On TESTGEN, An Environment for Protocol Test Sequence Generation, and Its Application to the FDDI MAC Protocol

By

YING LU

B.Sc. Tsinghua University, China, 1985
M.Sc. Peking Union Medical College, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1991

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date September 3, 1991

Abstract

Test suite generation and selection form important aspects of conformance testing. The test suite generation process is usually tedious and time-consuming. The selection of appropriate test cases with certain fault coverage requires intensive analysis of the protocol. It is difficult to manually generate test suites without errors. The test suite generation process must therefore be automated.

This thesis addresses the issues of design and development of the front end of an automatic test suite generation and selection environment named TESTGEN. TESTGEN generates TTCN test suites from the formal specification of protocols. TESTGEN adopts a new test suite generation method based on a combination of the extended transition system (ETS) and ASN.1 representation of protocols. The new test generation method integrates the testing of both control part and data part of protocols. The test suites generated by TESTGEN are expected to provide better fault coverage than other existing test generation methods.

An Estelle-like intermediate language called Estelle.Y is defined for the ETS-based formal description of protocols in TESTGEN. In order to test the data part, the control part of protocols and the protocol timers more thoroughly, we introduce ASN.1 and explicit timer constructs into Estelle.Y. In addition, we design a protocol data structure (PDS) as the internal representations of protocols based on the ETS/ASN.1 formalism. A parser is then designed and developed to translate the Estelle.Y/ASN.1 specification of protocols into the PDS. We test the parser and verify the correctness of the generated PDS by applying a set of consistency checking functions to PDS. The correctness of PDS can also be verified by printing out the protocol information represented in PDS using a set of printing functions.

In order to verify the viability of the TESTGEN environment, as well as to gain experience of conformance testing of high speed network protocols, we specify a specific protocol, the FDDI MAC protocol in Estelle.Y/ASN.1. The formal specification of the FDDI protocol in Estelle.Y and our experience with test sequence generation using TESTGEN are then presented.

Contents

Abstract ii

Contents iii

List of Figures vi

Acknowledgement vii

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 OSI conformance testing . . . 1
  - 1.1.2 Automatic test generation . . . 3
  - 1.1.3 FDDI . . . 3
- 1.2 Related works . . . 4
  - 1.2.1 Test generation method . . . 5
  - 1.2.2 Test generation tools . . . 6
- 1.3 Thesis objectives and contributions . . . 8
- 1.4 Thesis outline . . . 9

2 TESTGEN 11
- 2.1 Methodology . . . 11
  - 2.1.1 Extended Transition System + ASN.1 Formalism . . . 12
  - 2.1.2 Test generation method . . . 15
  - 2.1.3 Test generation constraints . . . 16
  - 2.1.4 Test suite generation engine . . . 17
- 2.2 TESTGEN components . . . 18

3 Estelle.Y and Protocol Data Structure (PDS) 20
- 3.1 motivation . . . 21
- 3.2 Estelle.Y . . . 22
  - 3.2.1 Estelle and ASN.1 . . . 22
  - 3.2.2 Design issues . . . 23

3.2.3 Estelle.Y definition . . . 24
3.2.4 Estelle.Y versus Estelle . . . 28
3.3 Protocol data structure . . . 30
3.3.1 Representation of protocol descriptors . . . 30
3.3.2 Protocol descriptor access . . . 31
3.3.3 PDS data structure definition . . . 33
3.3.4 ASN.1 type tree . . . 39
3.4 Summary . . . 43

4 TESTGEN parser **45**
4.1 Design considerations . . . 45
4.1.1 ASN.1 parser . . . 46
4.1.2 Lex/Yacc tools . . . 47
4.1.3 Abstract syntax tree . . . 48
4.2 Implementation . . . 50
4.2.1 Declaration part . . . 50
4.2.2 Initialization part . . . 52
4.2.3 State-transition part . . . 52
4.3 Testing . . . 53
4.3.1 Printing PDS . . . 53
4.3.2 Consistency checking of PDS . . . 54

5 Test generation for the FDDI MAC protocol **56**
5.1 FDDI MAC sublayer . . . 57
5.1.1 Services . . . 57
5.1.2 Facilities . . . 58
5.1.3 Operation . . . 59
5.1.4 Structure . . . 60
5.2 The formal specification of FDDI MAC protocol . . . 60
5.2.1 Data part in ASN.1 . . . 62
5.2.2 Control part in Estelle.Y . . . 62
5.3 Generating test suite using TESTGEN . . . 67
5.3.1 PDS of FDDI MAC protocol . . . 67
5.3.2 Tuning constraints for FDDI MAC protocol . . . 67
5.3.3 Test generation . . . 68
5.4 Summary . . . 69

6 Conclusions **70**
6.1 TESTGEN features . . . 70
6.2 TSG for FDDI using TESTGEN . . . 71
6.3 Future works . . . 72

A Estelle.Y BNF definition 79

B ASN.1 specification of FDDI MAC SPs and PDUs 86

C Excerpt of Estelle.Y Specification of FDDI MAC protocol 93

D TESTGEN menus 111

E Consistency Requirements of PDS 124

List of Figures

1.1 FDDI relationships to OSI model 4

2.1 Test Suite Generation Process 18

3.1 Example of the declaration part 26
3.2 Example of the initialization part 26
3.3 Example of a transition declaration in the state-transition part 27
3.4 Example of ASN.1 definition of protocol SPs 29
3.5 The main data structure of PDS 31
3.6 C structure definition of PDS 32
3.7 STATE definition 35
3.8 TRANS definition 36
3.9 VAR and CONST definitions 37
3.10 TIMER definition 37
3.11 The relation diagram of the PDS components 38
3.12 The data structure of template ENODE 40
3.13 ASN.1 type example 41
3.14 ASN.1 type tree of PduType 41
3.15 ISP definition 42
3.16 SPPARM definition 43

4.1 TESTGEN parser configuration 47
4.2 TESTGEN Parser generation 48
4.3 Examples of syntax trees 49
4.4 IFSTMT definition 49
4.5 EXPR definition 50

5.1 FDDI MAC service primitives 58
5.2 Format of MAC Protocol Data Units 59
5.3 Testing of FDDI MAC layer 61
5.4 FDDI MAC protocol state machine 67

Acknowledgements

First of all, I would like to express my sincere thanks to Dr. Son T. Vuong, my supervisor, for his guidance, commitment and understanding throughout my research work.

I would also like to thank Dr. Samuel Chanson for his helpful comments and careful reading of the final draft.

Special thanks are also due to Holger Janssen for his originating the research project and always finding time to help me when I needed it.

I would like to thank Mike Sample for his ASN.1 parser and helpful suggestions.

The financial support from the Department of Computer Science at the University of British Columbia and from the Japan Tobacco Company and INDE Electronics, Inc. in the form of an Research Assistantship is gratefully acknowledged.

Last but not least, a heartfelt thanks to my parents for their endless love and to Geng Lin for being my wonderful husband.

Chapter 1

Introduction

This thesis addresses issues in the development of an automatic protocol test generation and selection environment named TESTGEN and its applications. TESTGEN can automatically derive TTCN test suites from the formal description of protocols. TESTGEN generates test suites using a new test suite generation (TSG) method that integrates both the control flow testing and the data flow testing. This thesis presents the design and development of the TESTGEN front end and the application of TESTGEN to a high-speed network protocol, the FDDI MAC protocol. A formal specification of the FDDI MAC protocol is produced and problems with the formal specification for testing the FDDI MAC protocol are discussed. A TTCN test suite will be derived for the FDDI MAC protocol from the formal specification of the protocol by using TESTGEN.

1.1 Background

This section gives the general background for the thesis.

1.1.1 OSI conformance testing

As the OSI protocol standards become stable and their implementations proliferated, ISO focused its attention on the problem of interoperability of different protocol implementations. Consequencely the Conformance Testing Methodology and Framework [ISO-9646] standard is developed.

A protocol implementation can be submitted to conformance testing in order to increase confidence in its conformance to the protocol standard. The ISO 9646 standard states:

"The purpose of Conformance Testing is to increase the probability that different implementations are able to interwork".

In the conformance testing methodology, an Implementation Under Test (IUT) consists of the implementation of one or more layers of the OSI reference model. An IUT is said to **conform** to the OSI protocol standards if it fulfills the conformance requirements defined in the ISO 9646 standard.

The ISO test methods are based on the *black-box* test principle. Although aspects of both internal and external behavior are described in OSI protocol standards, only the external behaviors of the IUT is considered for conformance testing. The external behaviors of an (N)-entity are defined in terms of (N) Abstract Service Primitives ((N)-ASPs), (N-1) Abstract Service Primitives ((N-1)-ASPs) and (N) Protocol Data Units (PDUs).

The Points of Control and Observation (PCO) are where the test events (ASPs or PDUs) of an IUT can be observed and controlled within the test environment.

The ISO 9646 standard defines four test methods which can be distinguished by the location of the PCOs and the degree of control available at those PCOs as well as by the nature of the coordination between events exchanged at the PCOs (test coordination procedure).

Conformance testing is carried out by using test suites. A (conformance) test suite is the complete set of test cases that are needed to perform dynamic conformance testing for one or more OSI protocols. The ISO 9646 standard defines a hierarchical structure for description of test suites. The key level of a test suite is test case. The lowest level of the structure consists of test events. The test events are the transfer of a single PDU or ASP to or from the IUT. The Tabular and Tree Combined Notation (TTCN) [ISO-TTCN] is defined to specify abstract test suites. Abstract test suites are specific to a test method but test system independent.

1.1.2 Automatic test generation

The generation and selection of an appropriate test suite is an essential aspect of conformance testing. Communication protocols are extraordinarily complex. The selection and generation of test cases are challenging and tedious tasks for several reasons. Firstly, for a given protocol, a comprehensive test suite may contain from hundreds to thousands of test cases, where each test case may require hours to design and minutes to execute [PUH-88]. It is very difficult, if not impossible, to manually generate a test suite without errors [Wvong-90]. Secondly, the selection of appropriate test cases with certain fault coverage requires intensive analysis of a protocol. A promising solution for the problems is to develop an automated selective test generation environment. Automatic test suite generation has many advantages over the manual approach. It is easy to adapt to changes in protocol specifications and TTCN. More complete and consistent test cases can be generated. More flexibilities can be provided for the user to generate test suites for their particular testing purpose. Finally, certain fault coverage measure can be incorporated to allow user to control the fault coverage of generated test suites. For these reasons, the automatic generation of test suite from the formal descriptions of protocols is drawing more attentions.

1.1.3 FDDI

The Fiber Distributed Data Interface (FDDI) is a high speed local area network (LAN) standard developed by ANSI. FDDI is a general purpose multi-station network designed for efficient operations with a peak data rate of 100 megabits per second. It uses a Token Ring architecture with optical fibers as transmission medium [ANSI-1987] [ANSI-1989].

Although it is not an OSI standard, the FDDI standard has been developed in conformance with the the OSI reference model and the OSI management framework and layer management guidelines. The set of four components of the FDDI standard provides the Physical Layer and the lower sublayer of the Data Link Layer of the OSI reference model as depicted in Figure 1.1.

The FDDI Media Access Control (MAC) layer is the lower sublayer of the Data Link Layer. It is the most important component of the FDDI standard among the four (MAC, PHY, PMD

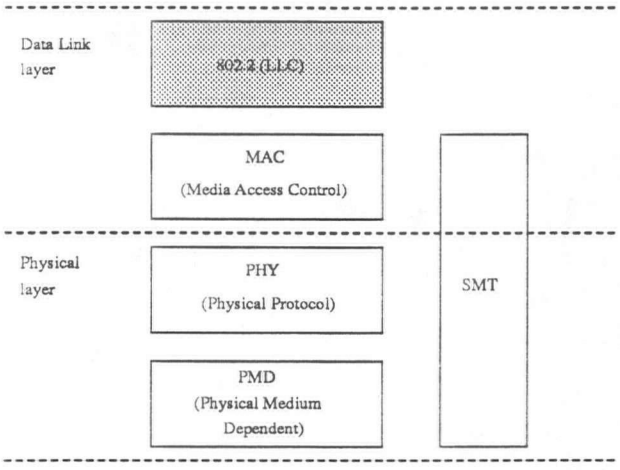


Figure 1.1: FDDI relationships to OSI model

and SMT). MAC standard is the core part of the FDDI standard that distinguishes FDDI from other IEEE LAN standards.

FDDI MAC protocol is developed based on the concepts of Token Ring Access Method defined in ANSI/IEEE 802.5-1985 while the original concepts have been modified to accommodate the higher FDDI speeds. [ROSS-86] [ROSS-90] give excellent introductions to FDDI.

High speed network is becoming an active research area recently. Major work has been done on the design of high speed (or high performance) protocols. FDDI has been widely accepted as the next generation of LAN standard. The FDDI network and its protocols are studied intensively [ROSS-86] [ROSS-90]. We are focusing our attention on the testing of FDDI implementations. The development of FDDI protocols are complicated due to its 100 Mb/s data rate, more rigorous timing requirements and the use of fiber optics techniques. It is important to test the FDDI products provided by different vendors for their conformance to the FDDI standard. Testing such complex protocols is a challenging problem.

1.2 Related works

A great deal of efforts have been made in development of feasible test generation method for automating the test generation process. In this section, we discusses the well known methods

and the tools developed based on those TSG methods.

1.2.1 Test generation method

Most of the well known protocol test generation methods assume the Finite State Machine (FSM) model for protocol specifications. T-method [Naito-81], U-method [Sabn-88], D-method [Gone-70] and W-method [Chow-78] are four well-known *formal techniques* for protocol testing based on the concept of Transition Tour and Characterizing Input/Output Sequence. See [Sidhu-89] for a detailed survey on FSM-based test sequence generation methods. However, these methods address the control part of protocols only. These formal methods were improved and optimized [Chan-89] [Vuong-89].

Recently, several test generation methods which test both of the control and data aspects of protocols have seen proposed. Sarikaya *et al.* [Sari-87] proposed a methodology that applies the concept of functional program testing to the generation of test sequences for testing data part of protocols. This method requires considerable manual efforts to identify functions and their relationships in the case of complex protocol specifications.

Another well known data flow coverage methods is proposed by Ural *et al.* [Ural-87-1] [Ural-87-2]. The method is based on the principles of data flow analysis techniques [Fosd-76]. The method traces the flow of data through the associations between assignments of values to variables (i.e. *definitions*) and the usage of these variables (i.e. *uses*) in either assigning values to other variables or determining the outcome of conditional branching. This method generates a set of test sequences to cover all definition and use pairs satisfying certain constraints. The resulting set of test sequences provides the capability of determining whether an IUT establishes the desired flow of data expressed in a given specification. The method is improved in [Ural-88] to derive test sequences with better fault coverage. The refined method is based on the identification of all inputs that influence each output in a given protocol specification. The flow of both control and data specified in the specification are modeled by a flowgraph. All associations between definitions and usages of variables employed in the specification are explicitly identified from the flowgraph. Associations between each output and those inputs that

influence the output are then identified. In fact, these associations represent the input-output relations through which protocol functions are defined. Finally test sequence are selected to cover each of such association at least once.

A new test generation method is presented in [EBE-89-2] [EBE-89-1]. An External Behavior Expression (EBE) is defined to specify only the external behaviors of a protocol. Test sequences are derived from the EBE specification of the protocols. The external behaviors of a protocol are described in terms of the input/output sequences and their logical (function and predicate) relations. The EBE specification of protocols can be obtained from formal protocol specifications in either Estelle or LOTOS.

It is pointed out [Vuong-91-1] that some side effects due to implementation errors may remain undetected by those existing methods [Sari-87] [Ural-87-1] [Ural-87-2] [Ural-88] [EBE-89-2] [EBE-89-1]. For example, in Ural's method [Ural-88] the conditions and effects on the variables are to be tested along only a single selected path between the definition-usage pairs, thus errors occurring in alternate paths are unlikely to be detected. The fault coverage and effectiveness of the test case will be increased if all known protocol conditions are verified along the definition-usage path of variables.

Another new approach is proposed in [Vuong-91-1] which is expected to produce test suite with a better fault coverage than those existing methods. The proposed test suite generation method verifies all specified conditions on the external behavior of a protocol implementation for a selected set of subtours of the protocol graph. A set of so-called test suite generation constraints are used to guide the subtour identification and test suite generation. A detailed description of this new approach can be found in Chapter 2 of this thesis.

1.2.2 Test generation tools

There are technical difficulties in deriving test sequences directly from the protocol specification in ISO FDTs, namely Estelle [ISO-9074], SDL and LOTOS [ISO-8807]. The first three methods which test both control and data parts of protocols assume that protocol specifications are given in Estelle as single module specifications (also called Normal Form Specifications). The EBE

approach assumes that protocol specifications are given in EBE.

A formal specification based test generation tool named CONTEST-ESTL is presented in [Sari-89]. It is a realization of the functional formal specification based on test generation method [Sari-87]. CONTEST-ESTL takes an Estelle specification as input and semi-automatically generate test sequences for the input specification. In CONTEST-ESTL, the normalization (transformation of an Estelle protocol specification into a specification in Estelle Normal Form) and the construction of control flow and data flow graphs required for test sequence generation are automated. The test sequence identification is, however, only semi-automated. The users are required to identify functions by merging so-called *blocks* of data flow graph and then incorporate the effects of the enabling conditions of the normalized transitions into the sequences identified from the control flow graph and data flow graph to generate test sequences that cover data flow.

In this thesis, we develop a software testing environment, TESTGEN, which automatically generates TTCN test suites from the formal specification of protocols, using the test generation method in [Vuong-91-1]. We assume protocols are specified in Estelle.Y. Estelle.Y is basically a single module Estelle enhanced by introducing explicit timers constructs and ASN.1 subset.

A Protocol Data Structure (PDS) is designed as the internal protocol representation from which all the possible subtours are identified by using the proposed test generation method. In order to generate test suite covering both control part and data part of protocols, the complete protocol information expressed in the Estelle.Y/ASN.1 specification must be represented by the PDS.

The test suite generation using TESTGEN is totally automated. Experienced users may use constraint editor to tune the default test suite generation constraints set by TESTGEN so that the generated test suite can better satisfy their particular testing purposes.

1.3 Thesis objectives and contributions

In this thesis, the research objectives are: 1) to design and develop the front end of the testing environment TESTGEN, and 2) to demonstrate the usefulness of the tool by applying it to a real world protocol and generate test suite for the protocol.

The front end of the testing environment is the most important part of the whole testing environment. It is responsible for providing internal representation of protocols in which the external behaviors of a protocol are precisely and completely described. The internal representation of protocols must be accessible to the subtour identification process which is the first step of the test generation process.

The design and development of TESTGEN front end involve several aspects. Firstly, an intermediate formalism is necessary for the precise and complete description of external behaviors of a protocol for conformance testing using TESTGEN. Secondly, a description language based on the defined intermediate formalism is required for formal specification of protocols. Thirdly, an internal representation of protocols is needed to allow the protocol knowledge described in the specification to be accessible to the subtour identification process which is an implementation of the test sequence generation algorithm. Finally a parser should be developed to transform the protocol specifications into their internal representations.

To demonstrate the viability of TESTGEN, we produce a formal specification of a real world protocol, the FDDI MAC protocol. The specification is then fed into TESTGEN. Constraints are tuned via a constraint editor. A test suite of the FDDI MAC protocol is to be generated.

The main contributions of this thesis include the following:

1. The refinement of the intermediate ETS/ASN.1 formalism for the representation of protocols in TESTGEN.

2. The definition of an intermediate formal description language Estelle.Y to allow communication protocols to be formally specified based on ETS/ASN.1 formalism representation and to allow both the control part and data part of protocols to be tested.

3. The design of a Protocol Data Structure (PDS) to internally represent communication protocols specified in Estelle.Y/ASN.1. PDS is defined as a machine accessible form of the ETS/ASN.1 formalism representation particularly for TESTGEN. It can also be used for other applications.

4. The design and implementation of a parser to translate the ETS/ASN.1 specifications of protocols into the PDS to allow the protocol knowledge in the Estelle.Y/ASN.1 specification to be accessed by the test sequence generation process in TESTGEN. We tested the parser through a set of consistency checking functions and through a set of printing functions, by checking items stored in the PDS.

5. The formal specification of a high-speed network protocol, the FDDI MAC protocol, in Estelle.Y/ASN.1 to which TESTGEN is to be applied to generate a test suite for the protocol.

6. A method of combining two extended transition systems into a single one equivalent behaviors. In fact, the method is easily extended to combine arbitrary number of extended transition systems.

1.4 Thesis outline

The rest of the thesis is organized as follows.

Chapter 2 gives an overview of TESTGEN. An introduction to the test generation methodology being used by TESTGEN and the architecture of the TESTGEN are presented.

Chapter 3 and chapter 4 discuss the design and implementation of the front end of the TESTGEN tool. The definition of a formal language Estelle.Y and the data structure version of the internal representation of protocols are described in Chapter 3. The design and implementation of the TESTGEN parser are presented in Chapter 4.

Chapter 5 discusses the issues of the test suite generation for FDDI mac protocol using TESTGEN. A review of the FDDI mac protocol is given first. The protocol is specified in

Estelle.Y and the problems with the formal specification are discussed. In this chapter, we present a method to combine two extended transition systems into a single one with equivalent behaviors. The issues of tuning default constraints for TSG of FDDI mac protocol are are presented.

Chapter 6 summarizes the important results and offers suggestions for future works.

Chapter 2

TESTGEN

The generation of TTCN test suite is a tedious and repetitious process. Test suites must often be updated or rewritten because both the specification of the protocol to be tested and the TTCN standard are subject to periodic modifications. The test suite generation process must therefore be automated. TESTGEN is a test generation and selection environment for the conformance testing of communication protocols proposed by Holger Janssen [Vuong-91-1] at UBC. It is a TSG automation tool which can derive TTCN test suites from formal specifications of protocols. It directly supports ASN.1 and Estelle.Y, a variation of Estelle that will be presented in Chapter 3.

It is designed as an automation tool for the testing of OSI protocols and services based on the OSI conformance testing methodology and framework [ISO-9646].

2.1 Methodology

This section presents the definition of the Extended Transition System + ASN.1 Formalism used in TESTGEN. The test generation method used in TESTGEN are also described.

2.1.1 Extended Transition System + ASN.1 Formalism

Motivation

In order to automatically generate a test suite, the protocol knowledge defined in the protocol specification must be formalized and accessible to the test generation engine.

The test suite generation is complicated by two factors: *choices* in the protocol specification account for protocol implementations with different but correct behavior and *nondeterminism* of the protocol accounts for unpredictable behavior of one implementation.

In order to preserve the complete protocol knowledge, an appropriate intermediate protocol representation formalism and its internal representation are necessary. The conceptual model is very crucial in that the test generation method and the architecture of the tool are both based on that model. We use a *pragmatic* representation based on extended Transition System (ETS) and ASN.1 which can represent nondeterminism [1] and implementation choices[2] as well as syntactical information such as type definition of service primitives and parameters.

