Information Processing System - Open Systems Interconnection - Basic connection-oriented session protocol specification.

[ISO 8650] Protocol Specification for the Association Control Service Elements, ISO DIS8650, January 1988.

[IS8807] ISO/IEC IS8807: LOTOS, a Formal Description Technique based on the

Temporal Ordering of Observable Behavior, ISO/TC97/SC21/WG1-FDT/SC-C, June 1988.

[IS8824] ISO 8824: Profile of Abstract Syntax Notation One, IS 8824, 1987.

[IS9074] ISO/IEC IS9074: Estelle - A Formal Description Technique Based on an Extended State Transition Model, ISO/TC97/SC21/WG1, 1987.

[ISO 9646] ISO/IEC 9646: Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework, 1991.

[ISO9646–3E] TTCN Extensions: Working Draft Amendments to ISO/IEC 9646-3, Phoenix Output.

[JACK 83] M. Jackson, *System Development*, Prentice Hall, 1983.

[KARJ 88] G. Karjoth, "Implementing Process Algebra Specifications by State Machines", *Proc. of the 8th. International Symposium on Protocol Specification, Testing, and Verification,* Atlantic City, June 1988.

[LAMP 80] L. Lamport, "Sometime" is sometimes "not never", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, NE, 1980, pp. 174-183.

[LAMP 83] L. Lamport, "Specifying Concurrent Program Modules", *ACM TOPLAS,* Vol. 5, No. 2, April 1983, pp. 190-222.

[LOGR 88] L. Logrippo, A. Obaid, J. P. Braind, and M. C. Fehri, "An Interpreter for LOTOS, A Specification Language for Distributed Systems", *Software Practice and Experience*, Vol. 18, April 1988, pp. 365-385.

[LU 86] C. S. Lu, *Automated Validation of Communication Protocols*, Ph.D. thesis, The Ohio State University, 1986.

[MILN 80] R. Milner, "A Calculus of Communicating Systems", *Lecture Notes in Computer Science, Vol. 92*, Springer-Verlag, 1980.

[NASA 90a] K. Naik and B. Sarikaya, "An Extended Finite State Machine Model for TTCN", *Proc. of the 15th. Biennial Symposium on Communications,* Kingston, Ontario, June, 1990.

[NASA 90b] K. Naik and B. Sarikaya, "Static Validation of TTCN Test Cases", *Proc. of the 3rd. International Workshop on Protocol Test Systems*, McLean, Virginia, October 1990.

[NASA 92a] K. Naik and B. Sarikaya, "Testing Communication Protocols", *IEEE Software*, January 1992, pp. 27-37.

[NASA 92b] K. Naik and B. Sarikaya, "Verification of Protocol Conformance Test Cases Using Reachability Analysis", *The Journal of Systems and Software*, July 1992.

[NCC 88] Abstract Test Suite for Transport Protocol Class 2, The National Computing Center Limited, UK, 1988.

[OWLA 82] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs", *ACM TOPLAS*, Vol. 4, No. 3, July 1982, pp. 455-495.

[PTT 90] Abstract Test Suite for ACSE Protocol Version 2.0, Dutch PTT Research, 1990.

[SAB 88] K. Sabnani, "An Algorithmic Technique for Protocol Verification", *IEEE Transaction on Comm.* Vol. COM-36, No. 8, August. 1988, pp. 924-931.

[SABO 84] B. Sarikaya and G. v. Bochmann, "Synchronization and Specification Issues in Protocol Testing", *IEEE Trans. on Comm.*, Vol. COM-32, No. 4, April 1984, pp. 389-395.

[SABO 86] B. Sarikaya and G. v. Bochmann, "Obtaining Normal Form Specifications for Protocols", COMPUTER NETWORK USAGE: Recent Experiences, Csaba, Tarnay, and Szentivayni edited, North-Holland, 1986.

[SARI 89] B. Sarikaya, "Conformance Testing: Architectures and Test Sequences", *Computer Networks and ISDN Systems* 17 (1989) pp. 111-126.

[SAWI 92] B. Sarikaya and A. Wiles, "Standard Conformance Test Specification Language TTCN", *Computer Standards and Interfaces*, April 1992.

[SILE 89] D. P. Sidhu and T-K. Leung, "Formal Methods for Protocol Testing: A

Detailed Study", *IEEE Trans. on Software Engineering*, Vol. 15, No. 4, April 1989, pp. 413-426.

[TRSA 90] P. Tripathy and B. Sarikaya, "Test Case Generation from LOTOS Specification", *IEEE Trans. on Computers*, Vol. 40, April 1991, pp. 543-552.

[UYDA 86] M. U. Uyar and A. T. Dahbura, "Optimal Test Sequence Generation for protocol: the Chinese Postman Algorithm Applied to Q.931", *Proc. of the IEEE Global Telecommunication Conference*, 1986.

[VUHU 87] S. T. Vuong, D. D. Hui, and D. D. Cowan, "VALIRA - A Tool for Protocol Validation Via Reachability Analysis", *Proc. of the 6th. Protocol Specification, Testing, and Verification Workshop*, North-Holland, 1987, pp. 35-41.

[WEST 78] C. H. West, "General Techniques for Communication Protocol Validation", *IBM Journal of Res. and Development*, Vol. 22, No. 4, July 1978, pp. 393-404.

[WEST 87] C. H. West, "Protocol Validation by Random State Exploration", *Proc. of the 6th. Protocol Specification, Testing, and Verification Workshop*, North-Holland, 1987, pp. 233-242.

[WEZA 78] C. H. West and P. Zafiropulo, "Automated Validation of a Communication Protocol: the CCITT X.21 Recommendation", *IBM Journal of Research and Development*, 22, 60 (1978).

[Z100] CCITT, Specification and Description Language SDL, Recommendation Z.100, 1988.

[ZAFI 78] P. Zafiropulo, "Protocol Validation by Duologue-Matrix Analysis", *IEEE Trans. on Comm.*, Vol. COM-26, No. 8, August 1978, pp. 1187-1194.

[ZAFI 80] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand, "Towards Analyzing and Synthesizing Protocols", *IEEE Trans. on Comm.*, Vol. COM-28, No. 4, April 1980, pp. 651-661.

[ZAFI 83] P. Zafiropulo, *et. al.*, "Protocol Analysis and Synthesis Using a State Tran-

sition Model", in *Computer Networks and Protocols,* P. E. Green, Ed. New York: Plenum, 1983, pp. 645-670.

[ZHBO 87] J. -R. Zhao and G. v. Bochmann, "Reduced Reachability Analysis of Communication Protocols: A New Approach", *Proc. of the 6th. Protocol Specification, Testing, and Verification Workshop,* North-Holland, 1987, pp. 243–254.

Appendix 1

Global state space of the RS Test Verification System in Chapter 5.

```
-----------------------------------------------------
VAR1  = {STATE="",TEST_BODY="",MODE_Supported=1,
         Verdict=Null}
VAR2  = {STATE=NONE,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Null}
VAR10 = {STATE=NONE,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Null}
VAR13 = {STATE=CONNECT,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Null}
VAR27 = {STATE=NONE,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Null}
VAR28 = {STATE=NONE,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Pass}
VAR34 = {STATE=NONE,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Inconclusive}
VAR39 = {STATE=CONNECT,TEST_BODY=TRUE,MODE_Supported=1,
         Verdict=Inconclusive}
-----------------------------------------------------
g1 = <1, 1, A, "", {E,E,E,E,E,E}, VAR1>{INIT}
g2 = <2, 1, A, "", {E,E,E,E,E,E}, VAR2>
       {AT(Tsend(U,A_ASCreq))}
g3 = <3, 1, A, "", {E,E,E,E,E,A_ASCreq()}, VAR2>
       {AFTER(Tsend(U,A_ASCreq))}
g4 = <4, 1, A, "", {E,E,E,E,E,A_ASCreq()}, VAR2>
       {AT(Sreceive(U,A_ASCreq))}
g5 = <4, 1, 1, "", {E,E,E,E,E,E}, VAR2>
       {AFTER(Sreceive(U,A_ASCreq)),AT(Ssend(P,P_CONreq
g6 = <4, 1, B, "", {E,E,E,P_CONreq(),E,E}, VAR2>
       {AFTER(Ssend(P,P_CONreq))}
g7 = <4, 2, B, "", {E,E,E,E,E,E}, VAR2>{}
g8 = <4, 1, B, "", {E,P_CONind(),E,E,E,E}, VAR2>
       {AT(Treceive(P,P_CONind))}
g9 = <5, 1, B, "", {E,E,E,E,E,E}, VAR2>
       {AFTER(Treceive(P,P_CONind))}
g10= <6, 1, B, "", {E,E,E,E,E,E}, VAR10>{}
g11= <7, 1, B, "", {E,E,E,E,E,E}, VAR10>
       {AT(Tsend(L,P_CONrsp))}
g12= <8, 1, B, "", {P_CONrsp(),E,E,E,E,E}, VAR10>
       {AFTER(Tsend(L,P_CONrsp))}
g13= <9, 1, B, "", {P_CONrsp(),E,E,E,E,E}, VAR13>
       {AT(Tsend(U,A_RELreq))}
g14= <10,1, B, "", {P_CONrsp(),E,E,E,E,E}, VAR13>
```

```
        {AFTER(Tsend(U,A_RELreq))}
g15= <11,1,B,"",{P_CONrsp(),E,E,E,E,A_RELreq()},VAR13>{}
g16= <11,5,B,"",{E,E,E,E,E,A_RELreq()},VAR13>{}
g17= <11,1,B,"",{E,E,E,P_CONcnf(),E,A_RELreq()},VAR13>
        {AT(Sreceive(P,P_CONcnf))}
g18= <11,1,10,"",{E,E,E,E,E,A_RELreq()},VAR13>
        {AFTER(Sreceive(P,P_CONcnf)),
         AT(Ssend(U,A_ASScnf))}
g19= <11,1,D,"",{E,E,E,E,A_ASCcnf(),A_RELreq()},VAR13>
        {AFTER(Ssend(U,A_ASScnf)),
         AT(Sreceive(U,A_RELreq))}
g20= <11,1,25,"",{E,E,E,E,A_ASCcnf(),E},VAR13>
        {AFTER(Sreceive(U,A_RELreq)),
         AT(Ssend(P,P_RELreq))}
g21= <11,1,E,"",{E,E,E,P_RELreq(),A_ASCcnf(),E},VAR13>
        {AFTER(Ssend(P,P_RELreq))}
g22= <11,10,E,"",{E,E,E,E,A_ASCcnf(),E},VAR13>{}
g23= <11,1,E,"",{E,P_RELind(),E,E,A_ASCcnf(),E},VAR13>
        {AT(Treceive(L,P_RELind))}
g24= <12,1,E,"",{E,E,E,E,A_ASCcnf(),E},VAR13>
        {AFTER(Treceive(L,P_RELind))}
g25= <13,1,E,"",{E,E,E,E,A_ASCcnf(),E},VAR13>
        {AT(Tsend(L,P_RELrsp))}
g26= <14,1,E,"",{P_RELrsp(),E,E,E,A_ASCcnf(),E},VAR13>
        {AFTER(Tsend(L,P_RELrsp))}
g27= <15,1,E,"",{P_RELrsp(),E,E,E,A_ASCcnf(),E},VAR27>{}
g28= <16,1,E,"",{P_RELrsp(),E,E,E,A_ASCcnf(),E},VAR28>{}
g29= <16,7,E,"",{E,E,E,E,A_ASCcnf(),E},VAR28>{}
g30= <16,1,E,"",{E,E,P_RELcnf(),E,A_ASCcnf(),E},VAR28>
        {AT(Sreceive(P,P_RELcnf))}
g31= <16,1,28,"",{E,E,E,E,A_ASCcnf(),E},VAR28>
        {AFTER(Sreceive(P,P_RELcnf)),
         AT(Ssend(U,A_RELcnf))}
g32= <16,1,A,"",{E,E,E,E,{A_ASCcnf(),A_RELcnf()},E},
       VAR28> {AFTER(Ssend(U,A_RELcnf))}
/* g33 is obtained from g6. */
g33= <21,1,B,"",{E,E,E,P_CONreq(),E,E},VAR2>{}
g34= <24,1,B,"",{E,E,E,P_CONreq(),E,E},VAR34>{}
g35= <24,2,B,"",{E,E,E,E,E,E},VAR34>{}
g36= <24,1,B,"",{E,P_CONind(),E,E,E,E},VAR34>{}
/* g37 is obtained from g21. */
g37= <25,1,E,"",{E,E,E,P_RELreq(),A_ASCcnf(),E},VAR13>{}
g38= <26,1,E,"",{E,E,E,P_RELreq(),A_ASCcnf(),E},VAR13>
        {AT(Tsend(L,P_UABreq))}
g39= <27,1,E,"",{P_UABreq(),E,E,P_RELreq(),
```

```
      A_ASCcnf(),E},VAR39>
      {AFTER(Tsend(L,P_UABreq))}
g40= <27,10,E,"",{P_UABreq(),E,E,E,A_ASCcnf(),E},VAR39>{}
g41= <27,1,E,"",{P_UABreq(),P_RELind(),E,E,
      A_ASCcnf(),E},VAR39>{}
g42= <27,11,E,"",{E,P_RELind,E,E,A_ASCcnf(),E},VAR39>{}
g43= <27,1,E,"",{E,P_RELind(),P_UABind(),E,A_ASCcnf(),E},
      VAR39>{AT(Sreceive(P,P_UABind))}
g44= <27,1,30,"",{E,P_RELind(),E,E,A_ASCcnf(),E},VAR39>
       {AFTER(Sreceive(P,P_UABind)),
        AT(Tsend(U,A_ABind))}
g45= <27,1,A,"",{E,P_RELind(),
      E,E,{A_ASCcnf(),A_ABind()},E},VAR39>
      {AT(Tsend(U,A_ABind))}
-----------------------------------------------

-----------------------------------------------
Global Transitions
-----------------------------------------------
r1  = <g1,  g2,  LT_1>,     r2  = <g2,  g3,  LT_2>,
r3  = <g3,  g4,  LT_3>,     r4  = <g4,  g5,  SPEC_1>,
r5  = <g5,  g6,  SPEC_2>,   r6  = <g6,  g7,  USP_1>,
r7  = <g7,  g8,  USP_2>,    r8  = <g8,  g9,  LT_4>,
r9  = <g9,  g10,  LT_5>     r10 = <g10, g11, LT_6>,
r11 = <g11, g12, LT_7>,     r12 = <g12, g13,  LT_8>,
r13 = <g13, g14, LT_9>,     r14 = <g14, g15, LT_10>,
r15 = <g15, g16,  USP_7>,   r16 = <g16, g17, USP_8>,
r17 = <g17, g18, SPEC_3>,   r18 = <g18, g19,  SPEC_4>,
r19 = <g19, g20, SPEC_5>,   r20 = <g20, g21, SPEC_6>,
r21 = <g21, g22,  USP_13>,  r22 = <g22, g23, USP_14>,
r23 = <g23, g24, LT_11>,    r24 = <g24, g25,  LT_12>,
r25 = <g25, g26, LT_13>,    r26 = <g26, g27, LT_14>,
r27 = <g27, g28,  LT_15>,   r28 = <g28, g29, USP_19>,
r29 = <g29, g30, USP_20>,   r30 = <g30, g31,  SPEC_7>,
r31 = <g31, g32, SPEC_8>,   r32 = <g6,  g33, LT_16>,
r33 = <g33, g34,  LT_18>,   r34 = <g34, g35, USP_1>,
r35 = <g35, g36, USP_2>,    r36 = <g21, g37,  LT_20>,
r37 = <g37, g38, LT_21>,    r38 = <g38, g39, LT_23>,
r39 = <g39, g40,  USP_13>,  r40 = <g40, g41, USP_14>,
r41 = <g41, g42, USP_11>,   r42 = <g42, g43,  USP_12>,
r43 = <g43, g44, SPEC_9>,   r44 = <g44, g45, SPEC_10>.
-----------------------------------------------
```

