SLICING APPROACH TO SPECIFICATION FOR TESTABILITY IN LOTOS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ahmet Feyzi Ateş
August, 1993

SLICING APPROACH TO SPECIFICATION FOR TESTABILITY IN LOTOS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

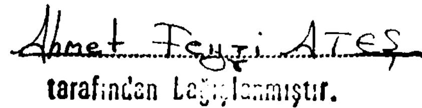
FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ahmet Feyzi ATEŞ

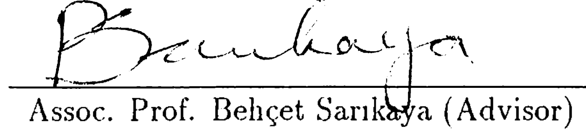
August, 1993



QA
76.73
.L65
A84
1993

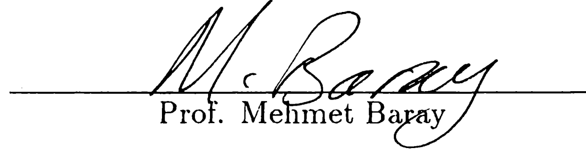
B014104

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



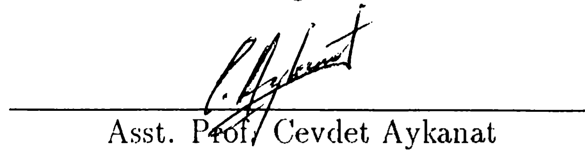
Assoc. Prof. Behçet Sarıkaya (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



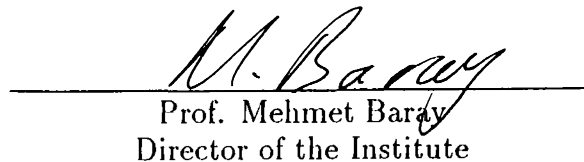
Prof. Mehmet Baray

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Asst. Prof. Cevdet Aykanat

Approved for the Institute of Engineering and Science:



Prof. Mehmet Baray
Director of the Institute

ABSTRACT

SLICING APPROACH TO SPECIFICATION FOR TESTABILITY IN LOTOS

Ahmet Feyzi ATEŞ
M.S. in Computer Engineering and Information Science
Advisor: Assoc. Prof. Behçet Sarıkaya
August, 1993

With the recent increase in the use of formal methods in specification of communication protocols, there is a need to base the conformance testing of protocol implementations on formal specifications. This brings in the problem of finding out special design issues to be used in the specification of such systems that facilitate test generation. This aspect is called *Specification For Testability*, and it is investigated in this study for the particular formal description technique LOTOS. Specification for testability is approached from the perspective of designing formal base protocol specifications, and then deriving functional specifications from base specifications in order to use in test generation. The method utilized for the derivation of functional specifications is called *slicing*. As inspired from previous work in software engineering, slices of protocol specifications are obtained systematically according to the hierarchically designed test suite structures, where each slice corresponds to a particular function of the protocol, and subsequent test generation is based on the obtained slices. The techniques developed are demonstrated on the simple state-oriented specifications of INRES and ACSE protocols along with a real base specification of the OSI Transport Protocol written in the constraint-oriented specification style. The results indicate that tests derived from functional specifications have some remarkable properties with respect to test case analysis and representation.

Keywords: Conformance Testing, Specification For Testability, LOTOS, Slicing.

ÖZET

LOTOS'DA TEST EDİLEBİLİRLİK İÇİN BELİRTİME DİLİMLEME YAKLAŞIMI

Ahmet Feyzi ATEŞ
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Doç. Dr. Behçet Sarıkaya
Ağustos, 1993

Yakın zamanlarda iletişim protokollarının belirtiminde biçimsel metodların kullanımının artmasıyla, protokol uyarlamalarının uygunluk testlerinin de bicimsel belirtimlere dayandırılması gereği doğmuştur. Bu durum, protokol belirtimlerinde, test üretmeyi kolaylaştıracak tasarım ilkeleri bulma problemini ortaya çıkarmıştır. Bu konuya *test edilebilirlik için belirtim* adı verilmektedir, ve bu calışmada biçimsel bir tanımlama tekniği olan LOTOS için incelenmiştir. Test edilebilirlik için belirtim konusuna, temel protokol belirtimleri tasarlama ve daha sonra test üretmede kullanılmak üzere işlevsel belirtimleri temel belirtimlerden elde etme perspektifinde yaklaşılmıştır. İşlevsel belirtimleri elde etmede kullanılan yönteme *dilimleme* adı verilmektedir. Yazılım mühendisliği dalında daha önce yapılan calışmalardan esinlenerek, protokol belirtimlerinin dilimleri, hiyerarşik biçimde tasarlanan test yapılarına göre, ve her dilim belirli bir protokol işlevine karşılık gelecek şekilde sistematik olarak elde edilmiş ve daha sonraki test üretme safhası elde edilen dilimlere dayandırılmıştır. Geliştirilen teknikler, basit, sistem-durumuna yönelik INRES ve ACSE protokolları ile birlikte, kıstasa yönelik belirtim tarzında yazılmiş gerçek bir temel belirtim olan OSI Transport Protokolu uzerinde gösterilmiştir. Sonuçlar şunu göstermektedir ki, işlevsel belirtimlerden çıkarılan testler, test durum analizi ve temsili açısından bazı dikkate değer ozellikler taşımaktadır.

Anahtar Sözcükler: Uygunluk testi, Test edilebilirlik için belirtim, LOTOS, Dilimleme.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my supervisor Assoc. Prof. Behçet Sarıkaya for his guidance, suggestions, and invaluable encouragement throughout the development of this thesis. I would also like to thank to Prof. Mehmet Baray and Asst. Prof. Cevdet Aykanat for reading and commenting on the thesis. I am grateful to my friends for their infinite moral support and help. Finally I want to thank my family for their invaluable support during my whole academic life.

Contents

1 INTRODUCTION **1**

1.1 OSI Architectural Concepts 3

1.2 Service Definitions and Protocol Specifications 5

1.3 Conformance Testing 7

1.4 Overview 10

2 FORMAL METHODS IN CONFORMANCE TESTING **12**

2.1 Introduction to LOTOS 13

2.2 Labelled Transition Systems 16

2.3 Theory of Conformance 18

2.4 Implementation Relations and Conformance 20

2.5 Language Based Systematic Test Suite Derivation 22

2.5.1 Test Derivation From Finite Systems 26

2.5.2 Test Derivation with Infinite Branching 27

2.6 Chart Based Test Generation 31

2.6.1 Normalization 32

2.6.2 Identification of the Functions to be Tested 35

2.6.3 Generation and Analysis of Test Cases 36

2.6.4 Test Selection and Representation 37

2.6.5 Application 38

3 SPECIFICATION FOR TESTABILITY 39

3.1 Base Specification Development 40

3.1.1 Parameterizing Specifications 41

3.1.2 Behaviour Specification 43

3.2 Functional and Profile Specifications 44

3.2.1 Hierarchical Test Selection 44

3.2.2 Protocol Slicing According to Test Suite Structure 47

3.2.3 Slicing and Behaviour Reductions 48

3.2.4 Systematic Protocol Slicing 50

3.2.4.1 Definitions 50

3.2.4.2 The Algorithm 52

3.3 Related Work 54

4 BASE SPECIFICATION DEVELOPMENT 57

4.1 INRES Protocol 58

4.1.1 PICS Parameterization 60

4.1.2 Behaviour Specification 61

4.2 ACSE Protocol 64

4.2.1 PICS Parameterization 67

4.2.2 Behaviour Specification 69

5 DERIVATION OF FUNCTIONAL SPECIFICATIONS 72

5.1 Behaviour Reductions On State-Oriented Specifications 73

5.1.1 Behaviour Reductions On INRES 73

5.1.2 Behaviour Reductions On ACSE 79

5.2 Behaviour Reductions On Constraint-Oriented Transport Protocol Specification 83

5.2.1 Introduction To The Transport Protocol And Its Base Specification 83

5.2.2 Behaviour Reductions 91

6 TEST DESIGN USING BASE AND FUNCTIONAL SPECIFICATIONS 104

6.1 Test Generation From Base Specifications 106

6.2 Test Generation From Functional Specifications 111

6.2.1 Chart Generation 111

6.2.2 Generation Of Test Cases 113

6.2.3 Test Case Reductions 115

6.2.4 Infeasible Test Cases 116

7 CONCLUSIONS 117

8 APPENDICES 124

A INRES Base Protocol Specification In LOTOS 125

B PICS Proforma For INRES 137

B.1 Date of Statement 137

B.2 Implementation Details 137

B.3 Global Statement of Conformance 138

B.4 Initiator/Responder Capability 138

B.5 Supported PDUs 138

B.6 Supported PDU Parameters 139

B.6.1 Data Transfer IPDU (DT) 139

B.6.2 Acknowledgement IPDU (AK) 139

B.7 Timers 140

C Test Suite Structure and Test Purposes for INRES 141

C.1 Initiator (I) 141

C.1.1 I/Basic Interconnection and Capability Tests (BIC) . . . 141

C.1.2 I/Valid Behaviour Tests (BV) 142

C.1.2.1 I/BV/Connection Establishment (CE) 142

C.1.2.2 I/BV/Data Transfer (DT) 142

C.1.2.3 I/BV/Disconnection (DC) 143

C.1.3 I/Invalid & Inopportune Behaviour Tests (BIO) 143

C.1.3.1 I/BIO/Disconnected State (STA0) 144

C.1.3.2 I/BIO/WaitforCC State (STA1) 144

C.1.3.3 I/BIO/Connected State (STA2) 144

C.1.3.4 I/BIO/Sending State (STA3) 144

C.2 Responder (R) 145

C.2.1 R/Basic Interconnection and Capability Tests (BIC) . . 145

C.2.2 R/Valid Behaviour Tests (BV) 145

C.2.2.1 R/BV/Connection Establishment (CE) 146

C.2.2.2 R/BV/Data Transfer (DT) 146

C.2.2.3 R/BV/Disconnection (DC) 146

C.2.3 R/Invalid & Inopportune Behaviour Tests (BIO) 147

C.2.3.1 R/BIO/Disconnected State (STA0) 147

C.2.3.2 R/BIO/WaitforICONresp State (STA1) 147

C.2.3.3 R/BIO/Connected State (STA2) 147

D ACSE Base Protocol Specification In LOTOS 149

List of Figures

1.1 OSI Reference Model 2

1.2 Layer Concept of OSI Reference Model 3

1.3 Encapsulation of PDUs 5

1.4 Conceptual Testing Architecture 9

2.1 Action Tree of a Simple System 18

2.2 Sample Test Case 28

3.1 Behaviour Reduction Algorithm 55

5.1 Test Suite Structure for INRES 74

5.2 Test Suite Structure for ACSE 80

5.3 Test Suite Structure for Transport Protocol Class 2 93

List of Tables

2.1 Notation for Labelled Transition Systems 17

2.2 Axioms and Inference Rules for Basic LOTOS 19

2.3 Compositional Computation of Acceptance Sets 26

2.4 Compositional Computation of Acceptance Sets for Infinite Systems 29

2.5 Compositional Computation of Subsequent Behaviour for Infinite Systems 30

4.1 INRES IPDUs 58

4.2 State Table for Initiator Protocol 62

4.3 State Table for Responder Protocol 62

4.4 ACSE APDUs 65

5.1 Transport TPDUs 84

6.1 Chart Sizes Of Functional Specifications For INRES 112

6.2 Chart Sizes Of Functional Specifications For ACSE 112

6.3 Test Cases Generated From Functional Specifications Of INRES 113

6.4 Test Cases Generated From Functional Specifications Of ACSE 114

Chapter 1

INTRODUCTION

In recent years, information technology (IT) has become a major part of human civilization. The sheer variety of IT has created the significant problem of interconnecting systems to form networks. Networks make the information available wherever it is most useful, thus increasing the value of information. One of the most challenging issues in the past few years has been the interconnection of multi-vendor network products to allow applications from different networks to inter-work with each other [1].

In an *Open System Environment* (OSE), the computer systems and software of different vendors are interchangeable and can be combined into an integrated operating environment. Open system standards, i.e. non-proprietary standards play the most important role in the realization of an OSE. They provide a standardized operating infrastructure for OSE. First, architectural standards are needed to build an OSE model. Second, base standards are needed to provide the specification of the different components of the model. Finally, functional standards are needed to adapt to specific environments.

In 1977, International Organization for Standardization (ISO) established a subcommittee to develop a structure (or architecture) that defines communication tasks. The aim was to establish a framework for coordinating the development of existing and future standards for the interconnection of heterogeneous computer systems. The result of the study of that subcommittee was the so called *Open System Interconnection* (OSI) *Reference Model*, which became an international standard, ISO 7498 [2], in 1984.

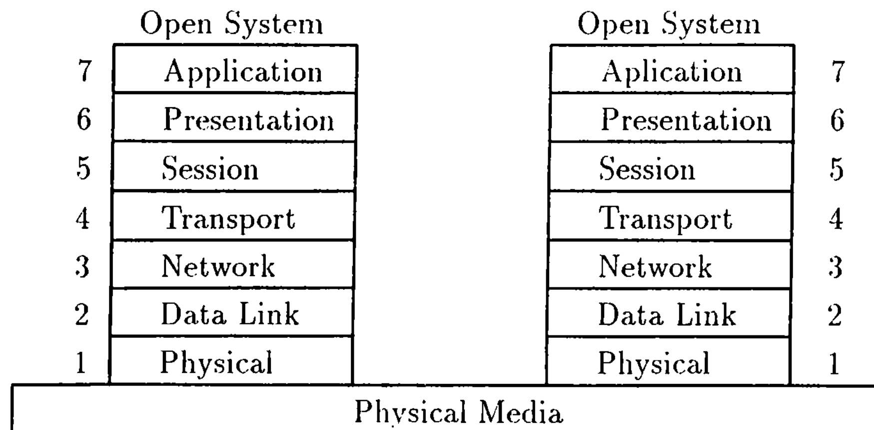


Figure 1.1. OSI Reference Model

To provide the communication among computer applications in a heterogeneous environment, complex hardware and software components are needed to provide the networking services. Using the divide and conquer principle, ISO divides the overall communication functions hierarchically into seven layers (Figure 1.1). Each layer provides services to the layer above by using the services provided by the layers beneath it. Applications access OSI communication services from the highest layer, i.e., layer seven. For each layer, protocols are defined to provide the specific set of services.

Layer one, the *Physical Layer*, is concerned with transmission of unstructured bit stream over physical medium; deals with the mechanical, electrical, functional, and procedural characteristics to access the physical medium. Layer two, the *Data Link Layer*, provides the reliable transfer of information across the physical link; sends blocks of data with the necessary synchronization, error control, and flow control. Layer three, the *Network Layer*, provides the interconnection service. It provides transparency over the topology of the underlying network as well as transparency over the transmission media used in each subnetwork comprising the network. Layer four, the *Transport Layer*, is responsible for moving data reliably form one end system to another end system. It provides end-to-end error recovery and flow control. The *Session Layer* is primarily responsible for the coordination of the communication. The *Presentation Layer* is responsible for the representation of data. Finally, *Application Layer* provides access to the OSI environment for users and also gives distributed information services.

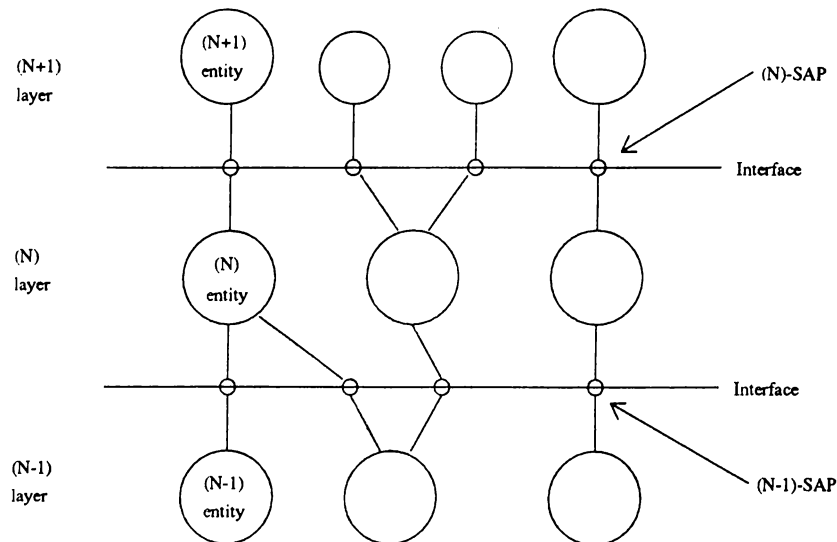


Figure 1.2. Layer Concept of OSI Reference Model

1.1 OSI Architectural Concepts

The OSI Reference Model provides a framework within which protocol standards can be specified for each layer. More fundamental to the framework are the OSI architectural concepts that are formulated independent of the layers and the associated protocols within each layer.

Layering divides the overall communication functions of an open system into a succession of smaller subsystems. Subsystems of the same rank (N) collectively form the *(N)-layer* of the reference model. The objects in the (N)-layer are called *(N)-entities.* These (N)-entities use the services provided by the (N-1)-layer in order to collectively provide the services of the (N)-layer. Entities in the same layer but in different open systems are called *peer* entities.

Cooperation among the peer (N)-entities is governed by one or more *(N)-protocols.* An (N)-protocol is a set of rules and formats that govern the exchange of information between two peer (N)-entities during an instance of communication. Entities in adjacent layers within an open system communicate with each other through their common boundary, across an interface. This interface is called the *Service Access Point* (SAP). Figure 1.2 shows the layer concept of the OSI Reference Model.

The service provided by (N)-entities to the layer above is called the *(N)-service.* The (N)-entities provide this service via invocation of service primitives. An (N)-SAP is characterized by the set of service primitives or abstract operations that can be invoked by an (N+1)-entity at that point.

The service offered by the (N)-layer can be *connection-oriented*, or *connectionless.* For connection-oriented communication, an (N)-association between the two communicating (N+1)-entities is set by establishing an (N)-connection between the two (N)-SAPs. The lifetime of a connection has three distinct phases: connection establishment, data transfer, and connection release. Once the connection is established, the service provider at each end provides a *connection endpoint identifier* to its local service user. Subsequent requests to transfer data may refer to the assigned connection endpoint identifier. In connectionless mode communication, there is neither connection establishment phase nor connection release phase. Each transmitted data unit is self-contained, and is independent of each other.

The information units exchanged among the peer (N)-entities are called *(N)-protocol-data-units* ((N)-PDUs). Every (N)-PDU has two major components: an *(N)-protocol-control-information* (N)-PCI and *user-data.* The user-data component is what an (N)-entity receives from its user, i.e., some (N+1)-entity. On receiving the user-data, the (N)-entity prefixes it with an (N)-protocol-control-information ((N)-PCI) to form its (N)-PDU and passes it down to (N-1)-entity. For an (N-1)-entity the (N)-PDU is treated as an *(N)-service-data-unit* ((N)-SDU, i.e., user-data). Each time when a PDU passes down to a lower layer in the source system, the provider of the lower layer prefixes a PCI to the PDU. By the time the PDU reaches the lowest layer, it has already been encapsulated with layers of PCIs (Figure 1.3). The encapsulated PDU at the physical layer is then ready to be transmitted across the transmission media to the lowest layer of the target system. At the target system, the encapsulated PDU is moved upwards towards the receiving (N)-entity. Each time the encapsulated PDU goes up a layer, one of its PCIs is stripped off. Precisely, a layer in the target system strips off PCI added earlier by the same layer in the source system.

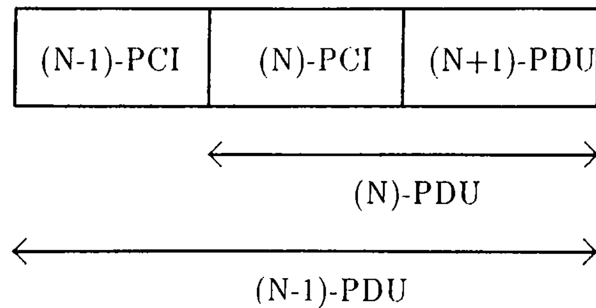


Figure 1.3. Encapsulation of PDUs

1.2 Service Definitions and Protocol Specifications

Every OSI protocol standard comprises two sets of documents : the *Service Definition* and the *Protocol Specification.* A service definition is used to describe the functional boundary between two adjacent layers. The major components of service definitions define *service elements* which consist of *service primitives.* Service primitives are classified as follows: The *Request* service primitive type is issued when a higher layer is requesting a service from the next lower layer. The *Indication* primitive type is issued by a layer providing the service to notify its user. The *Response* primitive type is issued by a layer to acknowledge the receipt of an indication primitive. The *Confirm* primitive type is issued by the layer providing the previously requested service to confirm that the activity has been completed. A *user-initiated* service primitive is one initiated by a service user (e.g. Request or Response), whereas a *provider-initiated* service primitive is one that is initiated by the service provider (e.g. Indication or Confirm).

A protocol specification is the specification which must be satisfied by all protocol entities in a layer [3]. The major components of a protocol specification document includes the following:

- definition of PDUs,
- elements of procedures,
- mapping to underlying services,
- the description of a protocol,
- the conformance requirements.

Finite State Machine (FSM) description is the most common tool to describe the behaviour of a protocol, which is the most important component

of a protocol specification. In short, an FSM description identifies a finite set of states to describe the operation of the protocol machine. This definition is augmented with explanation of the other components of the protocol specification with the aid of figures, tables, and English text. Protocol standards for Application Layer protocols often use the semi-formal notation called Abstract Syntax Notation One (ASN.1) [4] for the definition of the parameters and data structures of the PDUs.

However, protocol specifications written in natural language often contain ambiguities which must be resolved in implementations. For this reason, in the design, specification and analysis of protocols the use of formal methods increases. Most of the methods of protocol validation and testing require the protocols be specified in a formal language. The *Formal Description Techniques* (FDTs) that are presently considered for application in this area are Estelle [5, 6], LOTOS [7, 8], and SDL [9, 10]. Estelle and LOTOS are developed within ISO for application to OSI, but can also be used in other areas. SDL was originally developed by CCITT for the description of switching systems, but is also used in the description of communication protocols. Estelle and SDL are based on an Extended Finite State Machine (EFSM) model. Estelle is enhanced with Pascal data structures, expressions and statements for the description of interaction parameters, additional state variables, and related processing. SDL is largely oriented towards graphical representation, and supports abstract data types.

LOTOS, which is the main specification formalism used in this thesis, is a combination of a variation of Milner's CCS [11] and a particular notation for abstract data types, ACT ONE [12]. In LOTOS, a system is specified as a hierarchy of process definitions. The behaviour of each process is expressed by a behaviour expression. The formation rules for the behaviour expressions are an integral part of LOTOS. In general, the behaviour expressions can be expressed as possibly infinite tree-like structures. Synchronous communication among processes is handled by applying operators to processes, where semantics of the operators are expressed in terms of the behaviour expressions. This topic will be elaborated in Chapter 2.

1.3 Conformance Testing

A division of OSI objects into layers and entities allows protocol standards to be specified independent of actual implementations. However, to have successful communication among open systems it is not sufficient to specify and standardize communication protocols. It must also be possible to ascertain that the implementations of these protocols really conform to the standard protocol specifications. One way to do this is by testing. The activity of testing these protocol implementations against the relevant protocol specifications is known as *Protocol Conformance Testing.*

The component of protocol specifications related to conformance testing is the conformance requirements stated within the specifications. There are two types of conformance requirements. *Static conformance requirements* specify the limitations on the combinations of implemented capabilities permitted in an implementation which is claimed to conform to a protocol specification. *Dynamic conformance requirements* on the other hand, specify what observable behaviour is allowed by the relevant protocol standard. Conformance testing involves testing both the capabilities and behaviour of an implementation, and checking what is observed against both the conformance requirements in the relevant standard(s), and what the implementor states the implementation's capabilities are [13]. The purpose of conformance testing is to increase the probability that different implementations are able to inter-work, and gives confidence that an implementation has the required capabilities and its behaviour conforms consistently in representative instances of communication.

Conformance testing is a kind of *functional testing.* An implementation of a protocol entity is solely tested with respect to its specification. With functional testing externally observed functionality of an implementation is tested by using tests that are derived from the specification. It is also called *black-box testing*: a system is treated as a black box, whose functionality is determined by observing it, i.e., no reference is made to the internal structure of the implementation. Only the interactions of the system with the environment are available. The main goal is to determine whether the right product has been built or not.

ISO has lead the development of a standard methodology and framework for the conformance testing of OSI systems. This effort resulted in a standard called *Conformance Testing Methodology and Framework* (CTMF) which

consists of seven parts [14, 15, 16, 17, 18, 19, 20]. Part 1 introduces the general concepts and describes the test architectures. Part 2 defines the abstract test suite development methodology. Part 3 defines a language for specifying the abstract test suites, called the *Tree and Tabular Combined Notation* (TTCN). Part 4 is on test realization which consists of test selection and parameterization. Part 5 puts requirements on test laboratories and clients for the conformance assessment process. Various conformance test reports are defined in this part. Part 6 is about profile testing. Part 7 develops a number of proformas called implementation conformance statements in order to define the implementation flexibility allowed by the base protocol standards. According to CTMF, standard test suites are developed for each OSI protocol in order to be used by supplier or implementors in self-testing, by user of OSI products, by telecommunication administrations, or by third party testing organizations.

According to the terminology of CTMF [14], the implementation to be tested is called the *Implementation Under Test* (IUT). A number of tests designed to establish conformance of the IUT is called an *Abstract Test Suite* (ATS). Test suites are made up of *Test Cases*, each of which tests for the satisfaction of one or more *Test Purposes*, where a test purpose corresponds to a conformance requirement stated in the specification. Specifically, a test case consists of a test purpose, a *preamble* which is a sequence of events that transfers the IUT into the desired initial state for testing, a *test body* consisting of the actual events that test for the satisfaction of the purpose, and a *verdict assignment* formulated as the result of the application of the test case to the IUT, which can be *pass*, *fail* or *inconclusive*.

Testing an IUT requires a conceptual testing architecture in which the tester makes stimuli from the bottom and top SAPs and observes the results from these two SAPs. Sequences of interactions occurring at these *Points of Control and Observation* (PCOs) form the essential basis for determining whether or not an implementation conforms to the protocol standard. This brings a natural functional division for the testing functions : *Lower Tester* for ASP and PDU inputs/outputs at the bottom SAP, and *Upper Tester* for ASP inputs/outputs at the top SAP. Figure 1.4 illustrates the conceptual testing architecture. Based on this conceptual test architecture, ISO defines one local and three external abstract test methods for conformance testing. These are *Local Test Method, Distributed Test Method, Coordinated Test Method*, and *Remote Test Method* [14, 15].

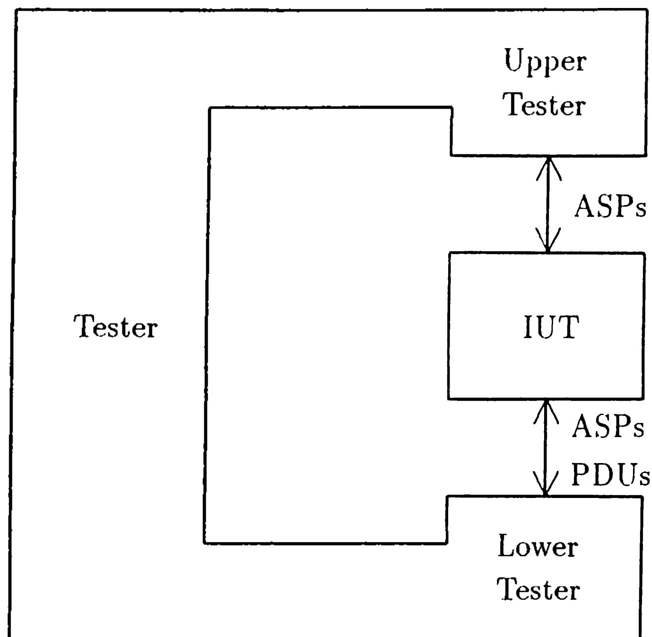


Figure 1.4. Conceptual Testing Architecture

The following types of tests are defined by ISO to be applied to protocol implementations:

1. **Static Conformance Review:** A review of the extent to which the static conformance requirements are met by the IUT.
2. **Basic Interconnection Tests:** Tests of an IUT which have limited scope to determine whether or not there is sufficient conformance to the relevant protocols for interconnection to be possible, without trying to perform thorough testing.
3. **Capability Tests:** Tests to verify the existence of one or more claimed capabilities of an IUT.
4. **Behaviour Tests:** Tests to determine the extent to which one or more dynamic conformance requirements are met by the IUT.

1.4 Overview

Protocol standardization is performed on a variety of levels. Base standards developed by international bodies define fundamental and generalized procedures along with a number of options. On regional or national level, functional standards are derived from the base standards according to the needs of specific applications. Since the aim of conformance testing is to test the satisfaction of the requirements corresponding to the implemented capabilities of a protocol, the derivation of test suites must be based on functional standards.

While on one hand the use of formal methods in the specification of distributed systems increases; on the other hand this brings in the question of how these formal specifications can be used in conformance testing. Since the specification of the same protocol can be made in many different forms and styles, a similar question arises on which form to use in order to facilitate conformance testing. This approach is called specification for testability.

Specification for testability deals with the development of completely decomposable formal specifications called base specifications. The test suite generation step must be taken into account at the initial design stage of a system while developing formal base specifications. Base specifications must be developed in such a way that the features that must be tested separately should be identifiable, and functional specifications can be systematically derived from the base specifications in order to be used as the basis of subsequent test generation.

The main contribution of this thesis is on the systematic derivation of functional specifications from base specifications written in the formal description technique LOTOS. The approach adopted for this purpose is called slicing which has been previously defined and used in software engineering. By making some transformations on protocol specifications, called behaviour reductions, slices of such specifications are obtained which define the behaviour corresponding to a specific function of the protocol. The functions are extracted according to the hierarchically structured test suites of the base protocols. The techniques developed are applied on some OSI protocols written in different specification styles.

Chapter 2 starts the discussion by introducing LOTOS and its underlying semantics. It gives the formal framework of conformance testing, along with the

some formal approaches from the literature. Chapter 3 is on specification for testability. It gives a design trajectory which results in testable protocol specifications. Chapters 4 and 5 are about the application of the methods described in Chapter 3 on some protocol specifications written in LOTOS. Specifically, Chapter 4 is on the development of base specifications whereas Chapter 5 gives examples on the derivation of functional specifications from base specifications. The steps covered in the derivation of complete test suites from base and functional specifications are explained and compared in Chapter 6 with several examples. Finally, Chapter 7 contains the conclusions and some comments on future work.

The contribution in Chapter 2 is the in depth comparison of the two major approaches to LOTOS based test design, namely the labelled transition system based approach of [21] and the EFSM-Chart based approach of [30]. In Chapter 3, several techniques that can be used while developing formal base specifications, including parameterization and behaviour specification, are given. Various types of behaviour reductions (horizontal, vertical, diagonal) are defined and an original algorithm is developed in order to perform the reductions in a systematic manner. In Chapter 4 previously developed specifications of INRES and ACSE protocols are taken, and several enhancements are carried out in order to obtain their formal base specifications. Specifically, the specifications are parameterized, behaviour and data types related to the static conformance review process are added, and the features that must be tested are included. In addition to its base specification, two documents for INRES protocol are developed. One of them (PICS proforma) is used in parameterizing the specification, and the other one which defines a test suite structure is used in the derivation of functional specifications. The contributions in Chapter 5 are the original applications of the behaviour reduction algorithm developed in Chapter 3 on three base protocol specifications. The most notable among these is the application of our methodology on the relatively large standard base specification of the Transport Protocol. The contribution in Chapter 6 is mainly the analysis of the tests generated from the functional specifications obtained by the techniques of this thesis.