This ETS+ASN.1 knowledge is stored in our Protocol Data Structure (PDS) for use of the test suite generation engine or other protocol engineering applications.

ETS + ASN.1 Formalism

The *Extended Transition System* (ETS) defined for TESTGEN environment provides a theoretical foundation for formal description of protocols to be tested by TESTGEN. A *transition system* is a model for formal description of processes in a distributed system. Recently, the transition system and its variations are popularly employed to model the behavior of communication protocols in protocol specification, verification and conformance testing.

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

- Q *is a set, the states of* T
- E *is a set, the events of* T,

[1] Nondeterminism in the protocol specification accounts for unpredictable IUT behavior.

[2] Choices in the protocol specification accounts for implementation with different correct behavior.

- $\rightarrow \subseteq Q \times E \times Q$ *is a relation, the* transitions *of* T,
- $init \in Q$ *is the* initial state *of* T.

For simplicity, $q \xrightarrow{e} q'$ will be used to represent $< q, e, q' > \in \rightarrow$.

An alternative definition of the transition system, so-called labeled transition system, is given in [Kel-76] by refining the notation of state and transition. The state set is represented as a set product of the set of *control states* and the set of *data states*. A transition $\xrightarrow{t}$ is represented by a pair $< P_t, F_t >$, where P_t is a predicate on Q and F_t is a partial function such that $F_t(q)$ is defined whenever $P_t(q)$ is true. P_t is called *enabling predicate* and F_t an *action function.*

We define an extended transition system by further refining the notation of state, transition, event and initial state of the basic transition system. Such an extended transition system can be particularly used to model the observable behaviors of a protocol for conformance testing based on the 9646 standard.

Definition 2.2 *An* Extended Transition System *(ETS) is a quadruple* $\mathbf{ETS} = (Q, E, \rightarrow, q_{init})$ *where*

- Q *is a set, the* states *of ETS,*
- E *is a set, the* events *of ETS,*
- $\rightarrow \subseteq Q \times E \times Q$ *is a relation, the* transitions *of ETS,*
- $q_{init} \in Q$ *is the* initial state *of ETS.*

$Q = STATE \times VAR \times C \times TIMER$, where $STATE$ is the set of *control states*, VAR is the set of *data states* also called *variables*, C is a set of degenerated variables used to represent protocol characteristics that are invariant to the protocol execution and $TIMER$ is a set of time constructs introduced to indicate time in the protocol representation.

q_{init} is defined by the initial control state and the initial values of all variables and timers.

$E = ISP \times OSP \times PDU$, where ISP is a set of Input Service Primitives (ISPs) accepted at the protocol's Service Access Points (SAPs), OSP is a set of Output Service Primitives (OSPs)

offered at the protocol's SAPs and PDU is a set of Protocol Data Units (PDUs) which may be embedded in ISPs or OSPs.

A transition $\xrightarrow{t}$ is represented by a pair $< P_t, F_t >$, where P_t is the *condition predicate* on the set product $Q \times E$ and F_t is the action function on the set product $Q \times E$.

A transition is enabled if and only if the ISP and PDU associated with the transition (if any) are received and if the enabling predicate is true. When a transition is executed the associated action function is executed atomically: variables and timers are set, OSP(s) and PDU(s) are assembled (their parameters are set) and sent. The protocol changes from the current control state into the next control state.

Furthermore, ASN.1 abstract syntax is introduced to specify the structure and type of the elements in E, namely the ISPs, OSPs and PDUs and the parameters. There are several reasons for choosing ASN.1:

- ASN.1 is a standardized abstract syntax notation supported by ISO.
- Some higher level protocols are specified in ASN.1.
- ASN.1 is supported in TTCN.
- There are ASN.1 tools available.

A subset[3] of the basic ASN.1 abstract type notation defined in Section 1 and Section 2 of [ISO-8824] is used to specify the data part of the protocol, namely the structure and type of ISPs, OSPs and PDUs and their parameters. In our Extended Transition System (ETS), service primitive and protocol data unit parameters would be referenced by enabling predicates and action functions. We introduced an intuitive dot notation similar to the PASCAL or C structured type reference mechanism to allow the references to the parameters of ISPs, OSPs and PDUs. Thus the SP or PDU parameters described in the ASN.1 part of the protocol specification can be referenced by enabling predicates and action functions as follows:

[3]Other ASN.1 features such as ASN.1 value descriptions, selection types and macros are not currently supported by our tools (they are not necessary for the ETS representation of a communication protocol).

$< SPname > \{. < parametername >\}^+$
or $< PDUname > \{. < parametername >\}^+$

A detailed description of ETS/ASN.1 formalism is given in [Vuong-91-1]

2.1.2 Test generation method

The test generation method adopted in the TESTGEN integrates both the control flow testing and the data flow testing with parameter variation. Furthermore, test generation and selection are integrated and guided by user-defined test suite generation constraints and parameter variation constraints.

Well known hardware and software test generation methods have been applied to conformance testing with various degrees of success. Most of them are based on the state transition model. The Transition Tour and characterizing Sequences based T [Nai-81], U [Sab-88], D [Gon-70] and W-methods [Cho-78] were improved and optimized [Cha-89] [Vuo-89] but still have two major shortcomings. First, they are weak in discovering errors due to additional states or transitions in the implementation. Second, they only address the control part of protocols.

The data flow analysis based test suite generation method defined in [Ura-87], [EBE-89-1] and the flow coverage method defined in [Sar-87] address both the control flow part and the data part of protocols but are unlikely to detect errors due to side effects [Vuong-91-1]. As those errors are unpredictable, the only way to ensure their detection is to verify if all relevant conditions on the external behavior of the IUT are fulfilled along all branches of the tree representation of the protocol.

The strategy adopted by TESTGEN is to verify as many conditions as possible on as many different protocol subtours as possible. The Test Suite Generation (TSG) method used in TESTGEN integrates both the control flow testing and the data flow testing with parameter variation and can produce test cases to cover any path defined by the service primitives that an

IUT is allowed to exchange with its environment (peer entity or test system). Furthermore, it integrates the generation and selection of test suites by providing a menu driven environment where various TSG- and parameter variation constraints[4] can be interactively defined by the user to control the amount and the distribution[5] of test cases.

2.1.3 Test generation constraints

Since communication protocols are recursive, some protocol subtours can be of infinite length. Parameter variation on the exchanged service primitives leads to a practically infinite number of parameter value combinations. The TSG constraints mechanism enables the user to select the subtours for which test cases are to be generated. The flexibility offered by the TSG constraints mechanism allows us to compare and evaluate different approaches to the test suite generation problem.

The TSG-constraints define an upper and a lower bound on the number of times an ETS element is reached or used in one subtour. For example, to test the data part of a protocol implementation the user could specify the following constraints: the data transmission state can be reached between 2 and 10 times and the send_data and receive_data service primitives can be used used for at most 5 times.

The TSG-constraints are set to default values when a PDS is created. A user-interactive constraint editor allows to edit and change the complete set of TSG constraints. The interested readers may refer to Appendix D for the menu-driven TSG-constraint editor.

The parameter variation constraints define a set of values for each parameter of each ISP or PDU that can be sent to the IUT thus define the ISPs and PDUs that will be used to test the IUT.

The current version of TESTGEN supports three SP or PDU parameter types: boolean, integer and character-string. The default parameter variation constraints are defined as follows:

TRUE, FALSE for boolean,

[4]It should be noted that the term "constraint" is used in a different context than the one used in TTCN.

[5]The density of test cases can be increased for important or error prone protocol parts.

0, 99 for integer and

"test-string1" for character strings.

As an example, the default parameter constraint scheme would define twelve ($3 \times 2 \times 2 \times 1$) different instances of a service primitive containing one integer parameter, two boolean parameters and one character string parameter. Each of those ISPs will be sent to the IUT in all subtours that fulfill all the TSG and parameter variation constraints.

The ASP parameter-editor allows to edit and change the parameter variation constraints within the limits defined by the ASN.1 definition of the parameter types.

2.1.4 Test suite generation engine

Given the PDS representation of a communication protocol and a set of constraints for this PDS, the test suite generation engine identifies and stores all the subtours and derives a TTCN test case for each subtour.

The subtour identification function performs an exhaustive search by means of a backtracking depth first search over the tree representation of the protocol to be tested. A test-branch of this tree is said to be *valid* if and only if the subtour associated with this branch fulfills all the TSG constraints.

For each ETS elements, e.g. states, transitions, ..., there are two associated global parameters, "MAXUSE" and "MINUSE". "MAXUSE" limits the value range of the *max_reached* and *max_used* constraints. These constraints limit the length of each valid test-branch to be < MAXUSE $\times$#states. In each state only a finite number of transitions can be applied (according to the protocol definition). The parameter variation constraints on the parameter of the service primitives exchanged in each transition limits the number of different instances of the service primitive in that transition. Thus the length and the number of different valid test-branches are kept finite so that the backtracking algorithm is guaranteed to terminate. "MINUSE" limits the value range of the *min_reached* and *min_used* constraints. The default value for "MAXUSE" constraints is 99. For "MINUSE" constraints the default value is 0.

Elaborate constraints may be necessary to reduce the number of test cases that can be

generated for a complex protocol (e.g. for the FDDI MAC protocol). Adequate constraints are likely to be determined in a trial and error process.

2.2 TESTGEN components

The major components of TESTGEN are depicted in Figure 2.1.

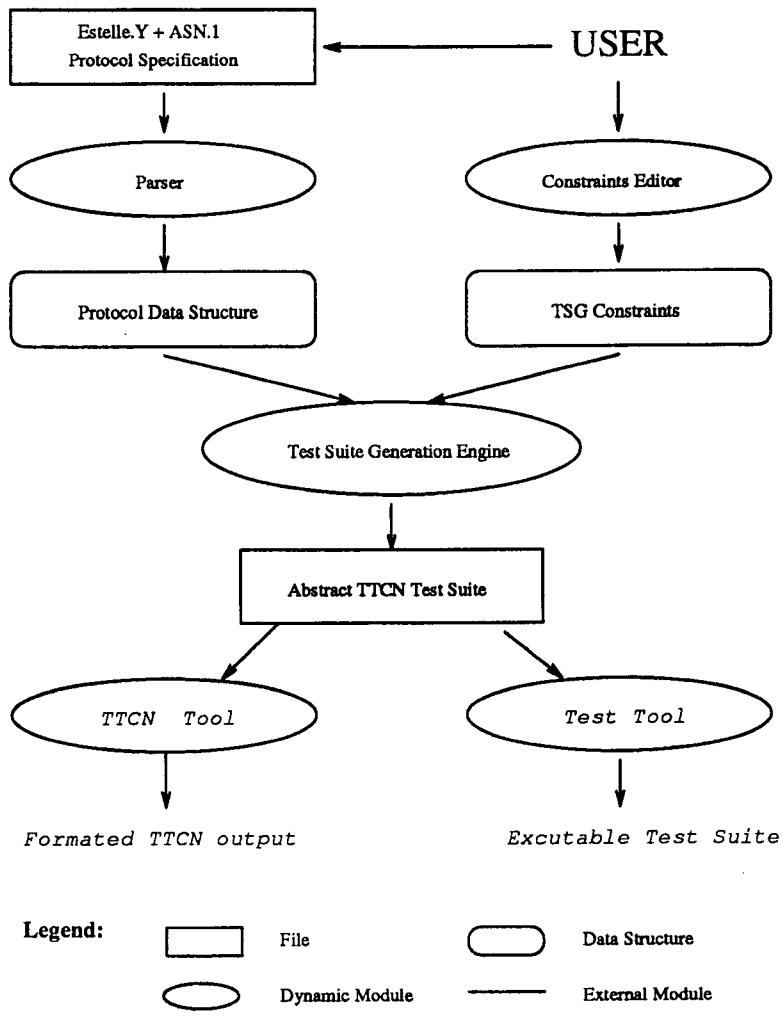


Figure 2.1: Test Suite Generation Process

The **parser** accepts the formal Estelle.Y/ASN.1 protocol description and translates the

protocol specification into a Protocol Data Structure (PDS) which serves as an internal representation of the combined extended transition system (ETS) and ASN.1 formalism. The Test Suite Generation (TSG) and parameter variation constraints are set to default values by TESTGEN automatically. The default constraints can be modified by the user interactively through the **constraints editor**. Based on the PDS representation of a protocol and the TSG constraints the **test suite generation engine** identifies all the subtours of the combined extended transition system and ASN.1 formalism representation of the protocol that fulfill the TSG Constraints and generates an abstract TTCN test case for each of the subtours identified.

A TTCN tool is used to edit and print the generated test suites and any TTCN supporting test tool can translate the abstract TTCN test suites to executable test suites in order to run them. Appropriate existing TTCN and test tools can be interfaced or incorporated to our environment to serve those purposes.

TESTGEN tool is menu-driven. The parser, constraint editor and the test suite generation engine are three major functions. Through the main menu the user can load a Estelle.Y/ASN.1 protocol specification into the system and call the parser to generate PDS from the protocol specification. A set of default constraints are set for the protocol and stored in the generated PDS automatically. Then the user can look at any of the items stored in the PDS through the PDS verification menu to check if the protocol information is properly represented in the PDS. The constraint editor is implemented as a menu-driven function. The user can set more TSG and SP parameter constraints or modify the existing TSG and SP parameter constraints through the constraint menu. Finally the user can call the test suite generation function to generate TTCN test cases through TSG menu. The user can view all the subtours identified by the subtour identification process or all the TTCN test cases generated from the identified subtours. The complete TESTGEN menu can be found in Appendix D.

The generation of PDS is a key step for test suite generation using TESTGEN. The design of the PDS is of particular importance since all subsequent steps make use of this PDS.

Chapter 3

Estelle.Y and Protocol Data Structure (PDS)

Since protocol specifications described in natural language often contain ambiguities, protocol implementations based on such specifications are likely to be incompatible. The interoperability of two implementations based on the same protocol standard is thus not guaranteed. On the other hand, the process of deriving test suites directly from informal specifications does not seem possible. The test suites generated automatically must be derived from the formal specification of protocols. A formal language that can fully describe the external behaviors of a protocol based on the ETS + ASN.1 formalism is therefore necessary for TESTGEN.

As there is no standardized formal protocol description for most standardized protocols, the choice of which Formal Description Technique (FDT) to be supported was open. After consideration of LOTOS [ISO-8807], Estelle [ISO-9074], SDL [SDL-88], CRS [CRS-89] and EBE [EBE-89-2] and Estelle Normal Form Specification (NFS) [Sari-87] we decided to use an Estelle-like formal language because Estelle is based on a conceptual model similar to the ETS+ASN.1 formalism. Furthermore, Estelle is supported by ISO and is more "user-friendly" than LOTOS. Also many protocol specifications in Estelle are available.

3.1 motivation

There are technical difficulties to derive test cases directly from the ISO FDTs such as LOTOS [ISO-8807], Estelle [ISO-9074] or SDL [SDL-88]. The existing test generation methods which generate test suite covering both control part and data part of protocols are mostly based on the Estelle Normal Form Specification (NFS) of protocols. EBE is another intermediate formal description mechanism which is dedicated to describe only the external behaviors of protocols.

Estelle.Y is defined for a similar reason. We are not intended to define a new FDT. However, we need an intermediate formal specification of protocols based on ETS/ASN.1 formalism representation of protocols. Estelle.Y is defined to be directly used with ASN.1. This is motivated by two facts. First, ASN.1 has been widely used to specify the protocol data types for higher layer protocols. The protocol data types of some other protocols such as the FDDI SMT protocol are specified in ASN.1 in the protocol standard as well. Second, ASN.1 is supported in TTCN. Since TESTGEN generates TTCN test suites, ASN.1 can be consistently used to specify the structure and data types of ISPs, OSPs and PDUs throughout the TSG process.

Furthermore, protocol timers are important for conformance testing in that they have non-trivial impact on the observable behaviors of a protocol. Timers must therefore be tested. To facilitate the testing of protocol timers, we provide explicit mechanism for specification of timers in Estelle.Y.

The specification of protocols in Estelle, SDL or CRS will be easily transformed to Estelle.Y specifications.

NFS and EBE are not suitable for TESTGEN because they are not defined to be used directly with ASN.1 and they do not explicitly support timers. Furthermore, NFS is not designed to be directly used to specify complex protocols. An NFS representation of a complex protocol is unreadable. Compared to NFS, Estelle.Y is more flexible since it supports more Pascal statements such as the conditional statements and loop statements. The Estelle.Y/ASN.1 specification tends to be more concise and readable than that in NFS.

In order to generate a test suite automatically, the syntactical and semantical information

of a protocol defined in the protocol specification must be accessible to the test generation engine. The Protocol Data Structure (PDS) is designed to be the machine accessible form of the ETS/ASN.1 based formal protocol specification. It holds both the control part and the data part of a protocol specification. The following sections introduce the definition of an intermediate formal description language Estelle.Y and the PDS for TESTGEN.

3.2 Estelle.Y

Estelle.Y is a formal language defined to specify protocols for automatic test suite generation. It is designed to be used with ASN.1, since the ETS+ASN.1 formalism is being used by TESTGEN. An Estelle.Y specification is a modified, single module Estelle specification enhanced by introducing explicit language support for timers and for structure references to SPs and PDUs and references to their parameters.

The structure of SPs and PDUs and the data type of their parameters are specified in ASN.1. The ASN.1 specification of SPs and PDUs is provided in a separate file along with the protocol specification. ASN.1 specifications are parsed to the ASN.1 type tree by the ASN.1 parser. The Estelle.Y specifications are parsed to the PDS by the TESTGEN parser. Also the TESTGEN parser incorporates the ASN.1 type tree into the PDS.

3.2.1 Estelle and ASN.1

Estelle is a formal description technique (FDT) standard developed by ISO. It is a formal specification language designed for the specification of communication protocols and services. It is based on the Extended Finite State Machine (EFSM) model. An Estelle specification describes a protocol and the services as a hierarchically structured system of non-deterministic sequential components (*instance of modules*) interchanging messages (called *interactions*) through bidirectional links between their ports (called *interaction points*) [ISO-9074]. It may be considered as an EFSM based extension of Pascal language. Estelle is implementation-oriented in that it is designed to specify both the internal and external behaviors of a protocol. Although it is designed to specify ISO protocols particularly, protocols standardized by other organizations

may also be specified in Estelle.

Abstract Syntax Notation One (ASN.1) is a notation or language for the definition of complicated data types and their values without determining the way and instance of this type is to be represented during transfer. ASN.1 is standardized by ISO [ISO-8824]. The ASN.1 language is described using Backus Naur Form (BNF). The ASN.1 definition of data structures or data types are very similar to those in the programming languages like Pascal or C. ASN.1's notation is similar to most programming languages in that it contains a set of simple built-in types, a set of rules for constructing programmer defined types and a mechanism to set the values of these types. ASN.1 has been widely accepted as a formal language to specify the data structure for higher level OSI protocols.

ASN.1 is also used in TTCN to define data structures of protocol service primitives and protocol data units.

3.2.2 Design issues

Estelle.Y is designed to formally specify the protocols and services for automatic test suite generation based on the ETS + ASN.1 formalism. As a formal intermediate notation for conformance testing, it should be able to describe the external behaviors of protocols precisely, completely and unambiguously but should not complicate the parser.

Estelle contains some features which are not necessary for the conformance testing based on the ETS+ASN.1 formalism. TESTGEN generates test suites for protocols based on a single extended finite state machine representation of protocols. Estelle.Y is then defined as an Estelle variation to support only one module. The Estelle language constructs for specifying interactions between different modules and the mechanism for creating instances for the modules are not supported by Estelle.Y.

In an Estelle specification, the data types of protocol variables and constants as well as SPs and PDUs are specified in Pascal.

We use Pascal to describe the data types of protocol variables and constants in Estelle.Y specification. The data structure and the data type of the SPs and PDUs and their parameters

are defined in ASN.1. The SP parameters or PDU fields defined in ASN.1 may be referenced in Estelle.Y specification through a dot notation mechanism similar to that a structure or record field is referenced in programming languages C or Pascal.

We consider timers as an important part of a protocol. Timers are also key issues for the conformance testing. Estelle does not provide explicit language support to specify timers. Estelle.Y supports timers explicitly.

3.2.3 Estelle.Y definition

The conceptual model

The ETS+ASN.1 formalism defined in section 2.1 is the conceptual model which defines the semantics of an Estelle.Y specification. The syntax of an Estelle.Y specification is similar to that of an Estelle specification of the single module.

The set of state Q is represented by control states, data states, constants and timers. Control states are a set of control values associated with the special identifier STATE. The data states are represented by values of variables, constants and the status of timers. These ETS elements are declared in the *declaration* part. q_{init} is defined in the *initialization* part. The observable behaviors of modules specified in terms of *the set of events* E are the effect of the module activity as described within an module body definition. *The set of transition* $\rightarrow$ is specified in the *state – transition* part. Each transition is syntactically composed of two parts: a clause group and a transition block. A clause group defines the enabling predicate of a transition. The clauses also define the control state from which a transition may take place and specify the next control state following the transition's execution. The events associated with a transition are also defined by the clauses. The transition block defines the action function to be executed by the transition.

Syntax and semantics

The syntax of Estelle.Y is defined in Backus-Naur Form (BNF) notation. The complete BNF definition of Estelle.Y is in appendix A. In its present stage, an Estelle.Y specification is a single

module definition. It contains three major parts: the declaration part, initialization part and the state-transition part. The protocol state machine is initialized in the initialization part. The state-transition part is to define the transitions of the protocol state machine.

- Declaration part

  The syntax of Estelle.Y variable and constant declarations are similar to those in Pascal. Estelle.Y supports three data types for the variables and constants: integer, boolean and character string. Timeout values are declared for Timers. PCOs are declared for ISPs and OSPs, and PDUs in case no embedding SPs are declared for PDUs. Embedding SPs or PCOs are declared for PDUs. Figure 3.1 is an example of the declaration part.

- Initialization part

  The initial states of the module are specified by the initialization part. The variables and timers may be assigned their initial values in this part. They will be assigned default values if not explicitly initialized.

- State-transition definition part

  The transitions are specified in this part. Figure 3.3 gives an example of a transition declaration. The clause group (FROM, To, WHEN, PROVIDED, etc) specify the present state and the next state of a transition, sending and receiving of the SPs, the priority and the enabling predicate. The enabling predicate is specified in Pascal as a boolean expression.

  The action function is specified as a group of Pascal statements. Four Pascal statements are supported. They are the assignment statement, if statement, while statement and compound statement. Moreover a set of timer statements are supported to specify the operations on timers.