Appendix 2

EFSM model of a Test Management Protocol Used in Chapter 6.

```
Transitions of a Test Management Protocol for testing a
transport protocol implementation in the CS architecture.
STATES:
IDLE1,IDLE_M3,IDLE_M9,IDLE_M10,IDLE_M12,IDLE_SAS,IDLE,
IUT_WFTRESP_M1,IUT_WFTRESP,IUT_WFTRESP_SAS,
IUT_WFTRESP_M1,IUT_WFTRESP_M12,IUT_WFTRESP_M10,OPEN,
OPEN_1,OPEN_2,OPEN_3,OPEN_8,OPEN_16.OPEN_M2,OPEN_M4,
OPEN_M5,OPEN_M6,OPEN_M7,OPEN_M9,OPEN_M12,OPEN_SAS,
OPEN_SAS_M6,OPEN_SAS_M7,RST STOP,UT_WFTCONF,
UT_WFTCONF_M10,UT_WFTCONF_M12,UT_WFTCONF_SAS.

Data Type Declaration:
The TMP contains THREE types of variables:
                   Counters, Modes, and Stored items.
The Counters are numbered C1 thru C38,
the Modes are numbered M1 thru M25,
and the Stored items are numbered S1 thru S28.
Integer C1;
Integer C2;
:
Integer C38;
Mode_type M1;
:
Mode_type M25;
OctetString S1;
:
OctetString S28;
NOTATION: "inc" function increments its parameters.
For example, inc(C1, C2) increment counts C1 and C2 by 1.
----------------------------------------------------
                T R A N S I T I O N S
----------------------------------------------------
This is the INITIALIZING transition.
<IDLE, IDLE_M9, Internal_START, [T],
 {C1:=0, C2:= 0,.., C38:=0,
  M1:=A4, M2:=A0,.., M10:=A0, M11:=A5, M12:=A5,
  M13:=A0,...,M25:=A0,
  S1:= .., S2:=..,..,S5:="1",S6:="you",
  S7:="No",S8:="any_data",..,S28}>
```

```
<IDLE,IUT_WFTRESP_M1,U?TCONind,[T],
 {S16:=TCONind.Called_address,
  S17:=TCONind.Calling_address,
  S18:=TCONind.Exp_data_option,S19:=TCONind.Qos,
  S20:=TCONind.TSuser_data,inc(C1, C19)},1>
<UT_WFTCONF,UT_WFTCONF_M12,U?TCONind,[T],
 {S16:=TCONind.Called_address,
  S17:=TCONind.Calling_address,
  S18:=TCONind.Expdata_option, S19:=TCONind.Qos,
  S20:=TCONind.TSuser_data,inc(C8, C20)},1>
<IUT_WFTRESP,IUT_WFTRESP_M12,U?TCONind,[T],
 {S16:=TCONind.Called_address,
  S17:=TCONind.Calling_address,
  S18:=TCONind.Exp_data_option,S19:=TCONind.Qos,
  S20:=TCONind.TSuser_data,inc(C8,C20)},1>
<OPEN, OPEN_M12, U?TCONind, [T],
 {S16:=TCONind.Called_address,
  S17:=TCONind.Calling_address,
  S18:=TCONind.Exp_data_option,S19:=TCONind.Qos,
  S20:=TCONind.TSuser_data,inc(C8, C20)},1>
<IDLE, IDLE_M12,U?TCONconf,[T],
 {S21:=TCONconf.Qos,S22:=TCONconf.Responding_address,
  S23:=TCONconf.Exp_data_option,
  S24:=TCONconf.TSuser_data.inc(C9, C20)},1>
<UT_WFTCONF, OPEN_M2,U?TCONconf,[T],
 {S21:=TCONconf.Qos,S22:=TCONconf.Responding_address,
  S23:=TCONconf.Exp_data_option,
  S24:=TCONconf.TSuser_data,inc(C2, C19)},1>
<IUT_WFTRESP,IUT_WFTRESP_M12,U?TCONconf,[T],
 {S21:=TCONconf.Qos,S22:=TCONconf.Responding_address,
  S23:=TCONconf.Exp_data_option,
  S24:=TCONconf.TSuser_data,inc(C9, C20)},1>
<OPEN,OPEN_M12,U?TCONconf,[T],
 {S21:=TCONconf.Qos,S22:=TCONconf.Responding_address,
  S23:=TCONconf.Exp_data_option,
  S24:=TCONconf.TSuser_data,inc(C9, C20)},1>
<IDLE,IDLE_M12,U?TDISind,[T],
 {S14:=TDISind.Reason,S15:=TDISind.TSuser_data,
  inc(C10,C20)},1>
<UT_WFTCONF,IDLE_M3,U?TDISind,[T],
 {S14:=TDISind.Reason,S15:=TDISind.TSuser_data,
  inc(C3,C19)},1>
```

```
<IUT_WFTRESP,IDLE_M3,U?TDISind,[T],

 {S14:=TDISind.Reason,S15:=TDISind.TSuser_data,
  inc(C3,C19)},1>
<OPEN,IDLE_M3,U?TDISind,[T],
 {S14:=TDISind.Reason,S15:=TDISind.TSuser_data,
  inc(C3,C10)},1>
<IDLE,IDLE,U?TDTind,[not(EDTSDU) and P_SPF],
 {per octet(S25:=TDTind.data,inc(C15)},1>
<IDLE,IDLE,U?TDTind,[not(EDTSDU) and not(P_SPF)],
 {per octet(S25:=TDTind.data, inc(C11)},1>
<IDLE,IDLE,U?TDTind,[EDTSDU and P_SPF],
 {inc(C16,C21)},1>
<IDLE,IDLE_M12,U?TDTind,[EDTSDU and not(P_SPF)],
 {inc(C12,C20)},1>
<UT_WFTCONF, UT_WFTCONF, U?TDTind,
 [not(EDTSDU) and P_SPF],
 {per octet(S25:=TDTind.data, inc(C15)},1>
<UT_WFTCONF, UT_WFTCONF, U?TDTind,
 [not(EDTSDU) and not(P_SPF)],
 {per octet(S25:=TDTind.data, inc(C11)},1>
<UT_WFTCONF,UT_WFTCONF,U?TDTind,
 [EDTSDU and P_SPF],
 {inc(C16,C21)},1>
<UT_WFTCONF,UT_WFTCONF_M12,U?TDTind,
 [EDTSDU and not(P_SPF)],
 {inc(C12,C20)},1>
<IUT_WFTRESP,IUT_WFTRESP,U?TDTind,[not(EDTSDU)],
 {per octet(S25:=TDTind.data,inc(C11)},1>
<IUT_WFTRESP,IUT_WFTRESP_M12,U?TDTind,
 [EDTSDU],{inc(C12,C20)},1>
<OPEN,OPEN_M4,U?TDTind,[not(EDTSDU) and not(P_TMPDU)],
 {per octet(S25:=TDTind.data, inc(C4)},1>
<OPEN,OPEN_M5,U?TDTind,[EDTSDU and not(P_TMPDU)],
 {inc(C5,C19)},1>
/* When the TMP receives a TDTind containing a TMPDU */
<OPEN,OPEN_1,U?TDTind,[P_TMPDU],
 {inc(C4) per octet,inc(C5,C19)}>
<OPEN_1,OPEN_M9,Null,[TMPDU1 in TDTind],
 {M1:=TDTind.TMPDU1.M1,
 M2:=TDTind.TMPDU1.M2,..,M25:=TDTind.TMPDU1.M25},1>
<OPEN_1,OPEN,Null,[TMPDU2 in TDTind],
```

```
 {C1:=0, C2:= 0,.., C38:=0},2>
<OPEN_1,OPEN,Null,[TMPDU3 in TDTind],
 {S1:=TDTind.TMPDU3.S1},3>
<OPEN_1,OPEN,Null,[TMPDU4 in TDTind],
 {S5:=TDTind.TMPDU4.S5,
  S6:=TDTind.TMPDU4.S6,S7:=TDTind.TMPDU4.S7,
  S8:=TDTind.TMPDU4.S8},4>
<OPEN_1,OPEN,Null,[TMPDU5 in TDTind],
 {S9:=TDTind.TMPDU5.S9,
  S10:=TDTind.TMPDU5.S10, S11:=TDTind.TMPDU5.S11,
  S12:=TDTind.TMPDU5.S12,S13:=TDTind.TMPDU5.S13},5>
/* If a TMPDU6 is received, START auto. source
   in OPEN state.*/
<OPEN_1,OPEN_SAS,Null,
 [(TMPDU6 in TDTind) and (M15 <> 0)],{},6>
/*If a TMPDU7 is received, then generate a new
  UT_TMP_ENTRY.Here we do not generate a new
  UT_TMP_ENTRY.*/
<OPEN_1,OPEN,Null,[TMPDU7 in TDTind], {},7>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),
 [TMPDU8 in TDTind],
 {TSuser_data:=HERALD||"8"||"25"||M1||..||
                 M25||TRAILER},8>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),
 [TMPDU9 in TDTind],
 {TSuser_data:=HERALD||"9"||"38"||C1||..||
                 C38||TRAILER},9>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),
 [TMPDU10 in TDTind],{TSuser_data:=HERALD||"10"||
                       "2"||S14||S15||TRAILER},10>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),[TMPDU11 in TDTind],
 {TSuser_data := HERALD||"11"||"5"||S16||S17||
                   S18||S19||S20||TRAILER},11>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),[TMPDU12 in TDTind],
 {TSuser_data:=HERALD||"12"||"4"||S21||S22||
                 S23||S24||TRAILER},12>
<OPEN_1,OPEN,Null,[TMPDU13 in TDTind],{},13>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),[TMPDU14 in TDTind],
 {TSuser_data:=HERALD||"14"||"1"||S25||TRAILER},14>
<OPEN_1,OPEN,U!TDTreq(TSuser_data),[TMPDU15 in TDTind],
 {TSuser_data:=HERALD||"15"||"1"||S26||TRAILER},15>
/*An invalid TMPDU fires following transition. */
```

```
<OPEN_1,OPEN_16,Null,[T],{},16>
/*If you receive a TDTind while doing
  AUTOMATIC source*/
<OPEN_SAS, OPEN_2, U?TDTind, [P_TMPDU], {},1>
<OPEN_2,OPEN_M9, Null,[TMPDU1 in TDTind],

 {M1:=TDTind.TMPDU1.M1,
  M2:=TDTind.TMPDU1.M2,...,M25:=TDTind.TMPDU1.M25},1>
<OPEN_2,OPEN_SAS,Null,[TMPDU2 in TDTind],
 {C1:=0, C2:=0,...,C38:=0},2>
<OPEN_2,OPEN_SAS,Null,[TMPDU3 in TDTind],
 {S1:=TDTind.TMPDU3.S1},3>
<OPEN_2,OPEN_SAS,Null,[TMPDU4 in TDTind],
 {S5:=TDTind.TMPDU4.S5,
  S6:=TDTind.TMPDU4.S6, S7:=TDTind.TMPDU4.S7,
  S8:=TDTind.TMPDU4.S8},4>
<OPEN_2,OPEN_SAS,Null,[TMPDU5 in TDTind],
 {S9:=TDTind.TMPDU5.S9,
  S10:=TDTind.TMPDU5.S10,S11:=TDTind.TMPDU5.S11,
  S12:=TDTind.TMPDU5.S12, S13:=TDTind.TMPDU5.S13},5>
/* If a TMPDU6 is received,START automatic source
   in OPEN state. */
<OPEN_2,OPEN_SAS,Null,[(TMPDU6 in TDTind)],{},6>
/*If a TMPDU7 is received, then generate a new
  UT_TMP_ENTRY.
  Here we don't generate a new UT_TMP_ENTRY.*/
<OPEN_2,OPEN_SAS,Null,[TMPDU7 in TDTind],{},7>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU8 in TDTind],{TSuser_data:=HERALD||"8"||"25"||
                    M1||...||M25||TRAILER},8>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU9 in TDTind],{TSuser_data:=HERALD||"9"||
                    "38"||C1||..||C38||TRAILER},9>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU10 in TDTind],{TSuser_data:=HERALD||"10"||
                     "2"||S14||S15||TRAILER},10>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU11 in TDTind],{TSuser_data:=HERALD||"11"||
          "5"||S16||S17||S18||S19||S20||TRAILER},11>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU12 in TDTind],{TSuser_data:=HERALD||"12"||
  "4"||S21||S22||S23||S24||TRAILER},12>
```

```
<OPEN_2,OPEN_SAS,Null,[TMPDU13 in TDTind],{},13>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU14 in TDTind],{TSuser_data:=HERALD||"14"||
                          "1"||S25||TRAILER},14>
<OPEN_2,OPEN_SAS,U!TDTreq(TSuser_data),
 [TMPDU15 in TDTind],
 {TSuser_data:=HERALD||"15"||"1"||S26||TRAILER},15>
<OPEN_2,OPEN_S16,Null,[T],{},16>
<OPEN_SAS,OPEN_M4,U?TDTind,
 [not(EDTSDU) and not(P_TMPDU)],
 {per octet(S25:=TDTind.data, inc(C4)},1>
<OPEN_SAS,OPEN_M5,U?TDTind,[EDTSDU and not(P_TMPDU)],
 {inc(C5,C19)},1>
<OPEN_SAS OPEN_SAS_M6,U?TEXind,[not(EDTSDU)],
 {per octe' (S25:=TDTind.data, inc(C6)},1>
<OPEN_SAS,OPEN_SAS_M7,U?TEXind,[EDTSDU],{inc(C7,C10)},1>
/* If a CLE occurs while doing an AUTOMATIC source,
   fire the following transition. */
<OPEN_SAS, OPEN_8, ?CLE, [T], {CLE_count := 0}, 2>
/* Start AUTOMATIC source. */
<OPEN_SAS,OPEN_3,Null, [T],
 {TSuser_data := make_data(M16,M17,M18,M19,M20,M21)},3>
<OPEN_3,OPEN_SAS,U!TDTreq(TSuser_data),[P_AUTODT],
 {inc(C26) per octet,inc(C27,C37),
  S26:=last_octet_from_auto_source},1>
<OPEN_3,OPEN_SAS,U!TEXreq(TSuser_data),[not P_AUTODT],
 {inc(C28) per octet,inc(C29,C37),
  S26:=last_octet_from_auto_source},1>


/* TEXind */
<IDLE,IDLE,U?TEXind,[not(EDTSDU) and P_SPF],
 {per octet(S25:=TEXind.data, inc(C17)},1>
<IDLE,IDLE,U?TEXind,[not(EDTSDU) and not(P_SPF)],
 {per octet(S25:=TDTind.data,inc(C13)},1>
<IDLE,IDLE,U?TEXind,[EDTSDU and P_SPF],{inc(C18,C21)},1>
<IDLE,IDLE_M12,U?TEXind,[EDTSDU and not(P_SPF)],
 {inc(C14,C20)},1>
<UT_WFTCONF,UT_WFTCONF,U?TEXind,[not(EDTSDU) and P_SPF],
 {per octet(S25:=TDTind.data,inc(C17)},1>
<UT_WFTCONF,UT_WFTCONF, U?TEXind,
 [not(EDTSDU) and not(P_SPF)],
 {per octet(S25:=TDTind.data, inc(C13)},1>
```

```
<UT_WFTCONF,UT_WFTCONF,U?TEXind,[EDTSDU and P_SPF],
 {inc(C18,C21)},1>
<UT_WFTCONF,UT_WFTCONF_M12,U?TEXind,
 [EDTSDU and not(P_SPF)],{inc(C14,C20)},1>
<IUT_WFTRESP,IUT_WFTRESP,U?TEXind,[not(EDTSDU)],
 {per octet(S25:=TDTind.data, inc(C13)},1>
<IUT_WFTRESP,IUT_WFTRESP_M12,U?TEXind,[EDTSDU],
 {inc(C14,C20)},1>
<OPEN,OPEN_M6,U?TEXind,[not(EDTSDU)],
 {per octet(S25:=TDTind.data,inc(C6)},1>
<OPEN,OPEN_M7,U?TEXind,[EDTSDU],{inc(C7,C10)},1>
/* Internal Count Limit Event (CLE) */
<IDLE,IDLE_M10,?CLE,[T],{CLE_count:=0},1>
<UT_WFTCONF,UT_WFTCONF_M10,?CLE,[T],{CLE_count:=0},1>
<IUT_WFTRESP,IUT_WFTRESP_M10,?CLE,[T],{CLE_count:=0},1>
/* Automatic Source (TSDU Release) */
<IDLE_SAS,IDLE,Null,[P_AUTODT],
 {inc(C33) per octet, inc(C34,C38)},1>
<IDLE_SAS,IDLE,Null,[not(P_AUTODT)],
 {inc(C35) per octet,inc(C36,C38)},1>
<UT_WFTCONF_SAS,IDLE1,Null,[P_AUTODT],
 {inc(C33) per octet,inc(C34,C38)},1>
<IDLE1,IDLE,U!TDISreq(S4),[T],{},1>
<UT_WFTCONF_SAS,IDLE1,Null,[not(P_AUTODT)],
 {inc(C35) per octet,inc(C36,C38)},1>
<IUT_WFTRESP_SAS,IDLE1,Null,[P_AUTODT],
{inc(C33) per octet,inc(C34,C38)},1>
<IUT_WFTRESP_SAS,IDLE1,Null,[not(P_AUTODT)],
 {inc(C35) per octet,inc(C36,C38)},1>
/* If a TCONind is received in IDLE state,
   then do exec(M1). */
<IUT_WFTRESP_M1,IUT_WFTRESP,Null,[M1=A0],{},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S4),
 [M1=A1/A2/A3],{inc(C30,C38)},1>
<IUT_WFTRESP_M1,OPEN,U!TCONresp(S5,S6,S7,S8),[M1=A4],
 {inc(C24,C37)},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S2),
 [M1=A5 and P_ITMP],{inc(C25,C37)},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S3),
 [M1=A5 and P_POC,{inc(C25,C37)},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S1),
 [M1=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
```

```
<IUT_WFTRESP_M1,IUT_WFTRESP, Null,[M1=A6],
 {append S25 to S27},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S4),[M1=A7],
 {inc(C32) peroctet,inc(C34,C38)},1>
<IUT_WFTRESP_M1,IUT_WFTRESP,Null,[M1=A8],
 {append S25 to S28},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S4),[M1=A9],
 {inc(C35) peroctet,inc(C36,C38)},1>
