NOTICE

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

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

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

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

AVIS

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

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

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

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

ISBN 0-315-56122-X

Canada

AUTOMATIC TEST SUITE DERIVATION FROM ESTELLE SPECIFICATIONS

Behdad Forghani

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Masters of Engineering at
Concordia University
Montréal, Québec, Canada

February, 1990

ABSTRACT

Automatic Test Suite Derivation From Estelle Specifications

Behdad Forghani

This thesis develops a methodology to automatically derive conformance tests in TTCN (Tree and Tabular Combined Notation) from an Estelle specification of a protocol. A software tool implementing this methodology is described. The tool is integrated with a test generation tool previously developed(CONTEST—ESTL) and a TTCN editor. The input is Estelle specification, the output of TTCNGEN is the test cases in TTCN notation. The test generation tool transforms a given Estelle specification to a normalized form and the test cases (subtours) are derived to cover the data flow functions of the protocol. TTCN test generation technique is based on first converting the enabling conditions of transitions to a conjunctive normal form and then automatically generating the dynamic behaviour from Estelle transitions. Each normalized transition in the Estelle specification is translated to a TTCN test step. Spontaneous transitions give alternatives to the events in these test steps. A TTCN test case is obtained from each subtour by attaching the test steps corresponding to the transitions in the subtour. Constraints are derived from the enabling conditions and the actions related to the ASP/ PDU parameters. This way test suite derivation process is largely automated. The ISDN LAP-D and a simplified transport class 2 protocols are taken as examples throughout the thesis.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Behcet Sarikaya for guiding and supporting me throughout my graduate studies. He was always ready to listen to my questions and problems and gave thoughtful answers to my question and solution to my problems. I would also like to thank him specially for his support and guiding during the period of preparation of this thesis.

I would also like to thank my parents Ashrafolmolok and Mehdi for their continuous support, encouragement and understandings.

DEDICATION

I dedicate this thesis to my parents

Ashrafolmolok and Mehdi

Table of Contents

List of Figures

List of Tables

Chapter 1 INTRODUCTION

In 1977 International Organization for Standardization (ISO) established a subcommittee to develop a structure or architecture that defines communication tasks. This way standards can be easily defined for that architecture. The result of that subcommittee was the so called Open System Interconnection (OSI) reference model. This reference model provides a common basis for the coordination of standards development for the purpose of systems interconnection. The term "open" denotes the ability of any two systems conforming to the reference model and associated standards to connect. The technique chosen by ISO for structuring the reference model is layering. Within a system there are one or more active entities which implement the functions of that layer (usually referred as (N) layer). Each entity communicates with entities above and below across an interface. This interface is called Service Access Point (SAP). The (N-1) entity provides services to an (N) entity via invocation of primitives. A primitive specifies the function to be performed and is used to pass data and control information [1]. Figure 1 shows the layer concept of OSI reference model.

Principles used in defining OSI layers are[2]:

1. Do not create so many layers to make the system engineering task of describing and integrating layers more difficult than necessary.
2. Create boundary at a point where the description of services can be small and the number of interactions are minimized.

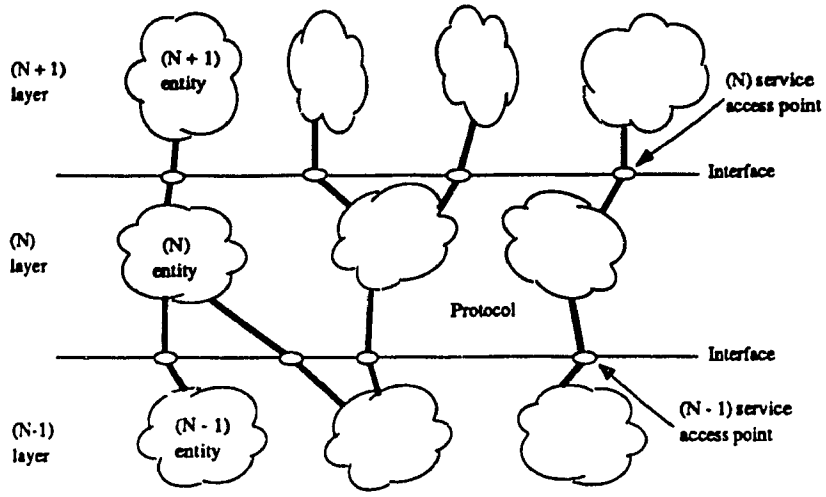


Figure 1 Layer Concept of OSI Reference Model[1].

3. Create a layer where there is a need for a different level of abstraction in the handling of data (e.g., morphology, syntax, semantics).
4. Allow changes of functions or protocols to be made without affecting other layers.

The resulting OSI reference model has seven layers. These layers are:

1. Physical :

   Concerned with transmission of unstructured bit stream over physical medium; deals with the mechanical, electrical, functional, and procedural characteristics to access the physical medium.
2. Data Link :

   Provides for the reliable transfer of information across the physical link; sends

blocks of data (frames) with the necessary synchronization,error control, and flow control.

3. Network :

   Provides upper layers with independence from the data transmission and switching technologies used to connect systems; responsible for establishing, maintaining,and terminating connections.

4. Transport :

   Provides reliable, transparent transfer of data between end points; provides end-to-end error recovery and data flow control.

5. Session :

   Provides the control structure for communication between applications; establishes, manages, and terminates connections (sessions) between cooperating applications.

6. Presentation :

   Provides independence to the application processes from differences in data representation (syntax).

7. Application :

   Provides access to the OSI environment for users and also provides distributed information services.

1.1 Basic Elements of Communication Protocols

The OSI model is connection-oriented. Two (N) entities, also called peer entities communicate, using a protocol, by means of an (N–1) connection. This

logical connection is provided by (N–1) entities between (N–1) SAPs. Figure 2 shows the OSI principles in operation. When application X has a message to send to application Y, it transfers those data to an application protocol entity. A header is appended to the data that contains the required information for the peer layer 7 protocol (encapsulation). This process continues down to layer 2, which generally adds both a header and a trailer. This layer 2 unit is called a frame and is passed by the physical layer onto the transmission medium.

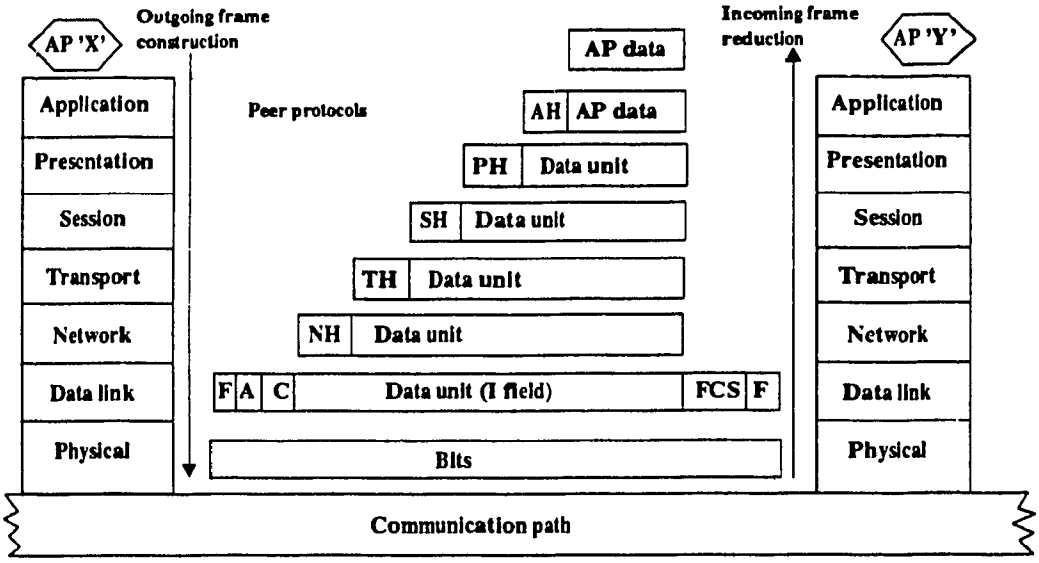


Figure 2 OSI operation[1].

A protocol consists of procedural rules for dialogue and precisely defined formats for each type of Protocol Data Unit (PDU) that can be used within the protocol. The unit of information to be transferred between (N + 1)-entities via (N)-service is called the (N)-Service-Data-Unit, or (N)-SDU.

The **REQUEST** primitive type is used when a higher layer is requesting a service form the next lower layer.

The **INDICATION** primitive type is used by a layer providing a service to notify the next higher layer of any specific activity which is service related. The **INDICATION** primitive may be the result of an activity of the lower layer related to the primitive type **REQUEST** at the peer entity.

The **RESPONSE** primitive type is used by a layer to acknowledge receipt, from a lower layer, of the primitive type **INDICATION**.

The **CONFIRM** primitive type is used by the layer providing the requested service to confirm that the activity has been completed.

Figure 3 shows the concept of a layer and interlayer communication. In the simplest case, the whole (N)-SDU fits into the user data field on the (N)-PDU, and (N)-PDU's map one-to-one into (N-1)-SDU's. However, the relationship between these three data units can be more complex. The OSI Reference Model defines three mappings[3]:

1. segmenting/reassembling, where one (N)-SDU is mapped into multiple (N)-PDU's;
2. blocking/deblocking, where multiple (N)-SDU's are mapped into a single (N)-PDU;
3. concatenation/separation, where multiple (N)-PDU's are mapped into a single (N-1)-SDU.

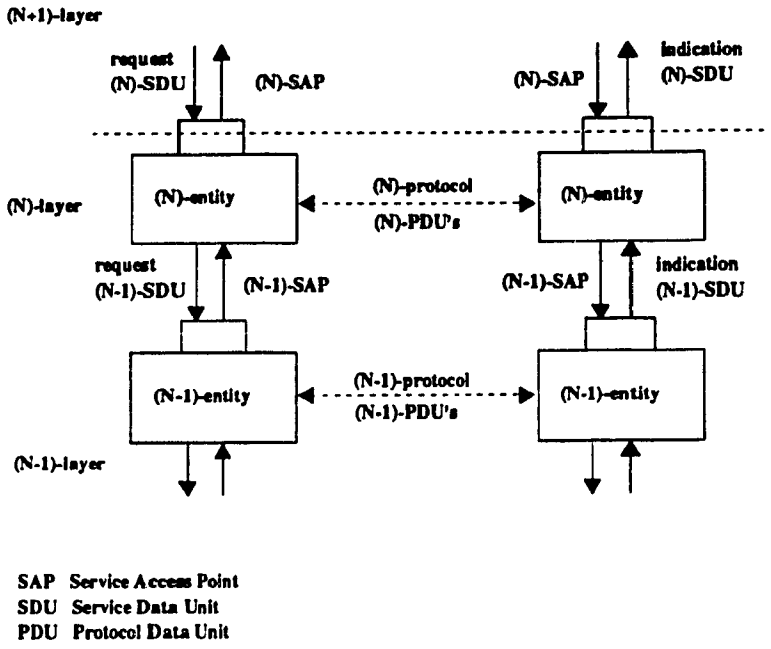


Figure 3 Interlayer communication[3].

In general, blocking and concatenation improve performance: they reduce the number of (N-1)-SDU's needed to transfer the (N)-SDU's and various control (N)-PDU's and consequently the processing overhead in layers (N-1) and below.

1.2 Conformance Testing

OSI protocols are presently being implemented by a large number of computer manufacturers and communication companies. In order to achieve the goal of OSI which is interworking of heterogeneous computers, we need to test the conformance of various implementations of protocols to the standards. Standard test suites must be developed for each OSI protocol standard, for use by supplier

or implementors in self testing, by user of OSI products, by telecommunications administrations or by third party testing organizations.

Conformance testing involves testing both the capabilities and behaviour of an implementation, and checking what is observed against both the conformance requirements in the relevant standard(s) and what the implementor states the implementation's capabilities are.

Conformance testing does not include assessment of the performance nor the robustness or reliability of an implementation. It cannot give judgement on the physical realization of the abstract service primitives, how a system is implemented, how it provides any requested service, nor the environment of the protocol implementation.The purpose of conformance testing is to increase the probability that different implementations are able to interwork and gives confidence that an implementation has the required capabilities and its behaviour conforms consistently in representative instances of communications.

Conformance testing can be done using a number of architectures defined by ISO [5]. A number of tests designed to establish conformance of the implementation under test (IUT) is called a test suite.

1.2.1 Types of Testing

ISO defines ten types of testing:

1. **static conformance review :** A review of the extent to which the static conformance requirements (requirements for the capabilities of an implemen-

tation) are met by Implementation Under Test (IUT).

2. **basic interconnection tests :** Limited tests of an IUT to determine whether or not there is sufficient conformance to the relevant protocol(s) for interconnection to be possible, without thorough testing.
3. **capability tests :** Tests to determine the capabilities of an IUT. This involves checking all mandatory capabilities and those optional ones that are stated in the Protocol Implementation Conformance Statement (PICS).
4. **behaviour tests :** Tests to determine the extent to which dynamic conformance requirements (permitted observable behaviour in instances of communication) are met by the IUT.
5. **conformance resolution tests :** Tests to determine in depth whether or not an implementation conforms to a limited number of specific requirements.
6. **conformance testing :** Testing the extent to which an IUT is a conforming implementation.
7. **conformance assessment process :** The complete process of accomplishing all conformance testing activities necessary to enable the conformance of an implementation or a system to one or more OSI standards to be assessed.
8. **test campaign :** The process of executing the parametrized executable test suite for a particular IUT and producing the conformance log.
9. **multi-layer testing :** Testing the behaviour of a multi-layer IUT as a whole, rather than testing it layer by layer.
10. **embedded testing :** Testing the behaviour of a single layer within a multi-layer IUT without accessing the layer boundaries for that layer within the

IUT.

As it can be seen not all types of testing defined by ISO are mutually exclusive and one can come up with a combination of certain types of testing such as conformance multi-layer testing.

1.2.2 Abstract Test Architecture

Test methods need to refer to an abstract testing methodology, based on OSI reference model. Abstract test methods are described in terms of outputs from the IUT are observed and what inputs to it can be controlled. The starting point for developing abstract test methods is the conceptual testing architecture, illustrated in figure 4.

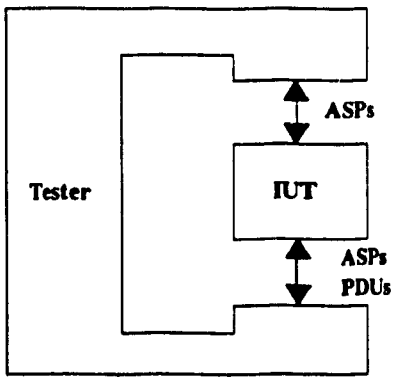


Figure 4 Conceptual testing architecture.

The action of the tester shown in figure 4 can be applied locally, in which case there is a direct coupling within the system under test, or externally via a link or network. The definition of abstract test methods requires that the Points of Control

and Observations (PCOs) be distributed over two abstract testing functions, the lower and upper tester.

The **lower tester** provides control and observations at the appropriate PCO either below the IUT or remote from IUT. If the action of the tester is external to System Under Test (SUT), the lower tester will rely on the (N-1)-service. The **upper tester** provides control and observation at the upper service boundary during test execution.

Four categories of test architectures are defined, one **local**, and three **external** which assume the lower tester is located remotely from the SUT. The local test architecture sometimes referred as generic test is the simplest to specify and defines the PCOs as being above and below the IUT. Local test architectures are not practical since the lower and upper tester must be implemented on every system under test separately.

Three different categories of external test architectures are **distributed, coordinated** and **remote**. They vary according to the level of requirement or standardization put on the test coordination procedures, on the access to the layer boundary above IUT, and the requirements on upper tester.

The **coordinated** test architecture requires that the **test coordination procedures** used to coordinate the realization of the upper and lower testers be achieved by means of **test management protocols**. The other two external test architectures do not make any assumptions about the test coordination procedures. The **distributed** and **coordinated** architectures require specific functions from the up-

per tester above IUT. The **remote** test architecture does not. Finally, only the **distributed** test architecture uses a PCO at the upper layer boundary of the IUT. Therefore, only distributed test architecture requires access to the upper boundary of the IUT. Figure 5 gives an overview of abstract test architectures.

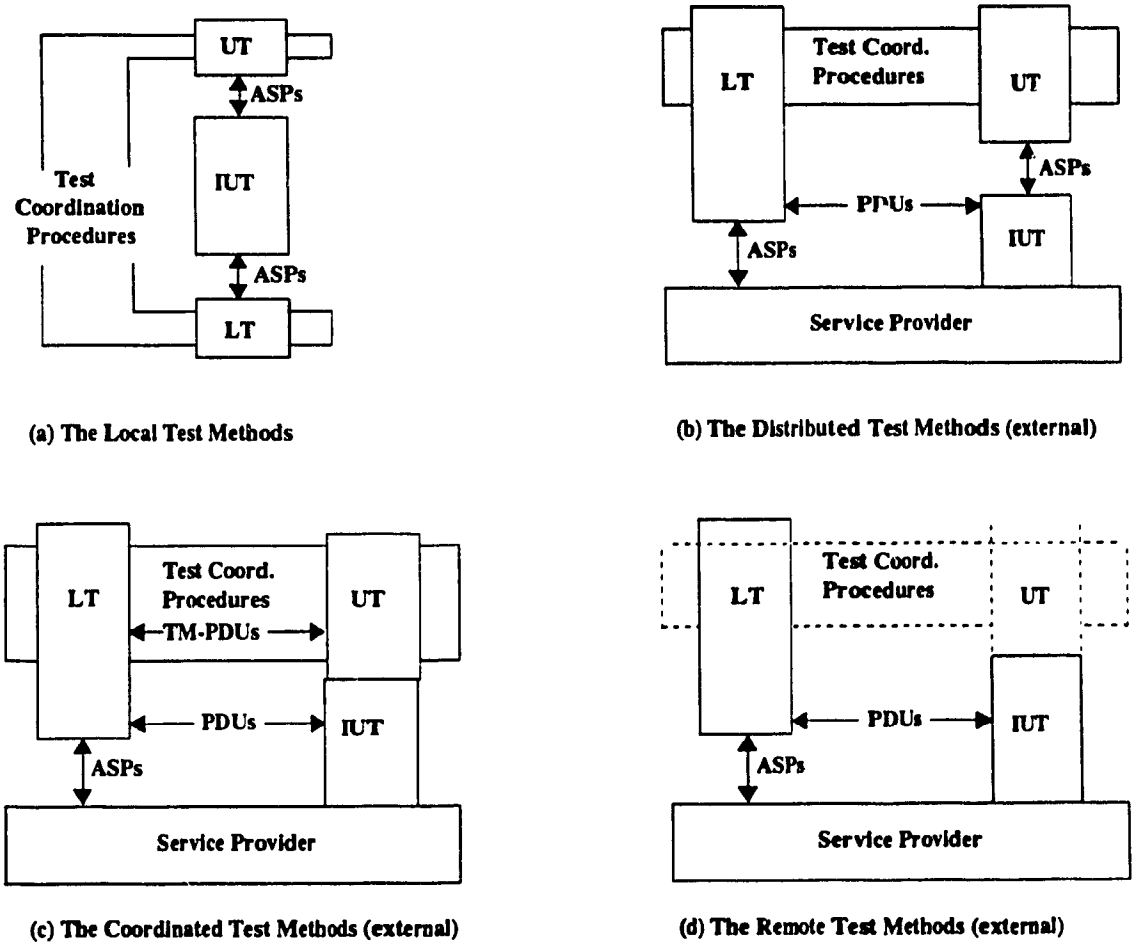


Figure 5 Abstract test methods[5]

1.3 Overview

The objective of this thesis is to implement a system which automates the process of test suite generation from the formal specification of a protocol. Protocols are complex systems and test suite derivation for them is very cumbersome, time consuming and an error prone process. While still some parts must be done manually, this system dramatically reduces the time and effort required to obtain test suites. To the best of our knowledge this work is original and no similar system has been developed earlier.

