ISSUES ON DESIGN AND IMPLEMENTATION OF PROTOCOL TEST SYSTEMS

By

Bernard P. Lee

B. Sc. (Computer Science) University of Washington

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

COMPUTER SCIENCE

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

September 1989

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date 11 Oct. 1989

DE-6 (2/88)

Abstract

The advant of *Open Systems Interconnection* (OSI) accentuates the importance of *conformance testing* of protocol implementations. Before a protocol implementation is delivered for conformance testing, it often has to go through several iterations of *diagnostic testing.*

This thesis discusses various issues that arise in the design and implementation of protocol test systems for conformance and diagnostic testing.

Issues discussed in this thesis include test system implementation decisions, test management, test development environment and tools, test languages, encoding and decoding, PDU specification and storage, and portability problems.

The experience of testing different protocol implementations on three different environments are also discussed.

Abbreviations

AFC	Active Ferry Clip
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
ATS	Abstract Test Suite
COS	Corporation for Open Systems
CPU	Central Processing Unit
E/D	Encoder / Decoder
ETS	Executable Test Suite
FCP	Ferry Control Protocol
FCTS	Ferry Clip based Test System
FSM	Finite State Machine
FTMP	Ferry Transfer Medium Protocol
INET	Internet
IPC	Interprocess Communication
ISO	International Standards Organization
ITL	IDACOM Test Language
IUT	Implementation Under Test
LMAP	Lower Mapping Module
LT	Lower Tester
MPT	Multi-port Protocol Tester
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PFC	Passive Ferry Clip
PT	Protocol Tester
PTE	Protocol Testing Environment
SAP	Service Access Point

SUT	System Under Test
TM	Test Manager
TP/0	Transport Layer class zero
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester

Table of Contents

List of Figures

Acknowledgement

First, I would like to extend a very special thanks to my supervisor Dr. Samuel T. Chanson. Throughout my research, he has always been there for discussion and advice. At times of difficulties, his encouragement and support contribute greatly to the completion of my thesis.

A special thanks to my partner and close friend Neville J. Parakh for his support and determination to see our project through to completion.

Thanks to Dr. H. X. Zeng for his advice and finding the time for being the second reader of my thesis.

Thanks to Dr. D. Rayner for his advice and encouragement. Thanks to B. Smith, I. Chan, H. See, S. Chan, V. Lee and C. Anderson for always finding the time to help me. Thanks to the technical staff in the Department of Computer Science, especially P. Phillips and M. Lau for all the help they gave us in setting up our system.

Thanks to the Department of Computer Science at the University of British Columbia, the Natural Sciences and Engineering Research Council (NSERC) and IDACOM Electronics for providing me with facilities and financial support during my research.

Last but not the least, a very special thanks to my parents. Even though far away from Vancouver, they have consistently provided me with the most thorough support and encouragement without which none of this would have been possible.

Chapter 1

Introduction

This chapter describes the motivation of our research. A brief description of the current work on protocol test systems at the University of British Columbia is given, followed by a layout of the rest of the thesis.

1.1 Communication and Protocols

Computers nowadays are often connected into a network to access shared hardware resources such as tape drives, printers and disks, as well as software resources such as files and databases. As well, data have to be moved from computers to computers to meet the needs of an information hungry society. Thus, modern computers must be able to communicate with other computers reliably and efficiently. In order to achieve this goal, *communication protocols* were developed and put into work.

1.1.1 The OSI Reference Model

A communication protocol is a set of rules or conventions by which two separate entities communicate with one another. In order to be able to handle all possible errors as well as different communication media, communication protocols have become too complex to be implemented in a single module. The *International Standards Organization* (ISO) has developed a model for structuring communication systems into seven distinct layers (Figure 1.1), called the The *Open Systems Interconnection* (OSI) Reference Model [7]. Each of the seven *protocol layers* performs a well-defined subset of the seven-layer *protocol*

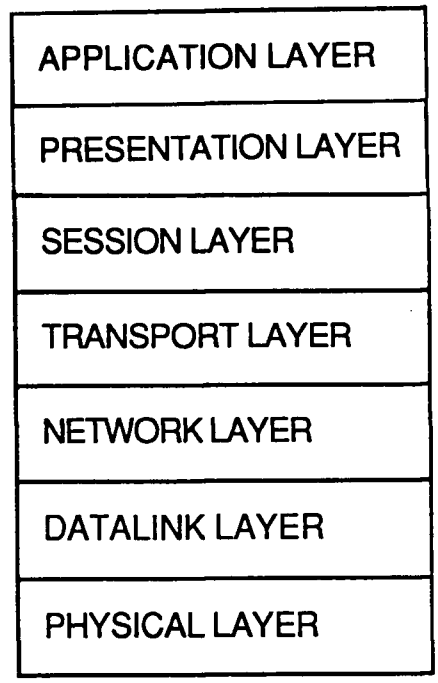

Figure 1.1: OSI Reference Model

stack. Each layer uses the services from its *service provider*, which is the layer immediately beneath it, and in turn provides service to its *service user*, which is the layer immediately above.

By clearly defining the services provided by each layer, the OSI reference model thus allows different protocol implementations to interwork together.

1.1.2 Protocol Services and Access

The OSI reference model states that *'Only the external behavior of Open Systems is retained as the standard of behavior of real Open Systems'*. Therefore, each protocol entity is only accessible through *service access points* (SAPs), namely the *upper SAPs* which interface with the layer immediately above, and the *lower SAPs* which interface

with the layer immediately below.

Services are defined by a set of *abstract service primitives* (ASPs). Each ASP is associated with a list of parameters, which constitute a complete description of a service provided.

Protocol layers use the data transfer services of the provider to transfer *protocol data units* (PDUs). PDUs carry all the information necessary for one protocol layer to communicate with its remote peer entity.

1.2 Conformance and Diagnostic Testing

In order for OSI to work, protocol implementations must be thoroughly tested to see if they conform to the specifications as defined by the standards to which they purport to adhere. The purpose of conformance testing is to increase the probability that different protocol implementations can interwork together. *Conformance testing* is usually done in test centers for certification purposes.

Before a protocol implementation is delivered for conformance testing, it often has to go through several iterations of *diagnostic testing*. Diagnostic testing is usually performed by the vendor at different development stages and often involves the use of protocol testing tools. Since diagnostic testing is performed by the vendor, all the available SAPs may be used, including those the vendor does not wish to expose to the outside world. Hence, diagnostic testing allows a higher degree of control and observation of the *implementation under test* (IUT) [1] as opposed to conformance testing.

[1] The Implementation Under Test is 'an implementation of one or more OSI protocols in an adjacent user/provider relationship being that part of a real open system which is to be studied by testing'.

1.3 Protocol Test Systems

Protocol test systems are specialized hardware and software tools designed to perform conformance and diagnostic testing of protocol implementations.

OSI has defined methods for protocol testing [6], namely the *remote test method*, the *distributed test method*, the *coordinated test method*, and the *local test method.*

The local test method, though the most powerful, has not been used much because it is impractical to implement. In order to do local testing, all test system software has to reside in the *system under test* (SUT) [2]. This means that the test system has to be developed on the same system that runs the IUT and must be completely rewritten for each different SUT.

Recently, the Ferry Clip [1] [2] [4] method was introduced to realize all the ISO test methods, including the local test method (see section 2.4 for a brief description). By using the Ferry Clip, the test system no longer has to reside in the SUT in order to do local testing. Thus, the local test method is more appropriately termed the *direct test method*, since the testing software is no longer "local" to the IUT.

As the direct test method is the most powerful test method available, and it is made feasible by the Ferry Clip, our research has been on design and implementation issues of test systems for direct testing. Such test systems can easily be adapted to do remote, distributed and coordinated testing.

1.4 Goals of the Thesis

The purpose of this research is to study the design and implementation issues of protocol test systems, especially how different requirements and constraints affect the design of these systems. Issues arising from doing actual conformance and diagnostic testing on

[2]The system under test is 'the real open system in which the IUT resides'.

different protocol implementations under different environments will also be discussed.

One of our main goals is to study the portability issues involved in building a test system on various hardware and software architectures. The test system should be structured such that when a new IUT is tested, the code that needs to be rewritten is minimized. By using the same test system for different IUTs, we also studied how we could minimize the porting efforts when testing IUTs implemented on different software environments.

As of this date, we have tested implementations on three different environments - Unix, MPT [12] and OSI-PTE [15]. Unix is a popular operating system which runs on many different mainframes and workstations; MPT is a general-purpose protocol tester manufactured by IDACOM Electronics; and the OSI-PTE is a sophisticated test system currently being developed at the University of British Columbia.

1.5 Layout of the Thesis

Following the introduction, Chapter 2 describes the different software and hardware architectures being considered, and the constraints and requirements involved when implementing the test software on these different architectures. Chapter 3 discusses test management and test language issues when building a test system. Chapter 4 investigates schemes for PDU encoding, decoding, storage and representation. Chapter 5 describes portability issues encountered while tailoring a *Ferry Clip based Test System* (FCTS) to test different IUTs in different SUTs, and chapter 6 concludes the thesis.

1.6 Relations to the Ferry Clip Project

Although much of the testing was done with the Ferry Clip based Test System (FCTS) developed at the University of British Columbia, most of the materials in this thesis are

not restricted to Ferry Clip applications alone.

Chapter 5, however, is dedicated to the study of portability issues encountered when using the FCTS on different protocol implementations under different SUTs.

The FCTS used in this thesis is developed jointly with Neville J. Parakh of the University of British Columbia. Details on the conceptual design and implementation of the FCTS can be found in his M.Sc. thesis [17].

Chapter 2

Environments

During the course of the research, several test software prototypes were built. Three programming environments, namely Unix, MPT and OSI-PTE are used for our test system prototype implementations. A discussion of these environments, together with a brief description of the Ferry Clip approach, is given in this chapter.

