A Unified Model for Protocol Test Suite Design

Priyadarshi Tripathy

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirement

for the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

November, 1992

ISBN 0-315-84672-0

Canada

ABSTRACT

A Unified Model for Protocol Test Suite Design

Priyadarshi Tripathy Ph. D.,

Concordia University, 1992.

This thesis is concerned with developing new algorithms for solving some basic problems of conformance testing. In particular, the following problems of conformance testing are considered: *i*) generation of test cases from **L**anguage **O**f **T**emporal **O**rdering **S**pecification (LOTOS) and **S**pecification and **D**escription **L**anguage (SDL), *ii*) selection of test cases which meet certain data flow coverage criteria, and *iii*) representation of test cases for **L**ocal **S**ingle-layer (LS) and **R**emote **S**ingle-layer (RS) architectures. The algorithms presented in this thesis can be used to solve in an efficient manner these fundamental problems of conformance testing. These algorithms rely heavily on two concepts: the **E**xtended **F**inite **S**tate **M**achine (EFSM) *chart* and the **I**nput/**O**utput (I/O) diagram. In this thesis, we introduce a unified model (using the EFSM chart and the I/O diagram) for existing protocol specification languages. Based on the new unified model, a conceptually simple, easy to implement and computationally efficient methodology is proposed in this thesis for studying conformance testing.

In this thesis, the protocol specification is mapped into an EFSM chart. The structure of input/output data is modeled by hierarchical diagrams called I/O diagrams. Test cases are generated from the EFSM chart. Furthermore, a data flow graph is constructed from the chart, and used to identify the protocol functions for testing the data flow aspects of an **I**mplementation **U**nder **T**est (IUT). The zero-one integer programming technique is used to select test cases to meet the data flow coverage requirement. The selected test cases are modeled as a dependency graph and then evaluated by taking predicate slices from the test case dependency graph. Predicate slices are used to identify infeasible test cases that must be eliminated. Redundant assignments and predicates in all the feasible test cases are removed by reducing the test cases. Reduction is achieved by using the test case dependency graph as well as the data flow graph. The reduced test case dependency

graph is adapted for LS and RS architectures. The tester's behaviour in each test case is obtained by a series of transformations called representation and selection. Test case representation refers to the steps of inverting the direction of events and the generation of base and dynamic constraints on the events. These constraints are generated in the form of an I/O diagram. Test case selection refers to the steps of assigning a test purpose according to the hierarchy of test cases in a test suite and then completing the tester's behaviour by assigning verdict and parameter value information.

TO MY PARENTS

PROFESSOR KUNJABEHARI AND SUREKHA

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor B. Sarikaya, for his continued guidance, suggestions, encouragement and his careful reading of the manuscript and providing criticism on writing. Without his supervision, this work would not have been possible. I owe my knowledge of protocol testing to him.

I would like to thank Professor G.v.Bochmann of University of Montreal for financial assistance. I would also like to thank Professor J. W. Atwood for going through the thesis and improving the technical quality of the manuscript.

It is with great pleasure that I acknowledge the support and motivation provided by all my colleagues of Room No. H847, and in particular Sagar, Raghu, Costas, Aparna and Hamid. I also wish to acknowledge Ashima for her careful reading and improving the English of this thesis.

Last but not the least, it is with pride that I acknowledge the support and inspiration of my parents Professor Kunjabehari and Surekha, my brothers Bipeen and Banabehari, my sisters Kannan, Nalini, Gitanjali, Krishna and Yasodhara and my brothers-in-law Benoy, Pratap, Debendra, Rajendra and Biju —back home.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ACSE	Association Control Service Elements
ACT	Asynchronous Communication Tree
ADT	Abstract Data Type
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
CCITT	International Consultative Committee for Telephones and Telegraphs
CCS	Calculus of Communicating Systems
CFBR	Control Flow Behaviour Representation
CS	Coordinated Single-layer
CSE	Coordinated Single-layer Embedded
CM	Coordinated Multi-layer
DFG	Data Flow Graph
DS	Distributed Single-layer
DM	Distributed Multi-layer
EFSM	Extended Finite State Machine
FDT	Formal Description Technique
FSM	Finite State Machine
HDLC	High-level Data Link Control
I/O	Input/Output
ISO	International Organization for Standardization
IUT	Implementation Under Test
LAPB	Link Access Procedure
LM	Local Multi-layer
LOTOS	Language Of Temporal Ordering Specification
LS	Local Single-layer
LSE	Local Single-layer Embedded
PCO	Point of Control and Observation

PDU	Protocol Data Unit
PIXIT	Protocol Implementation eXtra Information for Testing
REBC	Receive Event Base Constraint
RM	Remote Multi-layer
RS	Remote Single-layer
RSE	Remote Single-layer Embedded
RTCDG	Reduced Test Case Dependency Graph
SAP	Service Access Point
SDL	Specification and Description Language
SEBC	Send Event Base Constraint
TMP	Test Management Protocol
TN	Transition Number
TTCN	Tree Table Combined Notation

LIST OF FIGURES AND TABLES

CHAPTER I

INTRODUCTION

A computer communication protocol, or simply protocol, is a set of rules that govern the communication and interaction between various components in a distributed system. In order to organize the complexity of these rules, they are usually partitioned into a hierarchical structure of protocol layer entities (Figure 1.1), each built upon its predecessor layer.

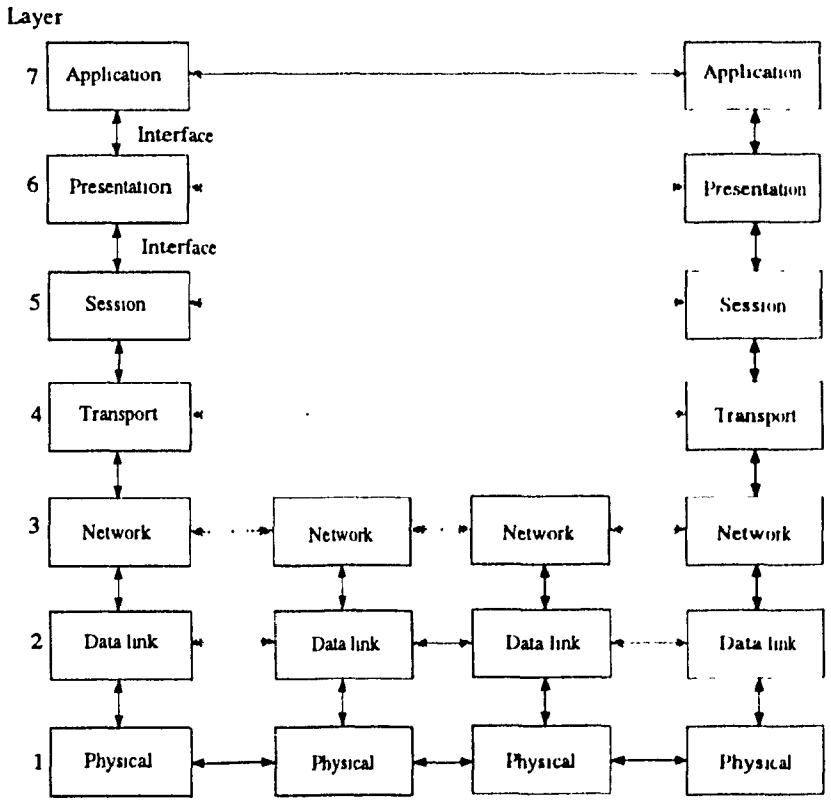


Figure 1.1 Protocol layers.

Although the number of layers may differ from network to net work, the purpose of an entity at layer N is to provide certain services, called (N)-services, to its upper layer entity using the services provided by the (N–1) layer while isolating the implementation details of the lower entities from the upper layers. Peer (N)-entities communicate with

each other through the (N–1)-service provider by using the (N)-protocol (Figure 1.2). An (N)-protocol defines the rules and conventions for the communication between two (N)-entities.

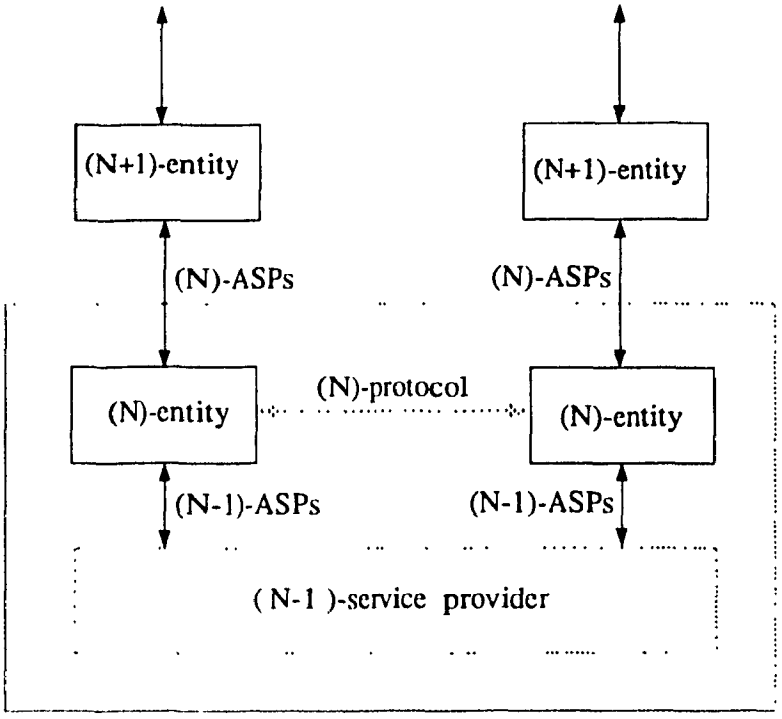


Figure 1.2 Abstraction of an entity in a multi-entity model.

Protocol design is not a new problem. The informal techniques traditionally used for designing and implementing many practical protocols have been largely successful, but also yield a disturbing number of errors or unexpected and undesirable behaviours in most protocols. The protocols being developed today are larger and more complicated than even before, and the automation of the whole design process is highly desirable. The fundamental and challenging problem that a protocol designer now faces is how to design a large set of communication and interaction rules for information exchange in such a way that the rules are minimum, logically consistent, complete and efficiently implemented.

Recently, formal methods and software engineering methodologies have been extensively applied to protocol design. As a result, a new field called protocol engineering [39]

has emerged. The domain of protocol engineering includes protocol and service specification, protocol validation and verification, protocol synthesis, protocol conversion, performance analysis, automatic implementation and conformance testing (see Figure 1.3). Protocol specification plays a key role in all these activities, and the use of formal specification allows the partial automation of some of these activities.

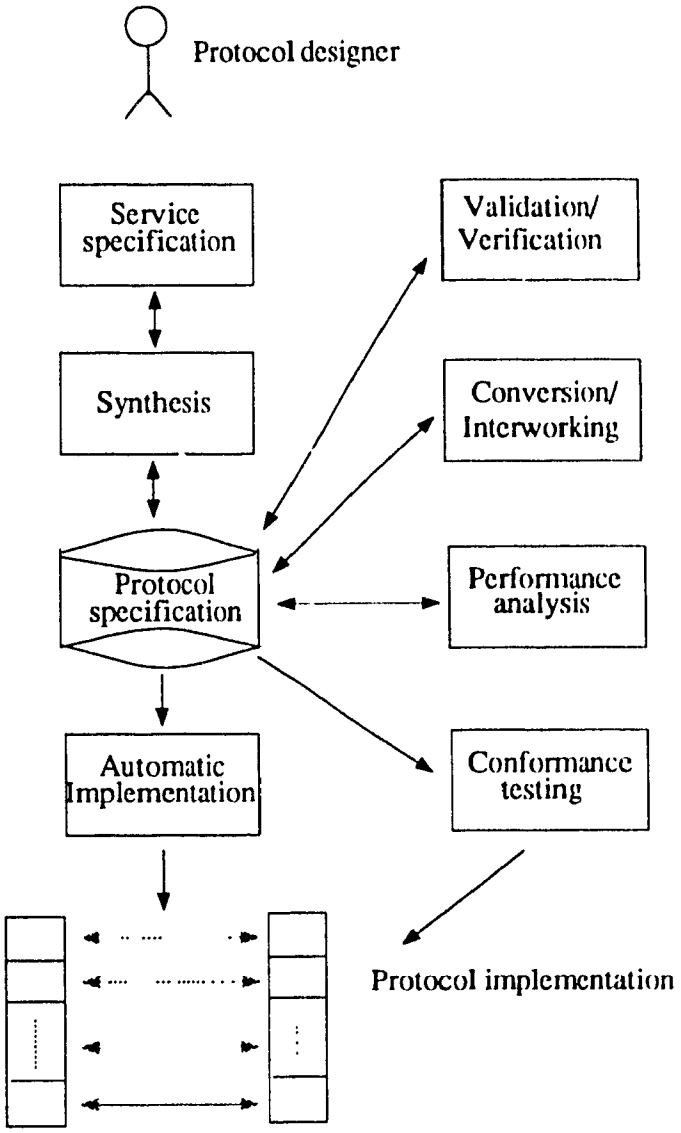


Figure 1.3 Domain of protocol engineering.

1.1 FORMAL SPECIFICATION

Many different formal specification languages have been developed for various purposes, and many of them have been applied to the description of distributed systems and communication protocols. The most important approaches are finite state machines (FSM), formal grammars, Petri nets, algebraic calculi, abstract data types, programming languages, logic programming and temporal logic. Various extensions of the above approaches have been defined by combining them with programming language or abstract data type approaches for the description of parameter values. In addition to the above formal methods, CCITT and ISO have developed so-called Formal Description Techniques (FDTs) for the description of protocol and services, namely Estelle [7], LOTOS [4] [26], and SDL [2], [9].

In Estelle, the specification module is modeled by an extended finite state machine. The interaction parameters and state variables are covered by type definitions, expressions and statements of the Pascal programming language. In addition, certain Estelle statements cover aspects related to the creation of the overall system structure consisting, in general, of a hierarchy of module instances. Communication between modules takes place through the interaction points of the module, which have been interconnected by the parent module. Communication is asynchronous, that is, an output message is stored in an input queue of the receiving module before it is processed.

LOTOS, a process algebraic language, is a combination of the Calculus of Communicating Systems (CCS) [41] formalism for behaviour description and Abstract Data Types (ADTs) called ACT ONE [13] for data description. A set of *composition rules* are used to derive larger specifications from the primitive notions of *event* and *processes*. A set of processes communicate among themselves through synchronization. Rather than simple value passing as in Estelle, LOTOS processes can agree on a common value when rendezvous is achieved.

SDL has the longest history. A subset of the present language was already recommended by CCITT in 1980. It is based on an EFSM model. For interaction parameters

and state variables, it uses the concepts of abstract data types with the addition of a notation of program variables and data structures, similar to what is included in Estelle. The communication is asynchronous and the destination process of an output message can be identified by various means, including process identifier or channel names. In contrast to other FDTs, SDL was developed, right from the beginning, with an orientation towards a graphical representation. The language includes graphical elements for the FSM aspects of a process and the overall structure of a specification.

In addition to FDTs described above, CCITT and ISO developed semiformal techniques Abstract Syntax Notation 1 (ASN.1) [28] for interaction parameters and Tree Tabular Combined Notation (TTCN) [30-31] for specifying test suites.

The Abstract Syntax Notation One was originally developed in conjunction with CCITT recommendations of 1984 on message handling systems. It is a notation for describing data structures, similar to data type definitions available in programming language such as Pascal or ADA. It is applied to the description of OSI application layer protocols, where it is used for the definition of the protocol data units (PDUs). Protocol data units are messages exchanged between different protocol entities. The same notation can also be used to define service primitives and PDUs of other layers. The notation includes a number of predefined data types, such as integers, reals, booleans, bit strings, octet strings and various kinds of character strings. It also allows the definition of composed data types, such as a group of elements (called SEQUENCE, corresponding to "record" in Pascal), a list of identical types (called SEQUENCE OF), a type of alternatives (called CHOICE, corresponding to Pascal's variant records), a tag defining a code to distinguish between different alternatives, and others.

The TTCN is relatively recent, and has been developed for the description of test cases for ISO conformance test suites [55]. The language includes several different notations. The overall organization of the language is in terms of a collection of tables defining different aspects of a test case, such as service primitives, PDUs and their parameters, order of interactions and constraints on parameter values. The interaction ordering is

defined in terms of a conceptual tree, where each branch represents a possible execution order. In addition to the tabular notation, a linear form of TTCN is being developed for the exchange of test cases in machine–readable form. The ASN.1 notation can also be used for certain aspects of test descriptions.

1.2 CONFORMANCE TESTING

OSI protocols are presently being implemented by a large number of computer manufacturers and communication companies. In order to achieve global interworking among heterogeneous systems, the most practical means is testing. There exist three types of testing: performance, interoperability and conformance testing. The aim of conformance testing is to check the conformance of the implementations to the protocol standard. Protocol implementations can be tested by considering a single-layer, multi-layer, or embedded entity as a whole and stimulating the entity from the layers above and below and observing the reactions of the Implementation Under Test (IUT). Stimulation/observation is done by sending/receiving (N)-service primitives, also called (N)-Abstract Service Primitives (ASPs) and (N–1)-ASPs (the latter including (N)-PDUs) by an entity called the tester. Service Access Points (SAP) used by the tester for this purpose are called Points of Control and Observation (PCO).

1.2.1 Test Method Overview

Various abstract methods [46] (Figure 1.4) are defined by ISO and CCITT based on the control and observation (i.e., availability of the ASPs) in an Implementation Under Test (IUT). The two main classes of test methods are *local* and *external*. In *local* test methods points of control and observation are defined at the lower and upper entities of the IUT. In other words, (N)-PDUs, (N)- and (N–1)-ASPs are assumed to be available such that upper and lower testers can control and observe them. External test methods, on the other hand, are characterized by the observation and control of PDUs taking place, on the other side of the underlying service provider from the IUT.

There are three types of external test methods: *distributed*, *coordinated*, and *remote*. Each comes in three variant forms: *single-layer*, *multi-layer*, and *embedded*. Single-layer methods are designed for testing a single-layer without reference to the layer above it. Multi-layer methods are designed for testing a multi-layer IUT as a whole. Embedded methods are designed for testing a single-layer within a multi-layer IUT, using knowledge of what protocols are implemented in the layers above the layer being tested. The external test methods vary according to their ability to define a *test management protocol* (TMP) to carry out the test coordination procedures or to express the test coordination procedures only in terms of requirements.

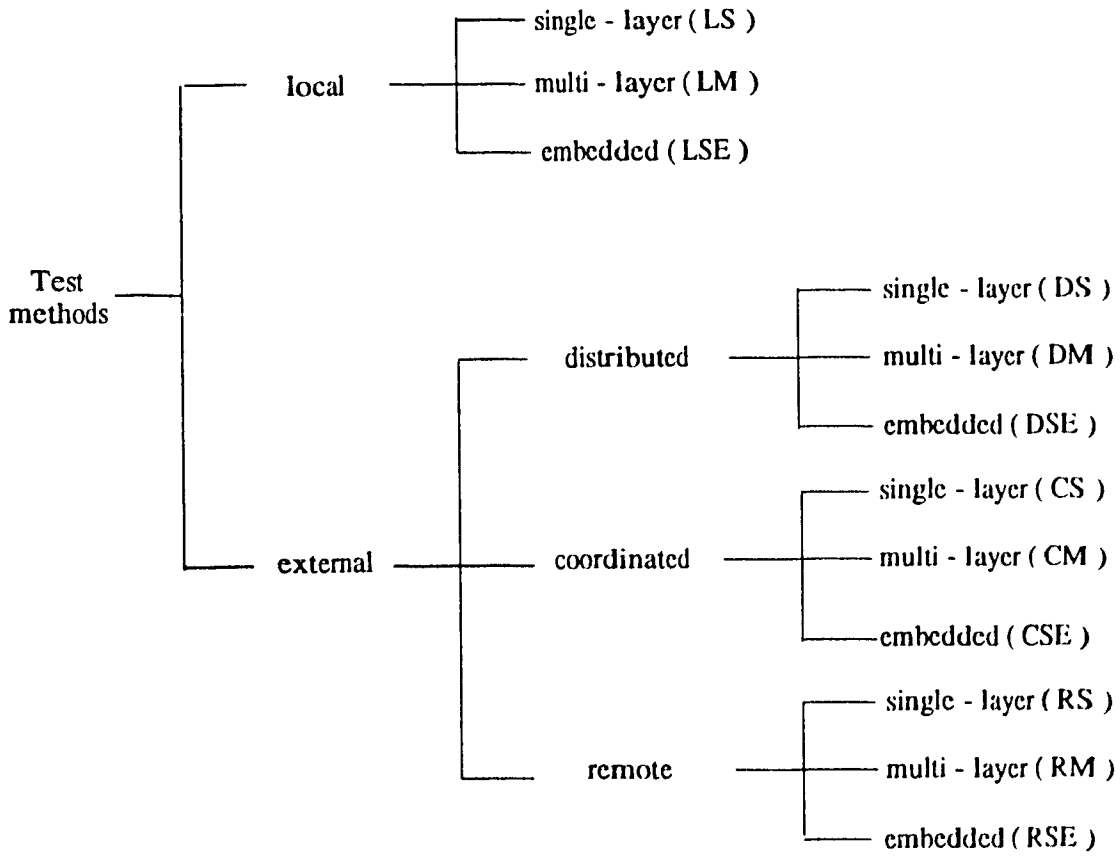


Figure 1.4 Abstract test methods.

1.2.2 Types of Testing

The objective of the conformance testing is to establish whether the implementation being tested conforms to the specifications in the relevant standard. Practical limitations make it impossible to be exhaustive, and economic consideration may further restrict testing. Therefore, four types of conformance testing are recommended by ISO, according to the extent to which they provide an indication of conformance.

i) Basic Interconnection Test: This provides a limited testing to check that the IUT can establish a basic interconnection before thorough testing is performed.

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

iii) Behaviour Tests: They test the *dynamic conformance requirements* of an IUT which are the requirements (and options) defining the observable behaviour of a protocol. A large part of behaviour tests, which constitute the major portion of conformance tests, can be generated from the formal specification.

iv) Conformance Resolution Tests: These tests are used to provide definite diagnostic answers to specific requirements, such as previously identified situations that may cause incorrect behaviour of an IUT. For example, they provide a yes/no type of answer for whether a particular feature, such as reset, is implemented in an IUT.

In addition to conformance testing, the following types can be performed on an IUT depending on the application: *interoperability tests*, to check whether two or more implementations that pass the conformance tests can operate together, *performance tests*, to measure the maximum throughput that can be obtained, and *robustness tests* to determine how well an IUT recovers from various error conditions.

1.2.3 Test Suite Structure

The *test events*, atomic interactions between the IUT and the upper or lower tester used in conformance testing, are described in an *abstract conformance test suite*. The

tests are represented in a hierarchical structure. The key level is called the *test case* which has a narrowly defined purpose. Test groups consist of several test cases according to a logical ordering of execution. A test case is divided into test steps each of which consists of several test events. Once the test cases are designed, they can be written in a test notation such as TTCN.

1.3 TEST SUITE DESIGN

Designing test cases (suites) can be considered to be the most active research area of protocol testing. Research in this area is inspired from the rich results obtained previously in hardware/software testing. Nevertheless, it is evolving towards its own set of techniques, tools and disciplines, possibly due to the distinct characteristics of protocols and their architectures. We will discuss the existing protocol test case design based on FSM, Estelle, LOTOS and SDL.

1.3.1 FSM Based Test Design

Protocols are typically modeled as an extended finite-state machine [3] where the control portion is the finite state machine and the data portion consists of the program segment. The control portion of a protocol (henceforth referred to as the protocol for simplicity) can be specified as a *deterministic finite state machine* (FSM) [35]. The *state* of a protocol is defined as a stable condition in which the protocol rests until a stimulus, called an *input* is applied. The protocol generates a response to the stimulus, called *output*, (which may be null) when an input is applied, and moves into a new state (which may be the same as the previous state) where it stays until the next input. This exercise is complicated by the limitations on the controllability and the observability of the protocol implementation. In most cases, because of the limited controllability, the implementation cannot be directly put into a desired state, usually requiring several additional state transitions. Unless efficient solutions are found, this limitation may result in test sequences with infeasibly large numbers of state transitions. Limited

observability prevent the external tester from directly observing the state of the protocol implementation, which is critical for a test to detect errors.

Typically, the formal conformance testing techniques generate a set of input sequences that will force the FSM implementation to undergo all specified transitions. These techniques can be classified [11] as the *transition tour method* [44], [51], [63], the *distinguishing sequence method* [43], [17–18], [21], [35], the *characterizing sequence method* [18], [21], [10], [35], and the unique *input/output sequence method* [48–49], [1]. All of these techniques assume the so-called *black box* approach where only the outputs generated by the implementation (upon receipt of inputs) are observable to the external tester.

The transition tour method generates a state tour that exercises every state transition of the implementation [44], [51] and does not address the observability problem described above. For certain protocols, which have a special message to determine the state of the protocol (i.e., the observability problem is solved by the specification), the length of the tour can be minimized by the technique given in [63], which is based on a graph theoretic concept called the *Chinese Postman Problem* [36].

The remaining three methods emphasize the observability problem. In the distinguishing sequence method, an input sequence is found for a protocol such that the outputs generated by the implementation will identify its state. The requirement for this method is a fully specified protocol, which may be too strong for most actual protocols. The characterizing method defines a *set* of input sequences for a subset of states such that the resulting set of output sequences ultimately distinguishes each state from the others. In both of these methods, the current state of the implementation is assumed to be unknown and the sequences generated by them are powerful enough to find the current state. In other words, both the distinguishing and characterizing sequences answer the question of "*what is the current state of the implementation*?". However, in the unique input/output sequence method, this question is relaxed to "*is the implementation currently in state x* ?", which results in much shorter sequences than the other two methods. A minimization

of the resulting sequences generated by the unique input/output method is given in [1], which is based on a more general form of the Chinese Postman problem called the Rural Chinese Postman problem. Experience with this method indicates that the test sequences generated are about one-third the size of those generated by ad hoc methods [1].

All of the above methods are based on the deterministic finite state machine. Systematic test design methods from the nondeterministic finite state machine are not yet well developed. This topic will be discussed in chapter V of the thesis.

1.3.2 Estelle Based Test Design

Since deterministic FSMs model only the control component of the protocol/services, there is a need to extend the FSM-based techniques to cover the data component, i.e., interaction primitive processing and data transfer mechanisms. Recently, two methods have been proposed to design test cases for testing the data flow aspects of an IUT. Both methods assume that protocol specifications are given in Estelle as a single module specification called a normal form specification. The first method [61-62] is based on data flow analysis techniques [45] and focuses on tracing the flow of data through the associations between assignments of values to variables and references of these variables in either assigning values to other variables or determining the outcome of conditional branching. The second method [52], [15], applies the principles of functional testing [25]. In the functional testing method two different graphs are obtained from the normalized specification: a control graph for major state changes and a data flow graph to show the flow of data from input service primitive/protocol data unit parameters to the context variables and from the context variables to output service primitive/protocol data unit parameters.

The Control graph is an FSM, thus the techniques described in section 1.3.1 are applicable. The data flow graph shows global flow of data over context variables. The data flow graph is partitioned into blocks, where each block corresponds to functions of the protocol. Test suite design with this methodology is based on obtaining the control

sequence for a test case and enumerating the parameters of the interaction primitives. Each block is tested with one or more test cases until all the arcs in the partitioned data flow graphs are covered. The resulting tests are used as *behaviour tests* for establishing the dynamic conformance of IUTs.

The above method is used in designing test suites from a more general model called an EFSM chart in this thesis. We use the zero-one integer programming technique for selecting the test cases, which will adequately exercise the protocol functions identified from the data flow graph. The zero-one integer programming technique is used in the path selection problem in software testing [65].

1.3.3 LOTOS Based Test Design

There has been much research on test suite generation from LOTOS specifications. The first related work can be found in [6] and [66], where the derivation of conformance testers T(S) for any specification S has been investigated. In [6] a failure model is used to identify processes that are testing equivalent, whereas in [66], a syntactical approach is explored, based on the work reported in [56]. The method is named the CO-OP method after its main components, the sets named COmpulsory and OPtional behaviours. At present the CO-OP method can produce canonical testers for basic LOTOS behaviour expressions that do not contain process abstraction. A conformance tester is not intended for practical testing, because of data, which is a significant aspect of protocol testing, not considered in the derivation of T(S) from the specification S, although some efforts to extend the approach have been reported [59].

In another related work [19], where manually an FSM model is derived by using LOTOS interpreter [40], classical methods are then applied to generate test cases. The authors have not yet considered the data flow aspect with the interpretation approach. Recently, a group of researchers at Neher laboratories [8] proposed an interactive methodology to generate test cases from a LOTOS specification. A symbolic evaluation

is used to derive test cases in the form of TTCN. However, the methodology is not algorithmic in nature. It is similar to the method proposed in [19].

1.3.4 SDL Based Test Design

Generation of test suite from SDL has been studied by Hogrefe[23]. In this model SDL processes are transformed into an intermediate form called the asynchronous communication tree (ACT). The ACT is a tree, where the root is the initial system state, the nodes are all other system states and the arcs represent the valid transitions. The tree is constructed by considering in turn every possible distinct event that may occur at each node. The arc is labeled with the event and the node is identified by the resulting process states and queue contents. The depth of the tree is restricted by comparing the current state with the states generated previously; if an identical state is found, that branch is terminated. Finally, this ACT is traversed to generate test cases. Nondeterminism has not been taken into consideration in the generation of test cases. The test generation algorithm proposed in this thesis, which deals with nondeterminism, can be used to generate a test suite from an ACT.

1.4 OBJECTIVE AND MOTIVATION

Among the problems of protocol engineering, conformance testing is the most important one. An efficient solution of this problem will demonstrate networks in which components made by different manufacturers can interwork properly and effectively, which in turn is necessary to fulfill the requirements of the OSI standard. Therefore, the main objective of this thesis is the development of algorithms for frequently encountered problems in test suite design.

In the last decade, a large number of algorithms have been proposed to design test suites from formal specifications. The emphasis in most of these algorithms is placed on fault detection capability and optimization of the generated test suites. However, the test architectures have not been taken into consideration in the design of test suites. Moreover,

the test generation procedure is restricted to the deterministic model. In this thesis, the emphasis is placed on the development of new algorithms, which are general in nature, through a more efficient formulation of the formal model. Thus, our intention is to give a unified formal model for generation and representation of test cases. The methodology has to be conceptually simple, easy to implement, computationally efficient, and general in nature.

1.5 ORIGINAL CONTRIBUTIONS

In the following, we outline a set of individual contributions which, when combined together, give rise to a test suite design methodology.

We introduce the unified model using the EFSM chart and the I/O diagram. The EFSM chart model is used to describe the behaviour of the protocol, and the I/O diagram model is used to describe the structure of ASPs/PDUs. Algorithms are presented to translate protocols specified in LOTOS or SDL into EFSM charts.

We present a new test generation algorithm, which takes nondeterminism into consideration. Furthermore, we propose an algorithm for the construction of the data flow graph that is used to identify the protocol functions.

We model a test case by a dependency graph and evaluate it by taking predicate slices. We propose two new algorithms to eliminate redundant assignments and predicates from the test case. The first one is based on the test case dependency graph and the second one is based on the data flow graph.

We propose algorithms to generate base and dynamic constraints. We obtain control flow behaviour algorithmically by inverting the direction of the actions and eliminating all the predicates from the test case dependency graph. Finally, we design test suites for RS and LS architectures.

We develop LOTEST, a computer aided software tool that partially implements our test suite design methodology for a LOTOS specification.

1.6 OUTLINE OF THE THESIS

This thesis presents a new methodology for the analysis and formulation of new algorithms for solving basic problems of conformance testing. The layout of the thesis is as follows. In Chapter II, the concepts of the EFSM chart and the I/O diagram are introduced. Chapter III is concerned with chart construction algorithms. New algorithms for generation and evaluation of test cases are proposed in chapter IV, while chapter V deals with test case representation. In particular, the main contents of each chapter are as follows:

Chapter II: Unified Model.

This chapter introduces ASN.1, SDL, LOTOS, the Extended Finite State Machine(EFSM) chart and the Input/Output diagram to be used throughout the thesis. The structure of ASPs/PDUs is briefly discussed. Also this chapter presents the relationship between I/O diagrams and abstract data types.

Chapter III: EFSM Chart of Specifications.

Algorithms to translate protocol specifications into Extended Finite State Machine Chart are developed in this chapter. The algorithm proposed by Karjoth [34] to translate a subset of LOTOS into EFSM chart is extended to full LOTOS and to SDL.

Chapter IV: Generation and Analysis of Test Cases.

Based on the EFSM chart, a new test suite generation algorithm is proposed, which takes nondeterminism into consideration. Furthermore, a new algorithm for the construction of data flow graph from the chart is developed, which is used to identify the protocol functions for testing the data flow aspect of an IUT. The zero-one integer programming technique is used to select test cases that are optimal in nature to meet certain test coverage requirements or to exercise certain protocol functions adequately. The generated test cases are modeled as a test case dependency graph and then evaluated by taking predicate slices from it. After a brief review of program slicing [64], we introduce the concept of predicate slices that are used to identify infeasible test cases that must be eliminated. Redundant

assignments and predicates in all the feasible test cases are removed by using the test case dependency graph as well as using the data flow graph.

Chapter V: Test Suite Selection and Representation.

Test cases generated from specifications define the behaviour of an IUT. The behaviour of LT and UT considered together comprise the tester's behaviour. The tester's behaviour is the dual of IUT's behaviour. Therefore the tester's behaviour can be obtained by behaviour inversion. Behaviour inversion is based on viewing each test case as an extended finite-state machine. Complete behaviour generation of the test case EFSMs is a complex process consisting of several steps. In the previous chapters we discussed the steps of specification transformation, test case generation and test case reduction. In this chapter we discuss how the control flow representation can be obtained followed by input/output data flow representation. The control flow representation refers to the steps of inverting the interaction whereas input/output data flow representation refers to the generation of base and dynamic constraints on the events, which are in the form of an I/O diagram. New algorithms are developed to generate base and dynamic constraints for receive and send events. Specification of tester's behaviour in a test case is completed by test selection. Assuming a test case hierarchy with test purposes as leaf nodes, the selection process associates a purpose to each test case and using this information, a verdict is associated with some of the events. Finally the test cases are adapted for the RS architecture.

Chapter VI: Applications.

We applied the test design methodology to two protocols: Association Control Service Element(ACSE) protocol written in LOTOS specification language and LAPB protocol written in SDL specification language.

Chapter VII: Conclusions and Future Work.

Basic contributions of the research described in this thesis are summarized. Also, possible extensions of the results to specific, or perhaps, new problems are discussed.

CHAPTER II

UNIFIED MODEL

In this chapter we give an overview of ASN.1, SDL, LOTOS and some other concepts. We provide an abstract syntax of algebraic data types in order to describe the data world on which the unified EFSM chart model is defined. Furthermore, we propose a unified model to describe the structure of ASP/PDUs: Input/Output(I/O) diagram, a graphical notation based on Jackson Structured Programming [33]. Representation of ASN.1 in terms of I/O diagrams is discussed first. The signature part of the abstract data type definition of ASP/PDUs can also be represented in the form of I/O diagram, which is discussed next. The I/O diagram will be used in test case representation.

2.1 ABSTRACT SYNTAX NOTATION 1

The key to the whole problem of representing, encoding, transmitting and decoding data structures is to have a way of describing the data structures that is flexible enough to be useful in a wide variety of applications, yet standard enough that everyone can agree on what it means. As part of the OSI development work, ISO has devised just such a notation. It is called **abstract syntax notation 1** or ASN.1 for short.

The ASN.1 primitive types are listed in Table 2.1. These types are built into the language and form the building blocks for more complex types. The names of these types are reserved words, and, like all ASN.1 reserved words, are always written in upper case letters.

Primitive type	Meaning
INTEGER	Arbitrary length integer
BOOLEAN	TRUE or FALSE
BIT STRING	List of 0 or more bits
OCTET STRING	List of 0 or more bits
ANY	Union of all types
NULL	No type at all
OBJECT IDENTIFIER	Object name (e.g., a library)

Table 2.1 The ASN.1 primitive types.

The primitive types can be combined to build more complex types. Table 2.2 shows the five principal constructors used in ASN.1 for this purpose.

Constructor	Meaning
SEQUENCE	Ordered list of various types
SEQUENCE OF	Ordered list of a single type, like an array
SET	Unordered collection of various types
SET OF	Unordered collection of a single type
CHOICE	Any one type taken from a given list

Table 2.2 The principal ASN.1 constructors.

In addition to the primitive types and the constructed types, the ASN 1 standard also discusses some predefined types that are useful in many applications. Eight different string types are defined. Each one is a different subset of OCTET STRING. The *NumericString* only includes the ten digits 0 through 9 and the space. The *PrintableString* includes the

upper and lower case letters, the ten digits, space, and the 11 characters in quotes: " () ' + - . , / := ? ". Other string types are provided for the teletex character set, the videotex character set, various international version of ASCII, and some graphics character sets.

Another useful type is *GeneralizedTime*. Using this format avoids endless discussion about whether 5/12 is the 5th of December or the 12th of May. An example value of *GeneralizedTime* is 19910704210538.3, which represents 5 minutes, 38.3 seconds after 9 P.M. on the Fourth of July, 1991.

In practice, it is common to define complex data types many of whose fields are optional. For example, *CONNECT REQUEST* PDUs frequently have a large number of optional parameters. If these parameters are not used for a particular connection establishment, they need not be transmitted. To handle this situation, ASN.1 allows fields to be declared OPTIONAL. Alternatively, they can be declared DEFAULT, followed by the value to be used by the receiver if the field is not transmitted. The existence of OPTIONAL and DEFAULT types potentially causes problems with identifying the data when they are received. Suppose that a SEQUENCE has ten fields, all of them are of type INTEGER and all OPTIONAL. Now suppose that only three of them are transmitted. How does the receiver know which three they are ?

This problem is solved by the concept of **tagging**. Four types of tags are allowed: UNIVERSAL, APPLICATION, PRIVATE, and CONTEXT SPECIFIC. Each tag consists of an integer, preceded by one of the reserved words UNIVERSAL, APPLICATION, PRIVATE, or no reserved words, in which the tag is CONTEXT SPECIFIC. Tags are written in square brackets.

2.2 SDL SPECIFICATION LANGUAGE

SDL, the specification and description language adopted by CCITT, is a standard for specifications of telecommunication systems. The language is built around the following concepts:

i. **structure**, which is described hierarchally by elements called systems, blocks, channels, processes, signal-routes and signals;

ii. **behaviour**, which is described using an extension of the finite state machine concept;

iii. **data**, which is defined as abstract data types with the type algebra ACT ONE [13];

iv. **communication**, which is asynchronous via channels and infinite queues.

A SDL specification (a *system*) consists of a number of blocks interconnected via *channels*. The *channels* are the media through which blocks communicate with each other or with the environment. A channel can be unidirectional or bidirectional. A bidirectional channel can be considered as two independent unidirectional channels. The channels are typed, i.e., they can only contain messages of certain types. In SDL the messages are called *signals*.

A block can contain either a substructure of blocks, or one or several processes interconnected by *signal-routes*. The signal-routes are the media through which processes can communicate with each other inside a block, and with the environment (i.e., everything outside the block). Signal-routes are, like the channels, typed unbounded FIFO queues.

In SDL, behaviour is defined by a set of processes that execute in parallel. To each process is associated exactly one input queue, into which all input signals from different input signal-routes are merged. A SDL process is described by an extended finite state machine, which consists of *states* and *transitions*. For each state there is a set of *input symbols*, each with an associated transition, and a set of *save symbols*. Signals can only be received from the input queue when the process is in a state, and the reception of a signal is the only event that can cause a transition to a new state to occur. The signals to be received are specified in the input symbols. To each input symbol, an *enabling condition* can be associated. The signal specified in the input symbol can only be received if the enabling condition evaluates to true. There is also a possibility to specify that some input signals should be bypassed when in a specific state. Such signals are specified in save symbols and are saved in the input queue as explained below.

All signals sent to an SDL process are buffered in the input queue in the order of

arrival, until the process is in a state in which it can accept input. Then there are four possibilities:

i. The signal at the front of the queue is specified as the beginning of a transition, in which case the signal is consumed by the process; it is removed from the queue and the process executes the associated transition.
ii. The signal at the front of the queue is specified in a save construct, in which case that signal remains in the queue and the signal behind the saved signal is examined.
iii. The signal at the front of the queue is neither specified as an input nor as a save, in which case the signal is removed from the queue and the next signal in the input queue is examined.
iv. The input queue is empty, in which case the process is suspended until a signal arrives to the input queue.

A transition performs a sequence of actions when the process changes state. The actions can be *outputs*, *tasks* and *decisions*. An output transmits a signal to another process. A task is a manipulation of internal data. A decision is a generalized case statement.

2.3 LOTOS SPECIFICATION LANGUAGE

LOTOS is a general-purpose language for specifying sequential, concurrent, or distributed systems based on CCS and ACT ONE concerning the concurrency and data type parts respectively.

LOTOS specifies a process by a *behaviour expression* that defines an ordering of *events*. LOTOS provides a set of temporal ordering *operators* that allow combining and structuring behaviour expression in different ways and so can be used to express important architectural concepts, such as orthogonality, sequential relationship, or non-deterministic operation. The operators *inaction*, *choice*, *action-prefix*, *parallel composition*, *hiding* and *guarding* act as a behaviour data type where the semantics of the corresponding LOTOS

behaviour expressions is given in term of action. Here, we explain briefly the process constructs. The formal definition can be found in [26].

internal action($i;B$): A behaviour can nondeterministically choose to be an unobservable internal action, denoted by i. It is called unobservable because it is not an interaction, yet it can cause a change in other behaviour.

action prefix(gd_1 .. $d_n[c]$, where d_i = ! t_i or ? v_i : s_i): The components of an action are either $!t_i$, denoting the offering of a certain value, or $?v_i : s_i$, offering the range of all values belonging to type s_i. The selected predicate c restricts the set of event values to those satisfying the predicate.

successful termination($exit(t_1, .. , t_n)$): The *exit* construct denotes successful termination of a behaviour. The values contained in it are passed on to the subsequent process. The value can also be *any s*. Exiting event is composed of an internal action followed by inaction *stop*.

inaction(*stop*): The process *stop* can not perform any action, and therefore does not have a successor behaviour. It is impossible to derive any transition from it.

local definition(let v_1: $s_1 = t_1$, .. , v_n: $s_n = t_n$ in B): An identifier can be given the value of an expression.

summation(choice v_1: s_1 , .. , v_1: s_1 [] B): This construct denotes the, possibly infinite, choice composition of copies of the behaviour B, where the variable v receives every possible value of the type s.

sequential composition(B1 >> accept v_1: s_1 , .. , v_1: s_1 in B2) : In this construct the right hand behaviour is enabled by the successful termination of the left hand behaviour. Besides this, the resulting values of the left hand behaviour are assigned to the identifiers v_1 , .. , v_n.

disabling(B1 [> B2): B1 [> B2 behaves like B1, until an event of B2 is performed, after which only events of B2 can occur. Once B1 has terminated successfully, events in B2 are no longer possible.

parallel composition(B1|[g_1 , .. , g_n]|B2): The expression B1|[g_1 , .. , g_n]|B2 allows

both behaviours to proceed independently, except for events on any of the gates g_1, .. , g_n. On those gates both processes must synchronize on the same event, which is then one of the events offered by the construct.

choice(B1 [] B2): A choice of behaviours offers all the events that any of the behaviour offer. In synchronization with the environment a choice is made, after which the corresponding behaviour results.

guarding([t] → B): A behaviour prefixed by a guard is only possible if the guard evaluates to true, and is equivalent to *stop* otherwise.

hiding(hide g_1 , .. , g_n in B) : Parallel composition is the mechanism to add processes to a synchronization on an event. Hiding is the mechanism that converts those events into internal events, so that other processes can no longer influence the events.

process instantiation(p[g_1 , .. , g_n] (t_1 , .. , t_n)): The behaviour of a process instantiation is the behaviour of its corresponding definition with the value expression substituted for the value parameter, and the formal gates replaced by actual gates.

The use of abstract data types (ADTs) complements the process part of LOTOS in achieving implementation independence. It is based on the equational specification of ADTs with an initial algebra semantics. An ADT definition identifies a mathematical object, namely an algebra, formed by sets of data values, called data carriers, and a set of associated operators. The name of data carriers are referred to as sorts. The declaration of every operation includes its domain, which consists of a list of zero or more sorts and range, which consists of exactly one sort. The sorts and operations of a data type are referred to as the signature of that data type. The most basic form of data type specification in LOTOS consists of a signature and possibly a list of equations.

2.4 I/O DIAGRAM

According to the OSI Reference Model, input/output messages of protocol entities are called Abstract Service Primitives (ASPs). Some of the ASPs contain as substructures the Protocol Data Units (PDUs) that are exchanged between two communicating peer

entities. ASPs and PDUs are in general complex data structures containing several substructures. This is specifically true for layer 7, i.e., the application layer. Having recognized this fact, standardization organizations have defined a data definition language called Abstract Syntax Notation One (ASN.1) to precisely specify PDUs [29]. On the other hand specification languages use different techniques to define data, e.g., abstract data types for LOTOS and SDL and Pascal types for Estelle.