Chapter 2

FORMAL METHODS IN CONFORMANCE TESTING

As a result of the recent increase in the use of formal methods for the specification of OSI protocols there is a need to systematically derive tests from such formal specifications. Formal methods in protocol conformance testing concern testing of protocol implementations for conformance with respect to their formal specifications, where testing is based on a formal definition of what constitutes conformance [21].

Using formal methods in conformance testing brings in some benefits. First, the concepts of conformance testing can be defined formally, and thus more precisely. Secondly, the use of FDTs makes it possible to do formal test validation, i.e., checking whether a test really tests what it is intended for. Thirdly, the use of formal methods allows the definition and use of test generation algorithms.

For the purpose of examining the consequences of the use of FDTs on conformance testing, ISO and CCITT started a joint project around 1983: *Formal Methods in Conformance Testing* (FMCT) [22]. The scope of the project is to define a general methodology on how to perform conformance testing of a protocol implementation given a formal specification of the protocol.

The aim of this chapter is to introduce the formal approach to conformance testing. Section 2.1 summarizes the main concepts of the Formal Description Technique LOTOS which is used throughout the thesis. Section 2.2 continues the discussion with the theory of processes, and gives the definition of a

labelled transition system which can be seen as a first formalization of the notion of a process. Section 2.3 introduces the theory of conformance and defines what constitutes conformance in a formal context. Section 2.4 describes some relations which can be used as the basis of conformance testing. Section 2.5 gives some methods from the literature for the systematic derivation of test suites from formal specifications that allow a semantic interpretation in terms of labelled transition systems. Finally, Section 2.6 discusses a relatively different approach to automatic derivation of test cases from LOTOS specifications based on the *Chart* formalism.

2.1 Introduction to LOTOS

LOTOS [8] is one of the two Formal Description Techniques developed within ISO for the formal specification of open distributed systems, and in particular for those related to the Open Systems Interconnection (OSI) computer network architecture [7]. In LOTOS a distributed and concurrent system is described in terms of a set of interacting *processes*. A process is an abstract entity that is able to perform *internal events* and communicates with other processes via *external events* at interaction points called *gates*. An event is an elementary unit of synchronization among processes [23]. The static picture of a process can be imagined as that of a black box capable of communicating with other processes. The mechanisms inside this box are not observable. Thus, all events occurring inside a process (i.e., internal events) are unobservable and denoted by i . In LOTOS, processes can be defined by the temporal relations between the events constituting their externally observable behaviour, and are expressed by ***behaviour expressions*** . The formation rules of the behaviour expressions are essential parts of LOTOS, and are based on a modification of the Calculus of Communicating Systems (CCS) [11]. In the case that an event involves exchange of data, the description of data structures and value expressions is based on the algebraic specification of abstract data types (ADTs) [12].

In LOTOS interactions (i.e., event structures) take the form $g\alpha_1\alpha_2\ldots\alpha_n[c]$, where g is the interaction point (gate) name and each α_i is either ***value*** or *variable declaration.* A value declaration has the form $!E$, where E is a LOTOS expression describing a data value, and a variable declaration takes the form $?x : t$, where x is a name of a variable and t is its sort (i.e, type) identifier. An action denotation may terminate with a predicate c, called the *selection*

predicate, which is used to impose restrictions on the values that may be bound to the variables. The syntax of a process definition header in LOTOS is :

$$\text{process P[gate_list]}(x_1 : t_1, \ldots, x_n : t_n) : \text{functionality}$$

with variables $x_1, x_2, \ldots, x_n$ occurring as free variables in the behaviour expression defining the process P. The *functionality* of a process is the list of the sorts of the values offered at the successful termination of that process.

A behaviour expression is produced by applying an operator to other behaviour expressions. A behaviour expression may also include instantiations of other processes. Given a behaviour expression B, for convenience B may also be called a process. There are several operators in LOTOS to describe behaviour expressions as follows.

- **inaction** (denoted by '**stop**')
  stop means that a process can execute no event.

- **successful termination** (denoted by '**exit**')
  exit is a process whose purpose is solely that of performing the successful termination action denoted by δ, after which it transforms into the dead process **stop**.

- **action prefix** (denoted by ';')
  The behaviour of $a; B$ is considered to be event a followed by the behaviour of B, where B is a behaviour expression.

- **choice** (denoted by '[]')
  The behaviour of B_1 [] B_2 will be the behaviour of either B_1 or B_2, where B_1 and B_2 are behaviour expressions. The choice offered is resolved in the interaction of the process with its environment.

- **parallel composition** (denoted by '$|[g_1, g_2, \ldots, g_n]|$')
  The behaviour of $B_1|[g_1, g_2, \ldots, g_n]|B_2$ will be a composition in which B_1 and B_2 must synchronize with respect to the set of gates $S = \{g_1, g_2, \ldots, g_n\}$. When the set of synchronization gates S is empty, then the parallel composition operator is denoted as $|||$, and when S is the set of all gates, the parallel composition operator is written as $||$.

- **hiding** (denoted by '**hide** $g_1, g_2, \ldots, g_n$ in B')
  Hiding allows one to transform some observable actions performed on

the gates $g_1, g_2, \ldots, g_n$ of a process given by the behaviour expression B into unobservable ones. These actions are thus made unavailable for synchronization with other processes.

- **process instantiation** (denoted by '$P[g_1, g_2, \ldots, g_n]$')
  A process instantiation is formed by a process identifier P with an associated list $[g_1, g_2, \ldots, g_n]$ of *actual gates* which correspond to the list of *formal gates* given in the respective process definition. Such a process instantiation occurs in the behaviour expression defining some other process, or process P itself. The instantiation of a process in LOTOS resembles the invocation of a procedure in a conventional programming language such as Pascal.

- **guarding** (denoted by '$[p] \longrightarrow B$')
  Any behaviour expression B may be preceded by a predicate (i.e., a guard) and an arrow. The interpretation is that if the predicate holds, then the behaviour described by the behaviour expression is possible, otherwise the whole expression is equivalent with **stop**.

- **sequential composition** (denoted by '$B_1 >> B_2$')
  The informal interpretation of this construct is that if B_1 terminates successfully, then the execution of B_2 is enabled. Parameters can be passed from the enabling process to the enabled process by using the **accept** construct.

- **disabling** (denoted by '$B_1 [> B_2$')
  Process B_1 may be disabled by the first action of process B_2 and the control is irreversibly transferred from the interrupted B_1 to the interrupting B_2. In the case that the interruptible B_1 performs a successful termination action, the disabling process B_2 disappears.

- **let** (denoted by '**let** $x_1 : t_1 = E_1, \ldots, x_n : t_n = E_n$ **in** $B(x_1, \ldots, x_n)$')
  Let construct of LOTOS is used to associate value expressions $E_1, \ldots, E_n$ to the free variables $x_1, \ldots, x_n$ of a behaviour expression $B(x_1, \ldots, x_n)$.

- **generalized choice** (denoted by '**choice** $x_1 : t_1, \ldots, x_n : t_n$ [] B', or
  '**choice** g **in** $[a_1, a_2, \ldots, a_n]$ [] B')
  Using the binary choice operator '[]' only finite number of alternatives can be expressed. Generalized choice allows an unknown number of alternatives to be specified. Sets of gate identifiers or variable declarations may be used for indexing.

A number of specification styles have been defined for LOTOS specifications [24]. In the *monolithic* style only observable interactions of a system are presented and ordered as a collection of alternative sequences of interactions in branching time. In the *constraint-oriented* style again only observable interactions are presented, but their temporal ordering is defined by a conjunction of different constraints. Monolithic and constraint oriented styles are *extensional* description styles in that they define a system in terms of its external observable behaviour, viz. their concern is *what* of the system. With the *state-oriented* style the system is regarded as a single resource whose internal state space is explicitly defined. In the *resource-oriented* style both observable and internal interactions are presented. The behaviour in terms of the observable interactions is defined by a composition of separate resources in which the internal interactions are hidden. In turn, these resources may be specified using any style. Both state-oriented and resource-oriented styles are *intensional* description styles, i.e., their concern is the *how* of a system.

LOTOS has the following two models; *labelled transition systems* as models for behaviour expressions, and *many sorted algebras* as models for data types. The model for behaviour expressions is not dependent upon the way data types are interpreted. The subsequent discussion in this chapter is mainly about behaviour expressions and labelled transition systems.

2.2 Labelled Transition Systems

The formalism of labelled transition systems is used for representing the behaviour of processes, so they are suitable for modelling distributed systems. Labeled transition systems serve as the semantic model for a number of specification languages, including LOTOS.

Definition 2.1 A *labelled transition system* is a 4-tuple $< S, L, T, s_0 >$ with

- S is a (countable) non-empty set of *states;*
- L is a (countable) set of *observable actions;*
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the *transition relation;*
- $s_0 \in S$ is the *initial state.*

Notation	**Meaning**
$B \xrightarrow{\mu} C$	$(B, \mu, C) \in T$
$B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$	$\exists B_0 \ldots B_n : B = B_0 \xrightarrow{\mu_1} B_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} B_n = C$
$B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$	$\exists C : B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$
$B \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$	$\neg \exists C : B \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} C$
$B \xRightarrow{\epsilon} C$	$B = C$ *or* $B \xrightarrow{\tau \cdot \tau \cdot \ldots \cdot \tau} C$
$B \xRightarrow{a} C$	$\exists B_1, B_2 : B \xRightarrow{\epsilon} B_1 \xrightarrow{a} B_2 \xRightarrow{\epsilon} C, a \in L$
$B \xRightarrow{a_1 \cdot \ldots \cdot a_n} C$	$\exists B_0 \ldots B_n : B = B_0 \xRightarrow{a_1} B_1 \xRightarrow{a_2} \ldots \xRightarrow{a_n} B_n = C$
$B \xRightarrow{a_1 \cdot \ldots \cdot a_n}$	$\exists C : \xRightarrow{a_1 \cdot \ldots \cdot a_n} C$
$B \not\xRightarrow{a_1 \cdot \ldots \cdot a_n}$	$\neg \exists C : \xRightarrow{a_1 \cdot \ldots \cdot a_n} C$
$out(B)$	$\{a \in L \mid B \xRightarrow{a}\}$
$Tr(B)$	$\{\sigma \in L^* \mid B \xRightarrow{\sigma}\}$
B **after** σ	$\{B' \mid B \xRightarrow{\sigma} B'\}$
B is *stable*	$B \not\xrightarrow{\tau}$

Table 2.1. Notation for Labelled Transition Systems

The labels in L represent the observable interactions of a system; the special label $\tau \notin L$ represents the unobservable, internal action. Table 2.1 introduces some notation and definitions for labelled transition systems. A *trace* is a sequence of observable actions. The traces of a labelled transition system specification S, $Tr(S)$, are all sequences of visible actions that S can perform. The set $out(S)$ contains traces of length 1. S **after** σ collects all states that can be reached after having performed σ. A *stable* process accepts only external events, and can not perform internal action. A *finite state* process is one in which the number of reachable states is finite.

Simple labelled transition systems can be represented by (action-) trees or graphs, where nodes represent states and edges labelled with actions represent transitions. Figure 2.1 illustrates the representation of a simple system.

It is cumbersome to represent complex systems such as protocols directly by action trees as labelled transition systems. A more sophisticated way of representation than graphs or trees is needed. That is a language that allows concise representation of (possibly infinite) labelled transition systems. The operational semantics of LOTOS, which can be obtained by a system of inference rules, generate a labelled transition system for each behaviour expression

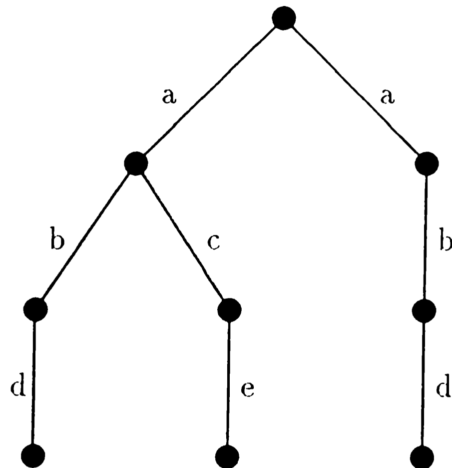


Figure 2.1. Action Tree of a Simple System

of the language. Table 2.2 gives the inference rules corresponding to the operators of the subset of the language employing a finite alphabet of observable actions with no exchange of data, which is called Basic LOTOS [7].

The simple labelled transition system in Figure 2.1 can be represented by a behaviour expression in LOTOS as :

$$a; (b; d; \mathbf{stop}\ []\ c; e; \mathbf{stop})\ []\ a; b; d; \mathbf{stop}. \tag{2.1}$$

In the following, no distinction will be made between B as a behaviour expression, or B as the initial state of its semantics labelled transition system. A process is identified with the labelled transition system modelling it, with the behaviour expression representing the labelled transition system, and with its initial state.

2.3 Theory of Conformance

The starting point for developing a formal approach to conformance testing is the definition of what constitutes conformance. Regarding the fact that the relevance of a protocol specification with respect to testing is its set of conformance requirements, a specification can be considered as the collection of conformance requirements. Formally, a specification S in a particular standard

Combinator	Axioms or Inference Rules
stop	none
exit	**exit** $\xrightarrow{\delta}$ **stop**
a; B	a; $B \xrightarrow{a} B$ (a $\in L$)
i; B	**i**; $B \xrightarrow{\tau} B$
B_1 [] B_2	$B_1 \xrightarrow{a} B_1' \vdash B_1 [] B_2 \xrightarrow{a} B_1'$
	$B_2 \xrightarrow{a} B_2' \vdash B_1 [] B_2 \xrightarrow{a} B_2'$
$B_1 \| B_2$	$B_1 \xrightarrow{a} B_1', B_2 \xrightarrow{a} B_2' \vdash B_1 \|\| B_2 \xrightarrow{a} B_1' \|\| B_2' \ (a \neq \tau)$
	$B_1 \xrightarrow{\tau} B_1' \vdash B_1 \|\| B_2 \xrightarrow{\tau} B_1' \|\| B_2$
	$B_2 \xrightarrow{\tau} B_2' \vdash B_1 \|\| B_2 \xrightarrow{\tau} B_1 \|\| B_2'$
$B_1 [> B_2$	$B_1 \xrightarrow{a} B_1' \vdash B_1 [> B_2 \xrightarrow{a} B_1' [> B_2 \ (a \neq \delta)$
	$B_1 \xrightarrow{\delta} B_1' \vdash B_1 [> B_2 \xrightarrow{\delta} B_1'$
	$B_2 \xrightarrow{a} B_2' \vdash B_1 [> B_2 \xrightarrow{a} B_2'$
$B \backslash (g_1, \ldots, g_n)$	$B \xrightarrow{a} B' \vdash B \backslash (g_1, \ldots, g_n) \xrightarrow{\tau} B' \backslash (g_1, \ldots, g_n) \quad a \in \{g_1, \ldots, g_n\}$
	$B \xrightarrow{a} B' \vdash B \backslash (g_1, \ldots, g_n) \xrightarrow{a} B' \backslash (g_1, \ldots, g_n) \quad a \notin \{g_1, \ldots, g_n\}$

Note : $B \backslash g_1, \ldots, g_n$ is a short notation for **hide** $g_1, \ldots, g_n$ **in** B.

Table 2.2. Axioms and Inference Rules for Basic LOTOS

can be written as

$$S = \{r_1, r_2, r_3, \ldots\}$$

where each r_i is a conformance requirement [21]. A conforming implementation then is the one which satisfies all requirements specified by the standard:

$$\forall r \in S \ : \ I \ \mathbf{sat} \ r \tag{2.2}$$

The formal language in which requirements are expressed is denoted by $\mathcal{L}_R$. Languages that express requirements or properties are called *Logical Languages*. Although they are powerful, logical languages are not constructive. For this reason current standardized FDTs (Estelle, LOTOS, SDL) are not based on logical languages. Formal descriptions written in these FDTs do not define requirements, but observable behaviour. For conformance testing, however, it is important to know which requirements are implicitly defined by the expressions in the formal descriptions.

Let $\mathcal{L}_{FDT}$ be the formal language of the FDT. By introducing the relation **spec**, the set of all requirements implicitly specified by an expression $B \in \mathcal{L}_{FDT}$

is defined as:

$$S_B = \{r \in \mathcal{L}_R \mid B \,\mathbf{spec}\, r\} \tag{2.3}$$

Combining (2.2) and (2.3), we have for an implementation I that conforms to B :

$$I \,\mathbf{conforms-to}\, B \;=_{def}\; \forall r \in \mathcal{L}_R : B \,\mathbf{spec}\, r \;\textit{implies}\; I \,\mathbf{sat}\, r \tag{2.4}$$

In this way, conformance can be defined as a relation between implementations and behaviour specifications, with the meaning that an implementation conforms to a specification if every requirement specified by the behaviour specification is satisfied by the implementation. Conformance as a relation is referred to as an implementation relation.

2.4 Implementation Relations and Conformance

An implementation relation formalizes the notion of correctness of an implementation I with respect to a specification S [21]. Implementation relations can be obtained by comparing observations made of I with observations made of S. For an implementation relation it is sufficient that observations of the implementation can be related to the observations of the specification, in the sense that, the behaviour of the implementation can be 'explained' from the behaviour of the specification. An implementation is considered to be correct if all observations made of it by any environment can be explained from the behaviour of the specification.

The definition of the conformance relation given in (2.4) depends on the language $\mathcal{L}_R$ and the relations **spec** and **sat**. Different choices for these allow the definition of different implementation relations, and result in different classes of conforming implementations with the same language $\mathcal{L}_{FDT}$. Many implementation relations have been proposed in the literature. They are either equivalences, in which case one must show that the implementation provides *exactly* the behaviour stipulated by the specification, or preorders, in which case one shows that the implementation provides *at least* the behaviour required.

If we choose the same language $\mathcal{L}_{FDT}$ as the formal language of the class of implementations, i.e., if the underlying semantics of implementations with respect to behaviour is the same as that of specifications, a basic relation

between processes is that of trace preorder $\leq_{tr}$. Formally,

$$I \leq_{tr} S \ =_{def} \ \forall \sigma \in L^* \ : \ I \overset{\sigma}{\Rightarrow} \ implies \ S \overset{\sigma}{\Rightarrow} \tag{2.5}$$

Informally, when defined in an observational way, the relation $\leq_{tr}$ specifies that traces observed by an implementation should also be observed by the specification. Equivalently, it specifies that any trace which is not in the specification shall not be in the implementation. A tester consisting of a single test case can be constructed for each specification in order to test for the satisfaction of the $\leq_{tr}$ relation [25]. This test case reaches a state marked with verdict *fail* whenever the first action of a trace that is not present in the specification is observed. For an infinite set of possible actions, this tester will be very large for practical specifications.

To judge the validity of an implementation it is not sufficient to consider only the sequences of actions it can perform, what an implementation can refuse to do must also be considered [25]. The deadlocks observed by I after having performed a certain sequence of actions must also be observed by S. The resulting implementation relation is called *testing preorder* or *failure preorder* $\leq_{te}$ (*reduction* **red** in [26]). Formally,

$$\begin{aligned} I \leq_{te} S \ =_{def} \quad & \forall \sigma \in L^*,\ \forall A \subseteq L : \\ & if\ \exists I' : (I \overset{\sigma}{\Rightarrow} I' \wedge \forall a \in A : I' \overset{a}{\not\Rightarrow}) \\ & then\ \exists S' : (S \overset{\sigma}{\Rightarrow} S' \wedge \forall a \in A : S' \overset{a}{\not\Rightarrow}) \end{aligned} \tag{2.6}$$

By introducing a specific requirement language $\mathcal{L}_{must}$;

$$\mathcal{L}_{must} \ =_{def} \ \{\textbf{after}\ \sigma\ \textbf{must}\ A \mid \sigma \in L^*, A \subseteq L\} \tag{2.7}$$

which states the requirement that after having performed σ, at least one of the actions in A must be performed, the relation given in (2.6) can be reformulated as :

$$\begin{aligned} I \leq_{te} S \ \textbf{iff} \quad & \forall \sigma \in L^*, \forall A \subseteq L : \\ & S\ \textbf{after}\ \sigma\ \textbf{must}\ A\ implies\ I\ \textbf{after}\ \sigma\ \textbf{must}\ A \end{aligned} \tag{2.8}$$

Like $\leq_{tr}$, $\leq_{te}$ has a severe disadvantage for conformance testing. It is characterized using a quantification over all $\sigma \in L^*$, which poses the problem of having to verify that the implementation does not have unspecified deadlocks for all $\sigma \in L^*$.

In [26] the implementation relation **conf** was introduced to reduce this problem. The relation **conf** reduces the quantification to traces in the specification, so it checks only the deadlock behaviour of an implementation after those sequences of actions that the specification is able to perform. By using **conf** it is not checked whether an implementation has extra traces not specified in the specification, i.e., extensions in the functionality of the implementation with respect to the specification remain undetected. Formally,

$$\begin{aligned} I\ \mathbf{conf}\ S \ =_{def} \quad & \forall \sigma \in Tr(S),\ \forall A \subseteq L : \\ & if\ \exists I' : (I \stackrel{\sigma}{\Rightarrow} I' \wedge \forall a \in A : I' \stackrel{a}{\not\Rightarrow}) \\ & then\ \exists S' : (S \stackrel{\sigma}{\Rightarrow} S' \wedge \forall a \in A : S' \stackrel{a}{\not\Rightarrow}) \end{aligned} \tag{2.9}$$

By using the requirement language $\mathcal{L}_{must}$, the implementation relation **conf** can be defined as :

$$\begin{aligned} I\ \mathbf{conf}\ S\ \ \mathbf{iff} \quad & \forall \sigma \in Tr(S), \forall A \subseteq L : \\ & S\ \mathbf{after}\ \sigma\ \mathbf{must}\ A\ implies\ I\ \mathbf{after}\ \sigma\ \mathbf{must}\ A \end{aligned} \tag{2.10}$$

Considered in itself, the **conf** relation is not attractive as an implementation relation. It is not transitive, and therefore not a preorder. However, this relation plays an important role in *incremental testing*: in order to test for $\leq_{te}$, correctness with respect to $\leq_{tr}$ and **conf** can be separately checked because when taken together relations $\leq_{tr}$ and **conf** exactly give $\leq_{te}$ [21].

2.5 Language Based Systematic Test Suite Derivation

In the realm of labelled transition systems the implementation relation **conf**, defined in the previous section, is a reasonable candidate to formalize the notion of conformance for the purpose of conformance testing [21]. Therefore, the next question is how to derive test suites for **conf** systematically from a labelled transition system specification S. This section introduces some test derivation algorithms starting from the fact that for a complete test suite the requirements specified by a specification should be exactly those tested by the test suite.

The *Canonical Tester T(S)* for a specification S consists of the test suite needed to establish the implementation relation **conf**. An implementation I conforms to a specification S according to **conf** if all required behaviour of S is implemented in I [27]. The Canonical Tester can be regarded as a

process which is the 'inverse' of the specification in the sense that the Canonical Tester of the Canonical Tester of a specification is the specification itself (i.e., $T(T(S)) = T(S)$). It runs in parallel with the IUT and communicates with it with full synchronization. According to [26], the basic idea behind the Canonical Tester $T(S)$ is:

- it is capable of exploring all and only traces in S, i.e., $Tr(T(S)) = Tr(S)$;
- I conforms to S if and only if every deadlock between I and the tester $T(S)$ can be explained by the tester having reached a terminal state.

If a representation is given by a behaviour expression with labeled transition system semantics, then there are two possibilities for the derivation of test cases. Either the behaviour expression is replaced by its semantics, from which tests are derived using algorithms for labelled transition systems, or the algorithms are transformed to work on behaviour expressions as well. The second option is more natural because behaviour expressions give finite, implementable representations of infinite labelled transition systems.

In the literature two main approaches to the construction of $T(S)$ from S can be found in [26] and [28]. The first one elaborated in [26] is an example of the first option stated above. It uses a model of processes (failure trees) identifying processes that are equivalent with respect to testing. This leads up to a complete algorithm that constructs canonical testers $T(S)$ which are unique for testing equivalent processes. The second one is the syntactical approach explored in [28], which is based on the second option of deriving tests from behaviour expressions. Here the tests are derived directly from the specifications' syntaxes. The method is named the CO-OP method after its main components. The three main attributes used in the compositional derivation of the canonical tester $T(B)$ of a behaviour expression B in Basic LOTOS are:

- **Compulsory(B)**, which is the set of sets of actions that may not be refused by B after it has reached any possible stable state, i.e., a state from which no internal action is possible.
- **Options(B)**, which is the set of actions that may be refused by B because of the presence of an alternative internal action **i** in an unstable state.

- **B after a**, which is the subsequent behaviour of B after having performed the action a.

The construction of the tester is compositional in the sense that the attributes needed to compute the tester $T(B_1 * B_2)$ can be obtained from the attributes of the testers $T(B_1)$ and $T(B_2)$ where $*$ is any Basic LOTOS operator. The approach developed in [28] results in a completely inverse behaviour of the specification, which consists of a single process, and in which it is difficult to identify individual test cases.

A recently developed method given in [21] extends the CO-OP method to a small subset of full LOTOS and makes it possible to obtain individual test cases. The method is based on the idea that, in order to derive complete set of test cases from a labelled transition system S, the requirements tested by these test cases must be exactly the requirements specified by S. Hence for every requirement of type **after** σ **must** A specified by S, i.e., for every σ, A with

$$S \stackrel{\sigma}{\Rightarrow} \text{ and } S \textbf{ after } \sigma \textbf{ must } A \tag{2.11}$$

there must be **after** σ **must** A among the tested requirements.

The first step in generating test cases is the derivation from S of the requirements **after** σ **must** A, according to (2.11). This procedure can start with $\sigma = \epsilon$, and then proceeds recursively. All $A \subseteq L$ with S **after** ϵ **must** A are determined, and for each A there must be a test case t_A:

$$t_A = \Sigma \, \{a; t_a \mid a \in A\}$$

where t_a is the behaviour of the test case after a (Σ operator represents the LOTOS choice operator among multiple behaviour expressions). It suffices to derive test cases for those sets A that are minimal with respect to $\subseteq$ because if a test case t_a tests requirement **after** ϵ **must** A, then it also tests the requirements for **after** ϵ **must** A' with $A \subseteq A'$. Since for each A satisfying S **after** ϵ **must** A, always $A \bigcap out(S) \subseteq A$, and S **after** ϵ **must** $(A \bigcap out(S))$, it is sufficient to consider only $A \subseteq out(S)$.

The next step in the construction of the behaviour of the test case after a, i.e., t_a. t_a must test the requirements **after** a **must** A. The above procedure can be recursively applied if the calculation can be repeated with $\sigma = \epsilon$, i.e. if there is an S' such that;

$$S \textbf{ after } a \textbf{ must } A \text{ iff } S' \textbf{ after } \epsilon \textbf{ must } A.$$

For any process S, the behaviour of the tester after interaction in an event is fully dependent on the behaviour of S after interaction in this event. If there is more than one state S' with $S \stackrel{a}{\Rightarrow} S'$, then, after interaction in a, it is not known to the tester which transition has been chosen within the process under test. The behaviour of S after interaction in a can then be seen as a non-deterministic choice, which is given by the expression:

$$\textbf{choice } S \textbf{ after } a =_{def} \Sigma \, \{i; S' \mid S' \in S \textbf{ after } a\}$$

The expression **choice** S **after** a defines the non-deterministic choice among all states that are reachable by S after having performed a (Σ represents the generalized choice operator).

So, the three attributes needed to derive a test case are $out(S)$, **choice** S **after** a, and sets A that are minimal with respect to $\subseteq$. In order to facilitate the construction of sets A from behaviour expressions, the concept of *acceptance sets*, denoted by $\overline{C_\epsilon}$, is introduced in [21]. These sets are defined using existential quantification over states, and each set consists of sets of external actions that can occur in a state, i.e., they represent the action capability of a state. In order to obtain acceptance sets that are minimal with respect to $\subseteq$, some optimization steps are performed on $\overline{C_\epsilon}$, which is denoted by the relation $\sqsubseteq$. The resulting sets are called *reduced acceptance sets.*

Based on the definitions presented above, the following algorithm generates a test case for a specification S (v represents the verdict associated with the particular state of the test case):

Algorithm 2.1 (Test Case Generation) A test case t for a specification S is obtained by:

- Step 1. Choose $M \sqsubseteq \Psi_0(C)$ where $C \sqsubseteq \overline{C_\epsilon}$;
  (The transformation $\Psi_0(C)$ defines a set of sets formed by taking one (or more) element from each set in C)

- Step 2. Choose $A \in M$ and $v(t)$ = **fail**,
  or if $M = \phi$ then $A = out(S)$ and $v(t)$ = **pass**,

- Step 3. $t_A = \Sigma \, \{a; t_a \mid a \in A\}$, where t_a is a test case for **choice** S **after** A.

B	$out(B)$	$st(B)$	$C(B)$	choice B after g
stop	ϕ	$true$	$\{\phi\}$	**stop**
$a; B_1$	$\{a\}$	$true$	$\{\{a\}\}$	**i**; B_1 $if\ a = g$ **stop** $if\ a \neq g$
i; B_1	$out(B_1)$	$false$	$C(B_1)$	**choice** B_1 **after** g
B_1 [] B_2	$out(B_1)$ $\bigcup out(B_2)$	$st(B)$ and $st(B_2)$	$\{out(B)\}$ $\bigcup if\ not(st(B_1))$ $then\ C(B_1)$ $\bigcup if\ not(st(B_2))$ $then\ C(B_2)$	**choice** B_1 **after** g []**choice** B_2 **after** g

Table 2.3. Compositional Computation of Acceptance Sets

A reduced acceptance set is related to the set *Compulsory* of the CO-OP method, the difference being that for a reduced acceptance set C, always $\bigcup C = out(S)$. As a consequence of this property, *Options* of the CO-OP method is not needed any more.

2.5.1 Test Derivation From Finite Systems

Having acceptance sets as a way to derive test cases, compositional rules can be defined with which acceptance sets can be compositionally derived from a restricted class of LOTOS behaviour expressions. Table 2.3 gives such rules. In order to obtain the compositional rule for [], information about the stability of the operands is required. The stability of a behaviour expression B is defined by the predicate $st(B)$, and is added as an extra attribute, which is also computed compositionally.