  A transition is firable if the enabling predicate is satisfied, an ISP, in which PDU may be embedded, is received and the protocol is in the right control state. A transition can fire only if it is firable and it has the highest priority among the firable transitions.

```
Specification FDDI_1_mac;

CONST
    yes = true:     boolean;
    high   = 20:    int;
    NULL_STR = "":  char_str;
VAR
    Ring_Operational:    boolean;
    Token_class:    int;
    frame_INFO:    char_str;
ISP
    PhUnitDataIndication    mac_phy;
OSP
    PhUnitDataRequest    mac_phy;
PDU
    Frame   sent_in PhUnitDataRequest,
            recv_in PhUnitDataIndication;
TIMER
    TVX     2350;
STATE
    Rx_data, Ck_frame;

...
```

Figure 3.1: Example of the declaration part

```
INITIALIZATION

  To Rx_data
  begin
      Ring_Operational := true;
      Token_class := 1;
      frame_INFO := "";
  end;
```

Figure 3.2: Example of the initialization part

```
TRANS

    FROM    Rx_data
    TO     Rx_data
    WHEN        PhUnitDataIndication
    PROVIDED (PhUnitDataIndication.phIndication = I)
               and (not Reset(TVX))
    PRIORITY high
    OUTPUT  PhUnitDataRequest
      BEGIN
        Reset(TVX);
        Start(TVX);
        PhUnitDataRequest.phRequest := I;
      END;
```

Figure 3.3: Example of a transition declaration in the state-transition part

Language extension for timers

Estelle.Y provides explicit language supports for the specification of timers. Timer expressions reflecting the current status of a timer are also supported. The timer expressions are:

- RESET(timer-name),
- STARTED(timer-name),
- STOPPED(timer-name) and
- READ(timer-name).

The first three are boolean expressions. The last one gives the current value of a timer. Four timer statements are supported to specify the operations on timers. They are

- RESET(timer-name),
- START(timer-name),

- STOP(timer-name) and
- SET(timer-name, expression).

A value may be assigned to a timer through SET statement.

ASN.1 subset

We adopt a subset of the ASN.1 notation defined in [ISO-8824] to specify the data structures of the protocol service primitives, protocol data units and their parameters. The basic ASN.1 type notation defined in Section 1 and Section 2 of [ISO-8824] is chosen as the ASN.1 subset for TESTGEN. The SPs, PDUs and parameters of a protocol are specified in ASN.1 in a separate file but they may be referenced by the Estelle.Y specification of the protocol.

The SP and PDU parameters are represented by dot notation mechanism in an Estelle.Y specification as the following:
$< SPname > \{. < parametername >\}^+$
or $< PDUname > \{. < parametername >\}^+$.

The parameters may be referenced in an enabling predicate or in some statements of an action function where the protocol variables may be referenced. Figure 3.4 is an excerpt of the ASN.1 specification of SPs of FDDI MAC protocol.

3.2.4 Estelle.Y versus Estelle

Estelle.Y is basically a single module Estelle. The process of transforming an Estelle specification to an Estelle.Y specification is similar to the normalization process in [Sari-89] and hence is feasible.

Two major transformations are required. The multiple modules in Estelle specification must be integrated into a single module and some of the statements e.g., case statement, delay statement must be replaced. The SPs and PDUs are specified in Pascal in an Estelle specification and must be specified in ASN.1 in the Estelle.Y/ASN.1 specification. This transformation is

```
FDDIMac DEFINITIONS ::=

BEGIN
...

PhyToMacAsp      ::=      CHOICE
{
PhUnitDataRequest,
PhUnitDataIndication,
PhUnitDataStatusIndication,
PhInvalidIndication
}

PhUnitDataRequest        ::=      SEQUENCE
{
phRequest        Symbol
}
...
```

Figure 3.4: Example of ASN.1 definition of protocol SPs

straightforward since ASN.1 describe abstract data types in a way similar to Pascal.

3.3 Protocol data structure

In order to generate TTCN test suites automatically from the formal specification of protocols, the information specified in the formal specification must be accessible to the test generation engine. The protocol data structure (PDS) is designed to represent the formal protocol specifications based on ETS+ASN.1 formalism in a machine accessible form.

The protocol information saved in the PDS will be directly used for the subtour identification process which is the key process of the automatic test suite generation. In other words, the TSG is completely based on the protocol information in the PDS. Therefore it is very important to ensure that the PDS correctly represents the information of the protocol specifications so that test suites can be generated correctly. It is also important that the PDS is easy to be accessed by the test generation engine. To facilitate the identification of subtours, the protocol information in the PDS is organized as an internal representation of the protocol state machine graph. Although the PDS is particularly designed for TESTGEN, it may also be used for other applications such as protocol validation tools.

A protocol data structure representing a complete, real world communication protocol could be very large and complex. To make the accessing and management easier, we organize the protocol information in a way similar to that the database uses to organize the data.

3.3.1 Representation of protocol descriptors

Based on the ETS+ASN.1 formalism, a protocol is formally described in terms of several sets of components, i.e. states, transitions, variables, constants, input service primitives, output service primitives, protocol data units and timers. We call these components *protocol descriptors.* In an Estelle.Y specification, some sets of the protocol descriptors are further described in terms of other sets of protocol descriptors. For instance, transitions may be described in terms of action functions, enabling predicates, expressions, statements, SP and PDU parameters etc.

In the PDS, a *structure* (or *type*) is defined for each set of protocol descriptors in the PDS.

Each of the protocol descriptors specified in the specification is then represented as an instance of a specific structure. For example, a state is an instance of structure STATE defined for states. The definition of the structures are given in the following subsections.

3.3.2 Protocol descriptor access

The main part of the PDS consists of pointer arrays. The pointers of each array point to a certain type of protocol descriptors. Each descriptor can then be accessed with its type (array name) and its key (array subscript) being known through the main structure.

Figure 3.5 illustrates the main data structure of the PDS.

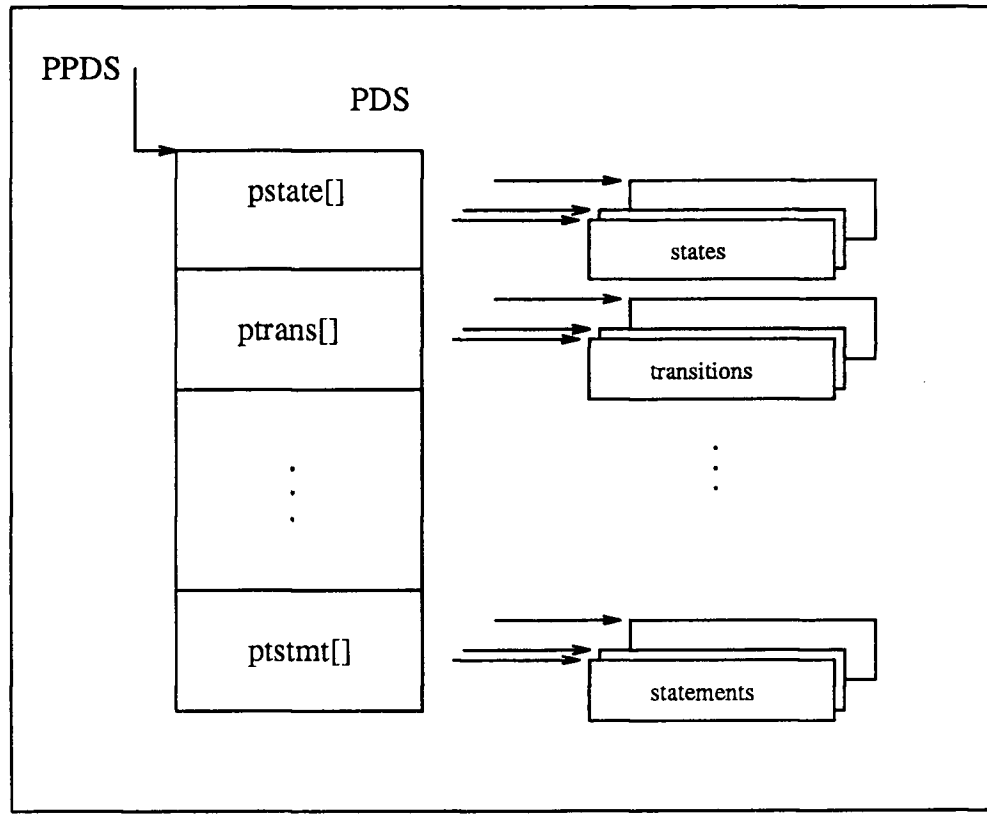


Figure 3.5: The main data structure of PDS

The C structure definition of the main structure of PDS is shown in figure 3.6.

```
typedef struct  {

int     init_state;

int     nb_of_states;
PSTATE  pstate[MAXSTATES -1];
int     nb_of_transitions;
PTRANS  ptrans[MAXTRANSITIONS - 1];
int     nb_of_variables;
PVAR    pvar[MAXVARIABLES - 1];
int     nb_of_constants;
PCONST  pconst[MAXCONSTANTS - 1];
int     nb_of_isps;
PISP    pisp[MAXISPS - 1];
int     nb_of_osps;
POSP    posp[MAXOSPS - 1];
int     nb_of_pdus;
PPDU    ppdu[MAXPDUS - 1];
int     nb_of_timers;
PTIMER  ptimer[MAXTIMERS - 1];
int     nb_of_efns;
PEFN    pefn[MAXEFNS - 1];
int     nb_of_spparms;
PSPPARM pspparm[MAXSPPARMS - 1];
......

} PDS, *PPDS;

}
```

Figure 3.6: C structure definition of PDS

The field *init_state* is the key to the initial control state of the protocol. PSTATE, PTRANS, PVAR, PCONST, PISP, POSP, PTIMER, PEFN, PSPPARM, ... are pointers to the structure STATE, TRANS, VAR, CONST, ISP, OSP, TIMER, AFN, SPPARM, ..., respectively.

The number of protocol descriptors used to describe a specific protocol is not known at the compile time. The C type definition of the PDS requires a fixed length of the pointer arrays. So a set of *MAX...* constants has to be introduced to indicate the maximal number of descriptors allowed. The *nb_of_...* fields indicate the actual total number of protocol descriptors of each set for a specific protocol.

The pointer arrays pointing to variables, constants and the SP and PDU parameters may be considered as some kinds of symbol tables as often used in the compiler constructions.

The relations between different descriptors are normally represented by *links* between the descriptors. Two possible approaches are considered. The first is that protocol descriptors may simply be linked directly by a pointer from one to another. The second is that the descriptors are indirectly linked together via the main data structure. In this way, a protocol descriptor indirectly "points" to other descriptors by remembering the types and keys of those descriptors and looking for physical pointers to them through the main data structure. Consider the complexity of a real-life protocol, the advantage of the latter is evident. Each set of protocol descriptors has its own index stored in the main structure so that the descriptors are easy to be accessed. The information of a protocol in the PDS is well-organized and therefore is easy to be accessed for TSG and the other applications.

If a set of protocol descriptors is specified in ASN.1 it stores a pointer to the data structure that used to represent the ASN.1 specification.

3.3.3 PDS data structure definition

The two major components of an ETS based protocol specification are states and transitions. The PDS is organized as a Finite State Machine (FSM) that has been extended to accommodate the following protocol descriptors other than states and transitions:

- ISPs and OSPs

- PDUs
- Protocol variables and constants
- Protocol Timers
- Enabling conditions of transitions
- Actions associated with transitions

The structure STATE, TRANS, ISP, OSP, PDU, VAR, CONST, TIMER, EXPR (for enabling conditions) and AFN are defined for the corresponding sets of protocol descriptors. The definitions of STATE, TRANS, VAR, CONST and TIMER are given in this subsection; ISP, OSP and PDU as well as SPPARM are defined in next subsection. The definition of EXPR and AFN are left for Chapter 4.

STATE as illustrated in Figure 3.7 is defined to represent the control states of a protocol.

- *key* field is a subscript of the *pstate* array. The pointer to this state is stored in *pstate*[*key*].
- *name* is the state's name.
- *numb_of_trans* indicates the total number of transitions applicable to this state.
- *trans_key* array stores the keys to the transitions that are applicable to this state.

Consequently, one will be able to access all the transitions applicable to a given state by looking up the pointer to a transition correspondent to the transition *key* stored in the *trans_key* array of applicable transitions. The purpose of storing this information in states is to facilitate the subtour identification process.

The *key* and *name* fields are also contained in the structure definitions of all other sets of protocol descriptors and also used in the similar way. They will be referred to without further explains later on.

TRANS is defined as depicted in Figure 3.8 to represent transitions.

- *from_state* and *to_state* are keys to the present state and the next state of a transition.
- *isp, ipdu, osp, osp2, opdu* and *opdu2* are keys to ISPs, OSPs or PDUs associated with a transition.

  Estelle.Y allows two output SPs or PDUs within a single transition. (This is why *osp2* and *opdu2* are included.)
- *epred* and *afn* are keys to the enabling predicate and the action function associated with a transition.
- *priority* field indicates the priority of a transition.

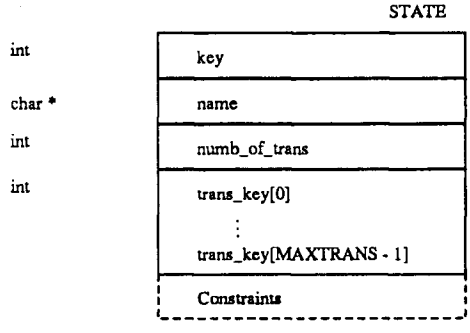


Figure 3.7: STATE definition

VAR and **CONST** are defined as shown in Figure 3.9. The *type* field indicates the data type of a variable or constant. Three supported data types are boolean, integer and character string. Boolean has two possible values, true and false; integer type is a four-byte integer; and character string is the same as defined in C. Fields *int_value, bool_value* and *char_str* store the value of a constant.

An enabling predicate (**EPRED**) is a boolean expression so that the *epred* field of a transition actually stores the key to an expression. **EXPR** is defined to represent expressions as well as enabling predicates.

Both an action function (**AFN**) and a compound statement (**CSTMT**) consist of a group of statements. The definitions of AFN and CSTMT are exactly the same. However, they are

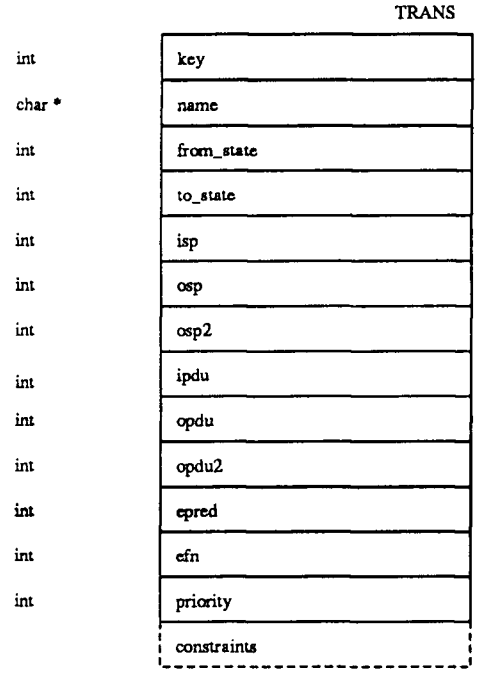


Figure 3.8: TRANS definition

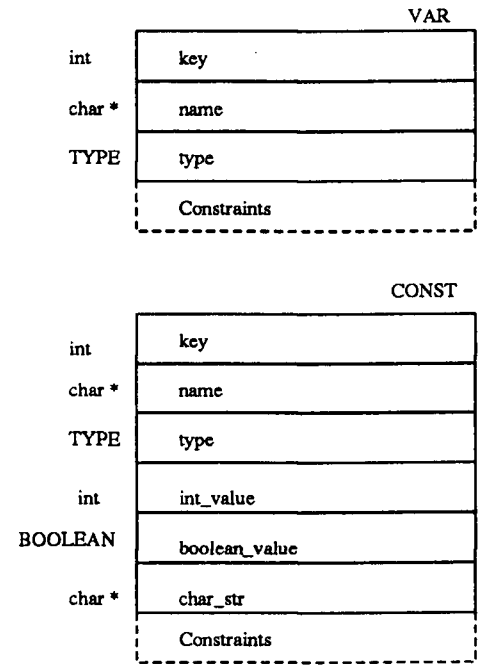


Figure 3.9: VAR and CONST definitions

defined as two independent types since they are conceptually different.

Figure 3.10 is the TIMER structure for the representation of timers. *timeout_value* is the time-out value of the timer. The TIME_UNIT is of integer type and the time unit is microsecond.

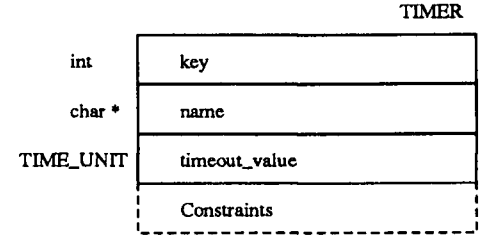


Figure 3.10: TIMER definition

The lower level structures of expressions (EXPR) and statements (IFSTMT, CSTMT, AFN, ...) are defined in Chapter 4 because they are closely related to the design of the TESTGEN parser.

Figure 3.11 illustrates the relations between all the components of the PDS. Note that some sets of descriptors are recursively defined by the same type of descriptors, e.g. an expression may be defined by a set of subexpressions. An enabling predicate (EPRED) is a boolean expression and the field of a transition actually stores the key to an expression. In figure 3.11 the dotted arrows from EPRED to EXPR indicates this type of relation.

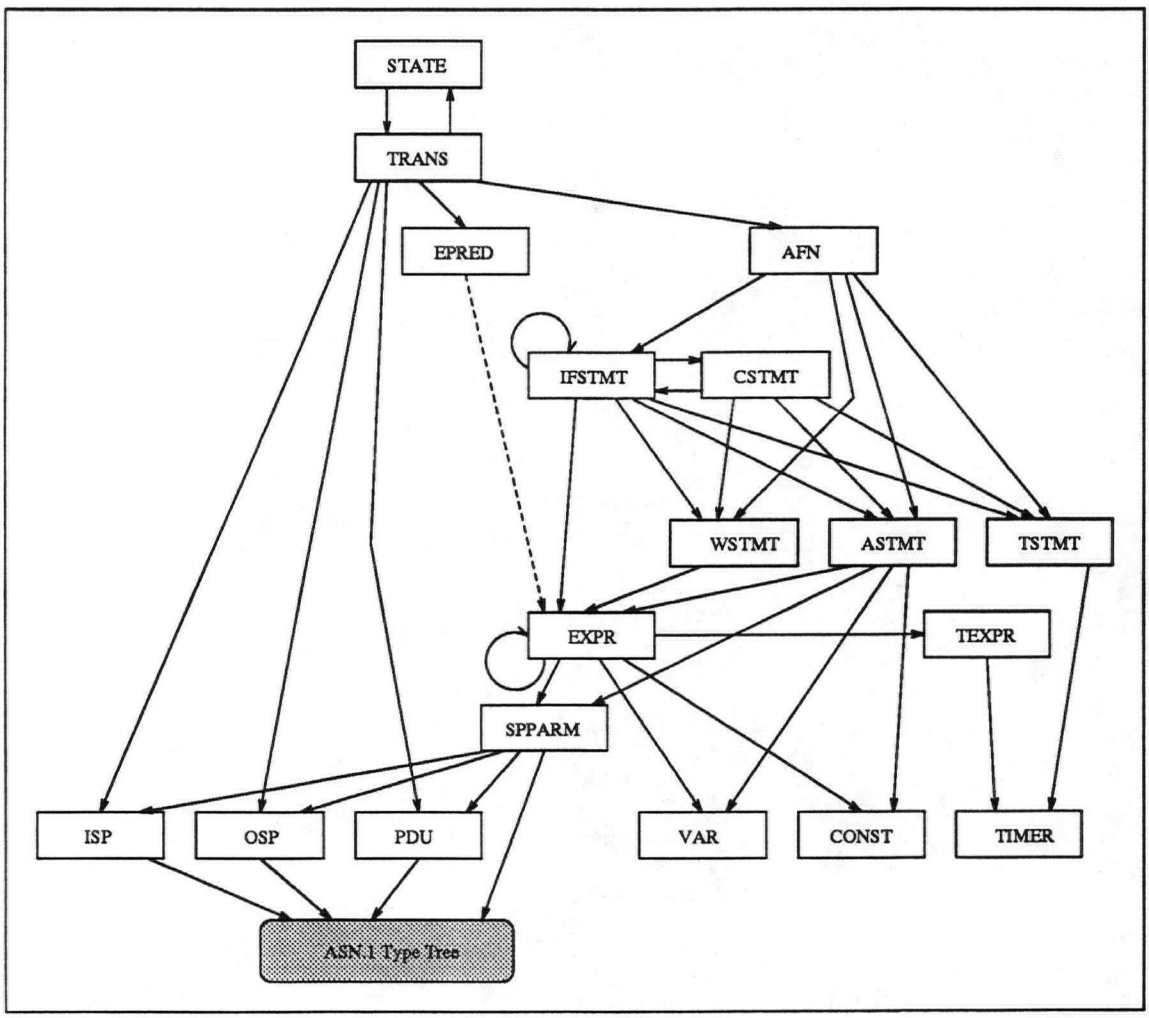


Figure 3.11: The relation diagram of the PDS components

Finally, some test generation and parameter variation constraints may be imposed on some sets of the descriptors. The "constraints" fields are included in the structure definitions of those sets to store the constraints. The reader may refer to [TR-90-x] for more detailed description of the PDS.

3.3.4 ASN.1 type tree

A set of data types defined in ASN.1 for a particular application is called an abstract syntax. Following this definition, the ASN.1 specification of SPs and PDUs of a specific protocol is an abstract syntax. The ASN.1 type tree [Sample-90] is an internal data structure that contains all the structuring and naming information of an ASN.1 abstract syntax.

The structure of input service primitives, output service primitives and protocol data units are required to be specified in ASN.1. To allow the ASN.1 information to be accessed from the PDS, these protocol descriptors have pointers to the ASN.1 type tree in which the ASN.1 information is stored.

We use ASN.1 type tree as the internal representation of the the input service primitives, output service primitives and protocol data units and parameters in ASN.1 part specification.

The ASN.1 type tree created by the ASN.1 parser is a tree of template ENODEs (T_ENODEs). Figure 3.12 is the C structure definition of a template enode. Each T_ENODE represents one layer of the structure/typing of ASN.1 types. Each T_ENODE holds information that describes an ASN.1 type. The components of a type definition (also called children of the type definition) are linked as a list. The pointer *child* of the type definition node points to the head node of the list of its children. The pointer *next* of the type definition points to one of its *siblings*. The siblings and the type definition itself are children of a higher layer type definition.

A simple example of an ASN.1 type definition and its type tree is illustrated in Figure 3.13 and Figure 3.14. Readers may refer to [Sample-90] for more details.

ISP structure is defined as in Figure 3.15.