2.1 The Unix Environment

Unix is probably the most widely networked operating system to date. Originally developed at AT&T as an operating system for interactive programming, it soon became one of the most used operating systems for minicomputers, workstations, and now PCs. We shall assume that the reader of this thesis is relatively familiar with the Unix operating system, and will not go into details.

2.1.1 Unix as a Test System Environment

The biggest advantage of implementing a test system in Unix is the availability of programming, debugging and text processing tools. We were able to produce code relatively quickly, and symbolic debuggers have cut our development time significantly.

Text processing tools in Unix played a particularly important role as an aid to reformat existing test suites. We found the macro facility and the ability to perform global search and replace operations using complex regular expressions in the vi and ex editors indispensable.

Lex and yacc are also extensively used to develop some of our prototypes, especially those which requires a formalized input syntax.

2.1.2 Prototypes built under Unix

A prototype test suite development tool has been built under Unix and is discussed in Chapter 3. A PDU Library module has also been prototyped in Unix. The motivation and design issues of the PDU Library is described in Chapter 4.

The OSI-PTE, which will be discussed later in this chapter, was also developed in Unix.

2.2 The IDACOM MPT Environment

The MPT368.2 [12] is a portable *protocol tester* (PT) manufactured by IDACOM Electronics. It runs a proprietary operating system with a built-in Forth interpreter. It has three Motorola 68000 CPUs, expandable to six. Memory is partitioned between the CPUs, but one CPU can access another CPU's memory partition. The CPUs communicate through shared memory and inter-CPU messages (See Figure 2.2).

2.2.1 Processors and Interfaces

One of the three CPUs is the main CPU, and has no dedicated test ports. The other CPUs are used as test CPUs, and each of them has its own set of interface adapters at the back.

Interface types supported include V.24 (RS-232C), V.11, V.35 and V.36, and interface configuration parameters can be tailored to suite most existing protocols.

In addition , an auxiliary serial port is available for file transfer and remote tester access. A printer port comes handy for test logging.

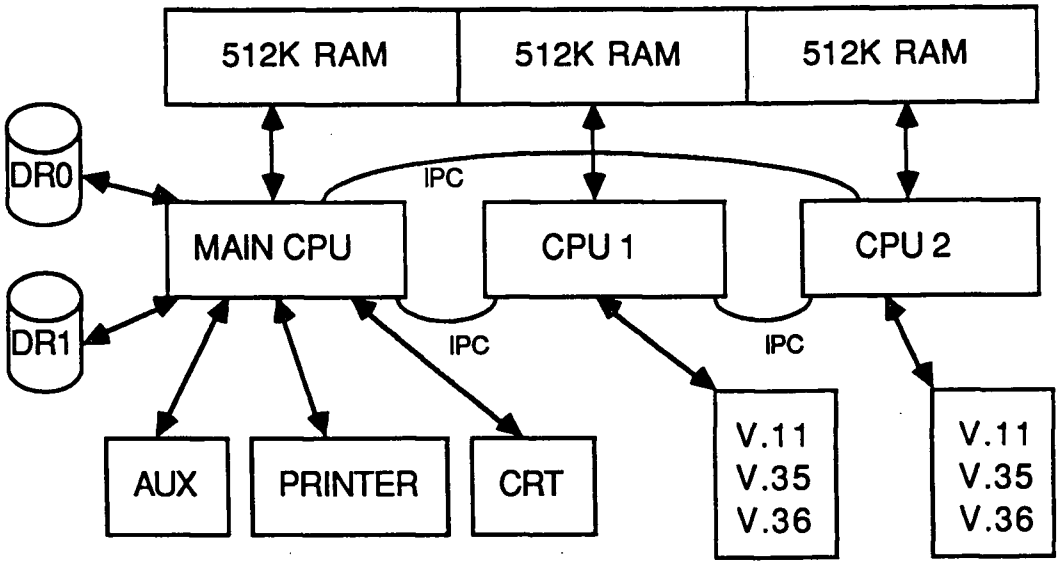

Figure 2.2: MPT Structural Block Diagram

2.2.2 Operating System and Programming Environment

The PT's operating system is event driven. Events are triggered by an incoming frame, a keyboard entry, a timer expiration or an inter-CPU message. Normally, the system handles all the events, but the user could write handlers to takeover selected events.

All CPUs are single threaded, but handlers can be triggered even if a program is running on a CPU. Thus, simultaneous tasks could be easily handled. This is a useful feature for multi-party testing, since more event handlers can be added to the tester as the number of connections required increases.

Currently, all programming on the MPT are done on an extension of FIG-Forth[1]. All system and user commands are also defined as Forth words. Eventually, this operating system will be replaced by a C and window based operating system, and the MPT will run a version of the OSI-PTE described later in this chapter.

2.2.3 Prototypes built under MPT

A full blown FCTS was developed under the MPT environment. The MPT environment was selected for our primary FCTS implementation.

The FCTS / MPT was used for studying the portability issues encountered when performing conformance and diagnostic testing on different environments. This will be discussed in Chapter 5.

The MPT is used commercially to perform conformance and diagnostic testing on protocol implementations. By building our primary test system on the MPT, we believe the issues that arise will be similar to that encountered in real life protocol testing.

[1]FIG-Forth is a version of Forth whose standard is defined by the Forth Interest Group.

2.3 The OSI-PT Environment

The OSI-PTE [15] [16] is a new environment for the implementation and testing of computer communications protocols. Currently it runs on the Unix operating system. Eventually it will be ported to the MPT.

The OSI-PTE is an event driven system. Each protocol is structured as a single or a group of Protocol entities (see Figure 2.3). Communication between protocol entities is performed through an event-posting scheme whereby one protocol entity posts an event to another protocol entity.

The important events are ASP Up events to the service user, ASP Down events to services provider, and timer expiry events from the system. Other events include PDU Up and PDU Down events which are used internally by each protocol entity state machine, and State Out and Error Out events which are 'probes' used for conformance and diagnostic testing purposes.

2.3.1 Protocol Entities

Each protocol entity in the OSI-PTE is uniquely identified by an Entity Identifier (NID). Entities must know the NIDs for both its service user and service provider in order to post events to them.

Request of service from the service provider is done by posting an ASP Down event on the service provider NID. Service data units are passed to the service user by posting an ASP Up event on the service user NID.

Each ASP for a protocol entity is identified by an Event Identifier (EID). Associated with each EID is an Event Parameter Area (EPA), a structure that contains all the parameters to the ASP.

Multiplexing and demultiplexing of services in each protocol entity are done through

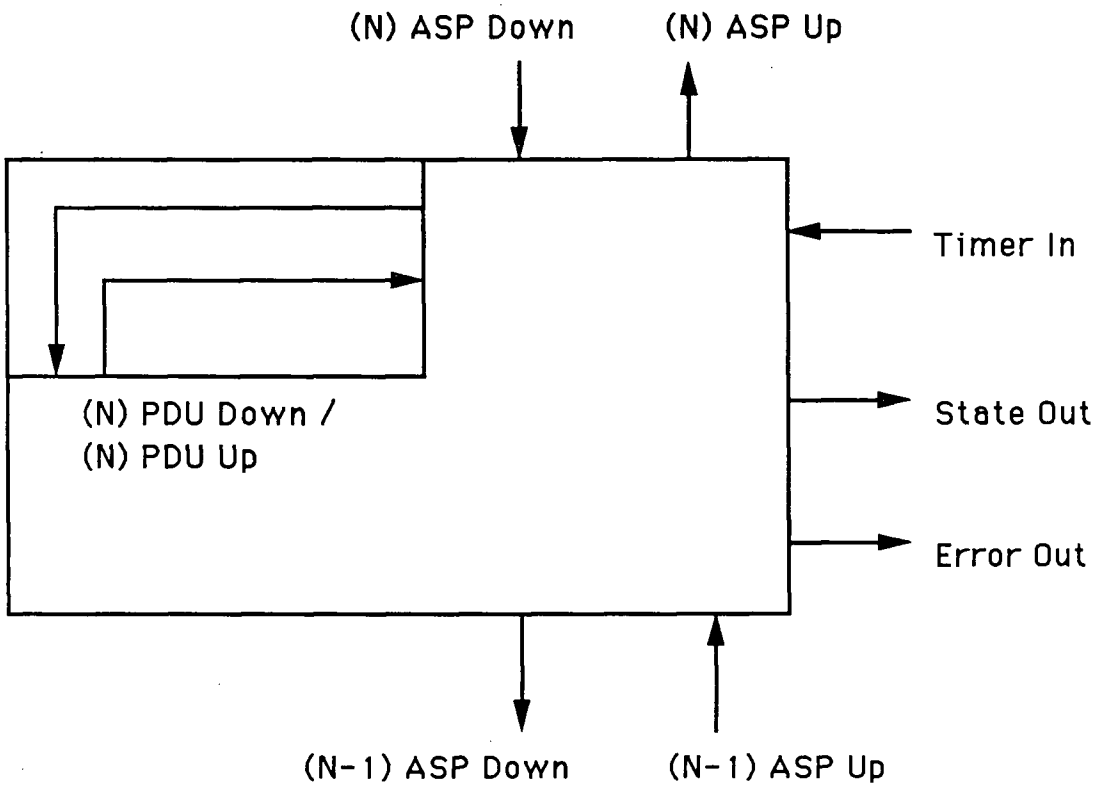

Figure 2.3: An OSI-PTE Entity

Connection Control Blocks (CCB). Each time a new connection is made to the protocol entity, a Connection Identifier (CID) is assigned and returned. Service users can then specify which connection to use by specifying the CID when posting.

Thus event posting is done in the form *PostEvent(NID, CID, EID, EPA)*, which specifies the entity, connection, event and parameters respectively. Overall, the system resembles the OSI Reference Model much more closely than the other two environments discussed.

2.3.2 Dispatching in the OSI-PTE

Since all entities are event driven, some mechanism is needed to arbitrate all incoming and outgoing data and turn them to events. We call this mechanism the dispatcher.