The most widely used technique for specifying the important properties of data is the Jackson design method [33]. The Jackson technique is useful to model two fundamental relationships between the elements of data: *composition* and *alternation*. Composition occurs when the data in one class is formed by composing data items from a number of different less abstract classes of data items. Alternation occurs when the data in one class consists of data from a number of possible other classes that contain less abstract data entities. In this section, we extend this idea to model ASP/PDU structure, which we call I/O diagram.

2.4.1 I/O Diagram for ASN.1

An I/O diagram is a tree as shown in Figure 2.1, whose root node is labeled with the name of the ASP/PDU and whose interior nodes are labeled with the name of composite types. The successors of a node are defined in terms of sequence, unordered sequence, alternation and iteration primitives as shown in Figure 2.2.

Two types of leaf nodes are defined, *primitive type* and *nonprimitive type*. We distinguish three kinds of primitive type leaf nodes: *optional_leaf_node*, *nonoptional_leaf_node* and *default_leaf_node*. A primitive type may have a value/range, which appears below the bar dividing the node. There is only one type of nonprimitive type leaf node: *extension_leaf_node*. The first field of the *extension_leaf_node* is labeled with a component type name and the second field is empty. Different kinds of leaf nodes are shown in Figure 2.3. Next, we describe representation of each composite type of ASN.1 using I/O diagram.

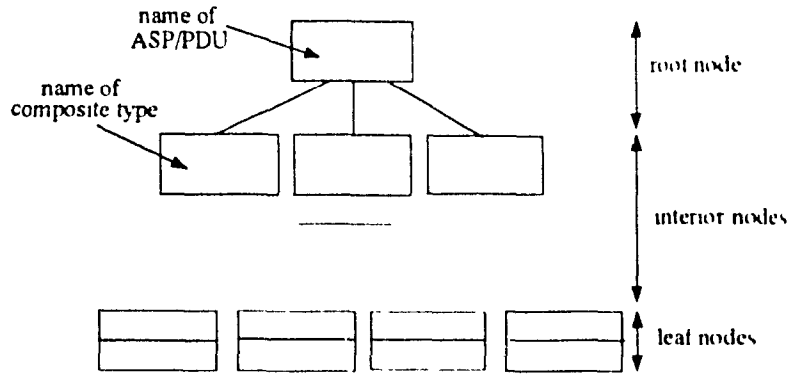


Figure 2.1 The components of an I/O diagram.

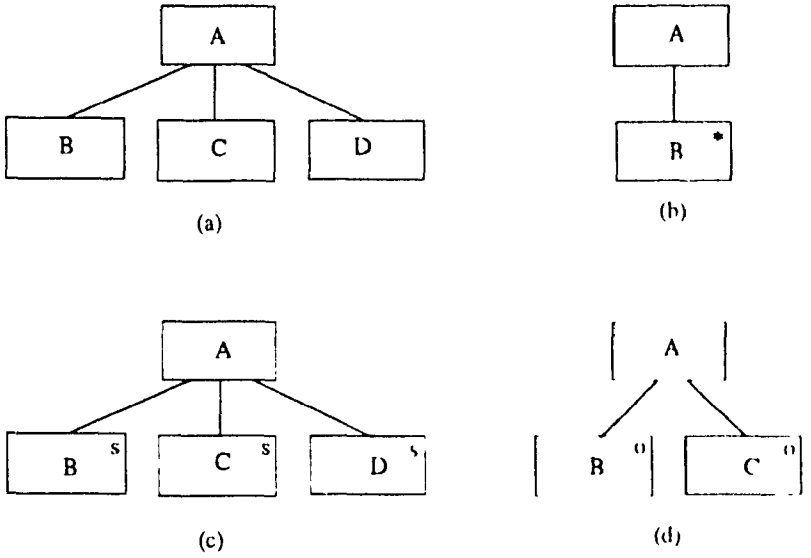


Figure 2.2 (a) Sequence: A consists of B, followed by C, followed by D. (b) Iteration: A consists of zero or more repetitions of B. (c) Unordered sequence: A consists of B, C and D in any order. (d) Alternation: A consists of B or C not both.

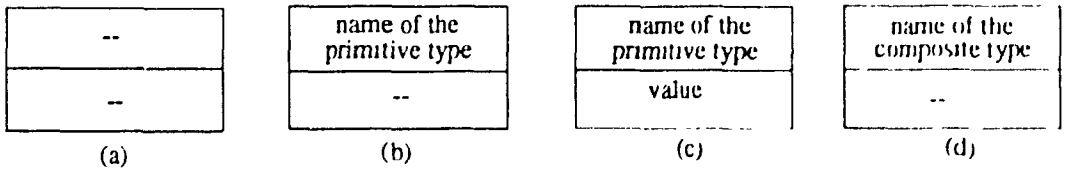


Figure 2.3 (a) Optional_leaf_node. (b) Nonoptional_leaf_node. (c) Default_leaf_node. (d) Extension_leaf_node.

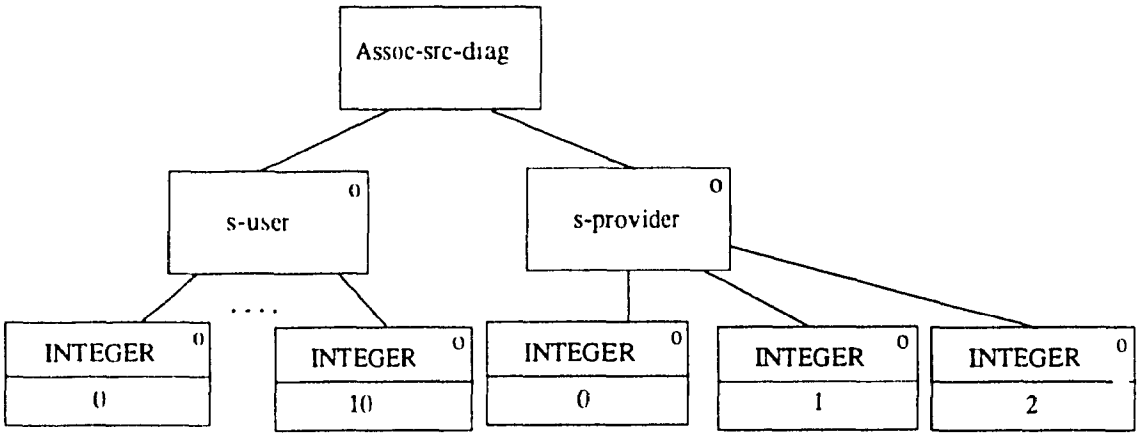
CHOICE: A CHOICE is represented using the alternation primitive. For example, the ASN.1 definition

Assoc-src-diag ::= CHOICE

{ s-user INTEGER { null (0), ... , called-AE-invalid-not-recognized(10) },

s-provider INTEGER { null (0), no-reason-given (1), no-common-acse-version (2)}

is represented as



SEQUENCE/OPTIONAL/DEFAULT: An ordered SEQUENCE is represented using the sequence primitive. Optional fields are incorporated by the addition of a null leaf node. Default values are represented by an additional optional leaf node carrying the default value.

As an example, let us consider the following ASN.1 definition

AARE-apdu ::= SEQUENCE { protocol-version BIT STRING

{ version1(0) }

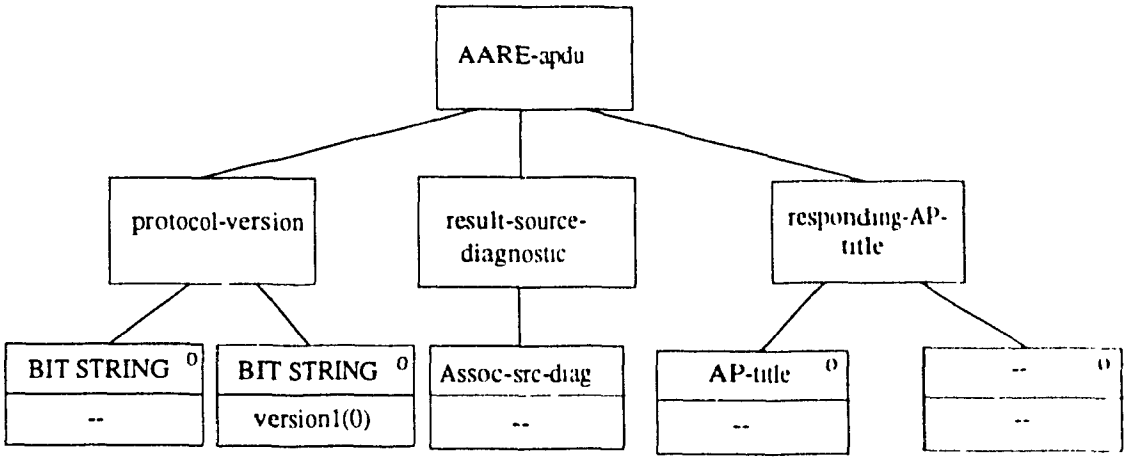
DEFAULT version1

result–source–diagnostic Assoc–src–diag

responding–AP–title AP–title OPTIONAL

}

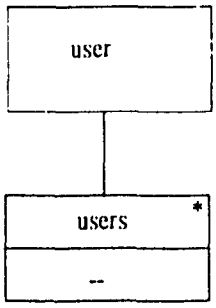
The I/O diagram of "AARE-apdu" is:



SEQUENCE OF: This ASN.1 construct is directly represented in terms of the iterative primitive. For example, the ASN.1 definition

user ::= SEQUENCE OF users

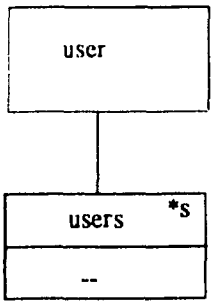
is represented as:



SET OF: This ASN.1 construct is represented by a combination of the unordered sequence and iteration primitives. For example, the ASN.1 definition

user ::= SET OF users

is represented as:



2.4.2 I/O Diagram for ADT

The I/O diagram describes the syntax of the ADT definition. Thus we need to be concerned with the operations part. We assume that in the operations there is an operation, that constructs the data type from its substructures. This operation is used to construct the I/O diagram. I/O diagrams for substructures are similarly constructed from the type definitions defining the substructures. Any constant values defined are mapped to primitive leaf nodes with values assigned. All the fields are assumed compulsory. For optional fields it is assumed that EFSM-chart contains send events in which omitted fields are assigned to *Not_present(field)*.

To illustrate the correspondence between an I/O diagram and the structure of an ADT, we consider the following ADT definition

type IPDUType **is** ISDUType, Sequencenumber

sorts IPDU

opns

DT : Sequencenumber, ISDU → IPDU

AK : Sequencenumber → IPDU

data : IPDU → ISDU

num : IPDU → Sequencenumber

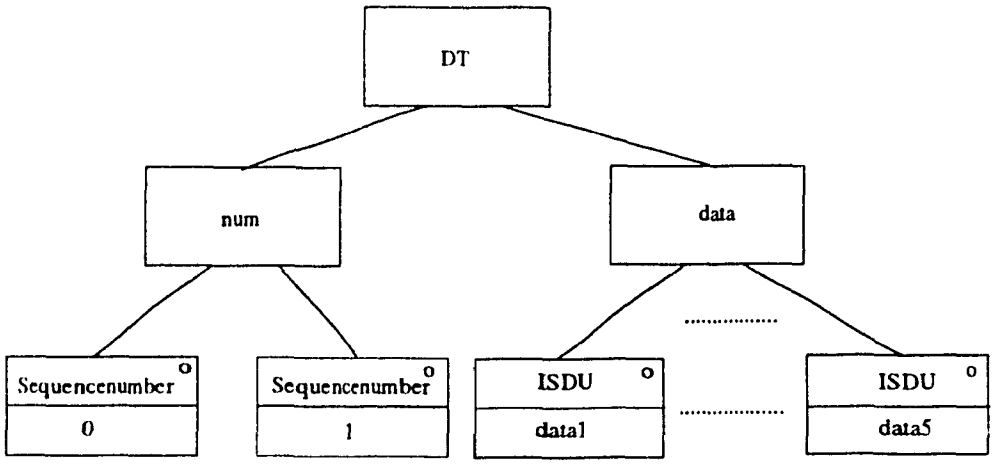
eqn

for all f: Sequencenumber, d: ISDU, ipdu: IPDU

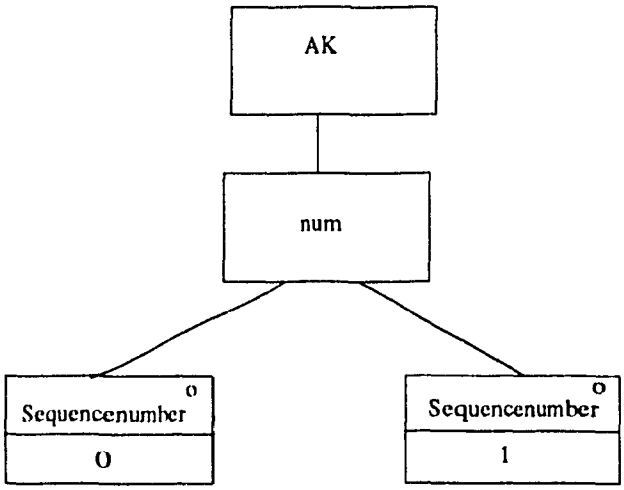
ofsort ISDU

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

type Sequencenumber is Boolean
    sorts Sequencenumber
    opns
        0 : → Sequencenumber
        1 : → Sequencenumber
        succ : Sequencenumber → Sequencenumber
    eqns
    ofsort Sequencenumber
        succ(0) = 1;
        succ(1) = 0
endtype

type ISDUType is
    sorts ISDU
    opns
        data1, data2, data3, data4, data5: → ISDU
endtype
```

The I/O diagram corresponding to the above abstract data type of DT pdu is shown below:



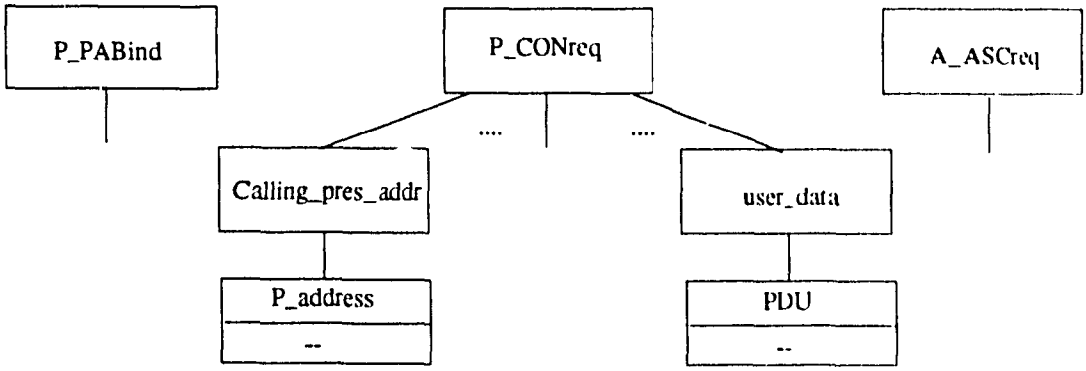
Similarly, the I/O diagram for the AK pdu is as follows:



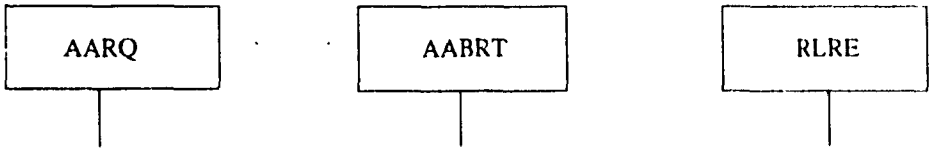
2.4.3 ASP/PDU Hierarchy from Specification

All the ASPs and PDUs defined in a specification can be mapped to I/O diagram hierarchies. There are three sets of I/O diagrams: one for ASPs, one for PDUs and another for all the substructures that ASPs/PDUs need.

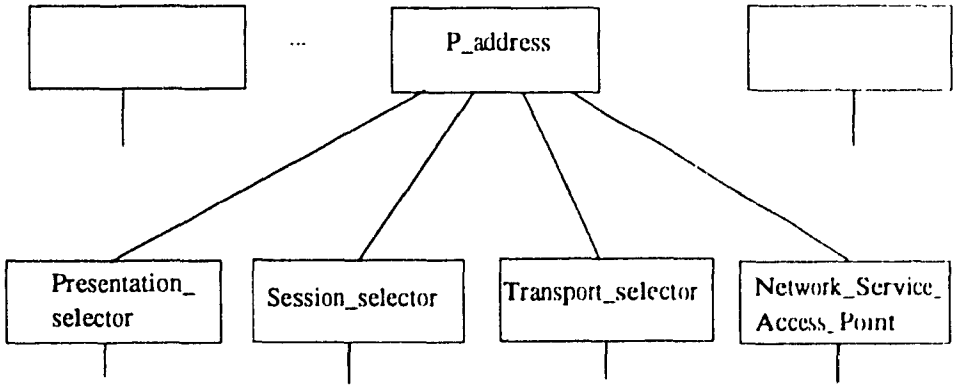
For example the set of ASP diagram:



a set of PDU diagram:



and finally another set for all the substructures:



2.5 TRANSITION SYSTEM AND EFSM CHART

Our starting point is a well-accepted formal notion known as transition system to represent operational semantics of concurrent systems.

Definition 2.1 A transition system is a quadruple $T =< Q, \Xi, \rightarrow, init >$, where

- Q is a set, the states of T,
- Ξ is a set, the events of T,
- $\rightarrow \subseteq Q \times \Xi \times Q$ is a relation, the transitions of T,
- $init \in Q$ is the initial state of T.

The chart introduced in [42] is a particular kind of transition system. In a chart, a state is labeled by zero or more identifiers; an identifier indicates states at which the behaviour of a transition system may be extended by substitution of another transition system for that identifier. Before giving the formal definition of the chart, we provide an abstract syntax of algebraic data types in order to describe the data part on which the chart system is to be defined.

Definition 2.2 Let S be a set whose elements we call sorts which are names for various data domains. A **signature** over S is an $S^* \times S$ -indexed family of sets $\Omega =< \Omega_{w,s} >_{w \in S^*, s \in S}$; the elements of a set $\Omega_{w,s}$ in the family Ω are called operators. A family of **variables** over S is an *S-indexed* family of sets $V = < V_s >_{s \in S}$; the elements of a set V_s, in the family V are called variables.

Given a signature Ω over a set of sort S and a family of variables V over S, term over Ω and V are constructed in the usual way and they also form an *S-index* family of sets $T_\Omega(V)$; T_Ω denote the subfamily of constant **terms** - i.e., terms not containing variables within $T_\Omega(V)$; Let $t_1, t_2, \ldots, t_n$ be metavariables over terms; the expression $E\{ t_1, t_2, \ldots, t_n / v_1, v_2, \ldots, v_n \}$ denote the result of a substitution, where each variable v_i of the expression E is replaced by the term t_i. The variable $v \in V$ is free in E, and the sort of the term is the same as the sort of the substituted variable. Let $D = D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$ be the joint domain of the variables, with $D_{v_i} \subseteq T_\Omega$ for each i.

Let G be the set of gates and channels; elements of G are denoted by g; let $\mathbf{G}_{sig}$ be the set of signals. S is a list of gate names. The distinguished symbols i, i_s, i_r shall not be member of G. The set of actions is the set

$$A = \{i, i_s, i_r, gd_1..d_n | g \in \mathbf{G}, d_i = !t_i \text{ or } !sig \text{ or } ?sig \text{ or } ?v_i : s_i,$$
$$t_i \in T_\Omega(V), sig \in \mathbf{G}_{sig}, v_i \in V, s_i \in S\}$$

For each action a the function *name(a)* is defined as follows:

$$name(a) = \begin{cases} g & if\ a = gd_1..d_n \\ i_s & if\ a = i_s \\ i_r & if\ a = i_r \\ i & \text{otherwise} \end{cases}$$

The direction of data flow associated with an action is defined by

$$offer_i(a) = \begin{cases} ! & if\ d_i = !t_i \text{ or } !Sig \\ ? & if\ d_i = ?v_i : s_i \text{ or } ?Sig \\ \text{undefined} & \text{otherwise} \end{cases}$$

Let I be the set of process identifiers. Each process identifier $X \in I$ has a fixed arity $n(X), n(X) \geq 0$, the number of its arguments. The total function $f_p : I \rightarrow V^*$ yields for each identifier X its formal parameter list. Fixed point operator μ is used for the process declaration $\mu X(v_1, .., v_n).B(t_1, .., t_n)$ as an alternative notation for process declaration used in LOTOS. This allows us to deal with a specification as being one expression. We are now in a position to define the EFSM chart formally.

Definition 2.3 An **EFSM chart** is a 8-tuple $m = < J, N, E, V, R, j_o, Z, h_o >$, where

- J is a finite set, the control states of m;
- N is a finite set, the transitions of m;
- $E \subseteq J \times I$ is a finite set, the extension of m;
- V is a finite set, the variables over S of m;
- R is a finite set, the rules of m (see below);
- $j_o \in J$ is the initial control state of m;

- $Z \subseteq J$ is a finite set, the terminal control states of m;
- $h_o \in \{v \leftarrow t | t \in T_{\Omega_D}\}$ 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 2.4 A rule of a chart is a 8-tuple $r =< a, j, j', n, p, c, f, h >$, where

- $a \in A$ 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;
- $p \in T_{\Omega_{Bool}}(V)$ is a predicate, the guard clause of r;
- $c \in T_{\Omega_{Bool}}(V)$ is a predicate, the condition clause of r;
- $f \in \{v \leftarrow t | t \in T_{\Omega_D}(V)\}$ is a function, the action clause of r;
- $h \in \{v \leftarrow t | t \in T_{\Omega_D}(V)\}$ is a function, the assignment clause of r.

The transition n occurs, when the chart is in 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 a, if the condition c is satisfied. This leads to the new control state j'.

We shall frequently write

$R(j) = \{< a, j', n, p, c, f, h | < a, j, j', n, p, c, f, h >\in R\}$ the rules of j

$E(j) = \{x | < j, x >\in E\}$ the extension of j

We shall use subscripts to identify rules, so that $when_r$ refers to the when clause of rule r.

CHAPTER III

EFSM CHART OF SPECIFICATIONS

In this chapter, we develop algorithms to transform a protocol specification into an Extended Finite State Machine (EFSM) chart. An algorithm is proposed in [34] to translate a subset of LOTOS into an EFSM chart. We extend the algorithm to full LOTOS and SDL for our requirements. In section 3.1 we present the LOTOS to chart generation algorithm. In section 3.2, we present the SDL to chart generation algorithm. Finally, the size of the generated chart with respect to states and rules is discussed.

3.1 FROM LOTOS TO EFSM CHART

LOTOS specifications can be transformed into EFSM charts in two phases. In the first phase the specification is transformed into a semantically equivalent form. In the second phase it is converted into an EFSM chart by bottom-up synthesis. The translation algorithm is confined to the dynamic behaviour of the LOTOS specification, because no adequate theory exists for the automatic translation of algebraically specified abstract data types into statements of an imperative language.

3.1.1 Transformation of a LOTOS Specification

In order to facilitate the chart construction algorithm, it is convenient first to apply some transformations to the LOTOS specification. The transformation rules are explained below:

- *Transformation of full synchronize composition to general composition*

A full synchronize operator between B_1 and B_2 ($B_1 \| B_2$) can be transformed to a generalized parallel operator by inserting all synchronized gates inside the parallel operator at which B_1 and B_2 synchronize (B_1 |[*synchronized gates*]| B_2). This transformation will reduce the number of rules in the chart construction algorithm.

- *Transformation of sequential composition to general composition*

In LOTOS internal actions occur in execution sequences either explicitly (an *i* in the specification) or because they result from the dynamic behaviour of the system (an enable operator in the specification). Here, we deal with internal events, those due to an enable operator.

Let us first explain how nondeterminism appears, due to the enable operator [20]. For instance, consider the following process:

```
process     P[a,b]:=
            ( exit
             [] a; exit ) >> b; stop
endproc
```

This behaviour is equivalent to

```
process     P[a,b]:=
            ( i ;b ; stop
             [] a ; i ; b ; stop ) endproc
```

Unfortunately, this transformation does not work if there is value passing in the enabling. For example

```
process     Q[a,b,c]:=
           ( exit(true)
            [] a ; exit(false)
           ) >> accept ok:bool in
                [ok] → b ; stop
              [] [not(ok)] → c ; stop
endproc
```

This type of situation can be handled by first introducing an internal action just before the *exit* construct of the enabling process, then transforming the sequential composition to parallel composition by introducing an auxiliary gate at which the enabling process synchronizes its last action with an action implicitly prefixed to the enabled process.

Also, we should regard this synchronization as private to the enabling and the enabled process. As an example the process P[a,b] above can be transformed to:

```
process     P[a,b]:=
              hide δ1 in
               ( i ; δ1 ; stop ;
                [] a ; i ;δ1 ; stop ;)
                  |[δ1]|
                δ1 ; b ; stop
endproc
```

Sequential composition with value passing can be similarly replaced by parallel composition. To achieve this, we have to do two things. First, replace the list of sorts of value offered at successful termination of the enabling process by value declarations attributed with the auxiliary gate. Second, replace the *accept* construct of the enabled process by variable declaration attributed with the auxiliary gate. For example, consider the previously defined process Q[a,b,c] which can be transformed to:

```
process     Q[a,b,c]:=
              hide δ1 in
               ( i ; δ1! true ;stop ;
                [] a ; i ;δ1! false ; stop ;)
                 |[δ1]|
                 δ1 ? ok: bool ;
                   [ok] → b ; stop
                   [] [not(ok)] → c ; stop
endproc
```

In those cases where the enabling process itself is a composition of parallel processes, the last action of these processes synchronizes within themselves at the auxiliary gate, which in turn synchronizes with the first action of the enabled process. If the parallel composition is a pure interleaving, then it has to be transformed to generalized parallel

composition with the auxiliary gate as a synchronization gate.

- *Conversion of generalized choice to choice expression*

The structure of the generalized choice is: **choice** g in [a_1, a_2, . . , a_n] [] **X**[g], where gate identifiers are used for indexing. It can be mapped into the construct **X**[a_1] [] , . . , [] **X**[a_n]. In this case, choice of n instances of **X** is created; for each one of them a formal gate g is actualized with a different element of the gatelist a_1, a_2, . . , a_n.

- *Process instantiation*

To identify recursion in an EFSM chart, it is necessary to expand the process before the construction of the EFSM chart. It can be achieved, by repeated process instantiation, until actual gate parameters are found to be identical to formal ones. This transformation ensures that all gates are constants, i.e., it is free from gate relabeling.

- *Removal of guarded internal events*

If an internal event is not the first action in a choice expression, it may be removed. For example, the expression *a ; A* [] *i ; b ; B* cannot be simplified to *a ; A* [] *b ; B*. However, the expression *a ; i ; B* may be reduced to *a ; B*.

- *Renaming of variables*

Variables in LOTOS processes have local significance. A variable can be used in more than one process for different purposes. To avoid global conflicts, variables are renamed uniquely.

3.1.2 The EFSM Chart Construction Algorithm

In this section we shall present the algorithm to translate the transformed LOTOS specification into an EFSM chart. Roughly speaking, for any transformed LOTOS behaviour expression, we can construct an EFSM chart. But for several reasons we have to impose restrictions on the form of the behaviour expressions.

Definition 3.1 A guard is an event a such that $name(a) = g$ A guard is said to be an exit guard if it precedes an exit. A guard is said to be free guard if it precedes a free identifier X.

Example 3.1 X is a free identifier in the left operand of the parallel composition in a ; (X | [S] | μY. (b ; Y)), where μ is a fixed point operator used for process declaration. The event a is an exit guard in a; *exit* but not in $a;b;exit$. Similarly the event a is a free guard in $a;X$ but not in $a;b;X$. However, the event b is a free guard in $a;b;X$ as well as exit guard in $a;b;exit$.

Definition 3.2 A free occurrence of X in B is guarded in B if it occurs within some subexpression $a;B'$ of B, otherwise unguarded in B.

Example 3.2 X is guarded in $a;X$ but neither in X nor in $a;X||X$.

Definition 3.3 Operands of the general parallel operator are said to be synchronous if the free guard and exit guard synchronize.

Example 3.3 In X:= $a;b;exit$ |[b]| $b;X$, the operands are synchronous since they synchronize at b, which is a free guard in $b;X$ and exit guard in $a;b;exit$.

In the chart construction algorithm, we will only treat behaviour expressions satisfying the following requirements:

1. if $\mu X.B$ is a subexpression of the process, then X is guarded in B (μ is a fixed point operator used for process declaration)
2. Operands of the general parallel operator are either closed or synchronous.
3. Operands of the pure interleaving operator are closed.

Algorithm: Chart Construction Algorithm.

In the following, we assume that for B' there is a chart $m' = < J', N', E', V', R', j_0, Z', h'_o >$. Similarly for B''. Let j_o be a control state not in J' or J''. If $f : D_1 \rightarrow D_2$ and $g : D_2 \rightarrow D_3$ then their composition $gof = g(f(x))$. Let $\epsilon : V \rightarrow D$ be an arbitrary but fixed function. The chart construction algorithm translates a LOTOS behaviour expression into a corresponding chart by bottom-up synthesis. The chart corresponding to

a behaviour B' is recursively built from the sub-chart corresponding to the sub-behaviour contained in B'. In the following, we explain how the LOTOS constructs are translated into the EFSM chart.

Base Cases:

- *stop* and *exit*

Choose $m =< \{j_o, \phi, \phi, \phi, \phi, j_o, \{j_o\}, \epsilon\}$

The chart corresponding to *stop* and *exit* has only one state, j_o and no transition. We treat an *exit* process the same as a *stop* process in the generation of the chart. It may need further explanation. For the *exit* process the associated axiom is: $exit - \delta \rightarrow stop$. The action δ (special action) plays a key role in the sequential composition of processes. However, we have already transformed the sequential composition to parallel composition. Hence its occurrence is just a successful termination after which it transforms into a dead process *stop*. Therefore, we treat an *exit* process same way as a *stop* process in the formation of the chart.

- $X(t_1...t_n)$

Choose $m =< \{j_1\}, \phi, < j_o, X >, f_p(X), R, j_1, \{j_1\}, \epsilon >$ with

$f_p(X) =< v_1 : s_1, v_2 : s_2, .., v_n : s_n >$ *and*

$R =< t_r, j_1, j_o, n_o, true, true, \epsilon, < v_1 \leftarrow t_1, v_2 \leftarrow t_2, .., v_n \leftarrow t_n >$

A new rule is created from state j_1 to state j_o. The process name is tagged with the state j_o.

Inductive Steps:

- $gd_1...d_n\,[c]\,;B'$

Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V, R, j_o, Z', \epsilon >$ where

$V = V' \cup \{v_i : s_i, |d_i = ?v_i : s_i\}$ *and*

$R = R' \cup \left\{< gd_1..d_n, j_o, j_o', n_o, true, c, \epsilon, h_o' \right\}$

A new chart is obtained by creating a new state j_o and generating a new rule from j_o to j_o', whose *when* and *condition* clauses are $gd_1...d_n$ and $[c]$ respectively. The assignment h_o' of the chart B' is stored in the assignment clause of the new rule.

- $B'[]B''$

Choose
$m =< \left(J' - \left\{j_o'\right\}\right) \cup \left(J'' - \left\{j_o''\right\}\right) \cup \{j_o\}, N' \cup N'', E, V' \cup V'',$
$R, j_o, Z' \cup Z'', h_o'' o h_o' >,$ *where*
$E = \left\{< j_o, X > | X \in \left(E'\left(j_o'\right) \cup E''\left(j_o''\right)\right)\right\} \cup \left(E' - \left\{E'\left(j_o'\right)\right\}\right)$
$\cup \left(E'' - \left\{E''\left(j_o''\right)\right\}\right)$

and
$R = \left\{< a, j_o, j, n, p, c, f, h > | < a, j, n, p, c, f, h > \in R'\left(j_o'\right) \cup R''\left(j_o''\right)\right\}$
$\cup \left(R' - R'\left(j_o'\right)\right) \cup \left(R'' - R''\left(j_o''\right)\right)$

A new state j_o is created and all the rules emanating from j_o' and j_o'' are modified to emanate from j_o and all the rules terminating in j_o' or j_o'' are made to terminate in j_o.

- $[t] \rightarrow B'$

Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V', R, j_o, Z', \epsilon >$ *where*
$R = R' \cup \left\{< i, j_o, j_o', n_o, t, true, \epsilon, h_o' >\right\}$
A new transition is created with an internal event and guard t.

- Let $v_1 : s_1 = t_1, ..., v_n : s_n = t_n$ *in* B'

Choose $m =< J', N', E', V, R', j_o, Z', h_o >$ *where*

$V = V' \cup \{v_1 : s_1, .., v_n : s_n\}$ *and*

$h_o =< v_1 \leftarrow t_1, .., v_n \leftarrow t_n > o\, h_o^{'}$

The chart m' is updated by addition of the assignment $v_1 \leftarrow t_1, .., v_n \leftarrow t_n$ in the initial assignment $h_o^{'}$ of the chart B'.

- Choice $v_1 : s_1, ..., v_n : s_n \,[]B'$

Choose $m =< J', N', E', V, R', j_o, Z', h_o >$ *where*

$V = V' \cup \{v_1 : s_1, .., v_n : s_n\}$ *and*

$h_o =< v_1 \leftarrow any_1 , .., v_n \leftarrow any_n > o\, h_o^{'}$

Here, the initial assignment is updated by the random assignments $v_1 \leftarrow any_1 , .., v_n \leftarrow any_n$, i.e., set $v_i \in V_{s_i}$ to an arbitrary value $any_i \in T_{\Omega_{S_i}}$, where $1 \leq i \leq n$.

- $i; B'$

Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V', R, j_o, Z', \epsilon >$ *where*

$R = R' \cup \left\{< i_s, j_o, j_o^{'}, n_o, true, true, \epsilon, h_o^{'} >\right\}$

We use i_s in the chart (silent transition) to distinguish the internal event in the specification from the internal event i due to hide and guard.

- $B' \,|[\mathbf{S}]|\, B''$

By restriction, we can get B' and B'', with $E' = E'' = \phi$ or $E' \neq \phi, E'' = \phi$ or $E' = \phi, E'' \neq \phi$. Let $\hat{v}$ be the fresh variables unique to all other variables used in the translation. Let $v_i^{'}$ stand for v_i if $d_i^{'} = ?v_i : s_i$ and accordingly let $t_i^{'}$ stand for t_i if $d_i^{'} = !t_i$. Let $\mathbf{Y} = \left\{q^{'} | < a, q^{'}, q, n, p, c, f, h >\in R' \wedge name\,(a) \in \mathbf{S}\right\}$

Choose

$m =< J' \times J'', (N' \times N'') \cup N' \times N'', E, V' \cup V'', R, < j_o^{'}, j_o^{''} >, Z, h_o^{'} o h_o^{''} >,$

where

$$E = \begin{cases} \left\{ < \left(j' \times J'' \right), X > \mid < j', X > \in E' \right\} & if\ E' \neq \phi \\ \left\{ < \left(J'' \times j'' \right), X > \mid < j'', X > \in E'' \right\} & if\ E'' \neq \phi \\ \phi & \text{otherwise} \end{cases}$$

$$R\left(< j', j'' > \right) =$$

$$\left\{ < a, < to', j'' >, n', p, c, f, h > \mid < a, to', n', p, c, f, h > \in R'\left(j'\right) \wedge name\left(a\right) \notin \mathbf{S} \right\}$$

$$\cup$$

$$\left\{ < a, < j', to'' >, n'', p, c, f, h > \mid < a, to'', n'', p, c, f, h > \in R''\left(j'\right) \wedge j' \in \mathbf{Y} \cup Z' \right.$$

$$\left. \wedge\ name\left(a\right) \notin \mathbf{S} \right\}$$

$$\cup$$

$$\left\{ r = r' \times r'' \mid r' \in R'\left(j'\right) \wedge r'' \in R''\left(j''\right) \wedge name\left(when_r\right) \in \mathbf{S} \right\}$$

$$Z = \left\{ J' \times j'' \mid j'' \in Z'' \right\}$$

$$r = < a, < to', to'' >, n, p, c, f, h > \quad and \quad name\left(a\right) = g$$

$$with\ \ a = \begin{cases} gd'_1, .., d'_i, .., d'_n & if\ offer_i\left(a'\right) =!; \\ gd'_1, .., d''_i, .., d'_n & if\ \ offer_i\left(a''\right) =!; \\ gd'_1, .., \zeta_i, .., d'_n & \text{otherwise} \end{cases}$$

$$where\ \ \zeta_i = ?\hat{v}_i : s_i$$

$$and\ \ V = \left\{ \hat{v}_i : s_i \mid \zeta_i = ?\hat{v}_i : s_i \right\}$$

$$and\ \ n = < n', r'' >$$

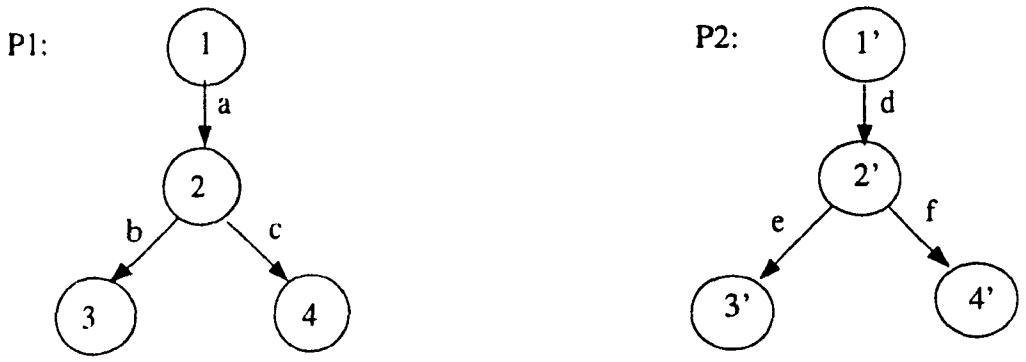
$$and\ \ p = \begin{cases} p' \wedge p'' \wedge t'_i = t''_i & if\ offer_i\left(a'\right) = offer_i\left(a''\right) =!; \\ p' \wedge p'' & \text{otherwise} \end{cases}$$

$$and\ \ c = c' \wedge c''$$

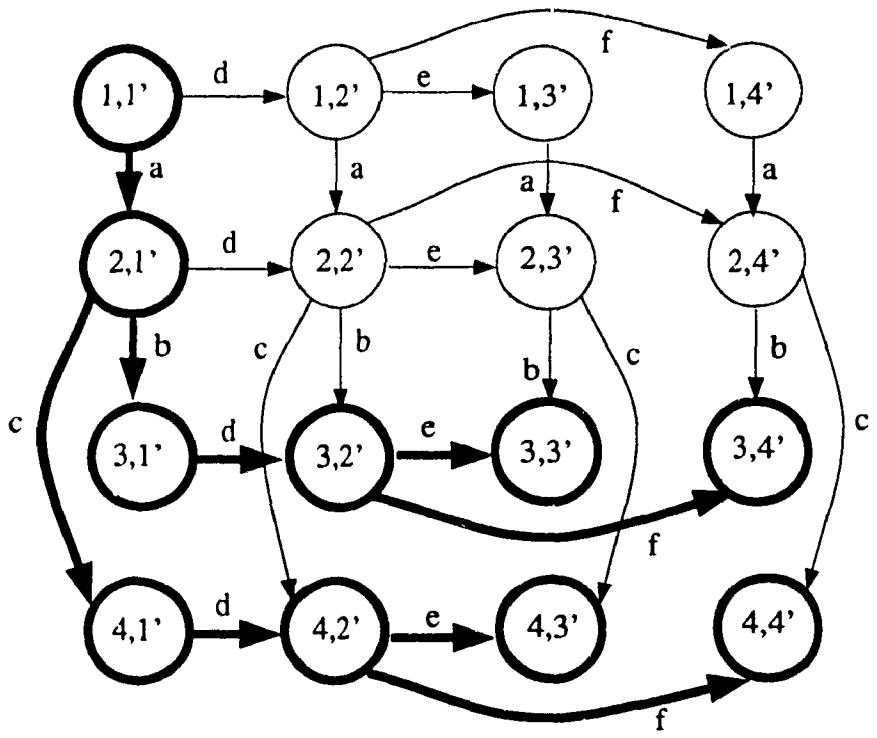
$$f = \begin{cases} f'of''o < v''_i \leftarrow t'_i & if\ \ offer_i\left(a'\right) \neq offer_i\left(a''\right) \wedge offer_i\left(a'\right) = ! \\ f'of''o < v'_i \leftarrow t''_i & if\ \ offer_i\left(a'\right) \neq offer_i\left(a''\right) \wedge offer_i\left(a''\right) = ! \\ f'of''o < v'_i \leftarrow \hat{v}_i,\ \ v''_i \leftarrow \hat{v}_i > & \text{otherwise} \end{cases}$$

$$and\ \ \ h = h'oh''$$

Here we deviate from [34] where a Cartesian product of the state machine of the sub-processes are taken. The above rule is composed in such a way that one will obtain a single sequence of events. As an illustration let us consider two state machines P1 and P2 shown below:



The product machine P1|||P2 obtained by the method proposed in [34] is as follows:



Our method will generate only the states and transitions that are bold. The rules are designed in such a way that priority is given to the state machine P1.

Two cases may need further explanation. When both the charts offer an output value(!), it can be decided only at run time whether the values will be equal. Therefore, this condition has to be made explicit in the guard clause. On the other side, when both the charts accept input, the actual value to be received is unknown. An auxiliary variable is introduced to store the value to be received. The execution of the action assigns this value to the variables of the when clause of both of the charts.

- $B' [> B''$

Let

$$H_1 = \{j | < i, j', n', p', c', f', h', > \in R'(j) \\ \wedge < a, j_1', n, p, c, f, h > \in R'(j') \wedge name(a) = \delta\}$$

$$H_2 = \{j' | < i, j', n', p', c', f', h', > \in R'(j) \\ \wedge < a, j_1', n, p, c, f, h > \in R'(j') \wedge name(a) = \delta\}$$

$$H_3 = \{j | < i_r, j', n', p', c', f', h', > \in R'(j)\}$$

Choose $m =< J' \cup J'' - \{j_o''\}, N' \cup N'', V' \cup V'', R, j_o', Z' \cup Z'', h_o' o h_o'' >$ *where*

$$E = E'' - \left\{E''\left(j_o''\right)\right\} \cup E' \cup E_1$$

$$R = R' \cup \left(R'' - R''\left(j_o''\right)\right) \cup R_1$$

$$E_1 = \left\{< j, X > | < j_o'', X > \in E'' \wedge R'(j) \neq \phi \wedge j \notin (H_1 \cup H_2 \cup H_3)\right\}$$

$$R_1 = \{< a, j, j'', n, p, c, f, h > | < a, j'', n, p, c, f, h > \in R''(j_o) \\ \wedge R'(j) \neq \phi \wedge j \notin (H_1 \cup H_2 \cup H_3)\}$$

The EFSM charts m' and m'' are combined by making all the rules of m'' emanating from j_o'' to emanate from every state in m'.

- $(\mu X (v_1, ..., v_n).B')(t_1, .., t_n))$

Choose $m =< J', N', E, V', R, j_o', Z', h_o >$ *where*

$$E(j) = \begin{cases} \left(E'(j) \cup E'\left(j'_o\right)\right) - X & if \ < j, X > \in E' \\ E'(j) & \text{otherwise} \end{cases}$$

$$R(j) = \begin{cases} < a, j'_o, n, p, c, f, h > & if \ < a, j, j', n, p, c, f, h > \in R' \wedge < j', X > \in E' \\ R'(j) & \text{otherwise} \end{cases}$$

and $h_o =< v_1 \leftarrow t_1, .., v_n \leftarrow t_n > o\ h'_o$

The recursion $\mu X\ B'$ is formed by replacing each extension X in m' by the rules and extension of j'_o in m'. Further, the actual values $t_1, .., t_n$ have to be assigned to the variables $v_1, .., v_n$.

As an illustration, let us consider the following **Readytosend** process which is actualized in some other process as **Readytosend**[ISAP,IPdu,d3] (number5)

process Readytosend[ISAP,IPdu,d3] (number7:Sequencenumber):exit :=
(ISAP?sp7:SP;
([isIDATreq(sp7)]
—> IPdu!DT(number7,Data(sp7)); d3!s(0)!number7!Data(sp7);exit
[] [not(isIDATreq(sp7))] —> Readytosend[ISAP,IPdu,d3] (number7))
[] IPdu?ipdu7:IPDU[not(isDR(ipdu7))];
Readytosend[ISAP,IPdu,d3] (number7))

endproc

The recursive rule produces the following chart:

m1=< { 1, 2, .. , 6 }, {n1, n2, .. ,n7 }, ϕ, {sp7, number5, number7, ipdu7 },
R, 1, {5}, number7 ←number5 >, where R=
{ < ISAP?sp7:SP, 1, 2, n1, true, true, ϵ, ϵ >,
< IPdu?ipdu7:IPDU, 1, 3, n2, true, [not(isDR(ipdu7))], ϵ, ϵ >,

< i_r, 3, 1, n3, true, true, ϵ, { number7 $\leftarrow$ number7} >,

< i, 2, 4, n4, [isIDATreq(sp7)], true, ϵ, ϵ >,

< IPdu!DT(number7, Data(sp7)), 4, 5, n4, true, true, ϵ, ϵ >,

< d3!s(0)!number7!Data(sp7), 5, 6, n6, true, true, ϵ, ϵ >,

< i, 2, 7, n7, [not(isIDATreq(sp7))], true, ϵ, ϵ >,

< i_r, 7,1, n7, true, true, ϵ, { number7 $\leftarrow$ number7} >]

- *hide* **S** in B'

Choose $m =< J', N', E', V', R, j_o, Z', h_o' >$ *with*

$R = \{r | r \in R' \wedge name(when_r) \notin \mathbf{S}\} \cup \{r(\Psi) | r \in R' \wedge name(when_r) \in \mathbf{S}\}$

where $\Psi = i/when_r, f/action_r$ *with*

$$f = \begin{cases} action\ o\ (v_i \leftarrow any) & if\ when_r = gd_1..d_n \wedge d_i = ?v_i : s_i \\ action_r & \text{otherwise} \end{cases}$$

The chart corresponding to *hide* **S** in B' is obtained by modifying the rules of the chart corresponding to the behaviour B'. If there exists a rule with a *when* clause of the form $gd_1...d_n$, where $g \in$**S**, then the *when* clause is transformed to i, and for each d_i of the form $?v_i : s_i$, the assignment clause of the rule is updated by the random assignment $v_i \leftarrow any$.

As an illustration, let us consider the following behaviour

hide d3 in

Readytosend[ISAP,IPdu,d3] (number5)

|[d3]| d3?z6:DecNumb?number6:Sequencenumber?olddata6:ISDU; exit

We have already constructed chart *m1* for the process **Readytosend** in the example discussed to explain process instantiation and recursion. The chart for the behaviour

d3?z6:DecNumb?number6:Sequencenumber?olddata6:ISDU ; exit

is as follows

m2=< {7,8 }, {n8 }, ϕ, {z6, number6, olddata6 }, R, 7, {8}, ϵ >, where R=

{< d3?z6:DecNumb?number6:Sequencenumber?olddata6:ISDU, 7, 8, n8, true,

true, ,ϵ, ϵ >}

The composition of *m1* ||[d3]|| *m2* yields the following chart, where the states and transitions are renumbered.

m=< {9,.., 15}, {n9, .. , n16 }, ϕ, {sp7, number5, number7, ipdu7,z6,
number6, olddata6}, R, 9, {13}, number7 ←number5 >, where R=
{ < ISAP?sp7:SP, 9, 10, n9, true, true, ϵ, ϵ >,
< IPdu?ipdu7:IPDU, 9, 11, n10, true, [not(isDR(ipdu7))], ϵ, ϵ >,
< i_r, 11, 9, n11, true, true, ϵ, { number7 ← number7} >,
< *i*, 10, 12, n12, [isIDATreq(sp7)], true, ϵ, ϵ >,
< IPdu!DT(number7, Data(sp7)), 12, 13, n13, true, true, ϵ, ϵ >,
< d3!s(0)!number7!Data(sp7), 13, 14, n14, true, true, { z6←s(0),
number6←number7, olddata6←data(sp) }, ϵ >,
< *i*, 10, 15, n115, [not(isIDATreq(sp7))], true, ϵ, ϵ >,
< i_r, 15, 9, n16, true, true, ϵ, { number7 ← number7} >}

Now **hide** d3 in *m* produces the following chart:

m=< {9,.., 15}, {n9, .. , n16 }, ϕ, {sp7, number5, number7, ipdu7,z6,
number6, olddata6}, R, 9, {13}, number7 ←number5 >, where R=
{ < ISAP?sp7:SP, 9, 10, n9, true, true, ϵ, ϵ >,
< IPdu?ipdu7:IPDU, 9, 11, n10, true, [not(isDR(ipdu7))], ϵ, ϵ >,
< i_r, 11, 9, n11, true, true, ϵ, { number7 ← number7} >,
< *i*, 10, 12, n12, [isIDATreq(sp7)], true, ϵ, ϵ >,
< IPdu!DT(number7, Data(sp7)), 12, 13, n13, true, true, ϵ, ϵ >,
< *i*, 13, 14, n14, true, true, { z6←s(0),
number6←number7, olddata6←data(sp) }, ϵ >,
< *i*, 10, 15, n115, [not(isIDATreq(sp7))], true, ϵ, ϵ >,

< i_r, 15, 9, n16, true, true, ϵ, { number7 $\leftarrow$ number7} >}

In the EFSM chart derived from LOTOS, the action clause of rule r represents the variables that are updated by the function f due to value passing in the interprocess communication, whereas the assignment clause of rule r represents value passing due to process instantiation and the **let** construct.

Example 3.4 We consider another example, which we will use in the next chapter. It is the first half of the connection phase (i.e., up to and including Connection Indication or Disconnection Primitives) of a single connection of a Transport Service [5].