The system developed uses an earlier test sequence generation system based on the data flow analysis and normalization called CONTEST-ESTL. The contribution of this thesis is generation of tests in TTCN test notation. Estelle language was chosen as the basis of the work for the following two reasons. First there was already a tool developed for generating test sequences and performing data flow analysis for the Estelle language. Second in the standardization documents most of the protocols are specified as finite state machines and can easily be formally specified in Estelle. Protocol standards developed by CCITT like ISDN provide an SDL diagram which can directly be translated to Estelle.

The different modules of this test generation system were implemented in C language and LEX and YACC compiler facilities. The reason why C language was chosen is its portability, ease of maintenance and my own experience with this language.

Next the organization of the remaining chapters of this thesis is explained.

Chapter two gives an overview of formal description techniques and Integrated Services Digital Network (ISDN) and its Link Access Protocol on the D-channel (LAPD). At the end of the chapter the module decomposition of the Estelle specification of the LAPD protocol is explained.

Chapter three gives an overview of the methodology used to derive dynamic behaviour tests from Estelle specifications. This chapter first explains the different steps that must be taken to generate a complete test suite. Also an algorithm to generate transition tours from a collection of interconnected finite state machines is discussed in this chapter.

Chapter four explains the details of generating dynamic behaviour in TTCN format. First algorithms used to generate subtrees and test cases are given. Then the issue of the parametrization of the tests is discussed.

Chapter five explains the declaration and constraint generation module. The first section gives details of mapping between Estelle and TTCN declarations. Two declaration tables generated by the software are demonstrated. The next section gives the algorithm used in the constraint generation module.

Chapter six explains the details of the implementation of the software modules which perform the test suite derivation. The languages used in the implementation of the modules, module decomposition, and dependencies between the modules are explained in this chapter.

Chapter seven demonstrates some of the results obtained by applying the test generation tool on LAPD and transport class 2 protocol. Some statistics about the

test suite generated and the time performance of the software are also given.

Finally a user's guide of the system developed is given in appendix A.

Chapter 2 ISDN AND FORMAL SPECIFICATIONS

As a result of advances in technology, digital transmission techniques are being intensively introduced in a large number of countries. The result of this has been a general consensus in the world of telecommunications to lay down through the International Telegraph and Telephone Consultative Committee (CCITT) the basic elements of the universal network, the Integrated Services Digital Network (ISDN)[8].

The main feature of the ISDN concept is the support of a wide range of voice and non-voice applications in the same network. A key element of the service integration for an ISDN is the provision of a range of services using a limited set of connection types and multi-purpose user network interface arrangements.

The definition of ISDN is based on three fundamental elements:

1. **Digital connectivity for information transfer :** all types of signals are transmitted in digital form across the network.
2. **Common-channel signaling connectivity :** In the ISDN, signaling is transmitted over the entire network and between the terminals in the form of messages, containing addresses, information and protocol elements.
3. **Multipurpose capability of user-network interfaces :** the connection to the ISDN allows the user to have at his disposal such different services as voice, telematic or video communications from the same point.

CCITT has defined a common-channel signaling system known as protocol D. According to the principle of common channel signaling, a particular channel (Channel D) transports the messages between the user and the network completely independent of the information channels. Any signaling interchange can therefore be carried out outside or during a communication. In addition to common channel signalling telemetry and packet switched network data can be carried over this channel.

Figure 6 shows the architecture model and reference points of the ISDN. The definition of the functional groups are given below.

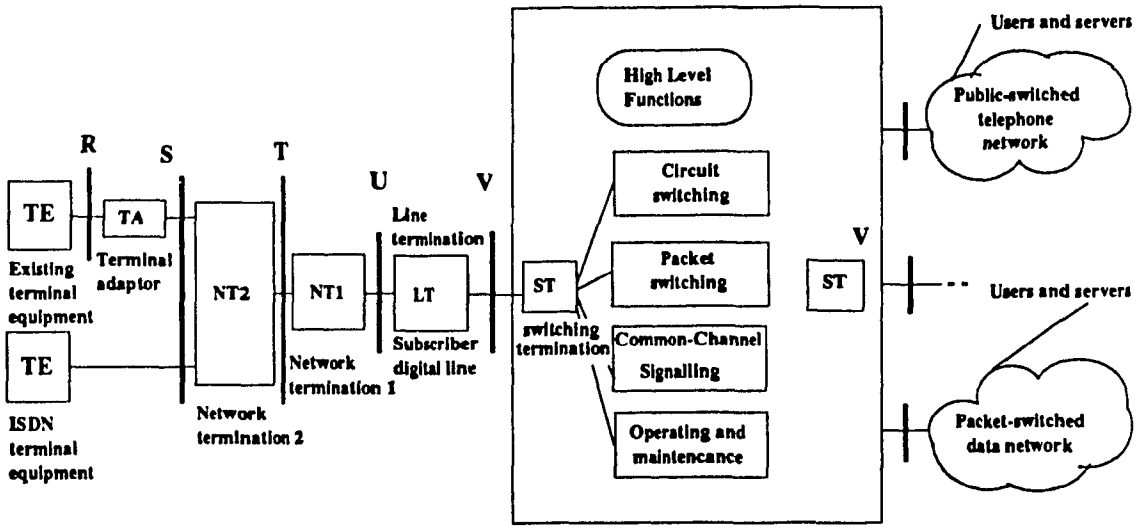


Figure 6 Functional groups and reference points of ISDN[7].

- **Terminal equipment** (TE) performs the functions of layers 1, 2, and 3 of the user side of ISDN user network interface.

- **Terminal adaptor (TA)** performs the adaptation of non ISDN terminals (TE2) allowing a terminal type 2 to be served by the ISDN user network interface.
- **Network termination 2 (NT2)** performs the functions of layers 1, 2 and 3 of the network-user interface. Private exchanges or intercommunication systems are examples of equipment which perform the NT2 functions.
- **Network termination 1 (NT1)** handles layer 1 on the network side of the interface T.

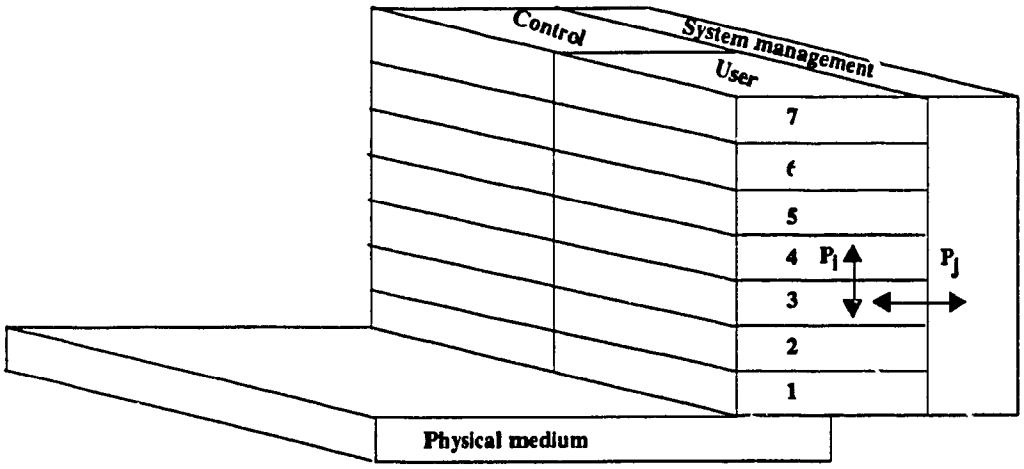
The user interface with the ISDN network is at points S and T. Rates and types of transmission are defined at these points. Two access rates have been defined at S/T interface. The first is called "basic rate" and has a usable rate of 144 kbits/sec. The second access rate is the primary access rate whose usable rate is 1984 kbits/sec (in Europe) or 1536 kbits/sec (in North America).

ISDN signalling capacity is multiplexed into two types of channels. A **B** channel at the rate of 64 kbit/sec. A **D** channel is used for common-channel signaling, telemetry and packet switched data. The structure of basic rate is 2B+D, the rate of D channel being 16 kbits/sec. For the primary rate the structure is 30B+D or 23B+D, for a total of 1984 or 1536 kbit/sec, this time the rate of D channel is 64 kbit/sec. For very small subscriber installations, one or more terminals should be capable of being connected by a single access directly to the public network, i.e. without NT2.

Three other types of channels are also defined for primary access, channel

H0 (384 kbit/sec), H11(1536 kbit/sec) and H12 (1920 kbit/sec) for services such as videoconferencing.

The definition of ISDN access is based on a functional separation of the signaling data flows from the user information flows exchanged between the users. Separation of signaling and information flow gives the three-dimensional model of ISDN protocol shown in figure 7.



P_I - service primitive between adjacent layers.
P_J - management service primitive between the control or user layer and the management entity.

Figure 7 Interactions between entities at the user-network interface[7].

The model consists of three planes:

1. **The control plane**, which is organized in seven layers, concerns signaling in the D channel and covers all the control protocols for service calls and facilities. So far only the service primitives of layers 1–3 have been defined for the protocols of the control plane.
2. **The user plane**, which is also organized in seven layers, contains protocols implemented for exchanging data relating to applications in the channels for user information transfer (D, B or H)
3. **The management plane**, which is not organized in layers, concerns the local operating functions of the NT2s and the terminals.

The control and user planes can communicate with the management entity by using management service primitives. Management entity coordinates the activities in control and user planes which do not communicate directly. Figure 8 gives an example of an application which transfers data on a circuit switched connection. At the beginning a circuit switched connection is set up by the control plane or the D channel. After the connection is set up the user data is transferred in the user plane in the B channel[7].

2.1 LAPD

The function of Link Access Protocol on the D-channel (LAPD) is to convey information between layer 3 entities of TE, NT2 and NT1 on the D-channel. LAPD is independent of transmission bit rate. It requires a duplex, bit transparent D-channel.

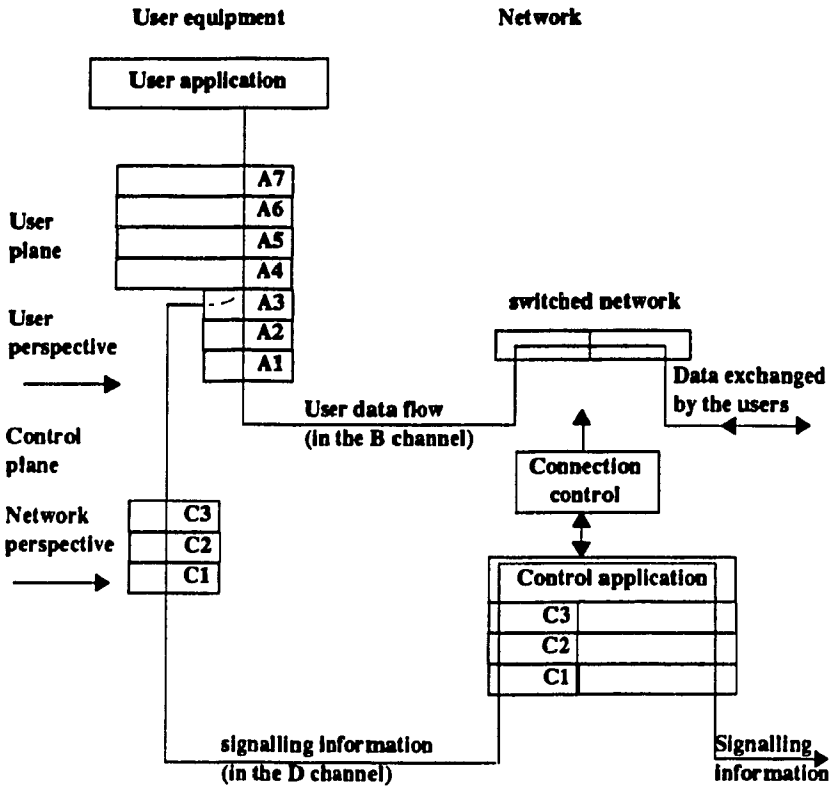


Figure 8 Control of circuit-switching connection[7].

The functions of LAPD are:

1. The provision of one or more data link connections on a D-channel. Discrimination between the data link connections is by means of a data link connection identifier (DLCI) contained in each frame;
2. Frame delimiting, alignment and transparency, allowing recognition of bits transmitted over a D-channel as a frame;
3. Sequence control, to maintain the sequential order of frames across a data link connection;
4. Detection of transmission, format and operational errors on a data link;

5. Recovery from detected transmission, format, and operational errors. Notification to the management entity of unrecoverable errors; and
6. Flow control.

2.2 Establishment of Information Identification

A data link connection is identified by a Data Link Connection Identifier (DLCI) carried in the address field of each frame. DLCI consists of two elements: the Service Access Point Identifier (SAPI) and the Terminal Endpoint Identifier (TEI). The SAPI is used to identify the service access point of the user-network interface. The TEI is used to identify a specific connection endpoint within a service access point.

Two different types of data link connections have been defined for LAPD. The first type is broadcast link which is always in information transfer mode and provides unacknowledged information transfer. The second type is point-to-point data link connection which provides both unacknowledged and acknowledged information transfer and corresponds to a particular SAPI. In order to transfer information in a point-to-point data link a connection must explicitly be established by exchanging SABME and UA command. Figure 9 shows the relationship between SAPI, TEI and DLCI. Each SAP identified by SAPI has a specific purpose. Recommendations Q.920 and Q.921 of CCITT define 4 SAPs : 0 for call control procedures, 1 for packet mode communications using Q.931 call control procedures, 16 for packet communication conforming to X.25 level 3 procedures and finally 63 for management procedures[8].

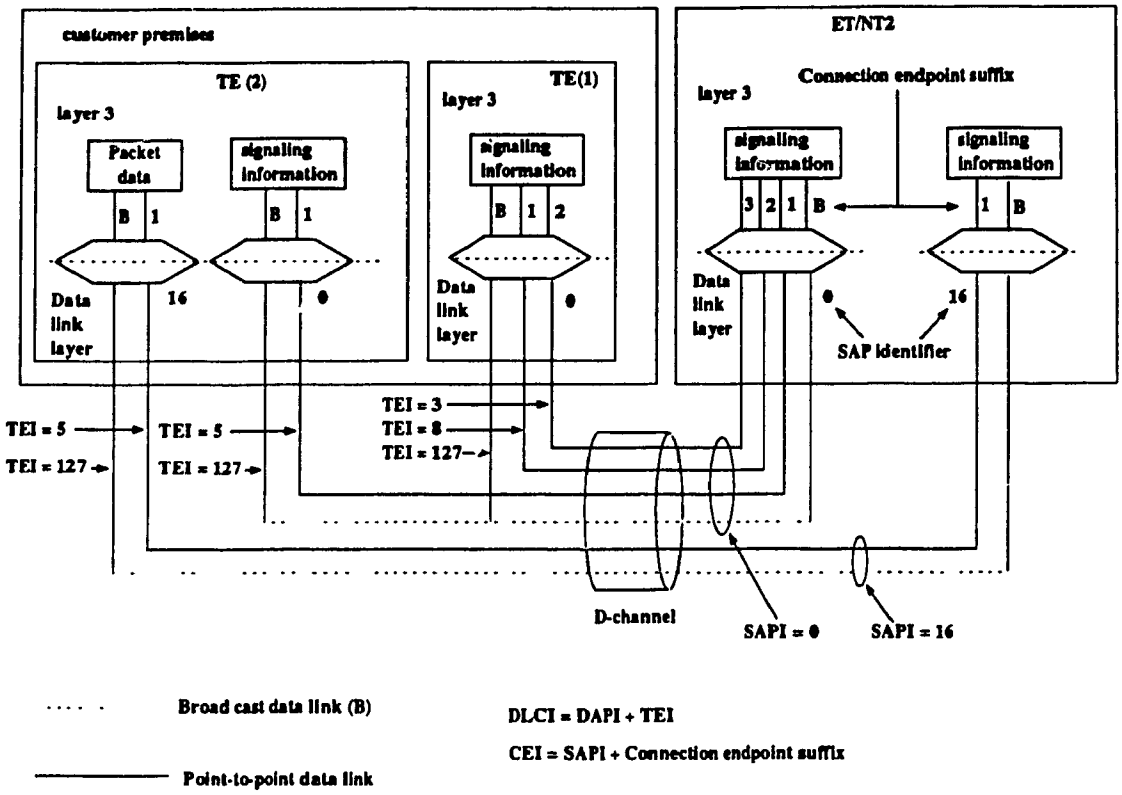


Figure 9 Relationship between SAPI, TEI and DLCI[8].

2.3 Estelle

Standard definitions are presently given in natural languages. Since these definitions contain ambiguities and impreciseness, standardization institutions such as ISO(International Organization for Standardization) and CCITT(International Telephone & Telegraph Consultative Committee) have developed formal description techniques(FDTs). The formal description technique accepted by our test generation system is Estelle[4], which is based on an extended finite state machine model.

The extended finite-state machine (EFSM) model describes a system such as a protocol entity as a collection of modules. Each module is a finite-state machine capable of having memory, i.e., extended finite-state machine. Modules of an entity can communicate with each other as well as with the environment over channels (FIFO queues). Messages carrying data paramters are exchanged in these channels. The queue management is done by Estelle language and the user is relieved from this task. Instances of these channels are called interaction points. Service primitives (exchanged with lower and upper layer entities), and internal interactions (with other modules) are communicated in the channels. PDU's exchanged between two protocol entities need not explicitly be defined, they are encoded and introduced as parameters to the service primitives. Decomposition of an entity into modules is usually functional: a module for timer management; a mapping module to map the PDUs into interactions with the environment, i.e. service primitives; an abstract protocol module for handling service primitives and forming PDUs, etc.

The language of Estelle is based on Pascal with extensions to facilitate protocol specification. To save space we only describe the constructs related to transitions. An Estelle transition is composed of an initial state, a final state. **FROM/ TO** clauses define initial/ final state(s) of the finite-state machine, respectively.

The arrival of an input interaction is expressed using **WHEN**. Transitions with no WHEN clause are called spontaneous. They are used to describe

nondeterminism (internal decisions of the entity, for example).

The conditions for firing the transition are described in a **PROVIDED** clause which is a Boolean expression on interaction primitive parameters and variables of the module. Variables of the module are called context variables. The presence of PROVIDED clause is optional in a transition and its absence is equivalent to PROVIDED FALSE.

Estelle transitions that do not have a WHEN clause may have a **DELAY** clause. The general format of DELAY clause is:

delay(min, max)

A transition with delay clause is not considered before min time units. It may be selected between the min and max time units. After max time units if the provided clause is satisfied it must be selected.