Example 2.1 If we consider the example behaviour expression given in (2.1) representing the process of Figure 2.1:

$$B = a; (b; d; \textbf{stop} \ [] \ c; e; \textbf{stop}) \ [] \ a; b; d; \textbf{stop}$$

The test case in Figure 2.2 is derived following Algorithm 2.1 as follows, using $B = B_1 \; [] \; B_2$, with:

$$B_1 = a; (b; d; \textbf{stop} \; [] \; c; e; \textbf{stop})$$
$$B_2 = a; b; d; \textbf{stop}$$

- $out(B) = \{a\}$, $st(B_1) = st(B_2) = st(B) = true$,
  $C(B) = \{\{a\}\}$, $\Psi_0(C(B)) = \{\{a\}\}$.
  Choose $A = \{a\}$, and $v =$ **fail**.
  choice B **after** a $=$ **i**;$(b; d; \textbf{stop} \; [] \; c; e; \textbf{stop}) \; [] \; \textbf{i}; (b; d; \textbf{stop})$

- Let B' $=$ **choice** B **after** a $=$ **i**;$(b; d; \textbf{stop} \; [] \; c; e; \textbf{stop}) \; [] \; \textbf{i}; (b; d; \textbf{stop})$
  $out(B') = \{b, c\}$, $st(B') = false$,
  $C(B') = \{out(B')\}$
  $\bigcup C(\textbf{i}; (b; d; \textbf{stop} \; [] \; c; e; \textbf{stop}))$
  $\bigcup C(\textbf{i}; b; d; \textbf{stop})$
  $= \{\{b, c\}, \{b\}\}$, $\Psi_0(C(B')) = \{\{b, c\}, \{b\}\}$.
  Choose $A = \{b\}$, and $v =$ **fail**.
  choice B' **after** b $=$ **choice** **i**;$(b; d; \textbf{stop} \; [] \; c; e; \textbf{stop})$ **after** b
  $[]$ **choice** **i**; $b; d;$ **stop** **after** b
  $=$ **i**;d;**stop**

- Let B'' $=$ **choice** B' **after** b $=$ **i**;d; **stop**
  $out(B'') = \{d\}$, $st(B'') = false$,
  $C(B'') = \{\{d\}\}$, $\Psi_0(c(B'')) = \{\{d\}\}$.
  Choose $A = \{d\}$, and $v =$ **fail**.
  choice B'' **after** d $=$ **i**;**stop**

- Let B''' $=$ **choice** B'' **after** d $=$ **i**;**stop**
  $out(B''') = \phi$, $C(B''') = \{\phi\}$, and $\Psi_0(C(B''')) = \phi$.
  $A = out(B''') = \phi$, and $v =$ **pass**.

2.5.2 Test Derivation with Infinite Branching

A language that allows infinite branching is accomplished by introducing actions that consist of a pair of gates and values, denoted by $< g, v >$. The usual interpretation of such actions is that of *value communication*. A gate

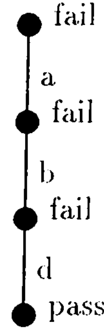


Figure 2.2. Sample Test Case

represents a place where communication takes place, e.g. a SAP, and the value represents the message that is communicated at that place. This is the kind of communication that occurs in the FDT LOTOS.

When non-finite systems are considered, the method proposed in [21] derives the test cases in two steps. First, the initial behaviour of test cases, i.e., how to determine the set A in the test case $t_A = \Sigma\ \{a; t_a \mid a \in A\}$ is considered, and then the subsequent behaviour of the tester, viz., the part t_a is determined.

The initial behaviour of a test case involves determining a set $A \subseteq L$ such that $A \in M \sqsubseteq \Psi_0(C)$ where $C \sqsubseteq \overline{C_\epsilon}(S)$. Test derivation from behaviour expressions with value communication suffers from the problem of infiniteness. Because of infinite branching, although finite reduced acceptance sets exist, elements of such sets, also sets, turn out to be infinite.

To deal with these infinite sets while computing $\Psi_0(C)$, the elements of the acceptance set C can be divided into subsets. Each element A of C is represented by the union of sets $D'' \in D'$, where sets D' together form a larger set D, to which a bijection can be defined from C. This brings in the computation of $\Psi_0(C)$ in two steps. The first step is the computation of $\Psi_0(D)$ where D is a set of sets of sets of actions. The next step involves, for each $E \in \Psi_0(D)$, the computation of $\Psi_0(E)$. This set has a simpler structure; it is a set of sets of actions. For the derivation of test cases, an $A \in \Psi_0(E)$ is required. Such an A is always finite and has the form $\{< g_1, v_1 >,\ < g_2, v_2 >, \ldots, < g_n, v_n >\}$. The corresponding test case is:

$$\Sigma\{a; t_a \mid a \in \{< g_1, v_1 >,\ < g_2, v_2 >, \ldots, < g_n, v_n >\}\}$$

B	$out(B)$	$st(B)$	$D(B)$
stop	ϕ	$true$	$\{\phi\}$
$g?x\ [p]; B_1$	$\{g?x\ [p]\}$	$true$	$\{\{g?x\ [p]\}\}$
$i; B_1$	$out(B_1)$	$false$	$D(B_1)$
$B_1\ []\ B_2$	$out(B_1)$ $\bigcup out(B_2)$	$st(B_1)$ and $st(B_2)$	$\{out(B)\}$ $\bigcup if\ not(st(B_1))\ then\ D(B_1)$ $\bigcup if\ not(st(B_2))\ then\ D(B_2)$

Table 2.4. Compositional Computation of Acceptance Sets for Infinite Systems

which can be written as

$$g_1!v_1; t_{<g_1,v_1>}\ []\ g_2!v_2; t_{<g_2,v_2>}\ []\ \cdots\ []\ g_n!v_n; t_{g_n,v_n>}$$

Compositional rules for the derivation of the attributes $out(B)$, $st(B)$, and $D(B)$ are given in Table 2.4.

For the subsequent behaviour of test cases it is sufficient to consider finite number of test cases t_a, where $t_a =$ **choice** B **after** $< g, v >$. Table 2.5 gives compositional rules for **choice** B **after** $< g, v >$ which is analogous to **choice** B **after** g in Table 2.3 ($B[v/x]$ denotes the expression B where each free occurrence of x is replaced by the value v, and $p[v/x]$ denotes the substitution of a value v for a variable x in a predicate p.)

Example 2.2 A test case for the following behaviour expression B is derived in detail :

$$B = g?x\ [x < 20]; h!(x+2); \mathbf{stop}\ []\ \mathbf{i}; g?y\ [y > 10]; h!y; \mathbf{stop}$$

- $out(B) = \{g?x\ [x < 20], g?y\ [y > 10]\}$,
  $st(g?x\ [x < 20]; h!(x+2); \mathbf{stop}) = true$,
  $st(\mathbf{i}; g?y\ [y > 10]; h!y; \mathbf{stop}) = false$,
  $D(B) = \{out(B)\} \bigcup D(\mathbf{i}; g?y\ [y > 10]; h!y; \mathbf{stop})$
  $= \{\{g?x\ [x < 20], g?y\ [y > 10]\}\} \bigcup \{\{g?y\ [y > 10]\}\}$
  $= \{\{g?x\ [x < 20], g?y\ [y > 10]\}, \{g?y\ [y > 10]\}\}$.

$\underline{B}$	$\underline{\textbf{choice } B \textbf{ after } A}$
stop	**stop**
$g?x\ [p]; B_1$	$\textbf{i}; B_1\ [v/x]$ if $g = h$ and $p\ [v/x]$ **stop** if $g \neq h$ or *not* $p\ [v/x]$
$i; B_1$	**choice** B_1 **after** $< h, v >$ if p **stop** if not p
$B_1\ []\ B_2$	**choice** B_1 **after** $< h, v >$ **choice** B_2 **after** $< h, v >$

Table 2.5. Compositional Computation of Subsequent Behaviour for Infinite Systems

Choose $E \in \Psi_0(D(B))$ and $A \in \Psi_0(E)$:
$\Psi_0(D(B)) = \{\{g?x\ [x < 20], g?y\ [y > 10]\}, \{g?y\ [y > 10]\}\}$,
$E = \{g?y\ [y > 10]\}$, and $A = \{< g, 15 >\}$.

The test case then is: $t = g!15; t_{<g,15>}$, where $t_{<g,15>}$ is a test case for **choice** B **after** $< g, 15 >$.

- **choice** B **after** $< g, 15 >$
  $= \textbf{choice } g?x\ [x < 20]; h!(x+2); \textbf{stop after } < g, 15 >$
  $[]\ \textbf{choice i}; g?y\ [y > 10]; h!y; \textbf{stop after } < g, 15 >$
  $= \textbf{i}; h!17; \textbf{stop}\ []\ \textbf{i}; h!15; \textbf{stop}.$

Let $B' = \textbf{choice } B \textbf{ after } < g, 15 > = \textbf{i}; h!17; \textbf{stop}\ []\ \textbf{i}; h!15; \textbf{stop}$
$out(B') = \{h!17, h!15\}$, $st(\textbf{i}; h!17; \textbf{stop}) = st(\textbf{i}; h!15; \textbf{stop}) = false$,
$D(B') = out(B') \bigcup out(\textbf{i}; h!17; \textbf{stop}) \bigcup out(\textbf{i}; h!15; \textbf{stop})$
$= \{\{h!17, h!15\}, \{h!17\}, \{h!15\}\}$.

Choose $E \in \Psi_0(D(B')) = \{\{h!17, h!15\}\}$:
$E = \{h!17, h!15\}$, and $A \in \Psi_0(E) = \{< h, 15 >, < h, 17 >\}$.

The test case $t_{<g,15>}$ is :

$\Sigma\{a; t_a \mid a \in \{< h, 15 >, < h, 17 >\}\} = h!15; t_{<h,15>} \; [] \; h!17; t_{<h,17>}$.
Both equations **choice** B' **after** $< h, 15 >$, and **choice** B' **after** $< h, 17 >$ are equal to **i;stop**.

Taking the parts together a test case for B is :

$$t = g!15; (h!15; \textbf{stop} \; [] \; h!17; \textbf{stop})$$

2.6 Chart Based Test Generation

The framework presented in [21] for formal specification based test generation, is by no means complete, and some open questions remain. At some points simplifications and choices for formalizations are made. These choices are open for further discussion, depending on investigations whether the presented formalizations are workable ones. An example is the relation between physical objects (the IUT) and formal objects (the specification, requirements). It is assumed that implementations and test application can be formally modelled, and that observations calculated for the class of models are also valid for the physical implementations. For validation of the usefulness of the framework it should also be applied to other specification formalisms. The method is applied to toy specifications formed by a restricted class of LOTOS behaviour expressions. The extension of the method to handle full LOTOS behaviour expressions needs further study.

This section introduces another approach to the generation of test cases which is based on the idea that data flow as well as the control flow of the implementation must be tested. It combines FSM-based techniques with functional software testing and analysis [29]. The following phases are distinguished in the derivation of test suites from LOTOS specifications.

- Normalization of the specification,
- Identification of the functions to be tested,
- Generation and analysis of test cases,
- Test selection and representation.

The operational semantics of LOTOS is used to convert the specifications into an intermediate form, and then data and control flow is extracted from this

intermediate form. The second and fourth phases deal with the data flow aspect whereas the third phase deals with the control flow aspect of testing. In the following each of the above phases are explained in detail based on the ideas found in [30]. The applications of the methodology will be examined in Chapter 6.

2.6.1 Normalization

Individual test cases that make up a test suite can be seen as part of the behaviour defined in the specification which is then inverted to specify the behaviour of the tester. Therefore, it is necessary to identify the control and data flow in the specification. In order to identify the control and data flow, an intermediate representation called the normalized specification is needed. The intermediate representation for full LOTOS specifications is called the EFSM-Chart. Chart is originally introduced by Milner as a sequential nondeterministic interleave model for parallel computation [31].

Definition 2.2 A *chart* is a 7-tuple $m = < J, N, V, R, j_0, Z, h_0 >$ where

- J is a finite set, the *control states* of m,
- N is a finite set, the *transitions* of m,
- V is a finite set, the *variables* of m,
- R is a finite set, the *rules* of m,
- $j_0 \in J$ is the *initial control state* of m,
- $Z \subseteq J$ is a finite set, the *terminal control states* of m,
- h_0 is the *initial assignment* to the variables of m.

The possible transitions of a chart are defined by a set of rules whereby each rule defines a class of transitions.

Definition 2.3 A *rule* of a chart is an 8-tuple $r = < a, j, j', n, p, c, f, h >$ where

- a is an action, the *when* clause of r,
- $j \in J$ is a control state, the *from* clause of r,
- $j' \in J$ is a control state, the *to* clause of r,
- $n \in N$ is a transition number, the *transition* clause of r,

- p is a predicate, the *guard* clause of r,
- c is a predicate, the *condition* clause of r,
- f is a function, the *action* clause of r,
- h is a function, the *assignment* clause of r.

The semantic interpretation of EFSM-Chart is that; the transition n occurs when the chart is in the control state j, and the predicate p is true for the current assignment of the variables. Then it may participate in an event that matches the when clause of a if the condition c is satisfied. This leads the system to the new control state j'. The action clause of r represents the variables that are updated by the function f due to value passing in the interprocess communication, whereas the assignment clause h represent value passing due to process instantiations.

A LOTOS specification can be converted into EFSM-Chart in two phases. In the first phase the following syntactic transformations are performed on the input specification in order to facilitate the chart construction.

- All occurrences of full parallel composition are transformed into general parallel composition on all of the synchronized gates.
- All occurrences of sequential composition are replaced by a generalized parallel composition on a gate which is hidden.
- Generalized choice expressions on gate lists are converted into normal choice expressions on multiple instances of the same behaviour actualized with a different element of the gate list.
- Process instantiations are expanded in-line until actual gate parameters are found identical to formal ones.
- Internal events which are not the first action in a choice expression are removed.
- Since variables in LOTOS have local significance, they are renamed uniquely to avoid global conflicts in the chart.

In the second phase this resulting specification is converted into a chart by bottom-up synthesis. The chart corresponding to a behaviour B is recursively

built from the sub-charts corresponding to sub-behaviours contained in B. Behaviour operators are eliminated one by one by constructing the corresponding state machines.

The chart for **exit** and **stop** has only one state and no transition. Since all sequential compositions are transformed into parallel compositions in the first phase, **exit** and **stop** behaviours are treated in the same way while constructing the chart.

The chart corresponding to the action prefix operator $a[c]; B$ is obtained by generating a new state and adding a transition from this state to the initial state of the chart for B, with a *when* clause of a and a *condition* clause of c. The chart corresponding to the behaviour expression $i; B$ is constructed in the same way with the exception that the *when* clause of the new transition is i_s qualifying it as a *spontaneous* transition. The chart corresponding to the behaviour expression $[p] \rightarrow B$ is obtained by generating a new state and adding a new transition form this state to the initial state of the chart for B, whose *when* and *guard* clauses are i and $[p]$, respectively. In all of the above cases the *initial assignment* h_0 of the chart for B is stored in the *assignment* clause of the added transition.

For the choice operator, the transitions of the two machines are merged and a single machine with an initial state corresponding to the initial states of the merged machines, is obtained. The disable operator is interpreted like the choice operator, and a new machine is formed by adding the initial rules of the disabling process to every state of the disabled process as alternative behaviour.

The chart corresponding to the **let** $x_1 : t_1 = E_1, \ldots, x_n : t_n = E_n$ **in** B and **choice** $x_1 : t_1, \ldots, x_n : t_n$ [] B is obtained by updating the initial assignment h_0 of the chart for B. For the hide operator, all interactions that are hidden become internal, and only the resulting data flow in terms of assignments to the variables is kept.

For interleaved parallel composition, a Cartesian product of the two machines is calculated. While constructing the chart corresponding to the synchronized parallel composition, all possible execution paths are not considered, and a single sequence of actions is obtained. During this construction different synchronization features of LOTOS, i.e., *value matching, value passing* and *value generation* are handled separately.

Translation of process instantiations depends on whether the instantiation is recursive or not. No new transition is created for non-recursive instantiations, and the parameters passed in the instantiation are stored in the *initial assignment* of the chart. For recursive instantiations a new transition is created with a *when* clause of i_r.

Example 2.3 If we consider the behaviour expression given in Example 2.2 :

$$B = g?x\ [x < 20]; h!(x+2); \textbf{stop}\ []\ \textbf{i}; g?y\ [y > 10]; h!y; \textbf{stop}$$

The corresponding chart is :

$$m\ = <\{1,\ldots,5\}, \{n1, n2, \ldots, n5\}, \{x, y\}, R, 1, \{3, 6\}, \epsilon> \text{ where}$$

$$\begin{aligned} R\ = <\ & \{g?x : Nat, 1, 2, n1, true, [x < 20], \epsilon, \epsilon\}, \\ & \{i_s, 1, 3, n2, true, true, \epsilon, \epsilon\}, \\ & \{h!(x+2), 2, 4, n3, true, true, \epsilon, \epsilon\}, \\ & \{g?y : Nat, 3, 5, n4, true, [y > 10], \epsilon, \epsilon\}, \\ & \{h!y, 5, 6, n5, true, true, \epsilon, \epsilon\} >. \end{aligned}$$

2.6.2 Identification of the Functions to be Tested

Data flow analysis which is widely used in code optimization [32] plays an important role in software testing [33]. When applied to protocol testing, the flow of data reflects how input primitive parameters determine the values of context variables, and they in turn affect the values of output primitive parameters. Input/Output primitives are ASPs and PDUs. A *Data Flow Graph* (DFG) models the flow of data in the chart. In [30] an algorithm is given which constructs the data flow graph from the EFSM-Chart.

In order to obtain the protocol functions to be tested, the data flow graph must be decomposed into slices according to a user-defined criteria. Each slice thus obtained consists of a number of transitions from the chart which collectively represents a protocol function.

2.6.3 Generation and Analysis of Test Cases

A test case generated from a deterministic finite-state machine consists of sequence of events, i.e., it is linear; whereas the presence of the action i_s in the chart makes it highly nondeterministic. This necessitates the generation of nonlinear test cases. The algorithm proposed in [30] to generate unparameterized test cases from EFSM-Chart take nondeterminism into account. An outline of the algorithm is as follows:

First a transition tour of the EFSM-Chart is obtained. Transition tour generation is based on converting the EFSM-Chart into an Euler graph and then performing a depth-first traversal of this graph while each time passing through a node including a distinct edge into the tour. The tour is then divided into sequences that start from the initial state and end in the initial state or one of the final states. These sequences are called *partial test cases*. A partial test case may contain spontaneous transitions (i_s transitions). Next, the partial test case is checked if there exists a spontaneous transition which is not present in the partial test case, but is an alternative to any of the transitions in it. If so, the partial test case is updated by adding a sequence of transitions that start with the alternative spontaneous transition and ends in a final state or a state belonging to the partial test case. This procedure is repeated until no spontaneous transition exists that is alternative to the updated partial test case. An edge is virtually added from each final state to the initial state and the partial test case becomes a completed test case.

Since the above algorithm may generate redundant, uninteresting or infeasible test cases, the next step is the analysis of the generated test cases. In order to facilitate the analysis process, a graph which reveals the control structure of, as well as the data dependencies within test cases, is produced. This is called the *Test Case Dependence Graph* (TCDG) [34]. In some of the test cases, predicates can never be satisfied on a path due to the existence of an assignment on that path which causes the predicates always evaluate to false. By using TCDG, the slice of each predicate in a test case is obtained in order to be evaluated to detect any infeasibilities. The slice of a predicate with respect to a test case consists of all statements in the test case whose execution possibly affect the boolean value of the predicate. Since, the detection of infeasible test cases is an undecidable problem, the produced predicate slices must be evaluated by the test designer.

Some of the generated test cases may contain redundant assignments which do not have any affect on the function of the tester. So the next step is the reduction of the test cases. By using TCDG, every assignment in a test case onto which no data dependence is present (i.e., which has no effect on the subsequent behaviour) is dropped. Since any infeasibilities are already removed in the previous step, all predicates in a test case which do not depend on the parameter values of input primitives can also be dropped.

2.6.4 Test Selection and Representation

Specification of tester's behaviour in a test case is completed by test selection and representation [34]. Send event parameters left unassigned in the previous steps are also assigned to specific values.

If functional test selection is used, the protocol functions identified by using the data flow graph must be tested using the test cases obtained from the chart. For this purpose, it is necessary to extract the transition labels of each slice of the data flow graph. Generated test cases are then selected using transition labels of each of the data flow functions. Alternatively, hierarchical test selection of ISO [15] can also be used at this step. The generated test cases are placed in a test suite hierarchy, a test purpose is assigned to each test case, and using this information a verdict is associated with each final event of the test cases. A closely related approach with this method will be introduced in Chapter 3.

Final step in the derivation of test suites is test case representation. Since the standard notation for specifying conformance test suites defined within ISO is TTCN [16], the aim at this step is to obtain suitable representations of test cases according to TTCN. The behaviour of the tester is the dual of IUT's behaviour, and can be obtained by behaviour inversion. Except internal actions, the direction of all of the events in a test case are inverted, that is input events are converted to output events and vice versa. 'Pass' verdicts are assigned for each final event of the test cases, and in order to deal with unforeseen responses from the IUT, an OTHERWISE event with an associated verdict of 'Fail' is added as alternative to each receive event of tester. 'Inconclusive' verdicts are assigned to events resulting from internal transitions. Any constraints imposed on the values of ASPs and PDUs are determined, and event parameters left unassigned during previous steps are assigned to specific values.

Since the behaviour specification in LOTOS is not state-machine oriented but algebraic, LOTOS has no built in unidirectional input/output type of interactions. Instead LOTOS interactions are multi-directional with value passing, generation and matching. This raises some problems in performing behaviour inversion automatically because the directions of the events are not evident from the specifications. In order to overcome this problem additional value declarations can be added to every interaction to make the direction of the interactions explicit.

2.6.5 Application

When we apply the chart based test design methodology to the sample behaviour expression for which a test case is derived in Example 2.2 by using the approach given in [21], and for which the chart is produced in Example 2.3, we obtain the following two unparameterized test cases represented in TTCN form.

```
Test Case 1:
    g! x [x<20]
      h? y [y=(x+2)]    Pass
      h? y [y=x]        Inconclusive
      ?OTHERWISE        Fail

Test Case 2:
    g! x [x>10]
      h? y [y=x]        Pass
      ?Otherwise        Fail
```

The first test case tests for the correct execution of the path beginning with the external action 'g?x' while at the same time considering the internal transition that the implementation can make at any time. The second transition tests only for the execution of the spontaneous transition. Since, in the case of a value offer at gate 'g' which is less than 20 but greater than 10, it is possible for the implementation to reject the first case due to an internal transition, the second test case is needed. The parameter values of send and receive events can be varied to try sending the same events with different parameter values according to the constraints.

Chapter 3

SPECIFICATION FOR TESTABILITY

A standardized protocol specification is the first step in a complete development process that will eventually result in a piece of hardware or software, or a mixture of both, that can be called an implementation of the protocol standard [35]. An important aspect that should be considered in the specification of protocols is to ensure that the specifications will result in testable implementations, where testable in this context means facilitating testing. This aspect is called *Specification For Testability*.

Specification for testability deals with finding out special design issues that must be obeyed by protocol specifiers in order to end up with testable protocol specifications [36]. On one hand, high quality specifications are an essential ingredient of any automated or semi-automated process of test generation, while on the other hand, a specification of the same protocol can be made in many different forms and styles [24]. Not all of these styles are equivalent in terms of testability, i.e., one may facilitate conformance testing more than the other. Therefore, it is necessary that an FDT must be used in conjunction with a methodology and/or style which ensures that the resulting specifications are not only concise, correct and complete, but they are also testable and suitable for automated test case generation.

Within this perspective, specification for testability deals with the following activities:

- developing formal base specifications using certain styles that help obtaining testable specifications,
- including in the specifications the aspects of the protocols that need to be tested,
- deriving functional and profile specifications from base specifications,
- designing tests from functional specifications.

This chapter starts the discussion on to the concept of specification for testability by introducing a design trajectory based on the activities stated above. This first section is about the development of formal base specifications. It defines what a base specification is and presents some methods on obtaining formal base specifications from the respective base standards. An important use of base specifications is the systematic derivation of functional and profile specifications. Functional and profile specifications, and their relationship with test selection is discussed in the Section 3.2. This section introduces an alternative approach called specification selection which is based on the idea that selection of tests can be done before actually generating them. It also introduces our approach to specification selection which is referred to as slicing. Finally Section 3.3 is about the related approaches to specification selection from the literature. The concepts developed in this chapter are illustrated with a number of examples from some protocols specified in LOTOS, in chapters 4 and 5. Designing tests from functional specifications is the subject of Chapter 6.

3.1 Base Specification Development

Protocol standardization is done on a variety of levels. On an international level, base standards are developed by ISO and CCITT for each layer according to The Basic Reference Model given in [2]. Base standards, as developed by international bodies, define fundamental and generalized procedures, an infrastructure that can be used by a variety of applications. Most informal OSI base standards do not define the behaviour of a protocol entity uniquely, but by allowing multiple choices in service elements, service parameters, functional units, and elements of procedures, they leave some space for implementors. The requirements on a minimum set of capabilities to be satisfied by all of

the implementations are stated in the static conformance requirements of the standards. There is a need to reflect these in the formal specifications. For this purpose, base specifications are developed by considering mandatory and optional features of a system at all levels.

Development of base specifications that provide a suitable basis for test suite generation is an incremental process. The first step is the reflection of implementation capabilities in the specifications. This can be achieved by parameterizing the specification. Then comes the incorporation of the conformance requirements into the specifications, i.e., behaviour specification. Base specifications must be developed in such a way that features tested separately should be identifiable.

3.1.1 Parameterizing Specifications

For each protocol standard a *Protocol Implementation Conformance Statement Proforma* (PICS Proforma) has to be defined and standardized [20]. A PICS proforma states explicitly the implementation flexibility allowed by the protocol standard. It is a document in the form of a questionnaire which details the implementation options by listing all the possibilities for selection, and the legitimate range of variation of the global parameters controlling the implementation of the functions. The implementor states the implemented options of a protocol by filling in the entries in the standard proforma, which then becomes the *Protocol Implementation Conformance Statement* (PICS) of that particular implementation.

Appendix B defines a PICS Proforma for the INRES Protocol [37], for which the complete base specification in LOTOS is given in Appendix A. In compliance with the requirements for PICS proformas stated in [20], the PICS proforma for INRES contains prose entries identifying the implementation, the implementor, "yes/no" entries indicating the implementation of each optional and conditional capability. The required support for conditional capabilities depends on how the implementor completed particular optional entries. Furthermore, for PDU fields there are entries indicating the range of values implemented.

PICS proforma information must be incorporated into formal base specifications. The allowance for different capabilities and options in standards means

that specifications can be parameterized. The PICS proforma is the formal parameter of the specification, and the PICS for a particular implementation defines actual parameters of the specification for that particular implementation. Static conformance requirements (SCR) define constraints on the values of the PICS parameters.

Thus, a protocol specification S can be written as a parameterized specification with formal parameters PICS-Proforma of the type SCR-type, i.e., the set of all possible correct values, determined by the static conformance requirements. This can be expressed as:

$$S(PICS - Proforma : SCR - type)$$

The instantiation of S for a particular implementation I with its associated PICS_I is given by;

$$S(PICS_I)$$

The set of requirements derived from the behaviour specified by $\text{S}(\text{PICS}_I)$ define the dynamic conformance requirements for the implementation I.

Before conformance testing starts, the capabilities of the IUT should be checked for conformity to the static conformance requirements stated in the standard. This process is called the *Static Conformance Review*. Within the above framework, static conformance review can easily be implemented in base specifications. It corresponds to checking whether the PICS has a valid value with respect to the static conformance requirements, i.e., whether the PICS is of the type defined by SCR.

Since LOTOS allows the definition of specification parameters, base specifications in LOTOS can easily be parameterized. Implementation capabilities are defined in terms of predicates, and the names assigned to these predicates are made parameters of the specification. Static conformance review can be implemented as an ADT function which takes an instance of PICS proforma as input, and returns a boolean value which determines whether or not the value satisfies the static conformance requirements of the protocol. Only if static conformance requirements are satisfied does the specification describe active behaviour, viz. it provides the implementor or tester with an abstract model of the protocol dynamic conformance requirements.

3.1.2 Behaviour Specification

The base specification has to cover all mandatory and optional features described in the protocol standard. It should be developed in a hierarchical manner, and features tested separately should be identifiable. [38].

The normal (valid) behaviour is specified based on the state tables given in the informal standard using traditional techniques. As mentioned previously, PICS parameters can be used to specify conditions related to the optional features of the protocol wherever appropriate.

Like valid behaviour, invalid and inopportune behaviour specification is important with respect to conformance testing [39]. Invalid behaviour specifies the implementation's response to badly constructed incoming events (PDUs and/or ASPs), and events are called inopportune if they occur when not allowed by the protocol specification. Both invalid and inopportune behaviour specify the behaviour of the protocol machine when operating against an environment that behaves incorrectly. This kind of behaviour is important when the implementations are subject to *robustness testing* which can be seen as part of conformance testing.

Since the default behaviour of LOTOS specifications in response to unspecified events is to run into deadlock, both invalid and inopportune behaviour must be explicitly defined in the base specifications. This can be done on a state by state basis. The response of the protocol entities to invalid/inopportune events is specified with the consideration of such inputs in each state. Obviously, this can be achieved more easily if the state-oriented specification style [24] of LOTOS is used.

It is not possible to distinguish a normal behaviour from an invalid/inopportune behaviour without additional constructs such as predicates, comments, or keywords. Since new keywords require modification in the language, invalid/inopportune behaviour can be specified as alternative choice branches in LOTOS processes, and distinguished by using comments.

3.2 Functional and Profile Specifications

The existence of base standards does not make the design of an OSI system straightforward. OSI base standards, as discussed in the previous section, allow multiple choices in service elements, service parameters, functional units, and elements of procedures. It is not expected that an OSI product would provide the implementation of every possible feature stated in the base standard. A careful selection of such choices is required by any OSI implementation. An implementation for a special area must therefore be restricted to a certain subset of the base standard.

On a regional level, e.g. Europe, the selection of options is harmonized in order to facilitate interoperation [36]. The resulting protocol standards are called *profiles*. A profile is a set of one or more base standards and the identification of the chosen classes, subsets , options and parameters of those base standards necessary for accomplishing a particular function [39]. In order to ascertain interoperability among different profiles, standardized profiles called *International Standardized Profiles* (ISP) are defined to identify groups of related profiles.

A profile may cover a number of protocols from different layers. A *functional standard* defines, for a particular layer and its associated base standard, a precise combination of options and procedures to be used in a given profile. Functional standards are currently developed by regional or national standardization bodies based on the informal base standards. *Functional specifications* are the corresponding formal definitions of functional standards. In order to base the test generation process on functional specifications, a systematic way of deriving functional specifications from base specifications is needed. The ways to achieve this and its advantages are discussed in the sequel.

3.2.1 Hierarchical Test Selection

Through conformance testing, conformance to only those implemented functions of a protocol which have been chosen from a number of options, can be shown. Therefore, the test generation methods have to consider the idea of choosing from options. Such a reduction of the size of the generated test suite by choosing an appropriate subset is called *test selection*.

The standardized approach to conformance testing given in the OSI Conformance Testing Methodology and Framework suggests the hierarchical development of test suites [15]. The initial development stage for any protocol test suite is the *Test Suite* ***Structure*** *and Test Purposes* (TSS&TP) standard, where the test suite structure is designed in terms of nested test groups in a top down manner, and the test purposes are explained in plain English. This document forms a stable basis for the development of the abstract test suites. A good test suite structure has to satisfy several needs in order to be used as the basis of a suitable test suite. It is important in ensuring the coverage of the resulting test suites, and needs to demonstrate that it is possible to map any feature which has a conformance requirement to at least one appropriate test purpose.

In order to be used in test selection it is also important that the test suite structure is capable of being subsetted in some convenient manner [40]. That is, if parts of the protocol are optional then it must be convenient to exclude from the test suite all test cases for those features which are not implemented. In turn this implies that the structure must be expandable to an arbitrary level of detail. The standardized method of achieving this is by a tree structure where each non-leaf node, representing a test group, contains basic structuring information on how its branches are to be developed, and where the leaves of the tree are the test purposes themselves.