```
P= ConReq-A;
      ( ConInd-B; exit
       [] DisReq-A
            ( i ; exit
             [] ConInd-B ; exit )
      [] i ; DisInd-A;
            ( i ; exit
             [] ConInd-B ; exit )
      )
```

The address at which a primitive occurs is indicated by extending the name of the primitive with –A(calling user) and –B(called user). The chart construction algorithm produces the following EFSM chart:

$m =< \{1,..,10\}, \{n1,..,n9\}, \phi, \phi, R, 1, \{4,7,8,9,10\}, \epsilon >$ *where* $R =$

{ < ConReq-A, 1, 2, n1, true, true, ϵ, ϵ >,

< i_s , 2, 3, n2, true, true, ϵ, ϵ >,

< ConInd-B, 2, 4, n3, true, true, ϵ, ϵ >,

< DisReq-A, 2, 5, n4, true, true, ϵ, ϵ >,

< DisInd-A, 3, 6, n5, true, true, ϵ, ϵ >,

< ConInd-B, 6, 7, n6, true, true, ϵ, ϵ >,

< i_s, 6, 8, n7, true, true, ϵ, ϵ >,

< ConInd-1', 5, 9, n8, true, true, ϵ, ϵ >,

< i_s, 5, 10, n9, true, true, ϵ, ϵ > }

3.2 FROM SDL TO EFSM CHART

SDL specifications can also be transformed into EFSM charts in two phases. In the first phase the specification is transformed into a semantically equivalent form. In the second phase it is converted into an EFSM chart by bottom-up synthesis.

3.2.1 Transformation of SDL Specification

In order to facilitate the chart construction algorithm, it is convenient first to apply some transformations to the SDL specification. The transformation rules are explained below:

- *Elimination of macro and procedure call*

The macro/procedure call can be replaced by an additional task construct and the corresponding description of that macro/procedure. The task construct can be used to assign the actual parameters to the formal ones. Local variables defined in the macro/procedure body are converted to global variables with unique identifiers. As an example, let us consider the data transfer process of HDLC protocol as shown in Figure 3.1 and the procedure retransmit_frames. The data transfer process is transformed to another one without any procedure call retransmit_frames as shown in Figure 3.2 but with the additional task construct to take account of the assignments.

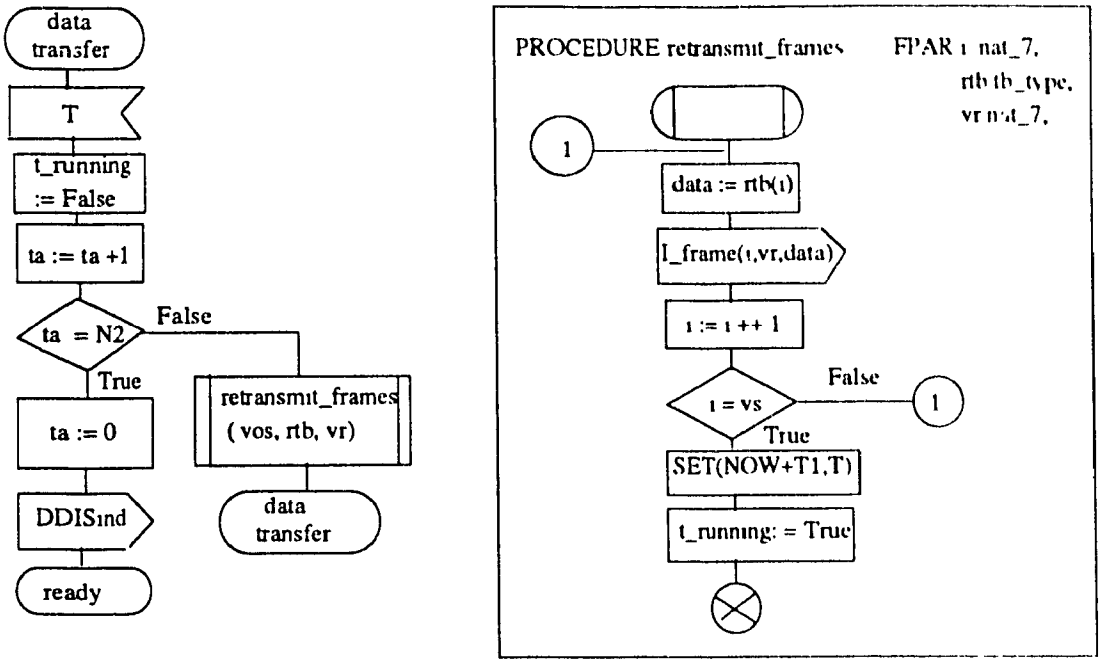


Figure 3.1 Data transfer process of HDLC and the procedure retransmit_frames.

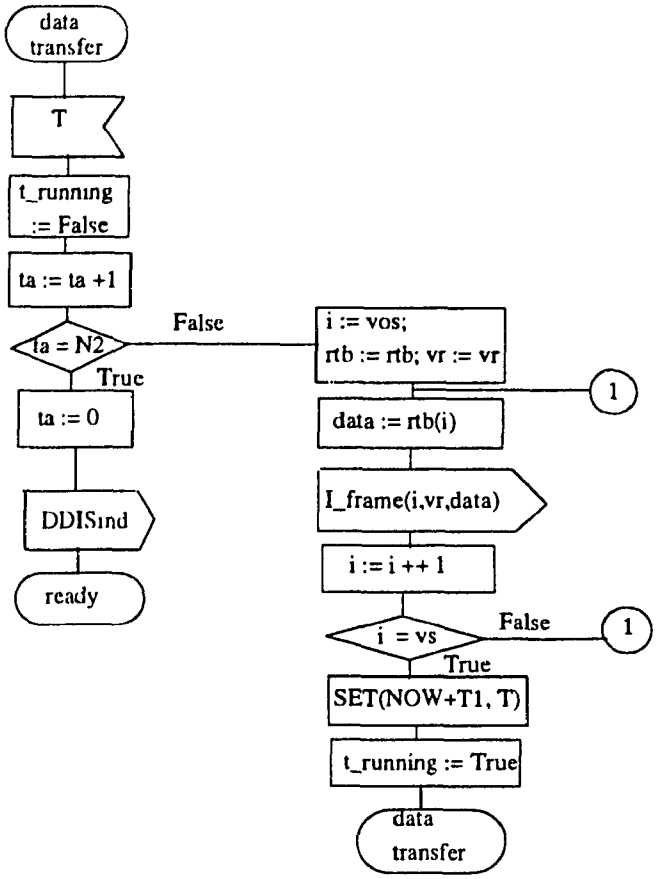


Figure 3.2 After the elimination of procedure call retransmit_frames.

- *Incorporation of channel/signal-routes name in input/output construct*

In SDL the channel/signal-route descriptions are specified only in the system diagram. It is not specified in the input/output construct. In order to achieve a compatible structure of the *action clause* in the rules of our unified chart model, we incorporate the channel/signal-route information in the input/output construct. This means, the channel/signal-route from which the signal is to be received and into which the signal is to be transmitted is specified in the input/output construct. As an illustration let us consider INRES-entity and process Coder_Ini [24] as shown Figure 3.3 and 3.4 respectively. In the input/output construct the channel name is not specified. For example, the input signal ICONreq is contained in the channel ISDU_in. However, it is not shown in the input/output construct explicitly. Coder_Ini process after the incorporation of channel name in input/output structure is shown in Figure 3.5.

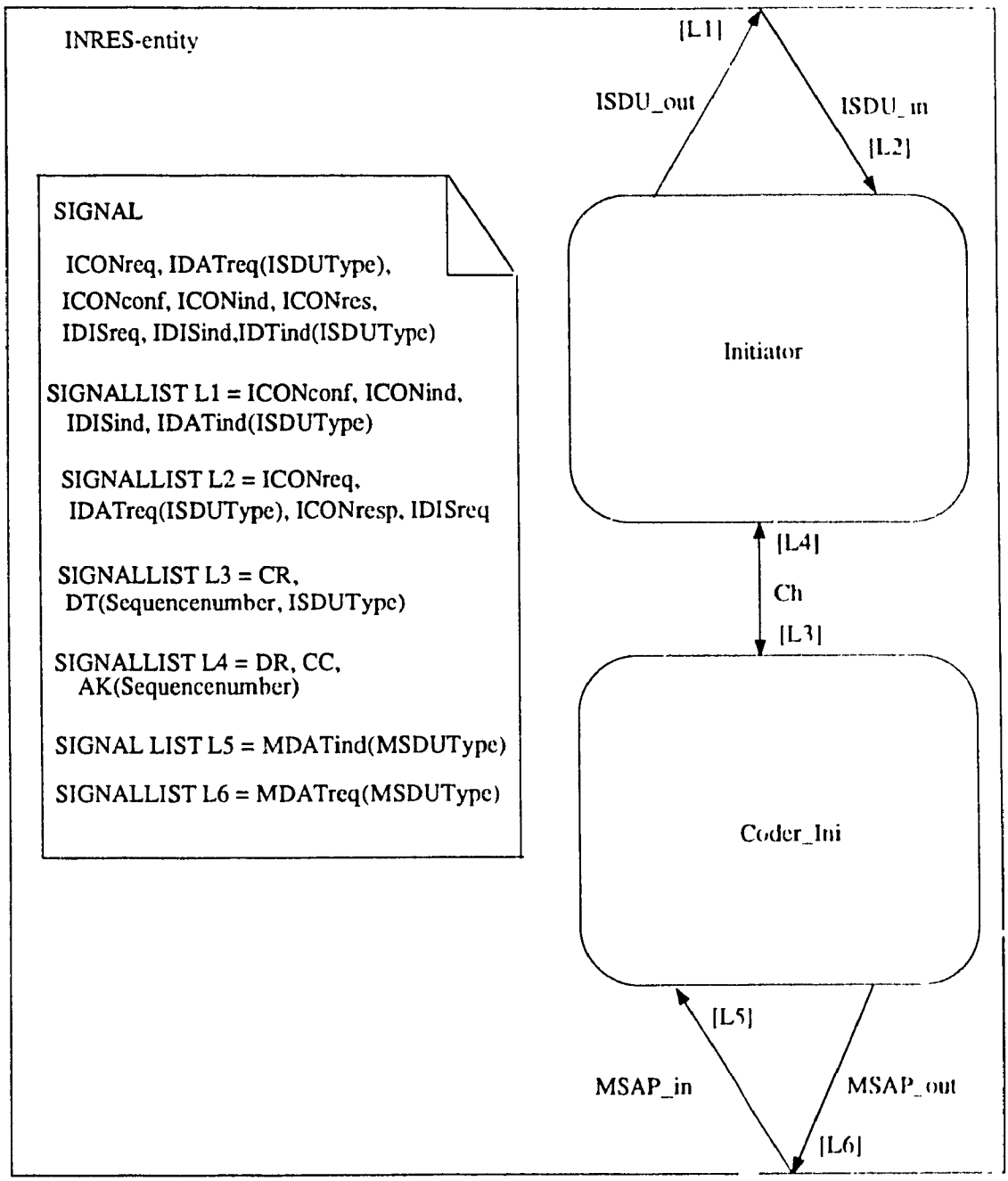


Figure 3.3 Definition of INRES_entity.

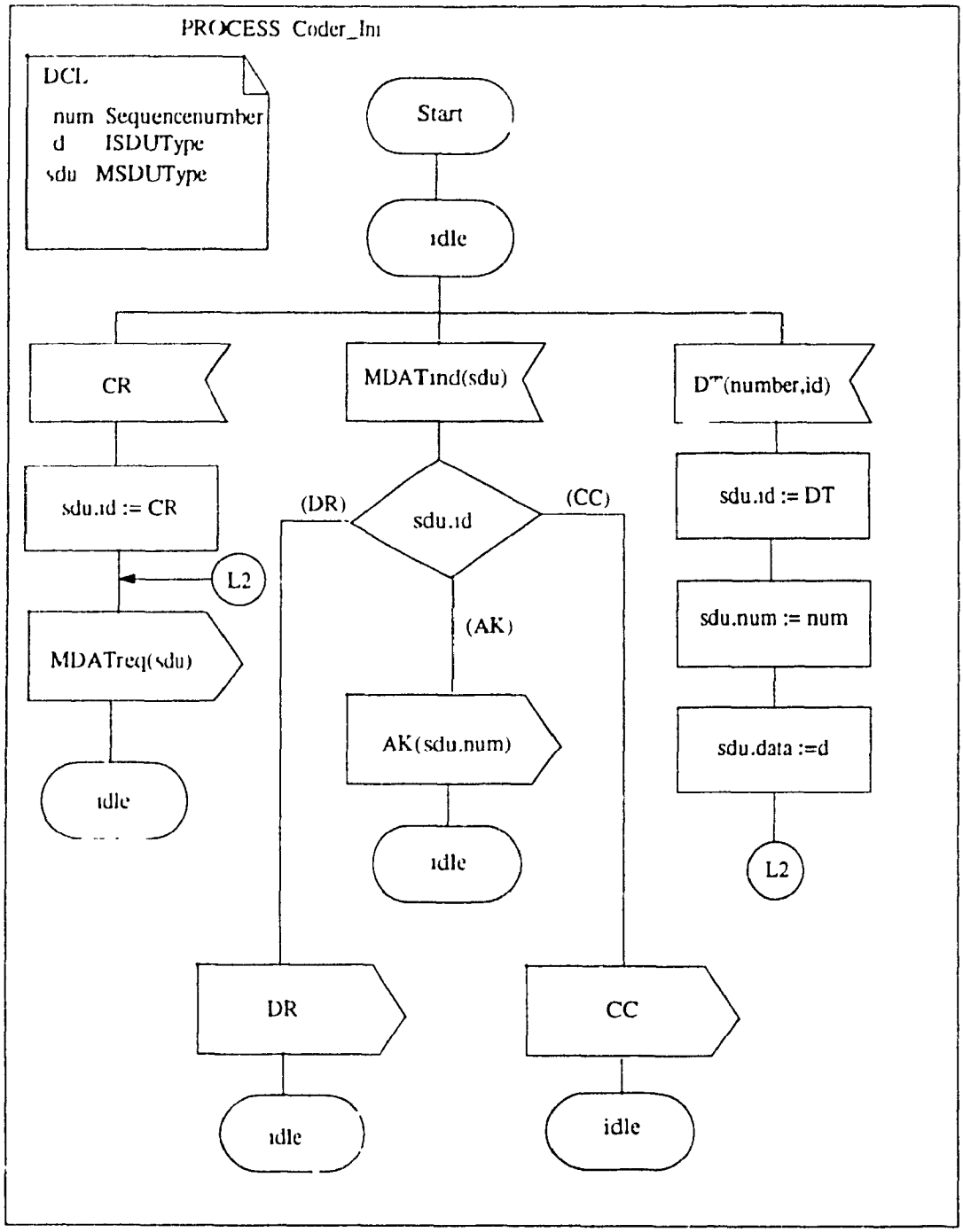


Figure 3.4 Coder_Ini process.

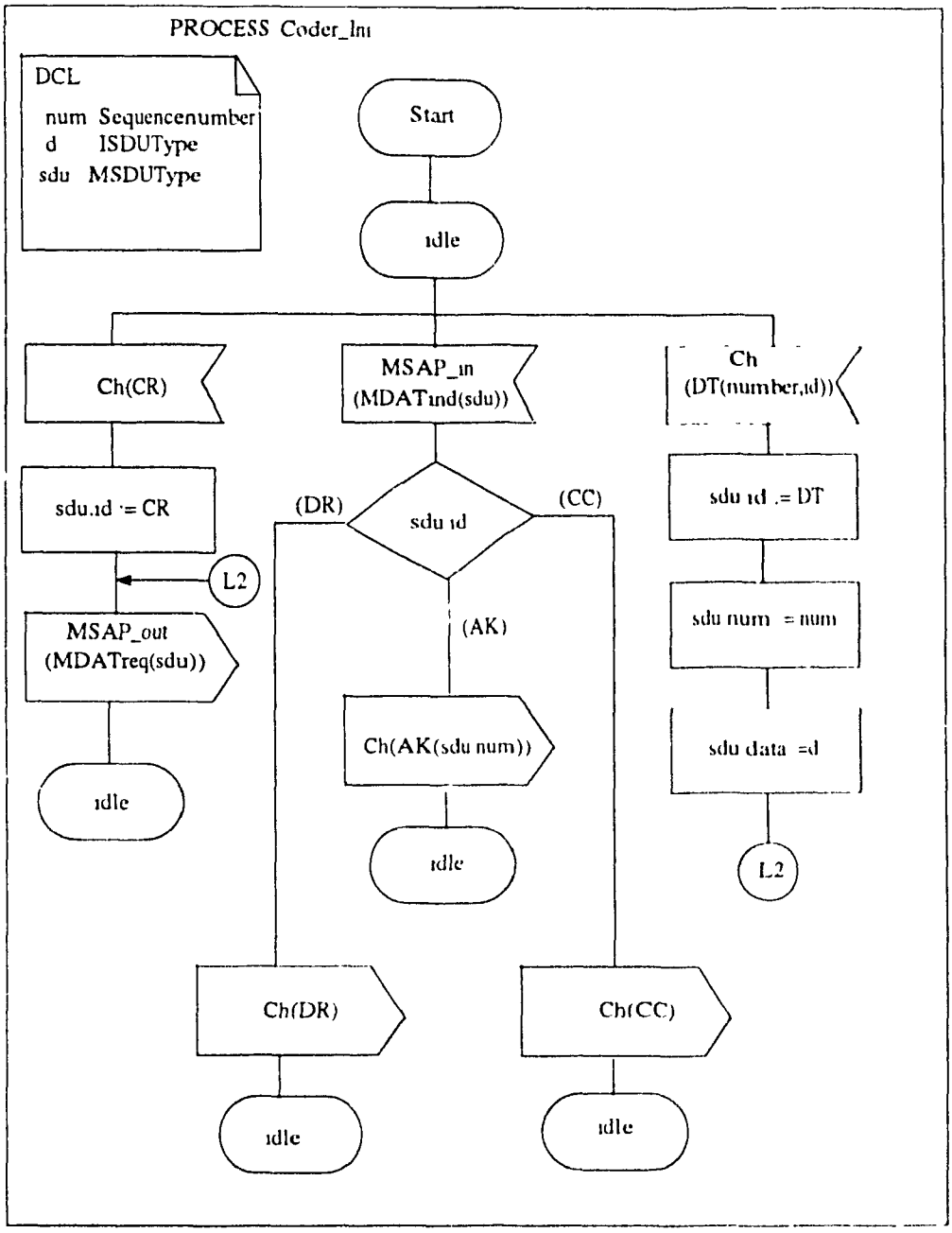


Figure 3.5 Modified Coder_Ini process.

- *Transformation of save construct to states and input construct*

The save construct can be eliminated by rewriting the SDL using other constructs. An algorithm proposed in [47] can be used for that purpose. The basic idea is to replace the save construct by additional states and input constructs. As an example consider a part

of the **Initiator** process of the Inres protocol shown in Figure 3.6, where the IDATreq is saved in the state **Sending**.

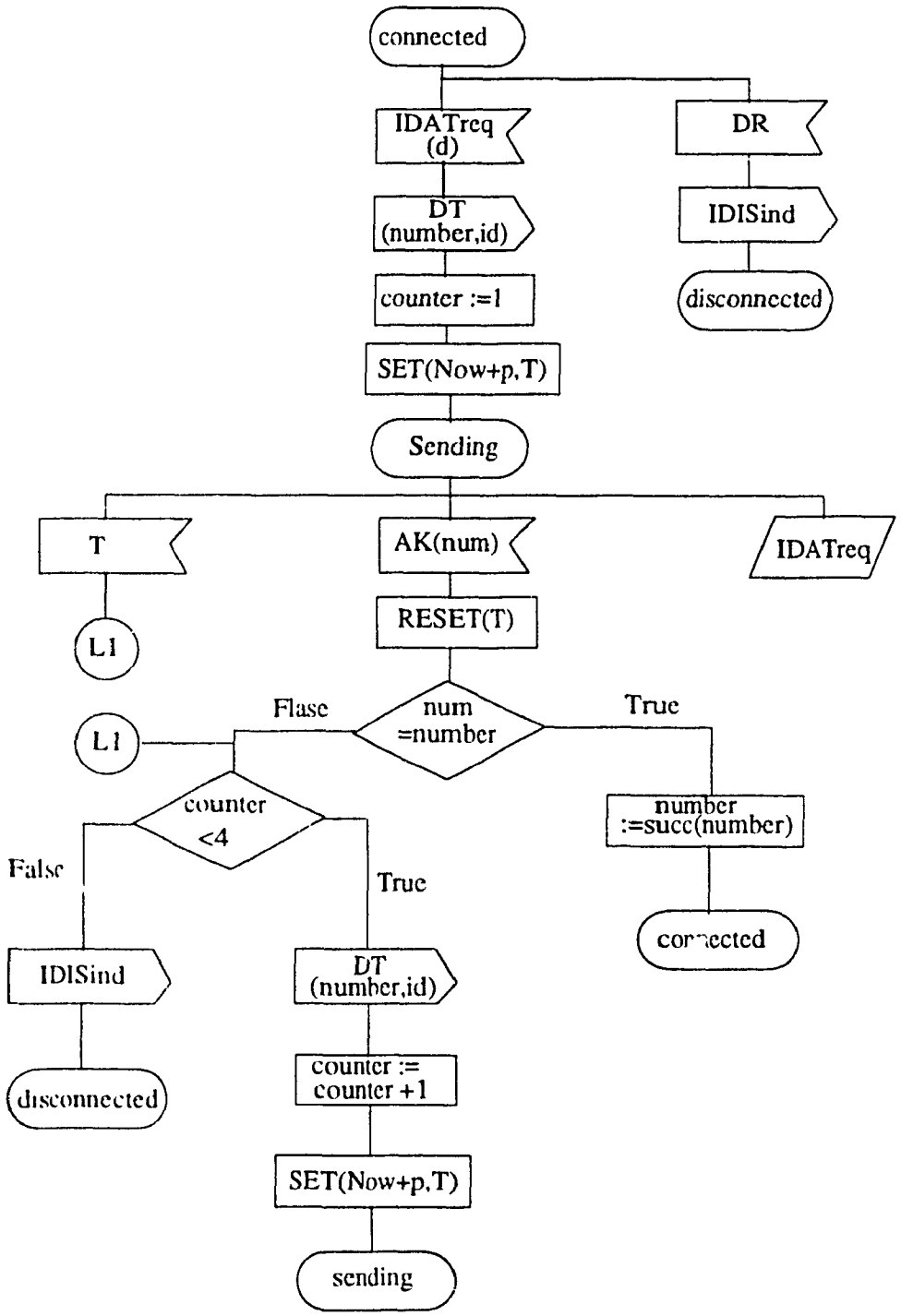


Figure 3.6 Part of the Initiator process of Inres protocol with Save construct.

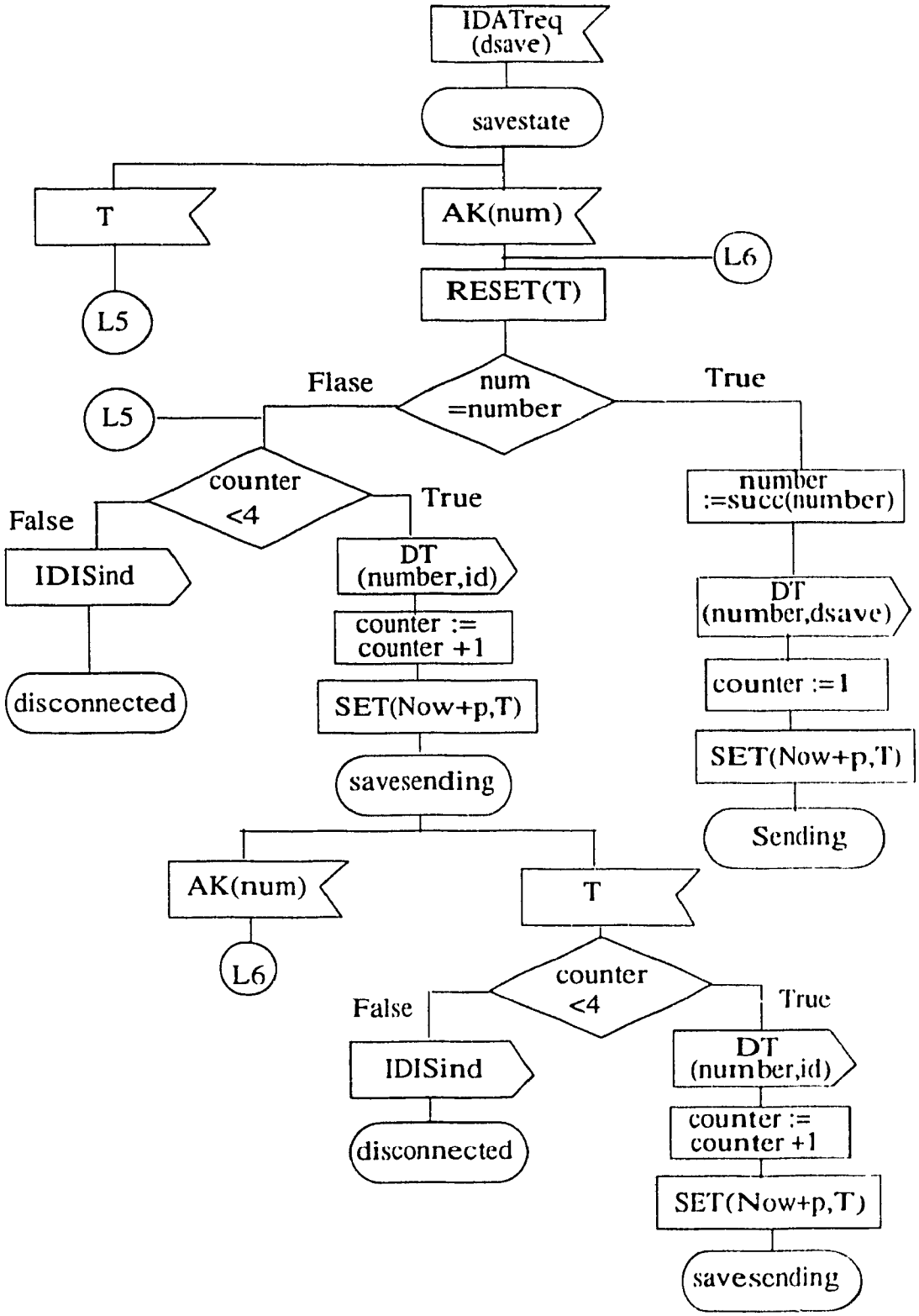


Figure 3.7 Behaviour that may be replaced for Save construct IDATreq.

New states are introduced to transform the save construct, which is shown in Figure 3.7.

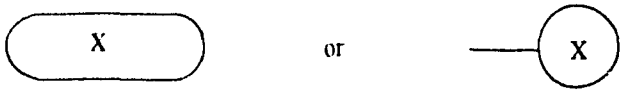
3.2.2 The Chart Construction Algorithm

In this section we shall present the algorithm to translate SDL to a chart. In the translation algorithm given below, the output statement *chan*($sig(t_1, \ldots, t_n)$) of SDL transformed into *chan*!$sig(t_1, \ldots, t_n)$ in the chart. Similarly, the input statement *chan*($sig(v_1, \ldots, v_n)$) can be transformed into *chan*?$sig(v_1, \ldots, v_n)$.

Algorithm: Chart Construction Algorithm:

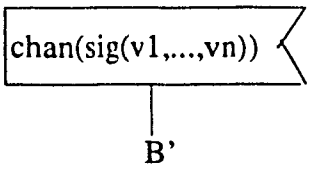
In the second phase the EFSM chart is constructed again by bottom up synthesis. This phase is straight forward and similar to that of the LOTOS chart construction algorithm. Here, we give details of SDL-GR constructs and the processing involved. We start with an empty chart. A complete chart is constructed bottom up following the graphical structure specified in SDL-GR. Steps of processing that correspond to different SDL constructs are given below. In the following, we assume that for the behaviour graph B' there is a chart $m' =< J', N', E', V', R', j_o', Z', h_o >$. Similarly for B''. Let j_o be a control state not in J' and J''. If $f{:}D_1 \rightarrow D_2$ and $g{:}D_2 \rightarrow D_3$, then their composition $gof{:}D_1 \rightarrow D_3$ is defined by $gof = g(f(x))$. Let $\epsilon{:}V \rightarrow D$ be an arbitrary but a fixed function. The action clause f of the rules generated from the SDL specification is empty, because of the single process assumption. However, due to compatibility with the rules generated from LOTOS specification, we keep the action clause in the rule. The EFSM chart construction is terminated when the start construct is encountered.

Next state and goto constructs



Choose $m =< \{j_o, \phi, < j_o, X >, \phi, \phi, j_o, \phi, \epsilon\}$.

Input construct

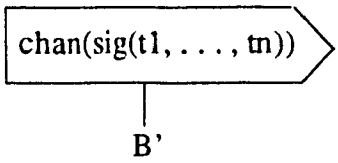


Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V, R, j_o, Z', \epsilon >$, *where*

$V = V' \cup \{v1, v2, .., vn\}$ *and*

$R = R' \cup \left\{ chan?sig\,(v1, .., vn) : signal, j_o, j_o^{'}, n_o, true, true, \epsilon, h_o^{'} \right\}$

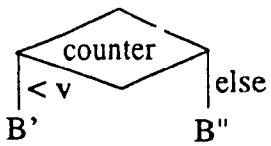
Output construct



Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V, R, j_o, Z', \epsilon >$, *where*

$R = R' \cup \left\{ chan!sig\,(t1, .., tn) : signal, j_o, j_o^{'}, n_o, true, true, \epsilon, h_o^{'} \right\}$

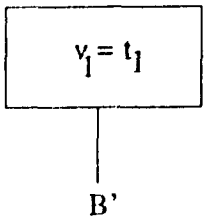
Decision construct



Choose $m =< J' \cup J'', \{j_o\}, N' \cup \{n_1, n_2\}, E' \cup E'', V' \cup V'', R, j_o, Z' \cup Z'', \epsilon >$, *where*

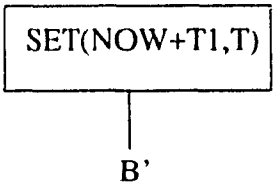
$$R = R' \cup R'' \cup \left\{ < i, j_o, j_o^{'}, n_1, \text{counter} < v, true, \epsilon, h_o^{'} > \right\}$$
$$\cup \left\{ < i, j_o, j_o^{''}, n_2, \text{counter} \geq v, true, \epsilon, h_o^{''} \right\}$$

Task construct



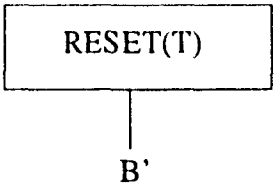
Choose $m =< J', N', E', V, R', j_o', Z', h_o >$, *where* $V = V' \cup \{v_1\}$ and $h_0 =< v_1 \leftarrow t_1 > o\ h_o'$.

Set construct



Choose $m =< J', N', E', V, R', j_o', Z', h_o >$, *where* $V = V' \cup \{v_1\}$ and $h_0 =< T \leftarrow NOW + T1 > o\ h_o'$.

Reset construct



Choose $m =< J', N', E', V, R', j_o', Z', h_o >$, *where* $V = V' \cup \{v_1\}$ and $h_0 =< T \leftarrow 0 > o\ h_o'$.

Choice construct

Choose

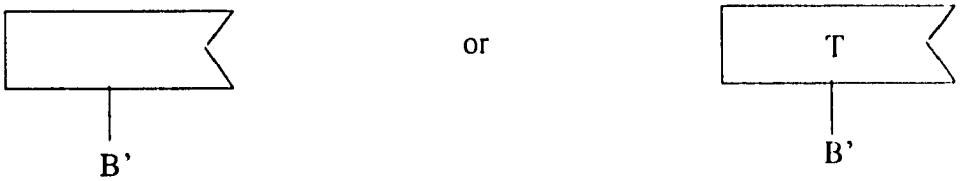
$$m =< \left(J' - \left\{j_o'\right\}\right) \cup \left(J'' - \left\{j_o''\right\}\right) \cup \{j_o\}, N' \cup N'', E',$$
$$V' \cup V'', R, j_o, Z' \cup Z'', h_o'' o h_o' >$$

$$where \quad E = \left\{< j_o, X > | X \in \left(E'\left(j_o'\right) \cup E''\left(j_o''\right)\right)\right\} \cup \left(E' - \left\{E'\left(j_o'\right)\right\}\right)$$
$$\cup \left(E'' - \left\{E''\left(j_o''\right)\right\}\right)$$

$$R = \left\{< a, j_o, j, n, p, c, f, h > | < a, j, n, p, c, f, h > \in R'\left(j_o'\right) \cup R''\left(j_o''\right)\right\}$$
$$\cup \left(R' - R'\left(j_o'\right) \cup \left(R'' - R''\left(j_o''\right)\right)\right)$$

The sum of two processes is formed by adding a new control state j_o which has the rules and extensions of both m' and m''.

Spontaneous and timeout transitions

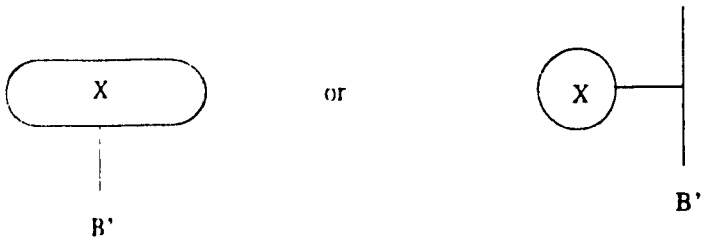


Choose $m =< J' \cup \{j_o\}, N' \cup \{n_o\}, E', V', R, j_o, Z', \epsilon >, \quad where$

$R = R' \cup \left\{< i_s, j_o, j_o', n_o, true, true, \epsilon, h_o' >\right\};$

i_s is used to represent the internal/timeout event. We deal with timeout transitions in the same way as spontaneous transitions.

State and label constructs



Choose $m =< J', N', E, V', R, j_o', Z', h_o' >$, $where$

$$E(j) = \begin{cases} (E'(j) \cup E'(j_o')) - X & if \ < j, X > \in E' \\ E'(j) & \text{otherwise} \end{cases}$$

$$R(j) = \begin{cases} < a, j_o', n, p, c, f, h > & if \ < a, j, j', n, p, c, f, h > \in R' \wedge < j', X > \in E' \\ R'(j) & \text{otherwise} \end{cases}$$

3.3 SIZE OF THE EFSM CHART

The size of the EFSM chart generated from SDL and LOTOS depends on the size of the specification as well as how it is structured. The upper limit to the number of control states generated from the LOTOS construct is given by the following formulae. The norm $|\ |_r$ and $|\ |_s$ means, the number of rules and states respectively.

$|stop|_s = |exit|_s = 1$

$|X|_s = 2$

$|a; B|_s = 1 + |B|_s$

$|B1 \,[]\, B2|_s = |B|_s + |B|_s + 1$

$|B1 [> B2|_s = |B1|_s + |B2|_s$

$|B1 \,|[S]|\, B2|_s = |B1|_s \times |B2|_s$

The upper limit to the number of rules generated from the LOTOS construct is given by the following formulae.

$|stop|_r = |exit|_r = 0$

$|X|_r = 1$

$|a; B|_r = |B|_r + 1$

$|B1 \,[]\, B2|_r = |B1|_r + |B2|_r$

$|B1 [> B2|_r = |B1|_r + |B2|_r + |B1|_s$

$|B1 \,[|S|]\, B2|_r = |B1|_r \times |B2|_r$

The number of control states and rules generated from the SDL construct is given by the following formulae.

$\|next\ state\|_s = 1$	$\|next\ state\|_r = 0$
$\|goto\|_s = 1$	$\|goto\|_r = 0$
$\|input\|_s = 1$	$\|input\|_r = 1$
$\|output\|_s = 1$	$\|output\|_r = 1$
$\|decision\|_s = 1$	$\|decision\|_r = n$
$\|reset\|_s = 0$	$\|reset\|_r = 0$
$\|task\|_s = 0$	$\|task\|_r = 0$
$\|set\|_s = 0$	$\|set\|_r = 0$
$\|timeout\|_s = 0$	$\|timeout\|_r = 1$
$\|state\|_s = 0$	$\|state\|_r = 0$
$\|label\|_s = 0$	$\|label\|_r = 0$

CHAPTER IV

GENERATION AND ANALYSIS OF TEST CASES

We develop a new test generation algorithm, which takes nondeterminism into consideration, in section 4.1. In section 4.2, we propose an algorithm for the construction of the data flow graph, which is used to identify the protocol functions. Section 4.3 deals with the use of the zero-one integer programming technique to select test cases that are optimal in nature to meet certain test coverage requirements or to exercise certain protocol functions adequately. In section 4.4, we model the test case by a dependency graph and evaluate it by taking predicate slices. The dependency graph is similar to a program dependency graph [14]. Finally in section 4.5, redundant assignments and predicates in all the feasible test cases are removed by reducing the test cases.

4.1 TEST CASE GENERATION ALGORITHM

The action i_s plays an important role in the generation of test cases. The presence of i_s in the EFSM chart makes it highly nondeterministic. A test case generated from a deterministic finite state machine consists of a sequence of events, i.e., it is linear, whereas a test case in the context of an EFSM chart, which is nondeterministic in nature takes the form of a rule of the chart. We shall eventually see how to generate test cases from the EFSM chart. Before that we need some definitions.

We use symbols $r_1, r_2, \ldots$ to represent the rules of a chart. For any arbitrary rule r_l, $from_{r_l}$ and to_{r_l} refer to the *from* and *to* clauses of rule r_l. A *directed path* in a chart $m =< J, N, E, V, R, j_o, Z, h_o >$ is finite sequence of rules $r_1, .., r_k$ such that $to_{r_1} = from_{r_2}$, $to_{r_2} = from_{r_3}$, .., $to_{r_{k-1}} = from_{r_k}$. A *directed path* $r_1, .., r_k$ is a *cycle* if $from_{r_1} = to_{r_k}$ A *directed path* is a *subtour* if $from_{r_1} = to_{r_k} = j_o$. The control state j is called an *unstable* state, *iff* there exists a rule $r \in R(j)$ such that $when_r = i_s$, otherwise it is called *stable*.

Next we define two unary operators: *Contraction* and *Expansion*. By *contraction* of a rule r we refer to the operation of removing r and identifying its control state $from_r$ and to_r. If $m =< J, N, E, V, R, j_o, Z, h_o >$ is a chart, then relative to any subset R_c of R, we define the chart $m_c = m@R_c =< J', \{J_c\}, \phi, \phi, R - R_c, J_c, \epsilon, \epsilon >$ to be a condensed chart of m, which is obtained by contracting the rules belonging to R_c. The control state $J_c = \{from_r,\ to_r | r \in R_c\}$ is called a *pseudo-control* state. By *expansion* of a *pseudo-control* state we refer to the operation of identifying its *pseudo-control* state, and forming an EFSM chart corresponding to the rules that have been *contracted* to form the *pseudo-control* state. Formally, if $m_c =< J' \cup \{J_c\}, \phi, \phi, \phi, R - R_c, J_c, \epsilon, \epsilon >$ where J_c is the *pseudo-control* state, then $m_e = Exp(m_c) =< J_c, \phi, \phi, \phi, R_c, j_o, \epsilon, \epsilon >$ is called an *expanded* EFSM chart, which is obtained by expanding the *pseudo-control* state J_c.

Algorithm: Test Case Generation

Input: Chart $m =< J, N, E, V, R, j_o, Z, h_o >$

Output: A list of test cases

Method: We define a procedure TESTGEN. The algorithm consists of a call to TESTGEN.

Procedure TESTGEN(*m*)

begin

1. let *ST* be the set of all subtours of the chart *m*
2. for each $R_s \in ST$ do

   begin
3. k:=1;
4. let m_{ck} be the condensed chart $m@R_s$;
5. while the *pseudo-control* state J_{ck} of m_{ck} is *unstable* do

   begin
6. find a cycle $C_k = \{r_1, .., r_l\}$ in the chart m_{ck} such that $when_{r_1} = i_s$ and $from_{r_1}, to_{r_l} \in J_{ck}$:
7. form a condensed chart $m_{ck+1} = m_{ck}@C_k$;

8. k:=k+1;

end;

9. let $Exp(m_{ck}) =< J_c, \phi, \phi, \phi, R_e, j_o, \epsilon, \epsilon >$;

10. Print R_e:

11. Print "end of one test case";

end;

end.

Example 4.1 The chart obtained from the Transport Service Specification of Example 3.4 is given in Figure 4.1. The set of control states, J is given as { 1, 2, 3, 5, 6 }. The initial control state j_o is 1. Here we assume that after *exit* the control is going back to the initial state. The set of rules is given as { n1, n2, n3, n4, n5, n6, n7, n8, n9 }. The rules are identified by their corresponding transition numbers.

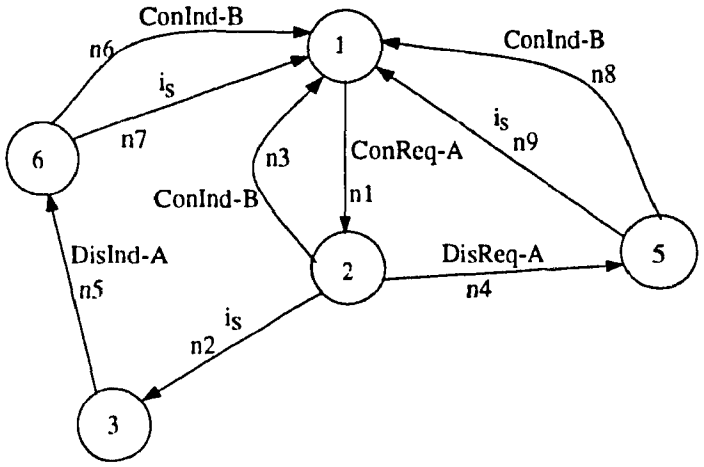


Figure 4.1 Transport service specification chart.

Let us consider the *subtour* R_s = {n1, n3 }. The condensed chart m_{c1} =$m@R_s$ is shown in Figure 4.2a and has the *pseudo-control* state J_{c1} = {1, 2}, which is *unstable*. Now consider the cycle C_1={n2, n5, n6} of the condensed chart m_{c1}. The chart, m_{c2}=$m_{c1}@C_1$, obtained by contracting the rules of the cycle C_1, is shown in Figure 4.2b. The new *pseudo-control* state J_{c2}={ 1, 2, 3, 6 } is still *unstable*. Proceeding one more step, we obtain the chart m_{c3}=m_{c2}@n7 shown in Figure 4.2c with the *pseudo-control* state J_{c3} =

{ 1, 2, 3, 6 }, which is *stable*. A test case is obtained by expanding the *pseudo-control* state J_{c3} and it is shown in Figure 4.2d.

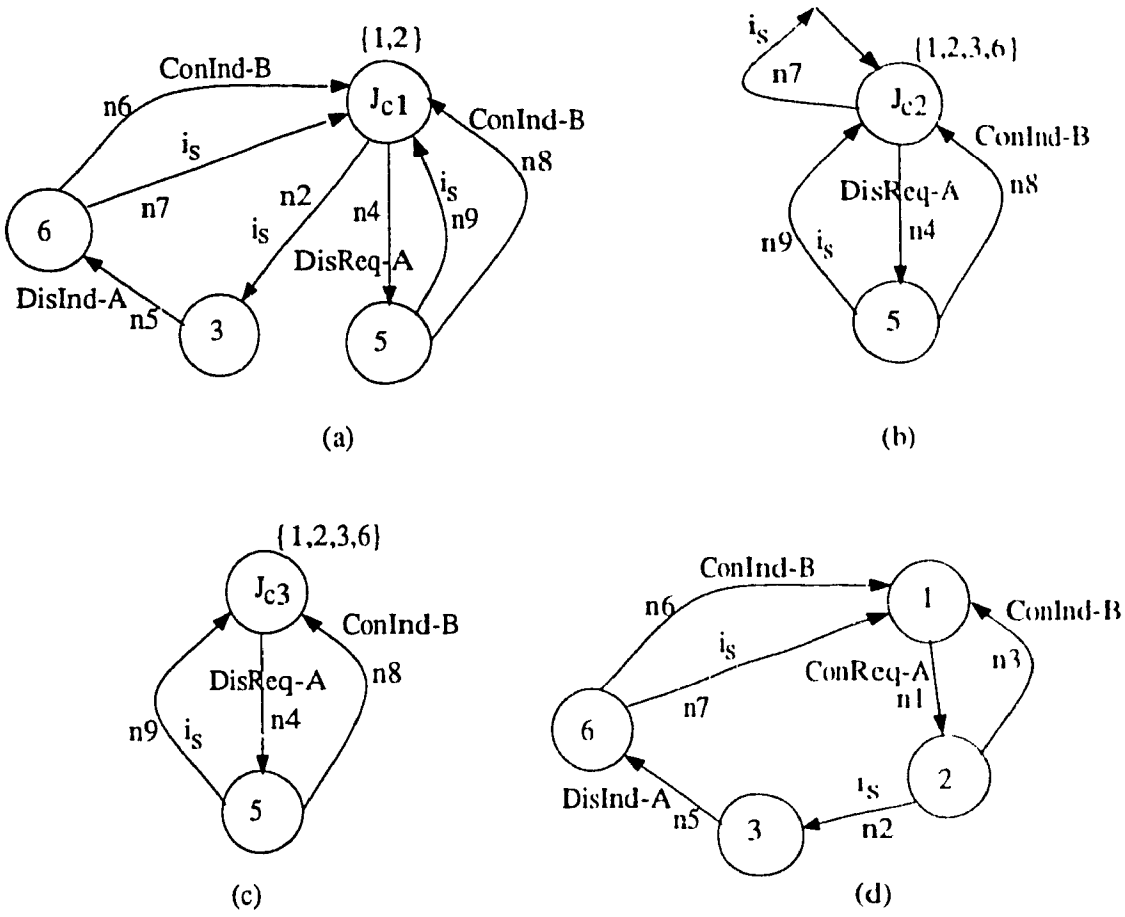


Figure 4.2 (a) m_{c1}, condensed chart $m@R_s$. (b) m_{c2}, condensed chart $m_{c1}@C_1$. (c) m_{c3},condensed chart m_{c2}@n7. (d) A test case of transport service specification.

The time complexity of the test case generation algorithm is of order $O(\ |J|^2|N|^2 + |N|_s(\ |J| + |N|\ ))$, where J, N, and $N_s \subset N$, are the control states, the transitions, and the spontaneous transitions respectively in the chart m. The first component, i.e., $|J|^2\ |N|^2$ is due to the partial test case generation and the second component is added to update the partial test cases to complete test cases.

The basic idea behind the algorithm is as follows. First a transition tour of the EFSM chart is generated. The tour is divided into sequences that start from the initial state and end either in the initial or one of the final states. This sequence is called

a *subtour* or a partial test case. The *subtour* may contain spontaneous transitions (i_s transitions). Next, the existence of a spontaneous transition that is not present in the *subtour*, but is an alternative to any one of the transitions in the subtour is checked. Then the *subtour* is updated by adding a sequence of transitions such that the sequence starts with the spontaneous transition and ends either in a final state or in a state belonging to the *subtour*. The procedure is repeated until no spontaneous transitions exist that are alternative to the updated subtour. The *subtour* is then a complete test case.

Transition tour generation is based on converting the EFSM chart into an Euler graph and then traversing this graph, each time including a distinct edge into the tour [11]. An edge is virtually added to the EFSM chart from each final state to go to the initial state.

The total number of test cases is determined as follows: d_{in}(initial state)+d'_{in}(initial state), where d_{in} (initial state) is the indegree of the initial state and d'_{in}(initial state) is the number of edges incoming to the initial state added during the conversion of the EFSM chart to an Euler graph.

4.2 DATA FLOW GRAPH

The flow of data reflects how input primitive parameters determine the values of the context variables and they in turn determine the parameter values of the output primitives. Input/output primitives in the case of protocols are the Abstract Service Primitives (ASPs) and Protocol Data Units (PDUs).

A Data Flow Graph (DFG) models the flow of data in the chart. Four types of nodes are used in a DFG:

- *i*–nodes represent input primitives
- *d*–nodes represent variables and data
- *f*–nodes represent ADT operators
- *o*–nodes represent output primitives

Edges are used to represent the flow of information. Each edge of the data flow graph is labeled with the Transition Number (TN) to which it belongs. A DFG is designed to

visualize the flow of data. Guards and selection predicates are not taken into consideration in the DFG to avoid cluttering of the graph.

Formally, a data flow graph is a digraph $G = (V, E)$, whose edges are assigned with a label $l : E \rightarrow 2^{\mathrm{TN}}$, where

V = {m | m is an *i*–node, *d*–node, *f*–node, or *o*–node}

E = {e | e is an *id*–edge, *if*–edge, *io*–edge, *df*–edge, *do*–edge, *fo*–edge, *dd*–edge, *ff*–edge}

$\mathrm{TN} = \{n | < a, j, j', n, p, c, f, h > \in R\} \cup \phi$

It is nece sary to identify operations associated with different kinds of the PDUs and ASPs. This can be achieved by looking at the PDU and ASP definition in the abstract data types. To draw the data flow graph we have to scan the *when*, *action*, and *assignment* clause of each rule of the chart. An algorithm for the construction of the data flow graph is given below:

Algorithm: Data flow graph construction.

Input: Chart m, Data type definitions.

Output: Data Flow Graph.

Method: We assume that appropriate data structures are available to create different types of nodes and arcs. Also available in the structure is a place for label for each node and a facility to create a linked list of attached transition numbers for each edge. In addition to these components, for simplicity we assume that the assignment statement is either 1)$x \leftarrow op(x, y)$, 2) $x \leftarrow op(y)$ or 3) $x \leftarrow y$. We define a procedure DATAFLOW_GRAPH

Procedure DATAFLOW_GRAPH

S1. For each assignment $h_o \in \{x \leftarrow c | c \in T_{\Omega_D}\}$ in m do

i. create two *d*–nodes: node(x) and node(c) with label x and c;

ii. construct an edge from node(c) to node(x) with label ϕ;

For each rule $r \in R$ in the chart m do the following steps.

S2. If $offer(when_r) \neq$ undefined do

i. if $when_r = g?x : s$; create an *i*–node node(*x*), with label *x*.

ii. if $when_r = g?Sig$, where *Sig* is of the form *Signal(y, z)*, *Signal(y)*, or *Signal*

{ we refer to case a), case b), and case c) }

If *d*–nodes for *y* and *z* are undefined, create those nodes with label *y* and *z*. Let those *d*–nodes be node(*y*) and node(*z*)