Another clause related to transitions is the **ANY** clause. The simple form of any clause is:

any **IDENTIFIER : DOMAIN**

If an ANY clause occurs in an expanded transition (normalized transitions are all expanded) then the corresponding transition declaration is only a shorthand notation for a sequence of transitions where the IDENTIFIER is replaced by it's value. For example:

```
any X : 1..2

  provided f(x)
```

is equivalent to two transitions. The provided clause of the first transition is:

```
provided f(1)
```

and the provided clause of the second transition is:

```
provided f(2).
```

Finally, the action of the transition is contained in a **BEGIN/END** block which can have assignments to context variables, calls to internal procedures, Pascal statements and produce output with **OUTPUT** statement[4].

Estelle supports nondeterminism by way of spontaneous transitions and allowing more than one transition from a given major state to have their predicates enabled. Once a transition is enabled, its execution is atomic. Abstractness of the specification, i.e. being away from implementation considerations could be achieved through the use of incomplete type definitions using the three-dot notation, such as in the type declaration:

```
buffer_type = ...;
```

or constant declaration:

```
max_buffer = any integer;
```

2.4 TTCN

A language has been defined by ISO to specify abstract test suites, i.e., expressing tests in a manner independent of their execution. This language is called the Tree and Tabular Combined Notation (TTCN).

The Tree and Tabular Combined Notation (TTCN) specifies a test suite in four parts: test suite overview, declarations, dynamic behavior and constraints.

The test suite overview table is for describing the purposes of the test suite and its individual tests. The points of control and observation (PCOs), abstract service primitives (ASPs) and Protocol Data Units (PDUs), global variables and timers are defined in declaration tables. The dynamic behavior table is used to define test cases as trees of events input(?)/ output(!) from the service access points (SAP) accessible to the upper/ lower testers. Dynamic behavior table contains behavior description, label, constraints reference, verdict and comments columns [1]. The constraints part specifies values for ASPs and PDUs that are symbolically referenced in the dynamic behavior tables.

A conformance test suite in TTCN consists of a number of test cases which test the implementations for conformance. Tests are hierarchically organized into test groups each consisting of one or more test cases. Test cases are specified using the tree notation and are made up of test steps.

For example, suppose the following events can occur during a test whose purpose is to establish a connection, exchange some data and then disconnect. The tree notation expresses this sequence of events as:

```
TREE[L]
  L! CONNECTrequest
    L ? CONNECTconfirm
      L ! DATAreq
        L ? DATAind
          L ! DISCONNECTreq
          L ? DISCONNECTind
        L ? DISCONNECTind
    L ? DISCONNECTind
```

Here, L stands for the PCO at which the lower tester exercises the test. The symbol ? and ! stand for receive and send respectively. So, L! CONNECTrequest means that the tester transmits the CONNECTrequest primitive at the PCO L at this point in the test. TREE[L] is the identifier for this behavior tree and L stands for the formal PCO used.

2.5 Estelle description of LAP-D

As part of the work for this thesis, a formal description of LAPD based on recommendations Q.920 and Q.921 was developed. Figure 10 shows the block diagram of modules of this description. Since the purpose of this formal description was automatic test suite generation, for simplicity only the modules labeled "DATA LINK ENTITY" and "FRAME EXCHANGE" (referred as "data link procedure" in recommendations) as well as timer T200 and T203 modules were explicitly described in Estelle.

Each Data link entity module is able to set up a link by exchanging SABME and UA frames. The link is released by exchanging DISC and UA frames. While the link is established acknowledged and unacknowledged frames can be exchanged.

Demultiplexing of frame is done based on the SAPI value of incoming frames. For the outgoing frames multiplexer assigns the value of SAPI to the frames based on the data link entity requesting transmission of the outgoing frame. The reason for making multiplexer responsible for assignment of SAPI value was that the data link entities were described as an array of identical Estelle modules.

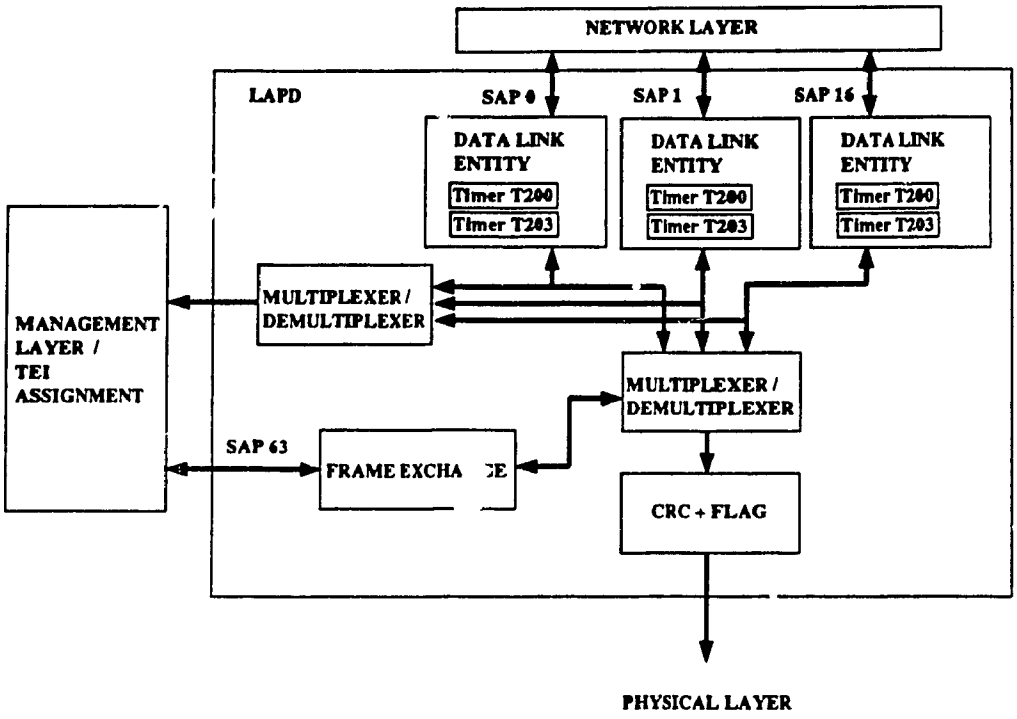


Figure 10 Estelle modules for formal description of LAPD protocol.

The reason for having two types of connections to management layer is that each individual data link must be able to communicate with the management layer in order to report errors and receive response to errors and the value of TEI. This is done by having a multiplexer/demultiplexer at the interface between LAPD and management layer. At the same time management is able to exchange information and identification frames (XID frame) which must be done by LAPD as a whole and not individual data links bound to a network layer SAP.

Another point worth mentioning is that the data link entities are not really independent as claimed by Q.921 and Q.931 recommendations, because TEI assignment and removal is common to all data links. All the data links must

change their states in the case of assignment and removal of TEI simultaneously. In order to specify the data link entities as independent finite state machines with the states specified in recommendations all service primitives leading to assignment and removal of TEI must be broadcast to all the data link entities. These service primitives are MDL-ASSIGN-REQUEST for assignment of TEI and MDL-REMOVE-REQUEST and MDL-ERROR-RESPONSE for removal of TEI which are broadcast in our approach.

Chapter 3 TEST SUITE DERIVATION

Formal description techniques such as Estelle can be used in (semi)- automatic generation of test suites. Finite-state machine based techniques [9, 10, 11] lead to state explosion therefore they are not considered in this thesis. Some adhoc methods developed to cope with context variables and primitive parameters are discussed in [12,13].

The methodology of interest to us in this thesis is the one that takes an Estelle specification of the protocol and derives tests from the control and data flow graph models of the normalized specification [14]. For specifications containing multiple modules the normalization is applied to each module. A tool (CONTEST-ESTL) implementing this methodology automates most of the steps [15, 16].

To derive test sequences the following steps must be taken: The formal specification of protocol in Estelle is given as input to CONTEST-ESTL for normalization followed by simplification of provided clauses. At this point three tasks are done in parallel. These are data flow analysis, test sequence generation and TTCN test step generation.

The resulting test sequences are called unparameterized test sequences since no parameter values are assigned for test inputs. Then the result of data flow analysis and test sequences are combined together and test cases in TTCN format are generated. By obtaining test cases and test steps most of the work for test suite generation is achieved and the user only needs to provide the suite overview.

The methodology is schematically shown in Figure 11. The contributions of this thesis to the test generation tool are simplification, test step generation, test sequence generation and finally test case generation modules. The box labeled fully automated means that the modules do not require any user intervention or interaction during their execution.

In this chapter normalization, simplification of provided clause, test sequence generation and data flow analysis are explained. Chapter 4 explains the details of generating dynamic behaviour in TTCN format. Chapter 5 explains the constraint generation module.

3.1 Normalization

Normalization is the first step of test suite derivation methodology. After the specification is written in Estelle the normalization module is activated. The process of normalization transforms the input specification into another Estelle specification which possibly contains more transitions each having a single path. The functions of normalization module are[17]:

i. Variant records are converted into records enumerating all case constants;
ii. Body of procedures and functions are replaced;
iii. Conditional statements(IF, CASE) are eliminated;
iv. State sets are enumerated so that each transition has one major state in the TO and FROM statements;
v. WITH statements are removed by record structure replacement.

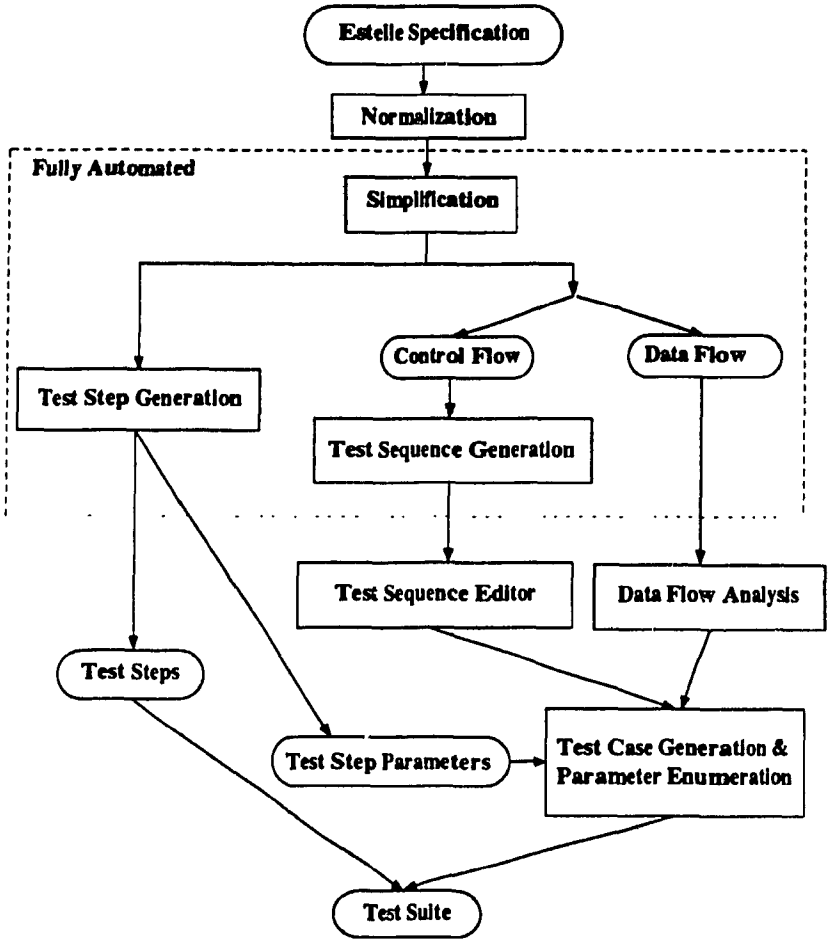


Figure 11 Test suite derivation methodology.

The following is a transition from the Estelle specification of ISDN LAPD protocol (ph is the physical layer and l3 is the network layer discussed in chapter 2):

```
from AwaitngEstab
 to MulFrmEstabd
  when Ph.PH_DtInd
   provided (FrmRecvd.Frame = UA) and (FrmRecvd.F = FINAL)
    begin
```

```
if L3Initd then
  output L3.DL_EstabConfmtn
else
 if V_S <> V_A then
   begin
     DiscIQ;
     output L3.DL_EstabInd;
   end;
output TIMER_T200.STOP;
T200_RUNNING := false;
output TIMER_T203.START;
V_A:=0;
V_S:=0;
V_R:=0;
end;
```

This transition occurs when a LAPD entity receives an Unnumbered Acknowledgement (UA) in response to a SABME frame.

Normalization transforms the above transition into 3 transitions. Normalization is achieved by moving the conditions in the IF statements to cover one of the paths to the PROVIDED clause. Each transition corresponds to a path of the IF statements. The following is one of those transitions which has a single path:

```
trans
      { 94 }
      when ph.ph_dtind_ua
      provided (((true) and (frmrecvd.f = final)) and (not (v_s <>v_a))) and
(not (l3initd))
      from awaitngestab
      to mulfrmestabd
      begin
         output timer_t200.stop;
         t200_running := false;
```

```
    output timer_t203.start;
    v_a := 0;
    v_s := 0;
    v_r := 0
end;
```

After normalization, we obtain 420 normalized transitions for the LAP-D specification.

3.2 Simplification of the Provided Clause

After normalization the PROVIDED clauses are simplified and decomposed into several simpler atomic predicates for which there is a unique way of satisfying. The method is the following: Let P1, P2 and P3 be elementary expressions such as 'frmrecvd.p_f = final'. An elementary expression is either a Boolean variable or a relational expression on variables and input primitive parameters. We perform the following simplifications in the order specified:

1. 'TRUE and P1' is replaced by 'P1';
2. 'TRUE or P1' is replaced by 'true';
3. 'FALSE and P1' is replaced by 'false';
4. 'FALSE or P1' is replaced by 'P1';
5. 'NOT' and relations are combined. For example 'NOT(X <= Y)' is converted to 'X > Y';
6. 'NOT(P1 and P2)' is replaced by 'NOT(P1) or NOT(P2)'.
   Similarly, 'NOT(P1 or P2)' is replaced by 'NOT(P1) and NOT(P2)'.

'NOT(NOT(P1))' is also replaced by P1. Then this rule is applied recursively to 'NOT(P1) ' and 'NOT(P2) ';

7. if the predicates 'P1' and 'P2' are mutually exclusive, 'P1 or P2' is replaced by:

   '(P1 AND NOT(P2)) OR (NOT(P1) AND P2) OR (P1 AND P2)'.

   In general 'P1 OR P2 .. Pn' mutually exclusive ORed predicates produce $2^n - 1$ ORed predicates;

8. steps 5 and 6 are repeated for the resulting terms which have the NOT(Pn) in step 7;
9. AND is distributed inside OR operation. 'P1 AND (P2 OR P3)' is replaced by '(P1 AND P2) OR (P1 AND P3)'. The rule is repeated of the terms 'P1 AND P2' and 'P1 AND P3' if either P1, P2 or P3 are not simple expressions;
10. Relations are combined. For example:

    '(EXPR1 = EXPR2) AND (EXPR1> = EXPR2) is replaced by:

    '(EXPR1 = EXPR2)';

11. Duplicate predicates are removed;
12. Predicates evaluating to a false value are replaced by FALSE. For example: '(EXPR1 < EXPR2) and (EXPR1 = EXPR2)' is replaced by 'FALSE'. Then rule 2 and 3 are reapplied to the resulting expression.

Rules 1 thru 12 above follow from straightforward logical equivalences. Rule 7 guarantees that every product (in the form of P1 AND P2 AND ...) contains all the elementary expressions exactly once. Transformation in rule 9 converts a

given predicate to the form (known as disjunctive normal form):

P1 OR P2 OR P3 ... OR Pn

where Pi's are products of elementary expressions, i.e. of the form P1 and P2 and ... Pn [11]. Because of the rule 7 each Pi contains all the elementary expressions exactly once, therefore Pi's are mutually exclusive. The next step in our simplification process is to create a separate transition for each elementary expression, i.e. each Pi becoming the PROVIDED clause of a separate transition. This is accomplished by separating Pi from the main predicate and substituting it into a copy of the original body and header, i.e. completing the other parts (WHEN, if any and the BEGIN-block) of the transitions. The advantage of this step is that now for each transition there is only one way to satisfy the PROVIDED clause. Also all the possibilities to fire a transition have been enumerated.

As an example we take the transition 259 from the LAP-D specification:

```
trans
        { 259 }
        when ph.ph_dtind_rej
        provided ((true) and not ((v_a <= frmrecvd.n_r) and (frmrecvd.n_r <=
v_s))) and (not ((frmrecvd.c_r = n2ucomand) and (frmrecvd.p_f = poll)))
        from timerrecovry
        to awaitngestab
        begin
            update_buffer(frmrecvd.n_r);
            peerrecbz := false;
            output lm.mdl_errind(j);
            peerrecbz := false;
            rejexcept := false;
            ownrecverbz := false;
            ack_pending := false;
```

```
    rc := 0;
    frmvar_sabme.tei := tei;
    frmvar_sabme.c_r := u2ncomand;
    frmvar_sabme.p := poll;
    output ph.ph_dtreq_sabme(frmvar_sabme);
    l3initd := false;
    output timer_t200.start;
    t200_running := true;
    output timer_t203.stop
  end;
```

Transition 259 gets expanded into six transitions. The following is one of those transitions with a simplified provided clause:

```
trans
    { 298 }
    when ph.ph_dtind_rej
    provided (frmrecvd.p_f = final) and (frmrecvd.c_r <> n2ucomand)
and (frmrecvd.n_r > v_s) and (v_a <= frmrecvd.n_r)
    from timerrecovry
    to awaitngestab
    begin
      update_buffer(frmrecvd.n_r);

      (* same as above *)

    end;
```

Provided clause simplification step increases the number of transitions of normalized LAPD specification from 420 to 490 transitions.

3.3 Data Flow Analysis

After simplification of the provided clause, data flow analysis of the protocol can be performed. This step generates two files. The first file contains the data flow information. A graphical representation of the actions of normalized transitions, called data flow graph (DFG) is done by a program called Dfgtool [16]. The second file contains the control flow information of the protocol which is the initial and final state of transitions as well as the input service primitive and output service primitives inside the body of transitions.

A data flow graph can be partitioned into blocks, a block representing the flow over a single context variable. These blocks can be merged together in order to obtain protocol functions or test purposes[19]. The test cases which have the same test purpose specified for a block are grouped together. Therefore the blocks inside the data flow graph make up the test groups in TTCN. Dfgtool lets the user name these test purposes to be used later by the test case generation program. Figure 12 shows a part of the data flow graph for the function "Transmission of Acknowledged Frames". Using data flow analysis technique 19 test purposes were identified for LAPD protocol. These functions are shown in table 1.

Transmission of DISC Frames	In this test group IUT is forced to transmit DISC Frames.
Transmission of DM Frames	In this test group IUT is forced to transmit DM Frames.

Table 1 Test purposes for LAPD protocol. (Continued . . .)