<IUT_WFTRESP_M1,IUT_WFTRESP_SAS,Null,[M1=A10],{},1>
<IUT_WFTRESP_M1,IUT_WFTRESP,Null,[M1=A11], {}, 1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S4),
 [M1=A12 and not(P_SINGDT)],
 {inc(C34) per octet,inc(C35, C37)},1>
<IUT_WFTRESP_M1,IDLE,U!TDISreq(S4),
 [M1=A12 and P_SINGDT],{inc(C33) per octet,
  inc(C34, C38)},1>
<IUT_WFTRESP_M1,IUT_WFTRESP,Null,[M1=A13], {}, 1>
<IUT_WFTRESP_M1,RST,Null,[M1=A14],{},1>
<IUT_WFTRESP_M1,STOP,Null,[M1=A15],{},1>
<UT_WFTCONF_M12,UT_WFTCONF,Null,[M12=A0], {},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S4),
 [M12=A1/A2/A3],{inc(C30,C38)},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S4),[M12=A4],
 {inc(C31,C38)},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S2),
 [M12=A5 and P_ITMP],{inc(C25,C37)},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S3),
 [M12=A5 and P_POC],{inc(C25,C37)},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S1),
[M12=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<UT_WFTCONF_M12,UT_WFTCONF,Null,[M12=A6],
 {append S25 to S27},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S4),[M12=A7],
 {inc(C32) peroctet,inc(C34,C38)},1>
<UT_WFTCONF_M12,UT_WFTCONF,Null,[M12=A8],
 {append S25 to S28},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S4),[M12=A9],
 {inc(C35) peroctet,inc(C36,C38)},1>
<UT_WFTCONF_M12,UT_WFTCONF_SAS,Null,[M10=A10],{},1>
<UT_WFTCONF_M12,UT_WFTCONF,Null,[M10=A11],{},1>
<UT_WFTCONF_M12,IDLE,U!TDISreq(S4),
 [M12=A12 and not(P_SINGDT)],
```

```
 {inc(C34) per octet, inc(C35, C37)},1>
<UT_WFTCONF_M12, IDLE, U!TDISreq(S4),
 [M12=A12 and P_SINGDT],
 {inc(C33) per octet, inc(C34, C38)},1>
<UT_WFTCONF_M12,UT_WFTCONF,Null,[M12=A13], {}, 1>
<UT_WFTCONF_M12,RST,Null,[M12=A14],{},1>
<UT_WFTCONF_M12,STOP,Null,[M12=A15], {}, 1>
<IUT_WFTRESP_M12,IUT_WFTRESP,Null,[M12=A0], {},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S4),
 [M12=A1/A2/A3],{inc(C30,C38)},1>
<IUT_WFTRESP_M12,OPEN,U!TCONresp(S5,S6,S7,S8),
 [M12=A4],{inc(C24,C37)},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S2),
 [M12=A5 and P_ITMP],{inc(C25,C37)},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S3),
 [M12=A5 and P_POC,{inc(C25,C37)},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S1),
 [M12=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<IUT_WFTRESP_M12,IUT_WFTRESP,Null,[M12=A6],
 {append S25 to S27},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S4),[M12=A7],
 {inc(C32) peroctet,inc(C34,C38)},1>
<IUT_WFTRESP_M12,IUT_WFTRESP,Null,[M12=A8],
 {append S25 to S28},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S4),[M12=A9],
 {inc(C35) peroctet,inc(C36,C38)},1>
<IUT_WFTRESP_M12,IUT_WFTRESP_SAS,Null,[M12=A10],{},1>
<IUT_WFTRESP_M12,IUT_WFTRESP,Null,[M12=A11],{},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S4),
 [M12=A12 and not(P_SINGDT)],
 {inc(C34) per octet, inc(C35, C37)},1>
<IUT_WFTRESP_M12,IDLE,U!TDISreq(S4),
 [M12=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34, C38)},1>
<IUT_WFTRESP_M12,IUT_WFTRESP,Null,[M12=A13], {}, 1>
<IUT_WFTRESP_M12,RST,Null,[M12=A14], {}, 1>
<IUT_WFTRESP_M12,STOP,Null,[M12=A15], {}, 1>
<OPEN_M12,IDLE,Null,[M12=A0], {},1>
<OPEN_M12, IDLE, U!TDISreq(S4), [M12=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M12,IDLE,U!TDISreq(S4),[M12=A4],
 {inc(C31,C38)},1>
```

```
<OPEN_M12,IDLE,U!TDISreq(S2),[M12=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M12,IDLE,U!TDISreq(S3),[M12=A5 and P_POC,
 {inc(C25,C37)},1>
<OPEN_M12,IDLE,U!TDISreq(S1),
 [M12=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M12,OPEN,Null,[M12=A6],{append S25 to S27},1>
<OPEN_M12,OPEN,U!TDTreq(S27),[M12=A7],
 {inc(C26) peroctet, inc(C27,C37)},1>
<OPEN_M12,OPEN,Null,[M12=A8],{append S25 to S28},1>
<OPEN_M12,OPEN,U!TEXreq(S27),[M12=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_M12,OPEN_SAS,Null,[M12=A10], {}, 1>
<OPEN_M12,OPEN,Null,[M12=A11],{},1>
<OPEN_M12,OPEN,U!TEXreq(single shot),
 [M12=A12 and not(P_SINGDT)],
 {inc(C28) per octet, inc(C29, C37)},1>
<OPEN_M12,OPEN,U!TDTreq(single short),
 [M12=A12 and P_SINGDT],
 {inc(C26) per octet, inc(C27, C37)},1>
<OPEN_M12,OPEN,Null,[M12=A13],{},1>
<OPEN_M12,RST,Null,[M12=A14],{},1>
<OPEN_M12,STOP,Null,[M12=A15],{},1>
<IDLE_M3,IDLE,Null,[M3=A0], {},1>
<IDLE_M3,UT_WFTCONF,U!TCONreq(S9,S10,S11,S12,S13),
 [M3=A1],{inc(C23,C37)},1>
<IDLE_M3,UT_WFTCONF,U!TCONreq(S17,S10,S11,S12,S13),
 [M3=A2],{inc(C23,C37)},1>
<IDLE_M3,UT_WFTCONF,U!TCONreq(S22,S10,S11,S12,S13),
 [M3=A3],{inc(C23,C37)},1>
<IDLE_M3,IDLE,Null,[M3=A4],{inc(C31,C38)},1>
<IDLE_M3,IDLE,Null,[M3=A5],{inc(C32,C38)},1>
<IDLE_M3,IDLE,Null,[M3=A6],{append S25 to S27},1>
<IDLE_M3,IDLE,Null,[M3=A7],
 {inc(C33) per octet,inc(C34,C38)},1>
<IDLE_M3,IDLE,Null,[M3=A8],{append S25 to S28},1>
<IDLE_M3,IDLE,Null,[M3=A9],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M3,IDLE_SAS,Null,[M3=A10],{},1>
<IDLE_M3,IDLE,Null,[M3=A11],{},1>
<IDLE_M3,IDLE,Null,[M3=A12 and not(P_SINGDT)],
 {inc(C35) per octet,inc(C36, C38)},1>
```

```
<IDLE_M3,IDLE, Null,[M3=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34, C38)},1>
<IDLE_M3,IDLE,Null,[M3=A13],{},1>
<IDLE_M3,RST,Null,[M3=A14],{},1>
<IDLE_M3,STOP,Null,[M3=A15],{},1>
<IDLE_M9,IDLE,Null,[M9=A0], {},1>
<IDLE_M9,UT_WFTCONF,U!TCONreq(S9,S10,S11,S12,S13),
 [M9=A1],{inc(C23,C37)},1>
<IDLE_M9,UT_WFTCONF,U!TCONreq(S17,S10,S11,S12,S13),
 [M9=A2],{inc(C23,C37)},1>
<IDLE_M9,UT_WFTCONF,U!TCONreq(S22,S10,S11,S12,S13),
 [M9=A3],{inc(C23,C37)},1>
<IDLE_M9,IDLE,Null,[M9=A4],{inc(C31,C38)},1>
<IDLE_M9,IDLE,Null,[M9=A5],{inc(C32,C38)},1>
<IDLE_M9,IDLE,Null,[M9=A6],{append S25 to S27},1>
<IDLE_M9,IDLE,Null,[M9=A7],
 {inc(C33) per octet, inc(C34,C38)},1>
<IDLE_M9,IDLE,Null,[M9=A8],{append S25 to S28},1>
<IDLE_M9,IDLE,Null,[M9=A9],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M9,IDLE_SAS,Null,[M9=A10],{},1>
<IDLE_M9,IDLE,Null,[M9=A11],{}.1>
<IDLE_M9,IDLE,Null,[M9=A12 and not(P_SINGDT)],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M9,IDLE,Null,[M9=A12 and P_SINGDT],
 {inc(C33) per octet, inc(C34, C38)},1>
<IDLE_M9,IDLE,Null,[M9=A13],{},1>
<IDLE_M9,RST,Null,[M9=A14],{},1>
<IDLE_M9,STOP,Null,[M9=A15],{},1>

/*If a TCONconf is received in IDLE state,
  then do exec(M12).*/
<IDLE_M12,IDLE,Null,[M12=A0],{},1>
<IDLE_M12,UT_WFTCONF,U!TCONreq(S9,S10,S11,S12,S13),
 [M12=A1],{inc(C23,C37)},1>
<IDLE_M12,UT_WFTCONF,U!TCONreq(S17,S10,S11,S12,S13),
 [M12=A2],{inc(C23,C37)},1>
<IDLE_M12,UT_WFTCONF,U!TCONreq(S22,S10,S11,S12,S13),
 [M12=A3],{inc(C23,C37)},1>
<IDLE_M12,IDLE,Null,[M12=A4],{inc(C31,C38)},1>
<IDLE_M12,IDLE,Null,[M12=A5],{inc(C32,C38)},1>
<IDLE_M12,IDLE,Null,[M12=A6],{append S25 to S27},1>
<IDLE_M12,IDLE,Null,[M12=A7],
```

```
 {inc(C33) per octet, inc(C34,C38)},1>
<IDLE_M12,IDLE,Null,[M12=A8],{append S25 to S28}>
<IDLE_M12,IDLE,Null,[M12=A9],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M12,IDLE_SAS,Null,[M12=A10],{}, 1>
<IDLE_M12,IDLE,Null,[M12=A11],{},1>
<IDLE_M12,IDLE,Null,[M12=A12 and not(P_SINGDT)],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M12,IDLE,Null,[M12=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34,C38)},1>
<IDLE_M12,IDLE,Null,[M12=A13],{},1>
<IDLE_M12,RST,Null,[M12=A14],{},1>
<IDLE_M12,STOP,Null,[M12=A15],{},1>
/* If a TCONconf is received in state OPEN,
   then do exec(M2). */
<OPEN_M2,OPEN,Null,[M2=A0],{},1>
<OPEN_M2,IDLE,U!TDISreq(S4),[M2=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M2,IDLE,U!TDISreq(S4),[M2=A4],{inc(C31,C38)},1>
<OPEN_M2,IDLE,U!TDISreq(S2),[M2=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M2,IDLE,U!TDISreq(S3),[M2=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_M2,IDLE,U!TDISreq(S1),
 [M2=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M2,OPEN,Null,[M2=A6],{append S25 to S27},1>
<OPEN_M2,OPEN,U!TDTreq(S27),[M2=A7],
 {inc(C26) peroctet, inc(C27,C37)},1>
<OPEN_M2,OPEN,Null,[M2=A8],{append S25 to S28},1>
<OPEN_M2,OPEN,U!TEXreq(S27),[M2=A9],
 {inc(C28) peroctet, inc(C29,C37)},1>
<OPEN_M2,OPEN_SAS,Null,[M2=A10], {}, 1>
<OPEN_M2,OPEN,Null,[M2=A11],{},1>
<OPEN_M2,OPEN,U!TEXreq(single shot),
 [M2=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29, C37)},1>
<OPEN_M2,OPEN,U!TDTreq(single short),
 [M2=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_M2,OPEN,Null,[M2=A13],{},1>
<OPEN_M2,RST,Null,[M2=A14],{},1>
<OPEN_M2,STOP,Null,[M2=A15],{},1>
```

```
/*If a TDTind is received in state OPEN,
  then do exec(M4).*/
<OPEN_M4,OPEN,Null,[M4=A0],{},1>
<OPEN_M4,IDLE,U!TDISreq(S4),[M4=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M4,IDLE,U!TDISreq(S4),[M4=A4],{inc(C31,C38)},1>
<OPEN_M4,IDLE,U!TDISreq(S2),[M4=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M4,IDLE,U!TDISreq(S3),[M4=A5 and P_POC,
 {inc(C25,C37)},1>
<OPEN_M4,IDLE,U!TDISreq(S1),
 [M4=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M4,OPEN,Null,[M4=A6],{append S25 to S27},1>
<OPEN_M4,OPEN,U!TDTreq(S27),[M4=A7],
 {inc(C26) peroctet, inc(C27,C37)},1>
<OPEN_M4,OPEN,Null,[M4=A8],{append S25 to S28},1>
<OPEN_M4,OPEN,U!TEXreq(S27),[M4=A9],
 {inc(C28) peroctet, inc(C29,C37)},1>
<OPEN_M4,OPEN_SAS,Null,[M4=A10],{},1>
<OPEN_M4,OPEN,Null,[M4=A11],{},1>
<OPEN_M4,OPEN,U!TEXreq(single shot),
 [M4=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29, C37)},1>
<OPEN_M4,OPEN,U!TDTreq(single short),
 [M4=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_M4,OPEN,Null,[M4=A13],{},1>
<OPEN_M4,RST,Null,[M4=A14],{},1>
<OPEN_M4,STOP,Null,[M4=A15],{},1>
/* If a TDTind is received in state OPEN,
   then do exec(M5). */
<OPEN_M5,OPEN,Null,[M5=A0],{},1>
<OPEN_M5,IDLE,U!TDISreq(S4),[M5=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M5,IDLE,U!TDISreq(S4),[M5=A4],{inc(C31,C38)},1>
<OPEN_M5,IDLE,U!TDISreq(S2),[M5=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M5,IDLE,U!TDISreq(S3),[M5=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_M5,IDLE,U!TDISreq(S1),
 [M5=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M5,OPEN,Null,[M5=A6],{append S25 to S27},1>
```

```
<OPEN_M5,OPEN,U!TDTreq(S27),[M5=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_M5,OPEN,Null,[M5=A8],{append S25 to S28},1>
<OPEN_M5,OPEN,U!TEXreq(S27),[M5=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_M5,OPEN_SAS,Null,[M9=A10],[],1>
<OPEN_M5,OPEN,Null,[M9=A11],[],1>
<OPEN_M5,OPEN,U!TEXreq(single shot),
 [M5=A12 and not(P_SINGDT)],
 {inc(C28) per octet, inc(C29, C37)},1>
<OPEN_M5,OPEN,U!TDTreq(single short),
 [M5=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27, C37)},1>
<OPEN_M5,OPEN,Null,[M5=A13],{},1>
<OPEN_M5,RST,Null,[M5=A14],{},1>
<OPEN_M5,STOP,Null,[M5=A15],{},1>
/*If a TEXind is received in state OPEN,
  then do exec(M6). */
<OPEN_M6,OPEN,Null,[M6=A0],{},1>
<OPEN_M6,IDLE,U!TDISreq(S4),[M6=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M6,IDLE,U!TDISreq(S4),[M6=A4],{inc(C31,C38)},1>
<OPEN_M6,IDLE,U!TDISreq(S2),[M6=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M6,IDLE,U!TDISreq(S3),[M6=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_M6,IDLE,U!TDISreq(S1),
 [M6=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M6,OPEN,Null,[M6=A6],{append S25 to S27},1>
<OPEN_M6,OPEN,U!TDTreq(S27),[M6=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_M6,OPEN,Null,[M6=A8],{append S25 to S28},1>
<OPEN_M6,OPEN,U!TEXreq(S27),[M6=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_M6,OPEN_SAS,Null,[M6=A10],{},1>
<OPEN_M6,OPEN,Null,[M6=A11],{},1>
<OPEN_M6,OPEN,U!TEXreq(single shot),
 [M6=A12 and not(P_SINGDT)],
 {inc(C28) per octet, inc(C29, C37)},1>
<OPEN_M6,OPEN,U!TDTreq(single short),
 [M6=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27, C37)},1>
```

```
<OPEN_M6,OPEN,Null,[M6=A13],{},1>
<OPEN_M6,RST,Null,[M6=A14],{},1>
<OPEN_M6,STOP,Null,[M6=A15],{},1>
/*If a TEXind is received in state OPEN_SAS,
  then do exec(M6).*/
<OPEN_SAS_M6, OPEN_SAS, Null,[M6=A0], {},1>
<OPEN_SAS_M6,IDLE,U!TDISreq(S4),[M6=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_SAS_M6,IDLE,U!TDISreq(S4),[M6=A4],
 {inc(C31,C38)},1>
<OPEN_SAS_M6,IDLE,U!TDISreq(S2),
 [M6=A5 and P_ITMP],{inc(C25,C37)},1>