The heart of a PTE implementation is the dispatcher. All incoming and outgoing frames, as wells as timer expiries are handled by the dispatcher. The dispatcher normally has complete control of the environment. When an incoming frame or a timer expiry occurs, the dispatcher posts an event to the appropriate entity, thus passing control to it.

By posting an event, an entity gives control to another entity. After the event is processed by each of the related entities through the chain of event posting, control is returned to the dispatcher, which could then clear the outgoing data and wait for another event. More details of the PTE environment can be found in [15] and [16].

2.4 The Ferry Clip

A brief description of the Ferry Clip is given here as background information to the materials discussed in Chapter 5.

The Ferry Clip concept is a generalization of the Ferry concept as defined by Zeng

[1]. The main idea of the Ferry Clip is to transport test data transparently from the *system under test* (SUT) to the test system, such that the test system can access either or both of the SAPs of the remote IUT as if they were local. The part of the test system software which originally resides in the SUT can thus be moved back to the test system, minimizing the amount of software which needs to be rewritten for each IUT. For other advantages of the Ferry approach see [4] and also section 2.4.3.

2.4.1 Structure of a FCTS

A Ferry Clip based Test System (FCTS) consists of two major components, an Active Ferry Clip (AFC) which resides in the test system, and a Passive Ferry Clip (PFC) which resides in the SUT (see Figure 2.4).

The two ferries runs the *Ferry Control Protocol* (FCP) in order to transfer test data between the *Test Manager* (TM) in the test system and an external IUT residing in the SUT. The FCP utilizes the data transfer service of some existing reliable protocol such as X.25 or TCP/IP. The protocol which provides the data transfer service to the FCP is known as the *ferry transfer medium protocol* (FTMP).

2.4.2 Realization of Abstract Test methods

The Ferry Clip approach can be used to realize the abstract test methods defined by ISO. The direct test method is realized by attaching the passive ferry clip to both SAPs of the IUT, allowing the test system to directly observe and control data in and out of the SUT. The remote test method can be realized by attaching just the lower arm of the passive ferry clip. The distributed and coordinated test methods can be realized by using the upper arm of the passive ferry clip while the (N-1) protocol stacks are used to transfer data from the test system to the IUT.

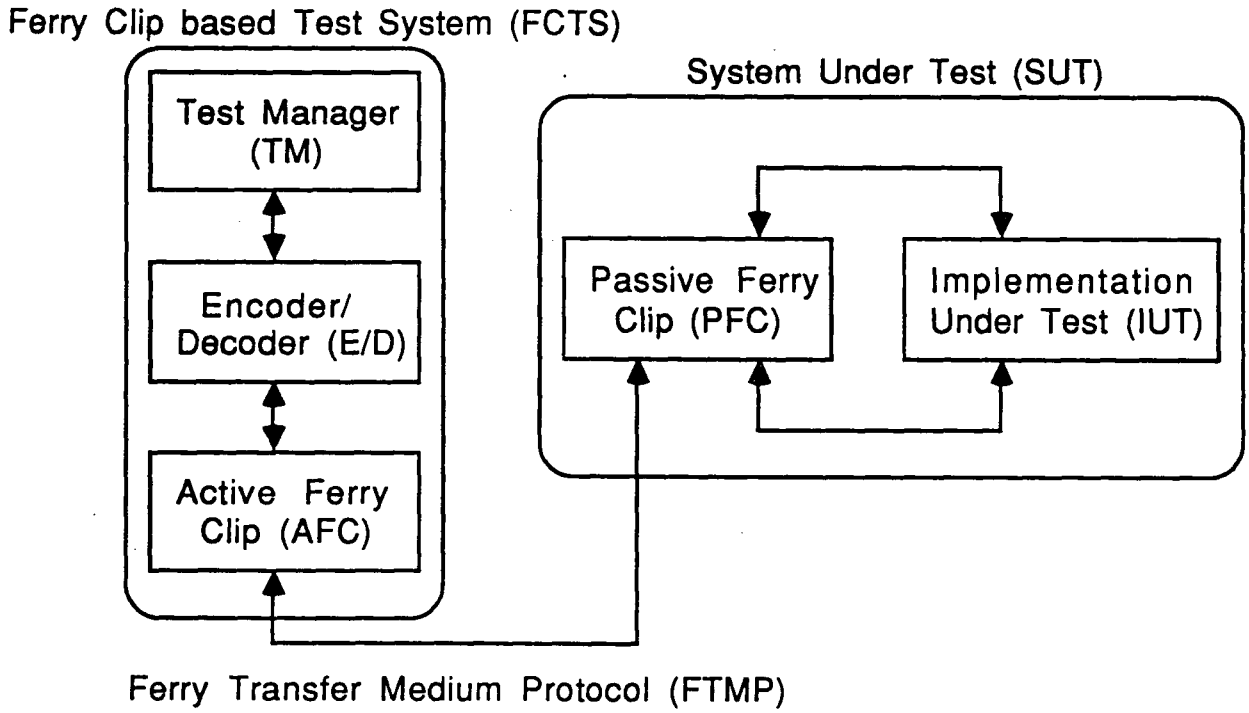

Figure 2.4: Structure of a FCTS

2.4.3 Advantages of using a FCTS

For all the cases described above, both the *Upper Tester* (UT) [2] and the *Lower Tester* (LT) [3] reside in the test system. This solves the synchronization, timing and ordering problems between the UT and the LT found in conventional test methods [17].

Once the FCP is standardized, the FCTS can be used to test different IUTs with little change required to the test system. This is in contrast to an ad hoc scheme where the entire test system or at least a major portion of it has to be rewritten to test different IUTs.

2.4.4 Implementations

The actual implementation of the FCTS prototypes will not be discussed in thesis, the interested reader is referred to [17]. However, components which are of special interest are outlined in Chapter 5 which is dedicated to the study of portability issues of using the FCTS to test protocol implementations on both the MPT and the OSI-PTE. Most of these components also apply to non-ferry applications.

[2]The Upper Tester is the piece of software which sends and receives data to and from the test system through the Upper SAP of the IUT.

[3]The Lower Tester is the piece of software which sends and receives data to and from the test system through the Lower SAP of the IUT.

Chapter 3

Test Suites and Test Management

This chapter discusses the role and operations of the *Test Manager* (TM). Different test languages and test development environments are compared, followed by a description of the design and implementation of a test development tool based on the Tree and Tabular Combined Notation (TTCN) [14].

3.1 The Test Manager

The TM is that component of the test system that oversees the operations of the system. It reads and executes the test script, performs PDU comparison and retrieval functions through the Encoder/Decoder, and logs all incoming and outgoing data exchanges for future analysis. Furthermore, it is the responsibility of the TM to continue or abort the execution of a test script if an abnormal condition is detected.

3.2 Test Management, Test Suites and Test Languages

Carrying out conformance and diagnostic testing involves three main steps. First, *abstract test suite* (ATS) for the protocol has to be derived or obtained. Second, the ATS has to be converted into *executable test suite* (ETS). The final step involves loading the ETS into the TM and running it against the IUT (Figure 3.5).

The structure of the TM on a test system is often dependent on the type of the test language [1] it will support. There are three main types of test languages, namely, TTCN,

[1] Test language is used to specify test cases.

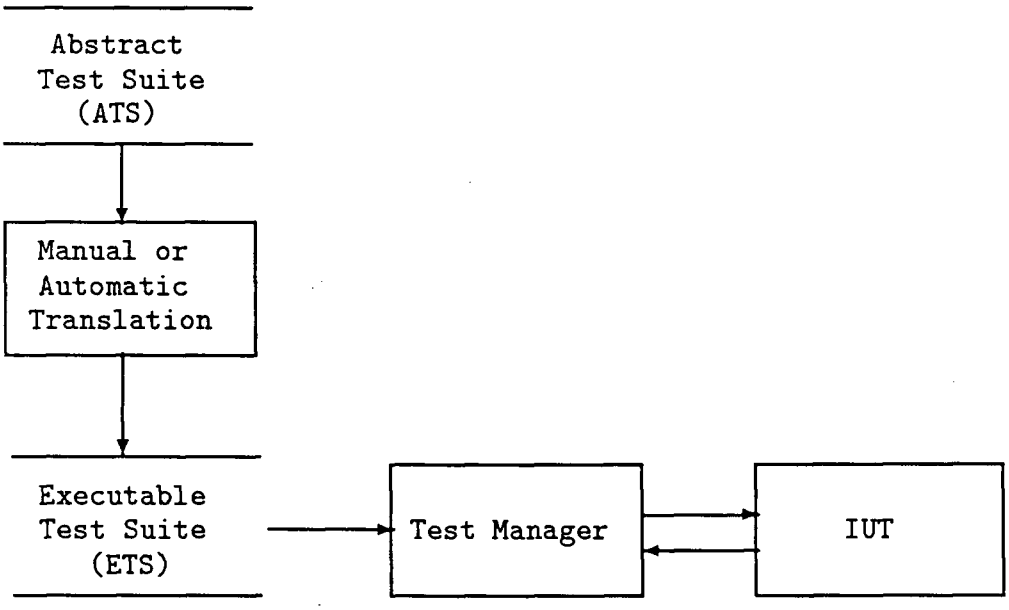

Figure 3.5: Procedure of Testing a Protocol Implementation

general purpose languages, and specialized languages.

3.2.1 Tree and Tabular Combined Notation

Tree and Tabular Combined Notation (TTCN) [14] is currently being developed by ISO as a test specification language. TTCN exists in two forms, namely the *Graphical Form* (TTCN-GR) and the *Machine Processable Form* (TTCN-MP). The former is used frequently to specify dynamic behaviors in ATS, while the latter is mainly used as an intermediate representation during the ATS to ETS translation process by TTCN test development tools. Since TTCN is a relatively new notation, its specification and usage is still undergoing revisions.

A Description of TTCN

The main purpose of TTCN is to provide a notation for specifying generic or abstract test cases which is independent of test methods, layers and protocols. The ISO document describes TTCN as 'an informal notation with clearly defined, but not formally defined, semantics'. In this way, ATS written in TTCN is intended to give precise instructions on how to carry out testing of protocol implementations in a human readable form.