case a): Create two *i*–nodes: node(*Signal.y*) and node(*Signal.z*) with label *Signal.y* and *Signal.z* respectively; construct edges from node(*Signal.y*) and node(*Signal.z*) to node(*y*) and node(*z*), respectively, with label $transition_r$.

case b): Create an *i*–node: node(*Signal.y*) with label *Signal.y*; construct an edge from node(*Signal.y*) to node(*y*) with $transition_r$.

case c): Create an *i*-node node(*Signal*) with label *Signal.*

iii. if $when_r = g!E$, where *E* is of the form *op(y, z)*, *op(y)*, or *y*

{ we refer to case a), case b), and case c) }

If *d*–nodes for *y* and *z* are undefined, create those nodes with label *y* and *z*.

Let those *d*–nodes be node(*y*), and node(*z*).

case a): Determine, if *op* is a constructor(ASP) operator; if yes, then determine operators *op*1 and *op*2 (from ADT definition) through which one can access *y* and *z*. Create two *o*–nodes: node(*op.op*1) and node(*op.op*2), with label *op.op*1 and *op.op*2 respectively; construct edges from node(*y*) and node(*z*) to node(*op.op*1) and node(*op.op*2), respectively, with label $transition_r$.

Otherwise, create an *o*–node node(*op*) with label *op*; construct two edges from node(*y*) and node(*z*) to node(*op*) with label $transition_r$.

case b): Create an *o*–node node(*op*) with label *op*. Construct an edge from node(*y*) to node(*op*) with label $transition_r$.

case c): Create an *o*–node node(*y*) with label *y*. Construct an edge from *f*–node: node(*y*) to *o*–node node(*y*) with label $transition_r$.

iv. if $when_r = g!Sig$, where *Sig* is of the form *Signal(y, z)*, *Signal(y)*, or *Signal*

we refer to case a), case b), and case c) }

If *d*–nodes for *y* and *z* are undefined, create those nodes with label *y* and *z*. Let those *d*–nodes be node(*y*) and node(*z*)

case a): Create two *o*–nodes: node(*Signal.y*) and node(*Signal.z*) with label *Signal.y* and *Signal.z* respectively; construct edges from node(*y*) and node(*z*) to node(*Signal.y*) and node(*Signal.z*) respectively, with label $transition_r$.

case b): Create an *o*–node: node(*Signal.y*) with label *Signal.y*; construct an edge from node(*y*) to node(*Signal.y*) with label $transition_r$.

case c): Create an *o*-node node(*Signal*) with label *Signal*.

S3. for each assignment statement of $action_i$ and $assignment_r$ do

i. let the assignment statement be either 1)$x \leftarrow op(y, z)$, 2) $x \leftarrow op(y)$ or 3) $x \leftarrow y$. If *d*–nodes x, y, and z are undefined, create those nodes with label x, y, and z. Let those nodes be node(x), node(y), and node(z).

ii. if the assignment statement is of the form $x \leftarrow op(y, z)$, then determine if *op* is a constructor operator(PDU). If yes, then determine operator *op*1, *op*2 (from ADT operator) through which one can access *y* and *z*. Create two *o*–nodes node(*op.op*1) and node(*op.op*2) with label *op.op*1 and *op.op*2, respectively. Construct edges from node(*y*) and node(*z*) to node(*op.op*1) and node(*op.op*2), respectively, with label $transition_r$;

Otherwise, determine if there is a *f*–node node(*op*) which is adjacent to node(*y*) and node(*z*) {this is to catch common subexpressions}. If not, create such an *f*–node; construct two edges from node(*y*) and node(*z*) to node(*op*) and one edge from node(*op*) to node(*x*) with labels $transition_r$; otherwise attach $transition_r$ to each edge incident on it, i.e., on node(*op*).

iii. if the assignment statement is of the form $x \leftarrow op(y)$, then determine if there is an *f*–node:node(*op*), which is adjacent to node(*y*). If not, create such an *f*–node:node(*op*): construct an edge from node(*y*) to node(*op*) and one edge from node(*op*) to node(*x*) with label $transition_r$ to the edge incident on it.

iv. if the assignment statement is of the form $x \leftarrow y$, then construct an edge from node(y) to node(x) with label $transition_r$.

4.2.1 Decomposition of Data Flow Graph

Here, we present a method that involves identification of protocol functions from the data flow graph similar to the one defined for Estelle specifications [52]. We are interested in obtaining slices of data flow according to user defined criteria. We call the final slices obtained data flow functions. Slicing is done in two steps: the first step is called blocking and second step is called merging, where the criteria must be provided by the user based on knowledge of the specification.

(1) *Blocking*: The data flow graph is divided into blocks $\mathcal{B}_1, \mathcal{B}_2, .., \mathcal{B}_N$ as follows:

- All variable d–nodes in the data flow graph are processed by creating a block for the d–node, or including it in one of the blocks already created if it feeds other d–nodes. In the block included are all f–nodes that feed d–nodes, all i–nodes that feed that d–node directly or indirectly. Constant d–nodes that feed the d–node are also included in the block. We also include all f–nodes, o–nodes that are fed by the d–node directly or indirectly in the block. We call the blocks created in this step type-1 blocks.
- In some cases where there is a direct data flow from i–nodes to o–nodes, a different block is created. If the flow is through one or more f–nodes these nodes are included in the block. We call blocks created in this step type-2 blocks.

The blocks that have incoming arc(s) from other blocks are called *data dependent* blocks. All other blocks are independent.

(2) *Merging*:

Step1. After a protocol function is selected, the first thing to do with the data flow graph is to determine related *d*–nodes. We can then merge all the type-1 blocks associated with these *d*–nodes.

Step2. Any type-2 blocks that belong to the function are merged with blocks created in step 1.

4.3 TEST SELECTION FROM PROTOCOL FUNCTION

The problem of selecting a minimal set of test cases, which meets certain coverage criteria, is referred as the test case selection problem. The criteria may be node testing, branch testing or some function of the protocol identified from the data flow graph. Once the criteria are identified by the test designer, a coverage frequency matrix $[a_{ij}]$ can be generated for those criteria. The general structure of the coverage frequency matrix $[a_{ij}]$ may be represented as:

	n_1	n_2	...	n_p
t_1	a_{11}	a_{12}	...	a_{1p}
t_2	a_{21}	a_{22}	...	a_{2p}
.	.	.	...	.
.	.	.	...	.
t_q	a_{q1}	a_{q2}	...	a_{qp}

where

(i) t_i stands for the *i*th test case generated from the chart.

(ii) n_j stands for *j*th test element, which is considered to be covered.

(iii) $[a_{ij}]$ stands for coverage frequency of test case t_i over the tested element n_j.

The complete test case and the set of tested elements of the coverage matrix can be identified as $T_c = \{t_1, t_2, .., t_q\}$ and $N_c = \{n_1, n_2, .., n_p\}$ respectively. Now the test case selection problem can be expressed as follows: Find a minimal set $T_c^* \subseteq T_c$ such that

each element of the N_r is covered at least once. This is a decision problem. It can be formulated in terms of a zero-one integer programming model as:

$$\begin{aligned} &\text{minimize} \quad z = \sum_{i=1}^{q} x_i \quad \text{subject to (s.t)} \quad \sum_{i=1}^{q} a_{ij} x_i \geq 1, \quad \text{for } j = 1,2,..,p \\ &\text{where } x_i = 1 \text{ or } 0, \quad \text{for } i = 1,2,..,q. \end{aligned} \tag{4.1}$$

where a_{ij} represents the coverage frequency matrix. Note that $z = \sum_{i=1}^{q} x_i$ stands for the actual number of selected test cases and $\sum_{i=1}^{q} a_{ij} x_i$ represent the actual total test case coverage over n_j. In fact it is required that for each n_j this coverage frequency matrix is to be no less than one. The equation 4.1 can be rewritten in the following form:

$$\begin{aligned} &\text{minimize} \quad z = \sum_{i=1}^{q} c_i x_i \quad \text{subject to (s.t)} \quad \sum_{i=1}^{q} b_{ij} x_i \leq d_j, \quad \text{for } j = 1,2,..,p \\ &\text{where } x_i = 1 \text{ or } 0, \quad \text{for } i = 1,2,..,q. \\ &\qquad [b_{ij}] = -[a_{ij}], \\ &\qquad d_j = -1, \quad \text{for } j = 1,2,..,p \\ &\qquad c_i = 1, \quad \text{for } i = 1,2,..,q \end{aligned} \tag{4.2}$$

Finding the solution to this problem is straightforward. Many algorithms are available [57]. Among them, the Balas' zero-one additive algorithm utilizing the branch and bound approach is considered to be the fastest one. This is discussed next.

Enumeration

To solve this problem, search algorithms are used which enumerate all 2^q possible zero-one vectors $\mathbf{x} = (x_1, x_2, .., x_q)$. In such procedures the vast majority of solutions are enumerated. The enumerative procedure can be illustrated by a search tree composed of nodes and branches. A node corresponds to a zero-one candidate solution **x**. Two nodes connected by a branch differ in the state of one variable. Each variable can be in one of three states: fixed at 1, fixed at 0, or free. A new node is defined by fixing a variable to 1 (forward step), and a node is revisited by fixing a variable to 0 (backtrack step). Figure 4.3 illustrates a search tree.

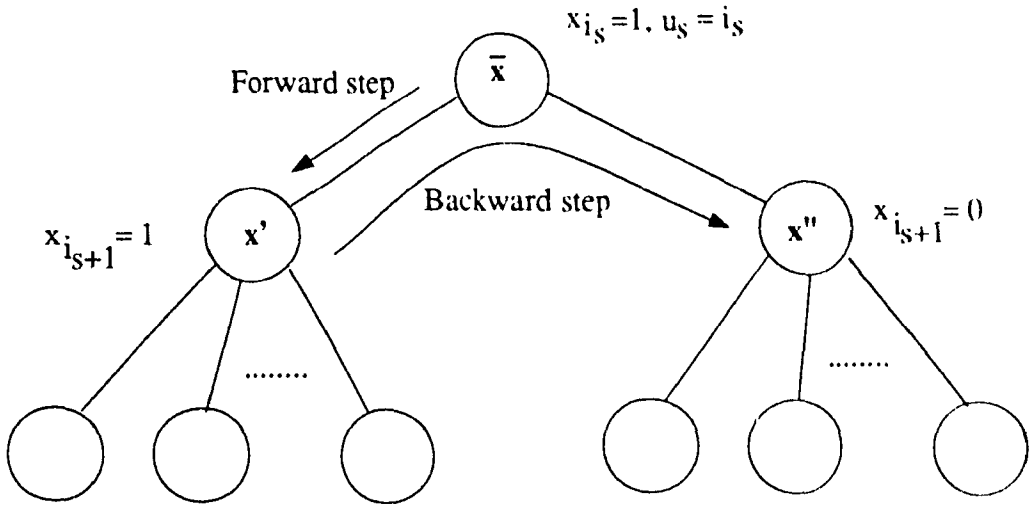


Figure 4.3 Search tree.

Let us assume that the current partial solution under consideration is given by the vector $\mathbf{u}^T = (u_1, u_2, .., u_s, 0, 0, 0, ..0)$ with entries interpreted as follows:

$$u_k = \begin{cases} i_k & if\ x_{i_k} = 1 \text{ and its complement has not yet been considered} \\ -i_k & if\ x_{i_k} = 1\ or\ 0 \text{ and its complement has been considered} \\ 0 & if\ k > s \end{cases}$$

We assume that we have an incumbent solution $\hat{\mathbf{x}}$, which is the best feasible solution so far. Finally, for the purpose of our discussion, let us assume that $x_{i_s} = 1$, $u_s = i_{s+1}$, i.e., the value of $x_{i_s} = 0$ has not been yet considered, and that at the node x the partial solution $\bar{\mathbf{x}}^T = (x_{i_1}, x_{i_2}, ..., x_{i_s}, 0, .., 0)$ is infeasible. Suppose now that we use a test that indicates that by adding another variable $x_{i_{s+1}}$, we can reduce the infeasibility and improve the objective function value. We move forward and add $u_{s+1} = i_{s+1}$ to the vector $\mathbf{u}$(i.e., set $x_{i_{s+1}} = 1$). At the new node $\mathbf{x}'$ we find that $\mathbf{x}'$ is feasible and it produces the objective function value better than the incumbent solution. We replace the incumbent solution by $\mathbf{x}'$ and update the accordingly. All *completions* in this partial solution with $x_{i_{s+1}} = 1$ have been enumerated since no feasible solution can be better by fixing any new free variable at 1. We say that partial solution has been *fathomed*; that is all completions of the partial solution have been implicitly enumerated. Thus we can consider all completions of $\bar{\mathbf{x}}$ with $x_{i_{s+1}} = 0$. We backtrack to the node $\mathbf{x}''$. Suppose

that there is no attractive completion for this node; that is, no completion of this partial solution can produce an optimal solution. At this point, we have enumerated all possible completions of $\bar{\mathbf{x}}$ with $x_{i_{s+1}} = 1$ and $x_{i_{s+1}} = 0$. We are ready to backtrack again and consider $x_{i_s} = 0$. The element u_{i_s} of the vector $\mathbf{u}$ is now set to $-i_s$ (we assumed previously that $x_{i_s} = 0$ had not been considered). If, on the other hand, $u_s = -i_s$ and $x_{i_s} = 0$ (i.e., $x_{i_s} = 1$ was considered previously), we find the rightmost positive element of $\mathbf{u}$, change its sign to negative, and examine the new partial completions. In this case we backtrack more than one branch of the search tree. To accomplish successfully the tree traversal we may have to move backward (backtrack) several branches of the tree. Resolving the current subproblems in the remaining free variables means enumerating explicitly each of the completions of the particular partial solution.

Implicit enumeration criteria

1. *Objective function and constraint improvement*

The purpose of this test is to find out if it is possible to improve the objective function value and reduce constraint infeasibilities. We can create the set $\mathbf{T}$ of variables such that

$$\mathbf{T} = \{ i : x_i \text{ is free}, \; z + c_i > \hat{z}, \; y_j < 0 \text{ for such } j \text{ such that } y_j = d_j - \sum_{i \in I} b_{ij} x_i < 0 \}$$

where I is the set of the current partial solution indices, $z = \sum_{i \in I} c_i$, and $\hat{z}$ be the best function value found so far. If $\mathbf{T}$ is empty, then backtrack step is justified.

2. *Infeasibility test*

If $\mathbf{T} \neq 0$ we may be able to identify an index j such that $y_j < 0$ and $y_j - \sum_{i \in I} min\,(0, b_{ij}) < 0$. Thus even if all variables in $\mathbf{T}$ are 1, the jth constraint will remain infeasible. In such a case, backtracking is also justified, since there is no feasible continuation.

3. *The Balas branching test*

The selection of the free variable to fix at 1 may significantly influence the algorithm efficiency. This is particularly true at the beginning, where a poor selection of a free variable could result in a needless enumeration of a large number of points that are not near a solution. A good rule designed to direct the search toward a solution has been given by Balas.

For each free variable x_i we create a set M_i,

$$M_i = \{j : y_j - b_{ij} < 0\}$$

and calculate

$$v_i = \sum_{j \in M_i} (y_j - b_{ij})$$

or set $v_i = 0$, if M_i is empty. We determine which free variable, if set to 1, would reduce the total infeasibility the most. By **total infeasibility** we mean the sum of the absolute values of the amount by which all constraints are violated. Thus we select the variable x_i such that it maximizes v_i. If all the sets M_i are empty, a backtrack step is taken. An algorithm based on these criteria has been designed by Balas and is called the Balas' zero-one additive algorithm [57].

4.4 TEST CASE DEPENDENCY GRAPH

In this section, we define the test case dependency graph (TCDG) in terms of the control flow graph of a test case. The control flow graph CG=(V, E, en) of a test case is a directed graph having a unique entry node en. V is a set of nodes corresponding to assignments (s-node), actions (a-node), and predicates (p-node). Graphically, a-node, p-node, and s-node are represented by a circle, triangle, and a rectangle respectively. E is a set of control edges, which represents a possible transfer of control from one instruction to another. The control edge from node v_i to node v_j is denoted by $v_i \rightarrow_c v_j$.

The test case dependency graph of a test case is the control graph of the test case with the addition of data dependency edges. A data dependency edge from node v_i to node v_j implies that the computation performed at node v_i directly depends on the value computed at node v_j. More precisely, it means that the computation performed at node v_i uses a variable, *var* that is defined at node v_j and there is an execution path from v_j to v_i along which the variable, *var* is not (re-)defined. The data dependency edge from node v_i to node v_j is denoted by $v_i \rightarrow_d v_j$. Graphically, control edges and data dependency edges are represented by bold and dashed lines.

Formally, a test case dependency graph for a test case t is a digraph $G_t=(V_t,E_t,en)$ with $V_t=V_a \cup V_s \cup V_p$, $E_t=E_d \cup E_c$, and a unique entry node $en \in V_t$, where

$$V_a = \{ v \mid v \text{ is an a-node} \};$$

$$V_s = \{ v \mid v \text{ is an s-node} \};$$

$$V_p = \{ v \mid v \text{ is a p-node} \};$$

$$E_d = \{ (u,v) \mid u \rightarrow_d v \};$$

$$E_c = \{ (u,v) \mid u \rightarrow_c v \}.$$

The events i, and i_r are suppressed in the test case dependency graph, because their presence is insignificant in the analysis. However, a spontaneous transition (i_s transition) is represented by an a-node, which plays an important role in the representation of a test case. Also, the initial assignment h_o of the chart should be taken into consideration in the construction of the test case dependency graph.

As an illustration, let us consider the following test case $\mathbf{t_1}$ generated from the **Inres** protocol:

< ISAP?sp3:SP, 1, 2, 1, true, true, ϵ, ϵ >,

< i, 2, 3, 2, [isICONreq(sp3)], true, ipdu9 $\leftarrow$ CR, ϵ >,

< i, 3, 4, 3, true, true, ipdu9 $\leftarrow$ CR, ϵ >,

< MSAP!MDATreq(ipdu9), 4, 5, 4, true, true, ϵ, ϵ >,

< i, 5, 6, 5, true, true, { z2 $\leftarrow$ s(0)}, {z4 $\leftarrow$ z2} >,

< MSAP?sp9:MSP, 6, 7, 6, true, true, ϵ, ϵ >,

< i, 7, 8, 7, true, [not(isDR(ipdu4))], {ipdu4 $\leftarrow$ data(sp9)}, ϵ >,

< i, 8, 9, 8, [isCC(ipdu4)], true, ϵ, ϵ >,

< ISAP!ICONconf, 9, 10, 9, true, true, ϵ, ϵ>,

< i, 10, 11, 10, true, true, ϵ, {number5 $\leftarrow$ succ(0), number7 $\leftarrow$ number5} >,

< ISAP?sp7:SP, 11, 12, 11, true, true, ϵ, ϵ >,

< i, 12, 13, 12, [isIDATreq(sp7)], true, ϵ, ϵ >,

< i, 13, 14, n13, true, true, ipdu9 $\leftarrow$ DT(number7, Data(sp7)), ϵ >,

< MSAP!MDATreq(ipdu9), 14, 15, 14, true, true, ϵ, ϵ >,

< i, 15, 16, 15, true, true, ipdu9 $\leftarrow$ DR, ϵ >,

< MSAP!MDATreq(ipdu9), 16, 17, 16, true, true, ϵ, ϵ >,

< ISAP!IDISind, 17, 1, 17, true, true, ϵ, ϵ >,

< i_s, 6, 18, 18, true, true, ϵ, ϵ >,

< i, 18, 19, 19, [z4<4], true, ϵ, ϵ >,

< i, 19, 20, 10, true, true, ipdu9 $\leftarrow$ CR, ϵ >,

< MSAP!MDATreq(ipdu9), 20, 21, 21, true, true, ϵ, ϵ >,

< i_r, 21, 6, 22, true, true, ϵ, z4 $\leftarrow$ s(z4) >

Figure 4.4 shows the test case dependency graph of the test case $\mathbf{t_1}$. The nodes are numbered as follows: first the transition number is placed followed by a period and the tuple number. For a-nodes the tuple number is 1, for p-nodes 5 or 6 depending on whether it is in the guard or condition clause of the transition. Similarly, for s-nodes the tuple number is 7 or 8 depending on whether it is in the action or assignment clause of the transition. For each assignment in the action or assignment clause, s-nodes are numbered with lower case letters starting with "a" following the number 7 or 8.

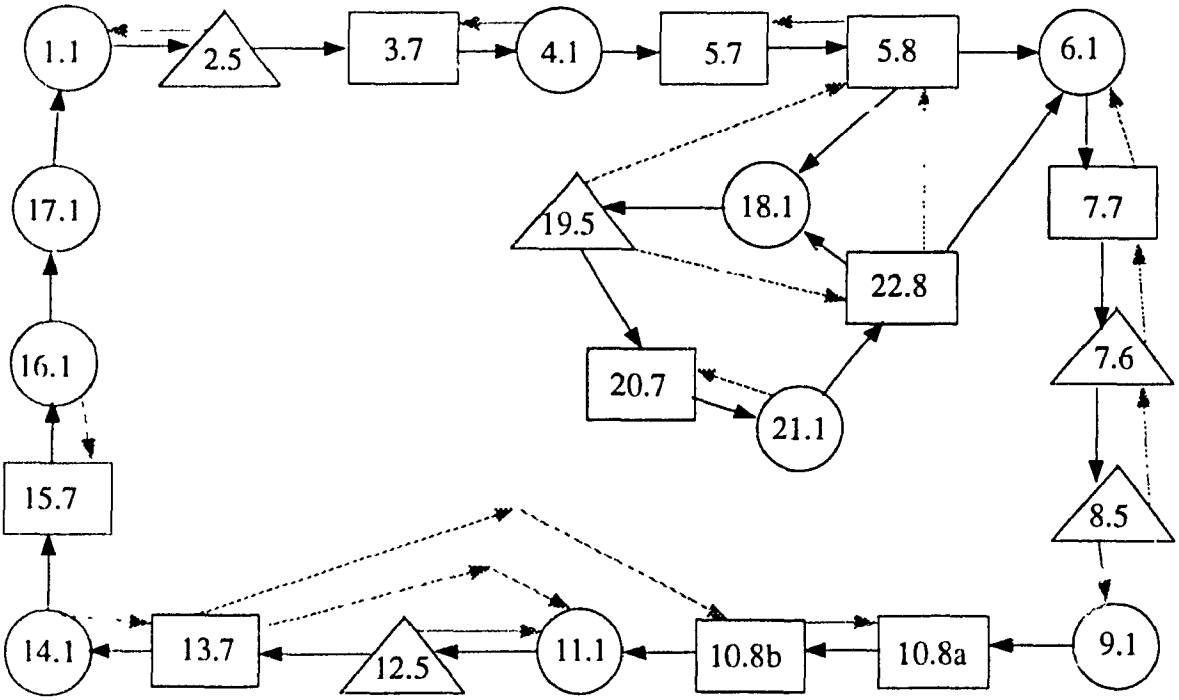


Figure 4.4 Test case dependency graph of the test case t_1.

4.4.1 Predicate Slices

The notion of *program slice*, originally introduced by Mark Weiser [64], is useful in program debugging, automatic parallelization, and program integration. Slicing is an abstraction of the set of statements that influence the value of a variable at a particular location. In this section, we use the concept of slicing with respect to a predicate in a test case to analyze the predicates in order to detect infeasible test cases. The predicate slice of a test case with respect to a predicate, pred at a p-node, consists of all nodes whose execution could possibly affect the boolean value of *pred* at the p-node. The predicate slice of a p-node can be easily constructed by traversing the data dependency edges of the test case dependency graph beginning at p-node. The nodes visited during traversal constitute the desired slice. We will provide an algorithm to get all the predicate slices from the test case dependency graph. Before that we need some definitions.

Given a node $v \in V_t$, we define the set $D_f[v]$, $D_t[v]$, $C_f[v]$, $C_t[v]$, $N_f[v]$, and $N_t[v]$ as follows:

$$D_t[v] = \{ (u,v) \mid (u,v) \in E_d \};$$

$$D_f[v] = \{ (v,w) \mid (v,w) \in E_d \};$$

$$C_t[v] = \{ (u,v) \mid (u,v) \in E_c \};$$

$$C_f[v] = \{ (v,w) \mid (v,w) \in E_c \};$$

$$N_t[v] = \{ u \mid (u,v) \in E_c \};$$

$$N_f[v] = \{ w \mid (v,w) \in E_c \}.$$

For a p-node p of a test case dependency graph G_t , the predicate slice of G_t with respect to p, denoted by $G_{t/p}$, is a graph containing all nodes on which p has a data dependency (i.e., all nodes that can reach from p via data dependency edge):$V(G_{t/p}) = \{ w \mid w \in V_t$ and $p \rightarrow^*_d w \}$. We extend the definition to the set of all p-nodes $V_p = \cup_i p_i$ as follows: $V(G_t / V_p) = V(G_t / \cup_i p_i) = \cup_i V(G_t / p_i)$. The edges in the graph G_t / V_p are essentially those in the subgraph of G_t induced by $V(G_t / V_p)$, with the restriction that only data dependency edges are included. We define $E(G_t / V_p) = \{ (v,w) \mid (v \rightarrow^*_d w) \in E_d$ and $v, w \in V(G_t / V_p) \}$.

Algorithm: PREDICATE_SLICES

Input: Test case dependency graph $G_t = (V_t , E_t)$

Output: Two sets $V' = V(G_t / v)$ and $E' = E (G_t / v_p)$ which represent predicate slices for each p-node $v_p \in V_t$.

The recursive procedure SLICE(v) adds edge (v,w) to E' if node w is first reached during the search by a data dependency edge from v. For each p-node, all the nodes are marked "new" and procedure SLICE is invoked.

1. Let V_p be the set of p-nodes in V_t;
2. $V' := V_p$;
3. $E' := \phi$;
4. for each v_p on V_p do
5. for all v in V_t do mark v "new";
6. SLICE(v_p);