<OPEN_SAS_M6,IDLE,U!TDISreq(S3),
 [M6=A5 and P_POC,{inc(C25,C37)},1>
<OPEN_SAS_M6,IDLE,U!TDISreq(S1),
 [M6=A5 and not(P_ITMP or P_POC)],
 {inc(C25,C37)},1>
<OPEN_SAS_M6,OPEN,Null,[M6=A6],{append S25 to S27},1>
<OPEN_SAS_M6,OPEN,U!TDTreq(S27),[M6=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_SAS_M6,OPEN,Null,[M6=A8],{append S25 to S28},1>
<OPEN_SAS_M6,OPEN,U!TEXreq(S27),[M6=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_SAS_M6,OPEN_SAS,Null,[M6=A10],{},1>
<OPEN_SAS_M6,OPEN,Null,[M6=A11],{},1>
<OPEN_SAS_M6,OPEN,U!TEXreq(single shot),
 [M6=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29, C37)},1>
<OPEN_SAS_M6,OPEN,U!TDTreq(single short),
 [M6=A12 and P_SINGDT],
 {inc(C26) per octet, inc(C27, C37)},1>
<OPEN_SAS_M6,OPEN,Null,[M6=A13],{}, 1>
<OPEN_SAS_M6,RST,Null,[M6=A14],{},1>
<OPEN_SAS_M6,STOP,Null,[M6=A15],{},1>
/*If a TEXind is received in state OPEN_SAS,
  then do exec(M7).*/
<OPEN_M7,OPEN,Null,[M7=A0],{},1>
<OPEN_M7,IDLE,U!TDISreq(S4),[M7=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M7,IDLE,U!TDISreq(S4),[M7=A4],{inc(C31,C38)},1>
<OPEN_M7,IDLE,U!TDISreq(S2),[M7=A5 and P_ITMP],
 {inc(C25,C37)},1>
```

```
<OPEN_M7,IDLE,U!TDISreq(S3),[M7=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_M7,IDLE,U!TDISreq(S1),
 [M7=A5 and not(P_ITMP or P_POC)],
 {inc(C25,C37)},1>
<OPEN_M7,OPEN_SAS,Null,[M7=A6],{append S25 to S27},1>
<OPEN_M7,OPEN_SAS,U!TDTreq(S27),[M7=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_M7,OPEN_SAS,Null,[M7=A8],{append S25 to S28},1>
<OPEN_M7,OPEN_SAS,U!TEXreq(S27),[M7=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_M7,OPEN_SAS,Null,[M7=A10],{},1>
<OPEN_M7,OPEN_SAS,Null,[M7=A11],{},1>
<OPEN_M7,OPEN_SAS,U!TEXreq(single shot),
 [M7=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29,C37)},1>
<OPEN_M7,OPEN_SAS,U!TDTreq(single short),
 [M7=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_M7,OPEN_SAS,Null,[M7=A13],{},1>
<OPEN_M7,RST,Null,[M7=A14],{},1>
<OPEN_M7,STOP,Null,[M7=A15],{},1>

/*If a TEXind is received in state OPEN_SAS,
   then do exec(M7).*/
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A0],{},1>
<OPEN_SAS_M7,IDLE,U!TDISreq(S4),[M7=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_SAS_M7,IDLE,U!TDISreq(S4),[M7=A4],
 {inc(C31,C38)},1>
<OPEN_SAS_M7,IDLE,U!TDISreq(S2),[M7=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_SAS_M7,IDLE,U!TDISreq(S3),[M7=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_SAS_M7,IDLE,U!TDISreq(S1),
 [M7=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A6],
 {append S25 to S27},1>
<OPEN_SAS_M7,OPEN_SAS,U!TDTreq(S27),[M7=A7],
 {inc(C26) peroctet,
 inc(C27,C37)},1>
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A8],
 {append S25 to S28},1>
```

```
<OPEN_SAS_M7,OPEN_SAS,U!TEXreq(S27),[M7=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A10],{},1>
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A11],{},1>
<OPEN_SAS_M7,OPEN_SAS,U!TEXreq(single shot),
 [M7=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29,C37)},1>
<OPEN_SAS_M7,OPEN_SAS,U!TDTreq(single short),
 [M7=A12 and P_SINGDT],
 {inc(C26) per octet, inc(C27, C37)},1>
<OPEN_SAS_M7,OPEN_SAS,Null,[M7=A13],{},1>
<OPEN_SAS_M7,RST,Null,[M7=A14],{},1>
<OPEN_SAS_M7,STOP,Null,[M7=A15],{},1>
<OPEN_M9,OPEN,Null,[M9=A0],{},1>
<OPEN_M9,IDLE,U!TDISreq(S4),[M9=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_M9,IDLE.U!TDISreq(S4),[M9=A4],{inc(C31,C38)},1>
<OPEN_M9,IDLE,U!TDISreq(S2),[M9=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_M9,IDLE,U!TDISreq(S3),[M9=A5 and P_POC],
 {inc(C25,C37)},1>
<OPEN_M9,IDLE,U!TDISreq(S1),
 [M9=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_M9,OPEN,Null,[M9=A6],{append S25 to S27},1>
<OPEN_M9,OPEN,U!TDTreq(S27),[M9=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_M9,OPEN,Null,[M9=A8],{append S25 to S28},1>
<OPEN_M9,OPEN,U!TEXreq(S27),[M9=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_M9,OPEN_SAS,Null,[M9=A10],{},1>
<OPEN_M9,OPEN,Null,[M9=A11],{},1>
<OPEN_M9,OPEN,U!TEXreq(single shot),
 [M9=A12 and not(P_SINGDT)],
 {inc(C28) per octet, inc(C29, C37)},1>
<OPEN_M9,OPEN,U!TDTreq(single short),
 [M9=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_M9,OPEN,Null,[M9=A13],{},1>
<OPEN_M9,RST,Null,[M9=A14],{},1>
<OPEN_M9,STOP,Null,[M9=A15],{},1>
/* After receiving a CLE in state IDLE,
   TMP does exec(M10).*/
```

```
<IDLE_M10,IDLE,Null,[M10=A0],{},1>
<IDLE_M10,UT_WFTCONF,U!TCONreq(S9,S10,S11,S12,S13),
 [M10=A1],{inc(C23,C37)},1>
<IDLE_M10,UT_WFTCONF,U!TCONreq(S17,S10,S11,S12,S13),
 [M10=A2],{inc(C23,C37)},1>
<IDLE_M10,UT_WFTCONF,U!TCONreq(S22,S10,S11,S12,S13),
 [M10=A3],{inc(C23,C37)},1>
<IDLE_M10,IDLE,Null,[M10=A4],{inc(C31,C38)},1>
<IDLE_M10,IDLE,Null,[M10=A5],{inc(C32,C38)},1>
<IDLE_M10,IDLE,Null,[M10=A6],{append S25 to S27},1>
<IDLE_M10,IDLE,Null,[M10=A7],
 {inc(C33) per octet,inc(C34,C38)},1>
<IDLE_M10,IDLE,Null,[M10=A8],{append S25 to S28},1>
<IDLE_M10,IDLE,Null,[M10=A9],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M10,IDLE_SAS,Null,[M10=A10],{},1>
<IDLE_M10,IDLE,Null,[M10=A11],{},1>
<IDLE_M10,IDLE,Null,[M10=A12 and not(P_SINGDT)],
 {inc(C35) per octet,inc(C36,C38)},1>
<IDLE_M10,IDLE,Null,[M10=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34,C38)},1>
<IDLE_M10,IDLE,Null,[M10=A13],{},1>
<IDLE_M10,RST,Null,[M10=A14],{},1>
<IDLE_M10,STOP,Null,[M10=A15],{},1>
/* After receiving a CLE in state UT_WFTCONF,
   TMP does exec(M10).*/
<UT_WFTCONF_M10,UT_WFTCONF,Null,[M10=A0],{},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),[M10=A1/A2/A3],
 {inc(C30,C38)},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),[M10=A4],
 {inc(C31,C38)},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S2),
 [M10=A5 and P_ITMP],{inc(C25,C37)},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S3),
 [M10=A5 and P_POC],{inc(C25,C37)},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S1),
 [M10=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<UT_WFTCONF_M10,UT_WFTCONF,Null,[M10=A6],
 {append S25 to S27},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),[M10=A7],
 {inc(C32) peroctet,inc(C34,C38)},1>
<UT_WFTCONF_M10,UT_WFTCONF,Null,[M10=A8],
```

```
 {append S25 to S28},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),[M10=A9],
 {inc(C35) peroctet,inc(C36,C38)},1>
<UT_WFTCONF_M10,UT_WFTCONF_SAS,Null,[M10=A10],{},1>
<UT_WFTCONF_M10,UT_WFTCONF,Null,[M10=A11],{},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),
 [M10=A12 and not(P_SINGDT)],
 {inc(C34) per octet,inc(C35, C37)},1>
<UT_WFTCONF_M10,IDLE,U!TDISreq(S4),
 [M10=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34,C38)},1>
<UT_WFTCONF_M10,UT_WFTCONF,Null,[M10=A13],{},1>
<UT_WFTCONF_M10,RST,Null,[M10=A14],{},1>
<UT_WFTCONF_M10,STOP,Null,[M10=A15],{},1>
/*After receiving CLE in state IUT_WFTRESP,
  TMP does exec(M10).*/
<IUT_WFTRESP_M10,IUT_WFTRESP,Null,[M01=A0],{},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S4),
 [M10=A1/A2/A3],{inc(C30,C38)},1>
<IUT_WFTRESP_M10,OPEN,U!TCONresp(S5,S6,S7,S8),
 [M10=A4],{inc(C24,C37)},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S2),
 [M10=A5 and P_ITMP],{inc(C25,C37)},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S3),
 [M10=A5 and P_POC,
 {inc(C25,C37)},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S1),
 [M10=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<IUT_WFTRESP_M10,IUT_WFTRESP,Null,[M10=A6],
 {append S25 to S27},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S4),[M10=A7],
 {inc(C32) peroctet,inc(C34,C38)},1>
<IUT_WFTRESP_M10,IUT_WFTRESP,Null,[M10=A8],
 {append S25 to S28},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S4),[M10=A9],
 {inc(C35) peroctet,inc(C36,C38)},1>
<IUT_WFTRESP_M10,IUT_WFTRESP_SAS,Null,[M10=A10],{},1>
<IUT_WFTRESP_M10,IUT_WFTRESP,Null,[M10=A11],{},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S4),
 [M10=A12 and not(P_SINGDT)],
 {inc(C34) per octet,inc(C35,C37)},1>
<IUT_WFTRESP_M10,IDLE,U!TDISreq(S4),
```

```
 [M10=A12 and P_SINGDT],
 {inc(C33) per octet,inc(C34,C38)},1>
<IUT_WFTRESP_M10,IUT_WFTRESP,Null,[M10=A13],{},1>
<IUT_WFTRESP_M10,RST,Null,[M10=A14],{},1>
<IUT_WFTRESP_M10,STOP,Null,[M10=A15],{},1>
/*After receiving a CLE in state OPEN,
  the TMP does exec(M10).*/
<OPEN_8,OPEN,Null,[M10=A0],{},1>
<OPEN_8,IDLE,U!TDISreq(S4),[M10=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_8,IDLE,U!TDISreq(S4),[M10=A4],{inc(C31,C38)},1>
<OPEN_8,IDLE,U!TDISreq(S2),[M10=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_8,IDLE,U!TDISreq(S3),[M10=A5 and P_POC,
 {inc(C25,C37)},1>
<OPEN_8,IDLE,U!TDISreq(S1),
 [M10=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_8,OPEN_SAS,Null,[M10=A6],{append S25 to S27},1>
<OPEN_8, OPEN_SAS, U!TDTreq(S27), [M10=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_8,OPEN_SAS,Null,[M10=A8],{append S25 to S28},1>
<OPEN_8,OPEN_SAS,U!TEXreq(S27),[M10=A9],
 {inc(C28) peroctet, inc(C29,C37)},1>
<OPEN_8,OPEN_SAS,Null,[M10=A10],{},1>
<OPEN_8,OPEN_SAS,Null,[M10=A11],{},1>
<OPEN_8,OPEN_SAS,U!TEXreq(single shot),
 [M10=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29,C37)},1>
<OPEN_8,OPEN_SAS,U!TDTreq(single short),
 [M10=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_8,OPEN_SAS,Null,[M10=A13],{},1>
<OPEN_8,RST,Null,[M10=A14],{},1>
<OPEN_8,STOP,Null,[M10=A15], {}, 1>

/* When the TMP receives an invalid TMPDU in the OPEN
 state,take the following action depending on M11. */
<OPEN_16,OPEN,Null,[M11=A0],{},1>
<OPEN_16,IDLE,U!TDISreq(S4),[M11=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_16,IDLE,U!TDISreq(S4),[M11=A4],
 {inc(C31,C38)},1>
<OPEN_16,IDLE,U!TDISreq(S2),[M11=A5 and P_ITMP],
```

```
 {inc(C25,C37)},1>
<OPEN_16,IDLE,U!TDISreq(S3),[M11=A5 and P_POC,
 {inc(C25,C37)},1>
<OPEN_16,IDLE,U!TDISreq(S1),
 [M11=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_16,OPEN,Null,[M11=A6],{append S25 to S27},1>
<OPEN_16,OPEN,U!TDTreq(S27),[M11=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_16,OPEN,Null,[M11=A8],{append S25 to S28},1>
<OPEN_16,OPEN,U!TEXreq(S27),[M11=A9],
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_16,OPEN_SAS,Null,[M11=A10],{},1>
<OPEN_16,OPEN,Null,[M11=A11],{},1>
<OPEN_16,OPEN,U!TEXreq(single shot),
 [M11=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29,C37)},1>
<OPEN_16,OPEN,U!TDTreq(single short),
 [M11=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_16,OPEN,Null,[M11=A13],{},1>
<OPEN_16,RST,Null,[M11=A14],{},1>
<OPEN_16,STOP,Null,[M11=A15],{},1>
/* When the TMP receives an invalid TMPDU in the OPEN_SAS
 state, take the following action depending on M11.*/
<OPEN_S16,OPEN_SAS,Null,[M11=A0],{},1>
<OPEN_S16,IDLE,U!TDISreq(S4),[M11=A1/A2/A3],
 {inc(C30,C38)},1>
<OPEN_S16,IDLE,U!TDISreq(S4),[M11=A4],
 {inc(C31,C38)},1>
<OPEN_S16,IDLE,U!TDISreq(S2),[M11=A5 and P_ITMP],
 {inc(C25,C37)},1>
<OPEN_S16,IDLE,U!TDISreq(S3),[M11=A5 and P_POC,
 {inc(C25,C37)},1>
<OPEN_S16,IDLE,U!TDISreq(S1),
 [M11=A5 and not(P_ITMP or P_POC)],{inc(C25,C37)},1>
<OPEN_S16,OPEN_SAS,Null,[M11=A6],
 {append S25 to S27},1>
<OPEN_S16,OPEN_SAS,U!TDTreq(S27),[M11=A7],
 {inc(C26) peroctet,inc(C27,C37)},1>
<OPEN_S16,OPEN_SAS,Null,[M11=A8],
 {append S25 to S28},1>
<OPEN_S16,OPEN_SAS,U!TEXreq(S27),[M11=A9],
```

```
 {inc(C28) peroctet,inc(C29,C37)},1>
<OPEN_S16,OPEN_SAS,Null,[M11=A10],{},1>
<OPEN_S16,OPEN_SAS,Null,[M11=A11],{},1>
<OPEN_S16,OPEN_SAS,U!TEXreq(single shot),
 [M11=A12 and not(P_SINGDT)],
 {inc(C28) per octet,inc(C29,C37)},1>
<OPEN_S16,OPEN_SAS,U!TDTreq(single short),
 [M11=A12 and P_SINGDT],
 {inc(C26) per octet,inc(C27,C37)},1>
<OPEN_S16,OPEN_SAS,Null,[M11=A13],{},1>
<OPEN_S16,RST,Null,[M11=A14],{},1>
<OPEN_S16,STOP,Null,[M11=A15],{},1>
<RST,IDLE_M9,Internal_START,[T],
 {C1:=0,C2:= 0,..,C38:=0,
  M1:=A4,M2:=A0,..,M10:=A0,M11:=A5,
  M12:=A5,M13:=A0,.., M25:=A0,
  S1:= ..., S2:=.., ... S28:=..},1>
---------------------------------------------------------
                         E   N   D
---------------------------------------------------------
```