- *PCO* is the name of the Point of Control and Observation at which the SP or PDU is exchanged by te IUT [ISO-9646].

```
typedef struct T_ENODE
{
    TAG                 univTag;       /* univ type tag (if not extra tag) */
    V_ENODE_PTR         tag;           /* actual tag to use for enc/decode */
    SUBTYPE_PTR         subtypes;      /* any subtyping info for this type */
    T_ENODE_PTR         next;          /* next elt  pts to sibling */

    V_ENODE_PTR         namedElmts;    /* named num/bits | enumerated defs */
    union          /* save some space with a union on mutually excl elmts */
    {
        T_ENODE_PTR     child;          /* child elmts if constructed */
        T_ENODE_PTR     choiceElmts;    /* elements of a choice type */
        T_ENODE_PTR     selectionType; /* type selection is from */
    } a;

    short               typeFlags;     /* type attrib flags: 2 bytes    */

    BYTE                sysFlags;      /* note: V_ENODE's use some of these */
                                       * ie children part of other type def */
    V_ENODE_PTR         defaultVal;

    IMPORT_ELMT_PTR     importRef;     /* if not null, imported type def ref*/

    char*               name;          /* field name if component else*/
                                       /* orig defined type name if redefined*/
    char*               typeName;      /* defined type name    */

} T_ENODE;
```

Figure 3.12: The data structure of template ENODE

```
PDU DEFINITIONS ::=
BEGIN -- a simple type defined in ASN.1 for the use as an example

PduType ::= SEQUENCE
{
    infoLength      INTEGER,
    info            CharacterString     OPTIONAL,
    status          BOOLEAN
}

END
```

Figure 3.13: ASN.1 type example

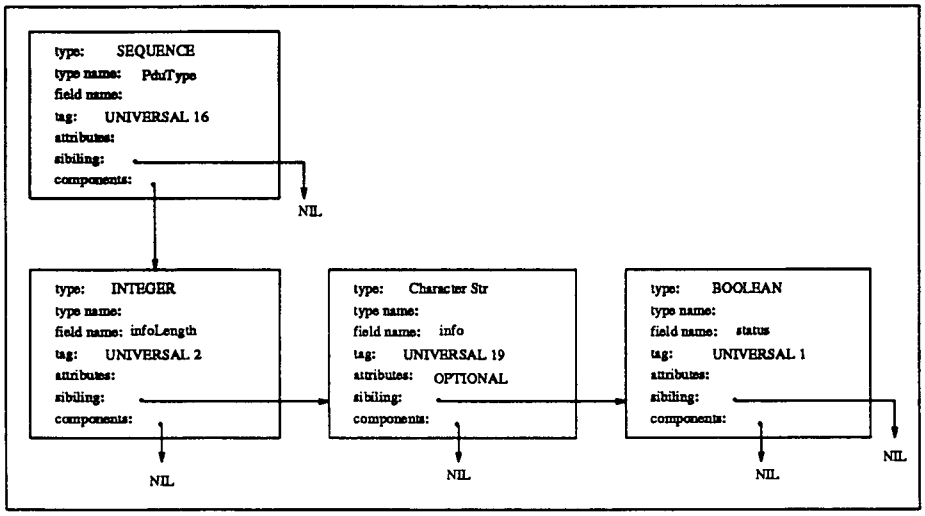


Figure 3.14: ASN.1 type tree of PduType

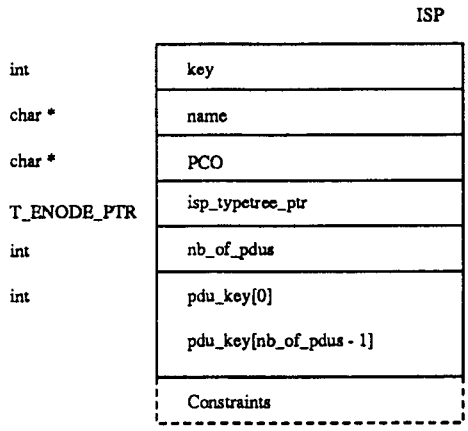


Figure 3.15: ISP definition

- *isp_typetree_ptr* is the pointer to an ASN.1 type tree that holds the ASN.1 definition of this ISP. The description of ASN.1 type tree is given in next subsection.
- *nb_of_pdus* is the number of different types of PDUs that can be encoded in an ISP.
- *pdu_key* is an array of keys to the PDUs that can be encoded in this input service primitive.

OSP is the same as ISP except that in OSP the pointer to ASN.1 type subtree is named *osp_typetree_ptr* instead of *isp_typetree_ptr*.

PDU is defined to have following fields:

- *key* and *name*
- *sent_in* and *recv_in* are the keys to the service primitives in which the PDU can be sent or received.
- *pdu_typetree_ptr* is a pointer to the ASN.1 type tree which holds the ASN.1 specification of the PDU.

SPPARM structure shown in Figure 3.16 is designed to represent the SP or PDU parameters. The parameters can be referenced or used just like a variable so that we treat the parameters as protocol descriptors and represent them in a way similar to that a variable is

represented in the PDS. On the other hand, for test suite generation, the TSG constraints are imposed on each of the parameters so that **SPPARM** structure is also necessary for storing of the constraints.

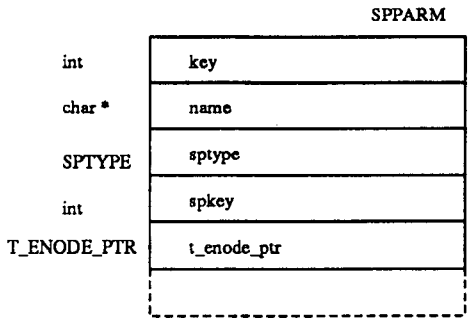


Figure 3.16: SPPARM definition

3.4 Summary

An Estelle.Y specification is more concise and more readable than an Estelle Normal Form specification [Sari-89] in that Estelle.Y supports conditional statement and loop statement (extra transitions are created due to the lack of conditional statements and loop statements in Estelle Normal Form). In addition, Estelle.Y supports timers explicitly to facilitate the testing of timers and supports ASN.1 directly. Estelle.Y is shown to be a good intermediate formal specification mechanism for automatic test generation or protocol validations.

The PDS can be easily extended to accommodate the protocol specifications with multiple modules. The current design of PDS can be used as a basic component of a composite PDS which is a higher level data structure that represents the hierarchical structure of multiple modules.

The major deficiency of PDS is due to the fact that the memory space is statically allocated for its main structure. The number of protocol descriptors varies with different protocols while the set of *MAX...* constants are fixed. When the specification of a specific protocol is being parsed, the number of protocol descriptors generated by the parser may exceed the *MAX...*

limit, even though we can define the *MAX...* constants large enough such that in normal cases the limit may not be exceeded. In case of limits being exceeded, users are asked to modify the set of *MAX...* constants defined some of the C source files. To avoid this deficiency, we can make use of a dynamic main data structure for PDS in which we use a dynamically allocated pointer list instead of the static pointer array. However, this dynamic data structure would be more complex and could require more accessing time. The current limits are set as follows:

```
MAXSTATES        = 40      MAXTRANSITIONS   = 150
MAXVARIABLES     = 100     MAXCONSTANTS     = 100
MAXISPS          = 20      MAXOSPS          = 20
MAXPDUS          = 20      MAXTIMERS        = 20
MAXTEXPRESSIONS  = 100     MAXEXPRESSIONS   = 1200
MAXSPPARMS       = 100     MAXSTMTS         = 100
MAXIFSTMTS       = 100     MAXWSTMTS        = 100
MAXASTMTS        = 400     MAXTSTMTS        = 100
MAXCSTMTS        = 100     MAXAFNS          = 150
```

Chapter 4

TESTGEN parser

In order to run the test suite generation applications, the formal Estelle.Y/ASN.1 specification must be translated into PDS. The internal PDS representation of the ETS+ASN.1 of a communication protocol is generally so complex that the manually generation (hard wired C code) of the PDS for each specific protocol would be too error prone and too labor intensive. On the other hand, the subtour identification and TTCN test suite generation processes of the TESTGEN are totally based on the protocol information saved in the PDS. In order to generate test suites properly, we must ensure the PDS to be generated correctly from the specification. A parser which can generate PDS from formal specifications is therefore required.

We developed the TESTGEN parser to parse the Estelle.Y/ASN.1 specifications to the PDS. The parser is developed in C on SUN workstations with about 8000 lines of C code.

This chapter discusses the issues of design and development of the TESTGEN parser. Section 1 is about the design issues. The implementation is discussed in Section 2. The testing of the TESTGEN parser is presented in the last section.

4.1 Design considerations

There are several possible approaches we can use to develop the TESTGEN parser. One way is to develop a parser that can recognize both Estelle.Y and ASN.1 languages. Such a parser can parse and translate the Estelle.Y/ASN.1 specification of protocols into PDS. However, a more

efficient way is to make use of the existing Estelle and ASN.1 tools.

A considerable amount of work has been done in the development of Estelle and ASN.1 tools [Neu-90] [Sample-90]. The available tools such as Estelle-C compiler [Vuong-88] and ASN.1-C compiler [Neu-90] generate C codes from an Estelle or ASN.1 specification. These *compiler* tools are not suitable for test suite generation because protocol *choices* information and *nondeterminisms* are lost during compilation [Vuong-91-1] so that they can not be used for development of TESTGEN.

An ASN.1 parser is presented in [Sample-90]. It can parse an ASN.1 specification to an internal data structure that holds the syntactic and semantic information of the ASN.1 specification. It is designed for conformance testing of application layer protocols. The ASN.1 parser generates an internal data structure, the ASN.1 type tree, from the ASN.1 specifications.

We develop a parser, the TESTGEN parser, to parse the Estelle.Y/ASN.1 specification of protocols into the PDS. The ASN.1 parser of [Sample-90] is then used to parse the ASN.1 specification to ASN.1 type tree. The generated ASN.1 type tree is linked to the PDS by TESTGEN parser when the Estelle.Y specification is being parsed. As the result, a PDS that includes both Estelle.Y and ASN.1 protocol information can be generated from an Estelle.Y/ASN.1 specification.

4.1.1 ASN.1 parser

The ASN.1 parser is designed for testing of OSI application layer protocols. It is developed by using UNIX Lex/Yacc tools. It produces the ASN.1 type tree from ASN.1 specifications. The ASN.1 type tree is written to a file as an relinkable block after it is generated. The main use of this tool is to build loadable ASN.1 type trees *off line*, which can subsequently be loaded during the execution of the protocol tester [Sample-90]. The ASN.1 parser provides a subroutine which can reload the ASN.1 type tree written in a file.

The reloadability of the ASN.1 type tree provided by the ASN.1 parser facilitates the incorporation of ASN.1 type tree into PDS generated by TESTGEN parser.

An intuitive way of the incorporation is to generate two data structures both on line and

then integrate them. However, considerable difficulties exist. The two parsers are developed independently and the naming conflicts of the subroutines and symbolic constants between the two parsers are difficult to be resolved, especially when they are developed by using Lex/Yacc tools and some machine-generated codes are used. On the other hand, we do not need to handle the naming conflicts of subroutines between two parsers if one of the data structure is generated off line.

Figure 4.1 illustrates the configuration of TESTGEN parser.

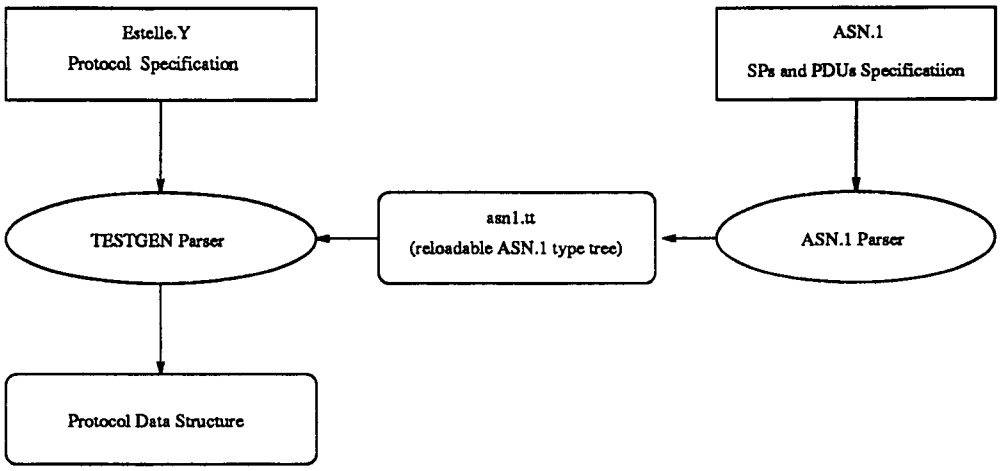


Figure 4.1: TESTGEN parser configuration

4.1.2 Lex/Yacc tools

The TESTGEN parser is implemented in C under the UNIX operating system. UNIX lex/yacc tools are designed to facilitate the construction of the front end compilers, i.e. the parser. We make use of these tools to implement the lexical analyzer and the TESTGEN parser itself.

Yacc input is produced based on the BNF rules of Estelle.Y. Figure 4.2 depicts the process of TESTGEN parser generation.

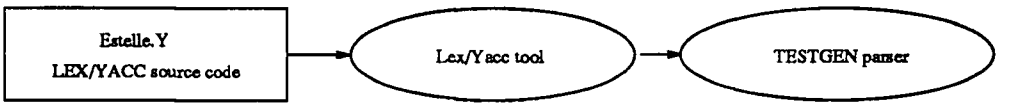


Figure 4.2: TESTGEN Parser generation

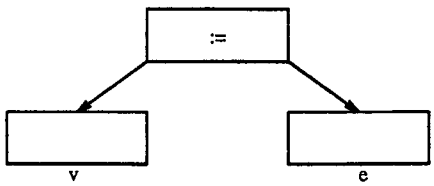
4.1.3 Abstract syntax tree

The *parse tree* is a popular intermediate representation form of source programs used for construction of many compilers. Given the BNF definition of a grammar, lex/yacc tools can construct the parse trees of source programs easily. In our application, lex/yacc tools are used to construct syntax trees, which are similar to parse trees and store the syntactical information of the Pascal statements and expressions in Estelle.Y/ASN.1 specification.

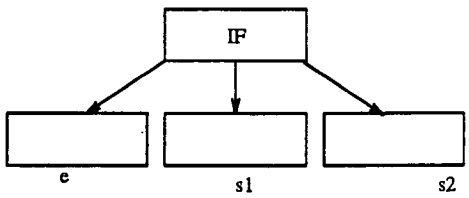
In a syntax tree, the control constructs of statements are represented as tree nodes. Figure 4.3a – c provides three examples of the syntax trees for statements. An assignment statement is depicted in Figure 4.3a. v could be a variable or a parameter which is the left-hand side operant of the assignment statement. The subtree e describes the expression of the assignment statement. Figure 4.3b illustrates an if statement. The subtree e represents the condition of the if statement. The subtree $s1$ and $s2$ describe the statements to be executed if e is true and false, respectively. Figure 4.3c illustrates a while loop. The loop control structure is represented by a single node. The subtree e describes the boolean expression of the loop and the subtree s represents the body of the loop.

IFSTMT, WSTMT and **ASTMT** are defined accordingly to represent the three Pascal statements supported by TESTGEN. The definition of **IFSTMT** for if statements as shown in Figure 4.4. They are actually the definition of the node structures of syntax trees.

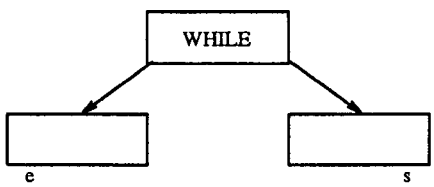
Similarly, the expressions are also represented by syntax trees. **EXPR** (also as **EPRED**) is defined as in Figure 4.5.



(a) v := e;



(b) if e then s1 else s2;



(c) while e do s;

Figure 4.3: Examples of syntax trees

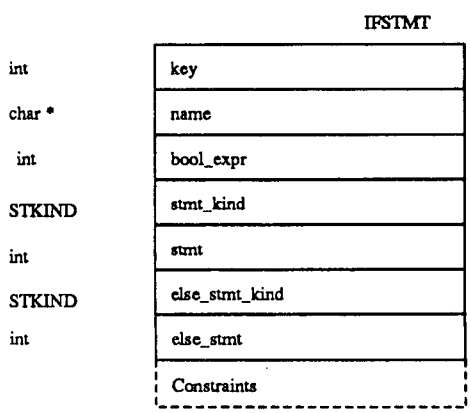


Figure 4.4: IFSTMT definition

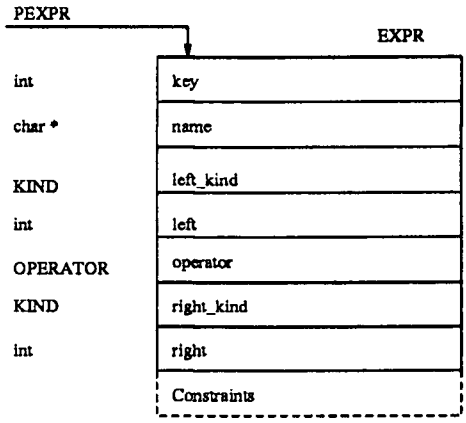


Figure 4.5: EXPR definition

4.2 Implementation

A protocol to be tested is specified in Estelle.Y. The ISPs, OSPs and PDUs and their parameters of a protocol are defined in ASN.1. The ASN.1 specification is parsed to ASN.1 type tree by the ASN.1 parser. Moreover, the ISPs, OSPs and PDUs must be declared in the Estelle.Y specification of the protocol. The parameters may be referenced in the Estelle.Y specification as well.

The TESTGEN parser first reloads the ASN.1 type tree; then calls the parsing function to parse the Estelle.Y specification of the protocol to the PDS. In the mean time, the ASN.1 type tree is linked to PDS. The TESTGEN parser set pointers to the ASN.1 type tree when the ISPs, OSPs and PDUs and parameters are created in the PDS.

Protocol variables and constants, ISPs, OSPs, PDUs, timers and states must be declared in the *declaration part* of the Estelle.Y specification. The TESTGEN parser creates appropriate types of protocol descriptors in the PDS when the declaration part is analyzed.

4.2.1 Declaration part

As mentioned before, an Estelle.Y specification consists of three parts: *declaration* part, *initialization* part and *state-transition* part. The declaration part is parsed first. The TESTGEN parser creats all declared variables, constants, ISPs, OSPs, PDUs, timers and states in the PDS

when the declaration part is parsed. The *trans_key*[] field of the states is empty at this stage and will be filled when the state-transition part of the specification is parsed. The appropriate default initial values are stored in VAR instances and TIMER instances created.

ISPs, OSPs and PDUs

When an input service primitive declaration is recognized, the parser creats an instance of ISP structure for it. Moreover, the TESTGEN parser looks for its ASN.1 definition in the ASN.1 type tree. If the definition is found, the pointer to the subtree holding that definition will be saved in the *...._typetree_ptr* field of the created ISP instance. The OSP and PDU declarations are treated similarly.

SP and PDU parameters

The SP or PDU parameters are not declared in an Estelle.Y specification. However, the parameters may be referenced and be used like a variable. Even though some of the parameters may not be referenced at all, some of the test suite generation constraints imposed on them must still be saved for TSG process. A instance of *SPPARM* must therefore be created for each of the ISP, OSP or PDU parameters when the ISP, OSP or PDU declaration is being parsed. The TESTGEN parser searches for the ASN.1 type tree for the definitions of each of the parameters . The pointers to the T_ENODEs of the ASN.1 type tree which store the type information of the parameters will be saved in a created parameter instance. Names of the parameters are formed based on dot notation.

Timers

All timers must be declared and there are instances of TIMER in the PDS created for each of them. The declared time-out values are stored in the corresponding instances.

Timer statements and timer expressions are treated as ordinary Pascal statements and expressions. They are represented as instances of TSTMT and TEXPR respectively. Timer

expressions have boolean values and can be evaluated as boolean expressions.

4.2.2 Initialization part

The TESTGEN parser sets default initial values for all variables, parameters and timers created in the PDS if they are not explicitly initialized. The default value is 0 for those of integer type, *false* for those boolean type and empty string for those of character string type. If these protocol descriptors are explicitly initialized in this part, the default initial values are ignored. The initial state may be assigned in this part. The state pointed by pstate[0] is assumed as the initial state otherwise.

4.2.3 State-transition part

The state-transition part consists of a group of transition definitions. The keys of transitions applicable to a state are stored in the *trans_key*[] field of that state when transitions are created in the PDS. The TESTGEN parser creates a transition in the PDS when a transition definition is recognized.

A transition is specified in terms of states, ISPs, OSPs, PDUs, EPREDs and AFN. The referenced states, ISP, OSP(s) and PDU(s) must have been created in the PDS since they must be declared in the declaration part. Other components such as the AFN and EPRED and their components such as expressions and statements are created when an action function or an enabling predicate is being parsed. The links to the referenced protocol descriptors are established by storing the types and keys to them.

The syntax trees of the action function (a set of Pascal or Timer statements) and the enabling predicate (a boolean expression) will be constructed. Whenever and the basic components such as variables, constants, parameters are referenced, their types and keys are stored by others. The timer expressions and timer statements are also created if they appear in an action function. The leaf nodes of a syntax tree may be variables, constants, parameters and timer constructs.

4.3 Testing

As mentioned before, the correctness of the PDS is very crucial to test suite generation using TESTGEN. The TESTGEN parser itself is designed carefully to prevent inconsistent information from being stored in the PDS. The *nb_of_...* fields in the main structure of the PDS record the number of protocol descriptors of each set. The values of these fields can only be changed through a subroutine. These fields are initialized to be 0s. For example, when a state is created, a key is assigned to it through the subroutine. The key is equal to the current value of the *nb_of_states* field and the value of that field increments immediately after the key is assigned. Finally, the pointer *pstate[key]* points to the created state. In addition, two verification mechanisms are developed to verify the correctness of a PDS generated.

4.3.1 Printing PDS

The PDS of a real life protocol could be very large and complex. Since TESTGEN is an interactive TSG tool, sometimes people may want to check a particular part (e.g. a particular state) of the PDS generated by the parser. It is more convenient to allow the user check information displayed on the screen than look for the information in the source specification file. A set of printing functions are provided for users to check the information stored in the PDS and make sure the PDS is consistent with the original specification. For example, users may use these printing functions to check if each protocol descriptors are correctly linked to other protocol descriptors by looking at the keys and types stored in these descriptors; or to check if the syntax tree is properly constructed by looking at the expression or statement information stored in the tree.

In order to verify the correctness of protocol information stored in the PDS, we can randomly pick up some protocol descriptors such as transitions and print them out using a set of printing functions. The printed protocol descriptors are then compared to those specified in the protocol specification.

4.3.2 Consistency checking of PDS

In addition to the printing functions, a set of consistency checking functions are developed to check the consistency of the PDS after it is generated. The consistency requirements on which the consistency checking functions are designed are as follow.