The graphical form of TTCN (TTCN-GR) uses indentation to convey a tree structure of a test case, which makes it easy for human to understand (Figure 3.6).

The dynamic part of a test case is presented in the leftmost column of the table. Execution is carried out from top to bottom, and from left to right. Whenever execution of a TTCN event line succeeds, execution continues to the right. Whenever execution of a TTCN event line fails, execution continues down to the closest event line with the same indentation.

A TTCN event line has five main forms: send, receive, timer events, attachment trees and jumps. Sends, receives and timers are specified in more or less the same format:

```
<SAP>[!|?]<event> [<label>] [<constraints-ref>] [<verdict>]
```

The ! or ? stands for a send event or a receive event respectively. The SAP field is a text label identifing the SAP of the IUT at which the event is to occur. The event field specifies either an ASP, a PDU or a timer event such as TIMEOUT.

The label field is used mainly by repeats and gotos. The constraints reference field is a text cross-reference of the ASP, PDU or event whose details are specified in a separate constraints list. The verdict represents a termination of the test case, where the execution

Test Case Dynamic Behavior				
Reference: TTCN_Examples/LC_Example Identifier: LC_1 Purpose: An Example of TTCN-GR Behavior Description				
Behavior Description	Label	Constraints Reference	Verdict	Comments
EG_001 [L]				
+INITIALIZE				attach tree
L!CALL		CALL_1		
L?ACCEPT		ACC_1		
!DATA		DAT_1		
L?RR		RR_1		
L!DISCON_REQ		DISCREQ_1		
L?DISCON_CNF		DISCNF_1	pass	
L?OTHERWISE			fail	1)
L?OTHERWISE			fail	2)
L?OTHERWISE			fail	3)
Extended Comments: 1) This OTHERWISE will match if event does not match DISCON_CNF. 2) This OTHERWISE will match if event does not match DISCON_REQ. 3) This OTHERWISE will match if event does not match ACCEPT.				

Figure 3.6: Sample TTCN-GR source file

of an event line with a non-empty verdict field will cause the test case to be assigned the verdict specified and terminated.

The first line in the behavior description in Figure 3.6 is not a event. It is a test case identification line which specifies the name of the test case and the SAPs used.

Using TTCN as a Test Language

Although TTCN is originally designed for the specification of ATS, it can also be used directly as a test language if appropriate editing and parsing tools are available. The advantage of using TTCN as a test language is that ATS written in TTCN can be directly used by the test system with little or no change. However, test systems that use TTCN as a test language require complex interpreting or compiling tools which are both difficult to implement and demanding on computing hardware.

3.2.2 General Purpose Languages

For testing of protocol implementations on a smaller scale, regular programming languages like C and Pascal are commonly used to implement test suites. Usually, functions are implemented to send and receive ASPs and PDUs, and each test case is written as one program, either by hand or by the use of some test development tools.

The TM which uses a general purpose language is responsible for the archival and coordination of the test programs and the supporting libraries, which is mostly the loading and execution of compiled test cases. The TM might be simple enough to be merged into test cases so that each test case is self contained and can be directly executed.

Test suites written using general purpose languages can easily be incorporated into the test system. They can be changed easily, and can be used to express very complicated operations.

However, translation from ATS can be time consuming, and the resulting ETS is not likely to be portable.

3.2.3 Specialized Executable Test Languages

There are two main reasons for developing specialized test languages for conformance testing. First, a specialized executable test language powerful enough to handle most test suites is a good intermediate step between specification languages and program code. Second, a specialized language designed especially for testing would be less complex than a general-purpose language and easier to use.

Owing to the fact that most protocols are specified as finite state machines, specialized test languages are mostly state based and event driven. IDACOM Electronics has developed their own specialized test language called the *IDACOM Test Language* (ITL) [13]. The Corporation for Open Systems (COS) uses both ITL and another specialized language they developed called *Executable TTCN* (ETTCN) [20].

Specialized test languages are compact and precise in describing dynamic behaviors, but there is no existing standards, and specification for them are often informal. Moreover, since most specialized test languages are developed commercially, specifications may not be available for the general public.

3.3 A Comparison of Test Development Environments

A capable test system often has a sophisticated test development environment which features tools that complement the operation of the TM. A set of well designed test development tools not only enhances the ease of use of the test system, it can actually perform a large part of the work for a test suite programmer quickly and with less possibility of errors.

Several test development environments currently being developed in North America and Europe are described in this section. Their functionality and usefulness are compared and discussed to gain insight of the desirable features in test development tools.

3.3.1 The TM in the IDACOM MPT

The IDACOM MPT represents a very simple yet effective approach to test suite development. By using special features of the Forth language, the MPT is able to built all the test language features right into the native language itself. The ETS is compiled into memory on demand, and the compilation process automatically links in the necessary routines to encode and decode ASPs and PDUs.

The IDACOM MPT test development environment consists of just an editor for input of ITL [13], IDACOM's proprietary test language. ETS can also be edited on a workstation and then downloaded into the tester.

The structure of the TM in the MPT is shown in Figure 3.7.

The advantage of MPT's approach to test development is that it can be done on the test site on a portable tester relatively quickly. During diagnostic testing, the test operator often wants to create test cases which are used once and then discarded. Using the MPT, he does not have to go to a workstation, does all the compilation and downloading and then come back to do a one time test case.

The disadvantage, of course, is that the large number of test suites written in TTCN that already exist will have to be manually transformed into ITL in order to do full conformance testing. Especially in higher layer protocols where each conformance test suite has at least a few hundred dynamic behavior specifications and a thousand or more constraints, manual translation would be extremely time consuming.

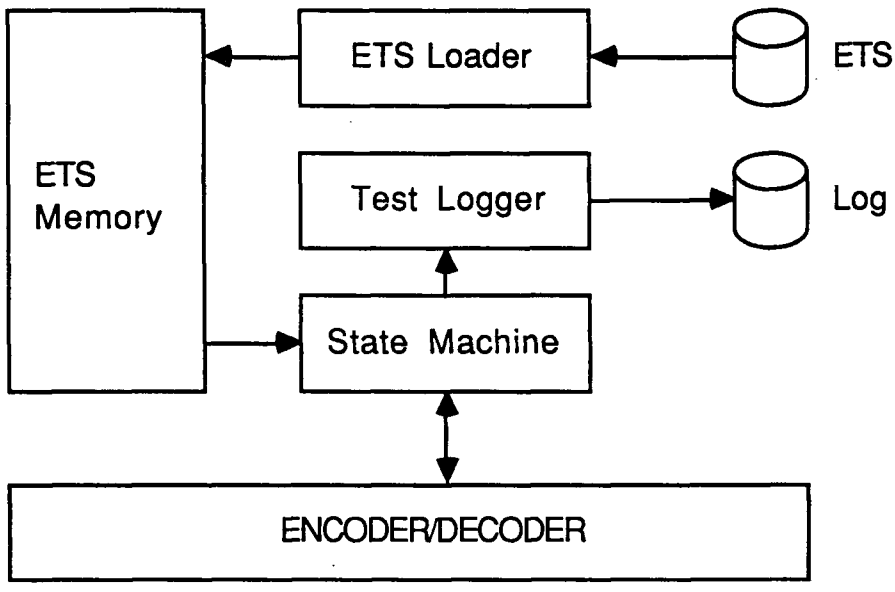

Figure 3.7: Structure of the TM in the MPT

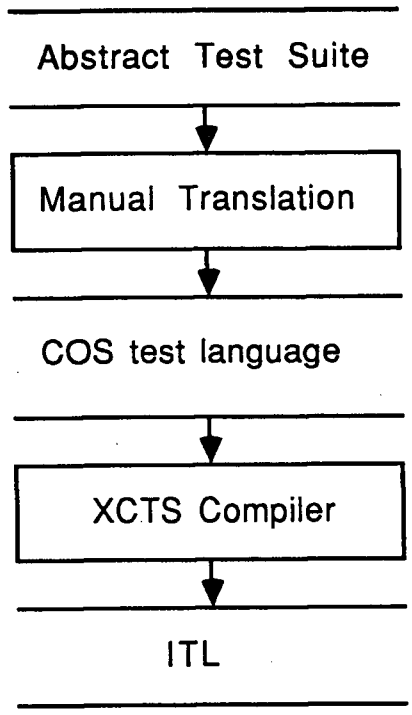

Figure 3.8: XCTS ATS to ETS Translation

3.3.2 The XCTS Project

The XCTS Project, a joint development effort by COS and IDACOM, tries to remedy the need to spend extensive effort in manually translating existing abstract test cases to ITL, mainly by moving the development environment to a SUN workstation and creating an intermediate source language between TTCN and ITL.

The structural diagram for ATS to ETS translation under XCTS environment is shown in Figure 3.8.

The intermediate test language is very similar to ITL, with additional features such as preprocessor statements and automatic parameter substitution for date and version. The main advantage, however, is the ability to archive large amount of test suites and selectively download as well as the availability of text processing tools on the workstation.

Since Abstract test cases in TTCN are still manually translated into the intermediate test language, which is state based and quite different from TTCN, development of test suites still takes time. However, with the use of the XCTS development environment, the ability to perform full conformance testing is much enhanced.

3.3.3 The ITEX environment

The ITEX environment, developed by Swedish Telecom, takes a big step forward in utilizing existing forms of ATS. A complete set of tools are available to edit and archive TTCN as defined in ISO DP9646-3. Specialized editors are used for different types of table inputs such as TTCN behaviors, test step libraries, and PDU / ASP constraints using TTCN tables or ASN.1 Modular Method [21].

The structure of the ITEX test suite development environment is shown in Figure 3.9.

Using this set of tools, ATS can be imported with just editing work. With the use of translation scripts which specify the mapping between internal representation and the object test languages, the test suites can be translated into different test languages with relative ease.