Appendix 3

Global state space of the single connection CS Test Verification System in Chapter 6.

```
----------------------------------------------------------------
Global States
Abbreviations:/* Vc is a set of constant values. */
Vc   = {VAL:="test_data", Called_addr:="you",
       Calling_addr:="me",Exp_option:="No",Qos:="1",
       User_data0:="any_data",TS5:="1",
       TS6:="you", TS7:="No", TS8:=VAL}
/* Var is attached to each state. */
Var = {(Vc,seqrecak,seqsendt),(opt,PRSeq,PRcredit),
       (S5,S6,S7,S8,M1,M3,M9),Verdict}
/* V1 is Var with initial values. */
V1  = {(Vc,seqrecak:=0,seqsendt:=0),
      (opt:=Null,PRSeq:=0,PRcredit:=2),
      (S5:="1",S6:="you",S7:="No",
       S8:="Null",M1:=A4,M3:=A0,M9:=A9),
      Verdict:=Null}
V6  = {(Vc,seqrecak:=0,seqsendt:=0),
      (opt:=Null,PRSeq:=0,PRcredit:=2),
      (S5:="1",S6:="you",S7:="No",S8:="Null",
       M1:=A4,M3:=A0,M9:=A9),Verdict:=Fail}
V9  = {(Vc,seqrecak:=0, seqsendt:=0),
      (opt:=1, PRSeq:=0,PRcredit:=2),

      (S5:="1",S6:="you",S7:="No",S8:="Null",
       M1:=A4,M3:=A0,M9:=A9),Verdict:=Null}
V20 = {(Vc,seqrecak:=0, seqsendt:=1),
       (opt:=1, PRSeq:=0,PRcredit:=2),
       (S5:="1",S6:="you",S7:="No",
        S8:="Null",M1:=A4,M3:=A0,M9:=A9),
        Verdict:=Null}
V26 = {(Vc,seqrecak:=0, seqsendt:=1),
       (opt:=1, PRSeq:=1,PRcredit:=1),
       (S5:="1",S6:="you",S7:="No",
        S8:="Null",M1:=A4,M3:=A0,M9:=A9),
        Verdict:=Null}
V28 = {(Vc,seqrecak:=0, seqsendt:=1),
       (opt:=1, PRSeq:=2,PRcredit:=0),
       (S5:="1",S6:="you",S7:="No",
        S8:="Null",M1:=A4,M3:=A0,M9:=A9),
```

```
        Verdict:=Null}
V33 = {(Vc,seqrecak:=0, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Null}
V40 = {(Vc,seqrecak:=0, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Fail}
V41 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Null}

V46 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Fail}
V60 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Fail}
V63 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=0,PRcredit:=2),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Null}
V71 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=0,PRcredit:=2),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Pass}
V78 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=0,PRcredit:=2),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Fail}
```

```
V96 = {(Vc,seqrecak:=0, seqsendt:=1),
       (opt:=1, PRSeq:=2,PRcredit:=0),
       (S5:="1",S6:="you",S7:="No",
        S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
        Verdict:=Fail}

V97 = {(Vc,seqrecak:=0, seqsendt:=1),
       (opt:=1, PRSeq:=2,PRcredit:=0),
       (S5:="1",S6:="you",S7:="No",
        S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
        Verdict:=Fail}

V103 = {(Vc,seqrecak:=2, seqsendt:=1),
        (opt:=1, PRSeq:=2,PRcredit:=0),
        (S5:="1",S6:="you",S7:="No",
         S8:="test_data",M1:=A4,M3:=A0,M9:=A9),
         Verdict:=Fail}

Verdict attachments:
Fail: g6,g40,g46,g60,g78,g96,g97,g102,g103,
      g104,g105,g106,g107,g108,g109,g110
Pass: g71

g1:  <2,1,1,IDLE, E,E,E,E,E,E, V1>{INIT}
g2:  <2,1,1,IDLE_M9, E,E,E,E,E,E, V1>{}
g3:  <2,1,1,IDLE, E,E,E,E,E,E, V1>
     {AT(Tsend(L, NDTreq(CR)))}
g4:  <3,1,1,IDLE, NDTreq(CR("you","me","No","1",
      "any_data")),E,E,E,E,E, V1>
      {AFTER(Tsend(L, NDTreq(CR)))}
g5:  <4,1,1,IDLE,
      NDTreq(CR("you","me","No","1","any_data")),
      E,E,E,E,E, V1>{}
g7:  <4,4,1,IDLE, E,E,E,E,E,E, V1>{}
g8:  <4,1,1,IDLE, E,E,NDTind(CR("you","me","No",
      "1","any_data")),E,E,E, V1>
      {AT(Sreceive(N,NDTind(CR)))}
g9:  <4,1,5,IDLE, E,E,E,E,E,E, V9>
      {AFTER(Sreceive(N,NDTind(CR))),
       AT(Ssend(U,TCONind))}
g10: <4,1,6,IDLE, E,E,E,E,
      TCONind("you","me","No","1","any_data"),E, V9>
      {AFTER(Ssend(U,TCONind)),
```

```
        AT(Treceive(U,TCONind))}
g11: <4,1,6,IUT_WFTRESP_M1, E,E,E,E,E,E, V9>u
      {AFTER(Treceive(U,TCONind)),
       AT(Tsend(U,TCONresp))}
g12: <4,1,6,OPEN, E,E,E,E,E,
      TCONresp("1","you","No","any_data"), V9>
      {AFTER(Tsend(U,TCONresp)),
       AT(Sreceive(U,TCONresp))}
g13: <4,1,7,OPEN, E,E,E,E,E,E, V9>
      {AFTER(Sreceive(U,TCONresp)),
       AT(Ssend(N,NDTreq(CC)))}
g14: <4,1,10,OPEN, E,E,E,
      NDTreq(CC("me","1","No","2","any_data")),E,E,V9>
      {AFTER(Ssend(N,NDTreq(CC)))}
g15: <4,8,10,OPEN, E,E,E,E,E,E, V9>{}
g16: <4,1,10,OPEN, E,
      NDTind(CC("me","1","No","2","any_data")),
      E,E,E,E,V9>
      {AT(Treceive(L,NDTind(CC)))}
g6:  <5,1,10,OPEN, E,NDTind(CC("me","1","No","2",
      "any_data")),E,E,E,E, V6>
      {(Verdict = Fail)}
g17: <6,1,10,OPEN, E,E,E,E,E,E, V9>
      {AFTER(Treceive(L,NDTind(CC)))}
g18: <7,1,10,OPEN, E,E,E,E,E,E, V9>
     {AT(Tsend(L,NDTreq(DT)))}
g19: <8,1,10,OPEN,
       NDTreq(DT("TMP1(M1=A4,M2=A0,.,M10=A0,M11=A5,
       M12=A5,M13=A0,.,M25=A0)","True"))),E,E,E,E,E,V9>
      {AFTER(Tsend(L,NDTreq(DT))),
       AFTER(Tsend(L,NDTreq(DT)))}
g20: <9,1,10,OPEN,
      {NDTreq(DT("TMP1(M1=A4,M2=A0,.,M10=A0,
       M11=A5,M12=A5,M13=A0,.,
       M25=A0)","True"))),
       NDTreq(DT("TMP4("1","you","No","test_data"))))},
         E,E,E,E,E,V20>
      {AFTER(Tsend(L,NDTreq(DT)))}
g21: <9,4,10,OPEN,
      NDTreq(DT("TMP4("1","you","No","test_data"))))},
      E,E,E,E,E,V20>{}
g22: <9,1,10,OPEN,
```

```
        NDTreq(DT("TMP4("1","you","No","test_data"))),E,
        NDTind(DT("TMP1(M1=A4,M2=A0,..,M10=A0,M11=A5,

        M12=A5,M13=A0,..,M25=A0)","True"))),
        E,E,E, V20>{}
g23: <9,4,10,OPEN, E,E,
         NDTind(DT("TMP1(M1=A4,M2=A0,...,M10=A0,M11=A5,
         M12=A5,M13=A0,..,M25=A0)","True"))),E,E,E,V20>{}
g24: <9,1,10,OPEN, E,E,
         {NDTind(DT("TMP1(M1=A4,M2=A0,..,M10=A0,M11=A5,
        M12=A5,M13=A0,..,M25=A0)","True"))),
        NDTind(DT("TMP4("1","you","No","test_data")))},
        E,E,E,V20>{AT(Sreceive(N, NDTind(DT)))}
g25: <9,1,17,OPEN,E,E,NDTind(DT("TMP4("1","you","No",
        "test_data"))),E,E,E, V20>
        {AFTER(Sreceive(N,NDTind(DT))),
         AT(Ssend(U,TDATAind))}
g26: <9,1,10,OPEN, E,E,
        NDTind(DT("TMP4("1","you","No","test_data"))),E,
         TDTind("any_data", "True"),E, V26>
        {AFTER(Ssend(U,TDATAind)),
         AT(Sreceive(N,NDTind(DT)))}
g27: <9,1,17,OPEN, E,E,E,E,
        TDTind("any_data","True"),E, V26>
        {AFTER(Sreceive(N,NDTind(DT))),
         AT(Ssend(U,TDATAind))}
g28: <9,1,10,OPEN, E,E,E,E,{TDTind("any_data","True",
                TDTind("test_data","True")},E, V28>
        {AFTER(Ssend(U,TDATAind))),
         AT(Treceive(U,TDATAind))}
g29: <9,1,10,OPEN_1, E,E,E,E,
        TDTind("test_data","True"),E, V28>
        {AFTER(Treceive(U,TDATAind))}
g30: <9,1,10,OPEN_M9, E,E,E,E,
        TDTind("test_data","True"),E, V28>{}
g31: <9,1,10,OPEN, E,E,E,E,
        TDTind("test_data","True"),E, V28>
        {AT(Treceive(U,TDATAind))}
g32: <9,1,10,OPEN_1, E,E,E,E,E,E, V28>
       {AFTER(Treceive(U,TDATAind))}
g33: <9,1,10,OPEN, E,E,E,E,E,E, V33>{}
g34: <9,1,18,OPEN, E,E,E,E,E,E, V33>
```

```
      {AT(Ssend(N,NDTreq(AK)))}
g35: <9,1,10,OPEN,E,E,E,NDTreq(AK("2","0")),E,E,V33>
      {AFTER(Ssend(N,NDTreq(AK)))}
g36: <9,8,10,OPEN, E,E,E,E,E,E, V33>{}
g37: <9,1,10,OPEN, E,
      NDTind(AK("2","0")),E,E,E,E, V33>{}
g38: <10,1,10,OPEN,E,
      NDTind(AK("2","0")),E,E,E,E, V33>{}
g39: <11,1,10,OPEN,E,
      NDTind(AK("2","0")),E,E,E,E, V33>{}
g40: <12,1,10,OPEN, E,
      NDTind(AK("2","0")),E,E,E,E, V40>
      {AT(Treceive(L,NDTind(AK))),(Verdict = Fail)}
g41: <13,1,10,OPEN, E,E,E,E,E,E, V41>
      {AFTER(Treceive(L,NDTind(AK)))}
g42: <9,1,10,OPEN, E,E,E,E,E,E, V41>{}
g43: <14,1,1',OPEN, E,E,E,E,E,E, V41>
      {AT(Tsend(L,NDTreq(DR)))}
g44: <15,1,10,OPEN,
      NDTreq(DR("normal_disconnect","Null")),
      E,E,E,E,E,V41>
      {AFTER(Tsend(L,NDTreq(DR)))}
g45: <16,1,10,OPEN,
      NDTreq(DR("normal_disconnect","Null")),
      E,E,E,E,E, V41>{}
g46: <17,1,10,OPEN,
      NDTreq(DR("normal_disconnect","Null")),
      E,E,E,E,E, V46>
      {(Verdict = Fail)}
g47: <16,4,10,OPEN, E,E,E,E,E,E, V41>{}
g48: <16,1,10,OPEN, E,E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E, V41>
      {AT(Sreceive(N,NDTind(DR)))}
g49: <16,1,14,OPEN, E,E,E,E,E,E, V41>
      {AFTER(Sreceive(N,NDTind(AK))),
       AT(Ssend(U,TDISind))}
g50: <16,1,15,OPEN, E,E,E,E,
      TDISind("normal_disconnect","Null"),E, V41>
      {AFTER(Ssend(U,TDISind)),
       AT(Ssend(N,NDTreq(DC)))}
g51: <16,1,1,OPEN, E,E,E,NDTreq(DC),
```

```
        TDISind("normal_disconnect","Null"),E, V41>
       {AFTER(Ssend(N,NDTreq(DC))),
        AT(Treceive(U,TDISind))}