- For each set of protocol descriptors, the value of its $nb\_of\_...$ field should not be greater than the corresponding $MAX...$ constant. For example:

  $ppds \rightarrow nb\_of\_states \leq MAXSTATES$

  $ppds \rightarrow nb\_of\_transitions \leq MAXTRANSITIONS$

- For each protocol descriptor, the key stored in the *key* field must be equal to the index of the pointer to that descriptor. Therefore,

  $\forall i, 0 \leq i < ppds \rightarrow nb\_of\_states \Rightarrow ppds \rightarrow pstate[i] \rightarrow key = i$

  $\forall i, 0 \leq i < ppds \rightarrow nb\_of\_transitions \Rightarrow ppds \rightarrow ptrans[i] \rightarrow key = i$

- Keys of any set of descriptors stored in any field of a protocol descriptor should not be greater than the number of descriptors of the set, namely

  $\forall i, 0 \leq i < ppds \rightarrow nb\_of\_trans \Rightarrow$

  $1) 0 \leq ppds \rightarrow ptrans[i] \rightarrow from\_st < ppds \rightarrow nb\_of\_states$, and

  $2) 0 \leq ppds \rightarrow ptrans[i] \rightarrow to\_st < ppds \rightarrow nb\_of\_states$, and

  $3) 0 \leq ppds \rightarrow ptrans[i] \rightarrow isp < ppds- > nb\_of\_isps$, and ...

- The *type* field of any constants or variables can only be the defined data types.

  $\forall i, 0 \leq i < ppds \rightarrow nb\_of\_variables \Rightarrow$:

  $1) ppds \rightarrow pvar[i] \rightarrow type = BOOL\_TYP$, or

  $2) ppds \rightarrow pvar[i] \rightarrow type = INT\_TYP$, or

  $3) ppds \rightarrow pvar[i] \rightarrow type = CHAR\_STR\_TYP$,

  where $BOOL\_TYP, INT\_TYP$ and $CHAR\_STR\_TYP$ are the three legal types.

- The timeout values of a timer must be a non-negative integer.
- The *left_kind* and *right_kind* field and the *operator* field of an expression should store only those defined kinds and operators, i.e.

  $\forall i, 0 \leq i < ppds \rightarrow nb\_of\_exprs \Rightarrow$:

  1)$ppds \rightarrow pexpr[i] \rightarrow right\_kind = VAR\_$, or

  2)$ppds \rightarrow pexpr[i] \rightarrow right\_kind = CONST\_$, or

  3)$ppds \rightarrow pexpr[i] \rightarrow right\_kind = TEXPR\_$, or

  4)$ppds \rightarrow pexpr[i] \rightarrow right\_kind = PARM\_$, or

  5)$ppds \rightarrow pexpr[i] \rightarrow right\_kind = NONE\_$,

  where $VAR\_, CONST\_, TEXPR\_, PARM\_$ and $NONE\_$ are legal kinds.
- The *stmt_kind* field of action functions should contain only those Pascal statements supported by TESTGEN.

The complete set of consistency requirements can be found in Appendix E.

Using TESTGEN parser, we parsed FDDI MAC protocol and its services, TP0 and LAPB and generated their corresponding PDSs. The consistencies of these PDSs are subsequently checked by the consistency checking functions.

Chapter 5

Test generation for the FDDI MAC protocol

To verify the viability of the TESTGEN and to study the feasibility of applying the concepts of OSI conformance testing to a high-speed network protocol, we apply TESTGEN to the FDDI MAC protocol to generate a test suite for it.

We developed a formal specification of the FDDI MAC protocol in Estelle.Y and ASN.1. The PDS is generated from the formal protocol specification by the parser. A test suite is then generated for the protocol without any human interventions by use of a set of default TSG constraints. However, in order to generate a more comprehensive test suite with reasonable size and fault coverage, we have to tune the default constraints for each specific protocol. An appropriate set of constraints are set through the constraint editor. Finally a test suite will be generated based on the PDS and the TSG constraints.

In this chapter, firstly a brief review of the FDDI MAC protocol is presented. The problems with the formal specification of FDDI MAC protocol and the services in Estelle.Y and ASN.1 are then discussed. The major problem with the specification is how to produce a single ETS representation of the FDDI MAC protocol. In this chapter, we also show how to combine two extended transition systems into to one with equivalent behaviors. Finally we discuss the issues of tuning TSG constraints and the test suite generation.

5.1 FDDI MAC sublayer

The FDDI Media Access Control (MAC) layer is the lower sublayer of the Data Link Layer. It is the most important component of the FDDI standard among the four (MAC, PHY, PMD and SMT). MAC standard is the core of the FDDI standard. It distinguishes FDDI from other IEEE LAN standards.

FDDI MAC protocol is developed based on the concepts of Token Ring Access Method defined in ANSI/IEEE 802.5-1985 while the original concepts have been modified to accommodate the higher FDDI speeds. The protocol is designed to be effective at 100 Mb/s data transmission rate using the Token Ring architecture. The ring monitoring functions of the protocol are fully distributed.

The protocol is considered as one of the most complicated media access control schemes [SKO-89]. A brief review of the three major parts of the MAC protocol standard (ANSI X3.139 - 1987) is given in following subsections.

5.1.1 Services

Section 3 of the MAC standard specifies the services provided by MAC and the services required by MAC. Three sets of service primitives are defined. For each service primitive, the semantics, the time when it is generated and the effect of being received are specified.

Table 1 is a list of the service primitives specified in MAC standard. FDDI MAC provides MAC-to-LLC *services* for LLC's data transmission via MAC to LLC service primitives.

FDDI MAC layer is designed to be compatible with IEEE 802.2 LLC so that it can be used as the super service provider of LLC. As the result, the MAC to LLC services of FDDI are very similar to those of the IEEE 802.5 Token Ring standard.

MAC to PHY services provided by PHY layer are used for MAC data transmission. MAC provides services for the SMT's data transmission as well. Through the MAC to SMT services, SMT controls the operation of MAC for purposes of the station management.

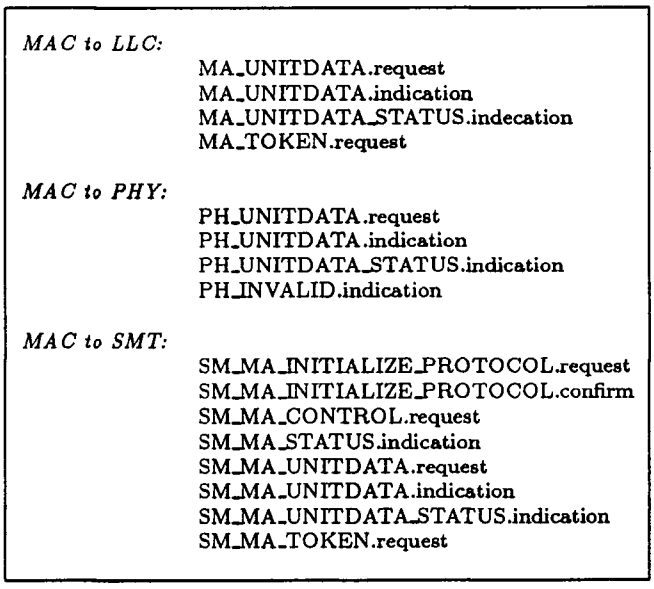


Figure 5.1: FDDI MAC service primitives

5.1.2 Facilities

This section of the MAC standard specifies the MAC protocol data units, timers and frame counts.

Protocol data units

The MAC layer processes both PHY layer protocol data units and MAC layer protocol data units. The PHY protocol data unit contains 4 bits of binary data, called a data *symbol.* Symbols are passed across the MAC to PHY interface via the service primitives. Figure 5.2 illustrates the formats of the two MAC PDUs: Tokens and Frames.

Timers

Timers are critical parts of the FDDI MAC Timed-Token protocol. MAC uses three timers. The Token-Holding Timer (THT) controls how long the station may transmit asynchronous frames. The Valid-Transmission Timer (TVX) is used to recover from transient ring error situations. The Token-Rotation Timer (TRT) is used to control ring scheduling during normal operation

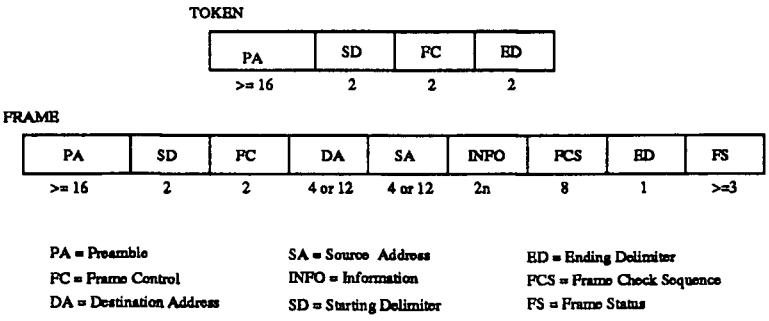


Figure 5.2: Format of MAC Protocol Data Units

and to detect and to recover from serious ring error situations.

5.1.3 Operation

The operation of mac protocol is described as several phases in the FDDI mac standard.

Frame transmission Upon a request for SDU transmission from LLC, MAC constructs a frame from the SDU by placing the SDU in the INFO field of the frame. The frame (encoded SDU) then remains queued awaiting the receipt of a token that may be used to transmit it. Upon the capture of an appropriate token, the station begins transmitting its queued frame(s) in accordance with the rules of token holding. In response to each data transmission request of LLC, a confirmation will be sent to the LLC after the SDU is transmitted.

Token transmission The station releases the token immediately after the transmission of the frame(s) is (are) completed.

Frame Stripping Each transmitting station will be responsible for removing from the ring the frames that it originated after being acknowledged by the destination station.

Ring scheduling Transmission of normal PDUs on the ring is controlled by a Timed Token Rotation protocol based on the Target Token Rotation Time which is negotiated by all of the stations attached to the ring during the ring initialization process.

MAC provides two asynchronous transmission modes by using two kinds of Tokens, nonrestricted and restricted Token.

Ring monitoring The MAC monitoring functions are distributed among all stations on

the ring. *Claim token process* allows all stations to bid for the right to initiate the token and negotiate the target token rotation time. When Claim token process is successfully completed, a station begins initialization process. *Beacon process* is used to signal all remaining stations that a significant logical break has occurred and to provide diagnosis or other assistance to the restoration process (via SMT).

5.1.4 Structure

MAC consists of two cooperating processes, the MAC receiver and the MAC transmitter. The two processes are specified both with text and state diagrams. The MAC receiver receives and validates information from the ring and detects ring errors and failures. The major triggering event to the receiver is the arrival of a symbol via a PH_UNITDATA.indication primitive. The MAC transmitter repeats information from the other stations on the ring, inserts information from its own station into the ring, and cooperates with other stations to coordinate priorities for use of the ring. The major triggering event to the transmitter is the the arrival of a symbol together with any event signals from the receiver. The event signals from the receiver could be the receipt of a token or a frame and others.

5.2 The formal specification of FDDI MAC protocol

This subsection discusses the Estelle.Y specification of the FDDI MAC layer protocol defined in the ANSI X3.139-1987 standard. A protocol standard specifies both the observable (external) and non-observable (internal) behaviors of a protocol. Since the ISO test methodology and its *black-box* test principle are chosen for TESTGEN, only the information relevant to the external behaviors of the FDDI MAC entity are needed to be specified.

Test method

MAC layer is relatively low-level compared to other layers of OSI reference model. The mac protocol has to deal with symbols passed across the MAC-to-PHY interface via MAC to PHY service primitives.

The behaviors of the mac protocol are described in terms of the exchange of MAC PDUs, the MAC to LLC service primitives as well as the MAC to PHY service primitives.

However, the abstract properties of the protocol, such as the data transmission, ring scheduling and ring monitoring functions are described in terms of the exchange of MAC PDUs and the MAC to LLC service primitives, although MAC PDUs are not exchanged directly with the peer entities but via MAC to PHY service primitives. The protocol behaviors involving MAC to PHY services are mostly internal. The observable behaviors of FDDI mac protocol are determined by the abstract properties of the protocol.

If control and observation are specified in terms of ASPs, it will include control and observation of the PDUs carried by those ASPs; but if it is specified solely in terms of PDUs (at layer N) then the underlying ASPs are not considered to be controlled or observed [ISO-9646].

We are not interested in controlling or observing MAC to PHY service primitives since protocol behaviors involving MAC to PHY services are trivial. Also the specification of these protocol behaviors is very tedious.

Figure 5.3 depicts the test method that is used to test FDDI MAC. It is conformed to the 9646 standard.

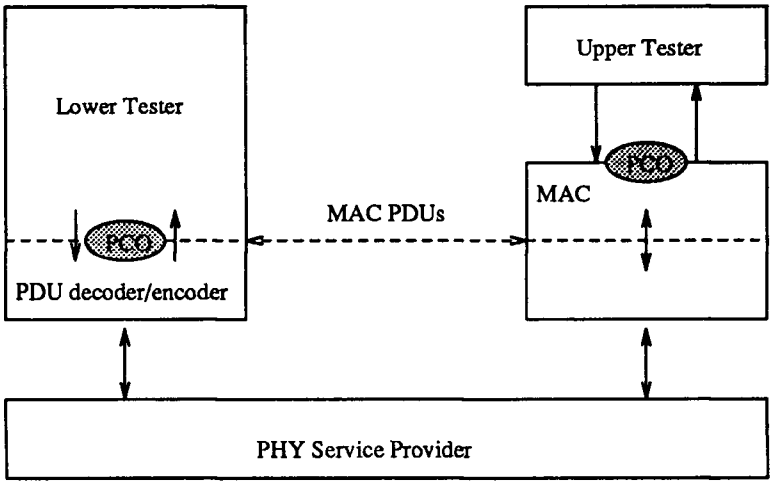


Figure 5.3: Testing of FDDI MAC layer

We specify only the abstract properties of the FDDI MAC protocol, namely the exchange of frames and tokens between peer MAC entities without concerning the interactions at MAC to PHY interface.

5.2.1 Data part in ASN.1

MAC to LLC service primitives, a subset of MAC to SMT service primitives and MAC Protocol Data Units and the data type of their parameters are specified in ASN.1 in a separated file. Although ASN.1 itself allows more complicated data types, only those supported by TESTGEN should be used to specify the MAC service primitives and PDUs.

The specification is straightforward. Normally a service primitive is specified as a SEQUENCE. The parameters are the components of the SEQUENCE and have data types such as INTEGER, BOOLEAN etc.

There are two problems with the data part specification. One problem is that a MA_UNITDATA.request may contains a set of subrequests. Each subrequest has its own set of parameters including the service data unit (SDU) parameter. To serve the request, MAC constructs a set of frames for each subrequest for their SDUs. There exists technical difficulties for TESTGEN to handle this kind of nested service primitives. We solve the problem by assuming that a data request may contain only one subrequest. Same problem exists when specifying SM_MA_UNITDATA.request.

Another difficult point is in the specification of symbols. Some of the MAC PDU fields are defined in terms of symbols in the FDDI MAC standard. There are two kinds of symbols passed across MAC to PHY interface. A data symbol contains 4 bits of binary data, and a control symbol contains 5 code bits. We temporarily use INTEGER type to define both control symbols and data symbols since no bit string type is supported by TESTGEN currently.

5.2.2 Control part in Estelle.Y

As mentioned in previous sections, an Estelle.Y specification is basically a single module Estelle specification modeled by an EFSM (or more precisely, a single ETS). In order to specify a pro-

tocol in Estelle.Y, we first need to represent the protocol as a single EFSM. For the conformance testing purpose, we also need to decide which aspects of a protocol need to be tested and thus need to be specified.

Data transmission

The data transmission process of the MAC protocol is characterized by the Token Ring access method. A LLC data request may be received by MAC any time but the data may not be sent out immediately. MAC must wait for the arrival of a special PDU, ie. a token to transmit data. As a consequence, there may be several outstanding data requests queued by MAC before the token is captured. The effect of receiving these data requests (service primitives) can not be observed unless a token (PDU) is captured. The frames constructed from the data requests will be sent out in their received order and data confirmations will be sent to LLC in response to each LLC data request.

The existence of outstanding data requests can only be represented by data states (namely Estelle.Y variables in the Estelle.Y specification) in the ETS-based representation of the FDDI MAC protocol.

It is natural to use a queue data structure to describe these data states. Unfortunately, a queue data structure is not currently supported by TESTGEN. We therefore simulate queues of limited lengths by using sets of simple variables.

Facilities

Since Estelle.Y provides explicit language constructs for specifying timers, in both of the declaration part and the transition part, the specification of FDDI MAC timers is straightforward.

MAC structure

As being required by TESTGEN, the FDDI MAC protocol must be represented as a single extended transition system (ETS) and specified as a single module in Estelle.Y.

As mentioned in the previous section, MAC consists of two cooperating processes, the MAC receiver and the MAC transmitter. The two processes are specified both with text and state diagrams which can be naturally represented as two individual ETSs.

Based on the information specified in the services section and operation section of the MAC standard, we construct an extended transition system which represents all aspects of the protocol behaviors that we intend to specify in Estelle.Y/ASN.1.

Before we present the construction of this ETS for MAC, we show how to combine two arbitrary extended transition systems into a single one with equivalent behavior.

Definition 5.1 *Given two extended transition systems* $ETS_1 = (Q_1, E_1, T_1, q^1_{init})$ *and* $ETS_2 = (Q_2, E_2, T_2, q^2_{init})$. *An extended transition system* $ETS = (Q, E, T, q_{init})$ *simulates* ETS_1 *and* ETS_2, *if there exists a relation* $R \subseteq Q_1 \times Q_2 \times Q$ *such that* $< q^1_{init}, q^2_{init}, q_{init} > \in R$ *and* $\forall < q_1, q_2, q > \in R$, $< q_1, e, q'_1 > \in T_1$ *implies that there exists a* $q' \in Q$ *such that* $< q, e, q' > \in T$ *and* $< q'_1, q_2, q' > \in R$, *or* $< q_2, e, q'_2 > \in T_2$ *implies that there exists a* $q' \in Q$ *such that* $< q, e, q' > \in T$ *and* $< q_1, q'_2, q' > \in R$.

Definition 5.2 *Given two extended transition systems* $ETS_1 = (Q_1, E_1, T_1, q^1_{init})$ *and* $ETS_2 = (Q_2, E_2, T_2, q^2_{init})$. *An extended transition system* $ETS = (Q, E, T, q_{init})$ *is simulated by* ETS_1 *and* ETS_2, *if there exists a relation* $R \subseteq Q_1 \times Q_2 \times Q$ *such that* $< q^1_{init}, q^2_{init}, q_{init} > \in R$ *and* $\forall < q_1, q_2, q > \in R$, $< q, e, q' > \in T$ *implies that there exists a* $q'_1 \in Q_1$ *such that* $< q_1, e, q'_1 > \in T_1$ *and* $< q'_1, q_2, q' > \in R$ *or there exists a* $q'_2 \in Q_2$ *such that* $< q_2, e, q'_2 > \in T_2$ *and* $< q_1, q'_2, q' > \in R$.

Definition 5.3 *Given two extended transition systems* ETS_1 *and* ETS_2. *An extended transition system* ETS *is equivalent to* ETS_1 *and* ETS_2, *if* ETS *simulates and is simulated by* ETS_1 *and* ETS_2.

Definition 5.4 *Given two extended transition systems* $ETS_1 = (Q_1, E_1, T_1, q^1_{init})$ *and* $ETS_2 = (Q_2, E_2, T_2, q^2_{init})$, *the product of* ETS_1 *and* ETS_2 *(written as* $ETS_1 \otimes ETS_2$*) is an extended transition system* $ETS = (Q, E, T, q_{init})$, *where* $Q = Q_1 \times Q_2$, $E = E_1 \cup E_2$, $T = \{<< q_1, q_2 >, e, < q'_1, q'_2 >> \mid < q_1, e, q'_1 > \in T_1$ *and* $q_2 = q'_2$ *or* $< q_2, e, q'_2 > \in T_2 and q_1 = q'_1\}$ *and* $q_{init} =< q^1_{init}, q^2_{init} >$.

Theorem 1 *Given two extended transition systems ETS_1 and ETS_2. $ETS_1 \otimes ETS_2$ is equivalent to ETS_1 and ETS_2.*

Proof. Let $ETS_1 = (Q_1, E_1, T_1, q^1_{init})$ and $ETS_2 = (Q_2, E_2, T_2, q^2_{init})$. By Definition 5.4, $ETS = ETS_1 \otimes ETS_2 = (Q, E, T, q_{init})$ where $Q = Q_1 \times Q_2$, $E = E_1 \cup E_2$, $T = \{<< q_1, q_2 >, e, < q'_1, q'_2 >> \mid < q_1, e, q'_1 > \in T_1$ *and* $q_2 = q'_2$ *or* $< q_2, e, q'_2 > \in T_2$ *and* $q_1 = q'_1\}$ and $q_{init} =< q^1_{init}, q^2_{init} >$. Define relation $R \subseteq Q_1 \times Q_2 \times Q$ as follow. Initially, $R = \{< q^1_{init}, q^2_{init}, q_{init} > \} = \{< q^1_{init}, q^2_{init}, < q^1_{init}, q^2_{init} >>\}$. Repeat the following procedure until R is not growing. For any $< q_1, q_2, < q_1, q_2 >> \in R$, if $< q_1, e, q'_1 > \in T_1$, $R \leftarrow R \cup < q'_1, q_2, < q'_1, q_2 >>$; if $< q_2, e, q'_2 > \in T_2$, $R \leftarrow R \cup < q_1, q'_2, < q_1, q'_2 >>$. It is clear that elements in R are all of the form $< q_x, q_y, < q_x, q_y >>$.

We now prove that $ETS_1 \otimes ETS_2$ simulates ETS_1 and ETS_2. It is obvious that $< q^1_{init}, q^2_{init}, q_{init} >=< q^1_{init}, q^2_{init}, < q^1_{init}, q^2_{init} >> \in R$. Consider any $< q_1, q_2, q > \in R$. By the above procedure, it must be that $q =< q_1, q_2 >$.

Case 1. If $< q_1, e, q'_1 > \in T_1$, then let $q' =< q'_1, q_2 >$. Thus by Definition 5.4 $<< q_1, q_2 >, e, < q'_1, q_2 >> \in ETS_1 \otimes ETS_2$, and by the above procedure generating R, $< q'_1, q_2, q' >=< q'_1, q_2, < q'_1, q_2 >> \in R$.

Case 2. If $< q_2, e, q'_2 > \in T_2$, then let $q' =< q_1, q'_2 >$. Thus by Definition 5.4 $<< q_1, q_2 >, e, < q_1, q'_2 >> \in ETS_1 \otimes ETS_2$, and by the above procedure generating R, $< q_1, q'_2, q' >=< q_1, q'_2, < q_1, q'_2 >> \in R$.

Thus in either case, we see R satisfy the condition in Definition 5.1. Therefore, $ETS_1 \otimes ETS_2$ simulates ETS_1 and ETS_2.

We now proceed to prove that $ETS_1 \otimes ETS_2$ is simulated by ETS_1 and ETS_2. It is obvious that $< q^1_{init}, q^2_{init}, q_{init} >=< q^1_{init}, q^2_{init}, < q^1_{init}, q^2_{init} >> \in R$. Consider any $< q_1, q_2, q > \in R$. q must be $< q_1, q_2 >$. If $< q, e, q' > \in T$, it must be i) $q' =< q'_1, q_2 >$ and $< q_1, e, q'_1 > \in T_1$ or ii) $q' =< q_1, q'_2 >$ and $< q_2, e, q'_2 > \in T_2$. It is clear that, by the above procedure generating R, in case i) $< q'_1, q_2, q' > \in R$, and in case ii) $< q_1, q'_2, q' > \in R$. Thus in either case, we see R satisfy the condition in Definition 5.2. Therefore, $ETS_1 \otimes ETS_2$ is simulated by ETS_1 and ETS_2. $\triangle$

Hence $ETS_1 \otimes ETS_2$ is equivalent to ETS_1 and ETS_2.