Appendix C contains the test suite structure and test purposes document developed for the INRES protocol, the detailed operation of which is explained in Chapter 4. The standard testing methodology gives guidance on what structure is required at the highest levels of a test suite. It identifies three requirements: Basic Interconnection Tests, Capability Tests and Behaviour tests [40]. The latter subdivides into three categories; Valid Behaviour tests, Invalid behaviour tests, and Inopportune behaviour tests. The meaning of these categories is that Basic Interconnection tests are used simply to see if a tester can establish communication with the implementation it is testing, Capability tests are intended to discover if the implementation of the protocol shows any signs of actually supporting each unit of functionality that it is claimed to support. The intention is to save unnecessary testing if a given feature is clearly unsupported. As there are only a small number of Capability tests for the INRES protocol, it is considered unnecessary to identify a subset of these to be used as Basic Interconnection tests; and Basic Interconnection tests and Capability tests are grouped in a single category. Since such tests can be achieved by a

selection of Behaviour tests, no special tests are devised for this category, but appropriate references to Behaviour tests are given, as recommended by the standard methodology given in [15].

As will be explained in detail in the following chapter, INRES is an asymmetric protocol in that an implementation can function as an initiator or responder, each role being quite different from each other. As this is a major consideration of an implementation it follows that this division forms the top level of the structure for the test suite.

For INRES TSS&TP the decision was made to place the subdivisions of Invalid and Inopportune Behaviour tests in a single group alongside the Valid Behaviour tests. This was based on the fact that PDUs do not contain large number of parameters, and the practical consideration that an extra subdivision would convey no useful information but would add an unnecessary depth of complexity to the tree.

For the valid behaviour section of the INRES TSS&TP it was found necessary to add a new substructure for clarity. This comprises the three phases of the protocol: Connection Establishment, Data Transfer, and Disconnection. On the other hand, the substructure for invalid/inopportune behaviour tests were divided on a state by state basis. These elements exercise the ability of the implementation to correctly perform state event transitions when given an appropriate stimulus.

Test selection can be based on the developed TSS&TP, by parameterizing the test suite, and then selecting those applicable tests according to the test suite structure. But, this means that test cases must exist for the whole protocol including all options. The number of test cases generated may be very large, or even infinite. This implies that the execution of all generated test cases is impossible, or simply too expensive. Alternatively, tests can be generated from functional specifications instead of base specifications, which will largely simplify the generation and selection of test cases. So there is a need to the systematic generation of the functional specifications from base specifications.

Since the two step procedure of first generating too many test cases and then making a selection from this set may be undesirable; this overproduction can be avoided by performing the selection on the specification instead. Before test generation, some transformations can be applied to the base specification

such that tests derived from the transformed, partial specification correspond to a selection of the tests from the original specification. This process is called *Specification Selection.*

3.2.2 Protocol Slicing According to Test Suite Structure

The approach to specification selection investigated in this thesis performs the transformations on base specifications according to the hierarchically structured test suite trees defined and standardized for each base protocol. It is based on the idea that before generating the test cases, an appropriate test suite structure has to be designed, and a base specification has to be decomposed into abridged specifications reflecting the nodes of the test suite structure tree on a suitable level. The nodes of the test suite structure tree are test groups, and decomposition stops at the test case level. This approach to specification selection can be considered as a kind of *slicing* applied to protocol specifications written in the formal description technique LOTOS.

The notion of program slicing, originally introduced in [41], is the process of finding all statements in a program that directly or indirectly affect the value of a variable at a specific point in the program. The statements selected constitute a *slice* of the original program with respect to that variable occurrence. A slice is a self-contained executable program with the simple meaning that it should evaluate the variable occurrence identically to the original program for all test cases [42]. Slicing is useful in program debugging, automatic parallelization, and program integration.

The established method of automatically obtaining program slices according to some user-defined criteria is based on the use of a particular program representation called the *Program Dependence Graph* (PDG). The PDG of a program has one node for each simple statement (assignment, read, write etc.), and one node for each control predicate expression (if-then-else, while-do etc.). It has two types of directed edges, namely data dependence edges, and control dependence edges. Data dependence edges reveal the relationship between the definition and use of each of the program variables. Control dependence edges on the other hand identify the possible execution paths in the program [42]. Once a program is represented by its program dependence graph, the

slicing problem is simply a vertex-reachibility problem, and thus slices may be computed in linear time [43].

The slicing method used in decomposing base specifications for the purpose of obtaining functional specifications is similar to program slicing in the sense that, the aim is to obtain self-contained and executable functional specifications which describe only the required behaviour with respect to a test group objective. But since LOTOS is not procedural language developed to write programs, but an algebraic specification formalism defined to describe observable behaviour of distributed systems, it is not straightforward to obtain the data and control dependencies within LOTOS specifications directly. An intermediate form called EFSM-Chart which corresponds to the normalized specification is devised for this purpose. But since the transformation of a LOTOS specification into state machine requires much effort, it is of no use when the aim is to obtain the protocol slices before the actual test generation procedure starts.

3.2.3 Slicing and Behaviour Reductions

Protocol slicing according to a specific test suite structure is a hierarchical process and leads to a tree of specifications. Each node of this tree corresponds to a non-leaf node of the related test suite structure tree, and the abridged specification describes only those dynamic conformance requirements that the objective of the test group focusses on. Since the functionally decomposed specification directly provides the behaviour related to the test cases constituting the corresponding test group, to base the generation of conformance test suites on functional specifications rather than base specifications can play an important role in automating the test generation process.

The derivation of subsets from a specification by slicing for the purpose of obtaining functional specifications, is referred to as *behaviour reduction*. The three types of transformations that can be defined on base specifications are; *vertical reduction*, *horizontal reduction* and *diagonal reduction*.

Vertical behaviour reductions deal with specification modularity and omit some of the processes completely which are not of interest to a specific functional specification. Reductions based on classes, functional units, entity roles, regimes and phases of protocols are mainly done by vertical reductions. As

an example, for asymmetric protocols each role (e.g. Initiator and Responder) can be considered as a separate protocol, and vertical reduction is applied to obtain the behaviour specific to each role.

While performing horizontal reductions, processes are not omitted completely but some behaviour expressions within the processes can be excluded. For example valid and invalid/inopportune behaviour are usually specified as alternative choice branches, and functional specifications consisting only of valid behaviour or invalid/inopportune behaviour can be obtained by applying horizontal reductions to choice branches such that irrelevant branches are simply discarded. Horizontal reductions can be applied to other LOTOS operators, as well.

Diagonal reductions on the other hand, do the reductions at much more lower levels in a more selective manner. Some specification constructs which can be subject to diagonal reductions are as follows:

- Optional PDUs, and PDU parameters,
- Formal variable parameters in the definition, and the corresponding actual variables in the instantiation of the processes,
- Formal gate parameters in the definition and the corresponding actual gate parameters in the instantiation of the processes,
- Some value and variable declarations of event structures.

Furthermore, diagonal reductions performed on the constructs stated above necessitate further reductions (horizontal, vertical, or diagonal) within the process bodies in the resulting specifications.

All of the three types of reductions performed on base specifications result in some of the data type definitions to become completely or partially redundant. Since the presence of redundant data types does not cause any problem with respect to LOTOS semantics, this situation has no effect on the executability of the obtained specifications. These redundant data types can be the subject of other reduction procedures which have to consider the data part as well as the behaviour part of the LOTOS specifications. These kinds of reductions are not discussed in this thesis.

3.2.4 Systematic Protocol Slicing

When we consider horizontal and vertical behaviour reductions, it is observed that they are closely related. A horizontal reduction performed within a process body which causes the elimination of, for example, a choice branch, will in most cases result in the removal of another process completely which happens to be a vertical reduction. Diagonal reductions on the other hand must be performed more selectively within the bodies and definitions of the processes that are to be included in the final reduced specification. By using these relationships among the three types of reductions, a method (actually a heuristic) can be defined to perform behaviour reductions on input specifications based on some user defined criteria. The complicated nature of the application of all types of reductions requires human expertise, and thus it is hard to automate these transformations.

This section presents our method to perform simple behaviour reductions on base protocol specifications, which is applied to sample protocols in Chapter 5. First some definitions and assumptions about the method is given, and then the algorithm is presented.

3.2.4.1 Definitions

As defined in Chapter 2, a behaviour expression in LOTOS is built by applying operators to other behaviour expressions. The following definition introduces the notion of composite behaviour expression which is used as the main construct in the reduction algorithm.

Definition 3.1 A *composite behaviour expression* (CBE) within the body of a process in a LOTOS specification is recursively defined by the following rules:

1. Each of the following are composite behaviour expressions, and also the *base actions* of their respective CBEs.

    i) inaction (**stop**),
    ii) successful termination (**exit**),
    iii) process instantiation.

2. If B is a composite behaviour expression, then so is;

   i) $a;B$, where a is any LOTOS event structure, and,
   ii) $[t] \longrightarrow B$, where $[t]$ is a guard expression.

The composite behaviour expressions of a process are separated with the following LOTOS operators which are called horizontal constructors.

- binary choice ([]),
- parallel composition ($|[L]|$, $|||$),
- disabling ([>).

The following operators can be applied to a single composite behaviour expression or multiple composite behaviour expressions separated by the horizontal constructors.

- hide,
- generalized choice,
- let.

The following example illustrates the typical use of CBEs in LOTOS processes.

Example 3.1 Within the following process definition :

```
process WaitforICONresp2[ISAP,MSAP]:noexit:=
 ISAP?sp:SP[isICONresp(sp)];MSAP!MDATreq(CC);
 Dataphase_Res[ISAP,MSAP](succ(1))
[]
 MSAP?sp:MSP[isMDATind(sp)];WaitforICONresp2[ISAP,MSAP]
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* WaitforICONresp2 *)
```

The two behaviour expressions separated by the choice operator are composite behaviour expressions with base actions Dataphase_Res, and WaitforICONresp2, respectively.

The starting point for the subsequent discussion is a flattened LOTOS specification S which consists of a data part D and a behaviour part B, i.e, $S = \{D, B\}$. The behaviour part B is defined as the set of all process definitions in the specification, where the processes are uniquely identified and are statically independent from each other.

$$B = \{P_0, P_1, \ldots, P_n\} \text{ with, } \forall i, j \text{ and } i \neq j, P_i \neq P_j$$

Each process body is made up of one or more composite behaviour expressions separated by horizontal constructors. If a composite behaviour expression includes a guard, then the guard must be the first expression in the CBE. For the sake of simplicity the specification itself can also be considered as a process, the root process of the specification, and the behaviour part defines its body.

3.2.4.2 The Algorithm

Before the actual behaviour reduction procedure starts the following transformations must be carried out on the input base specification in order to obtain a flat specification.

1. Carry out the first phase of the chart construction algorithm on the input base specification.
2. Redefine all processes in the base specification so that they are uniquely identified and are statically independent from each other.
3. Transform the guards depending on interaction primitives which are not the first expressions in their respective CBEs, into selection predicates.

The first step is the application of the first phase of the chart construction algorithm given in Section 2.6. This results in the elimination of sequential and full parallel composition constructs from the specification. Then all processes are redefined with unique identifiers, and are made statically independent from each other, i.e., no process apart from the root process includes the definition of any other process in its 'where' clause. In the third step, all guards which depend on the parameters of interaction primitives, and which are not the first expressions in a composite behaviour expression are eliminated, and their boolean expressions are placed in the selection predicates of the preceding external events. The following example illustrates the third step.

Example 3.2 The LOTOS code given below is from the Initiator part of the INRES protocol specification.

```
process Disconnected_Ini[ISAP,MSAP,d2]:exit:=
 ISAP?sp:SP;
  ([isICONreq(sp)]-> MSAP!MDATreq(CR);d2!s(0);exit
  []
   [not(isICONreq(sp))]-> Disconnected_Ini[ISAP,MSAP,d2])
   (* User errors are ignored (Invalid/Inopportune Beh.) *)

 ...
endproc (* Disconnected_Ini *)
```

Since the the guards are not the first expressions of their respective CBEs, and they depend on the values of interaction variables, they are dropped and their boolean expressions are put into the selection predicates of the preceding events. Since there are two guard expressions, two event structures and selection predicates must be created. The result of this transformation is given below.

```
process Disconnected_Ini[ISAP,MSAP,d2]:exit:=
 ISAP?sp:SP[isICONreq(sp)];MSAP!MDATreq(CR);d2!s(0);exit
[]
 ISAP?sp:SP[not(isICONreq(sp))];
                          Disconnected_Ini[ISAP,MSAP,d2]
   (* User errors are ignored (Invalid/Inopportune Beh.) *)

 ...
endproc (* Disconnected_Ini *)
```

The main ingredient of the behaviour reduction algorithm given in Figure 3.1 is the test suite structure (TSS) of the relevant base protocol. By taking the specification corresponding to their parent nodes as input, the algorithm produces a smaller functional specification for each node of the TSS tree. In this way every node defines the base specification for its successor nodes. The behaviour reductions corresponding to the TSS nodes are performed according to the externally provided abstract constraints that are assumed to be defined for each node of the TSS tree. These constraints are abstract in the sense that

neither any specific requirements are placed on, nor any way of realizing them are explicitly given. They can be annotations used to distinguish different types of behaviour, names assigned to service primitives and PDUs, selection predicates, guard expressions, PICS parameters, etc.

The algorithm starts with the root process, and inspects its every composite behaviour expression. Any composite behaviour expression satisfying the constraints is included in the functionally reduced specification, and then the process instantiated by the base action of the CBE is put in a list for subsequent consideration for possible inclusion in the output reduced specification. The algorithm proceeds recursively until all of the CBEs of the processes are inspected. Since this procedure may disrupt the functionalities of some of the processes in the resultant specification, the last step is the correction of these functionalities according to the rules of LOTOS.

If the total number of processes in a specification is p and the average number of CBEs within a process is n, then the time complexity of the algorithm given above in the worst case is $\mathcal{O}(np)$. Line 1, and 2 of **Behaviour_Reduction** take time $\mathcal{O}(1)$ irrespective of the number of processes and CBEs. If we do not consider the time spent in the actual process of performing vertical, horizontal, and diagonal reductions because they depend on the abstract constraints, the time spent in **Reduce**, exclusive of the recursive call to itself, is $\mathcal{O}(1)$. Since line 2 can be executed at most n times for a process, and the total number of processes is p, the total time spent in **Reduce** is $\mathcal{O}(np)$. The last step of **Behaviour_Reduction** takes $\mathcal{O}(np)$ time, so the worst case time complexity of the whole algorithm is $\mathcal{O}(np)$.

A number of examples of the application of the above algorithm on some OSI protocol specifications can be found in Chapter 5.

3.3 Related Work

Specification selection is elaborated in [21] for test case generation from specifications based on labelled transition systems with infinite branching, using the implementation relation **conf**. By using a restriction operator on labelled transition systems which prunes all branches with labels from a specific set, the specified behaviour is constrained to be finite, and this results in finite number

Algorithm 3.1 Behaviour_Reduction

Inputs : Flattened Base Specification, TSS (expressed as a set of abstract constraints)
Output : Reduced Specification
Method : The methodology employed is based on extracting relevant behaviour expressions and including them in the resultant specification.

Procedure **Behaviour_Reduction**

1. Let **P** be the root process defining the behaviour of the base specification.
2. Define the body of process **P** in the reduced specification.
3. **Reduce(P)**.
4. Correct any functionality mismatch in the resultant specification.

Procedure **Reduce**(P:Process);

1. **For** each CBE within the process **P** of the base specification **do** the following steps :
   - 1.1. **If** the CBE satisfies the related abstract constraints **Then**
     - 1.1.1. Perform any **diagonal reductions** on the CBE according to the abstract constraints.
     - 1.1.2. Include the CBE in the reduced specification within the body of the process **P**
     - 1.1.2. **If** the base action associated with CBE is a non-recursive process instantiation **and** the instantiated process is not defined in the reduced specification **Then**
       - 1.1.2.1. Define the body of the instantiated process in the reduced specification.
       - 1.1.2.2. Perform any **diagonal reductions** on the definition of the process.
       - 1.1.2.3. Put the instantiated process in an Included List.
   - 1.2. **Else**
     - 1.2.1. Apply a **horizontal reduction** to the CBE.
     - 1.2.2. **If** the base action associated with the CBE is a process instantiation **Then**
       - 1.2.2.1. Apply a **vertical reduction** to the body of the instantiated process.
2. **For** each Process **P** in the Included List :
   - 2.1. **Reduce(P)**

Figure 3.1. Behaviour Reduction Algorithm

of test cases to be generated. The obtained results reveal that apart from avoiding overproduction of test cases, performing test selection by transforming the specifications has the advantage that information about the structure of the specification can be used in the selection process. A specification written as a behaviour expression has a certain structure in terms of how the specification is built from simpler behaviour expressions, such as a process is composed in parallel or in sequence with another process. If it can be assumed that this structure is reflected in the structure of the implementation, it can be used to guide the test selection procedure. For example, if a process is composed of two independently parallel processes, it may not be necessary to test all possible interleavings of actions of those processes. Such structure information is lost if selection is performed from a set of test cases.

Another related approach to specification selection can be found in [44], which uses the structure information of LOTOS specifications to develop a framework for deriving test cases to consider only parts of the behaviour description that corresponds to the chosen test purposes. In order not to loose important structure information that can serve to reduce the efforts for test case generation and for determining test purposes, a notation called *minimal-hierarchical specification* is introduced. A specification is minimal-hierarchical if it satisfies the orthogonality condition defined in [24], which requires independent architectural requirements to be specified by independent definitions; and it consists of the minimum number of processes which are necessary to express the intended behaviour of the specification. An algorithm is presented in [44] to convert an arbitrary LOTOS specification into an equivalent minimal-hierarchical specification in two steps, each step producing a specification as output that satisfies one of the conditions stated above.

The minimal-hierarchical specification is used in the derivation of the test cases while not considering the whole specification, but only parts of it corresponding to specific test purposes. One of the main differences with our approach arises here when a test purpose in the context of a minimal-hierarchical specification is defined as the successful execution of a subprocess of the specification. According to our approach it is not possible to isolate the behaviour corresponding to a specific test purpose within the body of a single process, because the desired functionality may be spread over the whole specification. Thus the whole specification must be considered when the intention is to identify specific functions.

Chapter 4

BASE SPECIFICATION DEVELOPMENT

In case of formal specification based protocol engineering the starting point is a formal specification of the protocol. Formal specifications are written following certain basic structuring principles [39]. Concurrency and multiplicity are for modelling multiple independent connections. Recurrence and sequentiality are for the use of subsequent connections on the same connection endpoint, and for the distinct phases that compose a connection, respectively. In LOTOS, concurrency/multiplicity is expressed as interleaved parallel composition; recurrence/sequentiality is mapped to recursion and successful termination with value passing. A language dependent structuring principle is the definition of constraints as processes, and constraint-oriented specifications in LOTOS use this feature extensively to express mutual satisfaction of the constraints by multi-way synchronized parallel composition.

This chapter consists of the applications of base specification development principles discussed in the previous chapter, on some protocols specified in LOTOS. The principles are built on the basic structuring principles given above. Section 4.1, introduces the INRES protocol, and the its base specification. Similarly Section 4.2 is about the development of the ACSE base specification from the respective base protocol.

PDU	Meaning	Parameter	Respective SPs
CR	Connection Establishment	none	ICONreq,ICONind
CC	Connection Confirmation	none	ICONresp,ICONconf
DT	Data Transfer	sequence number,ISDU	IDATreq,IDATind
AK	Acknowledgement	sequence number	
DR	Disconnection	none	IDISreq,IDISind

Table 4.1. INRES IPDUs

4.1 INRES Protocol

The first example that we will consider is the fairly simple OSI-like Initiator-Responder (INRES) protocol [37], for which the complete flattened LOTOS specification in state-oriented style can be found in Appendix A.

INRES is a connection-oriented protocol that operates between two protocol entities, Initiator and Responder. The protocol entities communicate by exchanging the protocol data units CR, CC, DT, AK, and DR. The meaning of the INRES PDUs (IPDUs) are given in Table 4.1.

The communication between the two protocol entities takes place in three distinct phases: the connection-establishment phase, the data transmission phase, and the disconnection phase. In each phase only certain PDUs are meaningful.

Connection-Establishment: A connection establishment is initiated by the user of the Initiator entity with an ICONreq service primitive. The entity then sends a CR to the Responder entity. Responder answers with CC or DR. In the case of CC, Initiator issues an ICONconf to its user, and the data transfer phase can be entered. If Initiator receives a DR from the Responder, the disconnection phase is entered. If the Initiator receives nothing at all within 5 seconds, CR is transmitted again. If, after four attempts, still nothing is received by the Initiator, it enters the disconnection phase.

When Responder receives a CR from the Initiator, it issues a ICONind to its user. The user can respond with ICONresp or IDISreq. ICONresp indicates the willingness of the user to accept the connection, Responder entity thereafter sends a CC to Initiator, and the data transfer phase is entered. Upon receipt of an IDISreq, Responder enters the disconnection phase.

Data Transfer : When the Initiator user issues an IDATreq primitive, the Initiator sends a DT to the Responder and is then ready to receive another IDATreq from the user. IDATreq has one parameter that is a service data unit (ISDU), which is the information that the user wants to transmit to the peer user. This user data is transmitted transparently by the Initiator protocol entity as a parameter of the protocol data unit DT. After having sent a DT to the Responder, Initiator waits for 5 seconds for a respective acknowledgement AK. If AK is not received, the DT is sent again. After four unsuccessful transmissions, Initiator enters the disconnection phase.

DT and AK carry a one-bit sequence number as a parameter. Initiator, after having entered the data transmission phase, starts with the transmission of a DT with sequence number 1. A correct acknowledgement of a DT has the same sequence number. After receipt of a correct acknowledgement, the next DT with the next sequence number can be sent. If Initiator receives an AK with incorrect sequence number, it sends the last DT once again. A DT can only be sent four times. Afterwards, Initiator enters the disconnection phase. Disconnection phase is also entered upon receipt of DR.

Following the establishment of the connection, Responder expects the first DT with sequence number 1. After the receipt of a DT with the expected number, Responder gives the ISDU as a parameter of an IDATind service primitive to its user and sends an AK with the same sequence number to the Initiator. A DT with an unexpected sequence number is acknowledged with an AK with the sequence number of the last correctly received DT. The user data of an incorrect DT is ignored. If Responder receives a CR, it enters the connection establishment phase, and upon receipt of an IDISreq from its user,it enters the disconnection phase.

Disconnection : An IDISreq from the Responder user results in the sending of a DR by the Responder. Afterwards Responder can receive another connection establishment attempt CR from Initiator.

At the Initiator side, the DR results in an IDISind to be sent to the user. An IDISind is also sent to the user after DT or CR have not been sent successfully to the Responder. Then a new connection can be established.

4.1.1 PICS Parameterization

INRES is an asymmetric protocol, i.e., one side, the initiator, can only establish connections and send data while the other side, the responder, can accept connections, release them and receive data. An implementation may either behave as the initiator or the responder entity, but not both. This fact is reflected in the PICS Proforma for INRES given in Appendix B.

For the parameterization of the base specification based on the PICS Proforma, "yes/no" entries related to the conditional capabilities can be taken into consideration as boolean entries. Entries indicating the range of values are not taken as parameters, and the following specification definition is obtained :

```
specification Inres_Protocol[ISAP,MSAP] (c1, c2 : Bool):noexit
(* c1, c2 : PICS parameters. *)
(* c1 : Initiator capability is supported *)
(* c2 : Responder capability is supported *)
```

The behaviour part, i.e, the main body of the specification starts with the static conformance review. It is expressed as an ADT function as follows :

```
behaviour

  (* Static Conformance Review. *)
  [CapabilityConform (c1, c2)] -> INRES [ISAP,MSAP](c1,c2)
```

Only if static conformance requirements are satisfied does the specification describe active behaviour, viz. it provides the implementor or tester with an abstract model of the protocol dynamic conformance requirements described by the process named INRES.

CapabilityConform is a boolean valued function which returns true if and only if exactly one of the roles mentioned in the PICS proforma (i.e., either initiator or responder) is implemented, and returns false otherwise. This is formally expressed in the type declarations part of the specification by defining a particular data type for this function.

```
type StaticConformance is Boolean
  opns CapabilityConform : Bool, Bool -> Bool
  eqns  forall c1, c2 : Bool
    ofsort Bool
      CapabilityConform (c1,c2) =
           ((c1 and not(c2)) or (not(c1) and c2))
endtype  (* StaticConformance *)
```

The parameters c1 and c2 can be used in the behaviour part as follows:

```
process INRES[ISAP,MSAP](c1,c2 :Bool) :noexit:=
  [c1]-> Initiator[ISAP,MSAP]
 []
  [c2]-> Responder[ISAP,MSAP]
endproc (* INRES *)
```

which divides the whole specification into two disjoint parts, the part specifying the behaviour of the initiator entity, and the part specifying the behaviour of the responder entity. Since the predicates c1 and c2 can not be both true at the same time for a particular implementation, either one of the paths defined by the guard expressions is selected, and therefore no deadlock possibility exists.

4.1.2 Behaviour Specification

The valid behaviour is specified based on the state tables given in the informal standard using traditional techniques. The state tables for the initiator and Responder protocol entities are given in Table 4.2 and Table 4.3, respectively. The state tables show the interrelationship between the state of the protocol machines, the incoming events that occur, the actions taken, and finally, the resultant state of the protocol machines. The intersection of an incoming event (row) and a state (column) forms a cell. Some cells contain predicate expressions comprising boolean variable 'p', which is equivalent to the expression 'c=4' ($\neg$ represents the boolean not). Blank cells represent the combination of incoming events and states that are not defined for the respective state machines, i.e, they represent inopportune behaviour.

State / **Event**	**STA0** **(Disconnected)**	**STA1** **(WaitforCC)**	**STA2** **(Connected)**	**STA3** **(Sending)**
ICONreq	CR c=1 STA1			
CC		ICONconf STA2		
IDATreq			DT c=1 STA3	
AK^+				STA2
AK^-				p:DT c=c+1 STA3 ¬p1:IDISind STA0
Timeout		p:CR c=c+1 STA1 ¬p1:IDISind STA0		p:DT c=c+1 STA3 ¬p1:IDISind STA0
DR	IDISind STA0	IDISind STA0	IDISind STA0	IDISind STA0

Table 4.2. State Table for Initiator Protocol

State / **Event**	**STA0** **(Disconnected)**	**STA1** **(WaitforICONresp)**	**STA2** **(Connected)**
CR	ICONind STA1		ICONind STA1
ICONresp		CC STA2	
DT^+			AK, IDATind STA2
DT^-			AK STA2
IDISreq	DR STA0	DR STA0	DR STA0

Table 4.3. State Table for Responder Protocol

The top level structure of the specification for each entity reveals the decomposition of the protocol into three distinct phases; namely connection establishment, data transfer, and disconnection.

```
process Responder[ISAP,MSAP]:noexit:=
 (hide d in
         Connectionphase_Res[ISAP,MSAP,d]
 |[d]|   d;Dataphase_Res[ISAP,MSAP](succ(1)))
 [>Disconnection_Res[ISAP,MSAP]
endproc (* Responder *)
```

Since a state-oriented specification style has been employed, each state of the protocol at each side is represented by a separate process. For example, the following piece of code describes the the behaviour of the Initiator protocol machine when it receives an Acknowledgement (AK) PDU in the Sending state. It inspects the sequence number of the incoming AK-PDU and if it is the expected number, the machine returns to its previous state where it is ready for another send-data request. If the number is incorrect according to the protocol, then the data is retransmitted and the current state is not changed.

```
process Sending[ISAP,MSAP]
  (z:DecNumb,number:Sequencenumber, olddata:ISDU):noexit:=

 MSAP?sp:MSP
 [isMDATind(sp) and not(isDR(data(sp))) and
  isAK(data(sp)) and (num(data(sp)) eq number)];
 Dataphase_Ini[ISAP,MSAP](succ(number))
[]
 MSAP?sp:MSP
 [isMDATind(sp) and not(isDR(data(sp))) and isAK(data(sp))
  and (num(data(sp)) ne number) and (z < 4)];
 MSAP!MDATreq(DT(number,olddata));
 Sending[ISAP,MSAP](s(z),number,olddata)
[]

...
endproc (* Sending *)
```

The default behaviour of the INRES protocol entities in response to invalid and inopportune behaviour at each state, represented by the blank entries in the respective state tables, is to ignore such inputs. In the base specification, this is specified by alternative choice branches at each state, where no action is taken when an invalid or inopportune PDU is received.

```
process WaitforICONresp1[ISAP,MSAP,d]:exit:=
  ...

[]
 MSAP?sp:MSP[isMDATind(sp)];WaitforICONresp1[ISAP,MSAP,d]
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* WaitforICONresp1 *)
```

4.2 ACSE Protocol

For the second example we consider the protocol specification for the Association Control Service Element (ACSE) [45]. The state-oriented LOTOS specification of the behaviour part of this protocol is given in Appendix D.

ACSE is defined for the purpose of the management of application associations. It provides facilities for the establishment and release of application associations between Application Entities (AEs). Association is the term used for the connections established at the Application layer. An application association is a presentation connection with additional application layer semantics. There is a one-to-one mapping between application associations and presentation connections. Because the sole purpose of ACSE is to manage application associations, it does not provide any data transfer service elements.

The ACSE Protocol Machine (ACPM) operates in either the normal mode or the X.410-1984 mode. When operating in the X.410-1984 mode, the ACPM does not exchange any ACSE APDUs with its peer. The following discussion and the base specification in Appendix D assumes that the ACPM operates in the normal mode.

The ACPM is driven by the receipt of input events from its ACSE service user or the presentation service provider. It uses five APDUs (Table 4.4), and three procedures. These procedures are association establishment, normal

PDU	Name
AARQ	A-ASSOCIATE.request PDU
AARE	A-ASSOCIATE.response PDU
RLRQ	A-RELEASE.request PDU
RLRE	A-RELEASE.response PDU
ABRT	A-ABORT.request PDU

Table 4.4. ACSE APDUs

release, and abnormal release.

Association Establishment: A.ASSOCIATE service element is used in by an AE to establish an application association with a peer AE. This service element uses over thirty parameters. Some of them are described below.

- **Application Context Name:** This parameter identifies the application context used for the application association. An application context defines the rules governing the communication of the two AEs used for the entire application association. The initiating ACSE user first proposes an application context. The accepting user returns either the same or a different one. If the initiator can not operate in the acceptor's application context, it issues an A.ABORTrequest primitive.
- **mode:** This parameter specifies the mode in which the ACSE service will operate for the association. Its value can be either normal or X.410-1984.
- **AP Titles/Qualifiers:** An application entity is identified by AET (application entity title) which consists of APT (application process title) and an AE qualifier.
- **Result and Result Source:** The result parameter is provided by either the acceptor, the ACSE service-provider, or the presentation service provider. Its value can be "accepted", "rejected(permanent)", or "rejected(transient)". The result source parameter identifies the source of the result parameter, and the diagnostic parameter, if present. The value of the result source parameter can be either "ACSE service-user", "ACSE service-provider", or "presentation service-provider".

- **Diagnostic:** This parameter is used only if the result parameter has the value "rejected". It gives the reason behind the rejection of the association.

- **User information:** Either the initiating AE or the responding AE may include the user information. Its meaning depends on the application context that accompanies the request.