g52: <16,1,1,IDLE_M3, E,E,E,NDTreq(DC),E,E, V41>
       {AFTER(Treceive(U,TDISind))}
g53: <16,1,1,IDLE, E,E,E,NDTreq(DC),E,E, V41>{}
g54: <16,8,1,IDLE, E,E,E,E,E,E, V41>{}
g55: <16,1,1,IDLE, E,NDTind(DC),E,E,E,E, V41>
       {AT(Treceive(L,NDTind(DC)))}
g56: <18,1,1,IDLE, E,E,E,E,E,E, V41>
       {AFTER(Treceive(L,NDTind(DC)))}
g57: <19,1,1,IDLE, E,E,E,E,E,E, V41>
       {AT(Tsend(L,NDTreq(CR)))}
g58: <20,1,1,IDLE,
        NDTreq(CR("you","me","No","1","any_data")),
        E,E,E,E,E, V41>
       {AFTER(Tsend(L,NDTreq(CR)))}
g59: <21,1,1,IDLE,
       NDTreq(CR("you","me","No","1","any_data")),
       E,E,E,E,E, V41>{}
g60: <22,1,1,IDLE,
       NDTreq(CR("you","me","No","1","any_data")),
       E,E,E,E,E, V60>
       {(Verdict = Fail)}
g61: <21,4,1,IDLE, E,E,E,E,E,E, V41>{}
g62: <21,1,1,IDLE, E,E,
       NDTreq(CR("you","me","No","1","any_data")),
       E,E,E, V41>
       {AT(Sreceive(N,NDTind(CR)))}
g63: <21,1,5,IDLE, E,E,E,E,E,E, V63>
       {AFTER(Sreceive(N,NDTind(CR))),
        AT(Ssend(U,TCONind))}
g64: <21,1,6,IDLE, E,E,E,E,
       TCONind("you","me","No","1","any_data"),E, V63>
       {AFTER(Ssend(U,TCONind)),
        AT(Treceive(U,TCONind))}
g65: <21,1,6,IUT_WFTRESP_M1, E,E,E,E,E,E, V63>
       {AFTER(Treceive(U,TCONind)),
        AT(Tsend(U,TCONresp))}
g66: <21,1,6,OPEN, E,E,E,E,E,
       TCONresp("1","you","No","test_data"), V63>
       {AFTER(Tsend(U,TCONresp)),
```

```
     AT(Sreceive(U,TCONresp))}
g67: <21,1,7,OPEN, E,E,E,E,E,E, V63>
     {AFTER(Sreceive(U,TCONresp)),
      AT(Ssend(N,NDTreq(CC)))}
g68: <21,1,10,OPEN, E,E,E,
     NDTreq(CC("you","1","No",0,"test_data")),
     E,E, V63>
     {AFTER(Ssend(N,NDTreq(CC)))}
g69: <21,8,10,OPEN, E,E,E,E,E,E, V63>{}
g70: <21,1,10,OPEN, E,
     NDTind(CC("you","1","No","0","test_data")),
     E,E,E,E, V63>
     {AT(Treceive(L,NDTind(CC)))}
g71: <23,1,10,OPEN, E,E,E,E,E,E, V71>
     {AFTER(Treceive(L,NDTind(CC))),
      (NDTind.CC.User_data = VAL),(Verdict = Pass)}
g72: <24,1,10,OPEN, E,E,E,E,E,E, V71>
     {AT(Tsend(L,NDTreq(DR)))}
g73: <25,1,10,OPEN,
      NDTreq(DR("normal_disconnect","Null")),
      E,E,E,E,E, V71>
     {AFTER(Tsend(L,NDTreq(DR))),
      AT(Treceive(U,TDISind))}
g74: <25,4,10,OPEN, E,E,E,E,E,E, V71>
     {AFTER(Treceive(U,TDISind))}
g75: <25,1,10,OPEN, E,E,
     NDTind(DR("normal_disconnect","Null")),
     E,E,E, V71>
     {AT(Sreceive(N,NDTind(DR)))}
g76: <25,1,14,OPEN, E,E,E,E,E,E, V71>
     {AFTER(Sreceive(N,NDTind(DR)))}
g77: <26,1,14,OPEN, E,E,E,E,E,E, V71>{}
g78: <27,1,14,OPEN, E,E,E,E,E,E, V78>
     {AT(Ssend(U,TDISind)), (Verdict = Fail)}
g79: <26,1,15,OPEN, E,E,E,E,
     TDISind("normal_disconnect","Null"),E, V71>
     {AFTER(Ssend(U,TDISind)),
      AT(Ssend(N,NDTreq(DC)))}
g80: <26,1,1,OPEN, E,E,E,NDTreq(DC),
     TDISind("normal_disconnect","Null"),E, V71>
     {AFTER(Ssend(N,NDTreq(DC))),
      AT(Treceive(U,TDISind))}
```

```
g81: <26,1,1,IDLE_M3, E,E,E,NDTreq(DC),E,E, V71>
      {AFTER(Treceive(U,TDISind))}
g82: <26,1,1,IDLE, E,E,E,NDTreq(DC),E,E, V71>{}
g83: <26,8,1,IDLE, E,E,E,E,E,E, V71>{}
g84: <26,1,1,IDLE, E,NDTind(DC),E,E,E,E, V71>
      {AT(Tsend(L,NDTind(DC)))}
g85: <28,1,1,IDLE, E,E,E,E,E,E, V71>
      {AFTER(Tsend(L,NDTind(DC)))}
g86: <29,1,1,IDLE, E,E,E,E,E,E, V71>{}
g87: <9,1,11,OPEN, E,E,E,E,E,E, V33>
      {AT(Ssend(U,TDISind))}
g88: <9,1,12,CPEN, E,E,E,E,
      TDISind("normal_disconnect","Null"),E, V33>
      {AFTER(Ssend(U,TDISind)),
       AT(Ssend(N,NDTreq(DR)))}
g89: <9,1,9,OPEN, E,E,E,
       NDTreq(DR("normal_disconnect","Null")),
       TDISind("normal_disconnect","Null"),E, V33>
      {AFTER(Ssend(N,NDTreq(DR))),
       AT(Treceive(U,TDISind))}
g90: <9,1,9,IDLE_M3, E,E,E,
      NDTreq(DR("normal_disconnect","Null")),
      E,E, V33>
      {AFTER(Treceive(U,TDISind))}
g91: <9,1,9,IDLE, E,E,E,
      NDTreq(DR("normal_disconnect","Null")),
      E,E, V33>{}
g92: <9,8,9,IDLE, E,E,E,E,E,E, V33>{}
g93: <9,1,9,IDLE, E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E,E, V33>{(seqrecak < seqsendt)}
g94: <10,1,9,IDLE, E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E,E, V33>{}
g95: <11,1,9,IDLE, E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E,E, V33>{}
g96: <31,1,9,IDLE, E,E,E,E,E,E, V96>
     {(Verdict = Fail)}
g97: <12,1,9,IDLE, E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E,E, V97>{}
```

```
g98: <16,1,18,OPEN, E,E,
      NDTind(DR("normal_disconnect","Null")),
      E,E,E, V41>
      {AT(Ssend(N,NDTreq(AK)))}
g99: <16,1,10,OPEN, E,E,
      NDTind(DR("normal_disconnect","Null")),
      NDTreq(AK("2","0")),E,E, V41>
      {AFTER(Ssend(N,NDTreq(AK)))}
g100: <16,8,10,OPEN, E,E,
       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V41>{}
g101: <16,1,10,OPEN, E,NDTind(AK("2","0")),

       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V41>{}
g102: <17,1,10,OPEN, E,NDTind(AK("2","0")),

       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V102>{(Verdict = Fail)}
g103: <32,1,10,OPEN, E,E,
       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V103>{(Verdict = Fail)}
g104: <32,1,14,OPEN, E,E,E,E,E,E, V103>
       {(seqrecak >= seqsendt),(Verdict = Fail)}
g105: <32,1,15,OPEN, E,E,E,E,
       TDISind("normal_disconnect","Null"),E, V103>
       {AT(Tsend(L,NDTreq(DR))), (Verdict = Fail)}
g106: <32,1,1,OPEN,  E,E,E,NDTreq(DC),
       TDISind("normal_disconnect","Null"),E, V103>
       {AFTER(Tsend(L,NDTreq(DR))),
        AT(Treceive(U,TDISind)), (Verdict = Fail)}
g107: <32,1,1,IDLE_M3, E,E,E,NDTreq(DC),E,E, V103>
       {AFTER(Treceive(U,TDISind)),(Verdict = Fail)}
g108: <32,1,1,IDLE, E,E,E,NDTreq(DC),E,E, V103>
       {(Verdict = Fail)}
g109: <32,8,1,IDLE, E,E,E,E,E,E, V103>
       {(Verdict = Fail)}
g110: <32,1,1,IDLE, E,NDTind(DC),E,E,E,E, V103>
       {(Verdict = Fail)}
-------------------------------------------------

Global Transitions:
```

```
r1= <g1,g2,TMP_1>,
r2= <g2,g3,TMP_2>,
r3= <g3,g4,LT_1>
r4= <g4,g5,LT_2>,
r5= <g16,g6,LT_4>,
r6= <g5,g7,USP_9>,
r7= <g7,g8,USP_10>,
r8= <g8,g9,SPEC_1>,
r9= <g9,g10,SPEC_2>
r10=<g10,g11,TMP_3>,
r11= <g11,g12,TMP_4>,
r12= <g12,g13,SPEC_5>
r13= <g13,g14,SPEC_6>,
r14= <g14,g15,USP_11>,
r15= <g15,g16,USP_12>
r16= <g16,g17,LT_5>,
r17= <g17,g18,LT_6>,
r18= <g18,g19,LT_7>
r19= <g19,g20,LT_8>,
r20= <g20,g21,USP_9>,
r21= <g21,g22,USP_10>
r22= <g22,g23,USP_9>,
r23= <g23,g24,USP_10>,
r24= <g24,g25,SPEC_19>,
r25= <g25,g26,SPEC_20>,
r26= <g26,g27,SPEC_19>,
r27= <g27,g28,SPEC_20>
r28= <g28,g29,TMP_5>,
r29= <g29,g30,TMP_6>,
r30= <g30,g31,TMP_7>
r31= <g31,g32,TMP_5>,
r32= <g32,g33,TMP_8>,
r33= <g33,g34,SPEC_17>
r34= <g34,g35,SPEC_18>,
r35= <g35,g36,USP_11>,
r36= <g36,g37,USP_12>
r37= <g37,g38,LT_9>,
r38= <g38,g39,LT_10>,
r39= <g39,g40,LT_12>
r40= <g40,g41,LT_13>,
r41= <g41,g42,LT_14>,
r42= <g42,g43,LT_15>
r43= <g43,g44,LT_16>,
r44= <g44,g45,LT_17>,
r45= <g45,g46,LT_19>
r46= <g46,g47,USP_9>,
r47= <g47,g48,USP_10>,
r34= <g34,g35,SPEC_18>
r35= <g35,g36,USP_11>,
r36= <g36,g37,USP_12>,
r37= <g37,g38,LT_9>
r38= <g38,g39,LT_10>,
r39= <g39,g40,LT_12>,
r40= <g40,g41,LT_13>
r41= <g41,g42,LT_14>,
r42= <g42,g43,LT_15>,
r43= <g43,g44,LT_16>
r44= <g44,g45,LT_17>,
r45= <g45,g46,LT_19>,
r46= <g46,g47,USP_9>
r47= <g47,g48,USP_10>,
r48= <g48,g49,SPEC_14>,
r49= <g49,g50,SPEC_15>
r50= <g50,g51,SPEC_16>,
r51= <g51,g52,TMP_9>,
r52= <g52,g53,TMP_10>
r53= <g53,g54,USP_11>,
r54= <g54,g55,USP_12>,
r55= <g55,g56,LT_20>
r56= <g56,g57,LT_21>,
r57= <g57,g58,LT_22>,
r58= <g58,g59,LT_23>
r59= <g59,g60,LT_25>,
r60= <g59,g61,USP_9>,
r61= <g61,g62,USP_10>
r62= <g62,g63,SPEC_1>,
r63= <g63,g64,SPEC_2>,
r64= <g64,g65,TMP_3>
r65= <g65,g66,TMP_4>,
r66= <g66,g67,SPEC_5>,
r67= <g67,g68,SPEC_6>
r68= <g68,g69,USP_11>,
r69= <g69,g70,USP_12>,
r70= <g70,g71,LT_26>
```

r71= <g71,g72,LT_27>, r72= <g72,g73,LT_28>,
r73= <g73,g74,TMP_9> r74= <g74,g75,TMP_10>,
r75= <g75,g76,SPEC_14>, r76= <g76,g77,LT_29>
r77= <g77,g78,LT_31>, r78= <g78,g79,SPEC_15>,
r79= <g79,g80,SPEC_16> r80= <g80,g81,TMP_9>,
r81= <g81,g82,TMP_10>, r82= <g82, g83, USP_11>
r83= <g83,g84,USP_12>, r84= <g84,g85,LT_32>,
r85= <g85,g86,LT_33> r86= <g33,g87,SPEC_7>,
r87= <g87,g88,SPEC_8>, r88= <g88,g89,SPEC_9>
r89= <g89,g90,TMP_9>, r90= <g90,g91,TMP_10>,
r91= <g91,g92,USP_11> r92= <g92,g93,USP_12>,

r93= <g93,g94,LT_9>, r94= <g94,g95,LT_10>
r95= <g95,g96,LT_11>, r96= <g95,g97,LT_12>,
r97= <g47,g98,SPEC_17> r98= <g98,g99,SPEC_18>,
r99=<g99,g100,USP_11>, r100=<g100,g101,USP_12>
r101= <g101,g102,LT_19>, r102=<g101,g103,LT_18>,
r103=<g103,g104,LT_14>, r104= <g104,g105,LT_15>,
r105=<g105,g106,LT_16>, r106= <g106,g107,TMP_9>
r107= <g107,g108,TMP_10>, r108=<g108,g109,USP_11>,
r109=<g109,g110,USP_11>