We now proceed to describe the FDDI MAC protocol based on the extended transition system model.

Before the MAC receiver and transmitter are combined, some revisions to the original state diagrams are necessary. We have mentioned that the data transmission via the MAC to PHY service primitives are not intended to be controlled or observed. Currently, the process of combining two ETSs is not automated. It is very tedious to combine the original state diagrams into one manually. This is another reason for using the test method mentioned in Section 5.2.

The major triggering event of the MAC receiver is the occurrences of PHY_UNITDATA_indication which is a MAC to PHY service primitive. As the result of this kind of events being eliminated, the receiver state diagram is reduced to have two states. For the same reason, two of the six states of the original transmitter state diagram are combined into one state in the revised transmitter state machine.

We convert the two revised state diagrams, the MAC receiver and transmitter, into two individual extended transition systems ETS_1 and ETS_2. We then generate the *product* of ETS_1 and ETS_2 ($ETS_1 \otimes ETS_2$). The rules or the algorithm of generating $ETS_1 \otimes ETS_2$ is in fact provided in Definition 5.4. The correctness of $ETS_1 \otimes ETS_2$ is guaranteed by Theorem 1.

We noticed that the two state diagram provided by the MAC standard do not actually represent the complete knowledge of the behaviors of MAC. For example, the interactions through the MAC to LLC interface via MAC to LLC SPs are not specified in the state diagrams. We revise the resulting single ETS to accommodate all interested knowledge of the external behavior of the FDDI MAC protocol including those aspects missing in the original two state diagrams.

Figure 5.4 illustrates the simplified single ETS representation of FDDI MAC protocol. In Figure 5.4, each arc between two nodes (representing two states of the ETS) may represent several transitions associated with different events.

A complete ASN.1 specification and an excerpt of the Estelle.Y specification of FDDI MAC protocol are provided in Appendix B and C for reader reference.

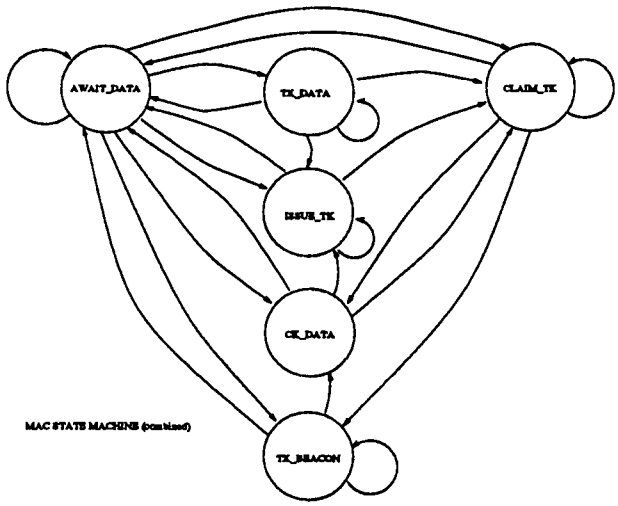


Figure 5.4: FDDI MAC protocol state machine

5.3 Generating test suite using TESTGEN

This section presents the test suite generation and selection of FDDI MAC protocol with TESTGEN.

5.3.1 PDS of FDDI MAC protocol

The Estelle.Y specification of FDDI MAC protocol is first parsed and the PDS is generated from the specification. The generated PDS contains about 1000 expressions (namely about 1000 instances of EXPR structure) and about 350 assignment statement (instances of ASTMT structure). The number of instances of other defined structures are less than 100.

5.3.2 Tuning constraints for FDDI MAC protocol

A test suite can be generated for the FDDI MAC protocol by using the set of default constraints.

In order to generate a test suite with reasonable size and better fault coverage for a specific protocol, we have to tune the constraints, namely reset or change the constraints through the constraint editor according to the characteristics of the protocol to be tested.

As mentioned in Chapter 2, the TSG-constraints define an upper and a lower bound on the

number of times an ETS element can be reached or used in one subtour.

The maximal number of times SPs and PDUs can be used within a subtour is set to 1 by default. It means that any transition having a SP or PDU associated with it can be used at most once in a subtour. We need to change the $max - used$ value imposed on one of the MAC PDUs, *Frame*, into a value more than 1 to allow the case of when several frames are sent due to the capture of a token to be tested.

The maximal number of times a state can be reached within a subtour is set to 99 by default. This will cause excessive subtours being generated. For example, if there is a transition which starts from a state and ends at the same state and no SPs or PDUs are associated with the transition, 99 different subtours will be identified with the state being reached for 1 to 99 times.

The parameter variation constraints define a set of values for each parameter of each ISP or PDU that can be sent to the IUT. The instances of ISPs and PDUs that will be used to test the IUT are thus defined. The default parameter variation constraints on three supported parameter type are:

TRUE,	FALSE	for boolean,
0,	99	for integer and
"test-string1"		for character strings.

These default values are likely not suitable for testing the ISPs and PDUs of a specific protocol. For example, the *frame control* field of *Frame*, one of the FDDI MAC PDUs, is used to distinguish different types of *frames*. There are about 10 different types of frames defined in the FDDI MAC protocol standard. To test all of them, we need a set of specific values, namely default parameter variation constraints to the *frame control* field of *Frame*.

5.3.3 Test generation

The test suite generation engine identifies subtours based on the PDS representation of a protocol generated by the parser as well as a set of constraints set through the constraint editor. The test suite generation engine is being developed. A comprehensive test suite can be successfully generated by using TESTGEN with an appropriate set of constraints after the test

suite generation engine is implemented.

Our purpose of applying TESTGEN to real life protocols is mainly to demonstrate the viability of the TESTGEN tool and obtain some experience in specifying and testing of high-speed network protocols. So the constraints set by us may not satisfy special testing purposes in a testing center or in the industry. However, by our experience, TESTGEN is a flexible and productive and easy to use. In fact, users are easy to control the size of the generated test suite and to generate test cases with required fault coverage according to their own testing purposes.

5.4 Summary

In this chapter, we discussed the issues of test suite generation for FDDI MAC protocol with emphasis on the formal specification of the protocol. In order to produce the formal specification, we combined the MAC receiver and transmitter. We presented a method of combining two ETSs into a single one with equivalent behavior. In fact, this method is easily improved to combine arbitrary number of ETSs into one with equivalent behavior. This shows that our ETS-based Estelle.Y/ASN.1 can be used to specify any protocols.

Chapter 6

Conclusions

This thesis presents and discusses the design and implementation of the front end of TESTGEN, a software environment for test suite generation and selection for conformance testing as well as its application to a specific protocol, FDDI MAC protocol.

6.1 TESTGEN features

TESTGEN adopts the TSG-constraints based test suite generation method which integrates the generation and selection of abstract TTCN test suites from formal specification of communication protocols. The generated test cases cover both the control and the data flow part of the protocol.

TESTGEN supports consistent end-to-end use of ASN.1 in the given formal protocol specification. The ASN.1 support is incorporated in the generated PDS as well as in the TSG constraints mechanism, which leads to coherent ASN.1 TTCN constraints. Moreover, the TSG constraints approach in TESTGEN offers a flexible mechanism for generating conformance test suites and special purpose test suites for real life protocols. TESTGEN thus serves as a useful test-bed for experimenting with protocol test generation and selection in addition to being a useful productive system.

The design and implementation of the TESTGEN front end is crucial to the development of the TESTGEN. The ETS/ASN.1 formalism is a well-defined intermediate representation form

which can represent complete protocol knowledge including both control and data parts precisely. Despite the fact that the ETS/ASN.1 formalism and its data structure representation (the PDS) are designed for automatic test generation using the TESTGEN tool, they may be used for other applications such as the development of other ETS based test generation tools. Furthermore, such intermediate representations and their internal data structure representations of protocols are useful for many other applications such as protocol validation tools and conformance testing tools based on trace analysis methodology.

To compare TESTGEN with other existing test sequence generation tools, we consider those proposed in [Ural-88], [EBE-89-1] and CONTEST_ESTL by [Sari-89].

The similarities between TESTGEN, CONTEST_ESTL and Ural's proposal are that they all generate TTCN test suites based on protocol specification in Estelle (or Estelle variants). The TSG method proposed in [EBE-89-1], on the other hand, derives test sequences based on *EBE*. The test generation method used in TESTGEN detects errors that are not detected by CONTEST_ESTL, Ural's method or *EBE* based method. TESTGEN is expected to provide better fault coverage by using a more elaborate test sequence generation algorithm. Moreover, TESTGEN completely automates the TSG process by using default setting of TSG constraints, as contrast to CONTEST_ESTL, in which the test sequence generation procedure is just semi-automated. On the other hand, CONTEST_ESTL provides graphical displaying of the control and data flow graphs of the specification as well as generating the test sequences [Sari-89], which is not available in TESTGEN. In addition, TESTGEN provides users with much flexibility through the interactive setting of constraints on each of ETS elements. Moreover, it supports ASN.1 directly. Estelle.Y can be served as either an intermediate or direct formal description language.

6.2 TSG for FDDI using TESTGEN

As we mentioned before, FDDI is not an ISO standard. However it is developed in conformance with the OSI reference model and other ISO guidelines. As a consequence, the ISO conformance

testing methodology and frame work [ISO-9646] can be applied to the testing of FDDI MAC protocol without much difficulties. As stated in the ISO 9646 standard, the defined methodology and framework can not be used for testing of the Physical Layer. The FDDI MAC layer protocol is the only component of FDDI that can be tested using the methodology and framework defined in the ISO 9646 standard.

Given the fact that the FDDI MAC protocol are specified in terms of two concurrent processes, efforts has been made to produce an ETS-based formal specification of the FDDI MAC protocol.

Although a test suite can be generated by TESTGEN for FDDI MAC protocol using a set of default constraints without any human interventions, in order to better satisfy testing purposes for a specific protocol, the user must tune the default constraints to allow some specific aspects of the protocol to be covered by the generated test suite.

Some knowledge about the behaviors of a protocol are described in the ETS/ASN.1 formalism is required for tuning constraints for the test generation, including the identification of major behaviors of the protocol. However, it is much easier for a user to understand the behaviors of a protocol specified in the ETS/ASN.1 formalism based specification than to provide the interventions required by CONTEST_ESTL [Ural-88].

6.3 Future works

The TESTGEN front end has been developed in a way to allow the enhancements to be done easily.

Many complex communication protocols are defined as multiple logical entities (e.g. the FDDI transmitter and receiver processes) in order to employ the parallelism and allow concurrency in a communication system. The multiple entities can not be naturally represented by a single module Estelle specification.

Furthermore, communication protocols are rarely tested in a stand-alone mode (local test method), but are combined with a service provider when using the distributed test method or a

Ferry Clip approach. In this case multiple modules are involved, including the service provider module as well as the IUT module in the test suite generation process.

A natural extension of TESTGEN will allow multiple-module Estelle specifications to be parsed into a composite protocol data structure. An enhanced test suite generation engine (TSG method and tool) will be able to identify the subtour combinations which cover also the observable service primitive exchange at the external gates(PCOs).

Only three data types allowed for variables and constants by the current version of TESTGEN. The three data types supported by the current version of TESTGEN are proved not enough to specify FDDI MAC protocol. Most of the fields of a PDU in Data link layer of OSI model are normally bit strings. The value of some of the fields determines the PDU's type. For example, the frame control field of FDDI MAC PDU distinguishes different types of PDUs. The MAC protocol examines some bits to identify the type of a received PDU, such as a token or a frame, and then take different actions based on the PDU's type.

To specify the behaviors of the FDDI MAC protocol precisely and naturally, bit type, bit string type and associated operations are necessary. For similar reasons, data structures such as queue and array are also required.

There are two possible approaches to support the needed data structures. One is to enhance the Pascal subset supported by TESTGEN. We need to know theoretically how complete a Pascal subset should be included in Estelle.Y for description of external behaviors of a protocol. Another possibility is to make use of ASN.1 for type definitions of variables (data states) and constants. For example, the ASN.1 *SEQUENCEOF* type which is similar to the array in Pascal can be used to define arrays of variables or constants by enhancing the internal data structure representation of protocols, namely the ASN.1 type tree and the PDS. Pointer type are necessary for queues of unlimited length. Pointer type can be supported by changing VAR structure slightly.

Finally a useful enhancement of TESTGEN is to incorporate a fault coverage measure into TESTGEN, based on the ongoing test coverage metric research conducted in [Vuong-91-2], into TESTGEN. The inclusion of this coverage measure will allow the user to compare and

evaluate the generated test suites with an objective measure and to tune the test suite generation constraints for the best results.

Bibliography

[ANSI-1987] American National Standard, FDDI Token Ring Media Accesss Control (MAC), ANSI X3.139-1987.

[ANSI-1989] Draft proposed American National Standard, FDDI Token Ring Station Management (SMT), ASC X3T9.5, Rev. 6, May 1989.

[BOCH-90] Gregor v. Bochmann, *Protocol Specification for OSI*, Computer Networks & ISDN Systems 18 (1989/1990).

[Chan-89] W. Y. L. Chan, S. T. Vuong and M. R. Ito, *An improved Protocol Test Generation Procedure Based on UIOs*, Proceedings of the ACM SIGCOMM '89 Symposium on Communication Architectures and Protocols, September 1989.

[Chow-78] T. S. Chow, *Testing Software Design Modeled by Finite State Machines*, IEEE Transactions on Computer, March 1978, Vol. 4, No. 3, pages 178-187.

[CRS-89] L. Mackert, J. Schneider, I. Neumeier-Mackert and R. Velthuys, *Executing Protocol knowledge*, European network Center IBM, Technical Report 43.8907, 1989

[EBE-89-1] J. P. Wu, S. T. Chanson, *A new Approach to Test Sequence Derivation based on External Behavior Expression (EBE)*, Technical Report 89-3, Department of Computer Science, University of British Columbia, Jan 1989. 1976, pp:305-330.

[EBE-89-2] J.P. Wu and S. T. Chanson, *Test Sequence Derivation Based on External Behavior Expression (EBE)*, 2nd International Workshop on Protocol Test Systems, Berlin, Germany, Oct. 1989.

[Fosd-76] L.D. Fosdick and L. J. Osterweil, *Data flow analysis in software reliability*, ACM Computing Serveys, Vol. 8, No.3,

[Gone-70] G. Gonenc, *A Method for the Design of Fault Detection Experiments*, IEEE Transactions on Computers, Vol. 19, No. 6, June 1970.

[ISO-8807] Information Processing System - Open System Interconnection - LOTOS -A Formal Description Technique based on the Temporal Ordering of Observational Behavior, ISO 8807, September 1987.

[ISO-8824] Information Processing System - Open System Interconnection - Specification of Abstract Syntax Notation One, ISO 8824, 1987.

[ISO-9074] Information Processing System - Open System Interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model, IS 9074, 1989.

[ISO-9646] Information Technology - OSI Conformance Testing Methodology and Framework, Draft International Standard, ISO/IEC DIS 9646 (5 Parts).

[ISO-TTCN] Information Technology - OSI Conformance Testing Methodology and Framework, Part 3:The Tree and Tabular Combined Notation, ISO/IEC DIS 9646-3, 1990.

[Kel-76] R. M. Keller, *Formal Verification of Parallel programs*, Communication of the ACM 19, pp371-384, 1976.

[LIN-90] Richard J. Linn, Jr., *Conformance Testing for OSI Protocols*, Computer Networks & ISDN Systems 18 (1989/1990).

[Naito-81] S. Naito and M. Tsunoyama, *Fault Detection for Sequential Machines by Transition Tours*, Proceedings of the 11th IEEE Fault-Tolerant Computing Symposium. pp.138-243, June 1981.

[Neu-90] G. W. Neufeld and Y. Yang, *The design and Implementation of an ASN.1 Compiler*, IEEE Transactions on Software Engineering, Vol.16, No. 10, Oct. 1990.

[PG-90] Marc Phalippou, Roland Groz, *From Estelle specifications to industrial test suites, using an empirical approach*, FORTE '90

[PUH-88] R. L. Probert, H. Ural and M. W. A. Hornbeek, *An Integrated Software Environment For Developing & Validating Standardized Conformance Tests*, Protocol Specification, Testing, and Verification VIII, S. Aggarwal and K. Sabnani (Editors), Elsevier Science Publishers B. V. (North-Holland).

[RAY-87] D. RAYNER, *OSI Conformance Testing*, Computer Networks & ISDN Systems 14 (1987).

[ROSS-86] Floyd E. Ross, *FDDI - A Tutorial*, IEEE Communications Magazine, May 1986 - Vol.24, No.5

[ROSS-90] Floyd E. Ross, James R. Hamstra and Robert L. Fink, *FDDI - A LAN Among MANs*, ACM Computer Communication Review, Vol. 20, No.3, July 1990

[Sabn-88] K.K. Sabnani and A. T. Dahbura, *A Protocol Test Generation Procedure*, Computer Networks and ISDN Systems, Vol. 15, No. 4, pp. 285-297, September 1988.

[Sample-90] M. Sample and G. Neufeld, *Support for ASN.1 within a Protocol Testing Environment*, The Third International Conference on Formal Description Techniques (FORTE '90), Madrid Spain, November 1990.

[Sari-87] B. Sarikaya, G. v. Bochman, and E. Cerny, *A Test Design Methodology for Protocol Testing*, IEEE Transactions on Software Engineering. Vol. 13, No. 5, pp. 518- 531, May 1987.

[Sari-89] Behcet Sarikaya, Srinivas Eswara, Vassilios Koukoulidis, *A Formal Specification Based Test Generation Tool*, ? Apr 1989

[SDL-88] CCITT Recommendation: Specification and Description Language SDL, CCITT Z.100, blue book, 1988

[Sidhu-89] D.P. Sidhu and T. -K. Leung, *Formal Methods for Protocol Testing: A Detailed Study*, IEEE Transactions on Software Engineering, Vol. 15, No. 4, pp. 413-426, April 1989.

[SKO-89] Morten Skov, *Implementation of physical and media access protocols for high-speed networks*, IEEE Communications Magazine, 1989.

[TR-90-x] H. Janssen, Y. Lu and P. Zhou, *Definition of a Protocol Data Structure Representation for Communication Protocols*, UBC Technical Report, planned for summer 1991.

[TR-90-y] Y. Lu and H. Janssen, *Integrating Estelle and ASN.1 for the generation of PDS*, UBC Technical Report, planned for summer 1991

[TR-90-z] P. Zhou and H. Janssen, *TSG constraints for PDS based test suite generation*, UBC Technical Report, planned for summer 1991

[Ural-87-1] H. Ural, *A Test Derivation Method for Protocol Conformance Testing*, University of Ottawa, Technical Report - TR-87-04, Jan. 87.

[Ural-87-2] H. Ural, *Test Sequence Selection Based on static data flow analysis*, Computer Communications, Vol. 10, No. 5, 1987, pp: 234-242.

[Ural-88] H. Ural, B. Yang, R. L. Probert, *A Test Sequence Selection Method for Protocols Specified in Estelle*, Technical Report TR-88-18, Department of Computer Science, University of Ottawa, June 1988.

[Vuong-88] S. T. Vuong, Allen C. Lau and R. Issac Chan, *Semiautomatic Implementation of Protocols Using an Estelle-C Compiler*, IEEE Transactions on Software Engineering, Vol. 14, No.3, March 1988.

[Vuong-89] S. T. Vuong, W. Y. L. Chan, and M. R. Ito, *The UIOv-Method for Protocol Test Sequence Generation*, Proceedings of the Second International Workshop on Protocol Test Systems, October 1989.

[Vuong-91-1] S. T. Vuong, H. Janssen and Y. Lu, *TESTGEN: An Environment for Test Suite Generation and Selection*, submitted to FORTE '91.

[Vuong-91-2] S. T. Vuong and J. Alilovic-Curgus, *A Metric Characterization of Infinite Computations in LOTOS*, submitted for publication, June 1991.

[KFNT-90] Kotaro Katsuyama,*et al. OSI Testing Environment Based on Standardized Formalisms*, FORTE '90

[Wvong-90] Russil Wvong, *A New Methodology for OSI Conformance Testing Based on Trace Analysis*, Master thesis, Department of Computer Science, UBC Oct 1990.