The AARQ APDU is used in the association establishment procedure. Upon receiving an A.ASSOCIATErequest primitive from its ACSE service user, the requesting ACPM forms an AARQ APDU. The AARQ APDU is then sent as the user data parameter of a P.CONNECTrequest primitive. When the accepting ACPM receives an AARQ APDU, it checks if the AARQ APDU is acceptable syntactically. If not, the association establishment procedure is disrupted and no A.ASSOCIATEindication is issued. If the accepting ACPM does not support the protocol version requested by the initiating ACPM, it will respond with an AARE APDU and indicate the result value of "rejected(permanent)". When the accepting ACPM receives an A.ASSOCIATEresponse primitive, the result parameter should specify whether its user has accepted or rejected the association. The accepting ACPM forms an AARE APDU and sends it to the requesting ACPM using a P.CONNECTresponse primitive. If its user accepted the association request, the accepting ACPM sets the result parameter of P.CONNECTresponse to "acceptance", otherwise it is set to "user-rejection" to indicate that its user has rejected the association.

Normal Release: An application association can be released in an orderly manner by means of **A.RELEASE** service element. The RLRQ and RLRE APDUs are used in the normal release of an association. When an ACPM receives a A.RELEASErequest primitive, it forms a RLRQ APDU and sends it as the user data of a P.RELEASErequest primitive. When the accepting ACPM receives an A.RELEASEresponse primitive, it forms a RLRE APDU and sends it as the user data of a P.RELEASEresponse primitive. Thus, the purpose of RLRE is to acknowledge the received RLRQ.

Abnormal Release: If an AE detects an unrecoverable error, it uses **A.ABORT** service element to abort an application association with possible loss of data that are in transit. The ABRT APDU is used in the abnormal release procedure There are only two fields in an ABRT APDU: abort

source (mandatory) and user information (optional). When an ACPM receives A.ABORTrequest, it forms an ABRT APDU with the abort source field set to "ACSE service user". It then sends the ABRT APDU as the user data of a P-U.ABORTrequest primitive to release the association. When an ACPM detects a protocol error, it issues an A.ABORTindication primitive to its service user, forms an ABRT APDU with the abort source field set to "ACSE service provider", and sends it as the user data of a P-U.ABORTrequest primitive.

The ACSE service provider can also abort an application association using **A-P.ABORT**. A-P.ABORT is a pass through service element from the Presentation layer, i.e., it is semantically identical to P-P.ABORT. When an ACPM receives a P-P.ABORTindication primitive, the ACPM issues an A-P.ABORTindication primitive to its user, and the association is released.

4.2.1 PICS Parameterization

[46] defines the PICS proforma for ACSE. Apart from the standard entries identifying the implementation, the implementor, and the entries indicating the range of PDU parameters, there are a number of boolean entries indicating the implementation of optional and conditional capabilities. The five conditional entries which are the possible candidates in parameterizing the base specification are:

1. c1 : Association initiator capability is supported,
2. c2 : Association responder capability is supported,
3. c3 : The implementation can behave as the requestor of the release of the association,
4. c4 : The underlying session protocol supports Version 2,
5. c5 : The described ACSE protocol version is greater than 1.

Since the test suite structure developed in [47] on which the behaviour reductions given in Chapter 5 are based, does not take into account the different types of behaviour resulting from the choice of options c4 and c5; only c1, c2 and c3 are used in parameterizing the base specification as shown below.

```
specification ACSE_Protocol[A,P] (c1,c2,c3 : Bool): noexit
(* c1, c2, c3 : PICS parameters. *)
(* c1 :  Association initiator capability is supported *)
(* c2 :  Association responder capability is supported *)
(* c3 :  RLRQ APDU is supported for transmission       *)
```

The static conformance review part is the same as that of INRES, whereas the ADT function checking the satisfaction of the static conformance requirements is slightly different reflecting the fact that ACSE is not an asymmetric protocol.

```
type StaticConformance is Boolean
  opns CapabilityConform :  Bool, Bool -> Bool
  eqns forall c1, c2 : Bool
    ofsort Bool
      CapabilityConform(c1, c2) = (c1 or c2)
endtype (* StaticConformance *)

behaviour

 (* Static Conformance Review *)
 [CapabilityConform (c1, c2)] -> ACSE[A,P](c1,c2,c3)
```

A major difference between the protocol specifications of ACSE and INRES is that, the implementations of the ACSE protocol can behave both as initiator and responder of the associations. Since ACSE is not an asymmetric protocol, the use of the parameters related to the role of the implementation, i.e, c1 and c2, in the behaviour part is quite different when compared with INRES. This time c1 and c2 can not be defined as guard expressions, because in the case that they are both true for a particular implementation, according to the semantics of LOTOS, one of the paths can be chosen nondeterministically before engaging in an interaction. This brings in the possibility of the protocol entity running into deadlock in case of an interaction request for which it is not ready for. For this reason, boolean parameters which do not represent the choice among mutually exclusive options must be used in selection predicates, not in guard expressions, while specifying alternative behaviour, as shown below:

```
process Unassociated[A,P](c1,c2,c3 : Bool):exit:=
 A ? x : primitive [IsAASCreq(x) and c1];
```

```
   ...

 []
  P ! Input ? x : primitive
  [IsPCONind(x) and IsAARQ(user_data(x)) and
   not(common_prot_version(get_AARQ(user_data(x)))) and c2];

   ...
endproc (* Unassociated *)
```

4.2.2 Behaviour Specification

As an Application layer protocol, the PDU structures of ACSE are defined in ASN.1. Therefore transformation of PDU type definitions from ASN.1 into the data type definition language ACT-ONE of LOTOS has to be done in order to obtain a complete base specification. The transformation scheme adopted is based on the approach developed in [48]. The size of the resulting data type definitions is much larger than the original ASN.1 definitions. Since ASPs and PDUs for Application layer protocols contain a large number of parameters, instead of writing the same interaction primitives every time within the behaviour part, they are defined in the data type part of the LOTOS specification as ADT functions, and names assigned to these functions are used in the behaviour part. The following example illustrates the behaviour of the ACSE protocol machine when it has been requested to initiate an association.

```
process Unassociated[A,P,d](c1,c2,c3 : Bool):exit:=
 A ? x : primitive [IsAASCreq(x) and c1];
 P ! Out ! PCONreq(make_AARQ(get_AASCreq(x)));
 awaitAARE[A,P,d](c1,c2,c3)
[]

  ...
endproc (* unassociated *)
```

The ADT function PCONreq(make_AARQ(get_AASCreq(x))) represents the formation of the AARQ PDU by using the parameters of the input service primitive AASCreq, and enclosing it in the user data of the P.CONNECTindication

primitive. The actual definition of the function is placed in the data type part.

```
type  primitive is
      AASCreq, AASCind, AASCrsp, AASCcnf, ARLSreq, ARLSind,


  opns
     ...
     PCONreq     : ACSE_apdu -> primitive
     ...
     make_AARQ   : AASCreq   -> ACSE_apdu


  eqns
    ...
     (****  make_AARQ ****)
     (* AASCreq -> ACSE_apdu *)

     ofsort ACSE_apdu
     forall AASC : AASCreq

     make_AARQ(AASC) = ACSE_apdu(AARQ_apdu(
        Bit(1),app_context_name(AASC),
        called_ap_title(AASC),
        called_ae_qualifier(AASC),
        called_ap_invocation_id(AASC),
        called_ae_invocation_id(AASC),
        calling_ap_title(AASC),
        calling_ae_qualifier(AASC),
        calling_ap_invocation_id(AASC),
        calling_ae_invocation_id(AASC),
        type_genere010(Not_Present),
        user_info(AASC)
    ))
   ...
    ofsort ACSE_apdu
    forall apdu : ACSE_apdu
```

```
      user_data(PCONreq(apdu)) = apdu;
    ...
  endtype (* primitive *)
```

According to [47], since each ACSE PDU is carried as the user data of a different presentation service primitive, and the PDU and the presentation service primitive carrying it together form the protocol, no inopportune tests are defined for ACSE, but only invalid tests. In each state, only one PDU as the user data of a specific presentation service primitive is considered to be valid, so combinations of the same presentation service primitive with other ACSE PDUs are considered to be invalid protocol behaviour. The code given below is specifies the behaviour of the protocol to invalid inputs in the associated state, which is represented by the process 'associated'. According to ACSE protocol definition, when an unexpected PDU is received, the abnormal release procedure is invoked. The abnormal release procedure is represented by the process 'protocol_error' in the base specification.

```
process Associated[A,P](c1,c2,c3 : Bool):exit :=
 ...

[]
 P ! Input ? x : primitive
 [IsPRLSind(x) and not(IsRLRQ(user_data(x)))];
 protocol_error[A,P](c1,c2,c3)
 (* Invalid Behaviour *)
 ...
endproc (* Associated *)

process protocol_error[A,P](c1,c2,c3 : Bool):noexit :=
 A ! make_AABRind;
 P ! Out ! PUABreq(make_ABRT);
 ACSE[A,P](c1,c2,c3)
endproc (* protocol_error *)
```

Chapter 5

DERIVATION OF FUNCTIONAL SPECIFICATIONS

An important use of base specifications is the systematic derivation of functional specifications. In this chapter, the slicing approach developed in Chapter 3 to the derivation of functional specifications is applied to sample base protocols. First state-oriented specifications are considered. In Section 5.1 functional specifications of INRES and ACSE protocols are obtained for each of the test groups defined on their respective test suite structures. Section 5.2 considers the more complicated transport protocol, and illustrates the application of diagonal reductions along with horizontal and vertical reductions on the base specification of this protocol.

The kind of behaviour reductions to be carried out depend on the test groups defined on the related test suite structure. As will be evident from the application examples given in this chapter, most of the test suites designed for OSI protocols are subdivided into the following test groups at the highest level:

- Basic Interconnection Tests,
- Capability Tests,
- Valid Behaviour Tests,
- Inopportune Behaviour Tests,
- Invalid Behaviour Tests.

Test cases for basic interconnection and capability tests can be obtained by selecting tests from valid behaviour group. Therefore, at the highest level, it is reasonable to consider only valid and invalid/inopportune behaviour tests, while decomposing base specifications. So, generally, the first step in behaviour reductions is the extraction of the parts from the specifications which define valid and invalid/inopportune behaviour. The test groups related to valid and invalid/inopportune behaviour tests are generally divided further into specific test groups depending on the nature of the base protocol in question. Behaviour reductions are performed for all test groups of the test suite structure.

5.1 Behaviour Reductions On State-Oriented Specifications

5.1.1 Behaviour Reductions On INRES

Figure 5.1 shows the test suite hierarchy of INRES protocol. Appendix C contains the full test suite structure and test purposes document. INRES is an asymmetric protocol in that an implementation can function as an initiator or as a responder, each role being different from each other. Since this is a major consideration of an implementation it follows that this division forms the top level of the structure for the test suite. Accordingly, the first application of the behaviour reduction algorithm is to obtain separate protocol specifications for each of the roles.

Before applying the behaviour reduction algorithm given in Chapter 3 to the base specification, some syntactic transformations have to be carried out. These are the first phase of the chart construction, redefinition of the processes so that they are syntactically independent form each other, and elimination of the guards which are not the first expressions in their respective CBEs. Appendix A contains the base specification of INRES obtained after having performed these transformations.

The two protocols corresponding to each of the roles can be obtained by performing a horizontal reduction inside the process named 'INRES', and then a vertical reduction to exclude all process definitions which do not describe the intended behaviour. For example, the specifications of the Initiator and

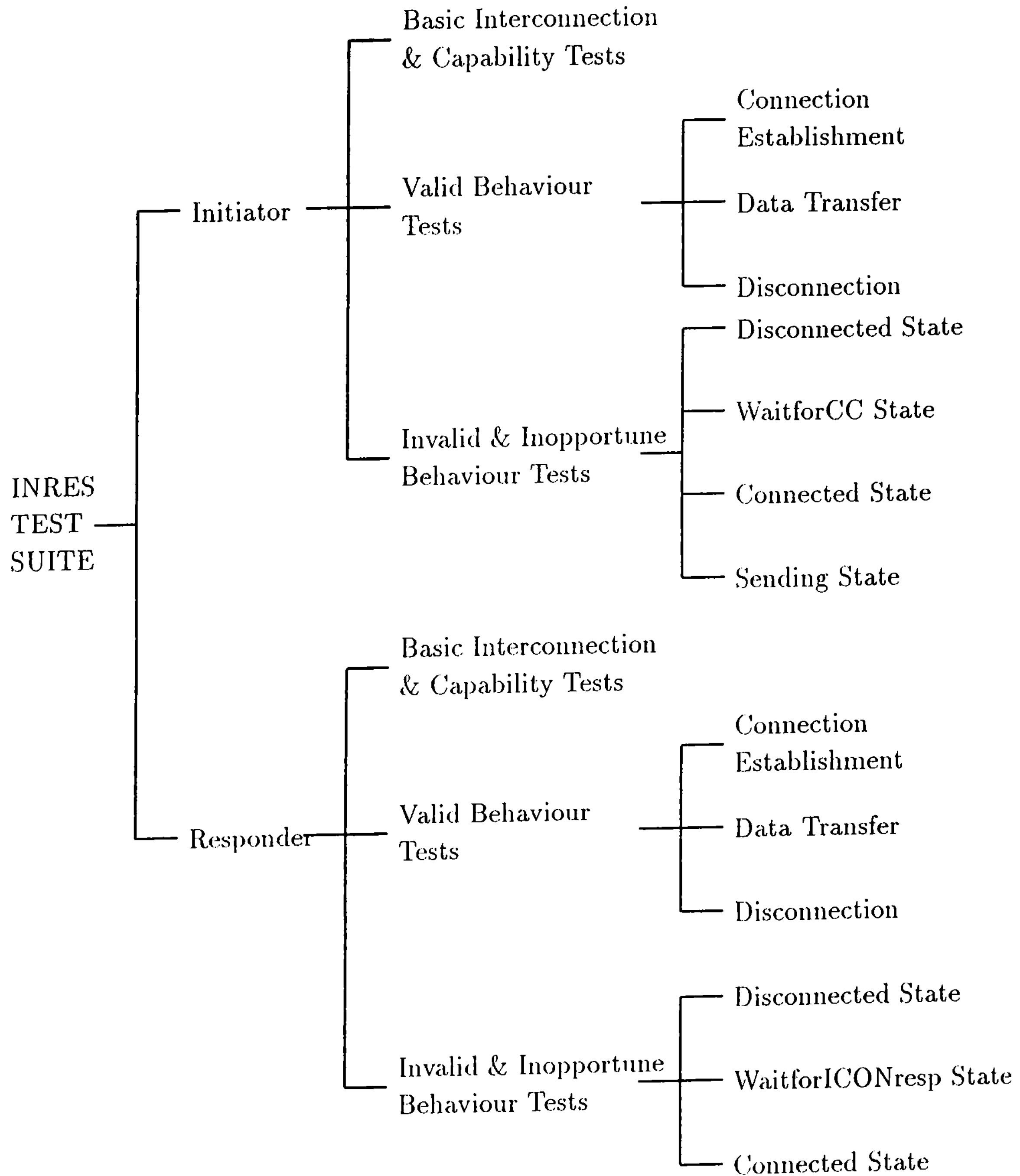


Figure 5.1. Test Suite Structure for INRES

Responder protocols can be obtained by eliminating the branches conditioned by the PICS parameters 'c2' and 'c1', respectively. Since when a process is vertically reduced from the specification its body is not considered, if they are not instantiated elsewhere ,all other processes which are directly or indirectly instantiated by the reduced process are omitted, too. Since PICS parameters differentiating between the two roles are not needed any more, they can be dropped. The resulting specifications corresponding to the two separate protocols have the following top level structure:

```
specification Inres_Protocol_I[ISAP,MSAP]:noexit

 (* Data types *)

 behaviour

  INRES [ISAP,MSAP]

  where

   process INRES[ISAP,MSAP]:noexit:=
    Initiator[ISAP,MSAP]
   endproc (* INRES *)
   ...
endspec

specification Inres_Protocol_R[ISAP,MSAP]:noexit

 (* Data types *)

 behaviour

  INRES [ISAP,MSAP]

  where

   process INRES[ISAP,MSAP]:noexit:=
    Responder[ISAP,MSAP]
   endproc (* INRES *)
```

```
  ...
endspec
```

After extracting out the behaviour specific to a particular role, valid behaviour and invalid/inopportune behaviour descriptions must obtained as two separate specifications. The reduced specifications for valid and invalid/inopportune behaviour can be produced by evaluating the annotations distinguishing the two types of behaviour, and performing horizontal reductions based on these annotations. The specification of valid behaviour can be obtained by omitting the branches describing invalid/inopportune behaviour. Since INRES takes no action in response to invalid/inopportune test events, and does not change its current state, such behaviour must be augmented with some valid behaviour expressions in order to end up with an executable specification. The following LOTOS code gives part of the specification of the invalid/inopportune behaviour of Initiator entity.

```
specification Inres_Protocol_I_BIO[ISAP,MSAP]:noexit


   process Connectionphase_Ini[ISAP,MSAP,d]:exit:=
    hide dd in
            Disconnected_Ini[ISAP,MSAP,dd]
    |[dd]|  dd?z:DecNumb;WaitforCC[ISAP,MSAP,d](z)
   endproc (* Connectionphase_Ini *)

   process Disconnected_Ini[ISAP,MSAP,dd]:exit:=
    ISAP?sp:SP[isICONreq(sp)];MSAP!MDATreq(CR);dd!s(0);exit
    (* CBE added to provide with functionality exit *)
   []
    ISAP?sp:SP[not(isICONreq(sp))];
    Disconnected_Ini[ISAP,MSAP,dd]
    (* User errors are ignored (Invalid/Inopportune Beh.) *)
   []
    MSAP?sp:MSP[isMDATind(sp) and not(isDR(data(sp)))];
    Disconnected_Ini[ISAP,MSAP,dd]
    (* DR is only accepted by process Disconnection *)
    (* System errors are ignored (Invalid/Inopportune Beh.) *)
   endproc (* Disconnected_Ini *)
```

```
  ...
endspec
```

The reduced specification corresponding to the valid behaviour tests is subdivided into three test groups each corresponding to one of the phases of the protocol, namely connection establishment, data transfer, and disconnection. Accordingly, the corresponding slices are obtained for each phase. A specification for each phase can be obtained by considering the modular structure of the specification. By performing horizontal and vertical reductions, a separate protocol for each phase can be obtained. Below is the specification of the valid behaviour of the initiator entity during disconnection phase. It is obtained from the respective base specification by horizontally reducing the instantiations of the processes 'Connectionphase_Ini' and 'Dataphase_Ini' from the body of the process 'Initiator'; and vertically reducing their definitions from the specification. In this case the abstract constraints refer to the names of the processes.

```
specification Inres_Protocol_I_BV_DC[ISAP,MSAP]:noexit

 (* Data types *)

 behaviour

  INRES [ISAP,MSAP]

  where

   process INRES[ISAP,MSAP]:noexit:=
    Initiator[ISAP,MSAP]
   endproc (* INRES *)

   process Initiator[ISAP,MSAP]:noexit:=
    Disconnection_Ini[ISAP,MSAP]
   endproc (* Initiator *)

   process Disconnection_Ini[ISAP,MSAP]:noexit:=
    MSAP?sp:MSP[isMDATind(sp) and isDR(data(sp))];ISAP!IDISind;
```

```
    Initiator[ISAP,MSAP]
   endproc (* Disconnection_Ini *)

endspec
```

The subsequent division under the invalid/inopportune behaviour test group is state based, i.e., the behaviour of the protocol entity is tested for invalid and inopportune behaviour in each state. The respective slices correspond to the behaviour described by each of the processes in the base specification. Below is the specification of the initiator entity in response to invalid/inopportune behaviour in its connected state.

```
specification Inres_Protocol_I_BIO_STA2[ISAP,MSAP]:noexit

 (* Data types *)

 behaviour

  INRES [ISAP,MSAP]

  where

   process INRES[ISAP,MSAP]:noexit:=
    Initiator[ISAP,MSAP]
   endproc (* INRES *)

   process Initiator[ISAP,MSAP]:noexit:=
    hide d in
           d;Dataphase_Ini[ISAP,MSAP] (succ(0))
   endproc (* Initiator *)

   process Dataphase_Ini[ISAP,MSAP]
           (number : Sequencenumber):noexit:=
    hide d in
           Connected_Ini[ISAP,MSAP,d](number)
           (* 1 is the first Sequencenumber *)
   endproc (* Dataphase_Ini *)
```

```
process Connected_Ini[ISAP,MSAP,d]
        (number:Sequencenumber):noexit:=
 ISAP?sp:SP[not(isIDATreq(sp))];
 Connected_Ini[ISAP,MSAP,d](number)
 (* User errors are ignored (Invalid/Inopportune Beh.) *)
[]
 MSAP?sp:MSP[isMDATind(sp) and not(isDR(data(sp)))];
 Connected_Ini[ISAP,MSAP,d](number)
 (* DR is only accepted by process Disconnection *)
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* Connected_Ini *)

endspec
```

5.1.2 Behaviour Reductions On ACSE

Figure 5.2 shows the test suite hierarchy defined for the ACSE protocol which is obtained from the complete test suite structure and test purposes document given in [47]. Since the base specification developed describes only the normal mode operation of the protocol, the parts related to the X-410.1984 mode of the protocol are omitted from the test suite structure.

Appendix D contains the flattened base specification of the ACSE protocol from which the functional specifications are obtained. As in the case of INRES, functional specifications corresponding to valid and invalid behaviour are obtained by considering each CBE in each of the processes. Since CBEs defining invalid behaviour are already distinguished by comments, each specification can be obtained by eliminating the irrelevant choice branches horizontally. The subsequent division under the invalid behaviour test group is state based. The code below is the specification of the invalid behaviour of ACSE protocol machine in its associated state. A diagonal reduction is applied to the event structure at internal gate **d** within the process 'ACSE', and also to the first parameter of the process 'Norm_Rel'. PICS parameters have been dropped, too.

```
specification ACSE_Protocol_BI_STA5[A,P]:noexit
```

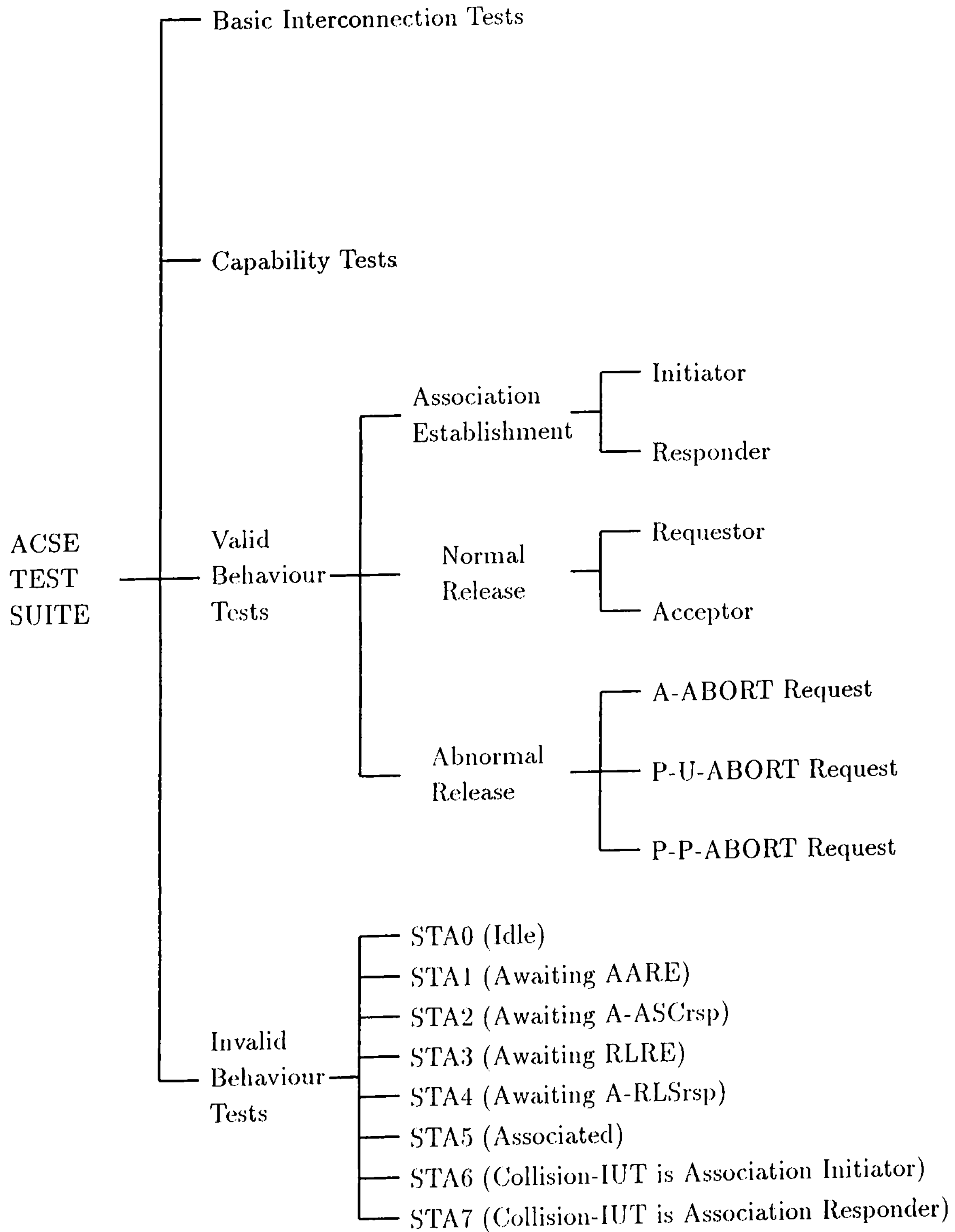


Figure 5.2. Test Suite Structure for ACSE

```
(* Data Types *)

 behaviour

  ACSE[A,P]

  where

   process ACSE[A,P]:noexit:=
    hide d in
           d;(Normal_Rel[A,P]
             [> Abort[A,P])
   endproc (* ACSE *)

   process Normal_Rel[A,P]:noexit:=
    hide d in
           Associated[A,P,d]
   endproc (* Normal_Rel *)

   process Associated[A,P,d]:noexit:=
    P ! Input ? x : primitive
    [IsPRLSind(x) and not(IsRLRQ(user_data(x)))];
    protocol_error[A,P]
    (* Invalid Behaviour *)
   endproc (* Associated *)

   process Abort[A,P]:noexit:=
    P ! Input ? x : primitive
    [IsPUABind(x) and not(IsABRT(user_data(x)))];
    protocol_error[A,P]
    (* Invalid Behaviour *)
   endproc (* Abort *)

   process protocol_error[A,P]:noexit:=
    A ! make_AABRind;
    P ! Out ! PUABreq(make_ABRT);
    ACSE[A,P]
   endproc (* protocol_error *)
```

```
endspec
```

The reduced specification corresponding to the valid behaviour tests is first subdivided into three tests groups each corresponding to one of the phases of the operation of the protocol, and then to individual test groups according to specific roles or service primitives. A specification for each phase of the protocol can be obtained by considering the modular structure of the specification, and a specification for each role can be extracted by performing the reductions based on the PICS parameters. In other words the PICS parameters serve as the abstract constraints in this case. The specification given below is for the valid behaviour of the ACSE entity as the acceptor in the normal release phase of the protocol. Based on the PICS parameter 'c3', all but one of the choice branches within the process 'Associated' are horizontally reduced, and their definitions are vertically reduced from the resultant specification.

```
specification ACSE_Protocol_BV_NR_AC[A,P]:noexit

(* Abstract Data Types *)

 behaviour

  ACSE[A,P]

  where

   process ACSE[A,P]:noexit:=
    hide d in
           d;Normal_Rel[A,P]
   endproc (* ACSE *)

   process Normal_Rel[A,P]:noexit:=
    hide d in
           Associated[A,P,d]
   |[d]|   d;AwaitARLSrsp[A,P]
   endproc (* Normal_Rel *)

   process Associated[A,P,d]:exit:=
```

```
    P ! Input ? x : primitive
    [IsPRLSind(x) and IsRLRQ(user_data(x))];
    A ! make_ARLSind(get_RLRQ(user_data(x)));
    d;exit
   endproc (* Associated *)

   process AwaitARLSrsp[A,P]:noexit:=
    A ? x : primitive
    [IsARLSrsp(x) and (result(get_ARLSrsp(x)) eq affirmative)];
    P ! Out ! PRLSrspA(make_RLRE(get_ARLSrsp(x)));
    ACSE[A,P]
   []
    A ? x : primitive
    [IsARLSrsp(x) and (result(get_ARLSrsp(x)) eq negative)];
    P ! Out ! PRLSrspR(make_RLRE(get_ARLSrsp(x)));
    Normal_Rel[A,P]
   endproc (* AwaitARLSrsp *)

endspec
```

5.2 Behaviour Reductions On Constraint-Oriented Transport Protocol Specification

5.2.1 Introduction To The Transport Protocol And Its Base Specification

The purpose of the transport layer is to provide transparent and reliable data transfer between transport service users. Like all other connection-oriented protocols, the operation of the transport protocol consists of three distinct phases: connection-establishment, data transfer, and connection release.

A transport service user uses **T-CONNECT** service element to set up a full duplex transport connection with its peer. During transport connection establishment, the two users and the transport service provider can negotiate the quality of the service to be provided. There are two data transfer

PDU	Name
CR	connection request
CC	connection confirm
DR	disconnect request
DC	disconnect confirm
DT	data
ED	expedited data
AK	data acknowledge
EA	expedited acknowledge
RJ	reject
ERR	TPDU error

Table 5.1. Transport TPDUs

service elements, **T-DATA**, and **T-EXPEDITED DATA**. **T-DATA** service element delivers the data reliably between transport users. If the expedited data option has been selected during connection establishment, then the **T-EXPEDITED-DATA** service element is used to convey expedited data, i.e., data which is not subject to normal flow control restrictions. After a connection has been established, either the service users or the provider may use the **T-DISCONNECT** service element to release the transport connection. Once this service is invoked, any TSDU in transit may be lost. Thus T-DISCONNECT is destructive. It can also be used for connection rejection by either the transport service user or the called user. The transport standard defines a total of ten TPDUs which are explained in Table 5.1.

If the underlying network is fairly reliable, then the transport protocol that is required to accomplish the data transfer does not need to do much work. But if the underlying network is unreliable, then some elaborate transport protocol mechanisms are required to cope with the deficiency. ISO defined three types of network services:

- **Type A Network Service:** Type A network service is essentially perfect. The fraction of packets that are lost, duplicated, or garbled is negligible. Network resets are so rare that they can be ignored.
- **Type B network service:** Type B network service provides network connections with an acceptable residual error rate but an unacceptable

signalled failure rate. Residual errors are those that are not corrected, and for which the transport service is not notified. On the other hand, a signalled failure is a failure detected by the network layer which then signals the transport entity for recovery.

- **Type C network service:** Type C network service is not reliable enough to be trusted at all; residual error rate is unacceptable. These networks do not detect the errors resulting from the loss, duplication, re-ordering and corruption of data.

Based on the three kinds of network services, ISO defines five transport classes:

- **TP 0:** TP 0 provides the simplest protocol mechanism to support a Type A network.
- **TP 1:** TP 1 provides a connection with minimal service to recover from network signalled failures.
- **TP 2:** TP 2 is basically an enhancement to TP 0, and permits multiplexing of transport connections, i.e., more than one transport connection can be provided by using a single network connection. It is used to support Type A networks.
- **TP 3:** TP 3 is basically a combination of TP 1 and TP 2. It allows an explicit flow control and has the ability to recover from a network failure. It is also used to support a Type B network.
- **TP 4:** TP 4 is designed for Type C network service. It is the most sophisticated transport protocol. It must be able to handle lost, duplicate, and garbled packets, as well as network failures.