```
procedure SLICE(v):

1.  add {v} to V;
2.  mark v "old";
3.  for each edge (v,w) on D_f[v] do
    begin
4.      if w is marked "new" then
            begin
5.                  add (v,w) to E';
6.                  SLICE(w);
            end;
    end;
```

The time complexity of the algorithm given above in the worst case is $O(|V_p|(|V_t| + |E_t|))$. Line 1 and 5 of PREDICATE_SLICES take time $O(|V_t|)$. The lines 5-6 are called exactly $|V_p|$ times. The time spent in SLICE is exclusive of recursive calls to itself, proportional to $|D_f[v]|$. Since $\sum_{v \in V_t} |D_f[v]| = O(|E_t|)$, the total cost of executing lines 3-5 of SLICE is $O(|E_t|)$. The procedure SLICE is called exactly once for each vertex $v \in V_t$, since v is marked "old" the first time SLICE is called. Thus the total time spent in PREDICATE_SLICES is $O(|V_p|(|V_t| + |E_t|))$

Figure 4.5 shows the graph that results from taking slices of the test case dependency graph from Figure 4.4 with respect to each of the p-nodes of the dependency graph. The same node numbering is used.

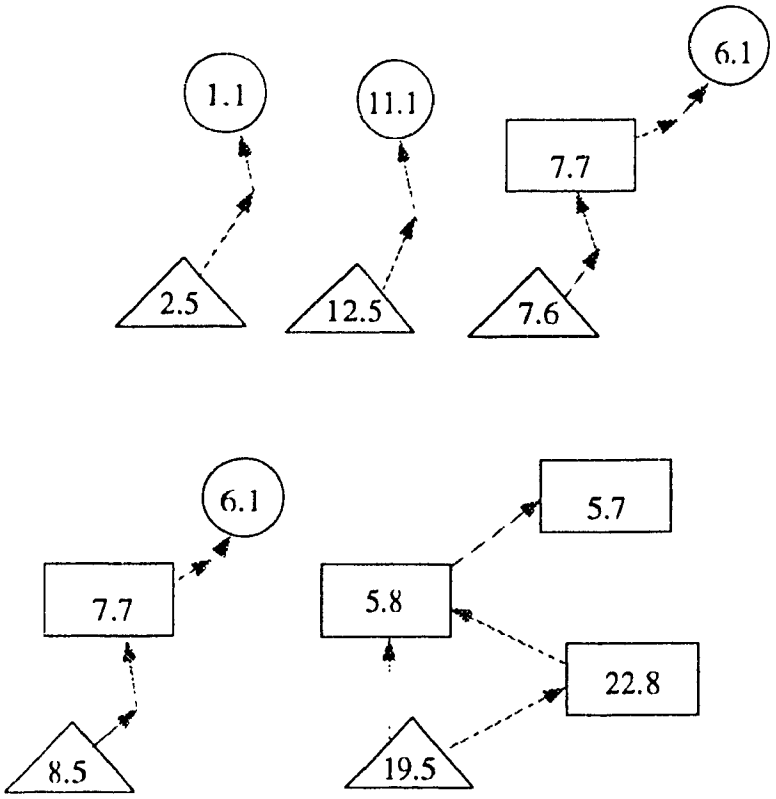


Figure 4.5 Predicate slices of the test case dependency graph t_1.

4.4.2 Infeasible Paths in Test Cases

Predicates play an important role in the evaluation of test cases generated from the protocol specification. Some of the generated test cases may not be feasible. In other words, the predicates can never be satisfied on a path due to the existence of an assignment on that path, that will cause the predicate to be set to false. Infeasible paths in a test case can be detected by evaluating the predicate slices.

To correct infeasibilities we have to exchange the transition containing the unsatisfiable predicate with a new transition, which means that the path following the old transition must be eliminated and a new path has to be selected following the new transition. The result of this is two-fold: the new path is the same as one of the test cases already generated, where no new test case is added; otherwise the new path becomes one of the test cases. In both cases the infeasible test case is eliminated.

4.5 REDUCTION OF TEST CASES

A test case can be reduced by eliminating extraneous statements that cannot affect the parameters of the input/output event or affect the control flow of the test case. One way of reducing the test case is by using test case dependency graph and the another way is by using data flow graph and the textual representation of the test case.

4.5.1 Reduction of Test Cases Using TCDG

Once the test case is modeled by a dependency graph, it can be reduced. The s-nodes in TCDG with no incoming data dependency edges can be eliminated, since these nodes neither affect the parameters of the output events nor affect the control flow of the test case. Since any infeasibilities are already removed before this step, we can reduce further the TCDG by eliminating all p-nodes from which a-nodes are not reachable through the data dependency edges. Clearly, these p-nodes have no *influence* on the input/output domain of the test case. In other words we keep all the predicates in the test case dependency graph that depend on the parameter values of the input primitives. Intuitively, redundant assignments and predicates are those that may be executed, but its elimination would not change the function of the test case computed over its domain. The resulting test case dependency graph obtained after elimination of redundant assignments and predicates is called the **reduced test case dependency graph.**

Algorithm: Reduced test case dependency graph (RTCDG).

Input: Test case dependency graph G = (V , E).

Output: Reduced test case dependency graph G′ = (V′, E′).

The recursive procedure SEARCH eliminates an s-node that has no data dependency edge incident on it. The main algorithm eliminates all the p-nodes from which a-nodes are not reachable through the data dependency edges.

1. V′:=V;
2. E′:=E;

3 Let I_p = { v $\in V_p$ | $v \not\rightarrow^*_d$ w, where w$\in V_a$ }

4. for each v$\in I_p$ do

begin

5. V′:=V′−v;

6. Let E_{new} := { (u,w) | w $\in N_f$ [v] and u $\in N_t$ [v]};

7. E′ := { E′− {D_f [v] ∪ C_f [v] ∪ C_t [v] }} ∪ E_{new};

end

8. SEARCH(V′ ,E′);

procedure SEARCH(V′, E′);

1. Let VD = { v | D_t [v] = ϕ and v$\in$ V′$_s$ $\subset$ V′ };

2. If VD <> Φ then

begin

3. Let v be any element in VD;

4. V′ := V′ − v;

5. Let E_{new} := { (u,w) | w$\in$ N_f [v] and u $\in$ N_t [v]};

6. E′ := { E′ − { D_f [v] ∪ C_f [v] ∪ C_t [v] }} ∪ E_{new} ;

7. SEARCH(V′ , E′);

end;

The time complexity of the algorithm given above in the worst case is $O(|V||V_s|)$. The cost of executing line 3 of RTCDG can be $O(|V||V_s|)$. The loop on the lines 5-7 of RTCDG is executed $|V_p|$ times. The total cost of executing SEARCH, exclusive of recursive call to itself is $O(|V|)$, since the total cost of executing the line 1 of SEARCH can be $O(|V|)$. The procedure SEARCH is invoked $|V_s|$ times, as each time one vertex $v \in V_s$ is deleted. Thus the total time spent in SEARCH is $O(|V||V_s|)$. Hence the time complexity of the RTCDG algorithm is $O(|V||V_s|)$.

The reduced test case dependency graph of the Inres test case $\mathbf{t_1}$ is shown in Figure 4.6. The algorithms discussed in this chapter are also applied to real life protocol such

as ACSE and HDLC protocol written in LOTOS and SDL respectively. The results are discussed in detail in chapter 6.

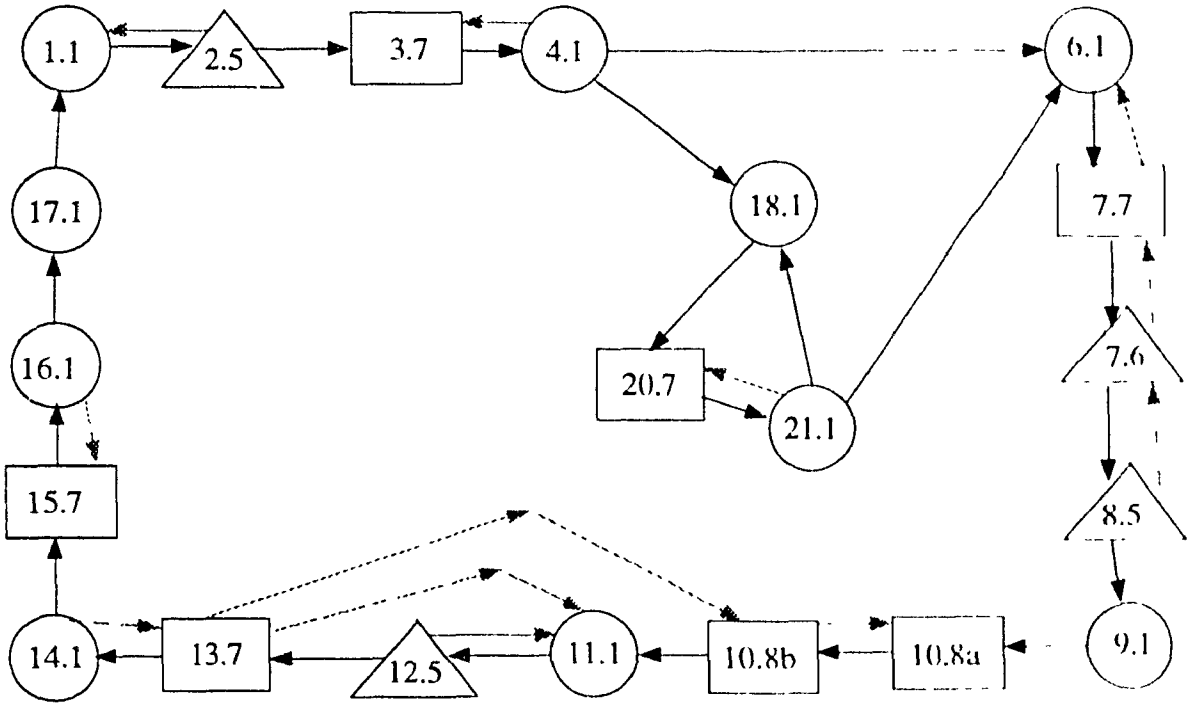


Figure 4.6 Reduced test case dependency graph of the test case t_1.

4.5.2 Reduction of Test Cases Using DFG

Data flow graph can be used to reduce the test cases. The way we have constructed the DFG is to show the dependency between the variables, i.e., where it is defined and where it is used. We use this dependency property to reduce the test cases. We need to define some quantities that will be useful in developing the algorithm.

Let $G = (V, E)$ be the data flow graph, whose edges are assigned with a label $l : E \rightarrow 2^{\mathrm{TN}}$, where $\mathrm{TN} = \{n | < a, j, j', n, p, c, f, h > \in R \} \cup \phi$. Let $E_{\mathrm{TNS}} \subset E$ with label $l_{\mathrm{TNS}} : E_{\mathrm{TNS}} \rightarrow 2^{\mathrm{TNS}}$, where $\mathrm{TNS} \subset \mathrm{TN}$ *and* $l_{\mathrm{TNS}} \subset l$. Assume that $G_{\mathrm{TNS}} = (V_{\mathrm{TNS}}, E_{\mathrm{TNS}})$, be an edge induced subgraph of G, induced by the edges E_{TNS}. An edge from node v_i to v_j in $G = (V, E)$ is denoted by $v_i \rightarrow v_j$. A node v_k is said to be reachable from a node v_1, if there exist nodes $v_2, v_3, \ldots v_{k-1}$ such that

$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_{k-1} \rightarrow v_k$, denoted by $v_1 \rightarrow^* v_k$. If a node v_k cannot be reachable from node v_1, then we write $v_1 \not\rightarrow^* v_k$. For each node $v \in V$, we define a function *label_name(v)*, which returns the name of the label(viz. name of a variable for a *d*-node).

Algorithm: Reduction of test case using DFG.

Input: DFG $G = (V, E)$ and the Test Case $m =< J_c, \phi, \phi, \phi, R_e, J_e, \epsilon, h_o >$.

Output: Reduced Test Case.

Method: We define a procedure REDJCTION.

Procedure REDUCTION

S1. Compute the set TNS $= \{n | < a, j, j', n, p, c, f, h > \in R_e \} \cup \phi$. Construct an induced subgraph $G_{\text{TNS}} = (V_{\text{TNS}}, E_{\text{TNS}})$ of $G = (V, E)$. Compute the sets:

$$D_1 = \{d - nodes \mid i - node \not\rightarrow^* d - node, \ for\, all\ i - node \in V_{\text{TNS}}\}$$

$$D = \left\{ d - nodes \mid d - node \overset{*}{\not\rightarrow} o - node, \ for\, all\ d - node \in D_1 \right.$$
$$\left. and\ o - node \in V_{\text{TNS}} \right\}$$

For each rule $r \in R_e$ do the following steps.

S2. For each assignment statement *Ass* of *action*$_r$ and *assignment*$_r$ do the following:

If for each variable *Var* in *Ass*, there exists some $d \in D$ such that *label_name(d)* = *Var*, then delete that assignment *Ass* from that rule *r*.

S3. For each predicate *Pred* in the *guard*$_r$ and *condition*$_r$ do the following:

If for each variable *Var* in the predicate *Pred*, there exists some variable $d \in D$ such that *label_name(d)* = *Var*, then delete that predicate *Pred*.

In the first step of the above algorithm, we consider a subgraph of the data flow graph whose edges are labeled with the transition numbers of the rules of the test case under consideration. Also, in this step, we collect all the *d*-nodes (variables) denoted by the set *D*, which can neither be reached from any *i*-nodes nor reached to any *o*-nodes. This means that we are considering only those variables, which have no influence on the input/output domain of the test case. In step 2 of the algorithm, we eliminate a statement from the test case, if each and every variable appearing in that statement is in the set

D. This means that the statement is independent of the input/output domain, hence it is eliminated. In the third step, the predicates that have no influence on the input/output domain are also eliminated.

As an example, let us consider the test case $\mathbf{t_1}$ generated from Inres protocol given in section 4.4. For this test case the set $D = \{z4, z2, 0\}$ is obtained from data flow graph. The algorithm eliminates the statements $\{z2 \leftarrow s(0), z4 \leftarrow z2, z4 \leftarrow s(z4)\}$ and the predicate $[z4 < 4]$ from the test case $\mathbf{t_1}$. The result is the same as that obtained by reducing the test case dependency graph.

The major advantage of reducing the test cases using DFG over TCDG is that it is not necessary to model the test cases. Also, DFG is useful in the selection of test cases as discussed in section 4.3. A software tool [53] to display the DFG graph is available, hence it could be used to reduce the test cases. However, DFG is not useful in representation and analysis of test case, whereas TCDG is useful in the representation of test cases. This is discussed in the following chapters of the thesis.

CHAPTER V

TEST SUITE SELECTION AND REPRESENTATION

According to the Local Single-layer (LS) test architecture of ISO, any implementation under test (IUT) can be tested by a lower tester (LT) and an upper tester (UT) located at the bottom and top interfaces, respectively (see Figure 5.1) [30].

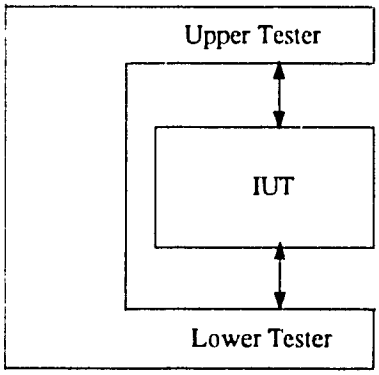


Figure 5.1 The LS test architecture

Test cases generated from the specification define the behaviour of the IUT. The behaviour of the LT and the UT considered together comprise the tester's behaviour. The tester's behaviour is the dual of the IUT's behaviour. Therefore the tester's behaviour can be obtained by behaviour inversion. Behaviour inversion is based on viewing each test case as an extended finite-state machine. Complete behaviour generation of the test case EFSMs is a complex process consisting of several steps. We have already discussed the steps of specification transformation, test case generation and test case reduction. In this chapter we discuss test suite representation and selection. First we discuss input/output data flow representation followed by test representation and selection. Finally the test cases are adapted for the RS architecture.

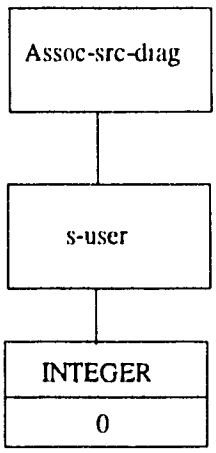
5.1 CONSTRAINT REPRESENTATION

Any constraints on the initial values of ASPs, PDUs and other substructures are defined as *base constraints*. Each test case imposes other constraints; these constraints are called *dynamic constraints*. We represent constraints with an I/O diagram.

5.1.1 Instantiated I/O Diagram

Primitive leaf nodes in the I/O diagram will contain the constraint information. The constraint information can be a constant or parametric value, a range value, e.g., > 5, a wild card such as "?" for any single value, "-" for omitting an optional field, or "*" to mean either "?" or "-". For more complicated constraints, Boolean expressions relating the field value to the variables are used.

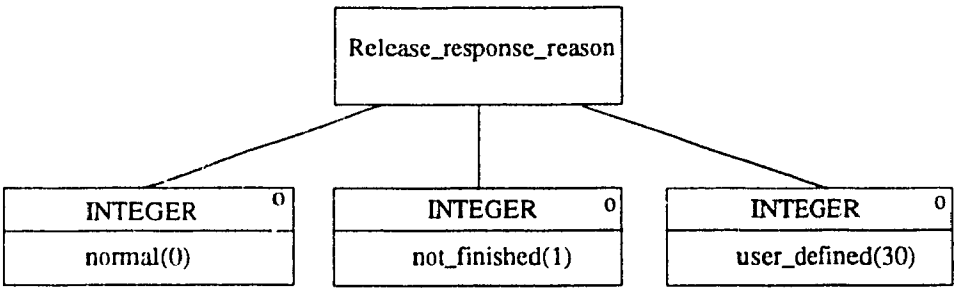
Example 5.1 Recalling the ASN.1 definition of Assoc-src-diag given in section 2.4.1 of chapter II, a constraint on the optional field *s-user*, i.e. *s-user* must be "null", can be represented as an instantiated I/O diagram *Assoc-src-diag* shown below:



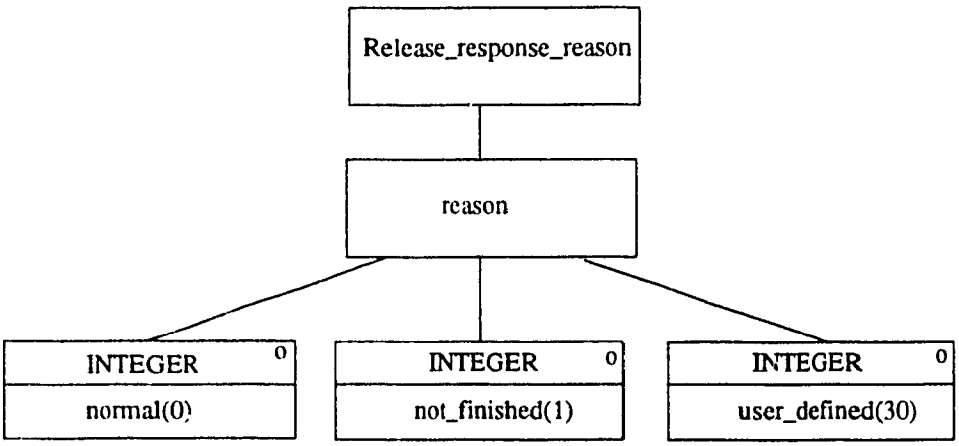
5.1.2 Base Constraints

For each ASP and PDU two base constraints are defined as instantiated I/O diagrams, one for input events and another for output events.

In this section, we shall present two algorithms to generate base constraints for input/output events. The algorithms take the I/O diagram hierarchy defined in section 2.4.3 of chapter II, as input and produce an instantiated I/O diagram as output. We assume the height of the I/O diagram is two. An I/O diagram of any height can be transformed to one or more I/O diagrams of height two. As an illustration, let us consider the following I/O diagram, whose height is one.



It can be transformed into a I/O diagram whose height is two by introducing an interior node of *composite type* of name *reason*, which is shown below:



Similarly, an I/O diagram of height more than two can be transformed into two or more I/O diagrams of height two by inserting a *nonprimitive type* leaf_node, i.e., an *extension_leaf_node*.

The algorithms we discuss in this chapter use the following operations:

1. LEFTMOST_CHILD(n, T) returns the leftmost child of node n in tree T, and it returns "null" if n is a leaf.
2. RIGHT_SIBLING(n, T) returns the right sibling of node n in tree T, defined to be that node *m* with the same parent p as n such that m lies immediately to the right of n in the ordering of the children of p.
3. DELETE_SIBLING(n, T) deletes all the siblings (including their subtrees) of the node n in tree T.
4. LABEL(n, T) returns the label of the interior node n in tree T.
5. LEAF_FIELD1(n, T) returns the value of the first field (upper part) of the leaf node n in tree T.
6. LEAF_FIELD2(n, T) returns the value of the second field (lower part) of the leaf node n in tree T.
7. DEFAULT(n, T) returns the first default_leaf child of node n found in tree T and it returns "null" if there are no default_leaf child nodes of n in tree T.
8. OPTIONAL(n, T) returns the optional_leaf child of node n in tree T or "null" if there is no optional_leaf child of node n in tree T.
9. EXTENSION(n, T) returns the extension_leaf child of node n in tree T or "null" if there is no extension_leaf child of node n in tree T.
10. ALTERNATION(n, T) returns "true" if node n is alternative to all of its siblings and returns "false" otherwise.
11. ASSIGN(FIELD2(n), V, T) assigns the value "V" to the second field of the node n in tree T.
12. CREATE_LEAF(l, v) returns a new leaf node n with label l and value v.
13. CREATEm(l, T_1, T_2, . . , T_m) returns a new node n with label l and gives it m children, which are the roots of the trees T_1, T_2, . . , T_m in order from the left.

Algorithm: Receive Event Base Constraint.

Input: The I/O diagram hierarchy.

Output: Base constraints I/O diagram for each ASP, PDU and substructures.

The procedure RECEIVE_CONSTRAINT is called for each I/O diagram T of ASP, PDU and substructure.

procedure RECEIVE_CONSTRAINT(T);

S1. Let r be the root of I/O diagram. Set q := LEFTMOST_CHILD(r, T) and i := 1;

S2. If q <> null then go to **S3** else CREATEi(LABEL(r,T)_base_R, $T_{n_{21}}, T_{n_{22}}, .., T_{n_{2i}}$) and STOP.

S3. If ALTERNATION(q,T) = "true" then DELETE_SIBLING (q,T).

S4. i. If p := DEFAULT (q, T) <> "null" then do

(a) n_i :=CREATE_LEAF(LEAF_FIELD1 (p, T), LEAF_FIELD2 (p, T));

(b) n_{2i} := CREATE1(LABEL (q, T),T_{n_i});

(c) i := i+1, go to **S5**

ii. If p := OPTIONAL (q,T) <> "null" then do

(a) n_i :=CREATE_LEAF(FIELD_LEAF1 (p,T), '*');

(b) n_{2i} := CREATE1(LABEL (q, T), T_{n_i});

(c) i := i+1, go to **S5.**

iii. If p := EXTENSION (q, T) <>"null" then do

(a) n_i :=CREATE_LEAF(FIELD_LEAF1 (p,T), FIELD_LEAF2 (p, T));

(b) n_{2i} := CREATE1(LABEL (q, T), T_{n_i});

(c) i :=i +1, go to **S5.**

iv. Otherwise set p := LEFTMOST_CHILD(q,T) and do

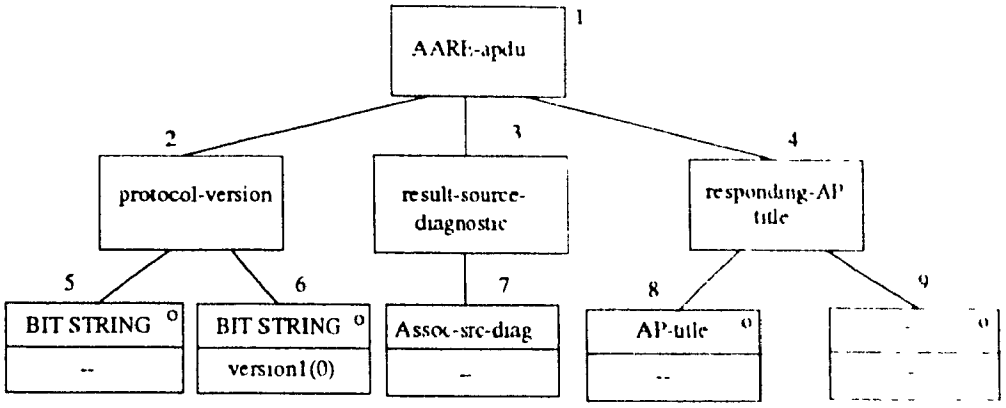
(a) n_i := CREATE_LEAF(LEAF_FIELD1(p,T), '?');

(b) n_{2i} := CREATE(LABEL(q, T), T_{n_i});

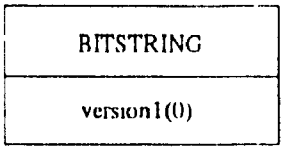
(c) i :=i +1, go to **S5.**

S5. Set q := RIGHT_SIBLING (q, T); go to **S2**

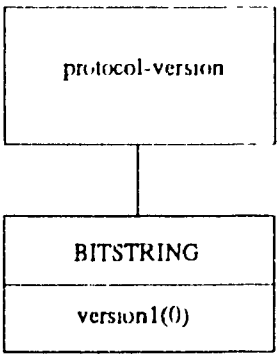
Example 5.2 As an illustration, we apply the algorithm to the I/O diagram of AARE-apdu shown below. For convenience, we numbered all the nodes uniquely.



The root node of the I/O diagram is *AARE-apdu*, i.e., node 1. In step 1 of the algorithm q is set equal to node 2 and *i* is initialized to 1. Now the interior node 2 is examined, which is of course not *null*. Step 3 is skipped because node 2 is not alternative to all of its siblings. In step 4, DEFAULT(2,T) returns node 6, which is a *default leaf* child of node 2. Then a leaf node n_1 is created in substep (a), which is shown below:

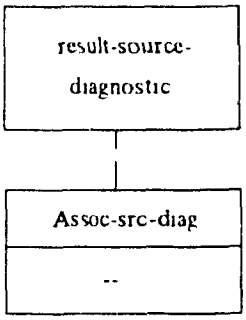


In substep (b) a tree with the root node n_{21} of label *protocol-version* is created, which is as follows:

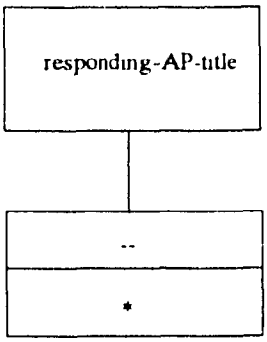


In substep (c) *i* is incremented by 1 and the control is transferred to step 5. In this step q is set to RIGHT_SIBLING(2,T), which is node 3 and the control is transferred to step 2.

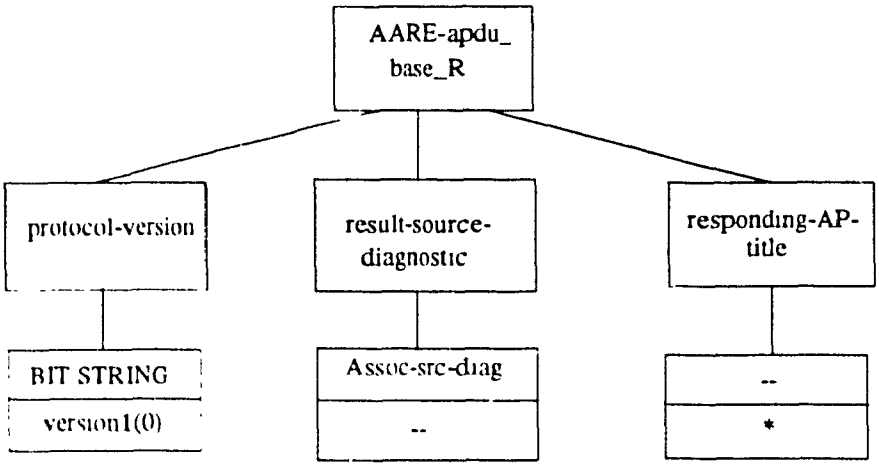
. In the next iteration a tree with the root node n_{22} is created as is shown below:



In the third iteration a tree with root node n_{23} with label *responding-AP-title* is created as shown below:

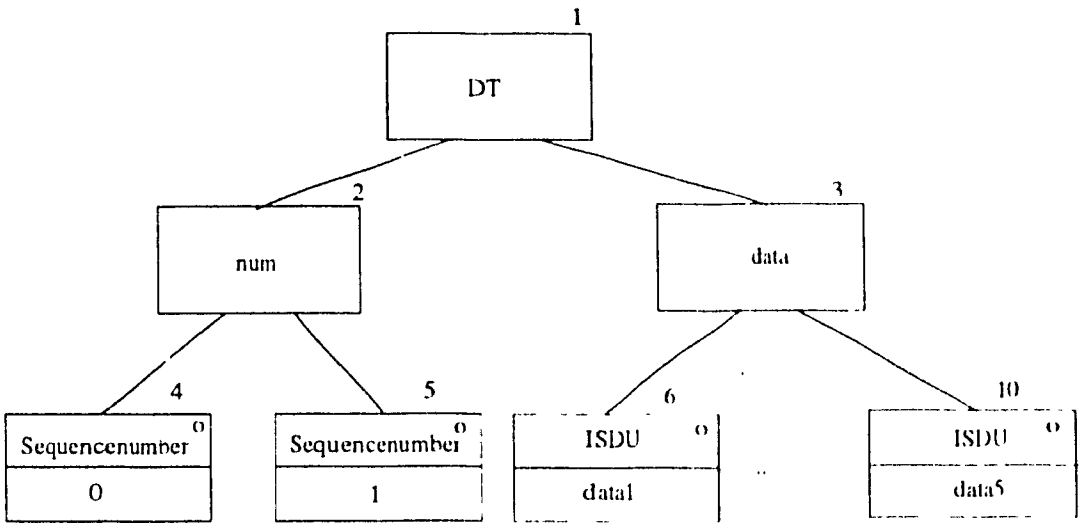


Finally, in step 2 CREATE3(AARE-apdu_base_R, $T_{n_{21}}, T_{n_{22}}, T_{n_{23}}$) is formed, which is nothing but the following I/O diagram:

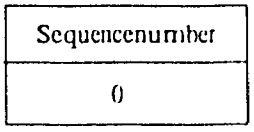


For the substructure *Ass-src-diag*, the constraint generated by applying the algorithm is given in example 5.1.

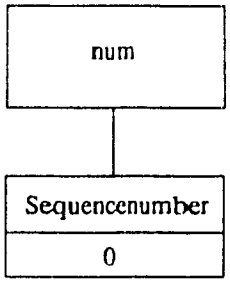
Example 5.3 Our second example is from DT pdu of Inres protocol. The I/O diagram for DT pdu, which is defined in section 2.4.2, is shown below:



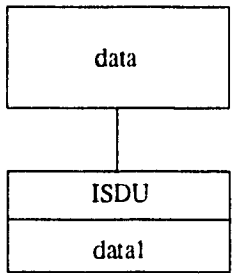
Once again, for convenience, we numbered all the nodes uniquely. The root node of the I/O diagram is DT, i.e., node 1. In step 1 of the algorithm q is set equal to node 2 and *i* is initialized to 1. Now, the interior node 2 is examined, which is of course not null. Step 3 is skipped because node 2 is not alternative to all of its siblings. In step 4, DEFAULT(2,T) returns node 4, which is a *default_leaf* child of node 2. Then a leaf node n_1 is created in substep (a) as shown below:



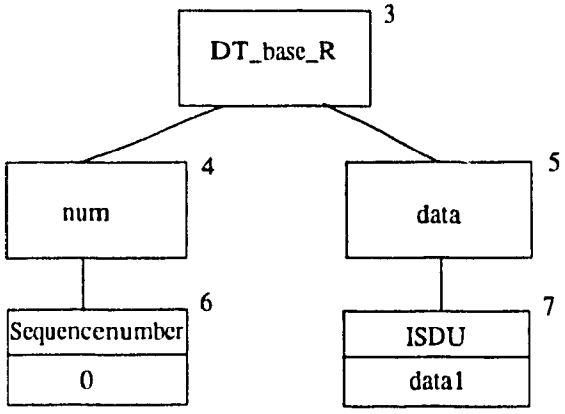
In substep (b) a tree with the root node n_{21} with label *num* is created, which is as follows:



In substep (c) *i* is incremented by 1 and the control is transferred to step 5. In this step q is set RIGHT_SIBLING(2,T), which is node 3. In the next iteration a tree with the root node n_{22} is created as shown below:



In the final iteration, CREATE2(DT_base_R, T_{n21}, T_{n22}) is formed, as shown below:



Algorithm: Send Event Base Constraint.

Input: The I/O diagram hierarchy of the specification

Output: Base constraints I/O diagram for each ASP, PDU and substructures.

The procedure SEND_CONSTRAINT is called for each I/O diagram T of ASP, PDU, substructures.

Procedure SEND_CONSTRAINT(T);

The same as RECEIVE _CONSTRAINT except for the optional_leaf_node the wild card '–'(OMIT) is generated instead of '*'(ANY_OR_OMIT) since stricter constraints must be imposed on the parameter values of send events.

5.1.3 Dynamic Constraints

In the following we outline procedures to generate the instantiated I/O diagram representing dynamic constraints on the parameter values of each SEND/RECEIVE event of each test case.

Algorithm: Receive Event Dynamic Constraints.

Input: RTCDG and Receive Event Base Constraints (REBC)

Output: Receive event dynamic constraints.

The base constraints I/O diagrams are modified by changing the value of the leaf nodes based on the send event structure in the RTCDG. We assume the structures of the send event are of the form: ASP(field_1, field_2, ... , field_n), where ASP is the name of the send event, and field_i stands for the term assigned to the corresponding field identifier d_i of the ASP.

For each send a-node in RTCDG do

S1. Let T be the receive base constraint for the ASP associated with a send a-node and Struc be the send event structure.

S2. RECEIVE_CONSTR_DYNAMIC(Struc, T, REBC);

procedure RECEIVE_CONSTR_DYNAMIC(SP, T, REBC);

S1. Let r be the root of the I/O diagram. Set q:= LEFTMOST_CHILD(r, T).

S2. for each field_i of SP do

i. If field_i is of PDU or substructure type then do

(a) Let T_b be the I/O diagram of the PDU or substructure receive base constraint.

(b) RECEIVE_CONSTR_DYNAMIC(field_i, T_b, REBC)

ii. If field_i is of the form op(Not_present) then set q:= RIGHT_SIBLING(q, T).

iii. Otherwise do

(a) ASSIGN(FIELD2(LEFTMOST_CHILD(q,T)), field_i, T)

(b) Set q:= RIGHT_SIBLING(q,T).

As an illustration, let us consider RTCDG(Figure 4.6) of test case $\mathbf{t_1}$ generated from Inres protocol. Let the SEND event a-node be 14.1, i.e., the event

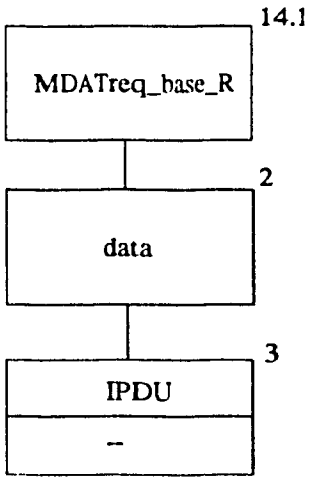
MSAP!MDATreq(ipdu9)

The event structure "Struc" can be rewritten in the form of

MDATreq(DT(number7, data(sp7))

because the variable ipdu9 is assigned the value DT(number7,data(sp7)) in the s-node 13.7.

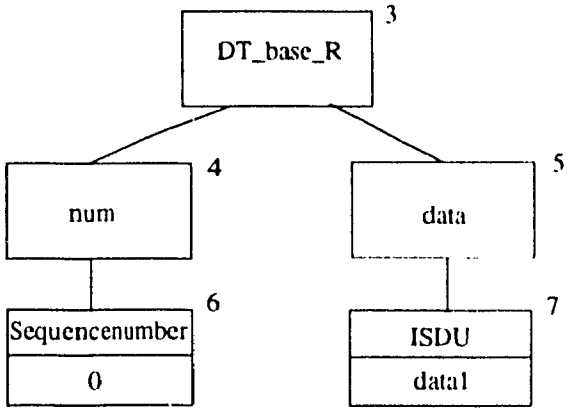
The base constraint of MDATreq in the form of an I/O diagram is shown below:



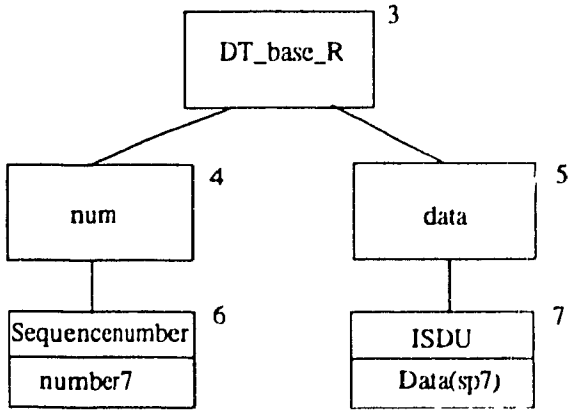
The REBC is the set of all Receive Event Base Constraints obtained for this protocol. Again, for convenience, we numbered the root node of I/O diagram as *a*-node number

and all other nodes of the I/O diagram with a unique positive integer. Note that the node 3 is an *extension_leaf_node*.

The step 1 of the procedure RECEIVE_CONSTR_DYNAMIC q is set equal to 2. In step 2 the field_1 of MDATreq which is DT(number7, data(sp7)) is checked for PDU type. The field_1 is of course DT pdu, for which the receive event base constraint is given below, in form of I/O diagram T_b:



Note that, we numbered the root node of the I/O diagram as the same number as that of *extension_leaf_node* and all other nodes of the I/O diagram with a unique positive integer. In substep (b) of step 2, the recursive procedure RECEIVE_CONSTR_DYNAMIC' is called again with "Struc" = DT(number7, Data(sp7)), and T = T_b. The receive dynamic constraint generated by the procedure for DT pdu is given below:



Algorithm: Send Event Dynamic Constraints.

Input: RTCDG, Send Event Base Constraints (SEBC)

Output: Send event dynamic constraints.

In generation of send event dynamic constraint, SEBC are used. The base constraint I/O diagrams are modified by changing the value of the leaf nodes based on the predicates in p-nodes of RTCDG. We assume that if the predicate p_i involves field_identifier d_i of the ASP/PDU, then it is either of the form of d_i op E_i or it can be transformed to that form, where op is an operator of the form $\leq, \geq, <, >, \neq, =$, and E_i is a term.

For each receive a-node in RTCDG do

S1. Let $\mathcal{T}$ be the set of all I/O diagrams corresponding to the send base constraint for the ASP (including the base constraints used by the ASP through the *extension_leaf_node*).

S2. Let Pred be the set of all predicates in conjunctive normal form p_1 and p_2 ... and p_n of the p-nodes that are data dependent on the receive a-node.

S3. For each $p_i \in$ Pred do

for each T$\in \mathcal{T}$ do SEND_CONSTR_DYNAMIC(T, p_i)

procedure SEND_CONSTR_DYNAMIC(T, p_i);

S1. Let r be the root of the I/O diagram. Set q:= LEFTMOST_CHILD(r, T).

S2. If q $\neq$"null" go to **S3**; else STOP.

S3. If p_i involves LABEL(q, T) then go to **S4**; else go to **S5**.

S4. (i) Transform the predicate pi into the form of LABEL(q, T) op E.

(ii) ASSIGN(FIELD2(LEFTMOST_CHILD(q, T)), "op E", T); go to **S5**.

S5. SET q:= RIGHT_SIBLING(q, T); go to **S2**.

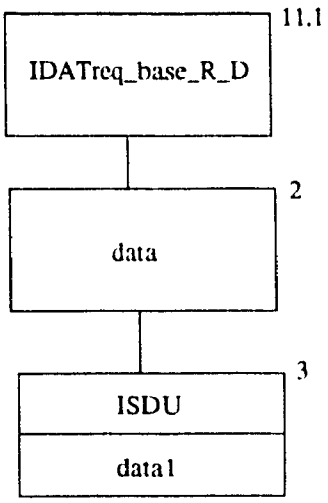
As an illustration, let us consider the RTCDG(Figure 4.6) of test case $\mathbf{t_1}$ of Inres protocol. Let the receive event a-node be 11.1, i.e., the event

ISAP ? sp7:SP

The p-node 12.5 has the dependency on this a-node and it reads as

[isIDATreq(sp7)]

This dependency does not impose any constraint on the data field of the service primitive IDATreq, so the algorithm returns the base constraint as the dynamic send event constraint for *IDATreq*:



More illustrative examples on generation of dynamic constraints for realistic protocol such as ACSE and HDLC are given in application chapter 6.

5.2 CONTROL FLOW BEHAVIOUR REPRESENTATION

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. Graphical representation of the control flow can be directly obtained from RTCDG by simply dropping the p-nodes and the associated s-nodes with no incoming data dependency edge. Except for i_s a-nodes, all other a-nodes are **inverted**, which means that interactions are inverted. For example the event P!ABRT_apdu can be inverted to P?ABRT_apdu.

Algorithm: Control Flow Behaviour Representation.

Input: Reduced Test Case Dependency Graph RTCDG = (V, E).

Output: Control Flow Behaviour Representation CFBR = (V′, E′).

The main algorithm eliminates all the p-nodes. The interactions are inverted. The procedure SEARCH is the same as was defined in the reduced test case dependency graph algorithm. We assume that the action v is in the form of gd, where d = !*t* or ?*v:s*.

1. V′:=V;
2. E′:=E;
3. for each $v \in V_p \subset V$ do
   begin
4. V′:=V′–v;
5. Let E_{new} := { (u,w) | w ∈ N_f [v] and u ∈ N_t [v]};
6. E′ := { E′– {D_f [v] ∪ C_f [v] ∪ C_t [v] }} ∪ E_{new};
   end
7. SEARCH(V′ ,E′);
8. For each action $v \in V_a' \subset V'$ do
   begin
9. If v is in the form of g!*t* then change it to g?*v:s* ;
10. else if v is in the form of g?*v:s* then change it to g!*t*;

   end

5.3 TEST CASE HIERARCHY

The international standard for conformance testing methodology [30–31] gives a framework for testing protocol implementations for conformance to the standards. The methodology requires the development of a standardized collection of tests called a test suite.

The test suites are hierarchically structured. The first level of the hierarchy consists of four major test groups: Basic Interconnection tests, Capability tests, Behaviour tests and Conformance Resolution tests. Behaviour testing is the main part of a test suite and has three subdivisions: valid behaviour testing, which aims to establish that valid

behaviour of the protocol can be exercised correctly; invalid behaviour testing, which confirms that invalid behaviour is handled properly as specified in the protocol standard; inopportune behaviour testing, which checks how unexpected or inopportune behaviour is handled. Each of these groups are further divided into a smaller number of lower level test groups. Figure 5.2 [46] is an example of a suitable structure for a single-layer test suite. The first level in test suite hierarchy is protocol independent but subsequent levels are protocol dependent. Also it is not known which hierarchy will be the best test suite for a given protocol. The test selection step assumes that the test suite hierarchy is externally decided. Test suite hierarchy also decides the test purposes, which are discussed next.

5.4 TEST PURPOSES

Once the test suite hierarchy is constructed, a naming strategy is developed to name the nodes of the tree. A node in the tree can be identified by the name of the path from the root to the node. At any node, the subtree rooted at that node can be considered as a test subgroup. Also, associated with every node there is a test subgroup objective, which conveys what the test subtree rooted at that node is designed to achieve. The leaves are the test purposes. Objectives and test purposes are presently informally specified [46].

5.5 VALID BEHAVIOUR TEST SELECTION

Once the test suite structure and test purposes are determined for each test purpose, one or more test cases must be selected from the represented test cases. If the specification defines only valid behaviour, then all the test cases belong to valid behaviour test hierarchy.

5.5.1 Behaviour Enhancements and Verdict Assignment

Control flow behaviour representation (CFBR) of the selected test cases is analyzed and several enhancements are carried out. In the following, we outline an algorithm to enhance CFBR of the selected valid behaviour test case.

A. Capability tests

A.1 Mandatory features
A.2 Optional features

B. Behaviour tests: response to valid behaviour by peer

B.1 Connection establishment phase (if relevant)

B.1.1 Focus on what is sent to the IUT

B.1.1.1 Test event variation in each state
B.1.1.2 Timing/timer variation
B.1.1.3 Encoding variation
B.1.1.4 Individual parameter value variation
B.1.1.5 Combination of parameter values

B.1.2 Focus on what is received from IUT
— substructured as B.1.1

B.1.3 Focus on interactions
— substructured as B 1 1

B.2 Data transfer phase
—substructured as B 1

B.3 Connection release phase (if relevant)
—substructured as B.1

C. Behaviour tests: response to syntactically invalid behaviour by peer

C.1 Connection establishment phase (if relevant)

C.1.1 Focus on what is sent to the IUT

C.1.1.1 Test event variation in each state
C.1.1.2 Encoding variation of the invalid event
C.1.1.3 Individual invalid parameter value variation
C.1.1.4 Invalid parameter value combination variation

C.1.2 Focus on what the IUT is requested to send

C.1.2.1 Individual invalid parameter values
C.1.1.2 Invalid combinations of parameter value

C.2 Data transfer phase
— substructured as C.1

C.3 Connection release phase (if relevant)
— substructured as C.1

D. Behaviour tests: response to inopportune events by peer

D.1 Connection establishment phase (if relevant)

D.1.1 Focus on what is sent to the IUT

D.1.1.1 Test event variation in each state
D.1.1.2 Timing/timer variation
D.1.1.3 Special encoding variations
D.1.1.4 Major individual parameter value variation
D.1.1.5 Variation in major combination of parameter values

D.1.2 Focus on what is requested to send by the IUT
—substructured as D.1.1

D.2 Data transfer phase
— substructured as D.1

D.3 Connection release phase (if relevant)
— substructured as D.1

Figure 5.2 Suitable structure for a single-layer test suite.

Algorithm: Behaviour Enhancement.

Input: Control Flow Behaviour Representation(CFBR).

Output: Enhanced CFBR.

All the i_s a-nodes that are alternative to a send a-node are removed due to controllability of the tester. A new type of receive a-node (?OTHERWISE) is created as an alternative to all receive a-nodes to specify tester's behaviour against invalid IUT behaviour. Verdicts are assigned to the receive and OTHERWISE a-nodes.

S1. For each i_s a-node A1 do

Find the first a-node A2 in the alternative path;

If A2 is a send a-node then delete the edge incident on A1.

S2. Repeat (i) until there is no node (except the initial node) without an edge incident on it.

(i) delete the node and all the outgoing edges.

S3. For each receive a-node do

add an alternative path which contains an arc and a receive a-node of type OTHERWISE. No other arcs are added to this path, i.e.,

OTHERWISE a-nodes can only be at the final states.

S4. For each OTHERWISE a-node O1 do

verdict(O1) := fail;

For each receive a-node R1 do

if R1 is the last event in the path and the path leads to the initial state do

verdict(R1) := pass;

S5. For each i_s a-node A1 do

for each receive a-node R2 following A1 do

if R2 is the final node or predecessor of the final node then

verdict(R2) := inconclusive

The time complexity of the algorithm given above in the worst case is $O(|V|^2)$, where $|V|$ is the number of nodes in the control flow behaviour representation graph.

5.5.2 Test Purposes and Parameter Values

Sometimes it is necessary to modify the enhanced CFBR to satisfy the test purpose. In the following we outline a heuristic to modify the enhanced CFBR.

Procedure:

Input: Enhanced CFBR.

Output: Modified Enhanced CFBR.

S1. Directed circuits in the enhanced CFBR are modified to reflect the test purpose. If the loop is needed only once then delete the arc that creates the circuit.

S2. If the loop needs to be executed more than once then it is expanded. In this case parameter values of send and receive a-nodes in the expanded CFBR must be varied to try sending the same events with different parameter values and verifying the responses from the receive event parameters.

S3. If the test purpose is achieved with a receive a-node for which no verdict is assigned, assign a pass verdict to this a-node.

5.6 ADAPTATION OF GENERATED TEST CASE FOR RS ARCHITECTURE

So far we have discussed the generation of test case for Local Single-layer(LS) architecture. In this section, we shall develop an algorithm to transform the generated test case to a form which is suitable for Remote Single-layer(RS) architecture. The significant features of RS model are that no interface at the top of the IUT is assumed and no explicit test coordination procedures are assumed, as shown in Figure 5.3.

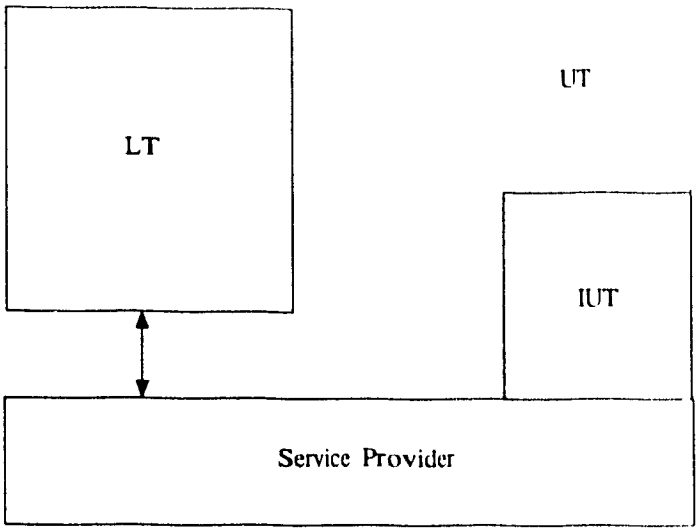


Figure 5.3 The RS test architecture

It may seem strange to test an initiator IUT by use of the remote test method and thus omitting service interface checks since no upper point of observation is used (control of the IUT is achieved by use of implicit send). However, in the case of embedded testing this is acceptable since the service interface is tested implicitly by the supported application. The implicit sends (to control the IUT) can be specified at any level and it is up to the IUT provider to specify in the Protocol Implementation Extra Information for Testing (PIXIT) how those events can be triggered.

Our starting point is the RTCDG obtained from the generated test case. The RTCDG is transformed to a form which is suitable for RS architecture. The basic ideas behind the transformation are to suppress the service provided by the IUT and to specify the implicit events. The suppression of the service provided by the IUT can be achieved by the elimination of send a-nodes that are occurring at the upper interface. The implicit event can be specified by replacing the interface name to "IUT" of the receive a-node that are occurring at the upper interface. In the above process, we may have to eliminate some s-nodes that cannot affect the parameters of the send a-nodes.

Algorithm: Remote Single–layer(RS) Reduced Test Case Dependency Graph.

Input: Reduced Test Case Dependency Graph RTCDG = (V, E).

Output: RS Reduced Test Case Dependency Graph $RTCDG_{RS}$ = (V′, E′).

The recursive procedure SEARCH eliminates s-nodes that have no incident data dependency edge. We assume that the operation CHANGE(name(*a*), "IUT") is available. CHANGE(name(*a*), "IUT") replaces the gate associated with the action "*a*" with "IUT".

1. V′ := V;
2. E′ := E;
3. Let I_a = { v ∈ V | name(v) = upper_gate }
4. For each v ∈ I_a do
   begin
5. If *offer*(v) = ! then
   begin
6. V′ := V′ − v;
7. Let E_{new} := { (u,w) | w ∈ N_f[v] ∧ u ∈ N_t[v] }
8. E′ := { E′ − { D_f[v] ∪ C_f[v] ∪ C_t[v] } } ∪ E_{new}
   end
9. else CHANGE(name(a), "IUT");
   end
10. SEARCH(V′, E′):

procedure SEARCH(V′, E′);

1. Let VD = { v | D_t[v] = ϕ and v ∈ V'_s ⊂ V′ };
2. If VD <> Φ then
   begin
3. Let v be any element in VD;
4. V′ := V′ − v;
5. Let E_{new} := { (u,w) | w ∈ N_f[v] and u ∈ N_t[v] };
6. E′ := { E′ − { D_f[v] ∪ C_f[v] ∪ C_t[v] } } ∪ E_{new} ;
7. SEARCH(V′ , E′);
   end;

The RS RTCDG can be used as a basis to represent the test case. The control flow can be directly obtained from the RS RTCDG by dropping the p-nodes and the associated s-nodes with no incoming data dependency edge. Similarly the dynamic constraints can be generated as explained in section 5.1.3. Also verdicts can be assigned as explained in section 5.5.1.

5.7 COMPARISON BETWEEN TEST SELECTION STRATEGIES

Two types of test selection strategies are discussed in this thesis:

1. Selection of test cases generated from the specification to meet certain coverage criteria.
2. Selection of test cases that satisfy the test purpose according to the hierarchy proposed by ISO[46].

In the first type of test selection method, the test designer fixes the criterion. This criterion may be branch testing, node testing or a particular function of the protocol. Once the criterion is fixed, i.e., a particular function of the protocol is identified from the data flow graph, we select a subset of the generated test cases by using zero-one integer programming method, which adequately exercises the protocol function. In the second type, the selected test case is categorized according to the test case hierarchy. In other words, the selected test cases (for a particular function) are divided into different subgroups that satisfy the subgroup objective in the test suite hierarchy. Also, the test case of the subgroup that satisfies a particular test purpose within that subgroup is identified.

In summary, the first type of selection method is to select a subset of the generated test cases that satisfies certain data flow functions, whereas the second type is to divide those selected test cases that satisfy certain coverage criteria, into different subgroups according to the test suite hierarchy designed externally.

CHAPTER VI

APPLICATIONS

In this chapter we apply the test suite design methodology to the ACSE and LAPB protocols specified in LOTOS and SDL language respectively. We also describe LOTEST, a computer-aided software tool that implements our test suite design methodology for a LOTOS specification.

6.1 ACSE SPECIFICATION IN LOTOS

An Application Entity (AE) may be modeled as a set of building blocks, each providing a well defined functionality. These building blocks are called Application Service Elements (ASEs). Each ASE co-operates with its peer by using a specific protocol. The AE chooses the ASEs needed to provide the type of communication required by the application protocol user. The Association Control Service Element (ACSE) is a special kind of AE that is used by other ASEs to open and release Presentation Layer connections between associated AEs.

The formal specification of the ACSE protocol [27] is written in the LOTOS specification language. The specification consists of two major parts. The first part describes abstract data types and structures used by ACSE (i.e., parameter types, PDUs and service primitives). The behaviour of ACSE is defined in the second part. The behaviour of the ACSE protocol in LOTOS is based on the following three procedures:

- *association establishment:*

The association establishment procedure is used to establish an association between two AEs. It supports the A-ASSOCIATE service. The association establishment procedure uses the A-ASSOCIATION-REQUEST(AARQ) and the A_ASSOCIATION-RESPONSE(AARE) APDUs.

- *normal release of an association establishment:*

The normal release procedure is used for the release of an association by an AE without loss of information in transit. It supports the A-RELEASE service. The normal release procedure uses the A-RELEASE-REQUEST(RLRQ) APDU and the A-RELEASE-RESPONSE(RLRE) APDU.

- *abnormal release of an association:*

The abnormal release procedure can be used at any time to force the abrupt release of the association by a requester in either AE, or by the presentation service provider. The abnormal release procedure supports the A-ABORT and A-P-ABORT services. The abnormal release procedure uses the A-ABORT(ABRT) APDU. Note that no PDUs are defined for the A-P-ABORT service since it is directly mapped from the P-PABORT service.

As an illustration, we take a very simple LOTOS process of the ACSE protocol to specify abnormal release of an association:

```
process abort[A,P]: noexit:=
   A?x:primitive[IsAABRreq(x)];
   P!ABRT_apdu(acse_service_provider,type023(Not_present));
   unassociated[A,P]
 []
   P?x:ACSE_apdu[IsABRT(x)];
   A!AABRind(acse_service_provider,
                          type_genere020(Not_present));
   unassociated[A,P]
  endproc (*abort*)
```

The LOTOS specification of the ACSE protocol is a mixture of resource and state oriented styles. It includes all the options specified in the standard for a total of some 2297 lines of LOTOS code, out of which 88% is about abstract data types.

6.2 THE LOTEST SYSTEM

LOTEST [53] is a tool for designing test cases from LOTOS specifications. The backbone of the LOTEST environment is a formal notation called chart which has: mathematically precise semantics, a formal algebraic product for the composition of two charts and simple representation of data flow. LOTEST has been developed on a SUN workstation using standard UNIX tools (LEX, YACC and the C compiler) and Prolog. The attractiveness of LOTEST lies in the user's ability to design tests interactively, with access to several helpful tools that implement the methods of compilation, generation of test cases, and identification of protocol functions. In this respect, two features of LOTEST are significant. First, the chart constructed from the LOTOS specification facilitates the generation of test cases as well as the construction of data flow graphs. Second, the menu selection facility relieves a test designer from having to remember all the commands and reduces the number the key strokes he or she has to enter for interaction with LOTEST. When the LOTEST is invoked, it displays the following menu:

```
TEST SEQUENCE GENERATION FROM LOTOS

ENTER 1 or l TO LIST LOTOS SPECIFICATION
ENTER 2 or c TO COMPILE A LOTOS SPECIFICATION ".l"
ENTER 3 or p TO LIST COMPILED LOTOS SPECIFICATION ".pl"
ENTER 4 or n TO GENERATE CHART FROM LOTOS SPEC. ".pl"
ENTER 5 or z TO LIST CONTROL GRAPH ".CTRL"
ENTER 6 or d TO GENERATE INPUT FOR CONTROL FLOW GRAPH ".ECTRL"
ENTER 7 or g TO LIST CONTROL GRAPHS FROM SPECIFICATION ".ECTRL"
ENTER 8 or f TO GENERATE INPUT FOR DATA FLOW GRAPH ".dfg"
ENTER 10 or d TO GENERATE DEADLOCK STATE FROM CTRL GRAPH ".lock"
ENTER 11 or t TO GENEARE TEST CASES FROM CTRL GRAPH ".test"
ENTER 12 or s TO GENERATE CHART FOR EDITTEST ".LIST"
ENTER pwd TO SHOW CURRENT DIRECTORY
ENTER m TO SHOW THIS MENU
ENTER e or exit TO QUIT FROM THIS MENU

CHOOSE ONE COMMAND .....