The only drawback of this system is its size and speed. The prototype development environment requires at least 8MB of main memory, and a minimum disk space of 200MB. The prototype is very slow, and much more effort has to be spent to produce a relatively bug free system of this size. In most applications other than specialized test centers, application of the tool might just be impractical.

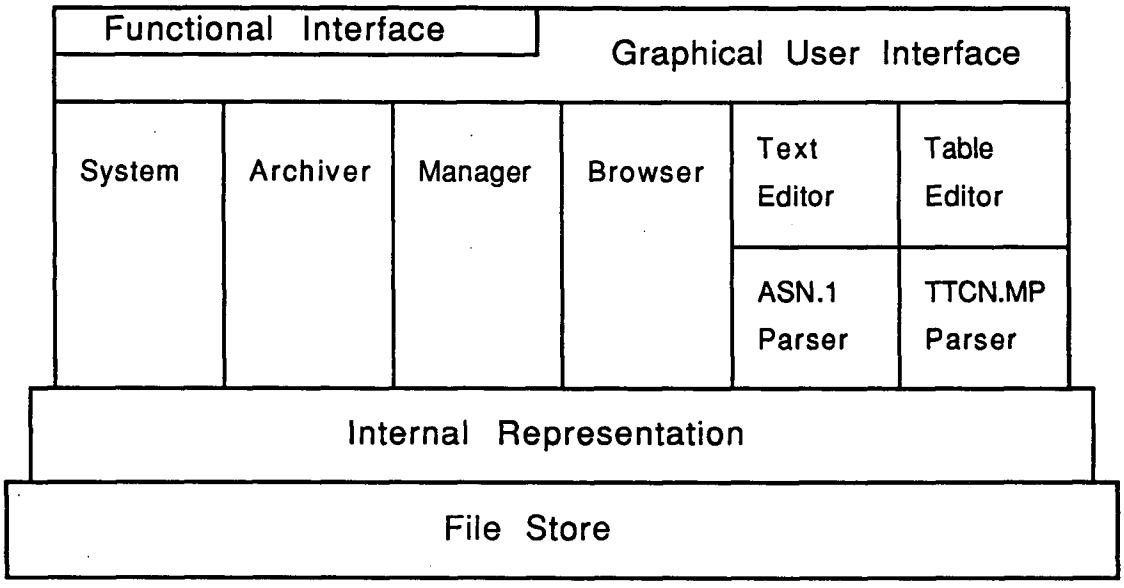

Figure 3.9: The ITEX Test Suite Development Environment

3.4 A Test Development Tool using TTCN

Based on a comparison of the environments described above, an attempt was made to build a TTCN test development tool that fits the best of both worlds. Our requirements are as follows:

- The input format of the tool must be close enough to the existing ATS that only editorial changes have to be made.
- It must provide output which a TM can execute directly.
- The size of the development tool must fit in an average workstation or a well equipped PC or portable tester, without placing heavy requirement on the hardware.

3.4.1 Description

We decided to build a parser which recognizes a plain text form of TTCN-GR (see Appendix B) and builds a test tree with the same structure. The tree is then stored as a binary image in a file, which will be loaded back into the test engine dynamically during testing.

3.4.2 Design decisions

Several design decisions were made. First of all, no special editors are to be used for input. The TTCN parser should be text based, and the tree structure is to be determined by the relative indentation of the behavior description. The reasons for these decisions are:

- Purely text based parsers are more portable.

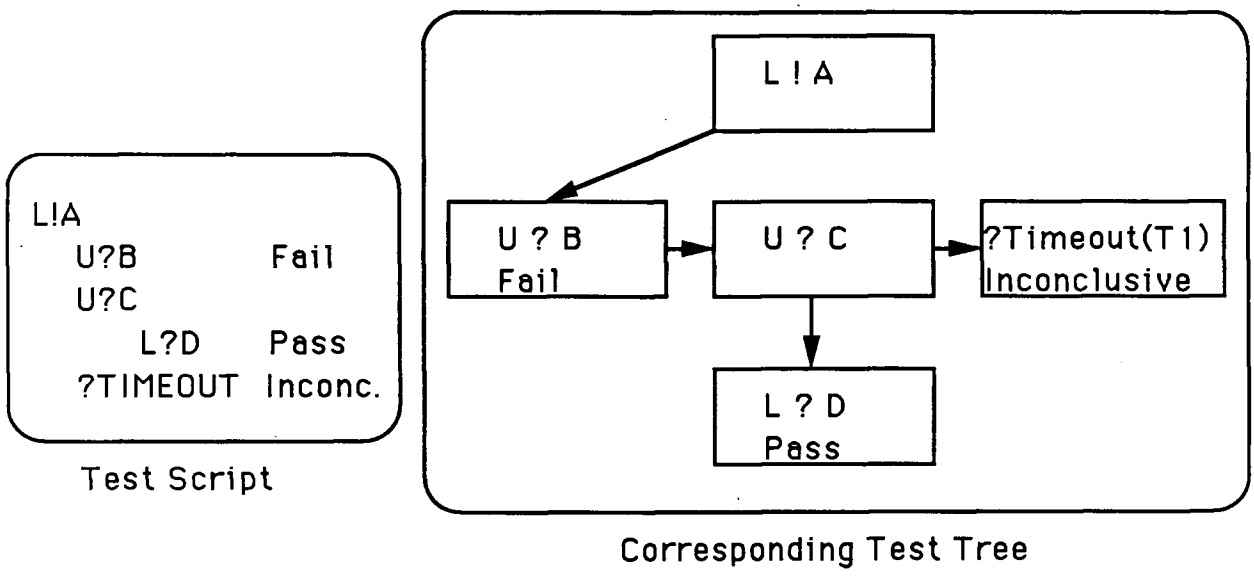

Figure 3.10: Structure of an Executable Test Tree

- Abstract test suites supplied in text format can be directly edited without having to go through complex format translation.
- The size of the development system will be reduced if no specialized editors are used.

Another decision made is that the output of the parser is to be in the form of linked trees instead of a source language. Linked trees generated by the TTCN Parser are stored in files. When the files are loaded back in, the pointers in the tree will be reconstructed. The tree can then be directly executed by the test engine.

An example of a TTCN behavior and the corresponding tree structure generated by the parser are shown in Figure 3.10.

The TM has to be specially written in order to execute the test tree. However, such a TM will be very simple since it only needs to keep track of a pointer to the tree. Each node would have the information containing what the TM should do (i.e. send an event, expect an event, and set or check a timer.), and pointers to other tree nodes. The information also includes what to do next if an action succeeds or what alternatives there are if it does not.

There are several more advantages of using linked trees:

- Linked trees fit into the architecture of an event driven system since control need not be transferred permanently to the ETS until execution terminates. Transitions from one node to another provide excellent break points for returning control to the TM.
- Re-entrance into the ETS is clean and easy since we only have to keep a pointer to a tree node.
- Linked trees are very flexible since pointers to constraints, timers, conditions and executable code can be added to a tree node easily.
- Since TTCN is itself expressed in a tree notation, implementation of TTCN features such as test step attachment in linked trees are straightforward.
- Since constraints are referenced by pointers, test suite parameters can be dynamically linked to the behavior tree at both configuration time and run time without the need for modifying and recompiling the test cases.

3.4.3 Application

With proper interface with the TM, the TTCN parser can be used to produce executable test trees from ATS quickly. Once the executable test trees are produced, the parser is

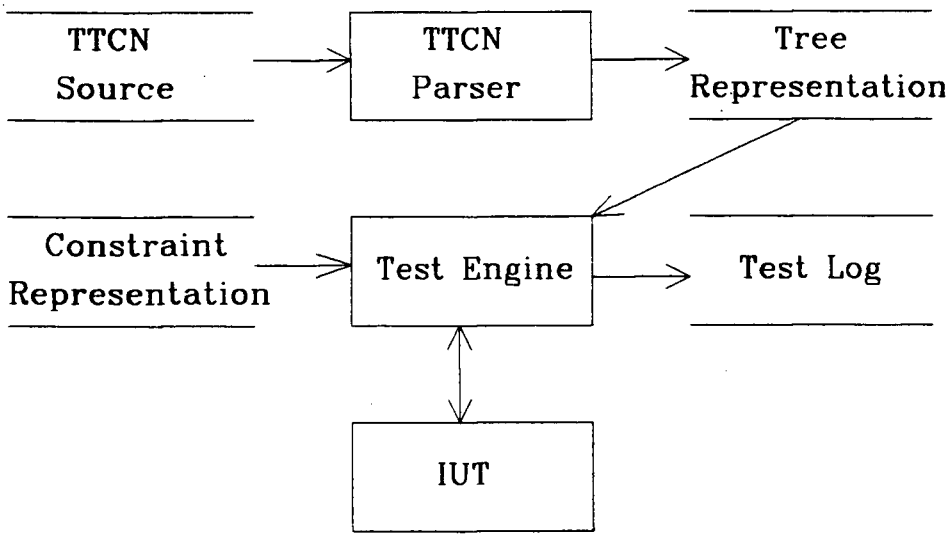

Figure 3.11: Test System utilizing the TTCN Parser

no longer needed for the execution of the test cases.

Pre-defined PDUs and ASPs are referenced by name. A separate module is built to archive the predefined PDUs and ASPs, and will be discussed in the next chapter. Awkward test cases can be hand-coded into programs whose entry points can be stored in the tree nodes. Test suite parameters can be linked into the test cases at any time because of the dynamic nature of the executable test trees.

The structure of a test system utilizing the TTCN parser is shown in Figure 3.11.

3.4.4 Implementation

A preliminary prototype of the parser was built on a SUN workstation running SUN OS 4.0. It parses basic TTCN-GR into linked trees, but certain features such as tree attachment, labels and loops have not yet been implemented. The prototype implementation has about 840 lines of C code, and is now being augmented into a full blown application at the joint UBC-IDACOM project at the University of British Columbia.

Chapter 4

PDU Encoding and Storage

A modular scheme for encoding and decoding PDUs is given in this chapter. Issues concerning encoding and decoding PDUs are discussed. The design and implementation of a PDU Library module, which is the functional equivalent of the Encoder/Decoder is also described.