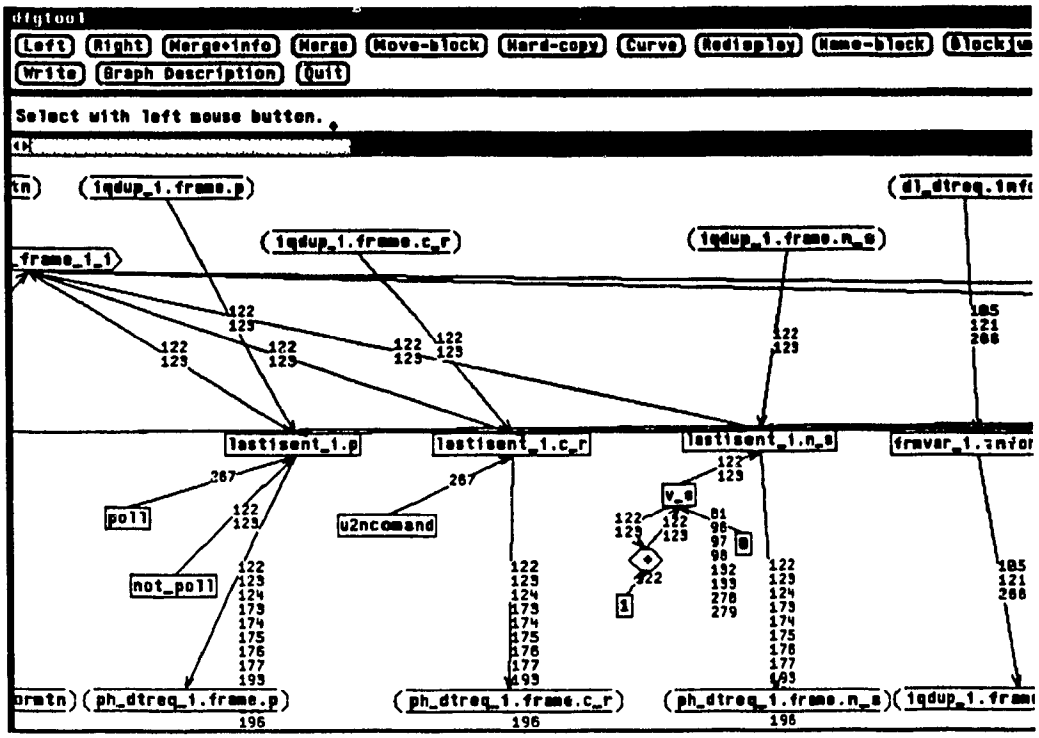


Figure 12 Example of a data flow graph.

Transmission of FRMR Frames	In this test group IUT is forced to transmit FRMR Frames.
Transmission of Acknowledged Information	In this test group IUT must transmit Acknowledged Information.
Transmission of REJECT Frames	In this test group IUT is forced to transmit REJECT Frames.
N(R) Value of Outgoing Frames	In this test group the correct setting, incrementing and resetting of N(R) sequence number of frames is verified.

Table 1 Test purposes for LAPD protocol. (Continued . . .)

Transmission of RNR Frames	In this test group IUT is forced to transmit RNR Frames.
Transmission of RR Frames	In this test group IUT is forced to transmit RR Frames.
Transmission of SABME Frames	In this test group IUT is forced to transmit SABME Frames.
Transmission of UA Frames	In this test group IUT is forced to transmit UA Frames.
Transmission of Unacknowledged Information	In this test group IUT is forced to transmit Unacknowledged Information.
Persistant Deactivation of Physical Layer	In this test group physical layer is deactivated and the TEI removal of the equipment is verified.
Retransmission counter	In this test group the correct setting, incrementing and resetting and the effects of values of retransmission counter are verified.
Reject Exception Condition	In this test group the correct setting and clearing of reject exception condition and its effects areverified.

Table 1 Test purposes for LAPD protocol. (Continued . . .)

Saving Release Request During Link Establishment	In this test group the link is requested to be released before it is established and it is verified that this request is saved and the link is released as soon as it is established.
N(R) Value of Incoming Frames	In this test case the effect of N(R) sequence number of frames sent by tester is verified.
Reception of Acknowledged Information	In this test group Acknowledged Informations are transmitted to IUT and the correct behaviour of IUT is verified
Reception of Unacknowledged Information	In this test group Unacknowledged Informations are transmitted to IUT and the correct behaviour of IUT is verified
Report of Errors to Management Layer	In this test group errors that must be reported to management entity are created.

Table 1 Test purposes for LAPD protocol.

3.4 Test Sequence Generation

This step uses the control flow information about different modules of protocol created in the previous step. There are several techniques to generate test sequences for finite state machines which guarantee the correctness of implementations which pass these tests, i.e. they have 100% error detection capabilities [20]. All these techniques use state identification sequences of finite state machines to verify that each time a transition is fired the FSM is in correct state.

A state identification sequence is a sequence of inputs for which the output sequence is different for different initial state of FSM. The main state identification sequences are:

1. Characterization or W sequences [22].
2. Distinguishing sequences [21].
3. Unique Input Output (UIO) sequences [10].

There are two types of distinguishing sequences: preset and adaptive. Preset state identification sequences are fixed for all states, adaptive sequences change depending on the initial state of FSM and therefore if the state of FSM is not known and if it is desired to find out what state FSM is in, the input sequence depends on the observed output sequence hence the name adaptive is given. For practical purposes the preset sequences are preferred over the adaptive sequences.

Unfortunately there are many problems with the technique of applying state identification sequences to protocols modeled as finite state machines which makes them impractical. These problems are :

1. Not all FSMs have distinguishing or characterization sequences. This problem is not specific to protocols modeled as FSMs and also exists for ordinary FSMs;
2. Protocols are extended finite state machines and have variables. The effect of the value of these variables might make applying identification sequences impossible;

3. The most important reason is that almost all FSM models of protocols are nondeterministic, i.e. for the same input and in the same state different transitions with different outputs have been defined. The proof of 100% error detection capabilities for all the three state identification sequences assumes that the FSMs are deterministic.

The nondeterminism of protocols modeled as FSMs is usually because they are extended finite state machines and the different behaviours of the protocol implementations are due to the variation of parameters of input primitives, e.g. sequence number of a packet or the variables inside the protocol or both. Combining the data with these sequences and extending the states of FSM due to effect of variables greatly solve these problems. Implementation of a system which derives test sequences by considering the data and extending the above mentioned test generation methods to consider data has not been implemented in our system.

Our test generation program generates a simple transition tour. The transition tour is divided to subtours. A subtour is a sequence of transitions that starts from the initial state of the protocol and goes back to the same state. This division is marked physically in the output file and this way the user is able to define subtours that go back to initial state more than once by editing the output file. Each subtour becomes a test case in TTCN. Conformance testing requires that test cases be independent and can be run in any arbitrary order. That is the reason why subtours must start from a stable state of the protocol and go back to (possibly different) stable state.

3.5 Implementation of Transition Tour Generator

The test sequence generation program called mstourgen (multiple module synchronizable tour generator) generates transition tours for a collection of finite state machines. These finite state machines are interconnected together by FIFO queues and messages put by a module are consumed by another module which has been coupled through a queue at that interaction point.

Besides the control flow information of each EFSM the program takes two more kinds of optional input files. The first kind of optional input file is specified by an option in the command line and describes the internal queues or interaction points of the protocol. The user must specify which interaction points are connected together.

If an interaction point is internal then a queue is associated with it and before a transition which takes its input from an internal interaction is fired this input must be present in the head of the queue. To avoid possible deadlocks, before a transition outputs an interaction into a queue the module which receives this interaction is checked to be able to consume that event.

The queues are managed in two ways. The preference is given to rendez-vous type of passing messages. This means that as soon as an event is put into the queue the other module consumes the event by firing a transition which consumes and dequeues that event. However if this is not possible for a transition, the event is put into the queue and can be consumed after.

The second type of the user in the command line enables the user to let a

tester monitor more than one external interaction point. For example the upper tester used for LAPD monitors the point-to-point and broadcast links as well as the interface of LAPD and management entity.

The user has the option of two types of transition tours. The first option minimizes the overall length of tour but subtours can be of any length. The second option minimizes the length of subtours but the length of overall tour is most probably much longer than using the first option.

The tests generated are synchronizable. This means that to each external interaction point a separate tester can be connected. At each instance one of these testers is active and applies inputs to IUT. Before switching from one active tester to another, the second tester will receive an interaction (input). This is the test coordination procedure needed to implement the test using distributed test architecture.

3.6 MultiModule Tour Generation Algorithm

We present an original algorithm to generate a transition tour from several FSMs connected by FIFO queues. Because of the huge number of possibilities in the search through multiple modules, efficiency is an important factor and many of the steps explained aim to maximize the efficiency.

The input was explained in the previous section. The output of this program is a transition tour which covers all the transitions of all FSMs.

Step-1: Group the transitions which are similar to reduce the search complexity. Two transitions are similar if:

a. Their next states are the same;
b. If the input interaction point or the interaction points of one of the outputs are internal, the other transition must have the same input or output interaction point and must put the same message in the queue;
c. The external interaction points of the input for non spontaneous transitions are the same or are controlled by the same tester;
d. For external interaction points in the output statements which are different from input interaction point and are not controlled by the same tester and therefore change the synchronization, check that the other transition also outputs a message to the same external interaction point or an interaction point which is controlled by the same tester;

Step-2: Do an approximate calculation of the maximum depth of search to cover a new transition. The length is the maximum number of states if no synchronization is requested and no internal queues are present (this is an exact value). If synchronization is required the length is doubled since for every transition that must be taken an extra synchronizable transition might be needed (this is a rough approximation). This value can be supplied by the user during the invocation of the program. The result is the maximum depth of the search. The maximum depth of the search is multiplied by the maximum number of internal interaction points in the

transitions to take into account the effect of rendez-vous which makes the length of sequence longer than the maximum depth of search. The latter number is the maximum length of transition sequence taken to cover a new transition;

Step-3: Do a depth first search of all possible sequences of transitions to find a sequence of synchronizable transitions which contains at least one group of transitions that one or more of them have not been covered and leave the queues empty (to avoid committing to transitions which yield to deadlock). To reduce the complexity of the search only the following group of transitions are considered during the search:

a. Group of transitions that change the state;
b. Group of transitions that have at least one transition that has not been covered.
c. Group of transitions that take their input from an internal queue.
d. Group of transitions that output to internal queues.
e. Group of transitions that change synchronization, i.e. output a message to an external interaction point which is not controlled.

Each group has associated flags to avoid searching inside the group each time. The depth is incremented from 1 up to the maximum number found in step 2.

Step-4: To further reduce the complexity the group of transitions which cause queueing underflow (take a message from internal queues that it is not

available) are avoided during the set up of sequence and all levels of back-tracking.

Step-5: If step 3 fails first turn off the synchronization checking. If it fails again stop tracing internal queues and if it fails again stop checking both synchronization and queues and repeat step 3. If each time the search is successful issue a warning to the user about the decision.

Step-6: If short subtours are requested:

a. Save the current state of search (major states of modules and the sequence of transitions considered so far) because the next search will go beyond the current search.

b. If the protocol is not in its global initial state after applying the sequence found in step 2, find a sequence that takes the protocol back to the initial global state and leaves the queues empty. The search algorithm is similar to the one in step 3 except that this time the goal is to reach the global initial state rather than covering a new transition.

c. Save the sequence of transitions that takes all the modules back to initial state.

d. Repeat a cycle of the sequence found in step 3 and the sequence of transitions found in (b) until all the transitions in all the groups are covered.

Step-7: Repeat steps 3 to 5 until all transitions are covered. In case of short

subtours resume the search path to what it was at the end of step 3 because the next search is longer than what it was in previous search.

Step-8: If the protocol is not in its global initial state find a sequence that takes the protocol back to the initial global state and leaves the queues empty.

3.7 Infeasible Paths and Edittour

Since the test generation program only uses the control flow information and the value and effects of variables are left out, some subtours may be impossible to execute because the enabling conditions of predicates are not satisfied. Therefore the user may want to change the sequence of transitions of a subtour for the previous reason or other reasons such as studying the effect of a certain value of a variable.

A program called Edittour was written to make this modification easier. This program shows the Estelle code of the transitions that will be executed in each subtour and enables the user to change the sequence of transitions in the subtour. An example display by Edittour is shown in figure 13.

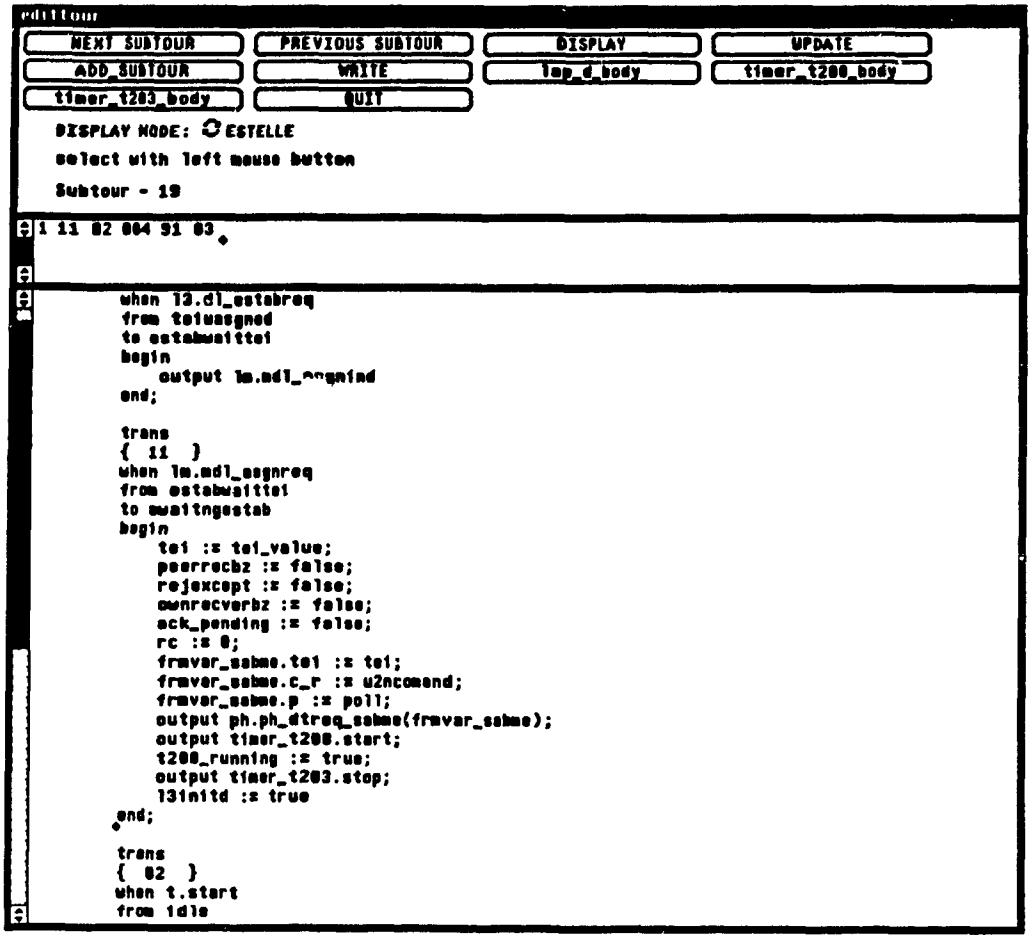


Figure 13 Edittour.

Chapter 4 DYNAMIC BEHAVIOUR GENERATION

After having normalized the specification and simplified the PROVIDED clauses our next step is generation of the dynamic behaviour part of TTCN. This task is divided into two parts. The most important part of dynamic behaviour generation is the generation of test steps. The second part is generation of the test cases. The program which generates test steps and the declaration part of TTCN is called TTCNgen.

The information needed for declaration and test step generation is:

1. Normalized and simplified Estelle specification.
2. List of interaction points which are internal. The same file which described the connections of interaction points for test sequence generation is used in this part.
3. Information about Pascal types which define the PDUs.

We need the third input because the normalization module assumes that PDUs are defined in Estelle as a variant record and it enumerates the tag of this variant record. Corresponding to each value of the record tag which is defined in the case statement of the variant record a new Pascal record is defined in Estelle by the normalization module. Therefore the original record which defines the PDUs is lost after the normalization process.

A program called 'mappdu' reads the original specification and creates a table with two entries. The first entry is the Pascal types which define different PDUs

and are defined by the normalization module. The second entry is the actual names of different PDUs. The following is the example of LAPD PDU definition:

First an enumerated type called **FrameTag** defining the different types of PDUs is declared as:

```
FrameTag = (I, RR, RNR, REJ, SABME, DM, UI, DISC, UA, FRMR ,
XID, WRONG);
```

Then LAPD PDUs are defined as:

```
Frame_type= { Layer 2 frames after being decoded }
  RECORD
    C_R : bit; { C/R bit from Octet 2 }
    SAPI: SAPs; { from Octet 2 but handled by the multiplexer }
    TEI : byte; { from Octet 3 }
    CASE Frame : FrameTag OF
     I:
      ( N_S : byte { send sequence number from Octet 4 } );

     RR, RNR, REJ, I :
      ( N_R : byte; { receive sequence number from Octet 5 });

     SABME, DISC, UI, I:
      ( P : bit { Poll bit from Octet 4 in case of SABME and DISC and
Octet 5 in case of UI and I frames } );

     DM, UA, FRMR:
      ( F : bit { Final bit from Octet 4 } );

     UI, I, XID:
      (Informtn: I_type { from Network or Management layer});

                .
                .
                .
```

```
END;
```

Normalization creates a different type for each constant in the case statement. An example is the information PDU (I). The record defined for this PDU is:

```
frame_type_i =
        record
           c_r: bit;
           sapi:saps;
           tei: byte;
           n_s: byte;
           n_r: byte;
           p: bit;
           informtn: i_type
        end;
```

The table mapping Pascal types to PDUs now contains:

frame_type_i	I
frame_type_rr	RR
frame_type_rnr	RNR
frame_type_rej	REJ
frame_type_sabme	SABME
frame_type_disc	DISC
frame_type_ui	UI
frame_type_dm	DM
frame_type_ua	UA
frame_type_frmr	FRMR
frame_type_xid	XID
frame_type_wrong	Wrong

4.1 Test Step Generation Module

The test steps that our program generates are suitable for local test architecture.

If the modules in the specification describe the behaviour of a single layer only then the test architecture will be single layer (LS for local test architecture). Ideally multi-layer tests can also be obtained if the modules of the specification describe the behaviour of more than one layer of the protocol.

The test step generation module maps each normalized and simplified transition into a TTCN test step. Then spontaneous transitions are considered and some of the test steps (corresponding to WHEN transitions) are modified as a result. Constraints for the input and output events are derived from the PROVIDED clauses and the actions.

The following is the algorithm which implements the test step generation. Estelle reserved words are shown in bold. The input is the Estelle description of the protocol and the tables of PDU and internal interaction points. The output is test steps in TTCN.

Step-1. Parse the body of **SPECIFICATION** and read the table binding Pascal types to PDUs and the list of internal queues;

Step-2. Parse an Estelle module.

Step-3. Start printing the subtree for the first transition.

Step-4. Print the subtree name as "Subtree_" + ((number of modules read so far –1) "0"s) + transition number. For example the test step corresponding to the first transition in the second module is called Subtree_01.

Step-5. Pass the number of external SAPs in the module as formal SAPs of subtree. This allows defining different names for different SAPs when

array of SAPs is used. This way there is no need to change the name of SAPs inside the subtree because the SAP names are formal.

Step-6. Print the formal parameters of subtree. Formal parameters to subtrees are the parameters whose values are not bound to any specific value and the decision on their value is left to the test case. These formal parameters are:

a. The identifier list of **ANY** clause that has been used inside the transition anywhere besides as an index of interaction point since the array of interaction point is dealt with by making the SAP names formal.

b. Parameters or fields of PDUs which are passed by **WHEN** clause whose interaction point is external and are not bound to any fixed value by an equation inside the **PROVIDED** clause.

c. Parameters or fields of PDU which are used in the **WHEN** clause whose interaction point is internal.

d. Parameters or fields of PDU which are used in the**OUTPUT** clause whose interaction point is internal. In this case the program makes sure that there is no name conflict with global variables and parameters passed by **WHEN** clause.