==>
```

Figure 6.1 shows the global structure of the tool. This section gives an overview of the three components.

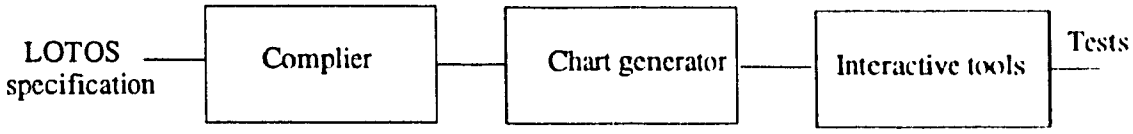


Figure 6.1 Global structure of LOTEST.

6.2.1 Compiler

The first step applied to the LOTOS specification is compilation. Here we adopt the compiler developed at the University of Ottawa as a part of LOTOS interpreter isla [40]. The compiler does lexical, syntactic and semantic analysis. If the specification is found to be correct, it is translated into an *internal* format, which represents the specification in the form of a Prolog list in a *flattened* name space.

6.2.2 Chart generator

After compilation the chart generator is activated, which translates the intermediate form of the specification in Prolog form into a chart by bottom-up synthesis. In the first stage, expansion of the LOTOS processes and renaming of variables are done, and then the chart is constructed. In the second stage, it prepares input for the interactive tools to display the data and control flow graphs. Finally, it generates test cases.

The chart, input for data flow graph, test cases, etc., can be generated interactively. In the generation of the chart, the process recursion can be resolved as soon as it encounters the process identifier or it can be resolved after all the LOTOS constructs in the specification are translated into the chart. The test designer can choose which recursion can be resolved right away by answering 'yes' or 'no' to the query when the recursion is encountered in the formation of the chart. The recursions that are not resolved at the time of construction of the chart are automatically resolved at the end of the chart construction.

The chart generator tool provides some facilities for locating specification errors. For example, detection of *deadlock* in the specification. Deadlocks can occur in a specification if different gates are used in a given interaction. We present in the following a simple example to demonstrate the deadlocks. Let us consider the following two processes:

```
Process     S1[a,b]: noexit :=
              a !succ(0); a; stop
endproc
Process     S2[a,b]: noexit :=
              a !succ(0); b; stop
endproc
```

When the two processes **S1** and **S2** (**S1||S2**) are composed, a deadlock occurs. In this example, it is intuitively clear that there is a deadlock in the specification. However, in more complicated situations it is not easy to locate these kinds of errors.

The chart generator is written in Prolog, and the current version is made up of approximately 4000 Prolog clauses.

6.2.3 Interactive tools

There are four interactive tools: ctool, dfgtool, edittest and testgen. The chart is displayed in the form of a finite state machine by ctool using Sun workstation graphics. After displaying the chart, ctool becomes a menu-driven interactive tool. The test designer can move the screen left, right, up and down as well as move the states anywhere on the screen by means of the mouse. The graph can be saved/loaded any time by a save/load command in the menu.

The dfgtool displays the data flow graph with automatic blocking and offers several facilities for block merging. The dfgtool, like ctool, has a menu-driven interactive user interface. Edittest displays the test case in one of the windows, the rules that occur in the test case in a text window, and the chart in another window. Edittest is designed to help the test designer to interactively go through the test cases and identify infeasible test

cases. The testgen tool gets the test cases from the edittest output and uses the transition numbers of each function from the data flow graph to generate full coverage of each of the data flow functions. All these interactive tools are derived from CONTEST_ESTL [54] by software reusability.

6.3 TEST SUITE DESIGN FROM ACSE PROTOCOL

The first step of test suite design is the construction of the EFSM chart. An EFSM chart was automatically constructed from the LOTOS specification of ACSE protocol using the LOTEST[53] tool. The resulting EFSM chart has 113 states and 169 transitions, and is given in appendix A.

6.3.1 GENERATION AND SELECTION OF TEST CASES

The next step is to generate test cases and to draw the data flow graph. As mentioned earlier, the total number of test cases is determined by the formula d_{in}(initial state)+d'_{in}(initial state), where d_{in}(initial state) is the indegree of the initial state and d'_{in}(initial state) is the number of edges incoming to the initial state added during the conversion of EFSM chart to Euler graph. Applying the above formula to the ACSE EFSM chart, d_{in} is 10 and d'_{in} is 34, therefore yielding 44 test cases. The test cases generated from ACSE protocol are given in appendix B. One complete test case, t_{27} is listed below:

Test case t_{27}

<A?x29, 256, 121, 1, true, [IsAASCreq(x29)], ϵ, ϵ >,
< P!ACSE_apdu(ACSE_apdu_genere_0(AARQ_apdu(BIT(1),
app_context_name(get_AASCreq(x29)), called_ap_title(get_AASCreq(x29)),
called_ae_qualifier(get_AASCreq(x29)), called_ap_invocation_id(get_AASCreq(x29)),
called_ae_invocation_id(get_AASCreq(x29)), calling_ap_title(get_AASCreq(x29)), calling_ae_qualifier(get_AASCreq(x29)), calling_ap_invocation_id(get_AASCreq(x29)),

calling_ae_invocation_id(get_AASCreq(x29)), type_genere010(Not_Present),
user_info(get_AASCreq(x29))))):ACSE_apdu, 121, 120, 4, true, true, ϵ, ϵ>,
< P?x14:ACSE_apdu, 120, 109, 9, true, [IsAARE(x14)], ϵ, ϵ >,
< i, 109, 103, 14, [eq(result(get_AARE(x14)),accepted)],true, ϵ, ϵ >,
< A!primitive(AASCcnf(application_context_name(get_AARE(x14)), respond-
ing_AP_title(get_AARE(x14)), responding_AE_qualifier(get_AARE(x14)), re-
sponding_AP_invocation_id(get_AARE(x14)), responding_AE_invocation_id
(get_AARE(x14)), user_information(get_AARE(x14)), result(get_AARE(x14)),
acse_service_user, optional(result_source_diagnostic(get_AARE(x14))),
empty_presentation_parms_set)): primitive, 103, 102, 22, true, true, ϵ, c12 $\leftarrow$ calling>,
< A?x12:primitive, 102, 90, 30, true, [IsARLSreq(x12)], ϵ, ϵ >,
<P!ACSE_apdu(ACSE_apdu_genere_2(RLRQ_apdu(reason(get_ARLSreq(x12)),
user_info(get_ARLSreq(x12))))):ACSE_apdu), 90, 89, 40, true, true, ϵ, c10 $\leftarrow$ c12 >,
< P?x10:ACSE_apdu, 89, 77, 53, true, [IsRLRQ(x10)], ϵ, ϵ >,
< A!primitive(ARLSind(ARLSind(reason(get_RLRQ(x10)), user_information
(get_RLRQ(x10)))) :primitive, 77, 76, 66, true, true, ϵ, ϵ >,
< i, 76, 58, 83, [eq(c10,called)], true, ϵ, ϵ>,
< A?x5:primitive, 58, 51, 97, true, [IsAABRreq(x5)], ϵ, ϵ >,
< P!ABRT_apdu(acse_service_user, type_genere023(Not_Present)):ABRT_apdu, 51, 50,
113, true, true, ϵ, ϵ >
< i_r, 50, 256, 129, true, true, ϵ, ϵ >

Part of the ACSE data flow graph produced by dfgtool of LOTEST is shown in Figure 6.2. A data flow graph is designed to visualize the flow of data. However, guard and selection predicates are not taken into consideration in the DFG to avoid cluttering of the graph. Generation of data flow graph is only possible when PDUs and ASPs are explicitly identified. The structures of the PDU and the ASP must be provided by the

user. Data flow dependencies between graphs of the ACSE protocol function are said to occur when the function has incoming arc(s) from another function. Another interesting property of the data flow graph is that block merging eliminates data flow dependencies among the data flow functions. It seems that in application layer protocols, context variables are seldom utilized. In other words, once appropriate parameter values are chosen for an input interaction, the expected parameter values of the output interaction can easily be determined.

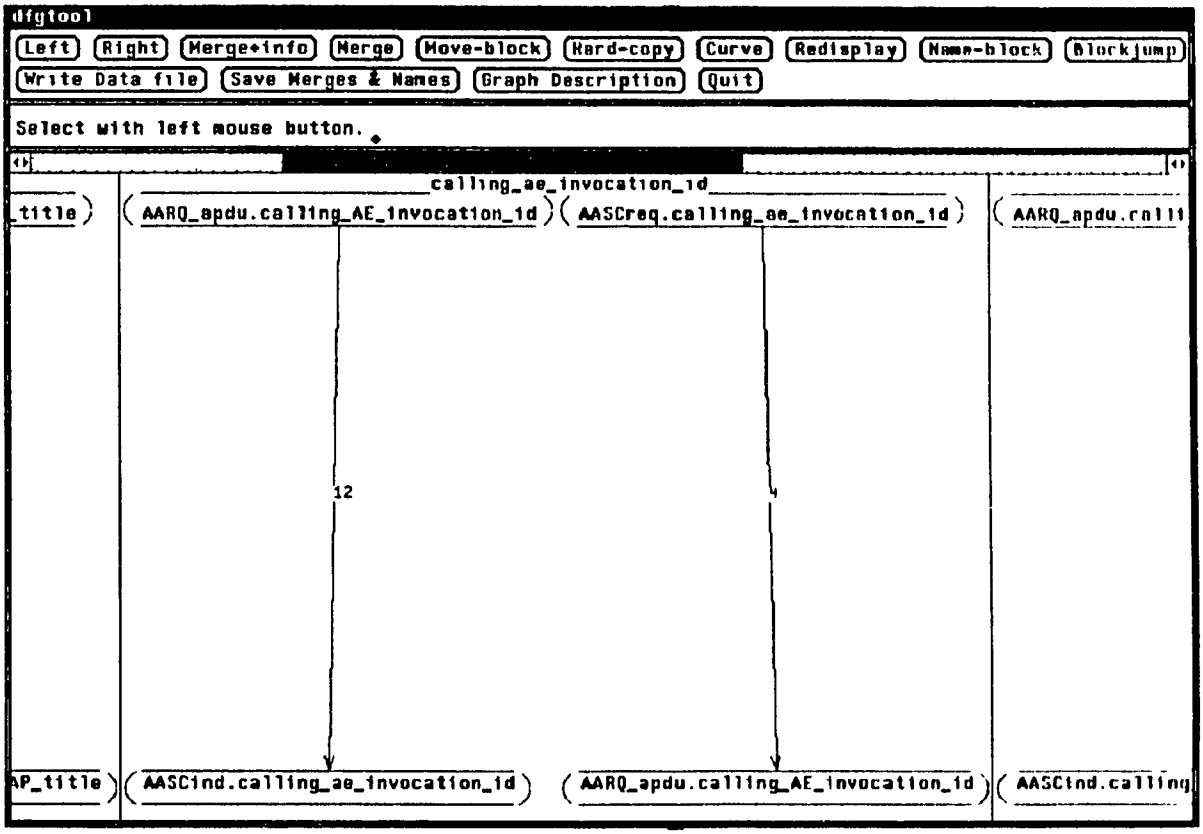


Figure 6.2 A part of the ACSE data flow graph displayed by LOTEST

Data flow graphs serve to represent protocol functions such as protocol_version, user_information, etc. These functions are obtained by the block merging procedure. This process yields twenty four functions for the ACSE protocol. These are given below with the transition numbers associated with each.

Functions	Transition numbers
source_information	2 17 29 42 60 64 68 90 94 114 117 136 139 143 160
user_information	2 17 29 42 60 64 68 90 94 114 117 136 139 143 160 4 7 16 28 41 59 63 67 89 93 113 116 135 138 142 159 11 12 22 23 35 36 39 57 66 92 40 58 65 91 112 13 4 78 79 103 104 131 150 153 15 4 164 165
protocol_version	4 11 35 36
application_context_name	4 12 11 22 23 35 36
called_ap_title	4 12
called_ae_qualifier	4 12
called_ap_invocation_id	4 12
called_ae_invocation_id	4 12
calling_ap_title	4 12
calling_ae_qualifier	4 12
calling_ap_invocation_id	4 12
calling_ae_invocation_id	4 12
implementation_information	4 11 35 36
abort_source	7 16 28 41 59 63 67 89 9 3 113 116 135 138 142 15 9
result	11 22 23 35 36 65 91 112 134
result_source_diagnostic	11 35 36 12 22 23
responding_ap_title	11 22 23 35 36
responding_ae_qualifier	11 22 23 35 36
responding_ap_invocation_id	11 22 23 35 36
responding_ae_invocation_id	11 22 23 35 36
get_pres_parms_set	12 22 23
source_result	22 23
flow_control	22 40 39 35 58 57 96 119 112 163 134 169 144 161
reason	39 57 66 92 40 58 65 91 112 134 78 79 103 104 131 150 153 154 164 165

6.3.2 ANALYSIS AND REDUCTION OF GENERATED TEST CASES

Let us consider the test case t_{27}. Figure 6.3 shows its test case dependency graph. The nodes are numbered as follows: first the transition number is placed followed by a period and the tuple number. For a-nodes the tuple number is 1, for p-nodes it is 5 or 6 depending on whether it is in the guard or condition clause of the transition. Similarly, for s-nodes the tuple number is 7 or 8 depending on whether it is in the action or assignment clause of the transition.

Figure 6.4 shows the graph that results from taking slices of the test case dependency graph from Figure 6.3 with respect to each of the p-nodes of the test case dependency graph. The same node numbering is used.

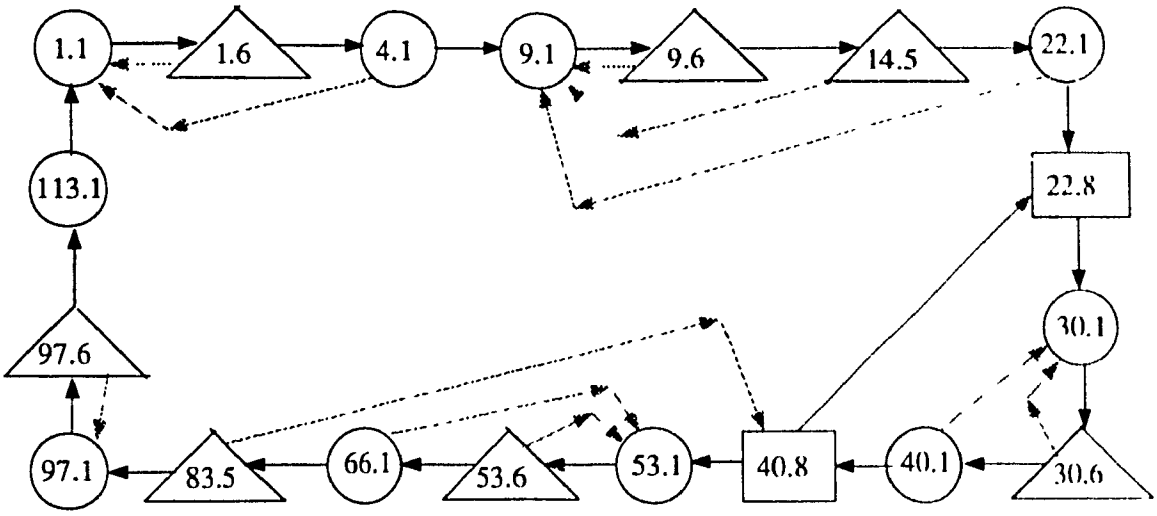


Figure 6.3 Test case dependency graph for t_{27}.

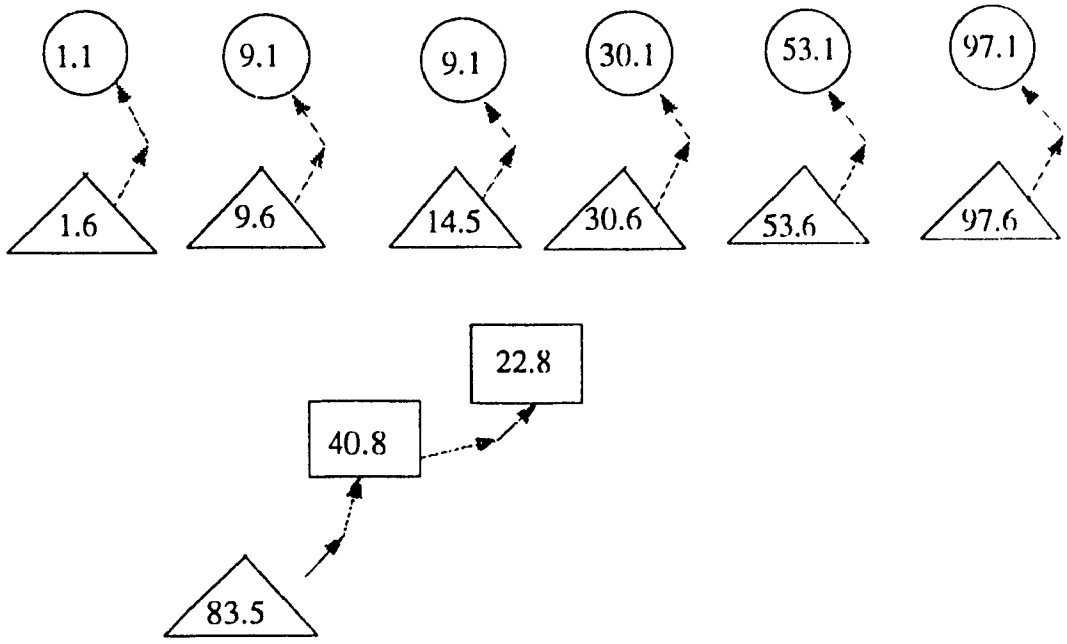


Figure 6.4 Predicate slices of the test case dependency graph t_{27}

In the predicate slice of Figure 6.4, the predicate node 83.5 is dependent on assignment node 40.8, which in turn is dependent on the assignment node 22.8. The predicate node evaluates to false. The reason is that at s-node 22.8, c12 is assigned to "calling", and then at s-node 40.8 is assigned to c10, i.e., now c10 has the value of "calling". However, the predicate at p-node 83.5 will be true only when c10 is "called". The test case t_{29} which is the same as t_{27} except for the last four transitions which are listed:

< i, 76, 74, 84, [eq(c10,calling)], true, ϵ, ϵ >,

< A?x7:primitive, 74, 67, 101, [IsAABRreq(x7)], ϵ, ϵ >,

< P!ABRT_apdu(acse_service_user, type_genere023(Not_Present)):ABRT_apdu,

67, 66, 116, true, true, ϵ, ϵ >,

< i_r, 66, 256, 132, true, true, ϵ, ϵ >

We reduce the dependency graph by eliminating all p-nodes from which a-nodes are not reachable through the data dependency edges. For example, in the predicate slice of Figure 6.4 no a-nodes can be reachable starting from the p-node 83.5. The reduced test case dependency graph of test case t_{29} is shown in Figure 6.5.

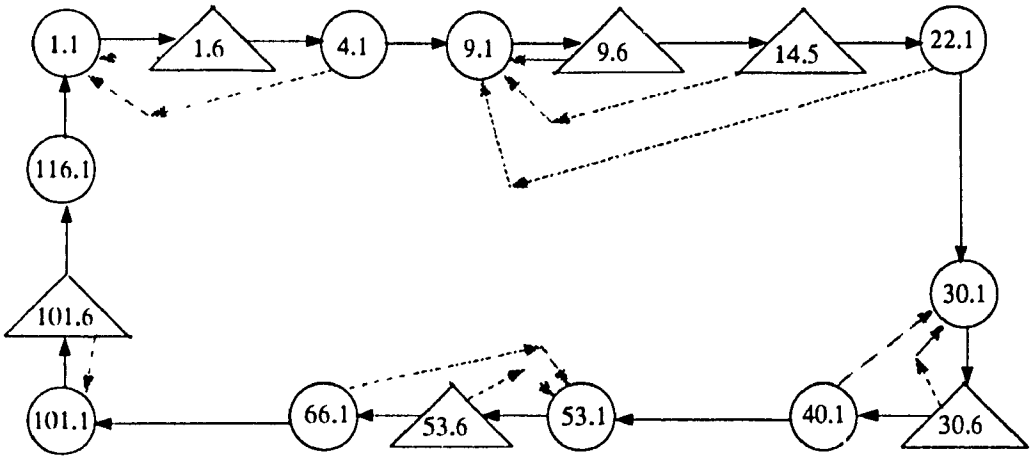


Figure 6.5 Reduced test case dependency graph of test case t_{29}.

We obtained the same results when we use DFG and Edittest tool to reduce the test cases. The set D obtained from the DFG for the test case t_{29} is $\{calling, c12, c10\}$. The algorithm eliminate the statements $\{c12 \leftarrow calling, c10 \leftarrow c12\}$ and the predicate

$[eq(c10, calling)]$ from the test case t_{29}. These are the only statements and predicates eliminated from the TCDG.

6.3.3 DYNAMIC CONSTRAINT REPRESENTATION

As an illustration let us consider the RTCDG of test case t_{29}. Let the SEND event a-node be 116.1, i.e., the event

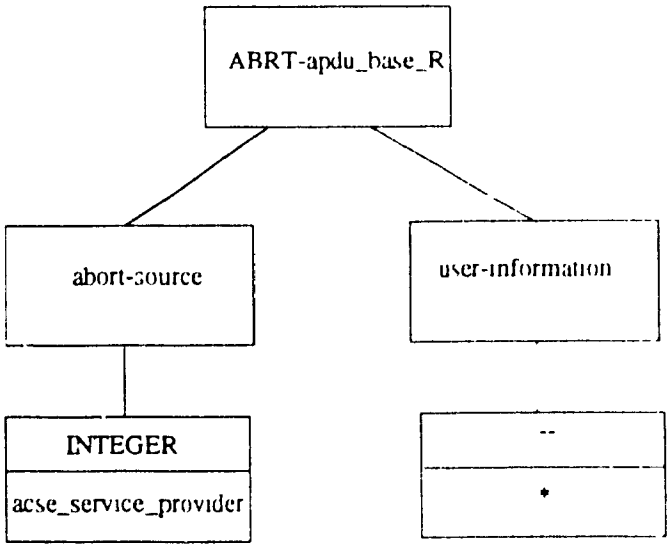
P!ABRT_apdu(acse_service_user, type_genere023 (Not_Present)): ABRT_apdu

which consists of two fields:

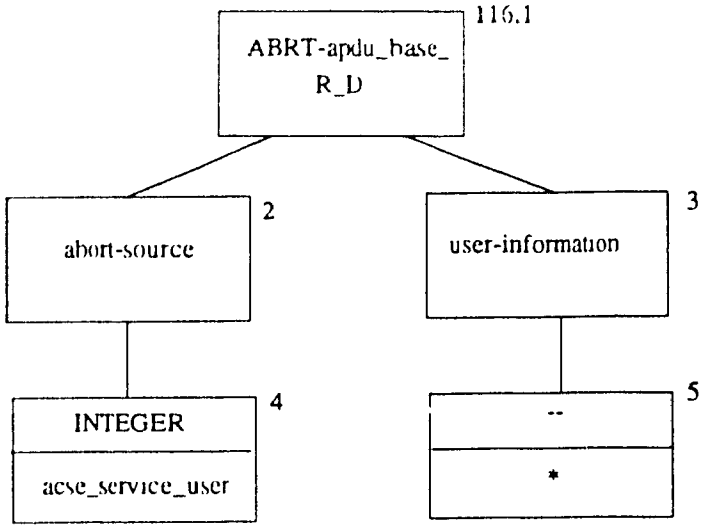
field_1 = "acse_service_user"

field_2 = "type_genere023 (Not_Present)".

The base constraint of ABRT-apdu is shown below:



The receive dynamic constraint generated by the algorithm is shown below:



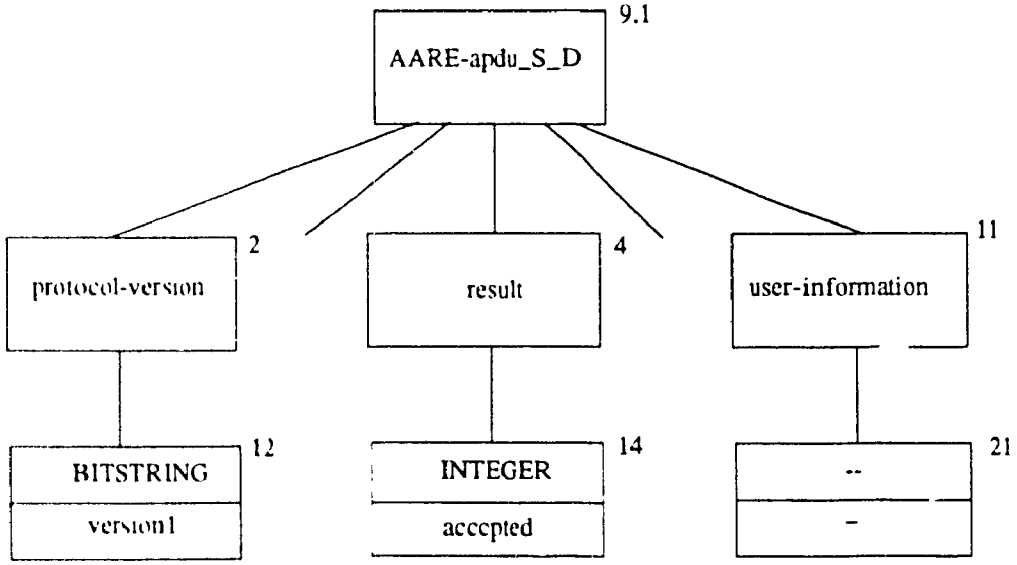
For send event dynamic constraints let us consider the receive event a-node to be 9.1, i.e., the event

P? x14:ACSE_apdu

The p-node 14.5 has a dependency on this a-node and it reads as

[eq(result(get_AARE(x14),accepted]

which can be transformed to [result = "accepted"]. The dynamic constraints generated by the algorithm for the event AARE_apdu by enhancing the base constraint AARE_apdu_base in the form of an I/O diagram are as follows:



All other constraints are given in appendix D.

6.3.4 TEST CASE SELECTION AND REPRESENTATION

For the ACSE protocol, the valid behaviour group can be subdivided into five sub-groups according to five functions: Association establishment, Normal release, Abnormal release, Parameter variations and Rules for extensibility. A complete structure of ACSE test suite hierarchy with elaborated Valid Behaviour group is shown in Figure 6.6 [32]. It is observed that for the ACSE protocol no inopportune tests can be identified since each ACSE PDU can only be mapped to a specific presentation ASP.

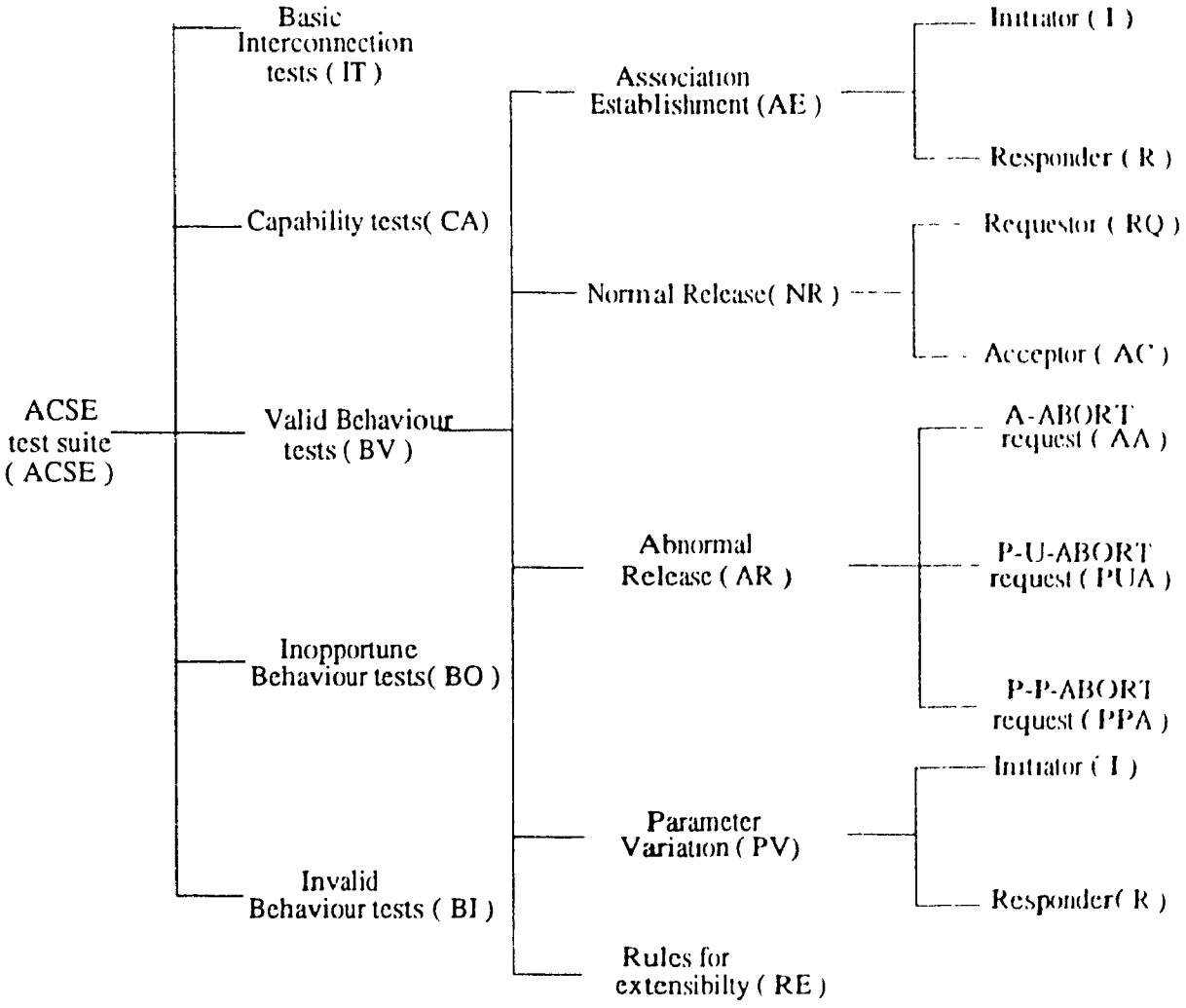


Figure 6.6 ACSE test suite structure.

The "A-ABORT request" node can be identified by the path ACSE/BV/AR/AA in Figure 6.6. At any node, the subtree rooted at that node can be considered as a test subgroup. The objective of this node is [32]:

IUT service user invokes the abort with an A-ABORT request primitive,

check the IUT sends an ABORT with abort-source= "acse_service_user"

Seven test purposes are defined under the test subgroup A-ABORT request:

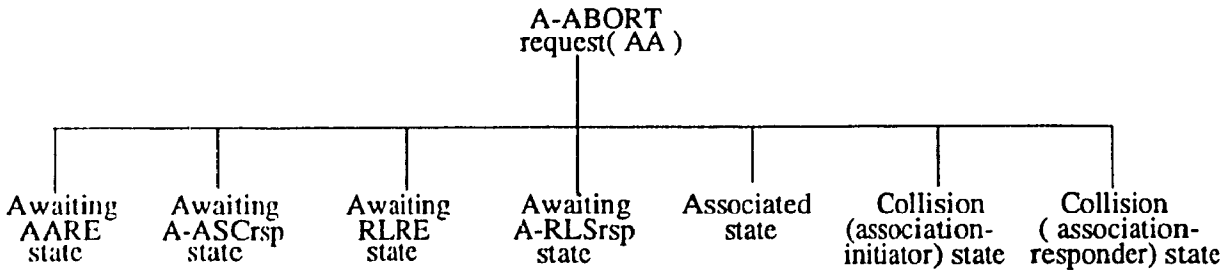


Figure 6.7 Test purposes of the test subgroup A-ABORT request.

Test case t_{29} clearly satisfies the ACSE/BV/AR/AA test group objective given in Figure 6.6. Now we have to identify, out of the seven test purposes in Figure 6.7, which test purpose is satisfied by this test case. The A-ABORT service primitive is invoked by the user at a collision state (a-node 101.1 of Figure 6.5), where IUT originates the association. Hence the test purpose six from the right of Figure 6.7 can possibly be satisfied by the test case t_{29}. Next comes the checking of base and dynamic constraints. Since the main events are A-ABORT request ASP and ABRT_apdu PDU, constraint validation is concentrated on these events. The send event A-ABORT request must satisfy the base constraints only, since there is no dynamic constraint associated with the event. However, the receive event ABRT_apdu (a-node 116.1 in Figure 6.5) must satisfy the dynamic constraint ABRT_apdu_D1, i.e., abort-source field of the received PDU must be equal to "acse_service_user".

The graphical control flow behaviour representation (CFBR) of the test case t_{29} is given in Figure 6.8. The "*" in the a-nodes represents the inversion.

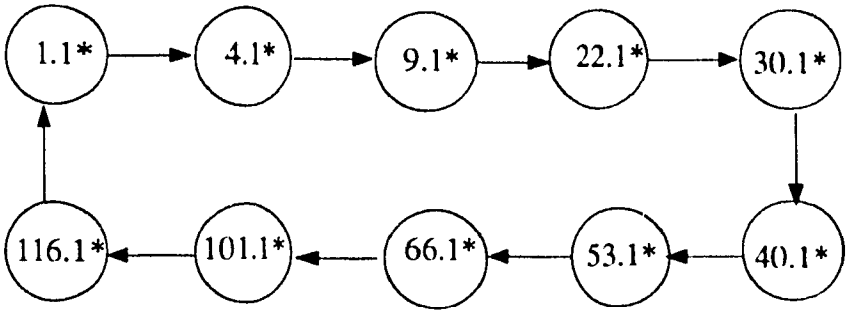


Figure 6.8 Control flow behaviour representation of the RTCDG of t_{29}

Applying the behaviour enhancement algorithm of Section 5.5.1 a pass verdict is associated with the a-node 116.1* and OTHERWISE a-nodes are added to become alternatives to the a-nodes 4.1*, 22.1*, 40.1*, 66.1*, and 116.1*. Step 3 assigns fail verdicts to all these OTHERWISE a-nodes. In this case, the procedure in section 5.5.2 does not modify the test case. The enhanced CFBR of the resulting test case is shown in Figure 6.9. The test starts at node 1.1* and ends successfully at a-node 116.1* or unsuccessfully at any OTHERWISE a-nodes.

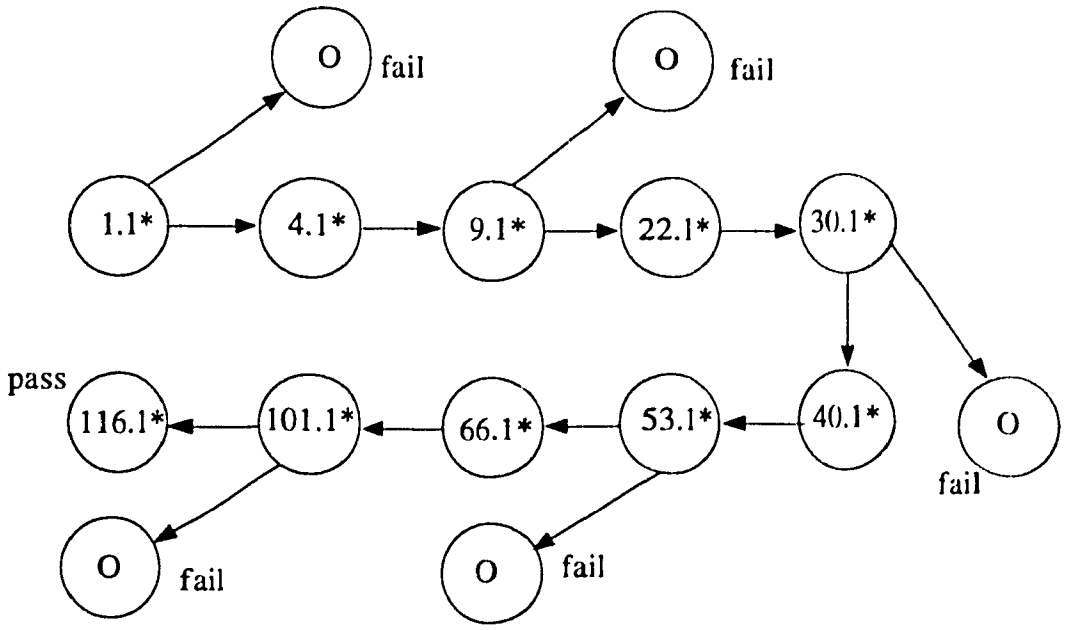


Figure 6.9 Enhanced CFBR of the selected test case t_{29}.

6.4 TEST SUITE DESIGN FOR LAPB PROTOCOL

A formal specification of the LAPB protocol in SDL is taken as the last example [22]. The specification contains four processes out of which we consider only the main "HDLC" process. An EFSM chart is manually generated from this specification, containing 68 states and 107 transitions. It is given in appendix C.

The next step is to generate test cases and to draw the data flow graph. As mentioned earlier, the total number of test cases is determined by the formula d_{in}(initial state)+d'_{in}(initial state), where d_{in}(initial state) is the indegree of the initial state and d'_{in}(initial state) is the number of edges incoming to the initial state added during the conversion of the EFSM chart to an Euler graph. Applying the above formula to the ACSE EFSM chart, d_{in} is 2 and d'_{in} is 33, therefore yielding 35 test cases. One complete test case, t_{17}, is listed below:

Test case t_{17}

< DSAP_in?DCONreq:signal, 1, 2, 1, true, true, ϵ, ϵ >,

< in2!SABM, 2, 8, 10, true, true, [T← NOW+T1, ta ← ta+1] >,

< i_s, 8, 9, 11, true, true, ϵ, ϵ >,

< i, 9, 2, 12, [not(ta=N2)], true, ϵ, ϵ >,

< in1?UA:signal, 8, 10,14, true, true, ϵ, [T← 0, REJ_sent ←False, t_running ← False,

vs ← 0, vr ← 0, vos ←0, tb ← qnew, nos ← 0] >,

< DSAP_out!DCONconf, 10, 12, 15, true, true, ϵ, ϵ >,

< DSAP_in?DDTreq(data):signal, 12, 13, 73, true, true, ϵ, [tb ← enqueue(data,tb),

ack ← False] >,

< i, 13, 14, 20, [not(empty(tb) OR (vs = vos ++ k))], true, ϵ, [data ← dequeue(tb),

tb ←rest(tb)] >,

< in2!I_frame(vs, vr, data), 14, 15, 21, true, true, ϵ, [rtb(vs) ← data, vs ← vs+1] >,

< i, 15, 13, 23, [not(t_running)], true, ϵ, [T← NOW+T1, t_running ← true] >,

< i, 13, 39, 18, [empty(tb) OR (vs = vos ++ k)], true, ϵ, ϵ >,

< i, 39, 12, 78, [not(ack)], true, ϵ, ϵ >,

< in1?DISC:signal, 12, 31, 75, true, true, ϵ, [ta $\leftarrow$0] >,

< in2!UA, 31, 32, 76, true, true, ϵ, ϵ >,

< DSAP_out!DDISind, 32, 1, 77, true, true, ϵ, ϵ >

To draw the data flow graph we have to scan the *when, action* and *assignment clause* of each rule of the chart. Part of the data flow graph of the HDLC protocol is shown in Figure 6.10. Data flow graphs serve to represent protocol functions such as counter, rej_frame etc. These function are obtained by applying the block merging procedure. Block merging process yields nine functions for HDLC protocol, which are listed below with a brief description of each function:

Function	Description of the function
V(R)	In this test group the correct setting of the value of V(R) state variable is tested
V(S)	In this test group the correct setting of the value of V(S) state variable is tested
I_frame	In this test group transmission of numbered information(I) frames is tested
User_to_peer	In this test group user_to_peer data transfer is tested
Peer_to_user	In this test group peer_to_user data transfer is tested
Rej_frame	In this test group IUT is expected to reject frames with bad sequence
Counter	In this test group retransmission counter is tested
Timer	In this test group timer of IUT is tested
Acknowledge	In this group the acknowledgement is tested

6.4.1 ANALYSIS AND REDUCTION OF GENERATED TEST CASES

Let us consider the test case t_{17}. Figure 6.11 shows its test case dependency graph. The same node numbering scheme is used except that s-nodes created for each assignment in the assignment clause are numbered with lower case letters starting with "a".

Figure 6.12 shows the graph that results from taking slices of the test case dependency graph from Figure 6.11 with respect to each of the p-nodes of the dependency graph. The same node numbering is used.

This test case, t_{17}, is feasible. However, a total six test cases are found to be infeasible.The reduced test case dependency graph of test case t_{17} is shown in Figure 6.13 . Here also we obtain the same results when we use data flow graph to reduce the test case.

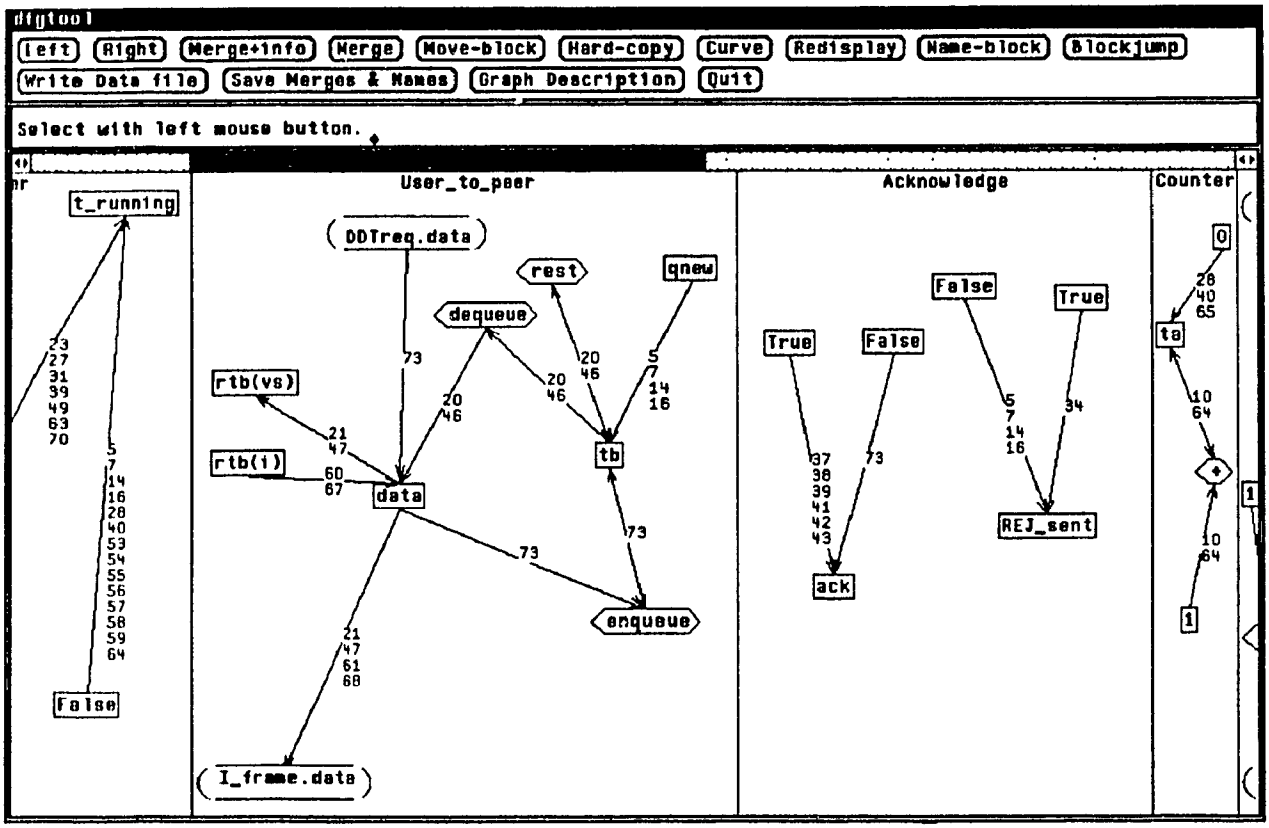


Figure 6.10 A part of the HDLC data flow graph.

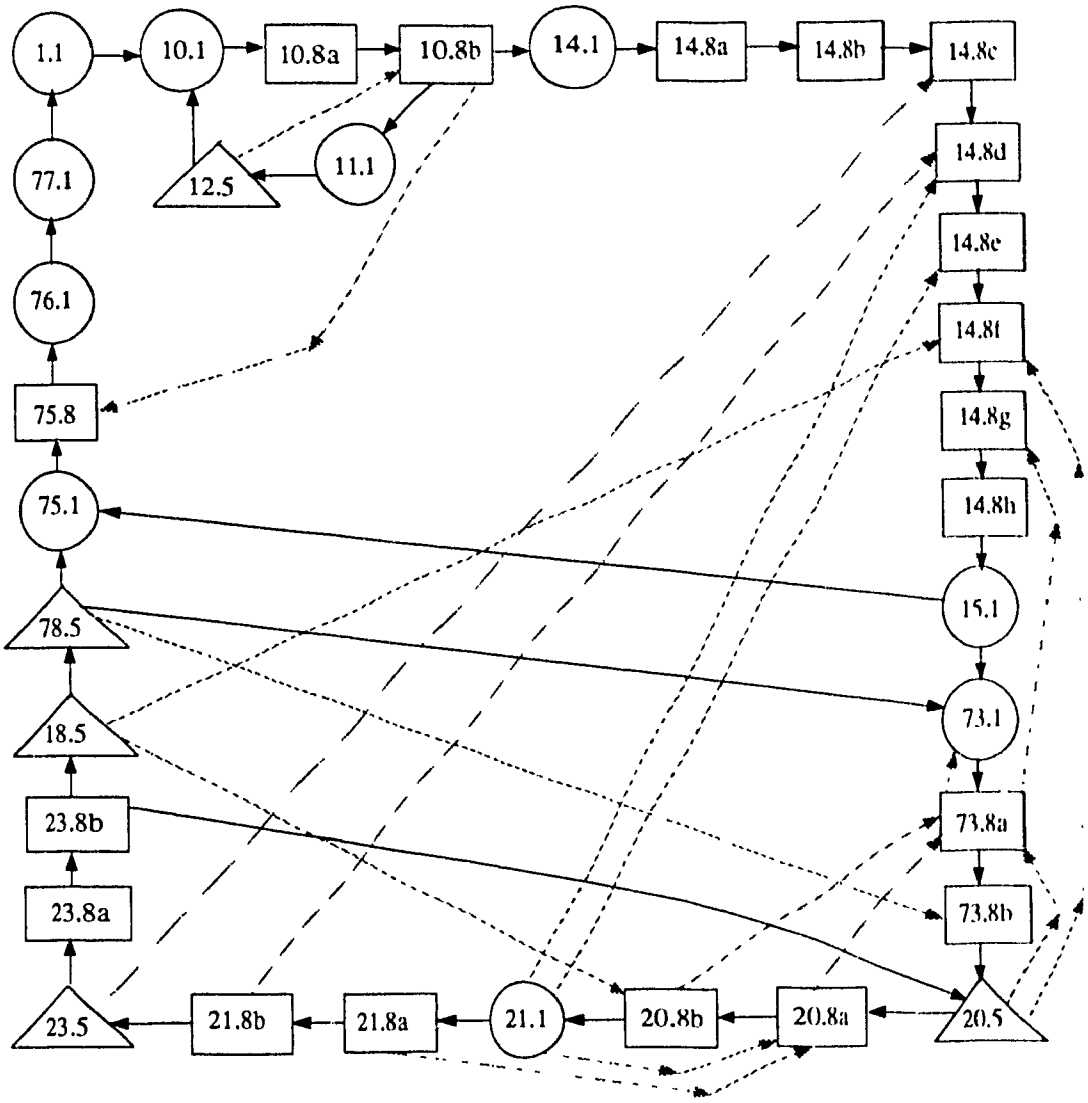


Figure 6.11 Test case dependency graph for t_{17}

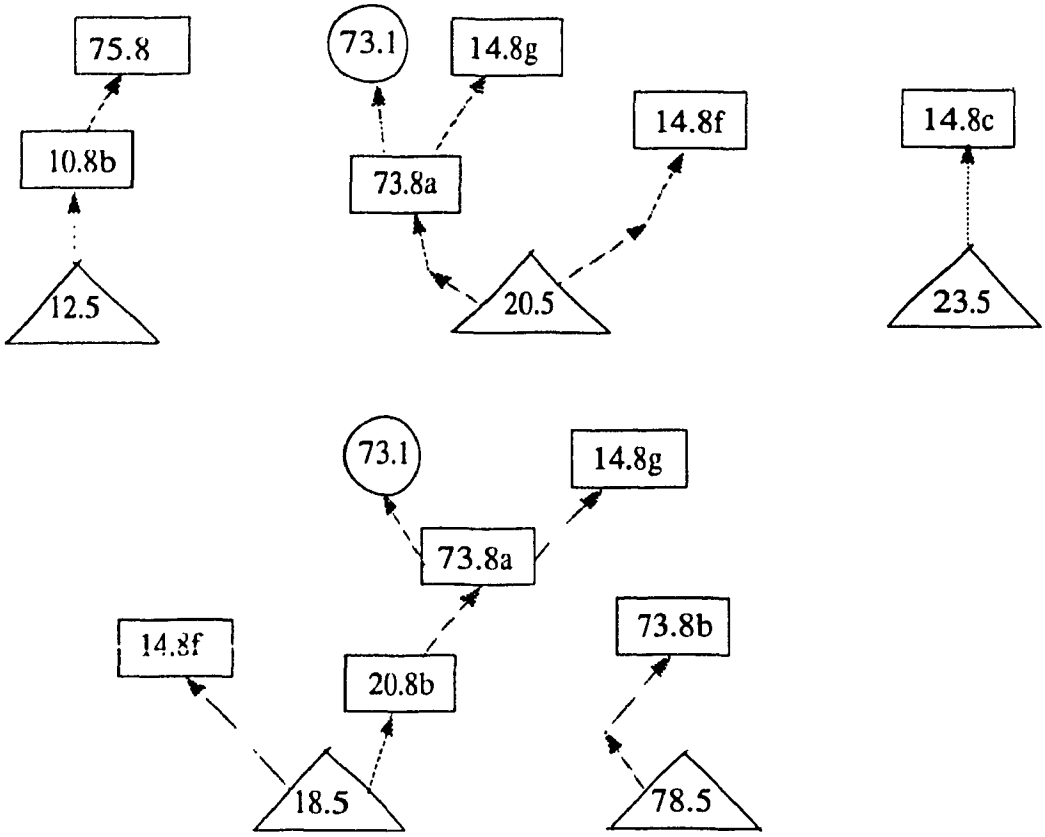


Figure 6.12 Predicate slices of the test case dependency graph t_{17}

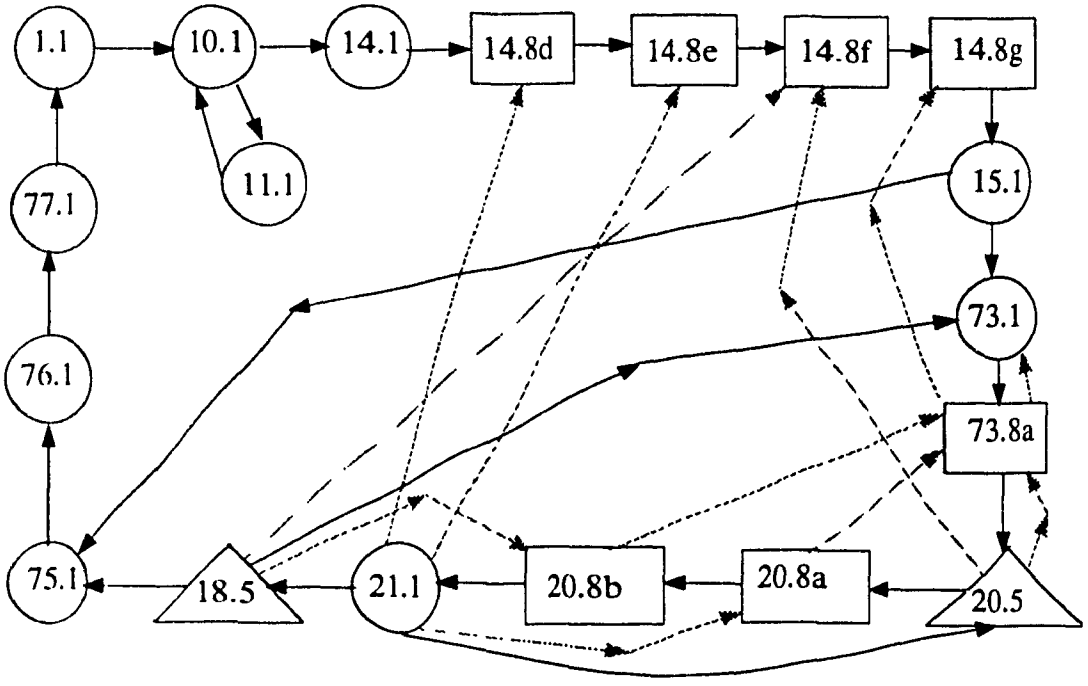


Figure 6.13 Reduced test case dependency graph of test case t_{17}.

6.4.2 TEST CASE SELECTION AND REPRESENTATION

The graphical control flow behaviour representation(CFBR) of the test case t_{17} is given in Figure 6.14. The "*" in the a-nodes represents the inversion.

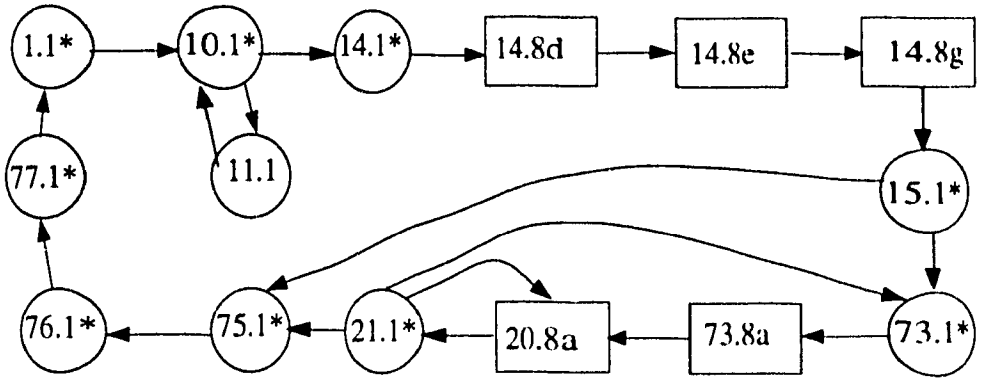


Figure 6.14 Control flow behaviour representation of the RTCDG of t_{17}

Test case t_{17} can be used to verify that IUT can send two I frames in the information transfer phase. This test belongs to the test group X.25–DL2/BV/IT which contains all the valid behaviour tests about the Information Transfer phase [29].

Step 1 and 2 of the algorithm in section 5.5.1 eliminates the loop from the i_s a-node 11.1^* as shown in the Figure 6.14 . In Step 2 OTHERWISE a-nodes are added to become alternatives to the a-nodes 10.1^*, 15.1^*, 21.1^*, 76.1^* and 77.1^*. Step 3 assigns fail verdicts to all these OTHERWISE a-nodes and associates a pass verdict to the a-node 77.1^*.

In step 1 the procedure in section 5.5.2 eliminates the edges from 15.1^* to 75.1^* and 21.1^* to 73.1^*. In step 2, the loop between nodes 21.1^* and $20.8a^*$ is expanded so that the loop in which an I-frame is transmitted is made to execute twice due to the test purpose. In step 3, a pass verdict is assigned to the second a-node 21.1^*, since the test purpose is achieved with the reception of this event (Figure 6.15). In step 2, dynamic constraints are modified to achieve the test purpose. Assuming a frame size of 256 bytes the send constraint on a-node 73.1^* (Upper tester sending a DDTreq) is modified so that *data* field of DDTreq is assigned to a string of x bytes with $256< x <512$. Then the receive

constraint on the first a-node 21.1^* is changed to receive the first 256 bytes of this string and the second a-node 21.1^* to receive the remaining bytes. The modified enhanced CFBR of the resulting test case is shown in Figure 6.15. The test starts at the a-node 1.1^* and ends successfully at the a-node 77.1^* or unsuccessfully at any OTHERWISE a-nodes. The constraints are given in appendix E.

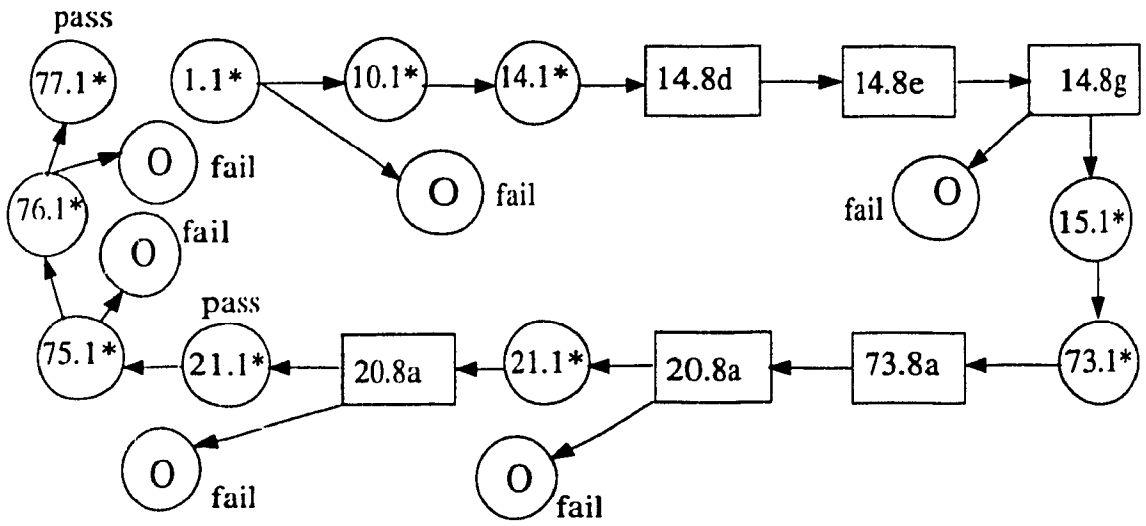


Figure 6.15 Enhanced CFBR of the selected test case t_{17}.

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

This thesis does not represent the start of a new field or the culmination of an existing one. It is a contribution to the significant effort that has been made and is being made by a number of researchers to make communication protocols an everyday reality. Our goal in this thesis has been to develop easy-to-use and general in nature algorithms for solving some basic problems of conformance testing. In particular, we have considered the following problems of conformance testing:

i. the generation of test cases from the formal specification languages LOTOS and SDL;
ii. the selection of test cases which meet certain data flow coverage criteria;
iii. the representation of test cases for LS and RS architectures.

These problems are chosen from practical considerations. As we have discussed in this thesis, their solution in an efficient manner is a prerequisite to the demonstration of networks in which components made by different manufacturers can interwork properly and effectively, which in turn is necessary to fulfill the requirements of the OSI standards. The main drawback in solving these problems with many of the existing methods in the test suite design literature is that they are based on deterministic finite state machines without any consideration of test architectures. The main feature of the algorithms presented in this thesis is that they are general in nature and therefore can be applied to almost all existing formal models.

The objective of this thesis as discussed is two fold: first, to introduce a unified model (EFSM chart and I/O diagram) for the existing protocol specification languages and to use this model as an interim step for the generation of tests from protocol specifications; and second, to develop a theory to specify LS and RS tester's behaviour from the

generated test cases. In order to achieve these two goals, we have developed several new algorithms and applied the algorithms to the design of test suites for the HDLC and ACSE protocols. The results presented in the thesis and related issues that may benefit from further investigation are summarized below.

In **chapter I**, the problem of protocol testing is introduced. The work in several related areas is surveyed including an introduction to the test architectures, test suite structure and various types of testing proposed by ISO for the testing of protocol implementations for conformance to the standard specification.

In **chapter II**, we introduced the unified model called EFSM chart and I/O diagram. Furthermore, we discussed the relationship between I/O diagrams and ASN.1.

In **chapter III**, we developed algorithms to transform a protocol specification written in the LOTOS or SDL specification languages into an Extended Finite State Machine chart. The specifications (LOTOS or SDL) are transformed into the corresponding EFSM chart in two phases. In the first phase the specification is transformed into a semantically equivalent form. In the second phase it is converted into the EFSM chart. In the SDL to EFSM chart construction algorithm, we deal with timeout transitions the same way as spontaneous transitions.

Based on the EFSM chart, **chapter IV** presented several algorithms to generate and analyze test cases. The test case generation algorithm developed takes nondeterminism into consideration. Furthermore, a new algorithm is proposed to construct a data flow graph from the chart, which is used to identify the protocol functions needed to test the data flow aspect of an IUT. The zero-one integer programming technique is used to select test cases to meet the data flow coverage requirement. Finally, the generated test cases are modeled as a test case dependency graph and then evaluated by taking predicate slices from it. Redundant assignments and predicates in all the feasible test cases are removed by reducing the test cases. This is achieved by using the test case dependency graph as well as the data flow graph.

In **chapter V**, we discussed input/output flow representation followed by control

flow representation. The input/output representation is in the form of an I/O diagram. It provides constraints on the events. Two types of input/output representations are considered: *base* and *dynamic*. New algorithms are developed to generate base and dynamic constraints. Finally, the control flow behaviour is represented by inverting the direction of the actions and eliminating all the predicates from the reduced test case dependency graph. Assuming a test case hierarchy with test purposes as leaf nodes, the selection process associates a purpose to each represented test case. Test cases are then adapted for the RS architecture.

Chapter VI applies the test design methodology to the ACSE and HDLC protocols written in the LOTOS and SDL specification languages respectively. We also discussed LOTEST, a software tool that partly implements the methodology discussed in this thesis.

7.2 FUTURE WORK

The nature of research is such that the solution of one problem often gives rise to many new questions or problems. In the case of the research that is presented in this thesis, the following questions surface naturally.

i) *Parallel/Distributed algorithm*: The test generation algorithm proposed in section 4.1 is sequential. It would be interesting to investigate the possibility of developing a parallel/distributed algorithm.

ii) *Adaptation to other test architectures*: In this thesis test suites are designed for LS and RS architectures. Abstract test suite design for other architectures used in practice such as distributed, coordinated architectures is a topic for future research.

iii) *Software tools*: The LOTEST [53] software tool developed partially implements the design methodology discussed in this thesis. In particular, the algorithms presented in chapters V, VI and VII have not been implemented. It would be interesting to implement these algorithms.

iv) *Test case representation in TTCN*: The representation of CFBR graph and I/O dynamic constraints in the form of TTCN is one of the most important lines of future developments.

Once the test cases are represented in TTCN, they can be executed to perform the real testing of the implementation.

REFERENCES

[1] A . V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An Optimization Technique for Protocol Conformance Test Generation based on UIO Sequences and Chinese Postman Tours", IEEE Transaction on Communication, Vol.39(11), pp.1604–1615, Nov. 1991.

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

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

[4] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN System, Vol.14, pp.25–59, 1987.

[5] E. Brinksma, and G. Karjoth, "A Specification of the OSI Transport Service in LOTOS", Proc 5th IFIP Symposium on Protocols, June 1985.

[6] E. Brinksma, "A Theory for the Derivation of Tests", Proc 8th IFIP Symposium on Protocols, Atlantic City, June 1988.

[7] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems", Computer Networks and ISDN Systems, Vol.14, pp.3–23, 1987.

[8] S. P. Van de Burgt, J. Kroon, and A. M. Peeters, "Interactive Test Generation from LOTOS Specification", Technical report T1–PU–92–XXX, PTT–Research, Neher Labs, January, 1992.

[9] CCITT, "Specification and Description Language SDL", Recommendation Z.100, 1988.

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

[11] A. T. Dahbura, K. K. Sabnani, and M. U. Uyar, "Formal Methods for Generating Protocol Conformance Test Sequences", Proceeding of the IEEE, Vol.78, pp. 1317-1326, 1990

[12] J. Edmonds and E. L. Johnson, "Matching, Euler Tours and the Chinese Postman," Mathematical Programming, Vol.5, pp.88-124, 1973.

[13] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification*, Springer-Verlag, Berlin, 1985.

[14] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The Program Dependence Graph and its Uses in Optimization", ACM Transactions on Programming Languages and Systems, Vol.9(3), pp.319-349, July 1987.

[15] B. Forghani and B. Sarikaya, "Semi-Automatic Test Suite Generation from Estelle", IEE Software Engineering Journal, to appear in 1992.

[16] A. Gibbons, *Algorithm Graph Theory*, Cambridge Univ. Press, 1985.

[17] A. Gill, "State-identification Experiments in Finite Automata," Information and Control, Vol.4, pp.132-154, 1961.

[18] A. Gill, *Introduction to the Theory of Finite-State Machines.* New York: McGraw-Hill, 1962.

[19] D. Gueraichi, and L. Logrippo, "Derivation of Test Cases for LAP-B from LOTOS Specification", FORTE '89.

[20] R. Guillemot and L. Logrippo, "Derivation of Useful Execution Trees From LOTOS Specification by Using an Interpreter", FORTE '88.

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

[22] D. Hogrefe, "Protocol and Service Specification with SDL: The X.25 Case Study", FBI–HH–B–134/88, Fachbereich Informatik.

[23] D. Hogrefe, "Automatic generation test case from SDL Specification", SDL Newsletter, No.12, June 1988.

[24] D. Hogrefe, "OSI Formal Specification Case Study: The Inres Protocol and Service", Technical Report, University of Bern, May 1991.

[25] W. E. Howden, *Functional Program Testing and Analysis*, McGraw Hill, 1987.

[26] [ISO IS8807] "LOTOS, a Formal Description Technique based on the Temporal Ordering of Observational Behavior", ISO/TC97/SC21/WG1-FDT/SC-C, June 1988.

[27] [ISO DIS8650] "Protocol Specification for the Association Control Service Elements", January 1988.

[28] [ISO 8824] "Profile of Abstract Syntax Notation-one", IS 8824, 1987.

[29] [ISO 8882–2] ISO/ IEC JTC1/ SC6, "X.25–DTE Conformance Testing: Data Link Layer Test Suite", 1990.

[30] [ISO/IEC 9646] "Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Frame work", 1991.

[31] [ISO/IEC 9646–3] "The Tree and Tabular Combined Notation", Part 3 of ISO/IEC 9646, September, 1991.

[32] [ISO DIS10169] "Information Technology – Open Systems Interconnection- Conformance test suite for ACSE Protocol, Part1: Test suite structure and Purpose", 1990.

[33] M. Jackson, *System Development*, Prentice Hall, 1983.

[34] G. Karjoth, " Implementing Process Algebra Specifications by State Machines", Proc 8th IFIP Symposium on Protocols, Atlantic City, June 1988.

[35] Z. Kohavi, *Switching and Finite Automata Theory*. New York : McGrawHill, 1978.

[36] M. K. Kuan, *Graphic programming using odd or even points*, Chinese Math, Vol.1, pp. 273-277, 1962.

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

[38] R. J. Linn, "Conformance Evaluation Mehodology and Protocol Testing", IEEE Transaction of Selected Area in Communications, Vol.7(7), pp. 1141–1158, September 1989.

[39] M. Liu, *Special Issue on Protocol Engineering*, IEEE Transaction On Computers,

Vol.40(4), 1991.

[40] L. Logrippo, A. Obaid, J. P. Braind and M. C. Fehri, "An Interpreter for LOTOS, A specification Language for Distributed Systems", Software-Practice and Experience, Vol.18(4), pp.365-385, April 1988.

[41] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol.92, Springer-Verlag, 1980.

[42] R. Milner, "A Complete Inference System for a Class of Regular Behaviors", Journal of Computer and System Sciences, Vol.28, pp. 439-466, 1984.

[43] E. F. Moore, "Gedanken-experiments on Sequential Machines", Automata Studies, Annals of Mathematical Studies, no. 34 Princeton Univ. Press. NJ., pp. 129-153, 1956.

[44] S. Naito and M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours," Proc. 11th IEEE Fault Tolerant Comput. Symp., IEEE Computer Soc. Press, pp. 238-243, 1981.

[45] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Transaction on Software Engineering, Vol.11, pp. 367–375, 1985.

[46] D. Rayner, "OSI Conformance Testing", Computer Networks and ISDN Systems, Vol.14(1), pp.79–98, 1987.

[47] A. Bourguet-Rouger and P. Combes, "Exhaustive Validation and Test Generation in Elvis", SDL-89:The language at work, North Holland, 1989.

[48] K. K. Sabnani and A. T. Dahbura, "A new technique for generating protocol tests," Proc. 9th Data Communication Symp., IEEE Computer Soc. Press, pp. 36-43, September 1985.

[49] K. K. Sabnani and A. T. Dahbura, " A Protocol Testing Procedure," Computer Networks and ISDN Systems, Vol.15(4), pp. 285-297, 1988.

[50] B. Sarikaya, " Conformance Testing: Architectures and Test Sequences", Computer Network and ISDN Systems 17, pp.111–126, 1989.

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

[52] B. Sarikaya, G. v. Bochman and E. Cerny, "A Test Design Methodology for Protocol Testing ", IEEE Transactions on Software Engineering, Vol.SE-13, May 1987.

[53] B. Sarikaya, P. Tripathy and S. Biedlingmaier, "LOTEST: A LOTOS Test Case Generation Tool", Tech. Rep., 1991.

[54] B. Sarikaya, B. Forghani, and S. Eswara, "An Estelle Based Test Generation Tool", Computer Communications, Nov. pp. 534-544, 1991.

[55] B. Sarikaya and A. Wiles, "Standard Conformance Test Specification Language TTCN", Computer Standards & Interfaces, April 1992.

[56] C. Steenbergen, *Conformance Testing of OSI Systems*, MSc Thesis, University of Twente, 1986.

[57] M. M. Syslo, N. Deo, and J. S. Kowalik, "Zero-one Integer Programming", in Discrete Optimization Algorithms, Prentice-Hall, 1983.

[58] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1983.

[59] J. Tretmans, "Test Case Derivation from LOTOS Specification", FORTE '89.

[60] P. Tripathy and B. Sarikaya, "Test Case Generation from LOTOS Specification" IEEE Trans. on Computers, Vol.40(4), pp.543-552, 1991.

[61] H. Ural, "Test Sequence Selection Based on Static Data Flow Analysis", Computer Communications, Vol.10(5), October, 1987.

[62] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing", IEEE Transaction on Communications, Vol.39(4), pp.514–523, 1991.

[63] M. U. Uyar and A. T. Dahbura, "Optimal test sequence generation for protocol: the Chinese postman algorithm applied to Q.931," Proc IEEE Global telecocommunication Conference, 1986.

[64] M. Weiser, "Program Slicing", IEEE Transactions on Software Engineering, Vol.

10(4), pp. 352-357, 1984.

[65] H. S. Wang, S. R. Hsu, and J. C. Lin, "A Generalized Optimal Path-selection Model for the Structure Program Testing", The Journal of Systems and Software, Vol.10, pp. 55–62, 1989.

[66] C. Wezeman, "The CO–OP Method for Compositional Derivation of Conformance Testers", Proc. 9th IFIP Symposium on Protocols, June 1989.

APPENDIX A

EFSM CHART OF ACSE PROTOCOL