4.1 The Need for Modularity

The task of an Encoder/Decoder (E/D) is to translate ASPs to PDUs and vice versa. Because translation varies from IUT to IUT, the E/D has to be rewritten for each IUT. To facilitate its replacement, the interface it provides to the TM should be clean and concise and the E/D itself should be well structured.

4.2 A Modular Scheme for Structuring the E/D

In the MPT test system implementation, The E/D module was subdivided into two parts: the primitive specification and the encoding/decoding parts. Different implementations of the same protocol might require the same primitives to be encoded differently. Hence, it should be possible for the encoding/decoding part to be replaced independently of the primitive specification part.

4.2.1 Primitive Specification

The primitive specification part defines the primitives available and their parameters. It describes what the TM is allowed to send and receive. Essentially, it provides a way for the TM to specify and access all the primitives and parameters available in the protocol definition.

The primitive specification part should be defined as completely as possible. It should include all possible primitives and all the parameters for the primitives, as specified in the protocol definition.

An example primitive specification for TP/0 is shown in Appendix C. Note that different primitives have different combination of parameters.

Regardless of which IUT being tested, a T_CONN_REQ would always be the command to request a Transport connection in the test script, and N_CONN_REQ would always be specified in the test script to match an incoming network connect request.

4.2.2 Encoding and Decoding

The encoding/decoding part is called by the primitive specification part. It does the actual transformation from primitives to PDUs or whatever representation the IUT requires.

The encoding/decoding part could choose to ignore those parameters in the primitive specification part which are not supported by a particular IUT. In this way, when a different implementation of the same protocol has to be tested, only the encoding/decoding part needs to be changed, and the test scripts [1] as well as the specification part can remain the same.

[1] the test script remains identical only if the both implementations support the same subset of functions.

4.3 Hard Coding PDUs

In the MPT E/D implementation, A Forth procedure is defined in the E/D for each primitive specified. Invoking this procedure either from the keyboard or from the test script would cause the corresponding primitive to be sent via the ferry. Decoded primitives received from the IUT via the ferry become events on the TM's event queue.

When the TM calls the E/D to send test data to the IUT, the E/D module does not return to the TM until the Active Ferry accepts the packet. The E/D module invokes the FTMP to clear the output packets if the buffers of the Active Ferry become full.

Multiple E/D modules can be provided in the test system. Different E/D modules for different layers can mix and match in order to do multi-layer protocol testing.

4.4 PDU Specification Languages

To minimize the effort required in replacing the encoding/decoding part, encoding formats of PDUs can be specified in a language.

For simple PDUs such as those in the datalink and Packet layers, one possible solution is to build an interpreter that accepts PDU specifications in some tabular form similar to that defined for the PDU definition part of TTCN-GR [14]. The primitives, parameter names for each primitive, parameter length and the allowable range for each parameter can be listed in the PDU definition part. PDUs can be built by reading in the columns in the table and reserving the correct number of bytes in the PDU buffer for each field, and then putting in the values of the parameters whose names correspond to the names assigned to the fields.

However, in higher layers such as the Session and Application layers, PDU parameters can be complex and of variable length, and a more powerful PDU specification language has to be used. For example, Abstract Syntax Notation One (ASN.1) [21] value definition

syntax is used to specify MHS test PDUs.

4.5 An E/D Equivalent - The PDU Library

Whenever PDU specification languages are used, it might be useful to pre-encode and store PDUs and receive constraints [2] , retrieving them only at run time. There are reasons why pre-encoding is sometimes preferred over direct encoding / decoding at runtime:

- PDUs in higher layers have very complex structures and a large number of parameters such that calls to the E/D to compose a PDU would be very complex and slow. In this case it is easier to pre-enode the PDU and just retrieve it by name or enumeration.

- Sometimes illegal PDUs have to be specified, and a preset encoding routine might not be able to handle illegal encodings. Examples include invalid field lengths and swapped fields.

A set of tools were prototyped under Unix for parsing ASN.1 value definitions of test PDUs and their storage and retrieval. The end goal of this set of tools is to create a library of all the test PDUs and constraints given their definitions. Since it has the power to retrieve an encoded PDU and compare a received PDU, it is functionally equivalent to an E/D module. Moreover, it is more flexible and powerful since the send PDUs and receive PDU constraints are syntactically specified instead of hard coded and compiled. The PDU Library is divided into two parts: the PDU Parser Module and the PDU Storage Module (Figure 4.12).

[2] receive PDU constraints are a list of conditions which a received PDU must match to satisfy.

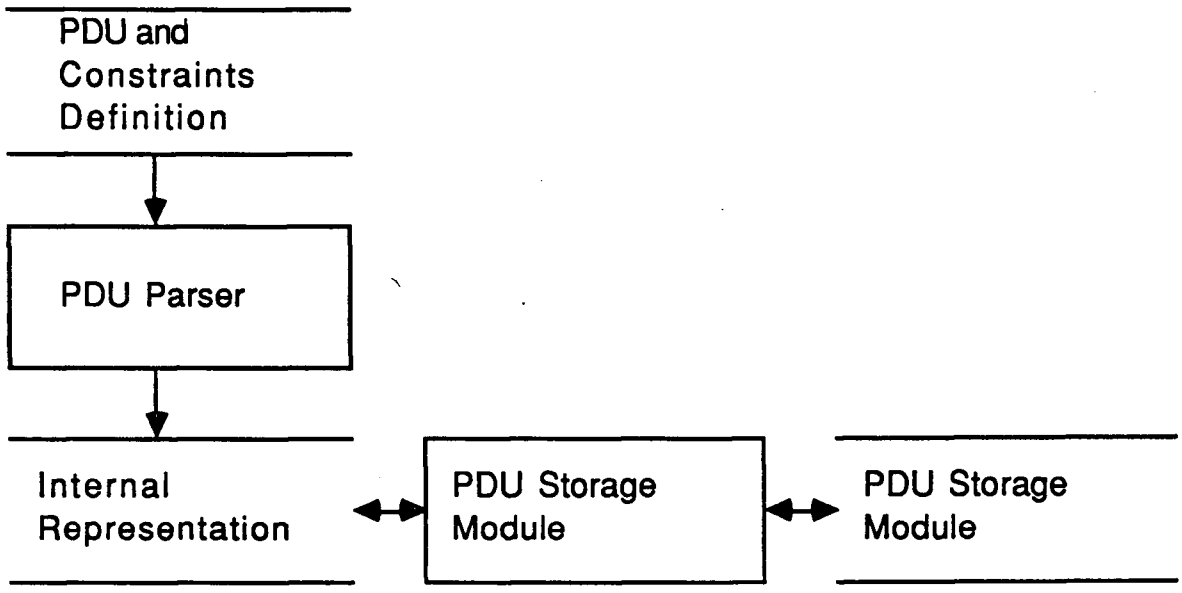

Figure 4.12: PDU Parser and PDU Storage Modules

4.5.1 The PDU Parser Module

The PDU Parser Module interprets static PDU specifications in ASN.1 Modular method [21] and generates encoded PDUs.

There are two main constructs that the PDU Parser has to deal with, namely send PDUs and receive PDU constraints.

Send PDU specification can be divided into the following:

- Base PDUs
- Pre-defined components
- Redefinition of base PDU structures
- Replacement of base PDU components

Several commonly used PDUs are defined as a whole in the base PDU library. Commonly used components are also predefined and named. Test PDUs are built from redefining and replacing base PDUs with new or predefined components.

Receive PDU constraints are specified as a list of "Components of interest". Each of the components has to be matched in order for the constraint to be satisfied.

A sample input is shown in Appendix D.

4.5.2 PDU Storage Module

The main function of the PDU Storage Module is to provide a directory service for the TM to look up a PDU. It also provides a means for the TM to compare a received PDU against the constraints. There are only two services seen by the TM:

- *Retrieve(pduName)* returns a pointer to a specific encoded PDU, given its name in plain text. The retrieved PDU can then be passed into a parameter to a send function.

- *Compare(PDU, constraintName)* compares the given PDU and see if it satisfies the constraints defined and stored under the text label constraintName. It returns TRUE if the constraints are satisfied and FALSE otherwise.

There are two reasons to separate the PDU Storage Module from the PDU Parser Module:

1. Different protocol layers may have different input syntax for PDU and constraint specifications. When switching protocol layers, the PDU Parser Module can be modified and replaced independently of the PDU Storage Module.

2. Between the PDU and constraint definitions and the retrieval and compare functions, the format of the internal representations might be dependent on the environment. For example, we might want to choose between using files or memory space to store the internal representation. By carefully designing the interface between the modules, change of internal representation is simplified.

4.5.3 Interaction between the Modules

The exact functionality of the PDU Parser depends on the power of the PDU Library module. The more powerful the PDU Library, the less the parser has to do. Two approaches are possible:

1. The PDU Parser translates Send PDUs into encoded PDU files, and the PDU Library only provides a directory service. Each receive PDU constraint is to be parsed into one compare function and stored in the PDU Library. When the TM gives a received PDU and the name of the constraint to be satisfied, the PDU Library simply calls the archived compare function corresponding to the constraint.

2. The PDU Parser simply parses Send PDUs into a tree structure, using labels to reference predefined components. The PDU Library provides a set of functions to build an encoded PDU from the tree structure. Receive PDUs are parsed into lists of field names and their correct values or ranges. When the TM gives a received PDU and the name of the constraint to be satisfied, the PDU Library will go to the list corresponding to the constraint, interpret it and compare the fields one at a time.

The exact division between these two approaches depends on how much information is known at compile time. Obviously, if most of the parameters are known at compile time, PDUs can be stored as pre-encoded files. However, if many of the parameters cannot be determined until the PICS and PIXIT are supplied for configuration, and if the parameters are of undetermined length, patching into pre-encoded PDUs will be difficult. The structures of the PDU would then have to be stored and the PDU Library will have to provide parameter encoding functions.