Step-7. Print a guard for attachment of subtrees. If the expression in the guard evaluates to false abort the test and give an inconclusive verdict. The guard contains:

a. Inequalities inside the **PROVIDED** clause.

b. Constraints on the global variables.

Step-8. If the transition has a **WHEN** clause whose interaction point is external then:

a. if all the parameters passed by the interaction are PDUs then produce SAP!PDU1 [,SAP!PDU2, .., SAP!PDUn]

b. Otherwise print SAP!ASP where ASP is the name of interaction.

Step-9. If the transition has a **DELAY** clause (delay(min, max)), two instances of a timer called 'module name'_delay with values 'min' and 'max + PropDelay' are started before the protocol enters the initial state of the transition. Then the program produces the code shown in table 2.

Step-10. Produce code corresponding to the body of transition in the following manner:

a. Look for the first OUTPUT in the body of the transition to make sure that that transition was fired. Produce a receive event for the OUTPUT statement.

- If all the parameters passed by the OUTPUT are PDUs then produce SAP?PDU1 [,SAP?PDU2, .., SAP?PDUn]
- Otherwise produce SAP?ASP where ASP is the name of interaction.

Behaviour Description	Label	CRef	V	Comments
?TimeOut ModuleName_delay min	[label]			1
SAP?First Output in transition				
(body of the Transition)			PASS	
?TimeOut ModuleName_delay max + PropDelay				2
SAP?Spontaneous transition output				
(body of the spontaneous transition)				3
[goto label]				4
SAP?OTHERWISE			FAIL	
SAP?Spontaneous transition output				
(body of the spontaneous transition)				
[goto label]				5
SAP?OTHERWISE			FAIL	
EXTENDED COMMENTS				
1 - A label is produced if spontaneous transitions from that state exist. 1, 2 - Two copies of timer ModuleName_delay with durations of 'min' and 'max + PropDelay' are started every time the state from which the transition with delay begins is entered. 3 - TTCN code to verify that the spontaneous transition was fired 4, 5 - The statement 'goto label' is produced if the spontaneous transition goes back to its initial state.				

Table 2 TTCN code corresponding to delay clause.

If the body of transition has no output statement produce the code shown in table 3.

b. Produce subtree attachments corresponding to procedure calls which occur before the receive event (subtree attachment can't be bound to a receive event).

c. Bind all the assignments prior to the first OUTPUT statement to the corresponding receive event.

Behaviour Description	Label	CRef	V	Comments
START MinRespTimer	[label]			
?TIMEOUT MinRespTimer			PASS	
(body of transition)				
?First output of spontaneous transition				
(body of spontaneous transition)				
goto label				
SAP?OTHERWISE			FAIL	

Table 3 TTCN code produced for transitions with no output.

Behaviour Description	Label	CRef	V	Comments
(IDENTIFIER := low bound of DOMAIN)				
STATEMENT	[label]			
[IDENTIFIER <> high bound of DOMAIN]				
(IDENTIFIER := IDENTIFIER + 1)				
goto label				

Table 4 TTCN code corresponding to ALL statement.

d. For all the following OUTPUT statements produce receive events but the assignments preceding the receive are performed before that event.

e. Produce subtree attachment for procedure calls.

f. For the **all** statement in the form of:

'**all** IDENTIFIER : DOMAIN do STATEMENT'

produce the code shown in table 4.

Step-11. Produce code for the body of spontaneous transitions with no **DELAY** clause in a manner similar to step 10 as an alternative to the first output statement. Condition these alternatives to the predicate of their

PROVIDED clause. If the spontaneous transition goes back to its initial state generate a TTCN goto statement which goes back and waits for the first output in the main transition. Otherwise stop the test and assign an inconclusive verdict to the result of the test.

Step-12. For all receive events and all the SAPs in the module produce:

'SAP?OTHERWISE' FAIL'.

Step-13. Repeat steps 1 to 12 for all the transitions in the module.

Step-14. Repeat steps 1 to 13 for all the modules.

The rest of the Estelle constructs which are related to dynamic features of Estelle are ignored at this point. If any of these constructs have an effect on the test suite they must be considered during test case generation.

4.2 Test Case Generation

In this step the information obtained from data flow analysis (explained in section 3.3) and transition tours obtained from mstourgen (explained in section 3.4) are combined to obtain test cases.

A program called testgen finds a series of subtours generated by mstourgen that cover the transitions of a particular data flow function. Each subtour makes up a test case. The test cases are grouped under the name of the data flow function.

During subtree generation TTCNgen saves the heading of subtrees which contains the formal parameters of subtrees in a separate file. A program called testcase reads the output of testgen and the headings of all subtrees and creates un-

parametrized test cases. However from the heading of the subtrees the parameters that must be passed to subtree are known.

The following is an example of output generated by testgen.

```
Name of Function => Transmission of DISC Frames

-------------------------
     3 subtours
-------------------------
 subtour : 0
teiasgned ph.ph_dtind_sabme [ph.ph_dtreq_ua,l3.dl_estabind,timer_t200.stop,timer_t203.start] 81
idle t.stop nil 04
idle t.start nil 002
mulfrmestabd l3.dl_relizreq [ph.ph_dtreq_disc,timer_t200.start,timer_t203.stop] 119
idle t.start nil 02
active t.stop nil 003
active nil [t.expiry] 05
awaitngreliz timer_t200.expiry [ph.ph_dtreq_disc,timer_t200.start] 117
idle t.start nil 02
teiasgned nil [l3.dl_relizconfmtn] 16
teiasgned l3.dl_estabreq [ph.ph_dtreq_sabme,timer_t200.start,timer_t203.stop] 14
active t.start nil 01
idle t.stop nil 004
awaitngestab nil [l3.dl_relizind,timer_t200.stop] 93
active t.stop nil 03
```

This subtour covers some of the transitions which are part of the 'Transmission of DISC Frames' function.

The testcase generated by 'testcase' program is shown in figure 14.

As it can be seen from figure 14 only the header of subtrees with the list of their free formal parameters are currently generated and the actual parameterization should later be performed.

This method of test case generation is compact and highly modular. The drawback is that the test step generation module only looks inside one transition

TEST CASE DYNAMIC BEHAVIOUR				
Reference: lap_d_protocol/Transmission_of_DISC_frames/Transmission_of_DISC_frames_1 **Identifier:** Transmission_of_DISC_frames_1 **Purpose:** *To test Transmission of DISC frames* **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
Transmission_of_DISC_frames_1				
+Subtree_81[13,13_bcast,lm,ph] # (_SABME_c_r : bit, _SABME_sapi : saps, # _SABME_tei : byte, _SABME_p : bit)				
+Subtree_04[13,13_bcast,lm,ph]				
+Subtree_002[13,13_bcast,lm,ph]				
+Subtree_119[13,13_bcast,lm,ph]				
+Subtree_02[13,13_bcast,lm,ph]				
+Subtree_003[13,13_bcast,lm,ph]				
+Subtree_05[13,13_bcast,lm,ph]				
+Subtree_117[13,13_bcast,lm,ph]				
+Subtree_02[13,13_bcast,lm,ph]				
+Subtree_16[13,13_bcast,lm,ph]				
+Subtree_14[13,13_bcast,lm,ph]				
+Subtree_01[13,13_bcast,lm,ph]				
+Subtree_004[13,13_bcast,lm,ph]				
+Subtree_93[13,13_bcast,lm,ph]				
+Subtree_03[13,13_bcast,lm,ph]				
EXTENDED COMMENTS				

Figure 14 Example of a non-parametrized test case.

for symbolic substitution and PDU identification. If this method fails due to certain styles of specifications the alternative is to generate a flat test case with no subtree attachments for each subtour.

4.3 Distributed Test Architecture

If the test sequences are synchronizable (explained in section 3.5), the tests can be implemented using distributed test architecture. In this case the upper tester

or the interactions at the upper boundary of the protocol must remain unchanged. However since the lower tester is using the services of the lower layer protocols a mapping between receive and send interactions (REQUEST and INDICATION) must be made. All REQUEST primitives must be changed by INDICATION and vice versa.

4.4 Parameterization of Subtrees

The process of parametrization can be automated by defining a convention to specify what can be enumerated and what is fixed and what a type of enumeration is needed in case of the types which have unlimited bounds like integers or types which have cover a big range of values. Enumeration can be done by defining a loop control either for each different data type or best for each individual parameters of subtrees which must be enumerated.

The following describes an automatic parameter enumeration algorithm.

4.4.1 Algorithm

We assume that the set of data flow functions F of the protocol is identified, the subtours are generated and the set of subtours S covering each function is determined. We define P(f) to be the set of input parameters (I-nodes) in each function f. Let S(f) to be the set of subtours covering function f with $s \in S(f)$ being composed of simplified normal form transitions t1, t2...tn. We define for each transition ti, input(ti) and provided(ti) to be the input interaction (or null) and the provided clause of the transition ti, respectively. Parameters of an input

interaction can be obtained with the application of Parameters(input(ti)). Note that these parameters and the provided clause are assumed to be expressed in Estelle.

Step1. Repeat the steps 2 through 5 for each function f in F, for each subtour s in S(f), for each ti in s if input(ti) <> null.

Step2. FuncParSet = Parameters(input(ti)) $\bigcap$ P(f);

ProvParSet = Distinct parameters referred to in provided(ti);

Step3. for each p $\in$ ProvParSet and p $\notin$ FuncParSet select a value so as to satisfy the provided clause.

Step4. enumpos(FuncParSet) = { i | for each $p \in FuncParSet$, i is the number of possible enumerations, i.e., the number of possible values for enumeration parameters, an implementation dependent value for others};

highbound = maximum (enumpos(FuncParSet));

Step5. for each p'' $\in$ Parameters(input(ti)) and p'' $\notin$ ProvParSet select a default value if it exists or else select a random value from its value domain;

for i := 1 to highbound do

begin for each p' $\in$ FuncParSet select a new value from its value domain, if all values are exhausted, assign one of the earlier values;

update the TTCN subtree instantiation for input(ti) with all of its parameters;

end;

Therefore if a parameter of input is not mentioned in the **PROVIDED** clause or a parameter inside the provided clause is not part of the data flow function,

only a default value is chosen and used during the test. If a parameter of input or variable is part of the data flow function and is also mentioned in the provided clause its values are enumerated.

4.4.2 Application of the algorithm

Figure 15 shows the same subtree shown in figure 14 after it was parametrized by the user by the application of parameterization algorithm. In this case the C/R bit of the SABME PDU (_SABME_c_r) and the poll bit of SABME PDU (_SABME_p) are enumerated using the **ENUM**(enumerate) statement which is expected to be included in TTCN [6]. The BNF for the **Enumerate** construct is:

Enumerate ::= ENUM["("ValueList")"]TreeReference

ValueList ::= Number {"," Number } | ";" ValueList

From figure 15 we can see that the **tei** and **sap** parameters of the PDU were not enumerated. The reason is that although the provided clause of the transition does not specify any constraint on the value of these parameters, in reality the value of **tei** field is fixed and the value of **sap** is constrained on the configuration of the LAPD and Network layer interface. In the case of **tei** checking the value of this field is handled by higher priority transitions. The value of the **sap** is specified in the specification of the multiplexer module.

TEST CASE DYNAMIC BEHAVIOUR				
Reference: lap_d_protocol/Transmission_of_DISC_frames/Transmission_of_DISC_frames_1 **Identifier:** Transmission_of_DISC_frames_1 **Purpose:** To test Transmission of DISC frames **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
ENUM(n2ucomand, n2uresp; poll, u2nresp #)LOCAL(SAPVar, PFBit)				
LOCAL(SAPVar, PFBit)				
+Subtree_81[l3,l3_bcast,lm,ph](SAPVar, # 0, tei, PFBit)				
+Subtree_04[l3,l3_bcast,lm,ph]				
+Subtree_002[l3,l3_bcast,lm,ph]				
+Subtree_119[l3,l3_bcast,lm,ph]				
+Subtree_02[l3,l3_bcast,lm,ph]				
+Subtree_003[l3,l3_bcast,lm,ph]				
+Subtree_05[l3,l3_bcast,lm,ph]				
+Subtree_117[l3,l3_bcast,lm,ph]				
+Subtree_02[l3,l3_bcast,lm,ph]				
+Subtree_16[l3,l3_bcast,lm,ph]				
+Subtree_14[l3,l3_bcast,lm,ph]				
+Subtree_01[l3,l3_bcast,lm,ph]				
+Subtree_004[l3,l3_bcast,lm,ph]				
+Subtree_93[l3,l3_bcast,lm,ph]				
+Subtree_03[l3,l3_bcast,lm,ph]				
EXTENDED COMMENTS				

Figure 15 Example of a parametrized test case.

Chapter 5 THE DECLARATIONS AND CONSTRAINT GENERATION MODULES

The processing of the transitions of Estelle was explained in the previous chapter. This chapter explains how the rest of Estelle declarations (e.g. variable and type declarations) and constraints are processed by TTCNgen.

5.1 Declarations Module

The first time that the declaration module is called by TTCNgen program is after the **SPECIFICATION** part is parsed. At this point Estelle global declarations are known and their corresponding declarations are made in TTCN. Then TTCNgen starts parsing each module separately. When the parsing of a module is finished, all its local declarations are available. Before the parsing of the next module is started they are destroyed. Therefore before starting to parse the next module the local declaration of that module must be processed.

The following are the TTCN declarations which are generated by the declarations module:

1. Subranges, enumerated and primitive types are declared as TTCN user-types. For Estelle primitive types no base type is declared in TTCN. Pascal records which do not define a PDU are defined as ASN.1 **SEQUENCE** type[25]. Pascal sets are defined using ASN.1 **SET OF** construct. Arrays are defined using ASN.1 **SEQUENCEOF** construct.

2. Estelle primitive functions are declared as TTCN operations. The arguments passed to operations match the arguments in the function declaration.
3. Constants in TTCN are declared in the same fashion as they are declared in Estelle.
4. All Estelle variables are declared as test case variables in TTCN.
5. Estelle interaction points which are external are declared as PCOs in TTCN.
6. A timer called MinRespTime is declared for every specification. The duration of this timer must be equal to the maximum time that it takes for the implementation to respond to an external behaviour. The duration of this timer must be supplied by the user. The function of this timer is to check if the implementation does not respond to events that must be ignored. Also for each module which has at least one transition with delay clause a timer name "ModuleName"_delay is declared.
7. For the interactions passed in external channels which have at least one non PDU parameter or have no parameters an ASP declaration is made. If the ASP has parameters a parametrized ASP constraint table is also defined in TTCN. The fields of ASP whose types are PDUs are defined as PDU field types.
8. For each Pascal record appearing in the previously mentioned table of PDUs a TTCN PDU declaration and a parametrized PDU constraint declaration is made.
9. For the interactions that have more than one PDU parameters or have a mixed non-PDU and PDU parameters a parametrized ASP constraint table is declared. Then corresponding to each PDU a chained parametrized PDU

constraint reference is created. Corresponding to each field of the PDU an argument is passed to the ASP constraint table and then the declaration of the ASP constraint passes those arguments to the parametrized PDU reference.

Figure 16 shows the TTCN user type definitions generated for LAPD protocol. Figure 17 gives an example of ASN.1 type definition for frame_type_frmr record which was declared in the specification of LAPD protocol.

USER TYPE DEFINITIONS			
Name	**Base Type**	**Definition**	**Comments**
byte	integer	0..255	
bit	integer	0..1	
saps		(sap0,sap1,sap16,sap63)	
i_type			
frametag		(i,rr,rnr,rej,sabme,dm,ui,disc,ua,frmr,xid,wrong)	
reason		(cferr,inc_length,info_np,undef,i2long)	
err_type		(a,b,c,d,e,f,g,h,i_err,j,k,l,m,n,o)	

Figure 16 Example of TTCN type definition.

User ASN.1 Type Definition
Type Name : frame_type_frmr
ASN.1 Definition or Reference

SEQUENCE OF {
c_r bit,
sapi saps,
tei byte,
f bit,
rejed_frame frametag,
control_field byte,
curr_v_s byte,
curr_v_r byte,
rej_c_r bit,
z bit,
y bit,
x bit,
w bit
}

Figure 17 Example of ASN.1 type definition.

5.2 Constraint Generation Module

This section explains the details of the module which generates the constraints references for TTCN events. For each input and output events of TTCN which has a parameter (PDU or SDU) a constraint must be specified. An event constraint specifies what the legal values for the parameters of the receive events or send events are. The constraint generation module is called right after creating the input and output events at the end of steps 8 and 10 of the algorithm that produces test steps explained in section 4.2.

All the constraints generated by this module are parameterized constraints. We chose parametrized constraint tables because the constraints on the event parameters often depend on the formal parameters of subtree or on operations and the only way of specifying these constraints is using parameterized constraints.

Two types of constraints are generated by the constraint generation module. The first type is the constraints for the send events and the second type is the constraints for the receive events. This is due to the fact that the send event is created from the WHEN clause and the receive event is created from the OUTPUT statement and the procedures must know whether the parameter passed is an OUTPUT statement or a WHEN clause.

There are some basic differences between the input and output parameters. One difference is that WHEN clause is similar to the heading of a procedure call and parameters passed are always a variable and not a constant and they are declared in the CHANNEL declaration part. Whereas output statement is similar

to a procedure call and therefore the parameters passed are explicitly stated in the body of the transition and can be of any sort, e.g. a constant.

The following describes the algorithm used in generation of parameterized constraints for the send events.

1. Find the parameters passed by the **WHEN** clause. If no parameter is passed by the **WHEN** clause do not generate a constraint reference and return.
2. If ASP has at least one non-PDU parameter the main constraint reference is called 'ASPName'_Con.

    a. For non-PDU parameters look inside the **PROVIDED** clause and see if they are directly used in an equation. If yes pass their value found from the equation to the constraint reference. Otherwise pass the formal parameter which was generated in step 6.b of the algorithm given in section 4.2, as its value.

    b. For PDU parameters create a parameterized PDU constraint reference and put it in place of the constraint for that field (This is called constraint chaining).

3. For each PDU parameter print the constraint name as 'PDUName'_Con. For each field of PDU look inside the PROVIDED clause and see if they are directly used in an equation. If yes pass their value found from the equation to the constraint reference. Otherwise pass the formal parameter which was generated in step 6.b of the algorithm given in section 4.2, as its value.

The following describes the algorithm used in generation of parameterized constraints for the receive events.

1. Look at the parameters passed by the **OUTPUT** statement. If no arguments are passed by the **OUTPUT** statement do not generate a constraint reference and return.
2. If ASP has at least one non PDU parameter the main constraint reference is called 'ASPName'_Con. For PDU arguments call the constraint reference 'PDUName'_Con. If the interaction has no non-PDU arguments place the constraint reference inside the subtree, otherwise chain them inside the ASP constraint.
3. If the argument passed to the **OUTPUT** statement is a constant or an expression put the exact value in the constraint table.
4. If the argument passed to the **OUTPUT** statement is a variable then look in the assignments inside the body of the transition prior to the **OUTPUT** statement. For non PDU parameters of the **OUTPUT** statement check if the exact variable appears in the left hand side (LHS) of an equation. For PDU parameters check if the reference to the field appears in the LHS of an equation (in the form of Variable.Field). In both cases the constraint is the right hand side (RHS) of the equation.
5. If the constraint can not be found in any of the equations inside the transition before the **OUTPUT** statement then assume that the assignment was done in earlier transitions. In the case of PDUs the other possibility is that the field

which does not appear in the assignment must or can be absent.