```
when:A?x(29):primitive
from:256    to:121     transition:1
guard:      selection_predicate:[IsAASCreq(x(29))]
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:256    to:254     transition:2
guard:      selection_predicate:
actions:    assignments:

when:P?x(29):ACSE_apdu
from:256    to:249     transition:3
guard:      selection_predicate:[IsAARQ(x(29))]
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_0(AARQ_apdu(Bit(1),
app_context_name(get_AASCreq(x(29))),
called_ap_title(get_AASCreq(x(29))),
called_ae_qualifier(get_AASCreq(x(29))),
called_ap_invocation_id(get_AASCreq(x(29))),
called_ae_invocation_id(get_AASCreq(x(29))),
calling_ap_title(get_AASCreq(x(29))),
calling_ae_qualifier(get_AASCreq(x(29))),
calling_ap_invocation_id(get_AASCreq(x(29))),
calling_ae_invocation_id(get_AASCreq(x(29))),
type_genere010(Not_Present),
user_info(get_AASCreq(x(29))))):ACSE_apdu
from:121    to:120     transition:4
guard:      selection_predicate:
actions:    assignments:

when:i
from:249    to:125     transition:5
guard:[not(common_prot_version(get_AARQ(x(29))))]
selection_predicate:
actions:    assignments:
```

```
when:i
from:249    to:247     transition:6
guard:[common_prot_version(get_AARQ(x(29)))]
selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:254    to:253     transition:7
guard:      selection_predicate:
actions:    assignments:

when:A?x(13):primitive
from:120    to:113     transition:8
guard:      selection_predicate:[IsAABRreq(x(13))]
actions:    assignments:

when:P?x(14):ACSE_apdu
from:120    to:109     transition:9
guard:      selection_predicate:[IsAARE(x(14))]
actions:    assignments:

when:P?x(13):ACSE_apdu
from:120    to:117     transition:10
guard:      selection_predicate:[IsABRT(x(13))]
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(Bit(1),
application_context_name(get_AARQ(x(29))),
rejected_permanent,
Associate_source_diagnostic(Associate_source_diagnostic_genere_1
(no_common_acse_version)),
type_genere013(Not_Present),
type_genere014(Not_Present),
type_genere015(Not_Present),
type_genere016(Not_Present),
type_genere017(Not_Present),
type_genere018(Not_Preset)))):ACSE_apdu
from:125    to:124     transition:11
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(AASCind(optional(normal),
```

```
application_context_name(get_AARQ(x(29))),
calling_AP_title(get_AARQ(x(29))),
calling_AE_qualifier(get_AARQ(x(29))),
calling_AP_invocation_id(get_AARQ(x(29))),
calling_AE_invocation_id(get_AARQ(x(29))),
called_AP_title(get_AARQ(x(29))),
called_AE_qualifier(get_AARQ(x(29))),
called_AP_invocation_id(get_AARQ(x(29))),
called_AE_invocation_id(get_AARQ(x(29))),
user_information(get_AARQ(x(29))),
empty_presentation_parms_set)):primitive
from:247    to:246     transition:12
guard:      selection_predicate:
actions:    assignments:

when:ir
from:253    to:256     transition:13
guard:      selection_predicate:
actions:    assignments:

when:i
from:109    to:103     transition:14
guard:[eq(result(get_AARE(x(14))),accepted)]
selection_predicate:
actions:    assignments:

when:i
from:109    to:107     transition:15
guard:[eq(result(get_AARE(x(14))),rejected_permanent)]
selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:113    to:112     transition:16
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:117    to:116     transition:17
guard:      selection_predicate:
actions:    assignments:
```

```
when:ir
from:124    to:256     transition:18
guard:      selection_predicate:
actions:    assignments:

when:A?x(28):primitive
from:246    to:235     transition:19
guard:      selection_predicate:[IsAASCrsp(x(28))]
actions:    assignments:

when:A?x(27):primitive
from:246    to:239     transition:20
guard:      selection_predicate:[IsAABRreq(x(27))]
actions:    assignments:

when:P?x(27):ACSE_apdu
from:246    to:243     transition:21
guard:      selection_predicate:[IsABRT(x(27))]
actions:    assignments:

when:A!primitive(AASCcnf(application_context_name
(get_AARE(x(14))),
responding_AP_title(get_AARE(x(14))),
responding_AE_qualifier(get_AARE(x(14))),
responding_AP_invocation_id(get_AARE(x(14))),
responding_AE_invocation_id(get_AARE(x(14))),
user_information(get_AARE(x(14))),
result(get_AARE(x(14))),
acse_service_user,
optional(result_source_diagnostic(get_AARE(x(14)))),
empty_presentation_parms_set)):primitive
from:103    to:102     transition:22
guard:      selection_predicate:
actions:    assignments:c(12):= calling

when:A!primitive(AASCcnf(application_context_name
(get_AARE(x(14))),
responding_AP_title(get_AARE(x(14))),
responding_AE_qualifier(get_AARE(x(14))),
responding_AP_invocation_id(get_AARE(x(14))),
responding_AE_invocation_id(get_AARE(x(14))),
user_information(get_AARE(x(14))),
result(get_AARE(x(14))),
acse_service_user,
```

```
optional(result_source_diagnostic(get_AARE(x(14)))),
empty_presentation_parms_set)):primitive
from:107     to:106      transition:23
guard:       selection_predicate:
actions:     assignments:

when:ir
from:112     to:256      transition:24
guard:       selection_predicate:
actions:     assignments:

when:ir
from:116     to:256      transition:25
guard:       selection_predicate:
actions:     assignments:

when:i
from:235     to:229      transition:26
guard:[eq(result(get_AASCrsp(x(28))),accepted)]
selection_predicate:
actions:     assignments:

when:i
from:235     to:233      transition:27
guard:[not(eq(result(get_AASCrsp(x(28))),accepted))]
selection_predicate:
actions:     assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:239     to:238      transition:28
guard:       selection_predicate:
actions:     assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:243     to:242      transition:29
guard:       selection_predicate:
actions:     assignments:


when:A?x(12):primitive
from:102     to:90       transition:30
guard:       selection_predicate:[IsARLSreq(x(12))]
```

```
actions:    assignments:

when:A?x(11):primitive
from:102    to:95     transition:31
guard:      selection_predicate:[IsAABRreq(x(11))]
actions:    assignments:

when:P?x(12):ACSE_apdu
from:102    to:21     transition:32
guard:      selection_predicate:[IsRLRQ(x(12))]
actions:    assignments:

when:P?x(11):ACSE_apdu
from:102    to:99     transition:33
guard:      selection_predicate:[IsABRT(x(11))]
actions:    assignments:

when:ir
from:106    to:256     transition:34
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(Bit(1),
app_context_name(get_AASCrsp(x(28))),
result(get_AASCrsp(x(28))),
Associate_source_diagnostic(Associate_source_diagnostic_genere_0
(no_reason_given))
,responding_ap_title(get_AASCrsp(x(28))),
responding_ae_qualifier(get_AASCrsp(x(28))),
responding_ap_invocation_id(get_AASCrsp(x(28))),
responding_ae_invocation_id(get_AASCrsp(x(28))),
type_genere017(Not_Present),
user_info(get_AASCrsp(x(28))))):ACSE_apdu
from:229    to:228     transition:35
guard:      selection_predicate:
actions:    assignments:c(26):= called

when:P!ACSE_apdu(ACSE_apdu_genere_1(AARE_apdu(Bit(1),
app_context_name(get_AASCrsp(x(28))),
result(get_AASCrsp(x(28))),
Associate_source_diagnostic(Associate_source_diagnostic_genere_0
(no_reason_given)),
responding_ap_title(get_AASCrsp(x(28))),
responding_ae_qualifier(get_AASCrsp(x(28))),
```

```
responding_ap_invcation_id(get_AASCrsp(x(28))),
responding_ae_invocation_id(get_AASCrsp(x(28))),
type_genere017(Not_Present),
ser_info(get_AASCrsp(x(28))))):ACSE_apdu
from:233    to:232     transition:36
guard:      selection_predicate:
actions:    assignments:

when:ir
from:238    to:256     transition:37
guard:      selection_predicate:
actions:    assignments:

when:ir
from:242    to:256     transition:38
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLSind(reason(get_RLRQ(x(12))),
user_information(get_RLRQ(x(12))))):primitive
from:21     to:20      transition:39
guard:
selection_predicate:
actions:
assignments:c(2):=c(12)

when:P!ACSE_apdu(ACSE_apdu_genere_2(RLRQ_apdu
(reason(get_ARLSreq(x(12))),
user_info(get_ARLSreq(x(12))))))):
ACSE_apdu
from:90     to:89       transition:40
guard:      selection_predicate:
actions:    assignments:c(10):=c(12)

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:95     to:94      transition:41
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:99     to:98      transition:42
guard:      selection_predicate:
```

```
actions:    assignments:

when:A?x(26):primitive
from:228    to:216      transition:43
guard:      selection_predicate:[IsARLSreq(x(26))]
actions:    assignments:

when:A?x(25):primitive
from:228    to:221      transition:44
guard:      selection_predicate:[IsAABRreq(x(25))]
actions:    assignments:



when:P?x(26):ACSE_apdu
from:228    to:147      transition:45
guard:      selection_predicate:[IsRLRQ(x(26))]
actions:    assignments:

when:P?x(25):ACSE_apdu
from:228    to:225      transition:46
guard:      selection_predicate:[IsABRT(x(25))]
actions:    assignments:

when:ir
from:232    to:256      transition:47
guard:      selection_predicate:
actions:    assignments:

when:A?x(2):primitive
from:20     to:9        transition:48
guard:      selection_predicate:[IsARLSrsp(x(2))]
actions:    assignments:

when:A?x(1):primitive
from:20     to:13       transition:49
guard:      selection_predicate:[IsAABRreq(x(1))]
actions:    assignments:

when:P?x(1):ACSE_apdu
from:20     to:17       transition:50
guard:      selection_predicate:[IsABRT(x(1))]
actions:    assignments:
```

```
when:A?x(9):primitive
from:89     to:82        transition:51
guard:      selection_predicate:[IsAABRreq(x(9))]
actions:    assignments:

when:P?x(10):ACSE_apdu
from:89     to:25        transition:52
guard:      selection_predicate:[IsRLRE(x(10))]
actions:    assignments:

when:P?x(10):ACSE_apdu
from:89     to:77        transition:53
guard:      selection_predicate:[IsRLRQ(x(10))]
actions:    assignments:

when:P?x(9):ACSE_apdu
from:89     to:86        transition:54
guard:      selection_predicate:[IsABRT(x(9))]
actions:    assignments:

when:ir
from:94     to:256      transition:55
guard:      selection_predicate:
actions:
assignments:

when:ir
from:98     to:256      transition:56
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLSind(reason(get_RLRQ(x(26))),
user_information(get_RLRQ(x(26))))):primitive
from:147     to:146     transition:57
guard:       selection_predicate:
actions:     assignments:c(16):=c(26)

when:P!ACSE_apdu(ACSE_apdu_genere_2(RLRQ_apdu
(reason(get_ARLSreq(x(26))),
user_info(get_ARLSreq(x(26))))):ACSE_apdu
from:216     to:215      transition:58
guard:       selection_predicate:
actions:     assignments:c(24):=c(26)
```

```
when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:221    to:220     transition:59
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:225    to:224     transition:60
guard:      selection_predicate:
actions:    assignments:

when:i
from:9    to:3         transition:61
guard:[eq(result(get_ARLSrsp(x(2))),accepted)]
selection_predicate:
actions:  assignments:

when:i
from:9    to:7         transition:62
guard:[eq(result(get_ARLSrsp(x(2))),rejected)]
selection_predicate:
actions:  assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:13     to:12      transition:63
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:17     to:16      transition:64
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLScnf(reason(get_RLRE(x(10))),
user_information(get_RLRE(x(10))),accept  .)):
primitive
from:25     to:24      transition:65
guard:      selection_predicate:
actions:    assignments:
```

```
when:A!primitive(ARLSind(reason(get_RLRQ(x(10))),
user_information(get_RLRQ(x(10))))):primitive
from:77     to:76        transition:66
guard:      selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:82     to:81        transition:67
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:86     to:85        transition:68
guard:      selection_predicate:
actions:    assignments:

when:A?x(16):primitive
from:146    to:135       transition:69
guard:      selection_predicate:[IsARLSrsp(x(16))]
actions:    assignments:

when:A?x(15):primitive
from:146    to:139       transition:70
guard:      selection_predicate:[IsAABRreq(x(15))]
actions:    assignments:

when:P?x(15):ACSE_apdu
from:146    to:143       transition:71
guard:      selection_predicate:[IsABRT(x(15))]
actions:    assignments:

when:A?x(23):primitive
from:215    to:208       transition:72
guard:      selection_predicate:[IsAABRreq(x(23))]
actions:    assignments:

when:A?x(24):ACSE_apdu
from:215    to:151       transition:73
guard:      selection_predicate:[IsRLRE(x(24))]
actions:    assignments:

when:P?x(24):ACSE_apdu
```

```
from:215    to:203      transition:74
guard:      selection_predicate:[IsRLRQ(x(24))]
actions:    assignments:

when:P?x(23):ACSE_apdu
from:215    to:212      transition:75
guard:      selection_predicate:[IsABRT(x(23))]
actions:    assignments:

when:ir
from:220    to:256      transition:76
guard:      selection_predicate:
actions:    assignments:

when:ir
from:224    to:256      transition:77
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(2))),
user_info(get_ARLSrsp(x(2))))):ACSE_apdu
from:3    to:2          transition:78
guard:    selection_predicate:
actions:  assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(2))),
user_info(get_ARLSrsp(x(2))))):ACSE_apdu
from:7    to:6          transition:79
guard:    selection_predicate:
actions:  assignments:

when:ir
from:12     to:256      transition:80
guard:      selection_predicate:
actions:    assignments:

when:ir
from:16     to:256      transition:81
guard:      selection_predicate:
actions:    assignments:

when:ir
```

```
from:24     to:256        transition:82
guard:      selection_predicate:
actions:    assignments:

when:i
from:76     to:58         transition:83
guard:[eq(c(10),called)] selection_predicate:
actions:    assignments:

when:i
from:76     to:74         transition:84
guard:[eq(c(10),calling)]selection_predicate:
actions:    assignments:

when:ir
from:81     to:256        transition:85
guard:      selection_predicate:
actions:    assignments:

when:ir
from:85     to:256        transition:86
guard:      selection_predicate:
actions:    assignments:

when:i
from:135     to:129       transition:87
guard:[eq(result(get_ARLSrsp(x(16))),accepted)]
selection_predicate:
actions:     assignments:




when:i
from:135     to:133       transition:88
guard:[eq(result(get_ARLSrsp(x(16))),rejected)]
selection_predicate:
actions:     assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
```

```
from:139    to:138      transition:89
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:143    to:142      transition:90
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLScnf(reason(get_RLRE(x(24))),
user_information(get_RLRE(x(24))),accepted)):primitive
from:151    to.150      transition:91
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLSind(reason(get_RLRQ(x(24))),
user_information(get_RLRQ(x(24))))):primitive
from:203    to:202      transition:92
guard:      selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:208    to:207      transition:93
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:212    to:211      transition:94
guard:      selection_predicate:
actions:    assignments:

when:ir
from:2    to:256        transition:95
guard:    selection_predicate:
actions:  assignments:

when:ir
from:6    to:102        transition:96
guard:    selection_predicate:
actions:  assignments:c(12):=c(2)
```

```
when:A?x(5):primitive
from:58    to:51        transition:97
guard:     selectic ı_predicate:[IsAABRreq(x(5))]
actions:   assignments:

when:P?x(6):ACSE_apdu
from:58    to:47        transition:98
guard:     selection_predicate:[IsRLRE(x(6))]
actions:   assignments:

when:P?x(5):ACSE_apdu
from:58    to:55        transition:99
guard:     selection_predicate:[IsABRT(x(5))]
actions:   assignments:

when:A?x(8):primitive
from:74    to:63        transition:100
guard:     selection_predicate:[IsARLSrsp(x(8))]
actions:   assignments:

when:A?x(7):primitive
from:74    to:67        transition:101
guard:     selection_predicate:[IsAABRreq(x(7))]
actions:   assignments:

when:P?x(7):ACSE_apdu
from:74    to:71        transition:102
guard:     selection_predicate:[IsABRT(x(7))]
actions:   assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x'16))),
user_info(get_ARLSrsp(x(16))))):ACSE_apdu
from:129     to:128       transition:103
guard:       selection_predicate:
actions:     assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(16))),
user_info(get_ARLSrsp(x(16))))):ACSE_apdu
from:133     to:132        transition:104
guard:       selection_predicate:
actions:     assignments:
```