4.5.4 Implementation

Prototypes of the PDU Parser and the PDU Storage Module are running under SUN OS 4.0. Linked tree structures call E-nodes, which were originally used in the EAN mail system at the University of British Columbia, were chosen as the intermediate representation between the Parser and the Storage Module.

LEX [3] and YACC [4] are used to generate the source code for the Parser. The Parser was written in two parts : a Template Parser that generates a PDU template from the ASN.1 PDU specifications, and a Component Parser which takes ASN.1 Modular Method (ASN.1 MM) value declarations and turns them into E-nodes. While E-node trees are

[3]UNIX lexical analyzer generator tool.
[4]UNIX tool, Yet Another Compiler Compiler.

built, field names and tags are resolved by referencing the PDU template generated by the Template Parser. Support routines that translate the E-node structures into ASN.1 bit streams are also running.

The breakdown of the code size is as follows: 750 lines of C code for the E-node manipulation routines (including E-node to bit stream conversion routines), 490 lines of LEX and YACC source for the template parser, and 620 lines of LEX and YACC source for the component parser. Note that the figures are for the prototype implementations, and they reflect more or less the very basic system.

Chapter 5

Switching SUTs and IUTs

This chapter discusses the various issues that arise when performing conformance and diagnostic testing on different implementations under different environments, using the FCTS built on the IDACOM MPT. The backgrounds for the different environments can be found in Chapter 2.

The FCTS on the MPT was originally tested against a packet layer IUT that resided on another MPT. Later on we also used the FCTS to debug and test a TP/0 IUT under OSI-PTE.

The following have to be resolved before the MPT can communicate with the SUN workstation:

- A physical interface has to be set up.
- An FTMP has to be running through the interface at the SUT side.
- The E/D has to be modified in order to work with the OSI-PTE primitive exchange formats.

These will be discussed in detail in this chapter.

5.1 Connecting the MPT to a SUN Workstation

There are two obvious ways to connect the MPT to the SUN workstation - through the network or through a serial port. Since the MPT does not have ethernet support, we connected our machines through the serial port at the back of the SUN workstation.

There are two ways to communicate through the serial port. We can use the existing tty driver together with I/O redirection, or we can open the serial driver and do read and write directly. We chose the second option for the following reasons:

- We do not have to worry about byte streams being processed and changed by the TTY driver.
- Direct access to the drivers is faster and more efficient.
- We already have X.25 running in our FCTS, and we wanted to run X.25 on the SUT as the FTMP.

Unfortunately, only an asynchronous serial driver was available at the time while X.25 requires a synchronous serial interface. Thus, we had to set up a synchronous serial driver.

5.1.1 The Synchronous Serial Driver

We managed to get hold of a kernel X.25 implementation with a synchronous serial driver which ran on one of the older SUNs in The University of British Columbia. We were unable to use the synchronous driver directly because of the reasons below:

- The X.25 implementation resides completely in the kernel, whereas the OSI-PTE resides in user space. A user program can only access the packet layer services, but not the serial driver itself.
- The synchronous driver was written under SUN OS 3.2, while all our accessible workstations run SUN OS 4.0.

There are two ways to deal with the first problem. We can rewrite the upper interface of the synchronous driver so that it communicates with user applications, or we can use

the same interface which accesses the packet layer services of the X.25 implementation, take out both the packet layer and the datalink layer, and connect the interface directly to the synchronous driver instead. It was necessary to take a closer look into the synchronous driver as well as the X.25 packet layer service interface before deciding which approach to take.

The synchronous driver can be divided into two layers. The lower layer consists of a set of low level interrupt handlers for sending and receiving raw data. The higher level consists of some kernel interface routines which passes data through mbufs [1]. Passing of mbufs between the datalink layer the higher level routines in the synchronous driver are done through software interrupts. Since system interrupts can not be carried over to user space, a set of device access routines would have to be written if direct interfacing of the driver to user space is desired. That means totally rewriting the higher level interface so that it could be opened as a /dev entry.

The X.25 implementation, however, has a socket interface that is accessible to the user programs. It creates an entry in the protocol switch table of the kernel, such that whenever a program opens an inet [2] socket with domain AF_CCITT and interface name zss$n$, the socket will connect to the X.25 implementation which communicates through the synchronous driver on serial port n. Since the kernel end of the socket interface also uses mbufs to pass data in and out of the sockets, we can easily bypass X.25 by rewriting the send and receive interrupt handlers such that mbufs are passed directly between the synchronous driver and the socket interface. This then was the approach we adopted.

[1] mbufs are standard Unix system memory buffers.
[2] Unix internet domain.

5.1.2 The Kernel to User Interface

The modified driver is accessed the same way the X.25 implementation is accessed. A socket of family AF_CCITT and ifname "zss0" opens a synchronous driver on serial port A, and the same socket with ifname "zss1" opens a synchronous driver on serial port B. After the correct socket is opened, the driver can be accessed through ioctl() calls on the socket.

There are 4 different primitives for an ioctl() call to the driver. SIOCGIFFLAGS and SIOCSIFFLAGS are used to read and write the status flags for the specified port, and SIOCGIFADDR and SIOCSIFADDR are used to read and write the configuration information.

A sample piece of code that access the driver is shown in Appendix C.

5.1.3 Port Configuration

The serial ports on the back of the SUN workstations were designed to talk to devices such as modems, and are configured as DTE ports. However, the test system is responsible for setting up a ferry connection, and is therefore classified as a DTE also. In order to connect the test system to the SUN workstation, we have to either use a null modem cable or modify the test system so that it talks to a DTE. Fortunately, the IDACOM MPT allows the role of the protocol (i.e., as a DTE or DCE) to be chosen independent of the interface, thus we were allowed to keep the testing software to operate as DTE while using a DCE type interface.

5.2 Adding a TP/0 E/D

5.2.1 Functions implemented in the TP/0 E/D

Like the packet layer E/D, there is no reference implementation in the TP/0 E/D. The only functions provided in the TP/0 E/D are sending and receiving of Transport services through the upper SAP, and the sending and receiving of TPDUs through the lower SAP.

5.2.2 Difference from the Packet Layer E/D

In the Packet Layer E/D, all primitives are encoded into a single linear buffer packed in one Ferry packet. The linear buffer is accepted directly by the IUT as input. The main difference of the TP/0 E/D is that it must turn the Transport Services into a form the IUT recognizes, which is non-linear. The TP/0 IUT takes parameters in a structure form, and the structure can have pointers to buffer structures. Obviously, one cannot pass linked structures between the Active and Passive Ferries. Thus, a Transport Service Primitive is broken down into three ferry packets before it is sent. The first packet consists of the event identifier which identifies the parameters to follow. The second packet contains a linear structure which has all the parameters to the primitive associated with the event ID, while the third packet has the service data packed into a linear buffer (see Figure 5.13). Note that in some primitives where service data is not present, the buffer structure does not exist. In the case where the primitive has no parameters, there may not even be an EPA. Thus in the first two bytes of the second ferry fragment, two boolean values denote the presence or absence of the EPA part of the second and third ferry packets respectively.

In the Packet Layer IUT, no useful information is carried inside the (N-1) SDU. However, TPDUs are carried in the Network Service Data. Thus another difference from the Packet Layer E/D is that the TP/0 E/D has to do encoding and decoding of data

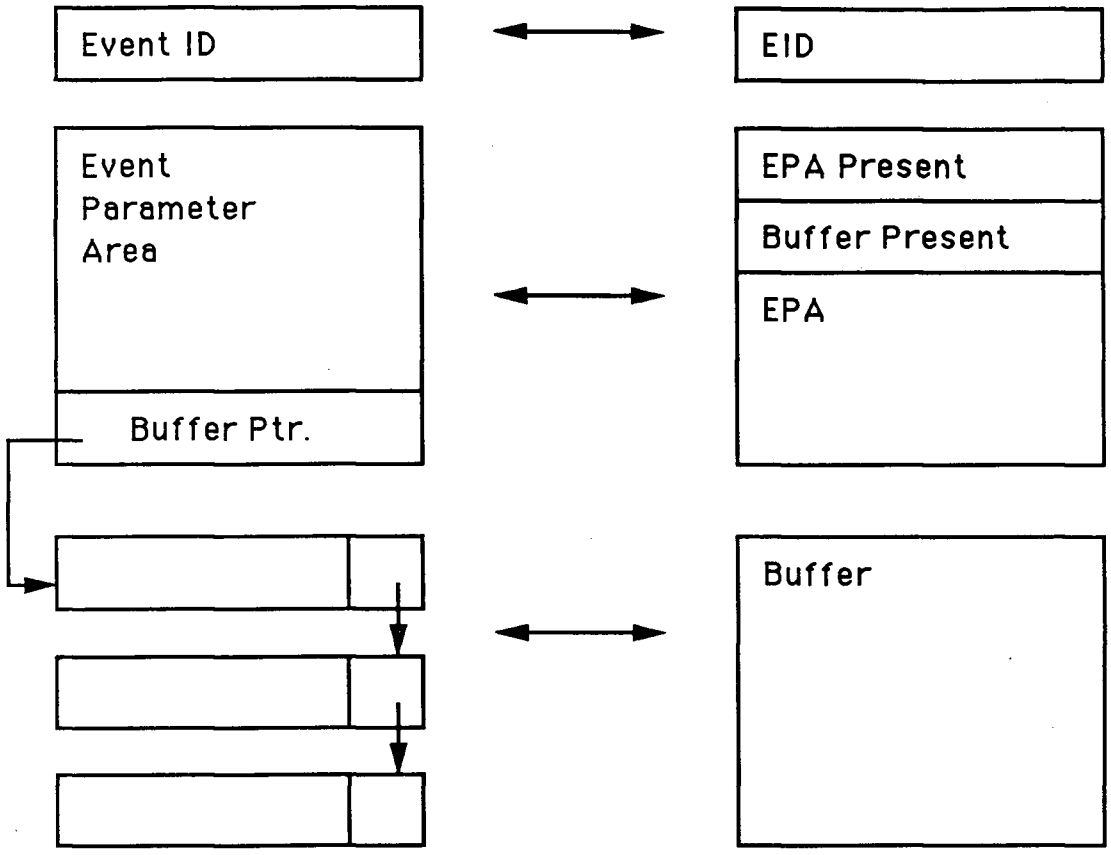

Figure 5.13: PTE ASP to Ferry Buffer Translation

packed inside the (N-1) service. When data comes from the lower SAP, the encoder has to determine the type of network service primitive, as well as what TPDU it contains in case the network primitive is a data indication.