The following is the definition of SABME PDU in Estelle.

```
frame_type_sabme =
        record
           c_r: bit;
           sapi: saps;
           tei: byte;
           p: bit
        end;
```

Figure 18 shows the constraint declaration for SABME PDU. An example of a subtree with constraint references is given in chapter 6.

PDU Constraint Declaration	
PDU Name: SABME	Constraint Name: SABME_Con(c_r_value, sapi_value, tei_value, p_value)
Field Value Information	
Field Name	Value
c_r	c_r_value
sapi	sapi_value
tei	tei_value
p	p_value

Figure 18 Example of a PDU constraint declaration.

Chapter 6 Implementation

This chapter explains the details of the implementation of the test generation tool. First the languages used in the implementation of different programs are specified. Then the module decomposition of the programs are explained. The principle used in partitioning the task in different modules was the separation of concerns and information hiding. Also wherever there was a need to give the user a chance to process or change the information manually, the modules were implemented as separate programs.

Information hiding is a software design method in which one hides the software design decisions from other modules. In particular, this is done whenever the design decisions are likely to change. The design decisions are mainly algorithms and the representation of data structures.

The languages used to implement the programs in our test generation tool and the dependency between different programs are shown in Figure 19. The arrows indicate that the output of the program is passed as a file to the next program. For the programs that were implemented by the author of the thesis, the languages which were used in their implementation are mentioned in the figure. The following sections explain the details of different modules implemented by the author.

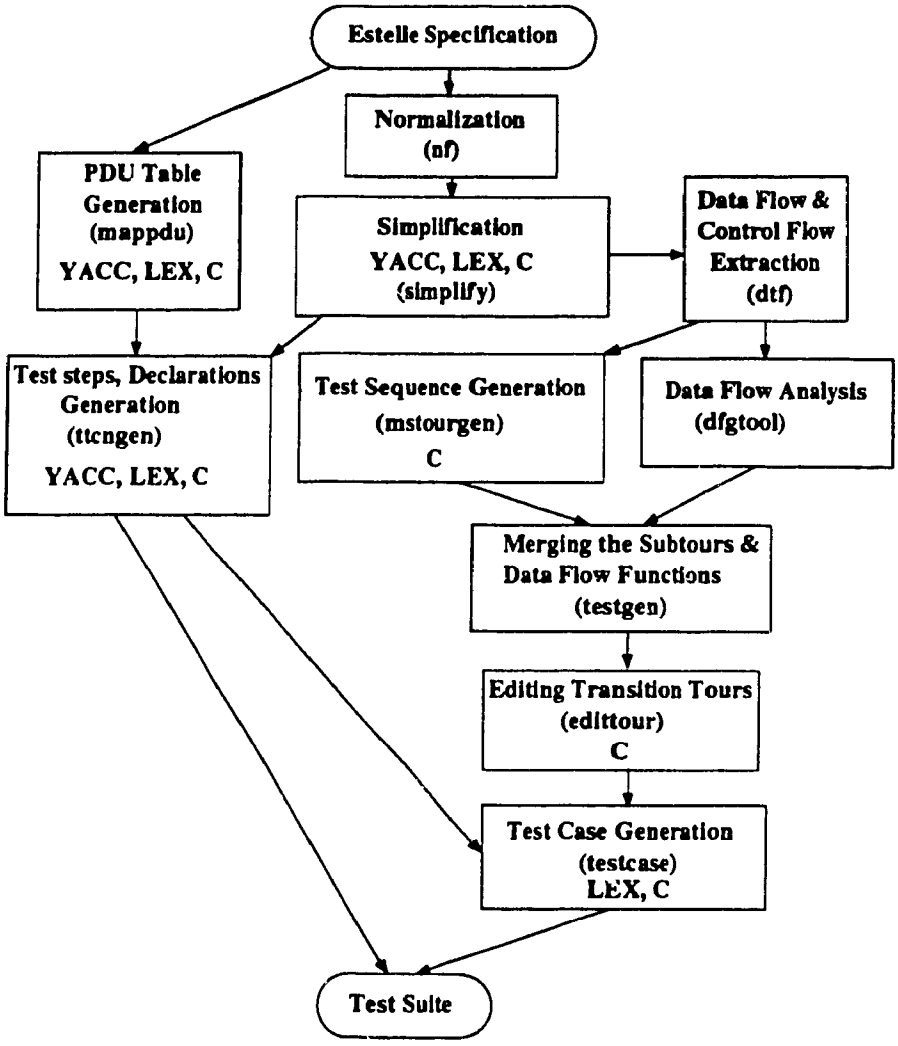


Figure 19 Main modules of the test generation tool.

6.1 mappdu

The **mappdu** program consists of two modules. The main module was written in LEX interleaved with the C language[25,26]. This module finds where the record definitions of PDUs are placed in the specification and then calls the parser. The second module is a parser which only contains the type declaration part of

PASCAL language. The parser makes a list of the case constants used in the record and then the main module creates a table which maps the types generated by the normalization module to the value of the case constants.

6.2 simplification

The box labeled **simplify** is really made of the two programs called **simplify** and **transform** programs. The **simplify** program was implemented in LEX and C languages and the **transform** program was implemented in LEX, YACC and C languages.

Usually the user only runs **simplify**. **simplify** creates a UNIX **pipe** to **transform** [27]. The **transform** program simplifies the predicates inside the provided clause of the transitions as explained in section 3.2 and sends them to **simplify**. Then **simplify** substitutes the new predicates in the provided clause of the transitions.

Sometimes it is desirable to change or delete some of the predicates generated by the transform. The following example shows a predicate generated by **transform** which can not be satisfied. The example is taken from the specification of LAPD protocol. The original predicate is:

(((true) and not ((v_a <= frmrecvd.n_r) and (frmrecvd.n_r <= v_s))) and (not (frmrecvd.p_f = final))) and (not (frmrecvd.c_r = n2ucomand))

transform generates the following three predicates:

(frmrecvd.n_r > v_s) and (v_a > frmrecvd.n_r) and (frmrecvd.p_f = final) and (frmrecvd.c_r <> n2ucomand)

(frmrecvd.n_r > v_s) and (v_a <= frmrecvd.n_r) and (frmrecvd.p_f = final) and (frmrecvd.c_r <> n2ucomand)

(frmrecvd.n_r <= v_s) and (v_a > frmrecvd.n_r) and (frmrecvd.p_f = final) and (frmrecvd.c_r <> n2ucomand)

In order to satisfy the first predicate the value of **v_a** (which is the sequence number of the last frame acknowledged) must be greater than **v_s** (which is the sequence number of the next frame to be transmitted) which is impossible.

The reason for implementing the task of simplification of the provided clauses in two separate programs was to give the user a chance to modify the predicates generated by the **transform** program before the transitions are generated. This can be done by asking the **transform** program to dump the predicates in a temporary file. Then the user can edit the temporary file and ask the **simplify** program to read the predicates from the temporary file rather than from the UNIX **pipe**.

Of course this can be done by editing the output of the simplification program. However the first solution is preferred because the transition numbers are indicated as a PASCAL comment inside the transition declaration. Deleting a transition from the middle by hand causes these numbers to be different from the real transition numbers. Also by editing the TempFile the user can concentrate on the predicates. If due to deleting or a adding a transition by hand the numbering of the transitions are found to be wrong the transitions can be renumbered by running the simplification program again.

6.3 ttcngen

TTCNgen is the most important part in Figure 19. The simplify and mappdu programs were just designed to prepare the inputs to this program. TTCNgen is the basis of the design of testcase generation programs (mstourgen, testgen, edittour and testcase), since they assume that there is a subtree made for each transition. This program is the link between the Estelle specification and the programs which use data and control flow graph abstraction of the specification.

We used an interactive TTCN editor in order to visualize the test cases/ steps, constraints and declaration tables generated and possibly produce hard copies of these tables. Such an editor is described in [28]. Our program produces the TTCN tables in the intermediary forms required by the editor. This step requires producing a separate file for each test step and test case in a certain format. The TTCN editor enables the user edit the tables interactively on a workstation.

The TTCNgen program is made of 18 different modules. Each module is made of many procedures and functions that have the same concern such as printing expressions, generating constraint references. The parser was written in YACC and the lexical analyzer was written in LEX. The other modules are written in C language. Most of these modules are compiler utilities. They perform operations such as parsing, hashing, semantic checking, etc.

This section uses some of the software engineering terminology described in [29]. The module decomposition and description of the dependencies and interfaces between the modules are explained in the following subsections.

6.3.1 Module Decomposition

6.3.1.1 System Modules

System modules are the modules which communicate with the environment. They are mostly system dependent. The system modules of TTCNgen were written for the systems running UNIX operating system. Most of the system interfaces in this module can easily be ported to any system which has a C compiler. The system modules of TTCNgen are:

lexio.c — This module reads the Estelle specification of the protocol and provides input to the LEX as an array of characters to improve the performance of LEX. It also merges the files that must be included in the specification because of the **include** compiler directive. It exports two interfaces to LEX:

char input() which returns a full line to lex.l and **void unput(char)** which puts a character back into the input stream (unreads the character).

files.c — This module opens and closes the files required for compact dynamic behaviour tables and declarations. It then returns an array of file pointers corresponding to the different fields of each table. The **void CloseFiles()** routine merges and then closes the temporary files where it is required. The exported interfaces are:

a. NameTmpFiles: Initializes the name of temporary files to be used by the compiler.
b. RemoveTemps: Deletes the temporary files.
c. OpenFiles: Opens the files required for the subtree and the temporary files.
d. CloseFiles: Closes the files required for the subtree and the temporary files.
e. LF: Places a line feed for all the columns of the table.
f. InitDefs: Opens and places the initialization codes in all the files used for the declarations.
g. OpenTypeFiles: Opens all the files used for type definitions.
h. CloseTypeFiles: Closes all the files used for type definitions.

6.3.1.2 Requirements Module

Requirement modules are the modules which are directly related to the specification of the problem. The requirement modules of the TTCNgen are:

ttcngen.c — This modules contains the main routine of the program. It picks up the command line arguments. Also checks if the input files exist.

lex.l — This module performs the lexical analysis of the specification. It exports the **yylex** interface to parse.y.

parse.y This module parses and checks the syntax of the specification. It exports the **yyparse** interface to ttcngen.c.

semantics.c This module checks the semantic of the specification. These three modules (lexical analyzer, parser and semantic checking) are an essential part of any compiler based program. The semantics.c module exports many semantic checking routines such as chkconnect (check a connect statement) to parse.y.

errmsg.c This module prints error messages for different syntax, semantics or exception conditions such as file errors. The exported interfaces are:

a. error: Prints a fatal error message.
b. warning: Prints a warning message.
c. unimpl: Prints a message for use of an unimplemented feature.
d. userror: Prints a fatal error message and exits the program.
e. cerror: print message and exit for internal compiler errors and exits the program.

print.c This module prints TTCN expression and statements from Estelle parse tree. The exported interfaces are:

a. PrintType: Prints an ASN.1 type from an Estelle type.
b. xpr: Prints any expression.
c. PrintIntOut: Prints the internal outputs in the form of an assignment.

d. PrintStmts: Converts and prints TTCN statements from Estelle statements.

subtree.c This module generates the **Behaviour Description** part of the test steps. It exports **PrintSubtree** Interface to code.c which prints all the subtrees.

constraint.c This module generates constraint references for the external events of the IUT. The exported interfaces are:

a. PrintOutCons: Prints constraint references for the output statement.

b. PrintInCons: Prints constraint references for the input (**when**) clause.

declarations.c This module generates the declarations part of the TTCN. The exported interfaces are:

a. genbodydefs: Generates constant, variable, PCO, ASP and function declarations from the module body definitions.

b. DefTimer: Generates timer declarations.

c. DeclProcs: Generates procedure declarations.

6.3.1.3 Software Decision Modules

Software decision modules contain the parts of the software (algorithms, data structures, etc.) which are not specified in the specification. Their need is

determined by the software engineer for implementation purposes. The software decision modules of ttcngen are:

code.c — This module is the link between the parser and subtree generation modules. It is called by the parser every time a module of the specification is parsed and this module calls the declarations and subtree generation modules. The exported interfaces are:

a. genbodycode: It is called each time a module body is parsed and generates the subtrees and the timer and procedure declarations.
b. abortcode: Aborts the TTCN code generations.

hash.c — This module stores the string in a hash table for faster look up and search for the symbols and identifiers. The exported interfaces of this module are:

a. entername: enters the name in the hash table.
b. uniqname: guaranties that the identifier has unique name by adding suffix to it if it has conflict with existing names.

types.c — This module keeps internal representation of the PASCAL defined types in the specification and stores their name. The exported interfaces of this module are:

a. typeof: Returns the type of an identifier.

b. Routines which let the user define user defined types such as records, subranges, etc. and representation of user defined types

symbol.c This module stores the name and types of symbols such as identifiers, functions, etc. and resolves the scope of identifiers.The exported interfaces of this module are:

a. declare: Declares a symbol and associates types and other attributes to it.

b. lookup: Looks up for an existing symbol.

trans.c This module stores some of the information about transitions like number of inputs and outputs in an array.

tree.c This module builds the parse tree.

utils.c This module contains many utilities. It provides the subtree generation the list manipulation facilities such as adding elements to the list and freeing a linked list independent of the types of the elements. It also generates constants for the low or high binds of undefined types. It also has routines to read optional inputs like connection of interaction points and the PDU table.

6.3.2 Module dependencies

Figure 20 shows the dependencies between the modules used in the implementation of ttcngen. Since every module except code.c uses the errmsg.c, this

module was not shown in the figure. The arcs represent dependency of a module on another. Arcs labeled H mean that the first module does not know and does not care how the information in the other module is represented and stored (the second module hides information).

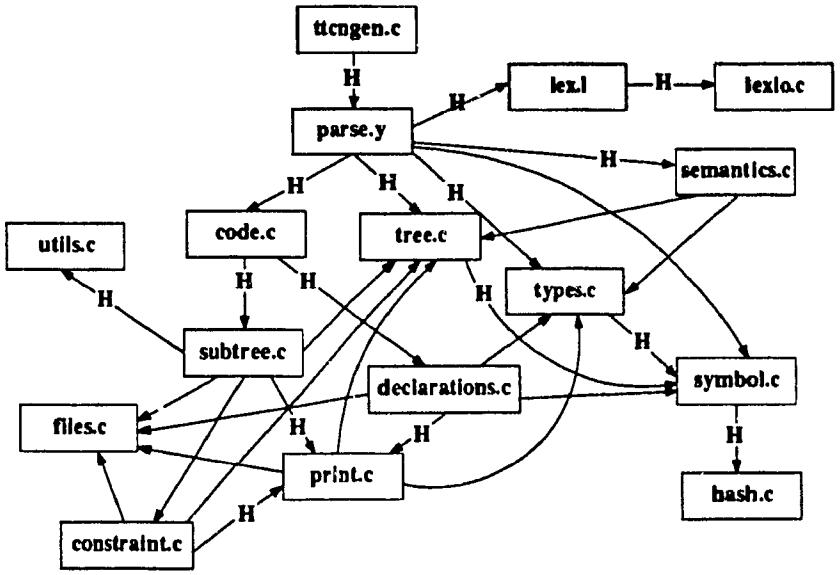


Figure 20 Module dependency diagram for ttcngen.

6.4 mstourgen, edittour and testcase

The **mstourgen** program was implemented in one single module in C language. The algorithm of this program was explained in section 3.6.

Edittour was also implemented in LEX and C language. Edittour makes calls to Sunview's library procedure and can only be run on SUN workstations. The details of this program is explained in section 3.7

The testcase program was implemented in LEX and C languages. The algorithm of this program is explained in section 4.2.

Chapter 7 Application of the TTCNgen on LAPD and Transport Protocols

In this chapter examples of TTCN subtrees generated by applying the ttcngen on the Estelle specification of LAPD and Simplified Class 2 protocol[31] are demonstrated. The examples demonstrate different features and capabilities of the ttcngen program.

7.1 Example of TTCN subtrees for LAPD protocol

7.1.1 Example 1

We use the subtree generated for transition 217 as our first example. This example shows the main features of the ttcngen program. Transition 217 fires when LAPD receives an I(acknowledged information) PDU with bad N(S) sequence number (<> V(R)). In this case LAPD rejects the frame. The declaration of the I PDU is shown in figure 21.

PDU Type Declaration		
PDU Name: I	PCO Type. I3	Comments:
PDU Field Information		
Field Name	Type	Comments
c_r	bit	
sapi	saps	
tei	byte	
n_s	byte	
n_r	byte	
p	bit	
informtn	i_type	

Figure 21 Declaration of the I PDU.

The declaration of the normalized and simplified transition 217 is:

```
trans
        { 217 }
        when ph.ph_dtind_i
        provided (frmrecvd.n_r = v_a) and (frmrecvd.n_r < v_s) and (frm-
recvd.c_r = n2ucomand) and not(rejexcept) and (frmrecvd.n_s <> v_r) and
not(ownrecverbz) and not(peerrecbz)
        from mulfrmestabd
        to mulfrmestabd
        begin
            update_buffer(frmrecvd.n_r);
            rejexcept := true;
            frmvar_rej.tei := tei;
            frmvar_rej.c_r := u2nresp;
            frmvar_rej.n_r := v_r;
            frmvar_rej.p_f := frmrecvd.p;
            output ph.ph_dtreq_rej(frmvar_rej);
            ack_pending := false
        end;
```

The definition of the interaction **ph_dtind_i** is:

```
channel phys_layer_ap(user, provider);
        by provider:
            ph_dtind_i(frmrecvd: frame_type_i);
```

Therefore framerecvd (received frame) is passed by the interaction **ph_dtind_i** and since the type **frame_type_i** appears in the PDU table declaration and declares it to be **I** PDU and all the references to this PDU **frmrecvd** are replaced by **I**. The declaration of **frame_type_i** is:

```
frame_type_i =
            record
                c_r: bit;
```

```
        sapi: saps;
        tei: byte;
        n_s: byte;
        n_r: byte;
        p: bit;
        informtn: i_type
    end;
```

Similarly the declaration of **ph_dtreq_rej** is:

```
ph_dtreq_rej(frame: frame_type_rej);
```

This declaration tells the program that this interaction outputs a **REJ** PDU. A spontaneous transition which may fire from the **mulfrmestabd** state instead of transition 217 is the transition 263. The following is the declaration of transition 263:

```
trans
        { 263 }
        provided ack_pending
        from mulfrmestabd
        to mulfrmestabd
        begin
            frmvar_rr.tei := tei;
            frmvar_rr.c_r := u2nresp;
            frmvar_rr.n_r := v_r;
            frmvar_rr.p_f := not_final;
            frmvar_rr.n_r := v_r;
            output ph.ph_dtreq_rr(frmvar_rr);
            ack_pending := false
        end;
```

After the steps explained in section 4.1 are applied to this example the subtree

for transition 217 shown in figure 22 is produced by the ttcngen program. In here we explain this tree in detail.