When a transport entity processes an event, it will call a transport procedure. The transport standard defines a total of 23 procedures. Each transport class uses only a subset of these procedures.

- **Assignment to a network connection :** This procedure, common to all classes, assigns either an existing or a new connection to a new transport connection.

- **Transfer of TPDUs :** This procedure, common to all classes, uses the normal and expedited data services provided by the Network layer to transfer TPDUs.
- **Segmentation and reassembly :** Because of system constraints, a sending transport entity (TE) may need to segment a TSDU into an ordered sequence of TPDUs. Reassembly is performed by the receiving TE.
- **Connection establishment :** In all transport classes but TP 4, a two way handshake protocol is used for connection establishment. This involves the sending of a CR TPDU by the initiating TE followed by the sending of a CC TPDU by the responding TE.
- **Implicit normal release :** This procedure applies to TP 0 only. A transport connection is released implicitly by releasing the underlying network connection. No TPDUs are exchanged in this procedure.
- **Error release :** When a signalled failure or a disconnect indication is received from the network service provider, the simple protocol mechanism used by TP 0 and TP 2 simply releases the transport connection without providing any recovery actions.
- **Association of TPDUs with transport connections :** Whenever a TE receives a TPDU from the Network layer, it will map the TPDU to an appropriate transport connection.
- **Treatment of protocol errors :** If the received TPDU can not be mapped to a transport connection, it is considered to be a protocol error.
- **Concatenation and separation :** The purpose of concatenation is to improve efficient use of a network connection. In this procedure, a number of TPDUs are concatenate into a single NSDU for transmission, and later on separated by the receiving TE.
- **Explicit normal release :** The explicit normal release procedure is used by a TE to terminate a transport connection. It involves the exchange of DR and DC TPDUs between the TEs.
- **Numbering of DT TPDUs :** To facilitate the use of synchronization, flow control, and resequencing procedures, it is necessary for each DT TPDU carry a sequence number.

- **Expedited data transfer** : This procedure places expedited user data into the data filed of ED TPDU. All transport classes but TP 1 use the normal data transfer service to deliver expedited transport data.

- **Reassignment after failure** : When a TE receives a network signalled failure, it calls this procedure to take care of the problem. The result is that the transport connection is assigned to a different network connection. When the reassignment is achieved , the resynchronization procedure is invoked.

- **Retention until acknowledgement of TPDUs** : This procedure, applying to TP1, TP3, and TP4, provides a mechanism for the sending TE to retain copies of the TPDUs which were sent, until it receives an acknowledgement. Should no acknowledgement be received before a certain period of time has elapsed, the unacknowledged TPDUs are retransmitted.

- **Resynchronization** : This procedure is used by TP 1 and TP 3 to restore a transport connection upon receipt of an N.RESETindication from the Network layer.

- **Frozen references** : This procedure is used to prevent re-use of source/destination references because the TPDUs associated with the old references may still exists somewhere in the network.

- **Multiplexing and demultiplexing of transport connections** : The multiplexing procedure allows multiple transport connections to share a single network connection. The receiving TE must perform demultiplexing.

- **Explicit flow control** : This procedure is used to regulate the flow of DT TPDUs between the TEs. This flow control is independent of the flow control present in the Network layer.

- **Checksum** : This procedure is mandatory for the CR TPDU which is used during the connection establishment phase. Its use is optional during data transfer phase.

- **Retransmission on timeout** : A sending TE uses a local retransmission timer to determine the appropriate time to retransmit an unacknowledged TPDU.

- **Resequencing :** This procedure is used to sort any misordering of DT TPDUs which may be caused by the underlying network.
- **Inactivity control :** This procedure deals with the unsignalled termination of a network connection or the failure of the peer TE (half open connections). It is invoked upon the expiration of an inactivity timer.
- **Splitting and recombining :** To achieve a higher throughput or a greater resilience against network failures, this procedure allows a transport connection to be assigned to multiple network connections.
- **Three way handshake connection establishment :** Unlike other transport classes which use a two way handshake for connection establishment, TP 4 uses a three way handshake. The only addition is that after receiving a CC, the initiator must respond with DT (ED) or AK (RJ) TPDUs.

The transport base protocol specification considered in this study is the constraint-oriented specification given in [49]. It describes the transport protocol, (Classes 0, 1, 2, 3) using the formal description technique LOTOS. Full account is taken of the multiplicity aspects of the protocol, and the behaviour of a never terminating transport entity capable of supporting multiple connections, is described. Encoding related behaviour specification and data types are also included.

According to the definition of the specification, some of the component processes describe constraints that apply to, and depend upon, the behaviour of the protocol entity at only one of the two service service boundaries (i.e., transport and network service boundaries). This class of constraints are referred to as *service constraints*, whilst the term *protocol constraints* refer to those which are described by the other components. The service constraints ensure, for instance, that the identification of a connection by means of a connection endpoint identifier is unique within the scope of any given address. Since the provision of multiple connections, and encoding related behaviour is not an immediate concern while deriving the abstract test cases, some transformations have been applied to the base specification, in order to eliminate the behaviour related to the provision of multiple connections and TPDU encoding. The resulting structure of the base specification is as follows:

```
specification TransportProtocolEntity[t,n]
              (tpeo : TPEOptions):noexit

(* Global Type Definitions *)

 behaviour

  [Conform(tpeo)] -> TPEntity[t,n](tpeo)

  where

   (* Local Type Definitions *)

   process TPEntity[t,n](tpeo:TPEOptions):noexit:=
    TPEConnections[t,n](tpeo)
   endproc (* TPEntity *)
```

A formal data type is provided for the implementor's declaration of classes and options that are defined in the conformance clause of the protocol. A boolean-valued function (**Conform**) is applied to the value of this parameter, that determines whether or not the value satisfies the static conformance requirements of the standard. The process 'TPEntity' specifies the valid behaviour of transport protocol entities. This process describes the relationship between provision of transport connections and usage of network connections by instantiating the process named 'TPEConnections'.

The further decomposition of the protocol constraints, represented by the process TPEConnections, exploits the usage of internal gates. The form of the definition of TPEConnections, and the design of event structures at the internal gates, are aimed at facilitating the representation of distinct protocol constraints by distinct processes.

```
process TPEConnections[t,n](tpeo:TPEOptions):noexit:=
 hide p,a,d,s in
              TSTP[t,p,a,d,s,n](tpeo)
|[p,a,d,s]|
              TPNS[p,a,d,s,n]
endproc (* TPEConnections *)
```

At **p** TPDU transfers are described, together with related information. The events at **p** have the following structure:

```
p?cl:Class?d:Dir?c:Copy?tpdu:ETPDU?err:TPErr
```

- cl:Class is the protocol class in which the TPDU is handled.
- d:Dir tells whether the TPDU is transmitted or received.
- c:Copy tells whether or not the TPDU is considered a duplicate.
- tpdu:ETPDU is the TPDU to be transferred.
- err:TPErr significant for received TPDUs, qualifies protocol errors.

The internal gates **a** and **d** facilitate the formal representation of assignment to an existing network connection. At **a** (re)assignment to network connection, and at **d** deassignment from network connection occurs. The information passed at **a** consists of network connection ownership, and parameters as determined upon successful connection establishment.

```
a?own:NCO?ncp:NCPar
```

The information passed at **d** consists of a qualifier indicating whether or not the network connection is to be disconnected.

```
d?w:Deassign
```

The reason for the internal gate **s** consists in transferring the information about occurrences of a reset of a network connection to the transport entity. There is no event structure defined for this gate.

Process 'TSTP' specifies the constraints that refer to the provision of transport connections, i.e., class and options negotiation, segmenting and reassembling, flow control, connection release, splitting and recombining; while process 'TPNS' specifies the constraints that refer to the usage of network connections including concatenation and separation, usage of network expedited.

The process 'TSTP' accesses the network service boundary in order to provide the TPDU acknowledgement by receipt confirmation option. The definition of TSTP includes the instantiation of the process 'TCs' that formulates the constraints local to a transport connection, and TCNAccess which is an auxiliary process by means of which the network service access by transport connection instances is restricted to the interactions of their concern (TPDU acknowledgement by the Receipt Confirmation option by the Class 1 transport connection instances).

```
process TSTP[t,p,a,d,s,n](tpeo : TPEOptions):noexit:=
 hide r in
        TCs[t,p,a,d,s,r,n](tpeo)
      |[p]|
        TCNAccess[p,d,n]
endproc (* TSTP *)
```

The internal gate **r** is introduced to indicate the release of a transport connection. Any interaction at **r** consists of the reason why the connection is released, which is needed for the release procedures.

```
r?w:RelReason
```

The constraints on the usage of network connections can be formulated locally to a network connection. Process 'TPNS' therefore consists of successive instances of process 'NC'.

```
process TPNS[p,a,d,n]:noexit:=
 hide dd in
        NC[p,a,d,n,dd]
      |[dd]|
        dd;TPNS [p,a,d,n]
endproc (* TPNS *)
```

5.2.2 Behaviour Reductions

In this section the slicing approach is applied to a real base specification developed by protocol experts in the constraint oriented style. The behaviour

reductions are performed based on the test suite hierarchy shown in Figure 5.3, which is obtained from the test suite developed for transport protocol Class 2, within the CTS-WAN Project [50]. Due to the size of the base specification, only a part of the TSS is considered, and our aim is to obtain a separate protocol specification describing the behaviour of Class 2 protocol entity during the release of a transport connection.

Since the base specification given in [49] defines the behaviour of all classes but Class 4, the first reduced specification corresponds to the behaviour of the Class 2 transport protocol entity. Among the 23 transport procedures explained above, the following are the ones that are applicable to Class 2 protocol entities.

- Assignment to a network connection,
- Transfer of TPDUs,
- Segmentation and reassembly,
- Connection establishment,
- Error release,
- Association of TPDUs with transport connections,
- Treatment of protocol errors,
- Concatenation and separation
- Explicit normal release,
- Numbering of DT TPDUs,
- Expedited data transfer (with normal network data transfer service),
- Multiplexing and demultiplexing,
- Explicit flow control.

Since Class 2 protocol operates on Type A network service where network resets are ignored, this fact must be reflected in the definition of the process 'TPEConnections' and the subsequent behaviour specification by dropping the internal gate s, which happens to be diagonal reduction.

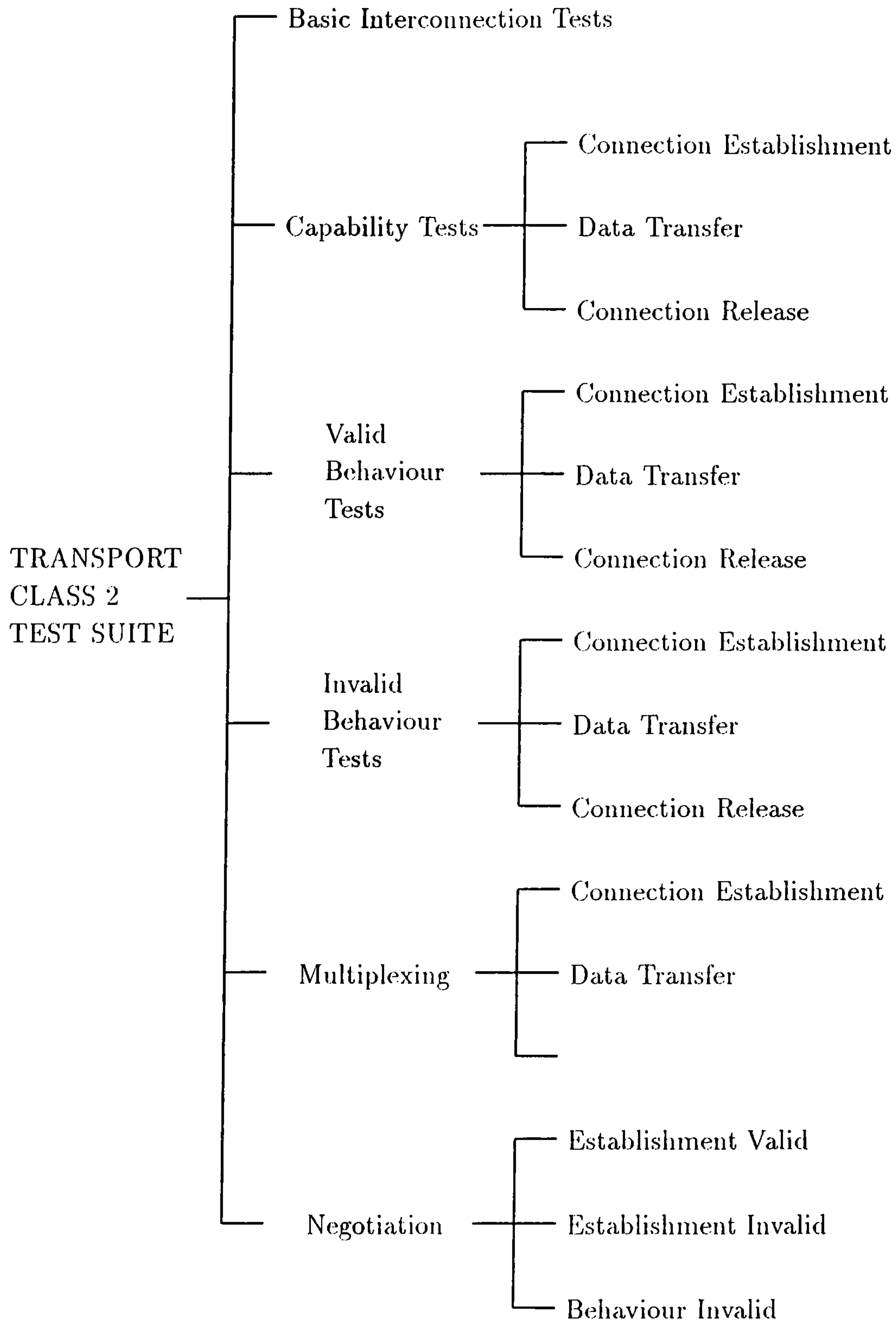


Figure 5.3. Test Suite Structure for Transport Protocol Class 2

```
process TPEConnections[t,n](tpeo:TPEOptions):noexit:=
  hide p,a,d in
            TSTP[t,p,a,d](tpeo)
 |[p,a,d]|
            TPNS[p,a,d,n]
endproc (* TPEConnections *)
```

As a consequence of this, the actual and corresponding formal gate parameters of the two instantiated processes, 'TSTP' and 'TPNS', have undergone a diagonal reduction process. The arguments of the parallel operator no longer contain the **s** gate. By the way the gate **n** in the instantiation of the process 'TSTP' is dropped because a Class 2 protocol entity does not need to access the network service boundary in order to acknowledge data by the receipt confirmation option. Accordingly, the instantiation of the process 'TCNAccess' is horizontally reduced from the body of the process 'TSTP'. Since it is not used elsewhere in the specification, its definition is also vertically reduced.

```
process TSTP[t,p,a,d](tpeo:TPEOptions):noexit:=
 hide r in
        TCs[t,p,a,d,r](tpeo)
endproc (* TSTP *)
```

Another diagonal reduction is applied to the event structure at gate **p**. Since only Class2 protocol is specified, there is no need to include a separate event to indicate the class of the particular transport connection. So the events at gate **p** in the resultant specification have the following structure:

```
p?d:Dir?c:Copy?tpdu:ETPDU?err:TPErr
```

Process 'TCs', consists of the instantiation of the process 'TC' which actually describes the behaviour of the transport protocol entity. According to the definition of the process given below, one connection has to be released (by the process 'TC' performing an interaction at gate **dd** and exiting), in order for the next one to be supported by the transport entity.

```
process TCs[t,p,a,d,r](tpeo:TPEOptions):noexit:=
 hide dd in
```

```
          TC[t,p,a,d,r,dd](tpeo)
        |[dd]|
          dd;TCs[t,p,a,d,r](tpeo)
  endproc (* TCs *)
```

Protocol constraints specified by the process 'TC' are classified as follows:

1. Service constraints, i.e, constraints at **t** only, are specified by TCEP.
2. Relationship between TSPs and TPDUs, specified by TSPTPDU,
3. Normal exchange of TPDUs between the two peer entities, specified by TPDUFlowProcedures independently at events at **t** and error recovery procedures,
4. (re)assignment of the transport connection to one (or more) network connection(s), specified by TCA. This process specifies also connection release that follows the expiration of the reassignment/resynchronization timers.
5. Connection release, specified by TCRelease,
6. Exception handling, specified by TCE. This comprises error recovery procedures, specified by NSErrorRecovery, and treatment of protocol errors, specified by ProtocolErrorHandling.

```
process TC[t,p,a,d,r,dd](tpeo:TPEOptions):exit:=
 TCEP[t](TSCallingRole) [] TCEP[t](TSCalledRole)
|[t]|
 (hide k in
         (TSPTPDU[t,p,k] [> exit
       |[p,k]|
          TPDUFlowProcedures[p,k](tpeo) [> exit)
 |[t,p,k]|
         TCA[t,p,a,d,r,k,dd](tpeo)
 |[t,p,d,r,k]|
         (TCRelease[t,p,r,dd]
       |[p,r]|
          TCE[p,d,r,k,n] [> exit))
endproc (* TC *)
```

```
process TCE[p,d,r,k,n]:noexit:=
 NSErrorRecovery[p,d,k,n]
|[p,d]|
 ProtocolErrorHandling[p,d,r]
endproc (* TCE *)
```

The internal gate **k**, introduced by the definition of TC, serves the purpose of separating concerns in the description of error recovery mechanisms, particularly relating to retention of TPDUs until acknowledgement and retransmission.

The procedures relating to normal flow of TPDUs between the two peer entities, described by the process 'TPDUFlowProcedures' is further structured as follows:

```
process TPDUFlowProcedures[p,k](tpeo:TPEOptions):noexit:=
 (SupportedOptions[p](tpeo)
 |[p]|
  TPNegotiations[p]
 |[p]|
  TPDUNumbering[p]
 |[p]|
  ExplicitFlowControl[p]
 |[p]|
  Checksum[p])
|[p]|
 DTRetransmission[p,k]
endproc (* TPDUFlowProcedures *)
```

Each instantiated process describes the required actions to be carried out during the execution of the respective transport procedure.

Various types of reductions are performed in the process 'TC'. Since there are no error recovery procedures specified in the Class 2 protocol, the internal gate **k** which is introduced to serve the purpose of separating concerns in the description of such procedures, can be dropped. Additionally, horizontal reduction is applied to the instantiation of the process 'NSErrorRecovery' within the process TCE. This further results in the vertical reduction of the definition of the process 'NSErrorRecovery' from the specification.

```
process TC[t,p,a,d,r,dd](tpeo:TPEOptions):exit:=
 TCEP[t](TSCallingRole) [] TCEP[t](TSCalledRole)
|[t]|
 ((TSPTPDU[t,p] [> exit
  |[p]|
   TPDUFlowProcedures[p](tpeo) [> exit)
 |[t,p]|
   TCA[t,p,a,d,r,dd](tpeo)
 |[t,p,d,r]|
  (TCRelease[t,p,r,dd]
  |[p,r]|
   TCE[p,d,r,n] [> exit))
endproc (* TC *)

process TCE[p,d,r,n]:noexit:=
 ProtocolErrorHandling[p,d,r]
endproc (* TCE *)
```

Since checksum and retransmission procedures are not applicable in Class 2, horizontal reduction is applied to their instantiations within the process 'TPDUFlowProcedures', and the respective process definitions are vertically reduced from the resulting specification.

```
process TPDUFlowProcedures[p,k](tpeo:TPEOptions):noexit:=
 SupportedOptions[p](tpeo)
|[p]|
 TPNegotiations[p]
|[p]|
 TPDUNumbering[p]
|[p]|
 ExplicitFlowControl[p]
endproc (* TPDUFlowProcedures *)
```

Since there is no retransmission procedure, there does not exist any possibility of receiving a duplicate TPDU, and the interaction at gate **p** related to the reception of duplicate TPDUS (i.e., ?c:Copy) can be diagonally reduced.

The behaviour of the base protocol in response to invalid inputs is specified by the process 'ProtocolErrorHandling'. So, valid behaviour specification can

be obtained by vertically reducing this process, and the process 'TCE' which instantiates only this process. So, the resulting structure of the process 'TC' is as follows:

```
process TC[t,p,a,d,r,dd](tpeo:TPEOptions):exit:=
 TCEP[t](TSCallingRole) [] TCEP[t](TSCalledRole)
|[t]|
 ((TSPTPDU[t,p] [> exit
  |[p]|
   TPDUFlowProcedures[p](tpeo) [> exit)
 |[t,p]|
   TCA[t,p,a,d,r,dd](tpeo)
 |[t,p,d,r]|
  TCRelease[t,p,r,dd])
endproc (* TC *)
```

Additionally, the event structure at the p gate is diagonally reduced one step more by dropping the value with type TPErr, which qualifies incoming TPDUs for protocol errors. The resulting event structure at gate p is as as follows:

```
p?d:Dir?tpdu:ETPDU
```

The last step in behaviour reductions is to obtain the specification of the Class 2 transport protocol during connection release phase. The specification defining the connection release procedure can be obtained by performing some horizontal reductions within the process 'TC', resulting in the final form given below.

```
process TC[t,p,a,d,r,dd](tpeo:TPEOptions):exit:=
 TCEP[t](TSCallingRole) [] TCEP[t](TSCalledRole)
|[t,p,a,d,r]|
 ((TSPTPDU[t,p] [> dd;exit)
 |[t,p,d,s,r]|
  (TCRelease[t,p,r,dd]))
  endproc (* TC *)
```

The release of a transport connection is initiated under any of the following cases:

- Transport connection refusal by responder, following an internal decision.
- Network connection failure, because error recovery and reassignment is not made use of in Class 2.
- The internal decision of releasing the transport connection upon detection of a protocol error which is part of the invalid behaviour.
- Transport connection release by the transport service user.

The release at the transport service interface, in relationship to events at gate **r**, is described by TCRelS, whereas the ordering of events at the entity internal gates, as determined by the peer-to-peer release procedures, is described by TCRelP.

```
process TCRelease[t,p,r,dd]:exit:=
 TCRelS[t,r,dd]
|[r]|
 TCRelP[p,r,dd]
endproc (* TCRelease *)
```

The following constraints are described by TCRelS:

- if the transport connection is to be released with no previous execution of a T.CONNECT primitive by the entity, then no service primitive is executed by the entity.
- Otherwise,
  - T.DISCONNECT primitive is to be executed, and
  - the release indicator passed at gate r shall be User if and only if a T.DISCONNECTrequest was previously executed.

```
process TCRelS[t,r,dd]:exit:=
 r?w:RelReason;dd;exit
[]
 t?tc:TSP[IsTCON(tc)] ;
 (r?rr:RelReason[rr ne Normal];TCEPRelease[t]
 []
```

```
  t?tdr:TSP[IsTDISreq(tdr)] ;
  r!Normal of RelReason;dd;exit)
endproc (* TCRelS *)
```

The service constraints related to the release of a connection are specified by the process 'TCEPRelease'.

```
process TCEPRelease[t]:exit:=
 t?tsp:TSP[IsTDIS(tsp)];exit
endproc (* TCEPRelease *)
```

TCRelP describes the connection release procedures but with no concern for the interactions with the TS user. These procedures define constraints relating to the following cases:

- Connection refusal by responder (TCRespRefusal),
- Release at responder (TCRespRelP),
- Release at initiator (TCInitRelP),
- Connection Release due to protocol error.

```
    process TCRelP[p,r,dd]:exit:=
     p?cl:Class!Recv?CR:ETPDU[IsCR(CR)] ;
     (TCRespRefusalP[p,r,dd](cl)
     []
      TCRespRelP[p,r,dd](cl)
     []
      Class2ErrorRelease[p,r,dd])
    []
     p?cl:Class!Send?CR:ETPDU[IsCR(CR)];
     (TCInitRelP[p,r,dd](cl)
     []
      Class2ErrorRelease[p,r,dd])
    []
     r?w:RelReason;dd;exit
     (* Release due to protocol_error *)
endproc (* TCRelP *)
```

Since we eliminated the interaction related to the class of the protocol (i.e., ?cl:Class) previously, there is no need to pass this value as parameter to the instantiated processes. This is another type of diagonal reduction which is applied to the formal and corresponding actual process parameters. After eliminating the branch related to the release of transport connection due to protocol error (since we deal only with valid behaviour), the resulting body of the process 'TCRelP' is:

```
process TCRelP[p,r,dd]:exit:=
 p!Recv?CR:ETPDU[IsCR(CR)];
 (TCRespRefusalP[p,r,dd]
 []
  TCRespRelP[p,r,dd]
 []
  Class2ErrorRelease[p,r,dd])
[]
 p!Send?CR:ETPDU[IsCR(CR)];
 (TCInitRelP[p,r,dd]
 []
  Class2ErrorRelease[p,r,dd])
endproc (* TCRelP *)
```

Following is the final reduced specification related to connection release procedure under the valid behaviour category of Class 2 transport protocol.

```
process TCRespRefusalP[p,r,dd]:exit:=
 choice etp:ETPDU,drr:DRReason []
  [IsDR(DR) and (drr IsReasonOf DR)] ->
   p!Send!etp;
   (AwaitDC[p,r]
   |||
    r?w:RelReason;dd;exit)
[]
 Class2ErrorRelease[p,r,dd]
endproc (* TCRespRefusalP *)

process TCRespRelP[p,r,dd]:exit:=
 choice ecc:ETPDU [] [IsCC(ecc)] ->
```

```
   p!Send!ecc;
   TCDataRelease[p,r,dd]
  []
   Class2ErrorRelease[p,r,dd]
 endproc (* TCRespRelP *)

 process TCInitRelP[p,r,dd]:exit:=
  p!Recv?DR:ETPDU[(IsDR(DR) or IsER(DR))];
  r?w:RelReason;dd;exit
 []
  p !Recv ?CC:ETPDU[IsCC(CC)];
  TCDataRelease[p,r,dd]
 []
  Class2ErrorRelease[p,r,dd]
 endproc (* TCInitRelP *)

 process TCDataRelease[p,r,dd]:exit:=
  r?w:RelReason;Release[p,r,dd]
 []
  Class2ErrorRelease[p,r,dd]
 endproc (* TCDataRelease *)

 process Release[p,r,dd]:exit:=
  ExplicitRelease[p,r,dd]
 endproc (* Release *)

 process ExplicitRelease[p,r,dd]:exit:=
  choice edr:ETPDU [] [IsDR(edr)] ->
   p!Send!edr;
   p!Recv?et:ETPDU[(IsDR(et) or IsDC(et))];
   dd;exit
 endproc (* ExplicitRelease *)

 process Class2ErrorRelease[p,r,dd]:exit:=
  r?rr:RelReason[rr eq NoReass or (rr eq NoResyn)];
  dd;exit
 endproc (* Class2ErrorRelease *)
```

```
process AwaitDC[p,r,dd]:exit:=
 p!Recv?dct:ETPDU[IsDC(dct)];
 r ?w:RelReason;dd;exit
endproc (* AwaitDC *)
```

Chapter 6

TEST DESIGN USING BASE AND FUNCTIONAL SPECIFICATIONS

As defined in Chapter 1, TTCN is the standardized specification language of test suites. It combines a tree notation for dynamic behaviour with a tabular representation of various language constructs [51]. An abstract test suite (ATS) in TTCN consists of 4 parts:

- **Test suite overview:** The test suite overview section first names the test suite and defines its context with respect to the appropriate IUT protocol standard, PICS, and test methods. It describes the structure of the abstract test suite and provides an index of its test cases.
- **Declarations:** An abstract test suite is composed of objects of many types, such as timers, PCOs, messages. Declarations of such objects are the next major component of a test suite specified in TTCN.
- **Constraints:** In the constraints section of an ATS, particular data values are specified for PDU fields and ASP parameters as used in the constrained test events in the dynamic behaviour part of the test suite.
- **Dynamic Behaviour:** The dynamic behaviour section of a TTCN abstract test suite comprises the main body of the ATS including the test cases, test steps and default behaviours.

A abstract test suite in TTCN can be generated from protocol specifications by following the steps given below.

1. Generation of the declarations part. Largely, this can be done by syntactic transformations.
2. Generation of base constraints using the PDU and ASP definitions.
3. Generation of modified constraints.
4. Generation of test cases.
5. Representing the selected test cases according to the syntax and semantics of TTCN.
6. Completion of the test suite overview.

This chapter is about the applications of the chart based test design methodology introduced in Chapter 2 within the perspective of testable specifications developed in the previous chapters. Test cases are derived from the respective base and functional specifications of INRES and ACSE protocols. Section 6.1 applies the methodology to base specifications, and Section 6.2 includes some important results on the generation of test cases from the reduced specifications.

The results given in this chapter are obtained by the computer-aided software tool LOTEST [27] which implements the test design methodology discussed in Chapter 2 Section 2.6 to derive test cases from LOTOS specifications. LOTEST is an interactive tool and includes a number of modules that facilitate various steps of test design. It has been developed on SUN workstations using the X_Window programming environment.

The backbone of the LOTEST environment is the formal notation of EFSM-Chart which has a mathematically precise semantics, and simple representation of data flow. The first step applied to a LOTOS specification is compilation. The compiler does lexical syntactic and semantic analysis, and produces an internal representation of the LOTOS specification in the Prolog clause form. After compilation, the chart generator is activated, which translates the intermediate form of the specification into a chart by bottom-up synthesis. Then input for data flow graph and test cases can be generated. There are four interactive tools: **ctool**, **dfgtool**, **edittest**, and **testgen**. The chart is displayed in the form of a finite state machine by ctool. The dfgtool displays the data flow graph with automatic blocking and offers several facilities for block merging. Edittest is used to help the test designer to interactively go through the

test cases and identify uninteresting ones. It is also used to eliminate redundant assignment and predicates from the test cases with the help of the data flow graph. Finally testgen is for selecting the unparameterized test sequences based on the data flow information.

6.1 Test Generation From Base Specifications

Before the actual test generation procedure can start, a LOTOS specification must be normalized in order to identify the control flow and data flow within the specification. As an EFSM representation, the chart reveals the control structure of the specification. The size of the chart generated depends on the size of the specification as well as how it is structured. For example, the chart generated from the ACSE base specification consists of 141 states and 194 transitions, while the charts obtained from the INRES Initiator and Responder protocol specifications contain 36 states, 50 transitions, and 26 states, 34 transitions, respectively.

Once the control structure of the specification is obtained, test cases can be generated by using the transition-tour method [52]. As explained in Chapter 2 the algorithm proposed in [30] generates the test cases from the EFSM-Chart by taking nondeterminism into consideration. From their respective charts, the algorithm generates 53 test cases for ACSE, and 16 test cases for the Initiator base specifications.

The generated test cases must be inspected in order to detect any infeasibilities and redundancy. The test case generation algorithm may also generate uninteresting test cases. Analysis of test cases is done by using the interactive tool **edittest**. Edittest displays the test cases in one text window, the rules that occur in the test case in one text window, and finally the EFSM chart in another text window. By editing the text windows, infeasible and uninteresting tests cases can be eliminated or replaced with feasible and meaningful ones. Out of the 53 test cases generated for the ACSE protocol, 6 test cases happened to be infeasible, and 17 of them found to be uninteresting and meaningless. When the infeasibilities have been resolved, all of them become equivalent to existing test cases, so they can also be considered as redundant. In the case of the Initiator protocol, none of the test cases are found to be infeasible, but 3 of them are redundant test cases, i.e., their functions can be achieved by using

others.