Appendix A

Estelle.Y BNF definition

```
specification ::=
        "specification" IDENTIFIER ";"
        body_definition
        "end." .

body_definition ::=
        declaration_part
        initialization_part
        state_trans_part .

declaration_part ::=
        constant_definition_part
        variable_declaration_part
        isp_declaration_part
        osp_declaration_part
        pdu_declaration_part
        timer_declaration_part
        state_definition_part .
```

```
constant_definition_part ::= ["CONST" constant_definition_group] .
constant_definition_group ::= +{ constant_definition ";"} .
constant_definition ::= IDENTIFIER "=" constant .
constant ::=
        INTEGER                 |
        CHARACTER_STRING        |
        "true"                  |
        "false" .

variable_declaration_part ::= [ "VAR" variable_declaration_group] .
variable_declaration_group ::= +{ variable_declaration ";"} .
variable_declaration ::= identifier_list ":" type_denotor .
identifier_list ::= IDENTIFIER {"," IDENTIFIER} .
type_denotor ::=
        "INT"         |
        "BOOLEAN"     |
        "CHAR_STR" .

isp_declaration_part  ::= "ISP" isp_declaration_group .
isp_declaration_group  ::= +{ isp_declaration } .
isp_declaration ::= isp_name pco_name ";" .
isp_name  ::= IDENTIFIER.
pco_name  ::= IDENTIFIER.

osp_declaration_part  ::= "OSP" osp_declaration_group .
osp_declaration_group  ::= +{ osp_declaration } .
osp_declaration ::= osp_name pco_name ";" .
```

```
osp_name ::= IDENTIFIER .

pdu_declaration_part  ::= "PDU" pdu_declaration_group .
pdu_declaration_group  ::= +{ pdu_declaration } .
pdu_declaration  ::=
        pdu_name send_recv_list ";"     |
        pdu_name pco_name ";" .
send_recv_list ::= send_recv sp_name { "," send_recv sp_name } .
pdu_name  ::= IDENTIFIER .
sp_name ::= IDENTIFIER .
send_recv ::= "sent_in" | "recv_in" .

timer_declaration_part ::= "TIMER" timer_declaration_group .
timer_declaration_group ::= +{ timer_declaration }.
timer_declaration  ::= timer_name timeout_value ";" .
timeout_value  ::= INTEGER .
timer_name  ::= IDENTIFIER .

initialization_part  ::= "INITIALIZATION" clause_group trans_block ";" .

state_definition_part  ::= "STATE" identifier_list ";" .

state_trans_part  ::= +{ state_trans_declaration } .
state_trans_declaration ::= "TRANS" transition_group .
transition_group  ::= +{ trans_declaration } .
trans_declaration  ::= from_clause trans_declaration_group .
from_clause  ::= "FROM" state_name .
```

```
trans_declaration_group ::= +{ trans_body_definition }.
trans_body_definition  ::= clause_group trans_block ";" .
clause_group  ::=
        to_clause
        when_clause
        provided_clause
        priority_clause
        output_clause
        trans_name .
to_clause  ::= "TO" state_name .
state_name  ::= IDENTIFIER .
when_clause  ::= { "WHEN" isp_or_pdu } .
isp_or_pdu ::= IDENTIFIER .
provided_clause ::= { "PROVIDED" bool_expression } .
bool_expression ::= expression .
priority_clause ::= { "PRIORITY" INTEGER | "PRIORITY"  IDENTIFIER } .
output_clause  ::= { "OUTPUT" osp_opdu_list } .
osp_opdu_list  ::= identifier_list .
trans_name  ::= { IDENTIFIER } .
trans_block  ::= statement_part .

statement_part  ::= { compound_statement } .
compound_statement  ::=
        "BEGIN" statement_sequence "END"        |
        "BEGIN" statement_sequence ";" "END" .
statement_sequence  ::= simple_statement { ";" simple_statement } .
simple_statement  ::=
        assignment_statement    |
```

```
        if_statement            |
        while_statement         |
        timer_statement .

assignment_statement  ::= variable_access ":=" expression .
variable_access  ::= IDENTIFIER { "." IDENTIFIER } .

if_statement  ::=
        if_prefix statement                |
        if_prefix statement else_part .
if_prefix ::= "IF" bool_expression "THEN" .
else_part  ::= "ELSE" statement .

statement  ::= simple_statement | compound_statement .

while_statement ::= "WHILE" bool_expression "DO" statement .

timer_statement ::=
        timer_operator "(" timer_name ")"        |
        "SET" "(" timer_name "," expression ")" .
timer_operator  ::= "RESET" | "START" | "STOP" .

expression ::=
        expression "+" expression        |
        expression "-" expression        |
        expression multop expression     |
        expression boolop expression     |
        expression relop expression      |
```

```
        "-" primary                    |
        "NOT" primary                  |
        primary                        |
        "NOT" expression               |
        "(" expression ")" .
primary  ::=
        constant_primary    |
        variable_access     |
        timer_expression .
constant_primary  ::= constant .
multop ::= "*" | "/" | "MOD" .
boolop ::= "AND" | "OR" .
relop ::= "=" | "<=" | "<" | ">=" | ">" | "<>" .

timer_expression  ::= timer_status "(" timer_name ")" .
timer_status  ::= "TIMEOUT" | "STARTED" | "STOPPED" | "RESET" | "READ" .

==================================
Metalanguage Symbols

Meta-symbol             Meaning

  ::=                   shall be defined to be
  |                     alternatively
  .                     end of definition
  [x]                   0 or 1 instance of x
  {x}                   0 or more instances of x
  +{x}                  1 or more instances of x
```

"xyz"	the terminal symbol xyz
meta-identifier	a nonterminal symbol

Appendix B

ASN.1 specification of FDDI MAC SPs and PDUs

```
FDDIMac DEFINITIONS ::=
BEGIN

MacToLlcAsp ::= CHOICE
        {
        MaUnitDataRequest,
        MaUnitDataIndication,
        MaUnitDataStatusIndication,
        MaTokenRequest
        }
MaUnitDataRequest ::= SEQUENCE
        {
        fcValue                 DataSymbolPair,
        destinationAddress      Address,
        mSDU                    OCTET STRING,
        requestedServiceClass   INTEGER {synchronous(0), asynchronous(1)},
```

```
        stream                  BOOLEAN DEFAULT FALSE,
        tokenClass              INTEGER
        }
MaUnitDataIndication  ::=  SEQUENCE
        {
        fcValue                 DataSymbolPair,
        destinationAddress      Address,
        sourceAddress           Address,
        mSDU                    OCTET STRING,
        receptionStatus         BOOLEAN
        }
MaUnitDataStatusIndication ::= SEQUENCE
        {
        numberOfSDUs INTEGER,
        transmissionStatus OCTET STRING, -- implementer defined
        providedServiceClass INTEGER {synchronous(0), asynchronous(1)}
        }
MaTokenRequest ::= SEQUENCE
        {
        requestedTokenClass INTEGER {restricted(0), nonrestricted(1)}
        }

PhyToMacAsp ::= CHOICE
        {
        PhUnitDataRequest,
        PhUnitDataIndication,
        PhUnitDataStatusIndication,
        PhInvalidIndication
```

```
        }

PhUnitDataRequest ::= SEQUENCE
        {
        phRequest Symbol
        }
PhUnitDataIndication ::= SEQUENCE
        {
        phIndication Symbol
        }
PhUnitDataStatusIndication ::= SEQUENCE
        {
        transmissionStatus BOOLEAN
        }
PhInvalidIndication ::= SEQUENCE
        {
        phInvalid INTEGER   -- this tpye is not defined in the standard
        }

MacPdu ::= CHOICE
        {
        Frame,
        Token
        }

Token ::=  SEQUENCE
        {
        pA      OCTET STRING,                -- 16 or more symbols
```

```
        sD      SD,
        fC      DataSymbolPair,
        eD      ControlSymbolPair
        }
Frame ::= SEQUENCE
        {
        pA      OCTET STRING,                -- 16 or more symbols
        sD      SD,
        fC      DataSymbolPair,
        dA      Address,
        sA      Address,
        info    InfoType,                    -- 0 or more data symbol pairs
        fCS     OCTET STRING (SIZE(4..4)),   -- 8 data symbols
        eD      TSymbol,
        fS      FSType                     -- 3 or more data symbols
        }

SD ::= SEQUENCE
        {
        j       JSymbol,
        k       KSymbol
        }
FSType ::= SEQUENCE
        {
        a       DataSymbol,
        c       DataSymbol,
        e       DataSymbol
        }
```

```
InfoType ::= SEQUENCE
        {
        first4Byte      OCTET STRING (SIZE(4..4)),
        restInfo        OCTET STRING
        }

MacToSmtAsps ::= CHOICE
        {
        SmMaInitProtocolReq,
        SmMaInitProtocolcfm,
        SmMaCtrlReq,
        SmMaStatusIndication
        }

SmMaInitProtocolReq ::= SEQUENCE
        {
        indivMACaddrS   OCTET STRING (SIZE(2..2)),
        indivMACaddrL   OCTET STRING (SIZE(6..6)),
        groupMACaddrs   OCTET STRING,
        tMin    INTEGER,
        tMax    INTEGER,
        tvx     INTEGER,
        tReq    INTEGER,
        tNeg    INTEGER,
        tNeg    INTEGER,
        tPri    INTEGER,
        indForOwnFr             BOOLEAN,
        indForRcvOnlyGoodFr     BOOLEAN
```

```
        }
SmMaInitProtocolcfm ::= SEQUENCE
        {
        status  BOOLEAN
        }
SmMaCtrlReq ::= SEQUENCE
        {
        ctrlAction      INTEGER    {macReset(0), beacon(1), presentStatus(2),
                                    resetCounters(3), interruptUponCond(4),
                                    sendBadFCS(5)},
        beaconInfo      OCTET STRING,
        requestedStatus MacStatus,
        requestedCond   INTEGER    {tkCaptured(0), frReceived(1), tkPassed(2)}
        }
MacStatus ::= SEQUENCE
{
        counterValue            INTEGER,    -- there are several Ct's
        currentTHTValue         INTEGER,
        currentTVXValue         INTEGER,
        currentTRTValue         INTEGER,
        rFlag                   INTEGER,
        currentRxState          INTEGER,
        currentTxState          INTEGER
}

SmMaStatusIndication ::= SEQUENCE
{
        statusReport    INTEGER
```

```
}

Address ::= CHOICE
{
        longAddress     INTEGER,      -- should be 48-bit BIT STRING
        shortAddress    INTEGER       -- should be 16-bit BIT STRING
}

JSymbol ::= ControlSymbol
KSymbol ::= ControlSymbol
TSymbol ::= ControlSymbol

Symbol          ::=     INTEGER         -- control symbol or data symbol
ControlSymbol   ::=     INTEGER         -- 5-bit BIT STRING
DataSymbol ::=  INTEGER -- 4-bit BIT STRING
DataSymbolPair  ::=  INTEGER -- 8-bit BIT STRING
ControlSymbolPair  ::=  INTEGER -- 10-bit BIT STRING

END
```

Appendix C

Excerpt of Estelle.Y Specification of FDDI MAC protocol

```
Specification fddi_1;

CONST
        I = 31: int;
        J = 24: int;
        K = 17: int;
        R = 7:  int;
        S = 25: int;
        T = 13: int;

        Restricted    = 1: int;
        Nonrestricted = 0: int;
        Void = 0: int;
        Implementer = 1: int;
        Claim_FR = 0: int;
        Beacon_FR = 1: int;
        S_Addrs = 0: int;
```

```
        L_Addrs = 0: int;

        zero = 0: int;
        one  = 1: int;
        Yes = true:  boolean;
        No  = false:  boolean;
        high   = 10: int;
        medium = 5: int;
        low    = 0: int;

        Mac_Reset = 0: int;
        Beacon = 1: int;
        NULL_STR = "": char_str;

        succ = 0: int;
        fail = 1: int;
VAR
        MSA:    int;
        MLA:    int;
        n:      int;

        T_Opr, T_Bid_Rc, T_Bid_Tx, T_Neg, T_Req, T_Max: int;

        Ring_Operational: boolean;
        Token_class: int;
        frame_Token_class: int;

        A_Flag, C_Flag, E_Flag, M_Flag, N_Flag, H_Flag, L_Flag, R_Flag: int;
```

```
        Frame_Ct, Error_Ct, Lost_Ct, Late_Ct: int;

        My_Claim, Lower_Claim, Higher_Claim, My_beacon, Other_beacon: boolean;
        Usable, FR_strip, FR_copied, nomoredata: boolean;

        Valid_Data_Length, Valid_FCS_Rc, Valid_Copy: boolean;
        Claim_FR_rec, Beacon_FR_rec: boolean;
        FCrIsToken, FCrEqNSA: boolean;

        Buffer_sD_J, Buffer_sD_K, Buffer_fC, Buffer_dA_L, Buffer_sA_L,
        Buffer_dA_S, Buffer_sA_S: int;
        Buffer_pA, Buffer_info, Buffer_fCS: char_str;
        Buffer_eD, Buffer_fS_a, Buffer_fS_c, Buffer_fS_e: int;
        frame_FC, frame_FC_Lr: int;
        frame_INFO, frame_FCS: char_str;
        frame_SA, frame_DA_L, frame_DA_S, frame_ED: int;
        req_service_class: int;
        Ex, Ax, Cx: int;
        num_of_frames, trans_status: int;
        Lr: int;

ISP
        MaUnitDataRequest               llc_mac;
        MaTokenRequest                  llc_mac;

        PhUnitDataIndication            mac_phy;
        PhUnitDataStatusIndication      mac_phy;
        PhInvalidIndication             mac_phy;
```

```
        SmMaInitProtocolReq             smt_mac;
        SmMaCtrlReq                     smt_mac;
OSP
        MaUnitDataIndication            llc_mac;
        MaUnitDataStatusIndication      llc_mac;

        PhUnitDataRequest               mac_phy;

        SmMaInitProtocolcfm             smt_mac;
        SmMaStatusIndication            smt_mac;

PDU
        Frame   sent_in PhUnitDataRequest,
                recv_in PhUnitDataIndication;

        Token   sent_in PhUnitDataRequest,
                recv_in PhUnitDataIndication;

TIMER
        TVX     2350;
        THT     0;
        TRT     165000;

STATE
        Rx_data, Ck_frame, Tx_data, Issue_TK, Claim_TK, Tx_beacon;

INITIALIZATION
```

```
        TO      Rx_data
            BEGIN
              Ring_Operational := true;
              nomoredata := true;
              FR_copied := false;
              FR_strip := false;
            END;

TRANS
        FROM    Rx_data
          TO    Rx_data
            WHEN  PhUnitDataIndication
            PROVIDED (PhUnitDataIndication.phIndication = I)
                        and (not Reset(TVX))
            PRIORITY  high
            OUTPUT PhUnitDataRequest
            BEGIN
              Reset(TVX);
              Start(TVX);
              PhUnitDataRequest.phRequest := I;
            END;

          TO    Rx_data
            WHEN  SmMaCtrlReq
            PROVIDED  (SmMaCtrlReq.ctrlAction = Mac_Reset)
            PRIORITY  high
            BEGIN
```

```
    T_Neg := T_Max;
  END;

TO    Rx_data
  WHEN  MaUnitDataRequest
  BEGIN
    frame_FC := MaUnitDataRequest.fcValue;
    frame_FC_Lr := (frame_FC/64) mod 2;
    if (frame_FC_Lr = one)
    then
      frame_DA_L := MaUnitDataRequest.destinationAddress.longAddress
    else
      frame_DA_S := MaUnitDataRequest.destinationAddress.shortAddress;
    frame_INFO := MaUnitDataRequest.mSDU;
    frame_SA := MSA;
    req_service_class := MaUnitDataRequest.requestedServiceClass;
    frame_Token_class := MaUnitDataRequest.tokenClass;
    nomoredata := No;
  END;

TO    Ck_frame
  WHEN  Frame
  BEGIN
    A_Flag := zero;
    C_Flag := zero;
    E_Flag := zero;
    N_Flag := zero;
    H_Flag := zero;
```

```
L_Flag := zero;
M_Flag := zero;

if (Frame.fC = 15) or (Frame.fC = 79)
then  N_Flag := one;

Lr := (Frame.fC/64) mod 2;
if (Frame.fC <> Void)
then
  if (Lr = zero and Frame.dA.shortAddress = S_Addrs) or
     (Lr = one and Frame.dA.longAddress = L_Addrs)
  then
  begin
    A_Flag := one;
    Buffer_pA := Frame.pA;
    Buffer_sD_J := Frame.sD.j;
    Buffer_sD_K := Frame.sD.k;
    Buffer_fC := Frame.fC;
    Buffer_dA_S := Frame.dA.shortAddress;
    Buffer_sA_S := Frame.sA.shortAddress;
    Buffer_dA_L := Frame.dA.longAddress;
    Buffer_sA_L := Frame.sA.longAddress;
    Buffer_info := Frame.info;
    Buffer_fCS := Frame.fCS;
    Buffer_eD := Frame.eD;
    Buffer_fS_a := Frame.fS.a;
    Buffer_fS_c := S;
    Buffer_fS_e := Frame.fS.e;
```

```
    FR_copied := Yes;
  end;

if (Lr = zero and Frame.sA.shortAddress = MSA and MSA > zero) or
   (Lr = one and Frame.sA.longAddress = MLA and MLA > zero)
then
  begin
  M_Flag := one;
  FR_strip := Yes;
end
else
  if (Lr = zero and Frame.sA.shortAddress > MSA and MLA = zero) or
     (Lr = one and Frame.sA.longAddress > MLA)
  then H_Flag := one
  else
    if (Frame.sA.shortAddress > zero) or (Frame.sA.longAddress > zero)
    then L_Flag := one;

T_Bid_Rc := Frame.info;
if (Frame.fC = Claim_FR)
then
begin
  if (T_Bid_Rc <> T_Req)
  then
    if L_Flag = one
    then
    begin
```

```
        H_Flag := one;
        L_Flag := zero;
      end
    else
      if H_Flag = one
      then
      begin
        L_Flag := one;
        H_Flag := zero;
      end;
    if (L_Flag = one) then FR_strip := Yes;
    Claim_FR_rec := Yes;
end;

Frame_Ct := Frame_Ct + one;
if (Valid_Data_Length and (Valid_FCS_Rc or (Frame.fC = Void or
  Frame.fC = Implementer)))
then
  begin
    Reset(TVX);
    Start(TVX);
    if A_Flag and Valid_Copy
    then C_Flag := one;
  end
else
  begin
    E_Flag := one;
    A_Flag := zero;
```

```
    H_Flag := zero;
    M_Flag := zero;
    L_Flag := zero;
  end;

if (Frame.fS.a = R) then N_Flag := zero;
if E_Flag = one then Ex := S else Ex := Frame.fS.e;
if A_Flag = one then Ax := S else Ax := Frame.fS.a;
if C_Flag = one then Cx := S else Cx := Frame.fS.c;

if (Frame.fS.c = S) and (M_Flag = one)
then trans_status := succ
else trans_status := fail;

if (Frame.fC = Claim_FR and A_Flag = 1 and M_Flag = 1)
then
  begin
  T_Neg := T_Bid_Rc;
  My_Claim := Yes;
  R_Flag := zero;
  Claim_FR_rec := Yes;
  end
else if (Frame.fC = Claim_FR and H_Flag = one)
then
  begin
    T_Neg := T_Bid_Rc;
    Higher_Claim := Yes;
    R_Flag := zero;
```

```
      Claim_FR_rec := Yes;
    end
  else if (Frame.fC = Claim_FR and L_Flag = one)
  then
    begin
      Lower_Claim := Yes;
      R_Flag := zero;
      Claim_FR_rec := Yes;
    end
  else if (Frame.fC = Beacon_FR and M_Flag = 1)
  then
    begin
      T_Neg := T_Max;
      My_beacon := Yes;
      R_Flag := zero;
      Beacon_FR_rec := Yes;
    end
  else if (Frame.fC = Beacon_FR and (M_Flag = 0 and E_Flag = 1))
  then
    begin
      T_Neg := T_Max;
      Other_beacon := Yes;
      R_Flag := zero;
      Beacon_FR_rec := Yes;
    end
  else if (E_Flag = 1) and (Frame.fS.e <> S)
  then Error_Ct := Error_Ct + one;
```

```
    END;

  TO Tx_data
    WHEN Token
    PROVIDED (Usable)
    OUTPUT Frame
    BEGIN
      Token_class := (Token.fC/64) mod 2;
      if (Token_class = Restricted)
      then
        if (R_Flag = zero)
        then
          begin
            R_Flag := one;
            SmMaStatusIndication.statusReport := 11;
          end
        else
        begin
          Reset(TVX);
          Start(TVX);
          R_Flag := zero;
        end;

      Stop(THT);
      if Late_Ct = zero
      then
        begin
          Set(THT, Read(TRT));
```

```
    Set(TRT, T_Opr);
    Start(TRT);
  end
else
  begin
    Set(THT, Timeout(THT));
    Late_Ct := zero;
  end;

Frame.pA    := zero;
Frame.sD.j  := J;
Frame.sD.k   := K;
Frame.fC    := frame_FC;
if (frame_FC_Lr = 1)
then
  Frame.dA.longAddress := frame_DA_L
else
  Frame.dA.shortAddress := frame_DA_S;
Frame.sA.longAddress    := frame_SA;
Frame.sA.shortAddress    := frame_SA;
Frame.info := frame_INFO;
Frame.fCS  := frame_FCS;
Frame.eD   := frame_ED;
Frame.fS.a   := R;
Frame.fS.c   := R;
Frame.fS.e   := R;

nomoredata:= Yes;
```

```
        END;

.....

      TO Claim_TK
        PROVIDED (Timeout(TVX) or (Timeout(TRT) and Late_Ct > zero) or
                    (Ring_Operational and T_Opr < T_Req))
        BEGIN
          T_Opr := T_Max;
          Set(TRT, T_Opr);
          Start(TRT);
          Ring_Operational := No;
        END;

      TO Tx_beacon
        WHEN SmMaCtrlReq
        PROVIDED (SmMaCtrlReq.ctrlAction = Beacon)
        BEGIN
          Set(TRT, T_Opr);
          Start(TRT);
        END;

TRANS
      FROM Tx_data

        TO Tx_data
        PROVIDED  (not nomoredata)
```

```
        OUTPUT Frame
        BEGIN
          Frame.sD.j := J;
          Frame.sD.k := K;
          Frame.fC := frame_FC;
          Frame.sA.shortAddress := frame_SA;
          Frame.sA.longAddress := frame_SA;
          Frame.dA.shortAddress := frame_DA_S;
          Frame.dA.longAddress := frame_DA_L;
          Frame.info := frame_INFO;
          Frame.eD := T;
          Frame.fS.e := R;
          Frame.fS.a := R;
          Frame.fS.c := R;
        END;

.....

        TO Tx_data
          WHEN  MaUnitDataRequest
          BEGIN
            frame_FC := MaUnitDataRequest.fcValue;
            frame_FC_Lr := (frame_FC/64) mod 2;
            if (frame_FC_Lr = one)
            then
              frame_DA_L := MaUnitDataRequest.destinationAddress.longAddress
            else
              frame_DA_S := MaUnitDataRequest.destinationAddress.shortAddress;
            frame_INFO := MaUnitDataRequest.mSDU;
```

```
        frame_SA := S_Addrs;
        req_service_class := MaUnitDataRequest.requestedServiceClass;
        frame_Token_class := MaUnitDataRequest.tokenClass;
        nomoredata := No;
      END;
.....
    TO Rx_data
      WHEN SmMaCtrlReq
      PROVIDED  (SmMaCtrlReq.ctrlAction = Mac_Reset)
      BEGIN
        T_Opr := T_Max;
        if (Ring_Operational or Late_Ct = zero)
        then
        begin
          Set(TRT, T_Opr);
          Start(TRT);
          Late_Ct := one;
          Ring_Operational := No;
        end;
      END;

    TO Claim_TK
      PROVIDED (Timeout(TVX) or (Timeout(TRT) and Late_Ct > zero) or
                  (Ring_Operational and T_Opr < T_Req))
      BEGIN
        T_Opr := T_Max;
        Set(TRT, T_Opr);
        Start(TRT);
```

```
        Ring_Operational := No;
      END;

TRANS
    FROM Issue_TK

      TO Rx_data
      OUTPUT Token
      ;

      TO Issue_TK
      WHEN Frame
      ;

      TO Rx_data
        WHEN SmMaCtrlReq
        PROVIDED  (SmMaCtrlReq.ctrlAction = Mac_Reset) or
                    (SmMaCtrlReq.ctrlAction = Beacon)
        BEGIN
          T_Opr := T_Max;
          if (Ring_Operational or Late_Ct = zero)
          then
            begin
              Set(TRT, T_Opr);
              Start(TRT);
              Late_Ct := one;
              Ring_Operational := No;
            end;
```

```
        END;

      TO Claim_TK
        PROVIDED (Timeout(TVX) or (Timeout(TRT) and Late_Ct > 0) or
                    (Ring_Operational and T_Opr < T_Req))
        BEGIN
          T_Opr := T_Max;
          Set(TRT, T_Opr);
          Start(TRT);
          Ring_Operational := No;
        END;

TRANS
      FROM Ck_frame
        TO Rx_data
          PROVIDED (FR_strip and not Claim_FR_rec and not Beacon_FR_rec)
          PRIORITY high
          OUTPUT MaUnitDataStatusIndication
          BEGIN
            MaUnitDataStatusIndication.numberOfSDUs := 1;
            MaUnitDataStatusIndication.transmissionStatus := trans_status;
            MaUnitDataStatusIndication.providedServiceClass := 1;
          END;

......

end.
```