TEST STEP DYNAMIC BEHAVIOUR				
Reference: lap_d_protocol/lap_d_body/Subtree_217 **Identifier:** Subtree_217(l3:l3sap,l3_bcast:l3bcastsap,lm:lmsap,ph:phsap, _I_sapi : saps, _I_tei : byte, _I_n_s : byte, _I_p : bit, _I_informtn : i_type) **Objective:** Unknown **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
[v_a < v_s AND NOT(rejexcept) AND # _I_n_s <> v_r AND NOT(ownrecverbz) AND # NOT(peerrecbz)]				
ph!I		I_Con (n2ucomand, _I _sapi, _I_tei, _I_n_s, v_a, _I_p, _I _informtn)		
ph?REJ(rejexcept := true)(frmvar_rej.tei := # tei)(frmvar_rej c_r := u2nresp)(frmvar_rej.n_r := # = v_r)(frmvar_rej p_f := _I_p)	1	REJ_Con (u2nresp, frmvar_rej .sapi, tei, v _r, _I_p)	(PASS)	
+update_buffer(v_a)				
(ack_pending := false)				
ph?RR[ack_pending](frmvar_rr tei := tei) # (frmvar_rr.c_r := u2nresp)(frmvar_rr.n_r := v_r) # (frmvar_rr.p_f := not_final)(frmvar_rr n_r := v_r)		RR_Con(u2nresp , frmvar_rr .sapi, tei, v _r, not_final)		
(ack_pending := false)				
[(v_a < v_s) AND (NOT(rejexcept)) # AND (_I_n_s <> v_r) AND (NOT # (ownrecverbz)) AND (NOT(peerrecbz))]				
-> 1				
[NOT((v_a < v_s) AND (NOT # (rejexcept)) AND (_I_n_s <> v_r) AND # (NOT(ownrecverbz)) AND (NOT(peerrecbz)) #)]			INCONC	
l3sap?OTHERWISE			FAIL	
l3bcastsap?OTHERWISE			FAIL	
lmsap?OTHERWISE			FAIL	
phsap?OTHERWISE			FAIL	
[NOT((v_a < v_s) AND (NOT(rejexcept)) # AND (_I_n_s <> v_r) AND (NOT # (ownrecverbz)) AND (NOT(peerrecbz)))]			INCONC	

Figure 22 TTCN subtree produced for subtree 217.

The **provided** clause of transition 217 binds the value of **n_r** field of the **I** PDU to the variable **v_a**. The **tei, sapi, n_s, p** and **informtn** fields of PDU are not bound to any fixed value, however, since the tester must send this PDU to IUT, their value must be known at execution time and are passed to the subtree as formal parameters : **_I_sapi, _I_tei, _I_n_s, _I_p** and **_I_informtn**. The **n_r** field of PDU is substituted by the variable **v_a** everywhere in the subtree.

The external SAPs of the LAPD module are **l3, l3_bcast, lm** and **ph** are passed as formal SAP parameters to the subtree. As an alternative to every receive event, ExternalSAP?OTHERWISE is specified which monitors these external SAPs and if an invalid behaviour is observed the tester will fail the IUT.

The condition on global and formal parameters for the test to be valid is:

[v_a < v_s and NOT(rejexcept) and _I_n_s <> v_r and NOT(ownrecverbz) and NOT(peerrecbz)]

Note that the expression *v_a < v_s* is concluded by the program by substituting the value of **frmrecvd.n_r** (**v_a**) in the expression *frmrecvd.n_r < v_s*. If the condition for attachment of subtree_217 in the value of global variables (**v_a, v_s, rejexcept, ownrecverbz** or **peerrecbz**) or the formal parameter **_I_n_s** evaluates to *FALSE* an inconclusive verdict is concluded and the test is stopped.

The first action taken by the tester is to send an I PDU. After doing so either a **REJ** PDU, as specified in the body of transition, or an **RR** PDU, as a result of the spontaneous transition, may be received. The reception of RR PDU is valid if the value of the global variable **ack_pending** is to *TRUE*. Therefore the validity

of reception of RR PDU is conditioned to the value of this variable.

At the beginning the subtree waits for the arrival of **REJ** or **RR** PDU's to identify which transition was fired. All the assignments appearing before the OUTPUT of **REJ** PDU are bound to the reception of this PDU. This is important since the spontaneous transition might depend on or use one of the variables appearing on the left hand side of these assignments and also if an error occurs the correct value of the variables before the error occurred will be known.

If an **RR** PDU is received the test step concludes that the transition 263 was fired instead of transition 217, which is a valid behaviour. In that case the body of transition 263 is traced and since transition 263 goes back to its initial state the tester must still receive a **REJ** PDU. Before doing so the subtree checks if the condition to fire subtree 217 is still valid. In that case the tester executes the **goto** statement. Otherwise an inconclusive verdict is assigned to the result of the test. Logically on the second run the receipt of RR PDU is not valid any more since the body of the transition 263 changes the value of **ack_pending** to *FALSE*.

After the receipt of **REJ** PDU the subtree **update_buffer** is called which is supposed to trace or verify the side effects of the procedure call made by the transition 217 before REJ PDU was sent. Note that the parameter passed to this procedure is also replaced by its value. Then the assignment 'ack_pending := false' is made in order to keep track of the value of this variable of IUT.

7.1.2 Example 2

The next example is the subtree generated for transition 121. This example

shows how ttcngen program handles the events of the internal interaction points. The declaration of the subtree 121 is:

```
trans
{ 121 }
when 13.dl_dtreq
from mulfrmestabd
to mulfrmestabd
begin
   frmvar_i.c_r := u2ncomand;
   frmvar_i.informtn := info;
   output i_q_in.iqdup_i(frmvar_i)
end;
```

The declaration of the **iqdup_i** interaction is:

```
iqdup_i(frame: frame_type_i);
```

According to the rule 6.d of the algorithm presented in section 4.1, the output to the internal interaction point is converted to an assignment. In this case the right hand side of the assignment which must be the parameter of the internal interaction is called **frmvar_i**. The left hand side of the assignment is the name of the subtree parameter which stores the value of the interaction parameter. Since the identifier **frame** is used for many other parameters it is uniquely named by the software. In the case of **iqdup_i** interaction it is given the name **ACframe**. The subtree generated is shown in figure 23.

According to rule 6.c, the parameter **frame** of the internal interaction **iqdup_i**

TEST STEP DYNAMIC BEHAVIOUR				
Reference: lap_d_protocol/lap_d_body/Subtree_121 **Identifier:** Subtree_121(l3:l3sap,l3_bcast:l3bcastsap,lm:lmsap,ph:phsap, _info : i_type, ACframe : I) **Objective:** Unknown **Default Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
l3!dl_dtreq		dl_dtreq_Con(_info)		
(frmvar_i.c_r := u2ncomand)				
(frmvar_i.informtn := _info)				
(ACframe := frmvar_i)				

Figure 23 Example of the subtree generated for transition 121.

in transition 121 is stored in a variable. This variable is later passed to a transition which takes this interaction as its input. The second transition which consumes the internal event is determined during the test case generation. An example of such a transition is transition 122. The declaration of transition 122 is:

```
trans
{ 122 }
when i_q_out.iqdup_i
provided not(peerrecbz) and (v_s <> (v_a + k16_signal)) and
t200_running from mulfrmestabd
to mulfrmestabd
begin
   lastisent_i := frame;
   lastisent_i.tei := tei;
   lastisent_i.n_s := v_s;
   lastisent_i.n_r := v_r;
   lastisent_i.p := not_poll;
   output ph.ph_dtreq_i(lastisent_i);
   save_i_frame_i_i(lastisent_i);
   v_s := v_s + 1;
```

```
        ack_pending := false
    end;
```

This parameter of the internal interaction **frame** is again named **ACframe.** The subtree generated for transition 122 is shown in figure 24.

TEST STEP DYNAMIC BEHAVIOUR				
Reference: lap_d_protocol/lap_d_body/Subtree_122 **Identifier:** Subtree_122(13:lsap,13_bcast:lbcastsap,lm:lmsap,ph:phsap, ACframe : I) **Objective:** Unknown **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
[((NOT(peerrecbz)) AND (v_s <> v_a + # k16_signal)) AND (t200_running)]				
ph?I(lastusent_i := ACframe)(lastusent_i.tei := # tei)(lastusent_i.n_s := v_s)(lastusent_i.n_r := # v_r)(lastusent_i.p := not_poll)	1	I_Con (lastusent_i.c_r, lastusent_i.sapi, tei, v_s, v_r, not_poll, lastusent_i.informtn)	(PASS)	
+save_i_frame_i_i(lastusent_i)				
(v_s := v_s + 1)				
(ack_pending := false)				
ph?RR[ack_pending](frmvar_rr.tei := tei) # (frmvar_rr.c_r := u2nresp)(frmvar_rr.n_r := v_r) # (frmvar_rr.p_f := not_final)(frmvar_rr.n_r := v_r)		RR_Con(u2nresp, frmvar_rr.sapi, tei, v_r, not_final)		
(ack_pending := false)				
[((NOT(peerrecbz)) AND (v_s <> v_a # + k16_signal)) AND (t200_running)]				
-> 1				
[NOT(((NOT(peerrecbz)) AND (v_s <> # v_a + k16_signal)) AND (t200_running))]			INCONC	
lsap?OTHERWISE			FAIL	
lbcastsap?OTHERWISE			FAIL	
lmsap?OTHERWISE			FAIL	
phsap?OTHERWISE			FAIL	
[NOT(((NOT(peerrecbz)) AND (v_s <> v_a # + k16_signal)) AND (t200_running))]			INCONC	

Figure 24 Subtree generated for transition 122.

In the subtree generated from transition 122 all the references to the variable **frame** which is the name of the parameter passed by **iqdup_i** interaction is replaced by the subtree formal parameter **ACframe**. In this subtree this parameter is assigned to the variable **lastisent_i** as it was specified in the specification. In the constraint reference for **I** frame, we can see the references to the **c_r** and **informtn** fields of **lastisent_i** to which the values **n2ucomand** and **_info** were assigned in the subtree created for transition 121.

7.1.3 Example 3

Our last example of the subtrees generated for testing LAPD protocol is the subtree generated for transition 143. This is an example for a subtree generated for transitions with no output statements. The declaration of transition 143 is:

```
trans
{ 143 }
when ph.ph_dtind_rr
provided (frmrecvd.n_r = v_a) and (frmrecvd.n_r < v_s) and (frm-
recvd.p_f <> final) and (frmrecvd.c_r = n2uresp)
from mulfrmestabd
to mulfrmestabd
begin
   update_buffer(frmrecvd.n_r);
   peerrecbz := false
end;
```

The subtree generated for this transition is shown in figure 25.

TEST STEP DYNAMIC BEHAVIOUR

Reference: lap_d_protocol/lap_d_bod[illegible]ree_143

Identifier: Subtree_143(l3:lcap,l3_bcast:lbcastsap,lm:lmsap,ph:phsap, _RR_sapi : saps, _RR_tei : byte, _RR_p_f : bit)
Objective: Unknown
Defaults Reference:

Behaviour Description	Label	CRef	V	Comments
[v_a < v_s AND _RR_p_f <> final]				
ph!RR		RR_Con(n2nresp , _RR_sapi, _RR_tei, v_a, _RR_p_f)		
START MinRespTimer	1			
?TIMEOUT MinRespTimer			(PASS)	
+update_buffer(v_a)				
(peerrecbz := false)				
ph?RR[ack_pending](frmvar_rr.tei := tei) # (frmvar_rr.c_r := u2nresp)(frmvar_rr.n_r := v_r) # (frmvar_r p_f := not_final)(frmvar_rr n_r := v_r)		RR_Con(u2nresp , frmvar_rr .sapi, tei, v _r, not_final)		
(ack_pending := false)				
[(v_a < v_s) AND (_RR_p_f <> final)]				
-> 1				
[NOT((v_a < v_s) AND (_RR_p_f <> final))]			INCONC	
lsap?OTHERWISE			FAIL	
lbcastsap?OTHERWISE			FAIL	
lmsap?OTHERWISE			FAIL	
phsap?OTHERWISE			FAIL	
[NOT((v_a < v_s) AND (_RR_p_f <> final))]			[illegible]	

Figure 25 Subtree generated for transition 143.

This transition fires when LAPD receives an RR PDU. When this transition is fired no output is sent. From the rule 10.a of the algorithm presented in section 4.1, the tester waits for the amount of MinRespTime to make sure that the IUT does not send any erroneous output. However since this transition starts from **mulfrmestabd** state the transition 263 might still be fired and therefore reception of **RR** PDU is still legal.

7.2 Transport Class 2 Protocol

In this section we give examples of subtrees generated for the specification of simplified class 2 transport protocol (TP2). This protocol is decomposed into two types of modules. The first module type is called AP. Each AP serves one transport user. APs perform the main protocol functions such as exchange of information and flow control. The second module type is called MAP module which multiplexes data from APs into a single channel. The MAP module performs an association between the real addresses of protocols and the AP entities. Figure 26 shows the block diagram of the TP2 module decomposition.

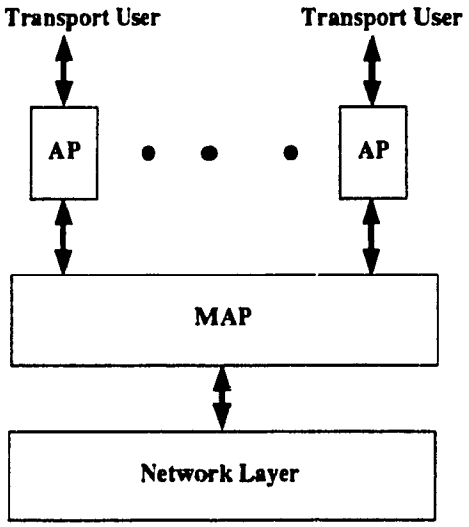


Figure 26 Module decomposition of TP2 protocol.

In TP2 the map module performs the decoding and encoding. Since the parameter of the interactions at external interaction points are encoded they do not correspond exactly to PDU types and our system failed to identify the PDUs. In

order to properly handle PDU constraints a slight modification on the specification was done and PDUs were explicitly defined in the interaction point declarations i.e., the encoding and decoding were made implicit. The tests generated are still correct because in TTCN encoding and decoding are also implicit.

7.2.1 Example 1

The next example shows how ALL statement and nested loops are handled by our program by applying rule (10.f) in section 4.1. The subtree was generated for the initialization part of **ap_body** module of TP2. The initialization part of the protocol is treated like any other transition. Implicitly for the initialization transition, the **from** and the **to** states are the initial state of the module and the initialize transition can not have an input (WHEN clause). The declaration of the initialize transition of this protocol is:

```
initialize
        to idle
        begin
            all t_suf: t_suftp do
                begin
                    all epid: tcepidtp do
                        begin
                            tc[t_suf, epid].assgnd_nc := undef_clsclrbfrs;
                            tc[t_suf, epid].pdu_buf[cr].full := false;
                            tc[t_suf, epid].pdu_buf[cc].full := false;
                            tc[t_suf, epid].pdu_buf[dr].full := false;
                            tc[t_suf, epid].pdu_buf[dc].full := false;
                            tc[t_suf, epid].pdu_buf[dt].full := false;
                            tc[t_suf, epid].pdu_buf[ak].full := false;
                            tc[t_suf, epid].pdu_buf[cr].is_last_pdu := false;
                            tc[t_suf, epid].pdu_buf[cc].is_last_pdu := false;
```

```
                    tc[t_suf, epid].pdu_buf[dr].is_last_pdu := false;
                    tc[t_suf, epid].pdu_buf[dc].is_last_pdu := false;
                    tc[t_suf, epid].pdu_buf[dt].is_last_pdu := false;
                    tc[t_suf, epid].pdu_buf[ak].is_last_pdu := false;
                    tc[t_suf, epid].pdu_buf[dc].is_last_pdu := true
                end
            end
        end;
```

The subtree generated for the initialization part is shown in figure 27. The two constants LOtsuftp and HItsuftp were generated by TTCNgen since the bound t_suftp was not declared in the specification.

7.2.2 Example 2

Our last example is the subtree generated for transition 23 from the MAP module of the TP2. Transition 23 shows the PDU chaining and consideration of the lifetime of PDUs. The declaration of transition 23 is:

```
trans
{ 23 }
any ncid: ncepidtp do
when ns[ncid].ndataind_cr
provided not(exists_tc(ncid, nsdufragm.destref))
from idle
to idle
begin
    nsdufragm.peeraddr := form_taddr(nc[ncid].remotenaddr,
nsdufragm.tsapid_calling);
    t_suf_5 := nsdufragm.tsapid_called;
    assgnnewtcepid(epid_6);
    assgnnewref(tc[t_suf_5, epid_6].localref, nc[ncid].activerefs);
```

TEST STEP DYNAMIC BEHAVIOUR				
Reference: simple_tp/map_body/Subtree_0 **Identifier:** Subtree_0 **Objective:** Unknown **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
Subtree_0				
(t_suf := LOt_suftp)				
(epid := 1)	1			
(tc[t_suf,epid].assgnd_nc := undef_ciscirbfrs)	2			
(tc[t_suf,epid].pdu_buf[cr] full := false)				
(tc[t_suf,epid].pdu_buf[cc].full := false)				
(tc[t_suf,epid].pdu_buf[dr].full := false)				
(tc[t_suf,epid].pdu_buf[dc].full := false)				
(tc[t_suf,epid].pdu_buf[dt].full := false)				
(tc[t_suf,epid].pdu_buf[ak].full := false)				
(tc[t_suf,epid].pdu_buf[cr].is_last_pdu # := false)				
(tc[t_suf,epid].pdu_buf[cc] # .is_last_pdu := false)				
(tc[t_suf,epid].pdu_buf[dr] # .is_last_pdu := false)				
(tc[t_suf,epid].pdu_buf[dc] # .is_last_pdu := false)				
(tc[t_suf,epid] pdu_buf[dt] # .is_last_pdu := false)				
(tc[t_suf,epid].pdu_buf[ak] # .is_last_pdu := false)				
(tc[t_suf,epid] pdu_buf[dc] # .is_last_pdu := true)				
[epid <> maxtcep]				
(epid := epid + 1)				
-> 2				
[epid = maxtcep]				
[t_suf <> Hlt_suftp]				
(t_suf := t_suf + 1)				
-> 1				
[t_suf = Hlt_suftp]				

Figure 27 Example of the code generated for nested 'all do' loops.

```
    tc[t_suf_5, epid_6].remoteref := nsdufragm.srcref;
    tc[t_suf_5, epid_6].assgnd_nc := ncid;
    output ap[t_suf_5, epid_6].transfer_cr(nsdufragm)
end;
```

The declaration of ndataind_cr is:

ndataind_cr(nsdufragm: tpduandctrlinf_cr; lastnsdufragm: boolean);

The first parameter of this interaction is CR PDU and the second parameter is a boolean type. The AP interaction point is internal and therefore a variable called Hpdu is declared. The nsdufragm which is the parameter of input PDU also appears in the internal output statement. Since we can refer to this variable only on the event line (send event) it can not directly be used to assign values to Hpdu. Therefore the program tries to find the value of each field. In this case the value of all the fields of CR PDU are passed to subtree as formal parameter because no equality constraint is mentioned about them. The constraint on the PDUs of this protocol are checked in the body of AP module. The subtree generated for this transition is shown in figure 28.

7.3 Performance

In this section the overall results of applying the test generation tool for LAPD and TP2 Estelle specification are discussed. Some basic statistics about the two specifications are given in table 5. Table 6 shows the time performance of the tool on the two protocols.