The last step in the analysis of the generated test cases is the elimination of unnecessary assignments and predicates, which is referred to as test case reduction. Most of the test cases generated from the ACSE base specification contain redundant assignment statements which have no effect on the functionality of the tester, and they have been dropped. The assignment statements in the test cases result from value passing in LOTOS specifications due to interprocess communications and process instantiations. The following is an example of the test cases that contain redundant assignments. Given in the from of the rules of the chart, it tests the IUT for the correct implementation of the abnormal release procedure in the 'Associated State' by stimulating it with a P-U.ABORTrequest primitive. The assignment statements 'c(22):=calling' and 'c(19):=c(22)' have no effect on the functionality of the test case, so they can be dropped.

```
<A?x(9):primitive,188,191,3,true,[IsAASCreq(x(9))],e,e>,
<P!Out!PCONreq(make_AARQ(get_AASCreq(x(9)))):primitive,
                                191,192,7,true,true,e,e>,
<P!Input?x(4):primitive,192,196,11,true,
[and(and(IsPCONcnfA(x(4)),IsAARE(user_data(x(4)))),
 eq(result(get_AARE(user_data(x(4)))),accepted))],e,e>,
<A!make_AASCcnfU(get_AARE(user_data(x(4)))),196,207,26,
                                         true,true,e,e>,
<i,207,223,41,true,true,c(22):=calling,c(19):=c(22)>,
<P!Input?x(21):primitive,223,242,58,true,
 [and(IsPUABind(x(21)),IsABRT(user_data(x(21))))],e,e>,
<A!make_AABRind(get_ABRT(user_data(x(21)))),242,265,72,
                                         true,true,e,e>,
<ir,265,188,79,true,true,e,e>.
```

Before being represented the tests must be selected according to some user-defined criteria. There are two approaches to the selection of tests generated from base specifications. The first approach is to apply the functional test selection, and to base the selection process on the protocol functions identified by the user. In order to identify the protocol functions, the flow of data in the specification must be extracted. The flow of data in a LOTOS specification reflects how input primitive parameters determine the values of the context variables, and they in turn determine the values of the output primitives [30].

A data flow graph (DFG) models the flow of data in the chart. Four types of nodes are used in DFG:

i-nodes represent input primitives,
d-nodes represent variables and data,
f-nodes represent ADT functions,
o-nodes represent output primitives.

The edges of the DFG are used to represent the flow of information and labelled with the transition number on which the flow is achieved.

Generation of the data flow graph from the chart is performed by using the algorithm proposed in [30]. The *when*, *action* and *assignment* clauses of each rule in the chart are scanned statically, and various types of nodes and arcs are created that reveal the flow of data within the chart. Generation of DFG is only possible when PDUs and ASPs are explicitly identified. Also, it is necessary to identify operations associated with different kinds of PDUs and ASPs, which is achieved by processing the abstract data type definitions. The structure of the PDUs and ASPs must be provided by the user.

In order to identify the individual protocol functions from the data flow graph, it must be sliced according to some user-defined criteria. Data flow graph slicing is performed in two phases by using the **dfgtool**. The first step is called *blocking* and the second step is called *merging* where the criteria must be provided by the user based on the knowledge of the specification. Related to test selection, protocol functions obtained from the data flow graph must be tested using the test cases obtained from the chart. For this purpose, it is necessary to extract the transition labels of each function of the data flow functions. If a single test case covers all the labels in a given function, it is this test case which is selected, otherwise more than one test case is selected for full coverage. The **testgen** tool gets test cases from the data flow graph and then generates full coverage of each of the data flow functions by test cases.

In the second approach to test selection, test cases are placed in a test suite hierarchy, and a test purpose is assigned to every test case. Once the hierarchy and the purposes are determined, for each test purpose one or more test cases must be selected from the generated test cases. If the specification defines invalid and inopportune behaviour along with valid behaviour, the resulting test cases must be considered regarding invalid and inopportune behaviour, as

well.

The test suite structure and test purposes document for ACSE [47] contains a total of 88 test purposes. Among the 53 test cases generated from the base specification, 30 of them are meaningful and can be used to cover 60 of the test purposes. For example, the following test case represented in the rules of the chart satisfies the purpose of testing the IUT in state 'AwaitAARE' when it receives an A.ABORTrequest from the user. It is actually the first test purpose within the ACSE/BV/AR/AA test group objective as defined in [47].

```
<A?x(9):primitive,188,191,3,true,[IsAASCreq(x(9))],e,e>,
<P!Out!PCONreq(make_AARQ(get_AASCreq(x(9)))),191,192,7,
                                         true,true,e,e>,
<A?x(3):primitive,192,203,18,true,[IsAABRreq(x(3))],e,e>,
<P!Out!PUABreq(make_ABRT(get_AABRreq(x(3)))),203,212,34,
                                         true,true,e,e>,
<ir,212,188,49,true,true,e,e>.
```

Due to the specification of invalid behaviour by using negation in selection predicates rather than specifying the related cases explicitly one test case can be interpreted to cover more than one test purpose. Also, by interpreting some of the generated test cases in different ways according to their preambles, bodies and postambles, some test cases can be interpreted to cover more than one test purpose. For example, the following test case which tests the behaviour of the ACSE protocol implementation in response to invalid inputs in its 'Unassociated' state covers all four of the test purposes under the group ACSE/BI/STA0. That is, 'not(IsAARQ(user_data(x(9))))' is a short way of saying 'IsAARE.. or IsRLRQ.. or IsRLRE.. or IsABRT..'.

```
<P!Input?x(9),188,190,2,true,
 [and(IsPCONind(x(9)),not(IsAARQ(user_data(x(9)))))],e,e>,
<A!make_AABRind,190,195,6,true,true,e,e>,
<P!Out!PUABreq(make_ABRT),195,206,10,true,true,e,e>,
<ir,206,188,25,true,true,e,e>.
```

Since the chart construction algorithm implemented within LOTEST treats the disable operator specially by creating an alternative transition not emanating from every state of the chart corresponding to the disabled behaviour

expression, but only the first state of it, some test purposes remain uncovered by the generated test cases.

When we consider the **Initiator** protocol, 14 meaningful test cases cover 16 test purposes of the TSS&TP for INRES given in Appendix C, and 3 test purposes remain uncovered due to the treatment of the disable operator as mentioned above. For example, the test case given below satisfies the only test purpose under the test group INRES/I/BIO/STA1.

```
<ISAP?sp(1),215,149,1,true,true,e,e>,
<MSAP!MDATreq(CR),149,152,5,true,true,e,e,>,
<i,152,155,9,true,true,z(3):=s(0),z(2):=z(3)>,
<MSAP?sp(2),155,160,13,true,
 [and(and(isMDATind(sp(2)),not(isDR(data(sp(2))))),
                             not(isCC(data(sp(2)))))],
                             e,e>,
<is,155,161,14,true,true,e,e>,
<ir,160,155,17,true,true,e,z(2):=z(2)>,
<i,161,166,19,[z(2)==4],true,e,e>,
<ISAP!IDISind,166,171,22,true,true,e,e>,
<ir,171,215,27,true,true,e,e>.
```

Since INRES protocol ignores any invalid and inopportune behaviour and does not change its state, test cases for such behaviour must be augmented with extra sequences that bring the IUT in its initial state. This makes the tests for invalid/inopportune behaviour longer.

Events and assignments in a test case comprise the dynamic behaviour of the test case. The flow of control is sequential except when there is a spontaneous transition. Before representing the test cases, except internal events, all other events are inverted, which means that input events are converted to output events and vice versa.

Any constraints on the initial values of ASPs, PDUs and other substructures are defined as *base constraints*. Each test case imposes other constraints which are called *dynamic constraints* or *modified constraints*. Default values and other information that are unchanged in many constraints are considered in base constraints. Base constraints can be modified by re-specifying a number of fields. For each ASP and PDU two base constraints are defined, one for

input events and another for output events.

When test case generation is based on base specifications, usually large number of test cases are produced where many of them are uninteresting or meaningless. This necessitates the selection of the generated tests based on some user-defined criteria. As mentioned above, test selection is either done by using functional analysis, or hierarchical test selection is employed. If hierarchical test selection is employed, an alternative way is to generate the test cases from functionally reduced specifications. By this way selection of the tests can be done before actually generating them, while at the same time the number of generated test cases can be minimized.

6.2 Test Generation From Functional Specifications

Once the base specification is sliced into reduced specifications, test cases can be generated, and depending on the level of subdivision, a reduced specification provides several test cases that represent the corresponding test group. The steps involved in the application of the methodology are almost the same as in the case of base specifications. But, as discussed above test selection step is omitted, and the technique used in the generation of test cases may be changed. Regarding the test case reduction, some reductions may not be needed any more. The steps involved in the generation of test cases from functional specifications is outlined in the following subsections.

6.2.1 Chart Generation

The first step is again normalization, i.e., chart generation. This time, since the specifications have much smaller sizes than the original base specification, the corresponding charts are also smaller in size. Table 6.1 and 6.2 give the sizes of the resulting charts for every functional specification of ACSE and INRES in terms of the number of transitions and states.

Func. Spec.	**# Of States**	**# Of Trans.**
I/BV/CE	11	13
I/BV/DT	15	19
I/BV/DC	3	3
I/BIO/STA0	3	4
I/BIO/STA1	4	5
I/BIO/STA2	4	5
I/BIO/STA3	5	6
R/BV/CE	6	6
R/BV/DT	9	11
R/BV/DC	3	3
R/BIO/STA0	3	4
R/BIO/STA1	3	3
R/BIO/STA2	4	5

Table 6.1. Chart Sizes Of Functional Specifications For INRES

Func. Spec.	**# Of States**	**# Of Trans.**
BV/AE/I	12	15
BV/AE/R	10	12
BV/NR/RQ	20	24
BV/NR/AC	9	10
BV/AR/AA	3	3
BV/AR/PUA	3	3
BV/AR/PPA	3	3
BI/STA0	4	4
BI/STA1	7	9
BI/STA2	4	4
BI/STA3	12	14
BI/STA4	4	4
BI/STA5	8	9
BI/STA6	4	4
BI/STA7	11	12

Table 6.2. Chart Sizes Of Functional Specifications For ACSE

Func. Spec.	# Of Test Cases	# Of Purposes Covered
I/BV/CE	3	3
I/BV/DT	5	5
I/BV/DC	1	4
I/BIO/STA0	3	4
I/BIO/STA1	-	-
I/BIO/STA2	-	-
I/BIO/STA3	-	-
R/BV/CE	1	1
R/BV/DT	3	3
R/BV/DC	1	3
R/BIO/STA0	1	1
R/BIO/STA1	-	-
R/BIO/STA2	-	-

Table 6.3. Test Cases Generated From Functional Specifications Of INRES

6.2.2 Generation Of Test Cases

The next step is the application of the test case generation algorithm to the charts derived from each of the reduced specifications. Tables 6.3 and 6.4 contain some results related to the test cases generated from the functional specifications of INRES and ACSE, respectively. As seen from the data, most of the test cases are useful in the sense that they can be used to test for the satisfaction of the purposes defined in the related TSS&TP documents. Since the test cases consist only of test bodies, one test case may be used to test for the satisfaction of more than one test purpose by prefixing it with different preambles. This is usually the case for state-oriented test groups where each test purpose corresponds to a particular state of the protocol machine. The test group INRES/I/BV/DC is an example of this kind of test group.

The algorithm proposed in [30] and implemented in LOTEST requires a fully connected state-machine in order to derive the test cases. Related to separate phases and states of the protocols, the EFSMs of the reduced specifications need not be strongly connected, i.e., their initial and final states may be different. In order to apply the method implemented in LOTEST to such specifications, virtual transitions must be added from each final state to the initial state of the charts. In other words, the final states must be converted to initial

Func. Spec.	**# Of Test Cases**	**# Of Purposes Covered.**
BV/AE/I	4	6
BV/AE/R	3	3
BV/NR/RQ	5	4
BV/NR/AC	2	2
BV/AR/AA	1	6
BV/AR/PUA	1	6
BV/AR/PPA	1	6
BI/STA0	1	4
BI/STA1	2	8
BI/STA2	1	4
BI/STA3	3	12
BI/STA4	1	4
BI/STA5	2	8
BI/STA6	1	4
BI/STA7	2	8

Table 6.4. Test Cases Generated From Functional Specifications Of ACSE

states. In spite of this, some entries in the table for INRES, such as the ones corresponding to INRES/I/BIO/STA1, INRES/R/BIO/STA2 test groups, are empty. The reason behind this is the fact that INRES protocol ignores invalid and inopportune test events. Since no state change occurs in response to invalid/inopportune behaviour, the fully connectedness requirement of the algorithm is violated, and no test cases are generated.

In order to overcome this problem, a different approach can be adopted while generating the test cases, and a straightforward mapping from the chart to TTCN can be applied. The EFSM can be mapped to a tree describing the possible sequences of input and output events. Test cases can be derived directly from the EFSMs representing the reduced specifications by developing the tree of possible valid sequences while each branch of the tree forms a separate test case. This tree can be described in the standardized test notation TTCN, and 'Pass' verdicts can be attached to the nodes corresponding to the final states of the EFSM.. The test cases are specified from the point of view of the tester, while the specification describes the behaviour of the IUT; therefore the direction of events must be inverted. Input events for the IUT are output events for the tester, output events sent by the IUT are input events,

i.e., receive events for the tester. For each receive event for the tester an alternative OTHERWISE event has to be specified with verdict 'Fail' in order to deal with unforeseen responses from the IUT, and with verdicts 'Inconclusive' to deal with spontaneous transitions that result from non-determinism within the IUT. Since there is no inherent directionality in LOTOS interactions, input and output events must be made explicit in the specifications. As mentioned at the end of Chapter 2, this can be achieved by adding extra event structures to the external gates of the specification.

6.2.3 Test Case Reductions

The generated test cases must again be inspected for any redundancy with respect to the assignments and predicates. When we consider the ACSE protocol, it is observed that test cases derived from the reduced specifications still contain redundant assignments. This is due to the absence of diagonal reductions on the parameters of the processes comprising the body of the ACSE specification. For example, the test case given below in TTCN form is generated from the reduced specification corresponding to the test group ACSE/BV/NR/RQ. It tests the correct operation of the IUT during the normal release procedure by stimulating it an A.RELEASErequest primitive and accepting the release of the association. Since the assignment statements 'c(2):=c(6)', and 'c(6):=c(7)' are redundant, they can be eliminated from the test case without disrupting its functionality. The preamble contains the necessary actions to put the IUT in the desired state.

```
+ Preamble
    A! ARLSreq                    C1
      P? PRLSreq                  C2
        c(6)=c(7)
          c(2)=c(6)
            P! PRLScnfA           C3
              A? ARLScnfA         C4        Pass
              ? OTHERWISE                   Fail
      ? OTHERWISE                           Fail
```

As stated in the previous section, the presence of the assignment statements in the rules of the chart is mainly due to value passing in interprocess synchronizations and process instantiations in LOTOS specifications. When diagonal reductions are performed on the instantiations and definitions of processes by removing some of the formal and corresponding actual process parameters, test case reductions carried out on the tests derived from base specifications might become irrelevant. In such cases, test case reduction step can also be skipped.

6.2.4 Infeasible Test Cases

When we consider the generated test cases with respect to infeasibility, it is observed that none of the test cases obtained from the reduced specifications of the ACSE protocol contain any infeasibilities. This is mainly due to the fact that, the test cases derived from functional specifications define the only the body of the actual tests to be executed on the specification without the specification of the preambles. Since the reason behind the infeasibility of a test case is an assignment statement within the preamble which causes a predicate to always evaluate to false; by prefixing the generated test bodies with suitable preambles, feasible test cases can be obtained. For example, the following test case given in TTCN form can be made feasible by assigning a proper value to the variable 'c(6)' in the preamble.

```
+ Preamble
    A! ARLSreq                  C1
      P? PRLSreq                C2
        P! PRLSind              C3
          A? ARLSind            C4
            [c(6)=called]
              A! ARLSrsp        C5
                P? PRLSrspA     C6        Pass
                ? OTHERWISE               Fail
          ? OTHERWISE                     Fail
      ? OTHERWISE                         Fail
```

Chapter 7

CONCLUSIONS

A design trajectory that results in testable protocol specifications is proposed, and applied to some protocol specifications written in the formal description technique LOTOS. The steps followed are the development of formal base specifications, systematic derivation of functional specifications from the base specifications, and test design from functional specifications.

Base specifications are developed in a hierarchical manner by defining the implementation options as parameters, and including the behaviours that must be tested. Several techniques are proposed that can be used in developing formal base specifications including PICS parameterization and invalid/inopportune behaviour specification along with valid behaviour specification. The proposed methods are applied on two protocols. Formal base specifications of the INRES and ACSE protocols are developed, and the problems encountered while applying the techniques are stated. The PICS proforma utilized in the development of the INRES protocol is defined according to the OSI standards.

The approach adopted in the derivation of functional specifications is slicing. By performing various types of behaviour reductions on base specifications, slices of those specifications are obtained in a systematic manner where each slice corresponds to a particular function of the protocol. The criteria on which the behaviour reductions are based are obtained from the hierarchically designed test suite structures of the base protocols. The behaviour slices of the developed base specifications INRES and ACSE protocols are obtained according to their respective test suite structures. A test suite structure and test purposes document is defined according to suggestions of the OSI standard

methodology and framework for conformance testing.

Since the charts obtained from the resulting reduced specifications are smaller in size when compared with those obtained from base specifications, the test generation process is simplified. The results indicate that the tests generated from the functional specifications by using the standard tools exhibit some important properties regarding the analysis and representation of test cases. Some problems arise with respect to the connectedness of the resulting EFSMs and this necessitates the application of a different methodology while deriving the test cases from reduced specification. A possible solution of deriving tests directly in TTCN form from the respective functional specifications is outlined in Chapter 6.

Possible future work includes the automation of the behaviour reduction algorithm given in Chapter 3 by using knowledge based techniques. Also, research is needed in the representation of the test cases generated from reduced specifications. This includes PDU/ASP identification, and constraint generation from the behaviour of the particular test cases by taking the dynamic nature of the test cases into account. Handling nondeterminism within the context of reduced specifications also deserves study. Since abstract data type definitions comprise the important part of LOTOS specifications, a similar study can be conducted by performing reductions on abstract data types along with behaviour reductions. The behaviour reductions considered in this study are executed on base specifications according to the hierarchically structured test suites up until the outermost test groups of the hierarchy. A possible extension of the methodology can be the generation of more refined specifications where each specification provides the behaviour of exactly the individual test cases rather than test groups.

Bibliography

[1] A. Tang and S. Scoggings, *Open Networking with OSI*, Prentice Hall Inc., New Jersey, 1992.

[2] International Organization for Standardization, *Basic reference Model for Open System Interconnection*, ISO 7498, 1984.

[3] G.V. Bochmann, *Specification of a Simplified Transport Protocol Using Different Formal Description Techniques*, Computer Networks and ISDN Systems, vol. 18, pp.335-377, 1990.

[4] Information Technology - OSI - Specification for the Abstract Syntax Notation One (ASN.1), International Standard ISO/IEC 8824, 1990.

[5] S. Budkowski, P. Dembinski, *An Introduction to Estelle : A Specification Language For Distributed Systems*, Computer Networks and ISDN Systems, vol. 14, pp. 3-23, 1987.

[6] ISO/IEC 9074, *Estelle : A Formal Description Technique Based on an Extended State Transition Model*, 1988.

[7] T. Bolognesi, E.Brinksma, *Introduction to the ISO Specification Language LOTOS*, Computer Networks and ISDN Systems, vol. 14, pp.25-59, 1987.

[8] ISO , *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO/IEC JTC1/SC21, IS 8807, 1988.

[9] F. Belina, D. Hogrefe, *The CCITT Specification and Description Language SDL*, Computer Networks and ISDN Systems, vol. 16, pp. 311-341, 1989.

[10] CCITT , *Specification and Description Language (SDL)*, Recommendation Z.100, CCITT SG X, 1992, 218p.

[11] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, (Springer, Berlin, 1980).

[12] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 1*, (Springer, Berlin, 1985).

[13] B. Forghani, *Automatic Test Suite Derivation From Estelle Specifications*, MS thesis, Concordia University, 1990.

[14] ISO. ISO/IEC 9646-1 : *Conformance Testing Methodology and Framework - Part 1 : General Concepts*, pages 1-31.ISO/IEC JTC/SC21, 1991.

[15] ISO. ISO/IEC 9646-2 : *Conformance Testing Methodology and Framework - Part 2 : Abstract Test Suite Specification*, pages 1-28.ISO/IEC JTC/SC21, 1991.

[16] ISO. ISO/IEC 9646-3 : *Conformance Testing Methodology and Framework - Part 3 : The Tree and Tabular Combined Notation*, pages 1-176.ISO/IEC JTC/SC21, 1991.

[17] ISO. ISO/IEC 9646-4 : *Conformance Testing Methodology and Framework - Part 4 : Test Realization*, pages 1-10.ISO/IEC JTC/SC21, 1991.

[18] ISO. ISO/IEC 9646-5 : *Conformance Testing Methodology and Framework - Part 5 : Requirements on Test Laboratories and Clients for the Conformance Assessment Process*, pages 1-25.ISO/IEC JTC/SC21, 1991.

[19] ISO. ISO/IEC 9646-6 : *Conformance Testing Methodology and Framework - Part 6 : Profile Test Specification*, pages 1-28.ISO/IEC JTC/SC21, 1991.

[20] ISO. ISO/IEC 9646-7 : *Conformance Testing Methodology and Framework - Part 7 : Implementation Conformance Statements*, pages 1-53.ISO/IEC JTC/SC21, 1992.

[21] G. J. Tretmans, *A Formal Approach to Conformance Testing*, PhD thesis, University of Twente, 1992.

[22] ISO/IEC/JTC1/SC21/P.54, *Formal Methods in Conformance Testing*, November 1992.

[23] N. Shiratori, H. Kaminaga, K. Takahashi, S. Noguchi, *A verification Method for LOTOS Specifications and its application*, PSTV IX, North-Holland, pp. 59-70, 1990.

[24] C. A. Vissers, G. Scollo, M. Sinderen, E. Brinksma, *Specification Styles in Distributed Systems Design and Verification*, Theoretical Computer Science, vol. 89, pp. 179-206, 1991.

[25] E. Brinksma, R. Alderen, R. Langerak, J.V. De Lagemaat, J. Tretmans, *A Formal Approach To Conformance Testing*, IFIP Protocol Test Systems, North-Holland, pp. 349-363, 1990.

[26] E. Brinksma, *A Theory for the Derivation of Tests*, IFIP PSTV VIII, North-Holland, pp.63-74, 1988.

[27] P. Tripathy, B. Sarıkaya, *LOTEST : A LOTOS Test Case Generation Tool*, Technical Report, Bilkent University, 1992.

[28] C. Wezeman, *The CO-OP Method for Compositional Derivation of Canonical Testers*, IFIP PSTV IX, North-Holland, Twente, Holland, pp.145-158, 1990.

[29] W. E. Howden, *Functional program testing nd analysis*, McGraw-Hill, 1987.

[30] P. Tripathy, B. Sarıkaya, *Test Generation from LOTOS specifications*, IEEE Transactions on Computers, vol. 40, pp.543-552, 1991.

[31] R. Milner, *A Complete Inference System for a Class of Regular Behaviours*, Journal of Computer and System Sciences, 24, pp.439-466, 1984.

[32] M. S. Hecht, *Flow Analysis of Computer Programs*, New York, North-Holland, 1984.

[33] S. Rapps, E. J. Weyuker, *Selecting Software Test Data Using Data Flow Information*, IEEE Transactions on Software Engineering, vol. 11, pp. 367-375, 1985.

[34] B. Sarıkaya, *Principles of Protocol Engineering and Conformance Testing*, Simon & Schuster, 1993.

[35] A. Wiles, *Specification Styles for SDL in order to Develop Testable Specifications*, ETSI TC-ATM, pp. 1-16, 1992.

[36] D. Hogrefe, *On the development of a standard for conformance testing based on formal specifications*, Computer Standards & Interfaces, vol. 14, pp.185-190, 1992.

[37] D. Hogrefe, *OSI Formal Specification Case Study : the INRES Protocol and Service*, report IAM-91-0/2, University of Bern, 1991.

[38] O. Henniger, B. Sarıkaya, S. Biedlingmaier, *Test suite generation for application layer protocols from formal specifications in Estelle*, Proceedings of 6th IWPTS, Pau, France, September 1993.

[39] B. Sarıkaya, *OSI Conformance Abstract Test Suite Design*, to be published in Computer Networks and ISDN Systems,1993.

[40] G. Cowin, *Experiences in developing a test suite structure and test purposes document for open systems*, Technical Report, National Physical Laboratory, 1991.

[41] M. Weizer, *Program Slicing*, IEEE Transactions on Software Engineering, vol. 10, pp. 352-357, 1984.

[42] H. Agrawal, J. R. Horgan, *Dynamic Program Slicing*, Proceedings of the ACM SIGPLAN' 90 Conference, White Plains, New York, pp. 246-256, 1990.

[43] S. Horwitz, T. Reps, D. Binkley, *Interprocedural Slicing Using Dependence Graphs*, ACM Transactions on Programming Languages and Systems, vol. 12, pp.26-60, 1990.

[44] A. Ulrich, H. König, *Test derivation from LOTOS using structure information*, Proceedings of 6th IWPTS, Pau, France, September 1993.

[45] Information Processing Systems-OSI-*Protocol Specification for the Association Control Service Element*, International Standard, IS 8650, 1988.

[46] Information Technology-OSI-*Protocol Specification for the Association Control Service Element ACSE* - Part 2: Protocol Implementation Conformance Statement (PICS) Proforma, Draft International Standard 8650-2, 1990.

[47] Information Technology-OSI-*Conformance test suite for the ACSE protocol - Part 1 : Test suite structure and test purposes*, Draft International Standard, 1990.

[48] G. V. Bochmann, M. Deslauriers, *Combining ASN.1 support with the LOTOS language*, PSTV IX, North-Holland, pp.175-186, 1990.

[49] ISO/IEC TR 10024, Information Technology - Telecommunications and information exchange between systems - *Formal description of ISO 8073 (Classes 0,1,2,3) in LOTOS*, 1992.

[50] *Abstract Test Suite for Transport Class 2*, The National Computing Center Limited, Manchester, UK, 1988.

[51] B. Sarıkaya, A. Wiles, *Standard conformance test specification language TTCN*, Computer Standards & Interfaces, vol. 14, pp.117-144, 1992.

[52] A. T. Dahbura, K. K. Sabnani, M. U. Uyar, *Formal methods for generating protocol conformance test sequences*, Proceedings of IEEE, vol. 78, pp. 1317-1325, 1990.