Appendix D

TESTGEN menus

```
##############################################
#   Test Suite Generation - Prototype V.2  #
##############################################

#=================================================================#
#                          TSG Main Menu                          #
#                          =============                          #
#                                                                 #
# Enter:                                                          #
#  p to access the parser menu,                                   #
#  v to access the PDS verification menu,                         #
#  c to set or modify the TSG and SPP constraints,                #
#  g to access the test case generation menu.                     #
#                                                                 #
#  - to return to previous menu,                                  #
#  h for help,                                                    #
#  e to exit TSG.                                                 #
#=================================================================#
```

```
#=====================================================================#
#                     TSG Estelle/ASN.1 parser                        #
#                     ========================                        #
#                                                                     #
# Ready to create PDS:                                                #
#  - loading: asn1.tt                                                 #
#  - parsing: spec.txt                                                #
# Enter:                                                              #
#  1 to create the PDS as specified,                                  #
#  2 to specify another ASN.1 file to load,                           #
#  3 to specify another specification file to parse.                  #
#                                                                     #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                     TSG  PDS verification                           #
#                     =====================                           #
#                                                                     #
# These functions allow to verify the parsed PDS. Enter:              #
#  0 to check the consistency of the parsed PDS,                      #
#                                                                     #
#  1 to display STATES,                                               #
#  2 to display TRANSITIONS,                                          #
#  3 to display ISPs (Input Service Primitives)                       #
#  4 to display OSPs (Output Service Primitives)                      #
#  5 to display PDUs (Protocol Data Units)                            #
#  6 to display Parameter of Service Primitive                        #
#  7 to display CONSTANTS,                                            #
#  8 to display VARIABLES,                                            #
#  9 to display TIMER,                                                #
#                                                                     #
#  1 to access the lower level PDS verification menu.                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                 TSG  low level PDS verification                     #
#                 ===============================                     #
#                                                                     #
# These functions allow to verify the parsed PDS. Enter:              #
#  1 to display Compound statements                                   #
#  2 to display If statements                                         #
#  3 to display Assignment statements                                 #
#  4 to display Timer statements                                      #
#  5 to display Timer expressions                                     #
#  6 to display Expressions                                           #
#  9 to display Efect Functions,                                      #
#                                                                     #
#  v for high level PDS verification,                                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                 Default setting of TSG Constraints                  #
#                 ==================================                  #
#                                                                     #
# STATES:           min-use = 0       max-use = 99                    #
# TRANSITIONS:      min-use = 0       max-use = 99                    #
# ISP:              min-use = 0       max-use = 1                     #
# OSP:              min-use = 0       max-use = 1                     #
# PDU:              min-use = 0       max-use = 1                     #
# CONSTANTS:        min-use = 0       max-use = 99                    #
# VARIABLES:        min-use = 0       max-use = 99                    #
#                   min-assigned = 0  max assigned = 99               #
# TIMER Operations: min-start = 0     max-start = 99                  #
#                   min-stop  = 0     max-stop  = 99                  #
#                   min-set   = 0     max-set   = 99                  #
#                   min-reset = 0     max-reset = 99                  #
# TIMER Condition checks:                                             #
#    min-read          = 0            min-read          = 99         #
#    min-check-reseted = 0            min-check-reseted = 99         #
#    min-check-started = 0            min-check-started = 99         #
#    min-check-stopped = 0            min-check-stopped = 99         #
#    min-check-timed-out = 0          min-check-timed_out = 99       #
#                                                                     #
#=====================================================================#
```

```
#=====================================================================#
#                  Default setting of ISP Parameter                   #
#                  ================================                   #
#                                                                     #
# ISP: MaUnitDataRequest                                              #
# +------+-------------------------------------------------------+    #
# | Type | recognized ISP parameter name                         |    #
# +------+-------------------------------------------------------+    #
# | int  | MaUnitDataRequest.requestedServiceClass               |    #
# | bool | MaUnitDataRequest.stream                              |    #
# | ...  | ...                                                   |    #
# +------+-------------------------------------------------------+    #
#                                                                     #
# ISP: MaTokenRequest                                                 #
# +------+-------------------------------------------------------+    #
# | Type | recognized ISP parameter name                         |    #
# +------+-------------------------------------------------------+    #
# | int  | MaTokenRequest.requestedTokenClass                    |    #
# +------+-------------------------------------------------------+    #
#                                                                     #
# ISP: SmMaInitProtocolReq                                            #
# +------+-------------------------------------------------------+    #
# | Type | recognized ISP parameter name                         |    #
# +------+-------------------------------------------------------+    #
# | char*| SmMaInitProtocolReq.indivMACaddrS                     |    #
# | bool | SmMaInitProtocolReq.indForRcvOnlyGoodFr               |    #
# | ...  | ...                                                   |    #
# +------+-------------------------------------------------------+    #
# ...                                                                 #
#                                                                     #
# PDU: Frame                                                          #
# +------+-------------------------------------------------------+    #
# | Type | recognized PDU parameter name                         |    #
# +------+-------------------------------------------------------+    #
# | char*| Frame.pA                                              |    #
# | int  | Frame.sD.j                                            |    #
# | ...  | ...                                                   |    #
# +------+-------------------------------------------------------+    #
# ...                                                                 #
#                                                                     #
# NOTE: Constraints are only defined for the parameters listed above. #
# Booleans (bool ) parameters are constraint to: TRUE, FALSE          #
# Interger (int  ) parameters are constraint to: 0, 1, 999            #
# Cstrings (char*) parameters are constraint to: 'parm_val1'          #
#=====================================================================#
```

```
#=====================================================================#
#                  Default setting of PDU Parameter                   #
#                  ================================                   #
#                                                                     #
# PDU: Frame                                                          #
# +------+-------------------------------------------------------+ #
# | Type | recognized PDU parameter name                          | #
# +------+-------------------------------------------------------+ #
# | char*| Frame.pA                                               | #
# | int  | Frame.sD.j                                             | #
# | int  | Frame.sD.k                                             | #
# | ...  | ...                                                    | #
# +------+-------------------------------------------------------+ #
#                                                                     #
# PDU: Token                                                          #
# +------+-------------------------------------------------------+ #
# | Type | recognized PDU parameter name                          | #
# +------+-------------------------------------------------------+ #
# | char*| Token.pA                                               | #
# | int  | Token.sD.j                                             | #
# | int  | Token.sD.k                                             | #
# | ...  | ...                                                    | #
# +------+-------------------------------------------------------+ #
#                                                                     #
# NOTE: Constraints are only defined for the parameters listed above. #
# Booleans (bool ) parameters are constraint to: TRUE, FALSE          #
# Interger (int  ) parameters are constraint to: 0, 1, 999            #
# Cstrings (char*) parameters are constraint to: 'parm_val1'          #
#=====================================================================#
```

```
#=====================================================================#
#                    Interactive Constraint Definition                #
#                    =================================                #
#                                                                     #
# The Test Suite Generation (TSG) allow to control the subtour        #
# selection mechanism. Enter:                                         #
#  1 to reset the TSG constraints to default values,                  #
#  t to call the TSG constraint editor.                               #
#                                                                     #
# The Service Primitive Parameter (SPP) constraints and the Protocol  #
# Data Unit Parameter (PDUP) constraints define the values of         #
# parameters of the service primitives sent to the IUT. Enter:        #
#  2 to reset the SPP constraints to default values,                  #
#  s to call the SPP constraint editor,                               #
#                                                                     #
# The Service Primitive Parameter (SPP) constraints and the Protocol  #
# Data Unit Parameter (PDUP) constraints define the values of         #
# parameters of the service primitives sent to the IUT. Enter:        #
#  3 to reset the PDUP constraints to default values,                 #
#  u to call the PDUP constraint editor.                              #
#                                                                     #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                    Interactive Constraint Definition                #
#                    =================================                #
#                                                                     #
# These functions allow to view and specify constraints on the        #
# the following Primitives. Enter:                                    #
#  1 to specify STATES constraints,                                   #
#  2 to specify TRANSITIONS constraints,                              #
#  3 to specify ISPs (Input Service Primitives) constraints,          #
#  4 to specify OSPs (Output Service Primitives) constraints,         #
#  5 to specify PDUs (Protocol Data Units) constraints,               #
#  6 to specify CONSTANTS constraints,                                #
#  7 to specify VARIABLES constraints,                                #
#  8 to specify TIMER constraints,                                    #
#                                                                     #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                  TSG Constraints Editor ( STATE )                   #
#                  ================================                   #
#                                                 (a)       (b)       #
# +-----+-----------------------------------------+---------+---------+ #
# | key | Name                                    | min-use | max-use | #
# +-----+-----------------------------------------+---------+---------+ #
# | 0   | Rx_data                                 | 0       | 99      | #
# | 1   | Ck_frame                                | 0       | 99      | #
# | 2   | Tx_data                                 | 0       | 99      | #
# | ... | ...                                     |         |         | #
# +-----+-----------------------------------------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a state constraint.     #
#  (ex: '2a3' will set the min-use (a) of state 2 (key) to 3 (value)) #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#


#=====================================================================#
#                TSG Constraints Editor ( TRANSITION )                #
#                =====================================                #
#                                                 (a)       (b)       #
# +-----+-------------+-----------+-----------+---------+---------+ #
# | key | Name        | ISP name  | OSP name  | min-use | max-use | #
# +-----+-------------+-----------+-----------+---------+---------+ #
# | 0   | -           | PhUnitData | PhUnitData | 0       | 99      | #
# | 1   | -           | SmMaCtrlRe | -          | 0       | 99      | #
# | 2   | -           | MaUnitData | -          | 0       | 99      | #
# | 3   | -           | -          | -          | 0       | 99      | #
# | 4   | -           | -          | -          | 0       | 99      | #
# | 5   | -           | -          | -          | 0       | 99      | #
# | 6   | -           | -          | -          | 0       | 99      | #
# | 7   | -           | SmMaCtrlRe | -          | 0       | 99      | #
# | ... | ...         |            |            |         |         | #
# +-----+-------------+-----------+-----------+---------+---------+ #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a trans constraint.     #
#  (ex: '2a3' will set the min-use (a) of trans 2 (key) to 3 (value)) #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
```

```
#=====================================================================#
#                   TSG Constraints Editor ( ISP )                    #
#                   ==============================                    #
#                                                (a)       (b)        #
# +-----+-----------------------------------------+---------+---------+ #
# | key | Name                                    | min-use | max-use | #
# +-----+-----------------------------------------+---------+---------+ #
# | 0   | MaUnitDataRequest                       | 0       | 1       | #
# | 1   | MaTokenRequest                          | 0       | 1       | #
# | 2   | PhUnitDataIndication                    | 0       | 1       | #
# +-----+-----------------------------------------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a ISP constraint.       #
#  (ex: '2a3' will set the min-use (a) of isp 2 (key) to 3 (value))   #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#


#=====================================================================#
#                   TSG Constraints Editor ( OSP )                    #
#                   ==============================                    #
#                                                                     #
#                                                (a)       (b)        #
# +-----+-----------------------------------------+---------+---------+ #
# | key | Name                                    | min-use | max-use | #
# +-----+-----------------------------------------+---------+---------+ #
# | 0   | MaUnitDataIndication                    | 0       | 1       | #
# | 1   | MaUnitDataStatusIndication              | 0       | 1       | #
# | 2   | PhUnitDataRequest                       | 0       | 1       | #
# +-----+-----------------------------------------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a OSP constraint.       #
#  (ex: '2a3' will set the min-use (a) of osp 2 (key) to 3 (value))   #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                   TSG Constraints Editor ( PDU )                    #
#                   ==============================                    #
#                                                                     #
#                                              (a)       (b)          #
# +-----+-----------------------------------+---------+---------+    #
# | key | Name                              | min-use | max-use |    #
# +-----+-----------------------------------+---------+---------+    #
# | 0   | Frame                             | 0       | 1       |    #
# | 1   | Token                             | 0       | 1       |    #
# +-----+-----------------------------------+---------+---------+    #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a PDU constraint.      #
#  (ex: '2a3' will set the min-use (a) of pdu 2 (key) to 3 (value))   #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#


#=====================================================================#
#                 TSG Constraints Editor ( CONSTANTS )                #
#                 ====================================                #
#                                                                     #
#                                              (a)       (b)          #
# +-----+-----------------------------------+---------+---------+    #
# | key | Name                              | min-use | max-use |    #
# +-----+-----------------------------------+---------+---------+    #
# | 0   | I                                 | 0       | 99      |    #
# | 1   | J                                 | 0       | 99      |    #
# | 2   | K                                 | 0       | 99      |    #
# | 3   | R                                 | 0       | 99      |    #
# | 4   | S                                 | 0       | 99      |    #
# | 5   | T                                 | 0       | 99      |    #
# +-----+-----------------------------------+---------+---------+    #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a const constraint.    #
#  (ex: '2a3' will set the min-use (a) of const 2 (key) to 3 (value)) #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#                  TSG Constraints Editor ( VARIABLES )               #
#                  ====================================               #
#                                                                     #
#                              (a)       (b)       (c)       (d)      #
# +-----+------------------+---------+---------+---------+---------+ #
# | key | Name             |         |         | min-    | max-    | #
# | key | Name             | min_use | max_use | assigned| assigned| #
# +-----+------------------+---------+---------+---------+---------+ #
# | 0   | MSA              | 0       | 99      | 0       | 99      | #
# | 1   | MLA              | 0       | 99      | 0       | 99      | #
# | 2   | n                | 0       | 99      | 0       | 99      | #
# | 3   | T_Opr            | 0       | 99      | 0       | 99      | #
# | 4   | T_Bid_Rc         | 0       | 99      | 0       | 99      | #
# | 5   | T_Bid_Tx         | 0       | 99      | 0       | 99      | #
# +-----+------------------+---------+---------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' [ab] 'value' :to change the value of a VARIABLE constraint.  #
#  (ex: '2a3' will set the min-use (a) of var 2 (key) to 3 (value))   #
#                                                                     #
#  t to return to the main TSG constraint editor menu                 #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=================================================================#
#                  TSG Constraints Editor ( TIMER )               #
#                  ================================               #
#                                                                 #
# Timer actions     | (a)   (b)   (c)   (d)   (e)   (f)   (g)   (h)  #
# +-----+----------+-----+-----+-----+-----+-----+-----+-----+-----+ #
# |     |          | min | max | min | max | min | max | min | max | #
# | key | Name     |start|start|stop |stop | set | set |reset|reset| #
# +-----+----------+-----+-----+-----+-----+-----+-----+-----+-----+ #
# | 0   | TVX      | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | #
# | 1   | THT      | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | #
# | 2   | TRT      | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | #
# +-----+----------+-----+-----+-----+-----+-----+-----+-----+-----+ #
#                                                                 #
# Timer conditions| (i)   (j)   (k)   (l)   (m)   (n)   (o)   (p)   (q)   (r)
# +-----+---------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
# |     |         | min | max | min | max | min | max | min | max | min | max |
# | key | Name    |check read |check reset|check start|check stopd|check timot|
# +-----+---------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
# | 0   | TVX     | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  |
# | 1   | THT     | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  |
# | 2   | TRT     | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  | 0   | 99  |
# +-----+---------+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
#                                                                 #
# Enter                                                           #
#  'key' [ab] 'value' :to change the value of a timer constraint. #
#  (ex: '2a3' will set the min-use (a) of timer 2 (key) to 3 (value)) #
#                                                                 #
#  t to return to the main TSG constraint editor menu             #
#=================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=================================================================#
```

```
#=====================================================================#
#           Interactive Service Primitive Parameter Definition        #
#           ================================================          #
#                                                                     #
# +-----+---------------------------------------+---------+---------+ #
# | key | Input Service Primitive name          | # parm. | # inst. | #
# +-----+---------------------------------------+---------+---------+ #
# |   0 | MaUnitDataRequest                     | 7       | 64      | #
# |   1 | MaTokenRequest                        | 1       | 2       | #
# |   2 | PhUnitDataIndication                  | 1       | 2       | #
# |   3 | PhUnitDataStatusIndication            | 1       | 2       | #
# |   4 | PhInvalidIndication                   | 1       | 2       | #
# |   5 | SmMaInitProtocolReq                   | 11      | 256     | #
# |   6 | SmMaCtrlReq                           | 10      | 512     | #
# +-----+---------------------------------------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' to define the parameter of the corresponding ISP,            #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=====================================================================#
#           Interactive Protocol Data Unit Parameter Definition       #
#           ===================================================       #
#                                                                     #
# +-----+---------------------------------------+---------+---------+ #
# | key | Protocol Data Unit name               | # parm. | # inst. | #
# +-----+---------------------------------------+---------+---------+ #
# |   0 | Frame                                 | 14      | 2048    | #
# |   1 | Token                                 | 5       | 16      | #
# +-----+---------------------------------------+---------+---------+ #
#                                                                     #
# Enter                                                               #
#  'key' to define the parameter of the corresponding PDU,            #
#=====================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit #
#=====================================================================#
```

```
#=========================================================================#
#                      Test Suite Generation                              #
#                      =====================                              #
#                                                                         #
# Enter:                                                                  #
#  1 to estimate the number of subtours,                                  #
#  2 to identify and store the subtours,                                  #
#  3 to store and print the identified subtours,                          #
#                                                                         #
#  5 to generate test cases for all subtours.                             #
#                                                                         #
#=========================================================================#
# -:previous, h:help, m:main, p:parser, c:constraints, g:gen...e:exit     #
#=========================================================================#
```

Appendix E

Consistency Requirements of PDS

Following are consistency requirements that must be satisfied by a protocol data structure. The defined conditions reflect the properties of the Protocol Data Structure. The conditions are used to verify the consistency of a protocol data structure.

General conditions on the Protocol Data Structure:

General conditions is examplified with STATE structure. Similar conditions apply to TRANS, ISP, OSP, PDU, SPPARM, CONST, VAR, TIMER, EXPR, TEXPR, AFN, ASTMT, IFSTMT, WSTMT and TSTMT.

```
ppds->nb_of_states <= MAXSTATES
ppds->nb_of_states == # (instances of STATE)
for all  0 < i < ppds->nb_of_states
   ppds->pstate[i]->key = i
for all  i >= nb_of_states
   ppds->pstate[i] == NULL pointer
for all STATE instances
   ppds->pstate[i]->key = i
```

where ppds is the pointer to main data structure of a PDS.

```
STATE:
for all instances of STATE (i.e. states),
 for all  0 < i < nb_of_tr
    ppds->pstate[tr_key[i]]->form_state = key;
 for all  i >= nb_of_tr
    tr_key[i] == NULL pointer;

TRANSITION:
for all instances of TRANS (i.e. transitions),
 there exists an i so that: ppds->pstate[from_st]->tr_key[i] == key;

 from_st < ppds->nb_of_states
 to_st < ppds->nb_of_states
 isp < ppds->nb_of_isps
 osp < ppds->nb_of_osps
 osp2 < ppds->nb_of_osps
 ipdu < ppds->nb_of_pdus
 opdu < ppds->nb_of_pdus
 opdu2 < ppds->nb_of_pdus
 epred < ppds->nb_of_exprs
 afn < ppds->nb_of_afns

ISP:
for all input service primitives,
 nb_of_pdus < ppds->nb_of_pdus;
```

```
 for all  0 < i < nb_of_pdus;
    ppds->ppdu[pdu_key[i]]->sent_in == key;
 for all  i >= nb_of_pdus;
    pdu_key[i] == NULL pointer;

OSP:
for all output service primitives,
 nb_of_pdus < ppds->nb_of_pdus;

 for all  0 < i < nb_of_pdus;
    ppds->ppdu[pdu_key[i]]->recv_in == key;
 for all  i >= nb_of_pdus;
    pdu_key[i] == NULL pointer;

PDU:
for all protocol data units,
(sent_in != NONE) OR (recv_in != NONE) OR (pco != NONE);
if (sent_in != NONE) there exists an i so that:
 ppds->pisp[sent_in]->pdu_key[i] == key;
if (recv_in != NONE) there exists an i so that:
 ppds->posp[recv_in]->pdu_key[i] == key;

CONSTANT:
for all constants,
type must be a member of set {BOOLEAN_TYPE, INTEGER_TYPE, CHAR_STRING_TYPE}
if (type == BOOLEAN_TYPE)
{  int_value == 0;
   char_ptr == NULL pointer;
```

```
}
if (type == INTEGER_TYPE)
{  boolean_value == 0;
   char_ptr == NULL pointer;
}
if (type == CHAR_STRING_TYPE)
{  int_value == 0;
   boolean_value == 0;
}

VARIABLE:
for all variables,
type must be a member of set {BOOLEAN_TYPE, INTEGER_TYPE, CHAR_STRING_TYPE}

TIMER:
======
 timeout_value > 0;

Help function:
=============
_result_type( x_kind, x, check_type)
{
  if (x_kind == VAR)
     pds->pvar[right]->type == check_type;
  if (x_kind == CONSTANT)
     pds->pconst[right]->type == check_type;
  if (x_kind == EXPR)
     _result_type(pds->pexpr[right]) == check_type;
```

```
}

EXPRESSION:
for all expressions,
if (operator == not)
{ left_kind == NONE;
  right_kind != NONE;
  _result_type(right_type, right, BOOLEAN_TYPE)
}

if ((operator == and) or (operator == or))
{ left_kind != NONE;
  right_kind != NONE;
  _result_type(left_type, left, BOOLEAN_TYPE)
  _result_type(right_type, right, BOOLEAN_TYPE)
}

if (operator is one of {+, -, *, div, mod})
{ left_kind != NONE;
  right_kind != NONE;
  _result_type(left_type, left, INTEGER_TYPE)
  _result_type(right_type, right, INTEGER_TYPE)
}

if (operator is one of {=, <, >, >=, <=, !=})
{ left_kind != NONE;
  right_kind != NONE;
  _result_type(left_type, left, INTEGER_TYPE)
```

```
  _result_type(right_type, right, INTEGER_TYPE)
}

Timer EXPRession:
status is one of {RESET, STARTED, STOPPED, TIMED_OUT}

ACTION FUNCTION:
for all  i < nb_of_statements
{ stmt_kind[i] is one of { ASTMT, IFSTMT, TSTMT }
  if (stmt_kind[i] == ASTMT) stmt_key[i] < MAXASTMT;
  if (stmt_kind[i] == IFSTMT) stmt_key[i] < MAXIFSTMT;
  if (stmt_kind[i] == TSTMT) stmt_key[i] < MAXTSTMT;
}

IF Statement:
=============
_result_type(expr, bool_expr, BOOLEAN_TYPE)
stmt_kind  is one of { ASTMT, IFSTMT, TSTMT }
else_stmt_kind  is one of { ASTMT, IFSTMT, TSTMT }

Assignment Statement:
=====================
The type of left kind must be the same as the type of right expression.

Timer Statements:
=================
timer_operator is one of {START, RESET, STOP}
```