From table 6 we can see that the bottleneck of the system for the time performance is the mstourgen program. The reason is that it is based on breadth-first-search and LAPD protocol is composed of three modules and has three internal queues. Therefore the search complexity is very high. Whereas since

TEST STEP DYNAMIC BEHAVIOUR				
Reference: simple_tp/map_body/Subtree_23				
Identifier: Subtree_23(ns:ncepprims, ncid : ncepidtp, _CR_order : ordertp, _CR_peeraddr : taddrtp, _CR_crvl : seqnumtp, _CR_destref : reftp, _CR_srcref : reftp, _CR_user_data : datatp, _CR_opts_ind : optip, _CR_tsapid_calling : t_suftp, _CR_tsapid_called : t_suftp, _lastnsdufragm : boolean, Hpdu : CR)				
Objective: Unknown **Defaults Reference:**				
Behaviour Description	**Label**	**CRef**	**V**	**Comments**
[NOT(exists_tc(ncid, _CR_destref))]				
ns!ndataind_cr		ndataind_cr _Con(CR_Con(_CR_order, _CR _peeraddr, _CR _crvl, _CR _destref, _CR _srcref, _CR _user_data, _CR_opts_ind, _CR_tsapid _calling, _CR _tsapid_called), _lastnsdufragm)		
(nsdufragm.peeraddr := form_taddr(nc[ncid] # .remotenaddr, _CR_tsapid_calling))				
(t_suf_5 := _CR_tsapid_called)				
+assgnnewtcepid'epid_6)				
+assgnnewref(tc[t_suf_5,epid_6] localref, # nc[ncid].activerefs)				
(tc[t_suf_5,epid_6].remoteref := _CR_srcref)				
(tc[t_suf_5,epid_6].assgnd_nc := ncid)				
(Hpdu.order := _CR_order, # Hpdu.peeraddr := _CR_peeraddr, # Hpdu.crvl := _CR_crvl, Hpdu # .destref := _CR_destref, Hpdu # .srcref := _CR_srcref, Hpdu # .user_data := _CR_user_data, Hpdu # .opts_ind := _CR_opts_ind, Hpdu # .tsapid_calling := # _CR_tsapid_calling, Hpdu # .tsapid_called := _CR_tsapid_called) #				
[exists_tc(ncid, _CR_destref)]			INCONC	

Figure 28 Subtree generated for transition 23.

TP2 has only one internal queue and two modules with much less number of transitions the program converges very fast.

Specification	Original number of transitions	Number of transitions after normalization	Number of transitons after simplification	No of Data flow functions	Number of test cases
LAPD	135	410	490	19	1094
TP2	23	67	69	11	53

Table 5 Results of the application of the test generation tool on LAPD and TP2 protocols

Specification	mappdu	transform	simplify	ttcngen	mstourgen	testcase	Total
LAPD	0.6 sec	18.3 sec	17.5 sec	48.3 sec	9334 sec	36.0 sec	9454.7 sec
TP2	0.3 sec	2.5 sec		19.8 sec	14 sec	1.6 sec	38.2 sec

Table 6 Performance of the test generation tool for LAPD and TP2

Chapter 8 CONCLUSIONS

8.1 Summary

The derivation of conformance tests for protocols is a cumbersome task. Most of this procedure can be automated. We described a technique for directly deriving test suites from Estelle specifications. We first transform the original specification to another simpler Estelle specification in which the enabling predicates of the transitions are converted to a disjunctive normal form. After simplification the test sequences are generated in a semi-automatic manner. Finally these sequences are parameterized and converted to TTCN notation automatically. This is done by mapping different constructs of Estelle specification to their corresponding constructs in TTCN. The advantages of our technique include ease of translation due to the use of normalized specification and ease of manipulation of the resulting TTCN tables due to the use of an interactive editor for TTCN.

There are some incompatibilities between TTCN and Estelle that have not been resolved yet and may yield to incorrect results. One is the way Estelle modules can be structured. Each Estelle module can be refined to submodules. In Estelle the user can define array of modules and they will have their corresponding array of timers, etc. This particular structuring i.e., having array of interaction points and timers can not be specified in TTCN. Currently the tests produced only test one instance of such modules.

The input specification must declare the PDUs to be a parameter of interaction at the external interactions points. Only then CONTEST-ESTL is able to successfully identify the PDUs. TTCNgen's PDU constraint extraction is based on this PDU identification process. PDU identification is difficult for protocols where concatenation is allowed, i.e., several PDUs in a single ASP, and/or multiplexing where several (N)-layer connections are mapped into a single connection. More research is needed to get rid of this restriction.

As it was seen in the case of transport protocol, since currently TTCN requires the PDUs to be identified on the event line, the input specification must also declare the PDU parameters in the interaction point declaration. This way the encoding and decoding are assumed to be implicit.

8.2 Suggestion for Further research

More research is needed in direction of improving test generation algorithm (mstourgen) by including data flow in the test case generation system. The present system only considers the control flow of the specification. This improvement has two advantages. The first advantage of this improvement is the non-determinism (taking different actions for the same input) in the FSM model will be eliminated because the inputs are differentiated. The second advantage is that the infeasible paths will be avoided by the software.

For higher layer protocols such as application and presentation layers adding ASN.1 constraint and declaration will greatly enhance the system. This can be done by mixing the standard definitions of PDUs in ASN.1 and Estelle language

and having a mechanism of mapping Estelle type definitions to ASN.1 type definitions.

We are currently generating the tests in local test architecture. In section 4.3 the conversion from local to distributed test architecture was discussed. More research is needed for converting LS to DS, CS and RS test architectures.

The existing software can be improved by probably attempting to generate the test cases in a tree form where more than one transition has the possibility of being fired. The existing test cases or test sequences are linear and inside the subtrees only the spontaneous transitions are taken as alternatives. Also currently the test sequences generated contain many infeasible paths and must be carefully be inspected by the user. This can be avoided by considering the data flow as well as the control flow and simulation of protocol during test case generation.

REFERENCES

1. William Stallings , "Data and Computer Communications", New York, NY, Macmillan Publishing Company, 1988.
2. International Organization for Standardization, "Basic Refence Model for Open Systems Interconnection", ISO 7498, 1984.
3. Liba Svobodova, "Implementing OSI systems", IEEE Journal on Selected Areas in Communications, VOL. 7, No. 7, September 1989, pp. 1115-1129.
4. ISO/TC 97/SC 21, "Estelle: A Formal Description Technique Based on an Extended State Transition Model", DIS 9074, 1987.
5. ISO/IEC JTC 1/ SC 21, "Methodology and Framework of Conformance Testing: Parts 1 & 2", ISO DIS 9646, 1988.
6. ISO/IEC JTC1/SC21/WG1 Florence Meeting, "Working draft Addendum on Extensions to TTCN, including Parallel Trees", DIS 9646 addenda topic, November 1989.
7. G. Dicenet, "Design and prospects for the ISDN", Norwood, MA, ARTECH HOUSE Inc., 1987.
8. CCITT, COM XI-R 43-E, "Report on the Meeting Held in Geneva from 3 to 14 November 1986 [Part C.6.1 - Recommendations Q.920 and Q.921]", December 1986.
9. B. Sarikaya, G.v. Bochmann, "Some Experience with Test Sequence Generation for Protocols", Proc. 2nd. Int. Workshop on Protocol Specification, Testing and Verification, 1982.

10. K. K. Sabnani, A. Dahbura, "A Protocol Test Generation Procedure", Computer Networks 15, 1988, pp. 285–297.

11. B. Kanunga, et. al., "A Useful FSM Representation for Test Suite Design and Development", Proc. of 6th Int. Workshop on Protocol Specification, Testing and Verification, 1986.

12. J-P. Favreau, R.J. Linn, "Automatic Generation of Test Scenario Skeletons from Protocol Specifications Written in Estelle", Proc. of 6th Int. Workshop on Protocol Specification, Testing and Verification, 1986.

13. Y. Tscha, Y. Choi, "A New Approach to Generate Test Sequences from Protocol Specifications Written in Estelle", Proc. of Globecom 87, 1987.

14. B. Sarikaya, G.v. Bochmann, E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Trans. on Soft. Eng., May 1987.

15. B. Sarikaya, S. Eswara, V. Koukoulidis, M. Barbeau, "A Formal Specification Based Test Generation Tool", Technical Report, Concordia University, 1988.

16. B. Forghani, S. Eswara, V. Koukoulidis, B. Sarikaya, "Estelle Based Test Generation Tool for Modular Specifications", Proc. FORTE89, Vancouver, Dec. 4-7, 1989.

17. V. Koukoulidis, "Full Implementation of a Test Design Methodology for Protocol Testing", M.Sc. Thesis, Concordia Univ., March 1989.

18. B. Sarikaya, "Test Design for Computer Network Protocols", Ph. D. Thesis, McGill University, March 1984.

19. B. Sarikaya, G.v. Bochmann, E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Trans. on Soft. Engr, May 1987, pp. 531-540.

20. B. Sarikaya, G. v. Bochmann, "Synchronization and Specification Issues in Protocol Testing", IEEE Trans. on Communications, April 1984, pp. 389-395.

21. Z. Kohavi, "Switching and Finite Automata Theory", New York NY, Macmillan Publishing Company, 1978.

22. T. S. Chow, "Testing Software Design Modeled by Finite State Machine", IEEE Trans. on Computers, vol. C-19, no. 6, June 1970.

23. Son T. Vuong, W. L. Chan, M. R. Ito, "The UIOv Method for Protocol Test Sequence Generation", 2nd International Workshop on Protocol Test Systems, Berlin (West), Germany, October 1989.

24. A. T. Shreiner, H. G. Friedman, Jr., Introduction to Compiler Construction with UNIX", Englewood Cliffs, NJ, Prentice Hall, 1985.

25. ISO/TC 97/SC 21, "Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)", IS 8824, 1989.

26. B. Kerninghan and D. Ritchie, "The C Programming Language", Englewood Cliffs, NJ, Prentice Hall, 1978.

27. B. Kernighan and R.Pike, "The UNIX Programming Environment", Prentice Hall, 1984.

28. S. Eswara, B. Sarikaya, "CONTEST-TTCN: An Editor for ISO's Protocol Test Notation", Proc. Canadian Electrical and Computer Engineering Conference, Vancouver, Nov. 1988.

29. David A. Lamb, "Software Engineering: Planning for Change", Prentice Hall, 1988.

30. R. Nigel Horspool, "C Programming in the Berkeley UNIX Environment",

Prentice Hall, 1984.

31. G.V.Bochmann, "Specification of a Simplified Transport Protocol Using Different Formal Description Techniques", Research Report, University of Montreal, Publication 623, April 1987.

GLOSSARY

Transition tour:

A sequence of transitions which covers all the transitions of a finite state machine.

Subtour:

A sequence of transitions which start from initial state and goes back to the initial state of the finite state machine.

Test suites:

A set of tests which verify the conformance of an implementation to the standard. Test suites are further refined into test groups and test cases.

Test cases:

Test trees which achieve a specific test purpose. Test cases must be independent of each other and the tester must be able to run them in any order.

Test steps:

A test tree which has a name and can have formal parameters similar to procedures in conventional programming languages. Test steps are grouped in a library and can be called(attached) from everywhere.

Interaction points:

An interface over which Estelle modules communicate with the environment and other modules. If the interaction point connects two modules of the specification together and is not visible from outside then it is called an **internal interaction point**. If the interaction point is used for communication with the environment it is called an **external interaction point**.

APPENDIX A

USER'S GUIDE

A.1 Introduction

This user's guide explains the commands that must be given by the user to obtain the test suite from the Estelle specifications of a program. To follow this guide you must be familiar with the methodology presented in this thesis.

A.2 Commands

The commands described in this section must be issued directly from the UNIX shell. Currently there is no user interface for the whole system. The reason is partly because of the simplicity of the commands.

A.2.1 nf (Normal Form) Command

This command activates the normalization module. Normalization is the first step of generating tests. If you have described the PDU as a PASCAL variant record you should also request for the first phase of data flow analysis which identifies the different records which define the PDUs. The syntax of this command is:

nf Estelle-specification

where Estelle-specification is the name of the file containing the Estelle specification of the protocol. This command only activates the normalization. To request for the first phase of data flow analysis you must type:

nf Estelle-specification record-name

where record-name is the name of the variant record describing the PDUs. The name of the file where the output of this program is written to is made of the name of the original program and a .LIST suffix (Estelle-specification.LIST).

A.2.2 simplify Command

This command invokes the simplification module. This command is really implemented in two different programs. The programs are called **simplify** and **transform.**

Usually you only need to issue **simplify** command. **simplify** creates a UNIX **pipe** to **transform.** The **transform** program simplifies the predicates inside the provided clause of the transitions as explained in section 3.2 and sends them to **simplify.** Then **simplify** substitutes the new predicates in the provided clause of the transitions.

The advantage of implementing the provided clause simplification phase as two separate programs is that the user can check the predicates generated by **transform** program before the transitions are generated. The simplification process is usually invoked by typing

simplify SpecFile.

In order to check the predicates before the transitions are generated the user must first type:

transform SpecFile > TempFile.

Now TempFile contains the transformed predicates. The predicates of different transitions are separated by an empty line. The user can then edit this file and generate the simplified specification by typing

simplify SpecFile TempFile.

Modification/deletion of predicates can be done by editing the output of the simplification program. However the first solution is preferred because the transition numbers are indicated as a PASCAL comment inside the transition declaration. Deleting a transition from the middle by hand causes these numbers to be different from the real transition number. Also by editing the TempFile the user can concentrate on the predicates. If due to deleting or adding a transition by hand the numbering of the transitions become inconsistent the transitions can be renumbered by running the simplification program again.

If the name of input file has .LIST suffix, which is usually the case since the input of the program is generated by nf command, the name of output file is obtained by substituting the .LIST suffix of the SpecFile by .cnf.LIST suffix. Otherwise the name fo the output file is derived by adding .cnf suffix to SpecFile.

A.2.3 mappdu Command

This command is used to create a table mapping PASCAL records to PDU types explained in chapter 4. The syntax of this command is:

mappdu input-file PDU-name

Input-file must contain the original Estelle specification of the protocol and PDU-name is the name of the record which defines the PDU as a PASCAL variant record. The program assumes that the case constant used in the definition of the variant record is the actual PDU name. If this is not the case the user can modify the name and supply the correct PDU name to be generated in TTCN.

The name of the output file generated by this program is derived from the input file by appending .map suffix to it.

A.2.4 dtf (Data and Control Flow) Command

The dtf command extracts the data and control flow information of the protocol. For each module in the specification a file containing the data flow information to be used by dfgtool program and a file containing the control flow information to be used by mstourgen program are created.

The syntax of this command is:

dtf Input-file

Input-file must contain the normalized and simplified Estelle specification of the protocol. This file is created by **simplify** program. The name of the output file is derived from the name of the modules in the specification by appending .DTF suffix for the file containing data flow information and .CTRL suffix for the file containing control flow information to the module name.

A.2.5 ttcngen Command

This command invokes the **ttcngen** program explained in chapters 4 and 5.

The syntax of this command is:

ttcngen Spec-file [-m Map-table] [-c Connection-file]

Spec-file must be the output of the **simplify** program.

The first optional arguments are -m Map-table tells **ttcngen** that Map-file contains the table mapping PDUs to PASCAL types probably generated by **mappdu** program.

The second option -c Connection-file indicates that Connection-file contains description of the queues connected together. Each line of the Connection-file must have the format:

Module-1:ip-1 Module-2:ip-2

The name of the test suite is the same as the name of the Estelle specification indicated by the Estelle keyword: **SPECIFICATION**. The files generated by this module are in intermediary forms required by TTCN editor described in [Eswa 89].

A.2.6 dfgtool Command

This command invokes the dfgtool program. The syntax of this command is:

dfgtool Input-file

Input-file must contain the data flow information of an Estelle module and is

generated by issuing **dtf** command. The name of the output file is derived from the input file name by appending .dat suffix to it.

A.2.7 mstourgen Command

This command invokes the **mstourgen** program. The syntax of the command is:

mstourgen f1...fn [-c cf1 ... -c cfn] [-e ef1 ... -e efn] [-s -ns -llength] output-file

f1 through **fn** must contain the control flow information of different modules of the specification and are generated by dtf command. **cf1** through **cfn** specify the internal connections and the associated queues between the modules and is usually the same file which is used for **ttcngen** command. Each **cf1** through **cfn** file names must be preceded with a -c switch.

ef1 through **efn** contain the description of the interaction points that are controlled by the same tester. The format of this file is the same as the Connection-file described in A.2.5. Although this format only lets you specify two interaction points to be equivalent in each line, due to the transitive property any number of interaction points can specified to be controlled by one tester by repeating one interaction point in more than one entry. Each **ef1** through **efn** file names must be preceded with a -e switch.

The -s switch indicates that short subtours must be generated and -ns switch causes the software to generate non-synchronizable tests.

Invoking the program by '-ns' switch tells the program not to check synchronization. The -l switch supplies a value which overrides the default maximum depth of search to cover a new transition before the program gives up.

The name of the output file is specified as the last argument of the command.

A.2.8 testgen Command

This command merges the transition tour obtained from issuing the **mstourgen** command and the result of the data flow analysis obtained from **dfgtool** command. The syntax of this command is simply:

testgen

After issuing this command a menu comes up and then you are given the three choices of: text-output, number-output and invoking edittour program. In order to generate test suite you must chose the text-output option. Once you choose this option, you will be prompted for the name of two input files explained in the previous paragraph. The name of the output file is derived from the name of the file containing the result of data flow information by appending .ttour suffix to it.

A.2.9 edittour Command

This command lets you to modify the output of the testgen command by showing the subtours generated by testgen vis-a-vis the Estelle specification of the protocol. You can optionally issue this command inside the testgen program as explained in the previous section.

The syntax of this command is:

edittour Subtour-file Spec-file

Subtour-file is the output of testgen and Spec-file is the normalized and simplified Estelle specification. The output is written back into Subtour-file.

A.2.10 testcase Command

This command generates the test cases from the subtour file. The syntax of this command is:

testcase Subtour-file [Suite-name]

Subtour-file is the output of **testgen** command which might have been modified by **edittour**. Suite-name is the name of test suite. If no Suite-name is specified the name of test suite is derived from Subtour-file by removing all the suffixes. The files generated by this module are in intermediary forms required by TTCN editor described in A.2.5.

A.3 Exceptions

This section describes the exceptions messages that the different modules of test generation tool print.

A.3.1 Compilation Exceptions

These exceptions apply to nf, simplify, dtf and ttcngen commands. These exceptions are caused by syntax or semantic errors in the specifications. In this case an error or a warning message is given to the user indicating the line number

and type of error. If an error message is issued to the user the code generation is stopped.

A.3.2 Input File Exceptions

These exceptions apply to all the commands. If a file does not exist a proper error message indicating the name of the file that couldn't be found is issued and the program stops.

A.3.3 Bad File Format Exceptions

These exceptions also apply to all the commands. The compiler based commands(nf, simplify, dtf and ttcngen) give syntax error messages. The rest of programs give the message:
Improper input file: name of the file
error message is displayed and the execution is aborted.

A.3.3 Capacity Exceptions

To avoid having capacity exceptions no limit was put on the size of specification, number of files, etc. Wherever the size of input was not known e.g., parameters of input, linked list were used instead of arrays. Therefore the only capacity exceptions are the size of memory of the system and the disk space. If both cases an appropriate error message is displayed and the execution of the programs are aborted.

References

[Eswa 89] S. Eswara, "An Editor for ISO's Protocol Test Notation for Test Suite Management", M. Eng. Thesis, Concordia University, Jan. 1989.