```
when:ir
from:138     to:256        transition:105
guard:       selection_predicate:
actions:     assignments:

when:ir
from:142     to:256        transition:106
guard:       selection_predicate:
actions:     assignments:

when:ir
from:150     to:256        transition:107
guard:       selection_predicate:
actions:     assignments:

when:i
from:202     to:184        transition:108
guard:[eq(c(24),called)] selection_predicate:
actions:     assignments:

when:i
from:202     to:200        transition:109
guard:[eq(c(24),calling)]selection_predicate:
actions:     assignments:

when:ir
from:207     to:256        transition:110
guard:       selection_predicate:
actions:     assignments:

when:ir
from:211     to:256        transition:111
guard:       selection_predicate:
actions:     assignments:

when:A!primitive(ARLScnf(reason(get_RLRE(x(6))),
user_information(get_RLRE(x(6))),accepted)):primitive
from:47      to:46         transition:112
guard:       selection_predicate:
actions:     assignments:c(4):= called

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:51      to:50         transition:113
```

```
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:55     to:54          transition:114
guard:      selection_predicate:
actions:    assignments:

when:i
from:63     to:62          transition:115
guard:[eq(result(get_ARLSrsp(x(8))),accepted)]
selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:67     to:66          transition:116
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:71     to:70          transition:117
guard:      selection_predicate:
actions:    assignments:

when:ir
from:128     to:256        transition:118
guard:       selection_predicate:
actions:     assignments:

when:ir
from:132     to:228        transition:119
guard:       selection_predicate:
actions:     assignments:c(26):=c(16)

when:A?x(19):primitive
from:184     to:177        transition:120
guard:       selection_predicate:[IsAABRreq(x(19))]
actions:     assignments:

when:P?x(20):ACSE_apdu
from:184     to:173        transition:121
```

```
guard:      selection_predicate:[IsRLRE(x(20))]
actions:    assignments:

when:P?x(19):ACSE_apdu
from:184    to:181        transition:122
guard:      selection_predicate:[IsABRT(x(19))]
actions:    assignments:

when:A?x(22):primitive
from:200    to:189        transition:123
guard:      selection_predicate:[IsARLSrsp(x(22))]
actions:    assignments:

when:A?x(21):primitive
from:200    to:193        transition:124
guard:      selection_predicate:[IsAABRreq(x(21))]
actions:    assignments:

when:P?x(21):ACSE_apdu
from:200    to:197        transition:125
guard:      selection_predicate:[IsABRT(x(21))]
actions:    assignments:

when:A?x(4):primitive
from:46     to:35         transition:126
guard:      selection_predicate:[IsARLSrsp(x(4))]
actions:    assignments:

when:A?x(3):primitive
from:46     to:39         transition:127
guard:      selection_predicate:[IsAABRreq(x(3))]
actions:    assignments:

when:P?x(3):ACSE_apdu
from:46     to:43         transition:128
guard:      selection_predicate:[IsABRT(x(3))]
actions:    assignments:

when:ir
from:50     to:256        transition:129
guard:      selection_predicate:
actions:    assignments:

when:ir
```

```
from:54     to:256          transition:130
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(8))),
user_info(get_ARLSrsp(x(8))))):ACSE_apdu
from:62     to:61           transition:131
guard:      selection_predicate:
actions:    assignments:

when:ir
from:66     to:256          transition:132
guard:      selection_predicate:
actions:    assignments:

when:ir
from:70     to:256         transition:133
guard:      selection_predicate:
actions:    assignments:

when:A!primitive(ARLScnf(reason(get_RLRE(x(20))),
user_information(get_RLRE(x(20))),accepted)):primitive
from:173    to:172         transition:134
guard:      selection_predicate:
actions:    assignments:c(18):= called

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:177    to:176         transition:135
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:181    to:180         transition:136
guard:      selection_predicate:
actions:    assignments:

when:i
from:189    to:188         transition:137
guard:[eq(result(get_ARLSrsp(x(22))),accepted)]
selection_predicate:
actions:    assignments:
```

```
when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:193    to:192      transition:138
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,type_genere020
(Not_Present)):AABRind
from:197    to:196      transition:139
guard:      selection_predicate:
actions:    assignments:

when:i
from:35     to:29       transition:140
guard:[eq(result(get_ARLSrsp(x(4))),accepted)]
selection_predicate:
actions:    assignments:

when:i
from:35     to:33       transition:141
guard:[eq(result(get_ARLSrsp(x(4))),rejected)]
selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:39     to:38       transition:142
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:43     to:42       transition:143
guard:      selection_predicate:
actions:    assignments:

when:ir
from:61     to:89       transition:144
guard:      selection_predicate:
actions:    assignments:c(10):= calling

when:A?x(18):primitive
from:172    to:161      transition:145
```

```
guard:      selection_predicate:[IsARLSrsp(x(18))]
actions:    assignments:

when:A?x(17):primitive
from:172    to:165      transition:146
guard:      selection_predicate:[IsAABRreq(x(17))]
actions:    assignments:

when:P?x(17):ACSE_apdu
from:172    to:169      transition:147
guard:      selection_predicate:[IsABRT(x(17))]
actions:    assignments:

when:ir
from:176    to:256      transition:148
guard:      selection_predicate:
actions:    assignments:

when:ir
from:180    to:256      transition:149
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(22))),
user_info(get_ARLSrsp(x(22))))))):ACSE_apdu
from:188    to:187      transition:150
guard:      selection_predicate:
actions:    assignments:

when:ir
from:192    to:256      transition:151
guard:      selection_predicate:
actions:    assignments:

when:ir
from:196    to:256      transition:152
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(4))),
user_info(get_ARLSrsp(x(4))))))):ACSE_apdu
from:29     to:28       transition:153
```

```
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(4))),
user_info(get_ARLSrsp(x(4))))):ACSE_apdu
from:33     to:32         transition:154
guard:      selection_predicate:
actions:    assignments:

when:ir
from:38     to:256        transition:155
guard:      selection_predicate:
actions:    assignments:

when:ir
from:42     to:256        transition:156
guard:      selection_predicate:
actions:    assignments:

when:i
from:161    to:155        transition:157
guard:[eq(result(get_ARLSrsp(x(18))),accepted)]
selection_predicate:
actions:    assignments:


when:i
from:161    to:159        transition:158
guard:[eq(result(get_ARLSrsp(x(18))),rejected)]
selection_predicate:
actions:    assignments:

when:P!ABRT_apdu(acse_service_provider,
type_genere023(Not_Present)):ABRT_apdu
from:165    to:164        transition:159
guard:      selection_predicate:
actions:    assignments:

when:A!AABRind(acse_service_provider,
type_genere020(Not_Present)):AABRind
from:169    to:168        transition:160
guard:      selection_predicate:
actions:    assignments:
```

```
when:ir
from:187    to:215      transition:161
guard:      selection_predicate:
actions:    assignments:c(24):= calling

when:ir
from:28     to:256      transition:162
guard:      selection_predicate:
actions:    assignments:

when:ir
from:32     to:102      transition:163
guard:      selection_predicate:
actions:    assignments:c(12):=c(4)

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(18))),
user_info(get_ARLSrsp(x(18))))):ACSE_apdu
from:155    to:154      transition:164
guard:      selection_predicate:
actions:    assignments:

when:P!ACSE_apdu(ACSE_apdu_genere_3(RLRE_apdu
(reason(get_ARLSrsp(x(18))),
user_info(get_ARLSrsp(x(18))))):ACSE_apdu
from:159    to:158      transition:165
guard:      selection_predicate:
actions:    assignments:

when:ir
from:164    to:256      transition:166
guard:      selection_predicate:
actions:    assignments:

when:ir
from:168    to:256      transition:167
guard:      selection_predicate:
actions:    assignments:

when:ir
from:154    to:256      transition:168
guard:      selection_predicate:
actions:    assignments:
```

```
when:ir
from:158     to:228       transition:169
guard:       selection_predicate:
actions:     assignments:c(26):=c(18)
```

APPENDIX B

TEST CASES GENERATED FROM ACSE PROTOCOL

(TRANSITIONS ONLY)

t_1 = 2 7 13
t_2 = 3 5 11 18
t_3 = 1 4 8 16 24
t_4 = 1 4 10 17 25
t_5 = 1 4 9 15 23 34
t_6 = 3 6 12 20 28 37
t_7 = 3 6 12 21 29 38
t_8 = 3 6 12 19 27 36 47
t_9 = 1 4 9 14 22 31 41 55
t_{10} = 1 4 9 14 22 33 42 56
t_{11} = 3 6 12 19 26 35 44 59 76
t_{12} = 3 6 12 19 26 35 46 60 77
t_{13} = 1 4 9 14 22 32 39 49 63 80
t_{14} = 1 4 9 14 22 32 39 50 64 81
t_{15} = 1 4 9 14 22 30 40 52 65 82
t_{16} = 1 4 9 14 22 30 40 51 67 85
t_{17} = 1 4 9 14 22 30 40 54 68 86
t_{18} = 1 4 9 14 22 32 39 48 61 78 95
t_{19} = 1 4 9 14 22 32 39 48 62 79 96 31 41 55
t_{20} = 3 6 12 19 26 35 45 57 70 89 105
t_{21} = 3 6 12 19 26 35 45 57 71 90 106
t_{22} = 3 6 12 19 26 35 43 58 73 91 107
t_{23} = 3 6 12 19 26 35 43 58 72 93 110
t_{24} = 3 6 12 19 26 35 43 58 75 94 111
t_{25} = 3 6 12 19 26 35 45 57 69 87 103 118
t_{26} = 3 6 12 19 26 35 45 57 69 88 104 119 44 59 76
t_{27} = 1 4 9 14 22 30 40 53 66 83 97 113 129
t_{28} = 1 4 9 14 22 30 40 53 66 83 99 114 130
t_{29} = 1 4 9 14 22 30 40 53 66 84 101 116 132
t_{30} = 1 4 9 14 22 30 40 53 66 84 102 117 133
t_{31} = 1 4 9 14 22 30 40 53 66 84 100 115 131 144 52 65 82
t_{32} = 3 6 12 19 26 35 43 58 74 92 108 120 135 148
t_{33} = 3 6 12 26 35 43 58 74 92 108 122 136 148
t_{34} = 3 6 12 19 26 35 43 58 74 92 109 124 138 151
t_{35} = 3 6 12 19 26 35 43 58 74 92 109 125 139 152
t_{36} =1 4 9 14 22 30 40 53 66 83 98 112 127 142 155
t_{37} = 1 4 9 14 22 30 40 53 66 83 98 112 128 143 156

t_{38} = 3 6 12 19 26 35 43 58 74 92 109 123 137 150 161 73 91 107
t_{39} = 1 4 9 14 22 30 40 53 66 83 98 112 126 140 153 162
t_{40} = 1 4 9 14 22 30 40 53 66 83 98 112 126 141 154 163 3141 55
t_{41} = 3 6 12 19 26 35 43 58 74 92 108 121 134 146 159 166
t_{42} = 3 6 12 19 26 35 43 58 74 92 108 121 134 147 160 167
t_{43} = 3 6 12 19 26 35 43 58 74 92 108 121 134 145 157 164 168
t_{44} = 3 6 12 19 26 35 43 58 74 92 108 121 134 145 158 165 16944 59 76

APPENDIX C

EFSM CHART OF HDLC PROTOCOL

```
when:DSAP_in?DCONreq:signal
from:1    to:2       transition:1
guard:    selection_predicate:
actions:  assignments:

when:in1?SABM:signal
from:1  to: 3        transition:2
guard:  selection_predicate:
actions:             assignments:

when:DSAP_out!DCONind
from:3  to: 4        transition:3
guard:  selection_predicate:
actions:             assignments:

when:in1?SABM:signal
from:4 to: 3         transition:4
guard: selection_predicate:
actions:             assignments:

when:DSAP_in?DCONresp:signal
from:4 to: 5         transition:5
guard: selection_predicate:
actions:
assignments: REJ_sent := False;
             t_running := False;
             vs := 0,
             vr:= 0;
             vos := 0;
             tb := qnew;
             nos := 0;

when:in2!UA
from:5 to: 12         transition:6
guard: selection_predicate:
actions:              assignments:

when:DSAP_in?DCONreq:signal
```

```
from:4 to: 6          transition:7
guard: selection_predicate:
actions:
assignments: REJ_sent := False;
             t_running := False;
             vs := 0;
             vr:= 0;
             vos := 0;
             tb := gnew;
             nos := 0;

when:DSAP_out!DCONconf
from:6 to: 7          transition:8
guard: selection_predicate:
actions:              assignments:

when:in2!UA
from:  7 to: 12       transition:9
guard:   selection_predicate:
actions:              assignments:

when:in2!SABM
from:  2 to: 8        transition:10
guard:   selection_predicate:
actions:
assignments: ta := ta+1;
             T := NOW + T1;

when:is
from:  8 to:9         transition:11
guard:   selection_predicate:
actions:              assignments:

when:i
from:  9 to: 2        transition:12
guard: [not(ta=N2)]   selection_predicate:
actions:              assignments:

when:DSAP_out!DDIsind
from:  9 to: 1        transition:13
guard: [ta=N2]        selection_predicate:
actions:              assignments:

when:in1?UA:signal
```

```
from:  8 to: 10        transition:14
guard:                 selection_predicate:
actions:
assignments: T := 0;
             REJ_sent := False;
             t_running := False;
             vs := 0;
             vr:= 0;
             vos := 0;
             tb := qnew;
             nos := 0;

when:DSAP_out!DCONconf
from:  10 to: 12       transition:15
guard:                 selection_predicate:
actions:               assignments:

when:in1?SABM:signal
from:  8 to: 11        transition:16
guard:                 selection_predicate:
actions:
assignments: T := 0;
             REJ_sent := False;
             t_running := False;
             vs := 0;
             vr:= 0;
             vos := 0;
             tb := qnew;
             nos := 0;

when:in2!UA
from:  11 to: 36       transition:74
guard:                 selection_predicate:
actions:               assignments:

when:DSAP_out!DCONconf
from:  36 to: 12       transition:17
guard:                 selection_predicate:
actions:               assignments:

when:DSAP_in?DDTreq(data):signal
from:  12 to: 13       transition:73
guard:                 selection_predicate:
actions:
```

```
assignments: tb := enqueue(data,tb);
             ack := False;

when:i
from:  13 to: 39      transition:18
guard: [empty(tb) OR (vs=vos++k) ]
selection_predicate:
actions:              assignments:

when:i
from:  39 to: 12      transition:78
guard: [not(ack)]     selection_predicate:
actions:              assignments:

when:i
from:  13 to:40       transition:19
guard:  [empty(tb) OR (vs=vos ++ k ]
selection_predicate:
actions:              assignments:

when: i
from: 40  to: 66      transition: 105
guard:  [ack]    selection_predicate:
actions:              assignments:

when:in2!RR(vr)
from:  66 to:  12     transition:79
guard:      selection_predicate:
actions:              assignments:

when:i
from:  13 to: 14      transition:20
guard: [not(empty(tb) OR (vs = vos ++k ))]
selection_predicate:
actions:
assignments:  data := dequeue(tb);

when:in2!I_frame(vs,vr,data)
from:  14 to: 15      transition: 21
guard:    selection_predicate:
actions:
assignments: rtb(vs):= data;
             vs := vs++1;
```

```
when:i
from:  15 to: 13       transition:22
guard: [t_running]     selection_predicate:
actions:               assignments:

when:i
from:  15 to: 13       transition:23
guard: [not(t_running)] selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running:= True;



when:in1?RR(nr):signal
from:  12 to: 16       transition:24
guard:                 selection_predicate:
actions:               assignments:

when:i
from:  16 to: 41       transition:25
guard: [not(inside_r(vs,nr,vos))]
selection_predicate:
actions:               assignments:

when:i
from:  41 to: 12       transition:80
guard: [vos=vs]        selection_predicate:
actions:               assignments:

when:i
from:  16 to: 42       transition:26
guard: [not(inside_r(vs,nr,vos)]
selection_predicate:
actions:               assignments:

when:i
from:  42 to: 43       transition:81
guard: [not(vos=vs)]   selection_predicate:
actions:               assignments:
when:i

from:  43 to: 12       transition:82
guard: [t_running]     selection_predicate:
```

```
actions:                assignments:

when:i
from:  16 to: 44        transition:27
guard: [not(inside_r(ns,nr,vos)]
selection_predicate:
actions:

when:i
from:  44 to: 45        transition:83
guard:  [not(vos=vs)] selection_predicate:
actions:
assignments:

when:i
from:  45 to: 12        transition:84
guard:[not(t_running)]selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running:= True

when:i
from:  16 to: 17        transition:28
guard: [inside_r(vw,nr,vos)]
selection_predicate:
actions:
assignments: ta:= 0;
             T := 0;
             t_runnirg:= False;
             vos := nr;

when:i
from: 17 to: 12         transition:29
guard: [vos=vs]         selection_predicate:
actions:                assignments:

when:i
from:  17 to: 46        transition:30
guard: [not(vos=vs)]
selection_predicate:
actions:                assignments:

when:i
from:  46 to: 12        transition:85
```

```
guard: [t_running]
selection_predicate:
actions:                assignments:

when:i
from:  17 to: 47        transition:31
guard: [not(vos=vs)]    selection_predicate:
actions:                assignments:

when:i
from:  47 to: 12        transition:86
guard:[not(t_running]   selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running := True;

when:in1?I_frame(ns,nr,data):signal
from:  12 to: 18        transition:32
guard:                  selection_predicate:
actions:                assignments:

when:i
from:  18 to: 48        transition:33
guard: [not(ns=vr)]
selection_predicate:
actions:                assignments:

when:i
from:  48 to: 20        transition:87
guard: [RJ_sent]        selection_predicate:
actions:                assignments:

when:i
from:  18 to: 49        transition:34
guard: [not(ns=vr)]     selection_predicate:
actions:                assignments:

when:i
from:  49 to: 67        transition:88
guard: [ not(RJ_sent)]  selection_predicate:
actions:                assignments:

when:in2!REJ(nos++1)
from: 67  to: 20        transition:106
```

```
guard:                  selection_predicate:
actions:
assignments: nos := ns;
             REJ_sent := True;


when:i
from:  18 to: 19        transition:35
guard: [ns=vr]          selection_predicate:
actions:
assignments: nos:= ns;
             REJ_sent := False


when:DSAP_out!DDTind(data)
from:  19 to: 20        transition:36
guard:                  selection_predicate:
actions:                assignments: vr := vr++1;


when:i
from:  20 to: 50        transition:37
guard: [not(inside_r(vs,nr,vos))]
selection_predicate:
actions:                assignments:

when:i
from:  50 to: 22        transition: 89
guard: [vos=vs]
selection_predicate:
actions:                assignments: ack := True;

when:i
from:  20 to: 51        transition:38
guard: [not(inside_r(vs,vr,vos))]
selection_predicate:
actions:                assignments:

when:i
from:  51 to: 52        transition:90
guard: [not(vos=vs)]    selection_predicate:
actions:                assignments:

when:i
```

```
from:  52 to: 22        transition:91
guard: [t_running]      selection_predicate:
actions:                assignments: ack := True
```

```
when:i
from:  20 to: 53        transition:39
guard: [not(inside_r(vs,vr,vos))]
selection_predicate:
actions:                assignments:

when:i
from:  53 to: 54        transition:92
guard: [not(vos=vs)]    selection_predicate:
actions:                assignments:

when:i
from:  54 to: 22        transition:93
guard:[not(t_running)] selection_predicate:
actions:
assignments: ack := True;
             T := NOW + T1;
             t_running := True;

when:i
from:  20 to: 21          transition:4
guard:[inside(vs,nr,vos)]selection_predicate:
actions:
assignments: ta := 0;
             T := 0;
             t_running := False;
             vos := nr;

when:i
from: 21  to: 22          transition:41
guard: [vos=vs]           selection_predicate:
actions:                  assignments: ack := True;

when:i
from: 21  to: 55          transition:42
guard: [not(vos=vs)]      selection_predicate:
actions:                  assignments:
```

```
when:i
from: 55  to: 22          transition:94
guard: [t_running]        selection_predicate:
actions:                  assignments: ack := True;

when:i
from:  21 to: 56          transition:43
guard: [not(vos=vs)]      selection_predicate:
actions:                  assignments:

when:i
from:  56 to: 22          transition: 95
guard: [not(t_running)]   selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running := True;
             ack := True;

when:i
from:  22 to: 57          transition:44
guard: [(empty(tb) OR vs= vos++k)]
selection_predicate:
actions:                  assignments:

when:i
from:  57 to: 12          transition:96
guard: [not(ack)]         selection_predicate:
actions:                  assignments:

when:i
from:  22 to: 58          transition:45
guard: [[empty(tb) OR (vs = vos++k ]
selection_predicate:
actions:                  assignments:

when:i
from:  58 to: 68          transition:97
guard: [ack]              selection_predicate:
actions:                  assignments:

when:in2!RR(vr)
from:  68 to: 12          transition:107
guard:                    selection_predicate:
```

```
actions:                    assignments:

when:i
from:  22 to: 23            transition:46
guard: [not(empty(tb) OR ( vs=vos++k)]
selection_predicate:
actions:
assignments: data := dequeue(tb);
             tb := rest(tb);

when:in2!I_frame(vs,vr,data)
from:  23 to: 24            transition:47
guard:                      selection_predicate:
actions:
assignments: rtb(vs) := data;
             vs := vs ++1;

when:i
from:  24 to: 22             transition:48
guard: [t_running]           selection_predicate:
actions:                     assignments:

when:i
from:  24 to: 22            transition:49
guard: [not(t_running)]     selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running:= True;


when:in1?REJ(nr):signal
from:  12 to: 25            transition:50
guard:                      selection_predicate:
actions:                    assignments:

when:i
from:  25 to: 12            transition:51
guard: [not(ge(nr,vs)]      selection_predicate:
actions:                    assignments:

when:in2!REJ_rec
from: 25  to: 26            transition:52
guard: [ge(nr,vs)]          selection_predicate:
actions:                    assignments:
```

```
when:i
from:  26 to: 59          transition:53
guard: [not(inside_r(vs,nr,vos)]
selection_predicate:
actions:                  assignments:

when:i
from:  59 to: 28          transition:98
guard: [vos=vs]           selection_predicate:
actions:
assignments: T := 0;
             t_running:= False;
             i := nr;
             vr:= vr;
             rtb := rtb;



when:i
from:  26 to: 60          transition:54
guard: [not(inside_r(vs,nr,vos)]
selection_predicate:
actions:                  assignments:

when:i
from:  60 to: 61          transition:99
guard: [not(vos=vs)]      selection_predicate:
actions:                  assignments:

when:i
from:  61 to: 28          transition:100
guard: [t_running]        selection_predicate:
actions:
assignments: T := 0;
             t_running:= False;
             i := nr;
             vr:= vr;
             rtb := rtb;

when:i
from:  26 to: 62          transition:55
guard: [not(inside_r(vs,nr,vos)]
selection_predicate:
```

```
actions:                    assignments:

when:i
from:  62 to: 63            transition:101
guard: [not(vos=vs)]        selection_predicate:
actions:                    assignments:

when:i
from:  63 to: 28            transition:102
guard: [not(t_running)]     selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running:= True;
             T := 0;
             t_running:= False;
             i := nr;
             vr:= vr;
             rtb := rtb;

when:i
from:  26 to: 27            transition:56
guard: [inside_r(vs,vr,vos)]
selection_predicate:
actions:
assignments: ta := 0;
             T := 0;
             t_running := False;
             vos := nr;
when:i
from: 27  to: 28            transition:57
guard: [vos=vs]             selection_predicate:
actions:
assignments: T :=0;
             t_running := False;
             i := nr;
             vr := vr;
             rtb : rtb;

when:i
from:  27 to: 64            transition:58
guard: [not(vos=vs)]
selection_predicate:
actions:
assignments:
```

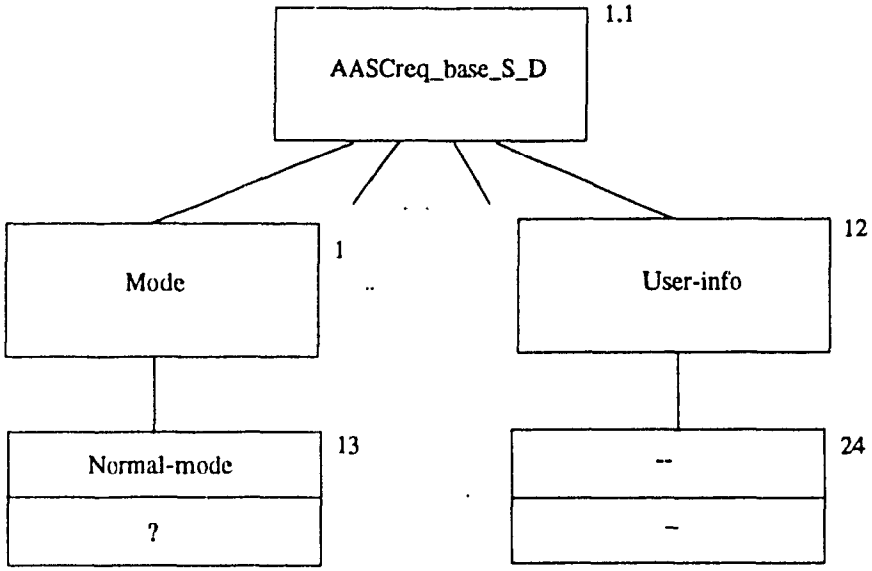
```
when:i
from:  64 to: 28          transition:103
guard: [t_running]        selection_predicate:
actions:
assignments: T := 0;
             t_running := False;
             i := nr;
             vr := vr;
             rtb : rtb;

when:i
from:  27 to: 65          transition:59
guard: [not(vos=vs)]      selection_predicate:
actions:                  assignments:

when:i
from:  65 to: 28          transition:104
guard: [not(t_running)]  selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running := True;
             T:= 0;
             t_running := False;
             i := nr;
             vr := vr;
             rtb := rtb;

when:i
from:  28 to: 29         transition:60
guard:                   selection_predicate:
actions:                 assignments: data := rtb(i);

when:in2!I_frame(i,vr,data):signal
from:  29 to: 30         transition:61
guard:                   selection_predicate:
actions:                 assignments: i := i+1;

when:i
from:  30 to:28          transition:62
guard: [not(i=vs)]       selection_predicate:
actions:                 assignments:

when:i
```

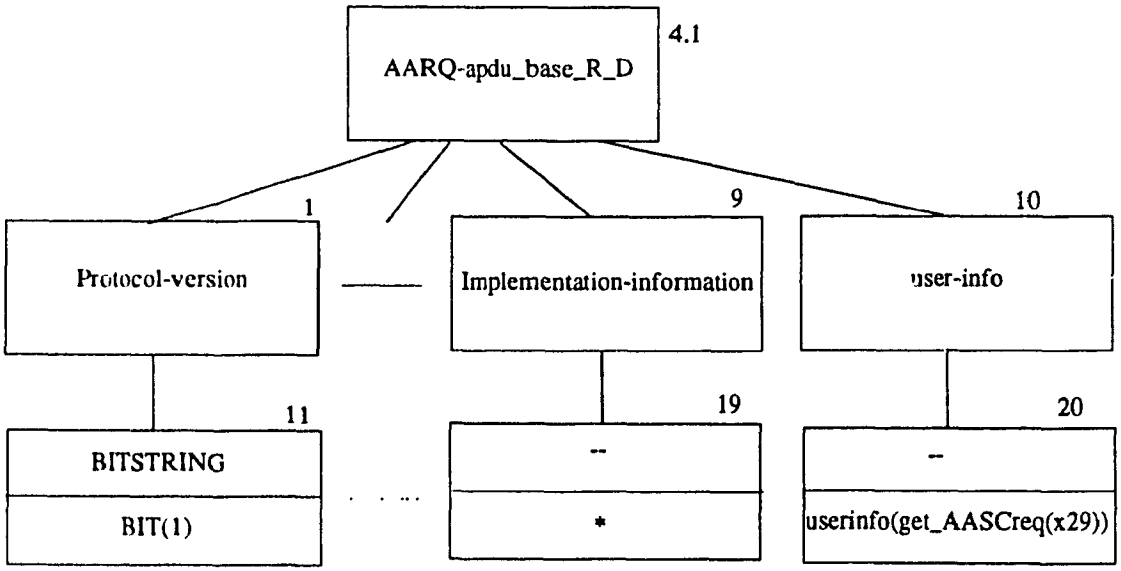
```
from:  30 to: 12          transition:63
guard: [i=vs]             selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running := True;


when:is
from:  12 to: 37          transition:64
guard:                    selection_predicate:
actions:
assignments: t_running := False;
             ta := ta+1;

when:i
from:  37 to: 38          transition:65
guard: [ta=N2]            selection_predicate:
actions:                  assignments: ta := 0;

when:DSAP_out!DDISind
from:  38 to: 1           transition:66
guard:                    selection_predicate:
actions:                  assignments:

when:i
from:  37 to: 33          transition:66
guard: [not(ta=N2)]       selection_predicate:
actions:
assignments: i := vos;
             rtb := rtb;
             vr := vr;

when:i
from:  33 to: 34          transition:67
guard:                    selection_predicate:
actions:                  assignments: data := rtb(i);

when:in2!I_frame(i,vr,data)
from:  34 to: 35          transition:68
guard:                    selection_predicate:
actions:                  assignments: i:= i++1;

when:i
from:  35 to:33           transition:69
```

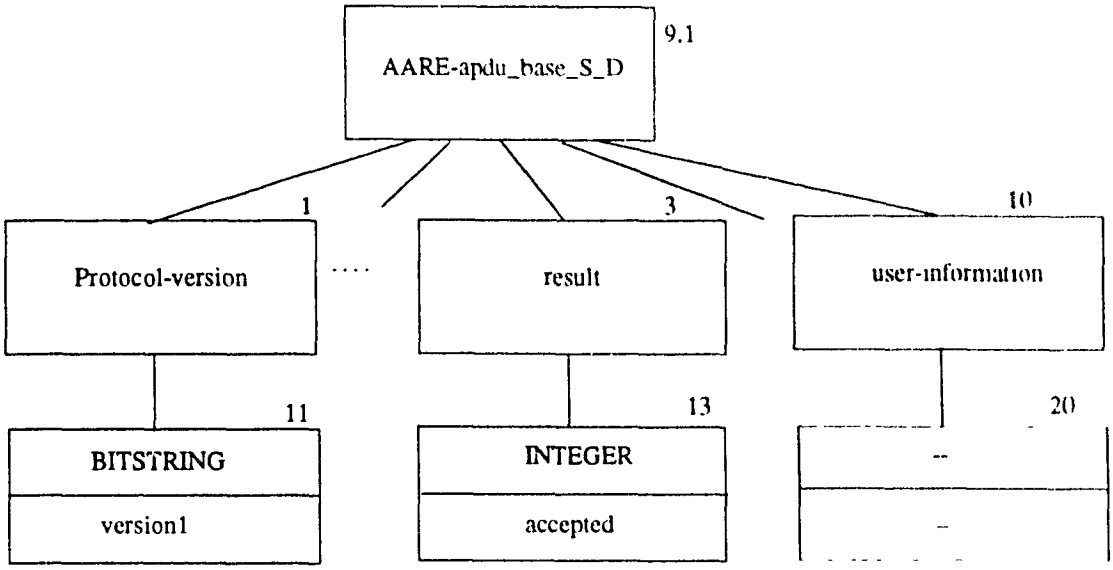
```
guard: [not(i=vs)]          selection_predicate:
actions:                    assignments:

when:i
from:  35 to: 12            transition:70
guard: [i=vs]               selection_predicate:
actions:
assignments: T := NOW + T1;
             t_running := True;

when:in1?SABM:signal
from:  12 to: 3             transition:71
guard:                      selection_predicate:
actions:                    assignments:

when:in1?DISC:signal
from:  12 to: 31            transition:75
guard:                      selection_predicate:
actions:                    assignments: ta:= 0;

when:in2!UA
from:  31 to: 32            transition:76
guard:                      selection_predicate:
actions:                    assignments:

when: DSAP_out!DDISind
from:  32 to: 1             transition:77
guard:                      selection_predicate:
actions:                    assignments:
```

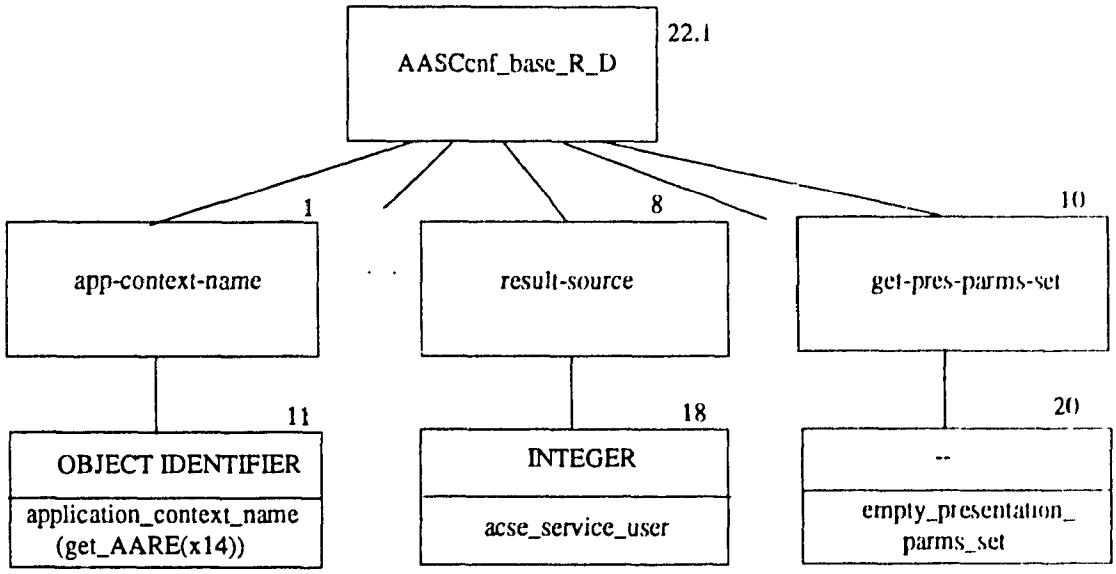
APPENDIX D

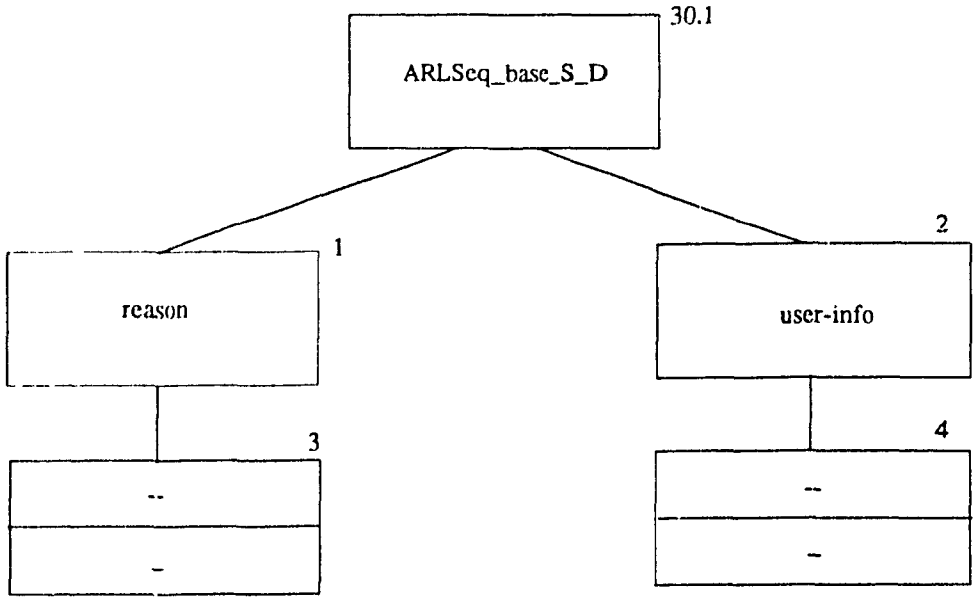
CONSTRAINTS GENERATED FOR THE TEST CASE T_{29}

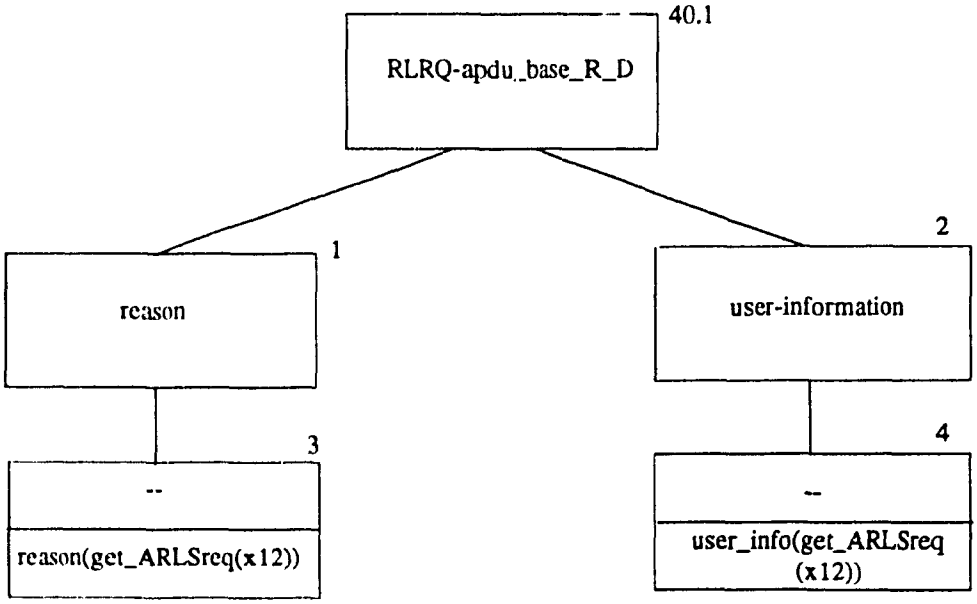


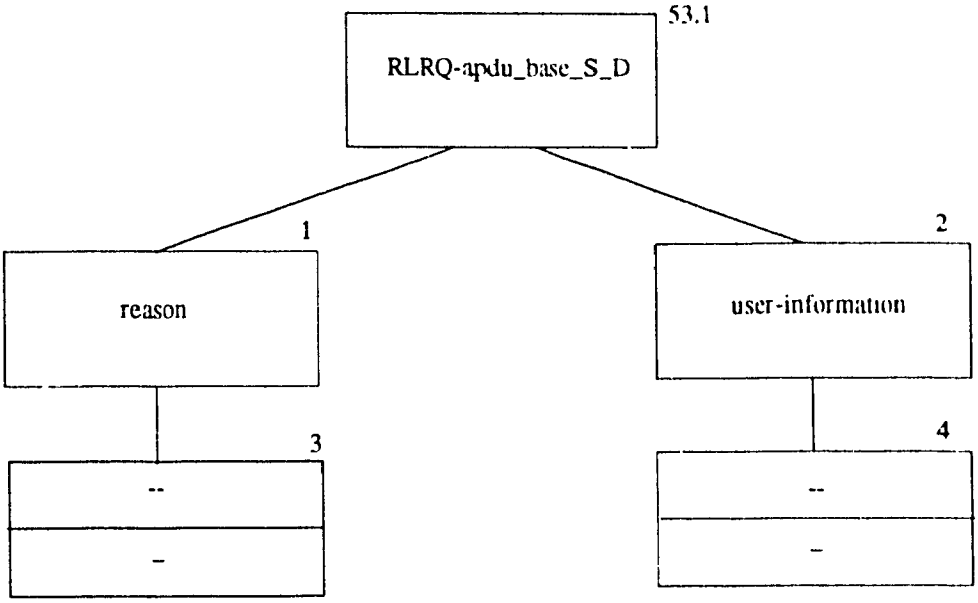


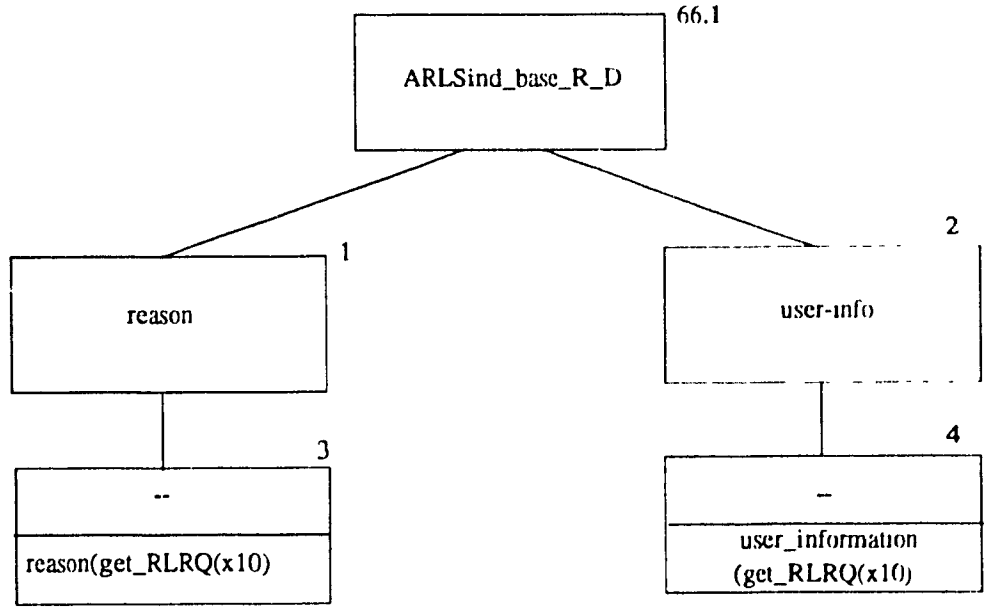

AARE-apdu_base_S_D
9.1
1
3
10
Protocol-version
result
user-information
11
13
20
BITSTRING
version1
INTEGER
accepted
--
--

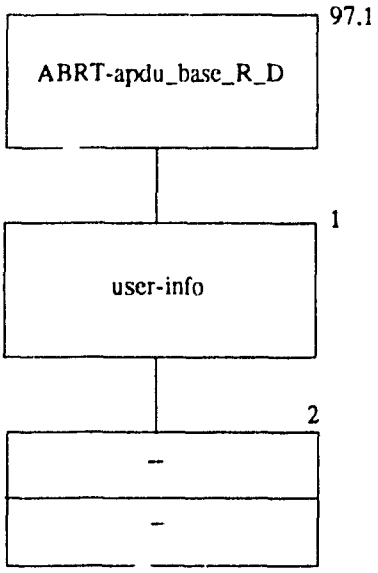

AASCcnf_base_R_D
22.1
1
8
10
app-context-name
result-source
get-pres-parms-set
11
18
20
OBJECT IDENTIFIER
application_context_name
(get_AARE(x14))
INTEGER
acse_service_user
--
empty_presentation_
parms_set

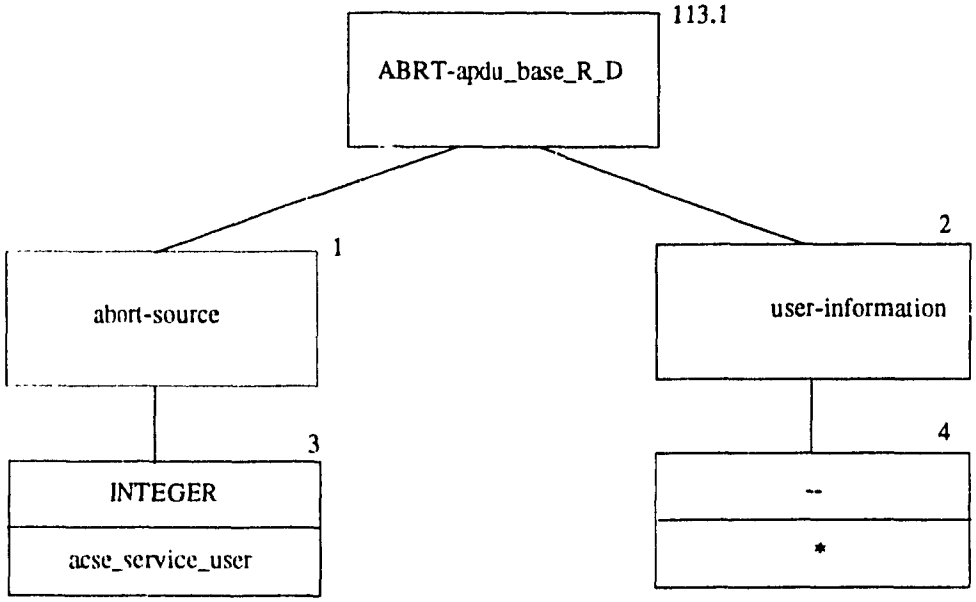

30.1
ARLSeq_base_S_D
1
reason
2
user-info
3
--
-
4
--
-


40.1
RLRQ-apdu_base_R_D
1
reason
2
user-information
3
--
reason(get_ARLSreq(x12))
4
--
user_info(get_ARLSreq (x12))


53.1
RLRQ-apdu_base_S_D
1
reason
2
user-information
3
--
-
4
--
-


66.1
ARLSind_base_R_D
1
reason
2
user-info
3
--
reason(get_RLRQ(x10)
4
--
user_information
(get_RLRQ(x10)


97.1
ABRT-apdu_base_R_D
1
user-info
2
--
-


113.1
ABRT-apdu_base_R_D
1
abort-source
3
INTEGER
acse_service_user
2
user-information
4
--
*

APPENDIX E

CONSTRAINTS GENERATED FOR THE TEST CASE T_{17}