Appendix 4

Global state space of the multiple connection CS Test Verification System in Chapter 7.

```
Global states
-------------------------------------------------------
Abbreviation:
/* Vc is a set of variables taking constant values. */
Vc = {Called_addr:="you",Calling_addr:="me",
      Exp_option:="No",Qos:="1",User_data0:="",
      TS1:="",TS5:="1",TS6:="you",TS7:="No",
      TS8:="",TS9:="me",TS10:="you",TS11:="1",
      TS12:="",TS13:="",M1:=A4,M3:=A0,M9:=A9}
/* Var is attached to each state. */
Var= {(Vc,seqrecak,seqsendt),(opt,PRSeq,PRcredit),
      (M1,M3,M9), Verdict}
/* V1 is Var with initial values. */
V1= {(Vc,seqrecak:=0,seqsendt:=0,TRcredit:=1,
      TScredit:=0),(opt:=Null,PRSeq:=0,PRcredit:=5,
      PSSeq:=0,PScredit:=0),Verdict:=Null}
V9 = {(Vc,seqrecak:=0,seqsendt:=0,TRcredit:=1,
       TScredit:=0),(opt:=Null,PRSeq:=0,PRcredit:=5,
       PSSeq:=0,PScredit:=1),Verdict:=Null}
V17 = {(Vc,seqrecak:=0,seqsendt:=0,TRcredit:=1,
        TScredit:=5),(opt:=Null,PRSeq:=0,PRcredit:=5,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V19 after the 1 DT sent by LT */
 V19 = {(Vc,seqrecak:=0,seqsendt:=1,TRcredit:=1,
         TScredit:=4),(opt:=Null,PRSeq:=0,PRcredit:=5
        ,PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V22 after the 1 DT received by SPEC */
V22 = {(Vc,seqrecak:=0,seqsendt:=0,TRcredit:=1,
        TScredit:=4),(opt:=Null,PRSeq:=1,PRcredit:=4,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V27 after the 2 DT sent by LT */
V27 = {(Vc,seqrecak:=0,seqsendt:=2,TRcredit:=1,
        TScredit:=3),(opt:=Null,PRSeq:=1,PRcredit:=4,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V30 after the 2 DT received by SPEC */

V30 = {(Vc,seqrecak:=0,seqsendt:=2,TRcredit:=1,
        TScredit:=3),(opt:=Null,PRSeq:=2,PRcredit:=3,
```

```
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V34 after the 3 DT sent by LT */
V34 = {(Vc,seqrecak:=0,seqsendt:=3,TRcredit:=1,
        TScredit:=2),(opt:=Null,PRSeq:=2,PRcredit:=3,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V37 after the 3 DT received by SPEC */
V37 = {(Vc,seqrecak:=0,seqsendt:=3,TRcredit:=1,
        TScredit:=2),(opt:=Null,PRSeq:=3,PRcredit:=2,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V41 after the 4 DT sent by LT */
V41 = {(Vc,seqrecak:=0,seqsendt:=4,TRcredit:=1,
        TScredit:=1),(opt:=Null,PRSeq:=3,PRcredit:=2,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V44 after the 4 DT received by SPEC */
V44 = {(Vc,seqrecak:=0,seqsendt:=4,TRcredit:=1,
        TScredit:=1),(opt:=Null,PRSeq:=4,PRcredit:=1,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V48 after the 5 DT sent by LT */
V48 = {(Vc,seqrecak:=0,seqsendt:=5,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=4,PRcredit:=1,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V51 after the 5 DT received by SPEC */
V51 = {(Vc,seqrecak:=0,seqsendt:=5,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V62 after the LT receives an AK. */
V62 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=0,PScredit:=1),Verdict:=Null}
/* V67 after the SPEC sends a DT. */
V67 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=0,PScredit:=0),Verdict:=Null}
/* V70 after the LT receives  DT. */
V70 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=0,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=1,PScredit:=0),
Verdict:=Null}
V76 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=0,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=1,PScredit:=0),
Verdict:=Pass}
```

```
V90 = {(Vc,seqrecak:=0,seqsendt:=0,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=0,PRcredit:=5,
        PSSeq:=0,PScredit:=1),Verdict:=Fail}
V91 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=1,
        TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
        PSSeq:=0,PScredit:=1),Verdict:=Fail}
V102 = {(Vc,seqrecak:=0,seqsendt:=5,TRcredit:=1,
         TScredit:=0),(opt:=Null,PRSeq:=4,PRcredit:=1,
         PSSeq:=0,PScredit:=1),Verdict:=Fail}
V104 = {(Vc,seqrecak:=5,seqsendt:=5,TRcredit:=0,
         TScredit:=0),(opt:=Null,PRSeq:=5,PRcredit:=0,
         PSSeq:=1,PScredit:=0),Verdict:=Pass}
-----------------------------------------------------
Verdict attachments:
Fail: g6,g90,g102,g103,g61,g108,g116
Pass: g89 (starts in g76)
-----------------------------------------------------

g1:  <2,1,1,IDLE, E,E,E,E,E,E, V1>{INIT}
g2:  <2,1,1,IDLE_M9, E,E,E,E,E,E, V1>{}
g3:  <2,1,1,IDLE, E,E,E,E,E,E, V1>
{AT(Tsend(L, NDTreq(CR)))}
g4:  <3,1,1,IDLE, NDTreq(CR("you","me","No","1","")),
      E,E,E,E,E, V1>
     {AFTER(Tsend(L, NDTreq(CR)))}
g5:  <4,1,1,IDLE, NDTreq(CR("you","me","No","1","")),
      E,E,E,E,E, V1>{}
g6:  <5,1,1,IDLE, NDTreq(CR("you","me","No","1","")),
      E,E,E,E,E, V1>
{(Verdict = Fail)}
g7:  <4,4,1,IDLE, E,E,E,E,E,E, V1>{}
g8:  <4,1,1,IDLE, E,E,
      NDTind(CR("you","me","No","1","")),E,E,E, V1>
      {AT(Sreceive(N,NDTind(CR)))}
g9:  <4,1,5,IDLE, E,E,E,E,E,E, V9>
      {AFTER(Sreceive(N,NDTind(CR))),
       AT(Ssend(U,TCONind))}
g10: <4,1,6,IDLE, E,E,E,E,
      TCONind("you","me","No","1",""),E, V9>
      {AFTER(Ssend(U,TCONind)),
       AT(Treceive(U,TCONind))}
g11: <4,1,6,IUT_WFTRESP_M1, E,E,E,E,E,E, V9>
{AFTER(Treceive(U,TCONind)),AT(Tsend(U,TCONresp))}
```

```
g12: <4,1,6,OPEN, E,E,E,E,E,
      TCONresp("1","you","No",""), V9>
      {AFTER(Tsend(U,TCONresp)),
       AT(Sreceive(U,TCONresp))}
g13: <4,1,7,OPEN, E,E,E,E,E,E, V9>
      {AFTER(Sreceive(U,TCONresp)),
       AT(Ssend(N,NDTreq(CC)))}
g14: <4,1,10,OPEN, E,E,E,
      NDTreq(CC("me", "1","No","2","")),E,E, V9>
      {AFTER(Ssend(N,NDTreq(CC)))}
----------------------------------------------------
g15: <4,8,10,OPEN, E,E,E,E,E,E, V9>{}
g16: <4,1,10,OPEN, E,
      NDTind(CC("me", "1","No","2","")),E,E,E,E, V9>
      {AT(Treceive(L,NDTind(CC)))}
g17: <6,1,10,OPEN, E,E,E,E,E,E, V17>
      {AFTER(Treceive(L,NDTind(CC)))}
g18: <7,1,10,OPEN, E,E,E,E,E,E, V17>
      {AT(Tsend(L,NDTreq(DT)))}
g19: <8,1,10,OPEN,
      NDTreq(DT("TMP1(M1=A4,M2=A0,.,M10=A0,
      M11=A5,M12=A5,
      M13=A0,.,M25=A0)","True"))),E,E,E,E,E, V19>
      {AFTER(Tsend(L,NDTreq(DT)))}
g20: <8,4,10,OPEN,E,E,E,E,E,E, V19>{}
g21: <8,1,10,OPEN,E,E,
      NDTind(DT(DT("TMP1(M1=A4,M2=A0,.,M10=A0,
      M11=A5,M12=A5,
      M13=A0,.,M25=A0)","True")),E,E,E, V20>
      {AT(Sreceive(L,NDTind(DT)))}
g22: <8,1,17,OPEN,E,E,E,E,E,E, V22>
      {AFTER(Sreceive(L,NDTind(DT))),
       AT(Ssend(U,TDATAind))}
g23: <8,1,10,OPEN,E,E,E,E,
      TDTind("TMP1(M1=A4,M2=A0,.,M10=A0,
      M11=A5,M12=A5,M13=A0,.,M25=A0)") ,E, V22>
      {AFTER(Ssend(U,TDATAind)),
       AT(Treceive(U,TDATAind))}
g24: <8,1,10,OPEN_1,E,E,E,E,E,E, V22>
      {A"TER(Treceive(U,TDATAind))}
g25: <8,1,10,OPEN_M9,E,E,E,E,E,E, V22>{}
g26: <8,1,10,OPEN,E,E,E,E,E,E, V22>
```

```
      {AT(Tsend(L,NDTreq(DT)))}
g27: <15,1,10,OPEN,
      NDTreq(DT("TMP3(TS1)","True"))),E,E,E,E,E, V27>
      {AFTER(Tsend(L,NDTreq(DT)))}
g28: <15,4,10,OPEN,E,E,E,E,E,E, V27>{}
g29: <15,1,10,OPEN,E,E,
      NDTind(DT("TMP3(TS1)","True"))),E,E,E, V27>
      {AT(Sreceive(L,NDTind(DT)))}
g30: <15,1,17,OPEN,E,E,E,E,E,E, V30>
      {AFTER(Sreceive(L,NDTind(DT))),
       AT(Ssend(U,TDTind))}
g31: <15,1,10,OPEN,E,E,E,E,
      TDTind("TMP3(S1)") ,E, V30>
      {AFTER(Ssend(U,TDTind)),
       AT(Treceive(U,TDTind))}
g32: <15,1,10,OPEN_1,E,E,E,E,E,E, V30>
      {AFTER(Treceive(U,TDTind))}
g33: <15,1,10,OPEN,E,E,E,E,E,E, V30>
      {AT(Tsend(L,TDTreq(DT)))}
g34: <16,1,10,OPEN,
      NDTreq(DT("TMP4("1","you","No","test_data"))),
      E,E,E,E,E, V34>
      {AFTER(Tsend(L,NDTreq(DT)))}
g35: <16,4,10,OPEN,E,E,E,E,E,E, V34>{}
g36: <16,1,10,OPEN,E,E,
      NDTind(DT("TMP4("1","you","No","test_data"))),
      E,E,E, V34>
     {AT(Sreceive(L,NDTind(DT)))}
g37: <16,1,17,OPEN,E,E,E,E,E,E, V37>
      {AFTER(Sreceive(L,NDTind(DT))),
       AT(Ssend(U,TDTind))}
g38: <16,1,10,OPEN,E,E,E,E,
      TDTind("TMP4("1","you","No","test_data")"),E,V37>
      {AFTER(Ssend(U,TDTind)),AT(Treceive(U,TDTind))}
g39: <16,1,10,OPEN_1,E,E,E,E,E,E, V37>
      {AFTER(Treceive(U,TDTind))}
---------------------------------------------------
g40: <16,1,10,OPEN,E,E,E,E,E,E, V37>
{AT(Tsend(L,NDTreq(DT)))}
g41: <17,1,10,OPEN,
      NDTreq(DT("TMP5()")), E, E,E,E,E, V41>
      {AFTER(Tsend(L,NDTreq(DT)))}
```

```
g42: <17,4,10,OPEN,E,E,E,E,E,E, V41>{}
g43: <17,1,10,OPEN,E,E,
      NDTind(DT("TMP5()")), E, E,E, V41>
      {AT(Sreceive(L,NDTind(DT)))}
g44: <17,1,17,OPEN,E,E,E,E,E,E, V44>
     {AFTER(Sreceive(L,NDTreq(DT))),
      AT(Ssend(U,TDTind))}
g45: <17,1,10,OPEN,E,E,E,E,TDTind("TMP5()"),E, V44>
      {AFTER(Ssend(U,TDTind)),AT(Treceive(U,TDTind))}
g46: <17,1,10,OPEN_1,E,E,E,E,E,E, V44>
      {AFTER(Treceive(U,TDTind))}
g47: <17,1,10,OPEN,E,E,E,E,E,E, V44>
      {AT(Tsend(L,NDTreq(DT)))}
g48: <9,1,10,OPEN,
      NDTreq(DT("TMP8()")), E, E,E,E,E, V48>
      {AFTER(Tsend(L,NDTreq(DT))),INTERNAL}
g49: <9,4,10,OPEN,E,E,E,E,E,E, V48>{}
g50: <9,1,10,OPEN,E,E,
      NDTind(DT("TMP8()")), E,E,E, V48>
      {AT(Sreceive(L,NDTind(DT)))}
g51: <9,1,17,OPEN,E,E,E,E,E,E, V51>
      {AFTER(Sreceive(L,NDTind(DT))),
       AT(Ssend(U,TDTind))}
g52: <9,1,10,OPEN,E,E,E,E,TDTind("TMP8()"),E, V51>
      {AFTER(Ssend(U,TDTind)),AT(Treceive(U,TDTind))}
g53: <9,1,10,OPEN_1,E,E,E,E,E,E, V51>
      {AFTER(Treceive(U,TDTind)),AT(Tsend(U,TDTreq))}
g54: <9,1,10,OPEN,E,E,E,E,E,TDTreq("TMP8r()", V51>
      {AFTER(Tsend(U,TDTreq)),INTERNAL}
g55: <9,1,18,OPEN, E,E,E,E,E,TDTreq("TMP8r()"), V51>
      {AT(Ssend(N,NDTreq(AK)))}

g56: <9,1,10,OPEN,E,E,E,NDTreq(AK("2","0")),