Chapter 8

APPENDICES

Appendix A

INRES Base Protocol Specification In LOTOS

The following LOTOS code is the state-oriented formal base specification of INRES Protocol. The specification is relatively different than the one given in [37]. Some modifications have been performed on the original specification in order to bring it in a suitable form for further processing according to the methods defined in previous chapters. The definition of the Medium Service is not included in order to obtain a specification which describes only the behaviour of a protocol entity. The specification is parameterized according to the parameters defined in the PICS Proforma given in Appendix B, and the behaviour and the associated data types related to the Static Conformance Review process, are added. The process named 'Coder', responsible for coding the INRES PDUs in Medium Service Primitives, is removed and its functionality is distributed into the other parts of the specification. The specification is flattened by applying the steps given in Chapter 3. Specifically, the first phase of the chart construction algorithm is applied, the process definitions are rewritten in order to make them statically independent, and guards which are not the first expressions in their composite behaviour expressions, and which depend on input variables are replaced by selection predicates. Finally, in the complete specification given below process identifiers are renamed according to the state tables given in Chapter 4.

```
specification Inres_Protocol[ISAP,MSAP](c1,c2 : Bool):noexit
(* c1, c2 : PICS parameters. *)
(* c1 : Initiator capability is supported *)
(* c2 : Responder capability is supported *)

 library Boolean
 endlib

 type DecNumb is Boolean
   sorts DecNumb
   opns
     0                  :                     -> DecNumb
     s                  : DecNumb             -> DecNumb
     1,2,3,4,5,6,7,8,9  :                     -> DecNumb
     _==_, _<_,
     _<=_, _>=_, _>_    : DecNumb , DecNumb   -> Bool

  eqns forall x,y: DecNumb
    ofsort Bool

      x == x = true;
      s(x) == s(y) = x == y;
      s(x) == 0 = false;
      0 == s(y) = false;
      x < x = false;
      s(x) < s(y) = x < y;
      0 < s(y) = true;
      s(x) < 0 = false;

      x <= y  = (x < y) or (x == y);
      x >= y  = not (x < y) ;
      x > y   = not (x <= y );

    ofsort DecNumb
      1 = s(0);
      2 = s(s(0));
      3 = s(s(s(0)));
      4 = s(s(s(s(0))));
```

```
      5 = s(s(s(s(s(0)))));
      6 = s(s(s(s(s(s(0))))));
      7 = s(s(s(s(s(s(s(0)))))));
      8 = s(s(s(s(s(s(s(s(0))))))));
      9 = s(s(s(s(s(s(s(s(s(0)))))))));
endtype (* DecNumb *)

type ISDUType is
  sorts ISDU
  opns   data1,data2,data3,data4,data5 : -> ISDU
endtype (* ISDUType *)

type Sequencenumber is Boolean
  sorts Sequencenumber
  opns
    0     :                                 -> Sequencenumber
    1     :                                 -> Sequencenumber
    succ  : Sequencenumber                  -> Sequencenumber
    _eq_, _ne_ : Sequencenumber,Sequencenumber -> Bool

  eqns forall a,b : Sequencenumber
    ofsort Sequencenumber
      succ(0) = 1;
      succ(1) = 0;

    ofsort Bool
      0 eq 0 = true;
      1 eq 1 = true;
      0 eq 1 = false;
      1 eq 0 = false;
      0 ne 1 = true;
      1 ne 0 = true;
      0 ne 0 = false;
      1 ne 1 = false;

endtype (* Sequencenumber *)

type InresSpType is Boolean, ISDUType, DecNumb
```

```
  sorts SP
  opns
    ICONreq,ICONconf,IDISind,
    ICONind,ICONresp,IDISreq                    :      -> SP
    IDATreq,IDATind                             : ISDU -> SP
    isICONreq,isICONconf,isIDISind,isIDATreq,
    isIDATind,isICONind,isICONresp,isIDISreq: SP   -> Bool
    data                                        : SP   -> ISDU
    map                                         : SP   -> DecNumb

  eqns forall d : ISDU, sp : SP
      ofsort DecNumb
      map(ICONreq)    = 0;
      map(ICONconf)   = 1;
      map(IDISind)    = 2;
      map(IDATreq(d)) = 3;
      map(IDATind(d)) = 4;
      map(ICONind)    = 5;
      map(ICONresp)   = 6;
      map(IDISreq)    = 7;

    ofsort ISDU
      data(IDATreq(d)) = d;
      data(IDATind(d)) = d;

    ofsort Bool
      isICONreq(sp)  = map(sp) == 0;
      isICONconf(sp) = map(sp) == 1;
      isIDISind(sp)  = map(sp) == 2;
      isIDATreq(sp)  = map(sp) == 3;
      isIDATind(sp)  = map(sp) == 4;
      isICONind(sp)  = map(sp) == 5;
      isICONresp(sp) = map(sp) == 6;
      isIDISreq(sp)  = map(sp) == 7;
endtype (* InresSpType *)

type IPDUType is Boolean, ISDUType, DecNumb, Sequencenumber
  sorts IPDU
```

```
  opns
    CR,CC,DR             :                        -> IPDU
    DT                   : Sequencenumber,ISDU -> IPDU
    AK                   : Sequencenumber       -> IPDU
    isCR,isCC,isDT,
    isAK,isDR            : IPDU                 -> Bool
    data                 : IPDU                 -> ISDU
    num                  : IPDU                 -> Sequencenumber
    map                  : IPDU                 -> DecNumb

  eqns forall f: Sequencenumber, d : ISDU, ipdu : IPDU
    ofsort DecNumb
      map(CR) = 0;
      map(CC) = 1;
      map(DT(f,d)) = 2;
      map(AK(f)) = 3;
      map(DR) = 4;

    ofsort ISDU
      data(DT(f,d)) = d;

    ofsort Sequencenumber
      num(DT(f,d)) = f;
      num(AK(f)) = f;

    ofsort Bool
      isCR(ipdu) = map(ipdu) == 0;
      isCC(ipdu) = map(ipdu) == 1;
      isDT(ipdu) = map(ipdu) == 2;
      isAK(ipdu) = map(ipdu) == 3;
      isDR(ipdu) = map(ipdu) == 4;
endtype (* IPDUType *)

type MediumSpType is Boolean, IPDUType, DecNumb
  sorts MSP
  opns
    MDATreq,MDATind       : IPDU -> MSP
    isMDATreq,isMDATind   : MSP  -> Bool
```

```
    data                      : MSP  -> IPDU
    map                       : MSP  -> DecNumb

  eqns forall d : IPDU, sp : MSP
    ofsort DecNumb
      map(MDATreq(d)) = 8;
      map(MDATind(d)) = 9;

    ofsort IPDU
      data(MDATreq(d)) = d;
      data(MDATind(d)) = d;

    ofsort Bool
      isMDATreq(sp) = map(sp) == 8;
      isMDATind(sp) = map(sp) == 9;
endtype (* MediumSpType *)

type StaticConformance is Boolean
  opns CapabilityConform : Bool, Bool -> Bool

  eqns  forall c1, c2 : Bool
    ofsort Bool
      CapabilityConform (c1, c2) =
           ((c1 and not(c2)) or (not(c1) and c2))
endtype  (* StaticConformance *)

behaviour

 (* Static Conformance Review. *)
 [CapabilityConform (c1, c2)] -> INRES [ISAP,MSAP](c1,c2)

 where

  process INRES[ISAP,MSAP](c1,c2 : Bool):noexit:=
   [c1]-> Initiator[ISAP,MSAP]
  []
   [c2]-> Responder[ISAP,MSAP]
  endproc (* INRES *)
```

```
process Initiator[ISAP,MSAP]:noexit:=
 (hide d in
         Connectionphase_Ini[ISAP,MSAP,d]
 |[d]|   d;Dataphase_Ini[ISAP,MSAP] (succ(0)))
 [>Disconnection_Ini[ISAP,MSAP]
endproc (* Initiator *)

process Connectionphase_Ini[ISAP,MSAP,d]:exit:=
 hide dd in
         Disconnected_Ini[ISAP,MSAP,dd]
 |[dd]|  dd?z:DecNumb;WaitforCC[ISAP,MSAP,d](z)
endproc (* Connectionphase_Ini *)

process Disconnected_Ini[ISAP,MSAP,dd]:exit:=
 ISAP?sp:SP[isICONreq(sp)];MSAP!MDATreq(CR);dd!s(0);exit
[]
 ISAP?sp:SP[not(isICONreq(sp))];
 Disconnected_Ini[ISAP,MSAP,dd]
 (* User errors are ignored (Invalid/Inopportune Beh.) *)
[]
 MSAP?sp:MSP[isMDATind(sp) and not(isDR(data(sp)))];
 Disconnected_Ini[ISAP,MSAP,dd]
 (* DR is only accepted by process Disconnection *)
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* Disconnected_Ini *)

process WaitforCC[ISAP,MSAP,d](z:DecNumb):exit:=
 MSAP?sp:MSP
 [isMDATind(sp) and not(isDR(data(sp))) and isCC(data(sp))];
 ISAP!ICONconf;d;exit
[]
 MSAP?sp:MSP[isMDATind(sp) and not(isDR(data(sp)))
             and not(isCC(data(sp)))];
 WaitforCC[ISAP,MSAP,d](z)
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
 (* DR is only accepted by process Disconnection *)
[]
```

```
   i; (* Timeout *)
    ([z < 4]-> MSAP!MDATreq(CR);WaitforCC[ISAP,MSAP,d](s(z))
    []
     [z == 4]-> ISAP!IDISind;Connectionphase_Ini[ISAP,MSAP,d])
  []
   ISAP?sp:SP[not(isIDISind(sp))];WaitforCC[ISAP,MSAP,d](z)
   (* User errors are ignored (Invalid/Inopportune Beh.) *)
  endproc (* WaitforCC *)

  process Dataphase_Ini[ISAP,MSAP]
          (number : Sequencenumber):noexit:=
   hide d in
          Connected_Ini[ISAP,MSAP,d](number)
          (* 1 is the first Sequencenumber *)
   |[d]|  d?z:DecNumb?number:Sequencenumber?olddata:ISDU;
          Sending[ISAP,MSAP](z,number,olddata)
          (* z is number of sendings. At the beginning z=1 *)
  endproc (* Dataphase_Ini *)

  process Connected_Ini[ISAP,MSAP,d]
          (number:Sequencenumber):exit:=
   ISAP?sp:SP[isIDATreq(sp)];
   MSAP!MDATreq(DT(number,data(sp)));
   d!s(0)!number!data(sp);exit
  []
   ISAP?sp:SP[not(isIDATreq(sp))];
   Connected_Ini[ISAP,MSAP,d](number)
   (* User errors are ignored (Invalid/Inopportune Beh.) *)
  []
   MSAP?sp:MSP[isMDATind(sp) and not(isDR(data(sp)))];
   Connected_Ini[ISAP,MSAP,d](number)
   (* DR is only accepted by process Disconnection *)
   (* System errors are ignored (Invalid/Inopportune Beh.) *)
  endproc (* Connected_Ini *)

  process Sending[ISAP,MSAP]
       (z:DecNumb,number:Sequencenumber, olddata:ISDU):noexit:=
   MSAP?sp:MSP
```

```
   [isMDATind(sp) and not(isDR(data(sp))) and
    isAK(data(sp)) and (num(data(sp)) eq number)];
   Dataphase_Ini[ISAP,MSAP](succ(number))
  []
   MSAP?sp:MSP
   [isMDATind(sp) and not(isDR(data(sp))) and isAK(data(sp))
    and (num(data(sp)) ne number) and (z < 4)];
   MSAP!MDATreq(DT(number,olddata));
   Sending[ISAP,MSAP](s(z),number,olddata)
  []
   MSAP?sp:MSP
   [isMDATind(sp) and not(isDR(data(sp))) and isAK(data(sp))
    and (num(data(sp)) ne number) and (z == 4)];
   ISAP!IDISind;Initiator[ISAP,MSAP]
   (* The Initiator shall not resend more than 4 times *)
   (* in case of faulty transmission *)
  []
   MSAP?sp:MSP
   [isMDATind(sp) and not(isDR(data(sp)))
    and not(isAK(data(sp)))];
   Sending[ISAP,MSAP](z,number,olddata)
   (* System errors are ignored (Invalid/Inopportune Beh.) *)
  []
   i; (* Timeout *)
    ([z < 4]-> MSAP!MDATreq(DT(number,olddata));
               Sending[ISAP,MSAP](s(z),number,olddata)
    []
     [z == 4]-> ISAP!IDISind;Initiator[ISAP,MSAP])
 []
  ISAP?sp:SP[not(isIDATreq(sp))];
  Sending[ISAP,MSAP](z,number,olddata)
  (* User errors are ignored (Invalid/Inopportune Beh.) *)
 endproc (* Sending *)

process Disconnection_Ini[ISAP,MSAP]:noexit:=
 MSAP?sp:MSP[isMDATind(sp) and isDR(data(sp))];ISAP!IDISind;
 Initiator[ISAP,MSAP]
endproc (* Disconnection_Ini *)
```

```
process Responder[ISAP,MSAP]:noexit:=
 (hide d in
         Connectionphase_Res[ISAP,MSAP,d]
 |[d]|   d;Dataphase_Res[ISAP,MSAP](succ(1)))
 [>Disconnection_Res[ISAP,MSAP]
endproc (* Responder *)

process Connectionphase_Res[ISAP,MSAP,d]:exit:=
 hide dd in
         Disconnected_Res[ISAP,MSAP,dd]
 |[dd]|  dd;WaitforICONresp1[ISAP,MSAP,d]
endproc (* Connectionphase_Res *)

process Disconnected_Res[ISAP,MSAP,dd]:exit:=
 MSAP?sp:MSP[isMDATind(sp) and isCR(data(sp))];
 ISAP!ICONind;dd;exit
[]
 MSAP?sp:MSP[isMDATind(sp) and not(isCR(data(sp)))];
 Disconnected_Res[ISAP,MSAP,dd]
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
[]
 ISAP?sp:SP[isICONresp(sp)];Disconnected_Res[ISAP,MSAP,dd]
 (* User errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* Disconnected_Res *)

process WaitforICONresp1[ISAP,MSAP,d]:exit:=
 ISAP?sp:SP[isICONresp(sp)];MSAP!MDATreq(CC);d;exit
[]
 MSAP?sp:MSP[isMDATind(sp)];WaitforICONresp1[ISAP,MSAP,d]
 (* System errors are ignored (Invalid/Inopportune Beh.) *)
endproc (* WaitforICONresp1 *)

process Dataphase_Res[ISAP,MSAP]
        (number : Sequencenumber):noexit:=
 (* number is the last acknowledged Sequencenumber *)
 hide d in
        Connected_Res[ISAP,MSAP,d](number)
```

```
  |[d]|  d;WaitforICONresp2[ISAP,MSAP]
 endproc (* Dataphase_Res *)

 process Connected_Res[ISAP,MSAP,d]
        (number : Sequencenumber):exit:=
  MSAP?sp:MSP
  [isMDATind(sp) and isDT(data(sp))
   and (num(data(sp)) eq succ(number))];
  ISAP!IDATind(data(data(sp)));
  MSAP!MDATreq(AK(num(data(sp))));
  Connected_Res[ISAP,MSAP,d](succ(number))
 []
  MSAP?sp:MSP
  [isMDATind(sp) and isDT(data(sp))
   and (num(data(sp)) eq number)];
  MSAP!MDATreq(AK(num(data(sp))));
  Connected_Res[ISAP,MSAP,d](number)
 []
  MSAP?sp:MSP[isMDATind(sp) and isCR(data(sp))];
  ISAP!ICONind;d;exit
 []
  MSAP?sp:MSP
  [isMDATind(sp) and not(isDT(data(sp)) or isCR(data(sp)))];
  Connected_Res[ISAP,MSAP,d](number)
  (* System errors are ignored (Invalid/Inopportune Beh.) *)
 []
  ISAP?sp:SP[isICONresp(sp)];
  Connected_Res[ISAP,MSAP,d](number)
  (* User errors are ignored (Invalid/Inopportune Beh.) *)
 endproc (* Connected_Res *)

 process WaitforICONresp2[ISAP,MSAP]:noexit:=
  ISAP?sp:SP[isICONresp(sp)];MSAP!MDATreq(CC);
  Dataphase_Res[ISAP,MSAP](succ(1))
 []
  MSAP?sp:MSP[isMDATind(sp)];WaitforICONresp2[ISAP,MSAP]
  (* System errors are ignored (Invalid/Inopportune Beh.) *)
 endproc (* WaitforICONresp2 *)
```

```
  process Disconnection_Res[ISAP,MSAP]:noexit:=
   ISAP?sp:SP[isIDISreq(sp)];MSAP!MDATreq(DR);
   Responder[ISAP,MSAP]
  endproc (* Disconnection_Res *)

endspec
```

Appendix B

PICS Proforma For INRES

B.1 Date of Statement

1	Date of statement yy-mm-dd

B.2 Implementation Details

The information necessary to uniquely identify the implementation and the system in which it may reside.

a) Supplier, implementation name, operating system, suitable hardware.
b) System supplier and/or client of the test laboratory.
c) Information on whom to contact concerning the contents of the PICS.

1	

B.3 Global Statement of Conformance

1	Are all mandatory features implemented ? Yes or No

NOTE - If a positive response is not given to this box, then the implementation does not confirm to the INRES protocol.

B.4 Initiator/Responder Capability

No	Capability	D I	
1	Initiator	o.1	
2	Responder	o.1	

NOTE - o.1 : Exactly one of the capabilities is mandatory for a system claiming conformance.

B.5 Supported PDUs

No	IPDU	Transmission D I	Reception D I
1	Connect Request IPDU (CR)	c1	c2
2	Connect Confirm IPDU (CC)	c2	c1
3	Data Transfer IPDU (DT)	c1	c2
4	Acknowledgement IPDU (AK)	c2	c1
5	Disconnect Request IPDU (DR)	c2	c1

NOTE - c1 : if 4/1 then m else -
c2 : if 4/2 then m else -

B.6 Supported PDU Parameters

This section identifies the parameters supported in INRES PDUs. The supplier shall indicate all of the parameters supported for which conformance is claimed, and also list any associated limitations.

B.6.1 Data Transfer IPDU (DT)

No	Sending Parameter		Value	
		D I	Allowed	Supported
1	Sequence Number	m		
2	User Information	m		

No	Receiving Parameter		Value	
		D I	Allowed	Supported
3	Sequence Number	m		
4	User Information	m		

B.6.2 Acknowledgement IPDU (AK)

No	Sending Parameter		Value	
		D I	Allowed	Supported
1	Sequence Number	m		

No	Receiving Parameter		Value	
		D I	Allowed	Supported
2	Sequence Number	m		

B.7 Timers

No	Timer	D I	Value	
			Allowed	Supported
1	Retransmission Timer	m		

Appendix C

Test Suite Structure and Test Purposes for INRES

C.1 Initiator (I)

Test Group Objective : To test the IUT in the role of initiator.

Subgroups :

1. Basic Interconnection and Capability Tests (BIC)
2. Valid Behaviour Tests (BV)
3. Invalid & Inopportune Behaviour Tests (BIO)

C.1.1 I/Basic Interconnection and Capability Tests (BIC)

Test Group Objective : To provide limited testing of each of the conformance requirements for INRES, to ascertain what capabilities of the IUT can be observed, and to check that those observable capabilities are valid with respect to the static conformance requirements and the PICS.

Test Purposes :

1. Test the IUT's ability to issue a CR and receive a CC (I/BV/CE/1).
2. Test the IUT's ability to issue a DT and receive an AK (I/BV/DT/1).

C.1.2 I/Valid Behaviour Tests (BV)

Test Group Objective : Behaviour tests test an implementation as throughly as is practical, over the full range of dynamic conformance requirements. Tests are included to check valid behaviour by the IUT in response to valid behaviour by the real tester.

Subgroups :

1. Connection Establishment (CE)
2. Data Transfer (DT)
3. Disconnection (DC)

C.1.2.1 I/BV/Connection Establishment (CE)

Test Group Objective : To test the connection establishment procedures by having the IUT generate a CR.

Test Purposes :

1. Respond with CC.Check that the IUT established the connection.
2. Respond with DR. Check that the IUT did not establish the connection.
3. Wait until the related timer expires. Check that the IUT retransmits the CR.
4. Cause the timer expire four times. Check that the IUT releases the connection

C.1.2.2 I/BV/Data Transfer (DT)

Test Group Objective : To test the data_transfer procedures of the IUT in the role of sender, by having it generate a DT.

Test Purposes :

1. Accept the data sent and send an AK IPDU. Check that IUT behaves accordingly.
2. Reject the data sent by the IUT and respond with a negative AK IPDU. Check that the IUT retransmits the same piece of data.
3. Reject the data sent by the IUT four times. Check that the IUT releases the connection.
4. Wait until the related timer expires. Check that the IUT retransmits the data.
5. Cause the timer expire four times. Check that the IUT releases the connection.

C.1.2.3 I/BV/Disconnection (DC)

Test Group Objective : To test the release of a connection by having the IUT receive a DR IPDU for each ICPM (Inres Control Protocol Machine) state.

Test Purposes :

1. Disconnected State.
2. WaitforCC State.
3. Connected State.
4. Sending State.

C.1.3 I/Invalid & Inopportune Behaviour Tests (BIO)

Test Group Objective : To check valid behaviour by the IUT in response to invalid and inopportune behaviour by the real tester. The subgroups are on a state by state basis. For each state the IPDUs which are not valid for the given state will be presented to the IUT.

Subgroups :

1. Disconnected State (STA0)
2. WaitforCC State (STA1)
3. Connected State (STA2)
4. Sending State (STA3)

C.1.3.1 I/BIO/Disconnected State (STA0)

Test Group Objective: Test the IUT's reaction to invalid/inopportune IPDUs in the Disconnected State.

Test Purposes :

1. CC IPDU.
2. AK IPDU.

C.1.3.2 I/BIO/WaitforCC State (STA1)

Test Group Objective : Test the IUT's reaction to invalid/inopportune IPDUs in the WaitforCC State.

Test Purposes :

1. AK IPDU.

C.1.3.3 I/BIO/Connected State (STA2)

Test Group Objective: Test the IUT's reaction to invalid/inopportune IPDUs in the Connected State.

Test Purposes :

1. CC IPDU.
2. AK IPDU.

C.1.3.4 I/BIO/Sending State (STA3)

Test Group Objective: Test the IUT's reaction to invalid/inopportune IPDUs in the Sending State.

Test Purposes :

1. CC IPDU.
2. AK IPDU with an invalid parameter value.

C.2 Responder (R)

Test Group Objective : To test the IUT in the role of responder.

Subgroups :

1. Basic Interconnection and Capability Tests (BIC)
2. Valid Behaviour Tests (BV)
3. Invalid & Inopportune Behaviour Tests (BIO)

C.2.1 R/Basic Interconnection and Capability Tests (BIC)

Test Group Objective : To provide limited testing of each of the conformance requirements for INRES, to ascertain what capabilities of the IUT can be observed, and to check that those observable capabilities are valid with respect to the static conformance requirements and the PICS.

Test Purposes :

1. Test the IUT's ability to receive a CR and issue a CC (R/BV/CE/1).
2. Test the IUT's ability to receive a DT and issue an AK (R/BV/DT/1).
3. Test the IUT's ability to issue a DR (R/BV/DC).

C.2.2 R/Valid Behaviour Tests (BV)

Test Group Objective : To test the valid behaviour of the IUT in response to valid behaviour by the real tester.

Subgroups :

1. Connection Establishment (CE)
2. Data Transfer (DT)
3. Disconnection (DC)

C.2.2.1 R/BV/Connection Establishment (CE)

Test Group Objective : To test the connection establishment procedures by attempting to establish a connection to the IUT (by sending a CR IPDU).

Test Purposes :

1. The service user at the IUT side accepts the connection. Check the IUT sends a CC, and establishes the connection.
2. The service user at the IUT side rejects the connection. Check the IUT sends a DR.

C.2.2.2 R/BV/Data Transfer (DT)

Test Group Objective : To test the data_transfer procedures of the IUT in the role of receiver (by sending a DT IPDU).

Test Purposes :

1. Present the IUT an in_sequence data (DT IPDU), and check the IUT responds with appropriate AK IPDU and delivers the data to its service_user.
2. Present the IUT an out_of_sequence data item, and check the IUT behaves appropriately.
3. Present the IUT with a CR IPDU. Check that it informs its user for the (new) connection request.

C.2.2.3 R/BV/Disconnection (DC)

Test Group Objective : To test the release of a connection by having the IUT receive an IDISreq primitive for each ICPM state.

Test Purposes :

1. Disconnected State.
2. WaitforICONresp State.
3. Connected State.

C.2.3 R/Invalid & Inopportune Behaviour Tests (BIO)

Test Group Objective : To check valid behaviour by the IUT in response to invalid and inopportune behaviour by the real tester. The subgroups are on a state by state basis. For each state the IPDUs which are not valid for the given state will be presented to the IUT.

Subgroups :

1. Disconnected State (STA0)
2. WaitforICONresp State (STA1)
3. Connected State (STA2)

C.2.3.1 R/BIO/Disconnected State (STA0)

Test Group Objective: Test the IUT's reaction to invalid/inopportune IPDUs in the Disconnected State.

Test Purposes :

1. DT IPDU.

C.2.3.2 R/BIO/WaitforICONresp State (STA1)

Test Group Objective : Test the IUT's reaction to invalid/inopportune IPDUs in the WaitforICONresp State.

Test Purposes :

1. DT IPDU.
2. CR IPDU.

C.2.3.3 R/BIO/Connected State (STA2)

Test Group Objective: Test the IUT's reaction to invalid/inopportune IPDUs in the Connected State.

Test Purposes :

1. DT IPDU with an invalid parameter value.

Appendix D

ACSE Base Protocol Specification In LOTOS

This Appendix contains the behaviour part of the LOTOS code of the ACSE protocol. The given specification is in the state-oriented style defined in [24]. Since it has a total of about 2500 lines of LOTOS code, out of which 90% consists of abstract data types, only the behaviour part is reproduced here. As in the case of INRES protocol, the specification is flattened by applying the steps mentioned in Chapter 3, and some behaviour expressions and data types related to PICS parameterization have been added.

```
specification ACSE_Protocol[A,P](c1,c2,c3 : Bool):noexit
(* c1, c2, c3 : PICS parameters. *)
(* c1 :  Association initiator capability is supported *)
(* c2 :  Association responder capability is supported *)
(* c3 :  RLRQ APDU is supported for transmission *)

(* Abstract Data Types *)

 behaviour

  (* Static Conformance Review *)
  [CapabilityConform (c1, c2)] -> ACSE[A,P](c1,c2,c3)

  where
```

```
process ACSE[A,P](c1,c2,c3 : Bool):noexit:=
 hide d in
        Assoc_Estab[A,P,d](c1,c2,c3)
 |[d]|  d?c:calltype;(Normal_Rel[A,P](c,c1,c2,c3)
                      [> Abort[A,P](c1,c2,c3))
endproc (* ACSE *)

process Assoc_Estab[A,P,d](c1,c2,c3 : Bool):exit:=
 hide dd in
         Unassociated[A,P,dd](c1,c2,c3)
|[dd]|   (dd?f:calltype[f = calling];
          AwaitAARE[A,P,d](c1,c2,c3)
         []
          dd?f:calltype[f = called];
          AwaitAASCrsp[A,P,d](c1,c2,c3))
endproc (* Assoc_Estab *)

process Unassociated[A,P,dd](c1,c2,c3 : Bool):exit:=
 A ? x : primitive [IsAASCreq(x) and c1];
 P ! Out ! PCONreq(make_AARQ(get_AASCreq(x)));
 dd!calling;exit
[]
 P ! Input ? x : primitive
 [IsPCONind(x) and IsAARQ(user_data(x))
  and not(common_prot_version(get_AARQ(user_data(x))))
  and c2];
 P ! Out ! PCONrspR(make_AARE(get_AARQ(user_data(x))));
 Unassociated[A,P,dd](c1,c2,c3)
[]
 P ! Input ? x : primitive
 [IsPCONind(x) and IsAARQ(user_data(x))
  and common_prot_version(get_AARQ(user_data(x)))
  and c2];
 A ! make_AASCind(get_AARQ(user_data(x)));
 dd!called;exit
[]
 P ! Input ? x : primitive
 [IsPCONind(x) and not(IsAARQ(user_data(x)))];
```

```
    protocol_error[A,P](c1,c2,c3)
    (* Invalid Behaviour *)
   endproc (* Unassociated *)

   process AwaitAARE[A,P,d](c1,c2,c3 : Bool):exit:=
    (P ! Input ? x : primitive
     [IsPCONcnfA(x) and IsAARE(user_data(x)) and
      (result(get_AARE(user_data(x))) eq accepted)];
     A ! make_AASCcnfU(get_AARE(user_data(x)));
     d!calling;exit
   [] (* Source : AC-peer (ACSE Service User Rejection) *)
    P ! Input ? x : primitive
    [IsPCONcnfUR(x) and IsAARE(user_data(x)) and
     not(result(get_AARE(user_data(x))) eq accepted)
     and IsAssociate_source_diagnostic_genere_0
     (result_source_diagnostic(get_AARE(user_data(x))))];
    A ! make_AASCcnfU(get_AARE(user_data(x)));
    Assoc_Estab[A,P,d](c1,c2,c3)
   [] (* Source : AC-peer (ACSE Service Provider Rejection) *)
    P ! Input ? x : primitive
    [IsPCONcnfUR(x) and IsAARE(user_data(x)) and
     not(result(get_AARE(user_data(x))) eq accepted)
     and IsAssociate_source_diagnostic_genere_1
     (result_source_diagnostic(get_AARE(user_data(x))))];
    A ! make_AASCcnfP(get_AARE(user_data(x)));
    Assoc_Estab[A,P,d](c1,c2,c3)
   [] (* Source : PS-provider (Provider Rejection) *)
    P ! Input ! PCONcnfPR;
    A ! make_AASCcnf;
    Assoc_Estab[A,P,d](c1,c2,c3)
   []
    P ! Input ? x : primitive
    [IsPCONcnfA(x) and not(IsAARE(user_data(x)))];
    protocol_error[A,P](c1,c2,c3))
    (* Invalid Behaviour *)
   [>
    Abort[A,P](c1,c2,c3)
   endproc (* AwaitAARE *)
```

```
process AwaitAASCrsp[A,P,d](c1,c2,c3 : Bool):exit:=
 (A ? x : primitive
  [IsAASCrsp(x) and (result(get_AASCrsp(x)) eq accepted)];
  P ! Out ! PCONrspA(make_AARE(get_AASCrsp(x)));
  d!called;exit
[]
 A ? x : primitive
 [IsAASCrsp(x) and not(result(get_AASCrsp(x)) eq accepted)];
 P ! Out ! PCONrspR(make_AARE(get_AASCrsp(x)));
 Assoc_Estab[A,P,d](c1,c2,c3))
[>
  Abort[A,P](c1,c2,c3)
endproc (* AwaitAASCrsp *)

process Normal_Rel[A,P](c:calltype,c1,c2,c3:Bool):noexit:=
 hide d in
        Associated[A,P,d](c1,c2,c3)
|[d]|  (d?f:calltype[f = calling];
        WaitforRLRE[A,P](c,c1,c2,c3)
       []
        d?f:calltype[f = called];
        AwaitARLSrsp[A,P](c,c1,c2,c3))
endproc (* Normal_Rel *)

process Associated[A,P,d](c1,c2,c3 : Bool):exit:=
 A ? x : primitive [IsARLSreq(x) and c3];
 P ! Out ! PRLSreq(make_RLRQ(get_ARLSreq(x)));
 d!calling;exit
[]
 P ! Input ? x : primitive
 [IsPRLSind(x) and IsRLRQ(user_data(x))];
 A ! make_ARLSind(get_RLRQ(user_data(x)));
 d!called;exit
[]
 P ! Input ? x : primitive
 [IsPRLSind(x) and not(IsRLRQ(user_data(x)))];
 protocol_error[A,P](c1,c2,c3)
```

```
  (* Invalid Behaviour *)
 endproc (* Associated *)

 process WaitforRLRE[A,P](c:calltype,c1,c2,c3:Bool):noexit:=
  hide d in
         AwaitRLRE[A,P,d](c,c1,c2,c3)
 |[d]|   d;([c = calling] ->
               Collision_association_initiator[A,P](c1,c2,c3)
           []
            [c = called]  ->
               Collision_association_responder[A,P](c1,c2,c3))
 endproc (* WaitforRLRE *)

 process AwaitRLRE[A,P,d](c:calltype,c1,c2,c3 : Bool):exit:=
  P ! Input ? x : primitive
  [IsPRLScnfA(x) and IsRLRE(user_data(x))];
  A ! make_ARLScnfA(get_RLRE(user_data(x)));
  ACSE[A,P](c1,c2,c3)
 []
  P ! Input ? x : primitive
  [IsPRLScnfR(x) and IsRLRE(user_data(x))];
  A ! make_ARLScnfR(get_RLRE(user_data(x)));
  Normal_Rel[A,P](c,c1,c2,c3)
 []
  P ! Input ? x : primitive
  [IsPRLSind(x) and IsRLRQ(user_data(x))];
  A ! make_ARLSind(get_RLRQ(user_data(x)));
  d;exit
 []
  P ! Input ? x : primitive
  [IsPRLSind(x) and not(IsRLRQ(user_data(x)))];
  protocol_error[A,P](c1,c2,c3)
  (* Invalid Behaviour *)
 []
  P ! Input ? x : primitive
  [IsPRLScnfA(x) and not(IsRLRE(user_data(x)))];
  protocol_error[A,P](c1,c2,c3)
  (* Invalid Behaviour *)
```

```
endproc (* AwaitRLRE *)

process Collision_association_initiator[A,P]
        (c1,c2,c3 : Bool):noexit:=
 A ? x : primitive
 [IsARLSrsp(x) and (result(get_ARLSrsp(x)) eq affirmative)];
 P ! Out ! PRLSrspA(make_RLRE(get_ARLSrsp(x)));
 WaitforRLRE[A,P](calling, c1,c2,c3)
endproc (* Collision_association_initiator *)

process Collision_association_responder[A,P]
        (c1,c2,c3 : Bool) : noexit :=
 P ! Input ? x : primitive
 [IsPRLScnfA(x) and IsRLRE(user_data(x))];
 A ! make_ARLScnfA(get_RLRE(user_data(x)));
 AwaitARLSrsp[A,P](called,c1,c2,c3)
[]
 P ! Input ? x : primitive
 [IsPRLScnfA(x) and not(IsRLRE(user_data(x)))];
 protocol_error[A,P](c1,c2,c3)
 (* Invalid Behaviour *)
endproc (* Collision_association_responder *)

process AwaitARLSrsp[A,P](c:calltype, c1,c2,c3:Bool):noexit:=
 A ? x : primitive
 [IsARLSrsp(x) and (result(get_ARLSrsp(x)) eq affirmative)];
 P ! Out ! PRLSrspA(make_RLRE(get_ARLSrsp(x)));
 ACSE[A,P](c1,c2,c3)
[]
 A ? x : primitive
 [IsARLSrsp(x) and (result(get_ARLSrsp(x)) eq negative)];
 P ! Out ! PRLSrspR(make_RLRE(get_ARLSrsp(x)));
 Normal_Rel[A,P](c,c1,c2,c3)
endproc (* AwaitARLSrsp *)

process Abort[A,P](c1,c2,c3 : Bool):noexit:=
 A ? x : primitive [IsAABRreq(x)];
 P ! Out ! PUABreq(make_ABRT(get_AABRreq(x)));
```

```
 ACSE[A,P](c1,c2,c3)
[]
 P ! Input ? x : primitive
 [IsPUABind(x) and IsABRT(user_data(x))];
 A ! make_AABRind(get_ABRT(user_data(x)));
 ACSE[A,P](c1,c2,c3)
[]
 P ! Input ! PPABind;
 A ! make_APABind;
 ACSE[A,P](c1,c2,c3)
[]
 P ! Input ? x : primitive
 [IsPUABind(x) and not(IsABRT(user_data(x)))];
 protocol_error[A,P](c1,c2,c3)
 (* Invalid Behaviour *)
endproc (* Abort *)

process protocol_error[A,P](c1,c2,c3 : Bool):noexit:=
 A ! make_AABRind;
 P ! Out ! PUABreq(make_ABRT);
 ACSE[A,P](c1,c2,c3)
endproc (* protocol_error *)

endspec
```