      E,TDTreq("TMP8r()"),V51>
     {AFTER(Ssend(N,NDTreq(AK)))}
g57: <9,8,10,OPEN,E,E,E,E,E,TDTreq("TMP8r()"),V51>{}
g58: <9,1,10,OPEN, E,NDTind(AK("2","0")),E,E,E,
      TDTreq("TMP8r()"), V51>{}
g59: <10,1,10,OPEN, E,NDTind(AK("2","0")),E,E,E,
      TDTreq("TMP8r()"), V51>{}
g60: <11,1,10,OPEN, E,NDTind(AK("2","0")),E,E,E,
      TDTreq("TMP8r()"), V51>
```

```
{AT(Treceive(L,NDTind(AK)))}
/* g59 is from g58 */
g61: <12,1,10,OPEN, E,NDTind(AK("2","0")),E,E,E,
      TDTreq("TMP8r()"), V61>
      {AT(Treceive(L,NDTind(AK))),(Verdict = Fail)}
/* g60 is from g58 */

g62: <13,1,10,OPEN, E,E,E,E,E,TDTreq("TMP8r()"), V62>
      {AFTER(Treceive(L,NDTind(AK)))}
g63: <9,1,10,OPEN,E,E,E,E,E,TDTreq("TMP8r()"),V62>{}
g64: <14,1,10,OPEN, E,E,E,E,E,TDTreq("TMP8r()"), V62>
      {seqrecak >=seqsendt}
g65: <18,1,10,OPEN,E,E,E,E,E,TDTreq("TMP8r()"),V62>
      {AT(Sreceive(U,TDTreq))}
g66: <18,1,16,OPEN, E,E,E,E,E,E, V62>
      {AFTER(Sreceive(U,TDTreq)),
       AT(Ssend(N,NDTreq(DT)))}
g67: <18,1,23,OPEN, E,E,E,
      NDTreq(DT("TMP8r()",EOT),E,E, V67>
     {AFTER(Ssend(N,NDTreq(DT))),INTERNAL}
g92:<18,1,10,OPEN, E,E,E,
     NDTreq(DT("TMP8r()",EOT),E,E, V67>{}
g68: <18,8,10,OPEN, E,E,E,E,E, V67>{}
g69: <18,1,10,OPEN, E,
       NDTind("TMP8r()"),E,E,E, V67>
      {AT(Treceive(L,NDTind(DT)))}
g70: <19,1,10,OPEN, E,E,E,E,E, V70>
      {AFTER(Treceive(L,NDTind(DT))),
       AT(Tsend(L,NDTreq(AK)))}
g71: <22,1,10,OPEN, NDTreq(AK()),E,E,E,E,E, V70>
      {AFTER(Tsend(L,NDTreq(AK)))}
g72: <22,4,10,OPEN, E,E,E,E,E,E, V70>{}
g73: <22,1,10,OPEN, E,E,NDTind(AK()),E,E,E, V70>
      {AT(Sreceive(NDTind(AK)))}
g74: <22,1,22,OPEN, E,E,E,E,E,E, V70>
      {AFTER(Sreceive(NDTind(AK)))}
g75: <22,1,10,OPEN, E,E,E,E,E,E, V70>{}
g76: <23,1,10,OPEN, E,E,E,E,E,E, V76>
      {AT(Tsend(L,NDTreq(DR)))}
g77: <24,1,10,OPEN, NDTreq(DR()),E,E,E,E,E, V76>
      {AFTER(Tsend(L,NDTreq(DR)))}
g78: <25,1,10,OPEN, NDTreq(DR()),E,E,E,E,E, V76>{}
```

```
g79: <25,8,10,OPEN, E,E,E,E,E,E, V76>{}
g80: <25,1,10,OPEN, E,E,NDTind(DR()),E,E,E, V76>
      {AT(Sreceive(NDTind(DR)))}
g81: <25,1,14,OPEN, E,E,E,E,E,E, V76>
      {AFTER(Sreceive(NDTind(DR))),
       AT(Ssend(U,TDISind))}
g82: <25,1,15,OPEN, E,E,E,E,TDISind(),E, V76>
      {AFTER(Ssend(U,TDISind)),
       AT(Ssend(N,NDTreq(DC)))}
g83: <25,1,1,OPEN,E,E,E,NDTreq(DC()),TDISind(),E,V76>
      {AFTER(Ssend(N,NDTreq(DC))),
       AT(Treceive(U,TDISind))}
g84: <25,1,1,IDLE_M3, E,E,E,NDTreq(DC()),E,E, V76>
      {AFTER(Treceive(U,TDISind))}
g85: <25,1,1,IDLE, E,E,E,NDTreq(DC()),E,E, V76>{}
g86: <25,8,1,IDLE, E,E,E,E,E,E, V76>{}
g87: <25,1,1,IDLE, E,NDTind((DC)),E,E,E,E, V76>
      {AT(Treceive(L,NDTind(DC)))}
g88: <28,1,1,IDLE, E,E,E,E,E,E, V76>
      {AFTER(Treceive(L,NDTind(DC)))}
g89: <29,1,1,IDLE, E,E,E,E,E,E, V76>{}
-------------------------------------------------------
/* reached from state g15 */
g90: <5,8,10,OPEN, E,E,E,E,E,E, V90>{(verdict=Fail)}
/* reached from state g66 */
g91: <21,8,10,OPEN, E,E,E,E,E, V91>{(verdict=Fail)}
-------------------------------------------------------
/* from g48 */
g93: <9,1,11,OPEN,NDTreq(DT("TMP8r()")),
      E,E,E,E,E,V48>
      {AT(Ssend(U,TDISind))}
g94: <9,1,12,OPEN, NDTreq(DT("TMP8r()")),E,E,E,
      TDISind("normal_disconnect","Null"),E, V48>
      {AFTER(Ssend(U,TDISind)),
       AT(Ssend(N,NDTreq(DR)))}
g95: <9,1,9,OPEN, NDTreq(DT("TMP8r()")),E,E,
      NDTreq(DR("normal_disconnect","Null")),
      TDISind("normal_disconnect","Null"),E, V48>
      {AFTER(Ssend(N,NDTreq(DR))),
       AT(Treceive(U,TDISind))}
g96: <9,1,9,IDLE_M3, NDTreq(DT("TMP8r()")),E,E,
      NDTreq(DR("normal_disconnect","Null")),E,E,V48>
```

```
      {AFTER(Treceive(U,TDISind))}
g97: <9,1,9,IDLE, NDTreq(DT("TMP8r()")),E,E,
     NDTreq(DR("normal_disconnect","Null")),E,E, V48>
     {}
g98: <9,8,9,IDLE,NDTreq(DT("TMP8r()")),E,E,E,E,E,V48>
     {}
g99: <9,1,9,IDLE, NDTreq(DT("TMP8r()")),
      NDTind(DR("normal_disconnect","Null")),
      E,E,E,E, V48>{*}*******
g100: <10,1,9,IDLE, NDTreq(DT("TMP8r()")),
       NDTind(DR("normal_disconnect","Null")),
E,E,E,E, V48>
      {(seqrecak < seqsendt)}
g101: <11,1,9,IDLE, NDTreq(DT("TMP8r()")),
       NDTind(DR("normal_disconnect","Null")),
E,E,E,E, V48>{}
g102: <31,1,9,IDLE,
      NDTreq(DT("TMP8r()")),E,E,E,E,E, V102>
      {(Verdict = Fail)}
g103: <12,1,9,IDLE, NDTreq(DT("TMP8r()")),
       NDTind(DR("normal_disconnect","Null")),
       E,E,E,E, V102>
      {(Verdict=Fail)}
----------------------------------------------------
/* from g80 */
g104: <25,1,18,OPEN,E,E,
       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V76> {AT(Ssend(N,NDTreq(AK)))}
g105: <25,1,10,OPEN,E,E,
       NDTind(DR("normal_disconnect","Null")),
       NDTreq(AK("2","0")),E,E, V76>{}
g106: <25,8,10,OPEN, E,E,
       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V76>{}
g107: <25,1,10,OPEN,E,NDTind(AK("2","0")),

       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V76>{}
g108: <27,1,10,OPEN, E,NDTind(AK("2","0")),

       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V108>{(Verdict = Fail)}
```

```
g109: <26,1,10,OPEN, E,E,
       NDTind(DR("normal_disconnect","Null")),
       E,E,E, V108> {(Verdict = Fail)}
g110: <26,1,14,OPEN, E,E,E,E,E,E, V108>
       {(seqrecak >= seqsendt),(Verdict = Fail)}
g111: <26,1,15,OPEN, E,E,E,E,
       TDISind("normal_disconnect","Null"),E, V108>
       {AT(Tsend(L,NDTreq(DR))), (Verdict = Fail)}
g112: <26,1,1,OPEN,  E,E,E,NDTreq(DC),
       TDISind("normal_disconnect","Null"),E, V108>
       {AFTER(Tsend(L,NDTreq(DR))),
        AT(Treceive(U,TDISind)),(Verdict = Fail)}
g113: <26,1,1,IDLE_M3, E,E,E,NDTreq(DC),E,E, V108>
       {AFTER(Treceive(U,TDISind)), (Verdict = Fail)}
g114: <26,1,1,IDLE, E,E,E,NDTreq(DC),E,E, V108>
       {(Verdict = Fail)}
g115: <26,8,1,IDLE, E,E,E,E,E,E, V108>
       {(Verdict = Fail)}
g116: <26,1,1,IDLE, E,NDTind(DC),E,E,E,E, V108>
       {(Verdict = Fail)}
-----------------------------------------------------
```

```
Global Transitions
------------------------------------------------------
r1:=  <g1, g2, TMP_1>,       r2:=  <g2, g3, TMP_2>,
r3:=  <g3, g4, LT_1>         r4:=  <g4, g5, LT_2>,
r5:=  <g5, g6, LT_4>,        r6:=  <g5, g7, USP_9>
r7:=  <g7, g8, USP_10>,      r8:=  <g8, g9, SPEC_1>,
r9:=  <g9, g10, SPEC_2>      r10:= <g10, g11, TMP_3>,
r11:= <g11, g12, TMP_4>,     r12:= <g12, g13, SPEC_5>
r13:= <g13, g14, SPEC_6>,    r14:= <g14, g15, USP_11>,
r15:= <g15, g16, USP_12>     r16:= <g16, g17, LT_5>,
r17:= <g17, g18, LT_6>,      r18:= <g18, g19, LT_7>
r19:= <g19, g20, USP_9>,     r20:= <g20, g21, USP_10>,
r21:= <g21, g22, SPEC_19>    r22:= <g22, g23, SPEC_20>,
r23:= <g23, g24, TMP_5>,     r24:= <g24, g25, TMP_6>
------------------------------------------------------
r25:= <g25, g26, TMP_7>,     r26:= <g26, g27, LT_8>,
r27:= <g27, g28, USP_9>      r28:= <g28, g29, USP_10>,
r29:= <g29, g30, SPEC_19>,   r30:= <g30, g31, SPEC_20>
r31:= <g31, g32, TMP_5>,     r32:= <g32, g33, TMP_11>,
r33:= <g33, g34, LT_16>
```

```
-----------------------------------------------------------
r34:= <g34, g35, USP_9>,     r35:= <g35, g36, USP_10>,
r36:= <g36, g37, SPEC_19>    r37:= <g37, g38, SPEC_20>
r38:= <g38, g39, TMP_5>,     r39:= <g39, g40, TMP_8>
r40:= <g40, g41, LT_17>,     r41:= <g41, g42, USP_9>,
r42:= <g42, g43, USP_10>     r43:= <g43, g44, SPEC_19>,
r44:= <g44, g45, SPEC_20>,   r45:= <g45, g46, TMP_5>
r46:= <g46, g47, TMP_12>,    r47:= <g47, g48, LT_18>,
r48:= <g48, g49, USP_9>      r49:= <g49, g50, USP_10>,
r50:= <g50, g51, SPEC_19>,   r51:= <g51, g52, SPEC_20>
r52:= <g52, g53, TMP_5>,     r53:= <g53, g54, TMP_13>,
r54:= <g54, g55, SPEC_17>    r55:= <g55, g56, SPEC_18>

r56:= <g56, g57, USP_11>     r57:= <g57, g58, USP_12>
r58:= <g58, g59, LT_9>,      r59:= <g59, g60, LT_10>,
r60:= <g60, g61, LT_12>      r61:= <g60, g62, LT_13>,
r62:= <g62, g63, LT_14>,     r63:= <g63, g64, LT_15>
r64:= <g64, g65, LT_19>,     r65:= <g65, g66, SPEC_31>,
r66:= <g66, g67, SPEC_32>    r67:= <g67, g92, SPEC_33>

r91:= <g92, g68, USP_11>,    r68:= <g68, g69, USP_12>
r69:= <g69, g70, LT_22>,     r70:= <g70, g71, LT_23>,
r71:= <g71, g72, USP_9>
-----------------------------------------------------------
r72:= <g72, g73, USP_10>,    r73:= <g73, g74, SPEC_28>,
r74:= <g74, g75, SPEC_30>    r75:= <g75, g76, LT_24>,
r76:= <g76, g77, LT_25>,     r77:= <g77, g78, LT_26>
r78:= <g78, g79, USP_11>,    r79:= <g79, g80, USP_12>,
r80:= <g80, g81, SPEC_14>    r81:= <g81, g82, SPEC_15>
r82:= <g82, g83, SPEC_16>,   r83:= <g83, g84, TMP_9>
r84:= <g84, g85, TMP_10>,    r85:= <g85, g86, USP_11>,
r86:= <g86, g87, USP_12>     r87:= <g87, g88, LT_29>,
r88:= <g88, g89, LT_30>,     r89:= <g15, g90, LT_4>
r90:= <g66, g91, LT_21>,     r92:= <g48, g93, SPEC_7>,
r93:= <g93, g94, SPEC_8>     r94:= <g94, g95, SPEC_9>,
r95:= <g95, g96, TMP_9>,     r96:= <g96, g97, TMP_10>
r97:= <g97, g98, USP_11>,    r98:= <g98, g99, USP_12>,
r99:= <g99, g100, LT_9>      r100:= <g100, g101, LT_10>
r101:= <g101, g102, LT_11>, r102:= <g101, g103, LT_12>
r103:= <g80, g104, SPEC_17>,r104:= <g104, g105, SPEC_18>,
r105:= <g105, g106, USP_11>,r106:= <g106, g107, USP_12>,
r107:= <g107, g108, LT_28>, r108:= <g107, g109, LT_27>
```

```
r109:= <g109,g110,SPEC_14>, r110:= <g110,g111,SPEC_15>,
r111:= <g111,g112,SPEC_16>, r112:= <g112,g113,TMP_9>
r113:= <g113,g114,TMP_10>,  r114:= <g114,g115,USP_11>
r115:= <g115, g116, USP_12>
----------------------------------------------------------
End
----------------------------------------------------------
```