5.3 Interfacing with the OSI-PTE

5.3.1 The Dispatcher

The dispatcher is written specifically to handle communication between the OSI-PTE stack and the serial interface. It is integrated with the physical layer. Its function is to turn incoming frames and timeouts into events and post them to the appropriate layers. As Unix sockets do not generate interrupts on receipt of data from the serial interface, incoming data is detected by periodic polling. The system timer generates interrupts which update the global PTE timer queue on timeouts, and expired timer events are extracted from the queue and processed between incoming frames. A simplified diagram of the dispatching algorithm is shown in Figure 5.14.

5.4 Diagnostic Testing a TP/0 Implementation in OSI-PTE

When we tested the TP/0 implementation (IUT) in the OSI-PTE, the implementation is not even complete. With the help of our test system, we were able to get the IUT running in considerably less time than if the test system were unavailable.

5.4.1 Using Interactive Mode testing in the TM

The main advantage of using Forth in the MPT test system is that Forth is an interpreted language. This means that interactive commands can be constructed and executed on the fly during actual testing. This speeds up diagnostic testing by eliminating the need to edit and recompile test programs.

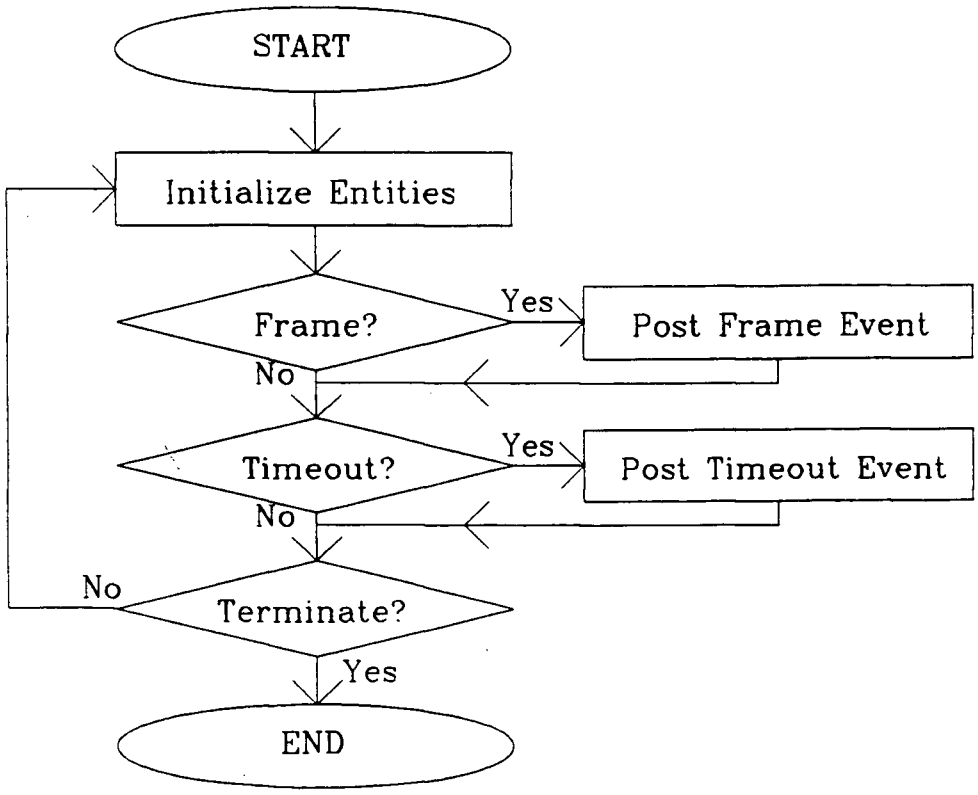

Figure 5.14: Algorithm of the PTE Dispatcher

Using the interactive mode of the TM, we were able to issue a send or start timer event to the IUT, watch for an event from the IUT on the monitor, decide what action to take, and then type in that action on the keyboard. We were also able to construct loops that send an arbitrary number of data primitives to the IUT in one sentence on the keyboard. As well, it was easy to peek in and change any parameter of any primitive to be sent by just modifying the corresponding Forth variable interactively. Since the built in Forth interpretor is used also as the command interpretor for the TM, no special command interpretor for those commands has to be written.

5.4.2 Tracing States in the SUT

The *state out* event in the OSI-PTE entity came in very handy when we tried to debug the TP/0 implementation. We wrote a separate entity that intercepts and prints all state out events posted by specified entities. Cross referencing the state diagram on the protocol specification, we were able to locate quickly the implementation errors.

The typical sequence of debugging using both the interactive mode of the TM and the state tracing entity is as follows:

1. Go to a chosen state.
2. Issue a selected event interactively on the test system.
3. If the IUT goes to the correct state, mark an OK in that entry of the protocol specification state table.
4. If the IUT fails to respond to that event, mark that entry of the protocol specification state table as a fail verdict.
5. For each failure edge, examine the IUT source code, find the case statement for that particular state and the case for that particular event and correct the code.

6. repeat until all edges are marked OK.

At this point of disgnostic testing, PDUs sent are mostly default PDUs. By the end of the diagnostic test sequence, the IUT should be working in all normal cases. Conformance testing can then be performed.

5.5 Conformance Testing the TP/0 Implementation

After the implementation was sufficiently debugged, we ran a set of test cases against it. The test cases were generated by traversing the External Behavior Expression (EBE) [18] graph of TP/0.

5.5.1 Test Suite Selection

Applying the algorithms in [18], a set of I/O subpaths were generated. Because of limitations in the protocol implementation, only control flow was considered. Thus a test case was derived from each I/O subpath.

5.5.2 Translating into ITL

The 29 I/O subpaths were manually translated into 14 ITL test scripts. The smallest of them consisted of two states, and the largest of them had nine states.

5.5.3 Conformance Test Results

A number of errors in the state machine were detected as a result of running the test scripts on the TP/0 implementation. These have been corrected.

In some of the events, connection information was not passed correctly between the transport service and the TPDUs, and that has been corrected also.

Some of the features that the IUT was supposed to support was found to be missing in the implementation, and that section of the code was added.

Overall, the 14 test scripts ran uncovered a surprisingly large amount of errors and irregularities in the implementation. These would be much more difficult to catch and to correct if testing was not carried out in the first place.

After the testing was performed, we also felt much more confident that the implementation will interwork with other implementations that conform to the specifications.

Chapter 6

Conclusions

During the period of our research, prototype test systems were implemented in the Unix, MPT and OSI-PTE environments. Several protocol implementations in the above environments were conformance and diagnostic tested. An attempt was made to address various issues of test system design and reported in this thesis, including the followings:

- Test system implementation issues in different protocol testing environments were discussed and compared.
- Existing test development tools and environments were studied, and new tools built for test development and test management were discussed.
- A scheme for structuring the E/D in a modular fashion was given. PDU specification methods were compared, and a mechanism for storing and retrieving test PDUs was given.
- Issues and difficulties in actual conformance and diagnostic testing were described, and their solutions summarized.

In summary, this thesis has covered a wide range of protocol testing issues in order to gain insight into what protocol testing in reality is all about.

By using the Ferry Clip based Test System extensively for all our testing and debugging, feasibility and usefulness of the Ferry Clip concept is confirmed.

Appendix A

Sample ITL Test Script

```
( SAMPLE FERRY CLIP ITL TEST SCRIPT                                    )
(                                                                      )
( ENTER FUNCTION KEY CF1 TO START THE TEST                             )

TCLR

0 STATE{ FK_CF1 ACTION{
                    PRINT_TIME WCR
                    " TEST STARTING" BTYPE WCR 20 COUNTER1 !
                    F_CONN 1 NEW_STATE
                }ACTION
  }STATE

1 STATE{  F_CONN_CONF 1 ?RX ACTION{
                                TPT:T_CONN_REQ
                                2 NEW_STATE
                              }ACTION
          OTHER_EVENT       ACTION{
                              " VERDICT : " RTYPE " INCONCLUSIVE" YTYPE WCR
                              TM_STOP
                            }ACTION
  }STATE

2 STATE{ T_CONN_CFM 1 ?RX ACTION{
                              TPT:T_DISC_REQ
                              3 NEW_STATE
                            }ACTION
          OTHER_EVENT       ACTION{
                              " VERDICT : " RTYPE " FAILED" RTYPE WCR
                              F_DISC TM_STOP
                            }ACTION
  }STATE

3 STATE{ N_DISC_REQ 1 ?RX ACTION{
```

```
                        F_DISC
                        " VERDICT : " RTYPE " PASSED" BTYPE WCR
                        " TEST FINISHED" BTYPE WCR TM_STOP
                       }ACTION
        OTHER_EVENT    ACTION{
                         " VERDICT : " RTYPE " FAILED" RTYPE WCR
                         F_DISC TM_STOP
                       }ACTION
}STATE
```

Appendix B

Sample Input to TTCN Parser

```
DEFAULT IDENTIFIER: LIB_otherwise
COMMENTS: can be used wherever unexpected and illegal events
  are to be trapped
DEFAULTS:
BEGIN
    LIB_otherwise[X]
        X?OTHERWISE fail
END

TEST IDENTIFIER: 306.2.1.3
SUMMARY: Test whether the Auto-forward-Indication service element of
    the IUT autoforwards an IM-UAPDU to the tester and generates a
    NonReceiptNotification with reason "autoforwarded"
BACKWARDS REFERENCE: 306.2.1.3
DEFAULTS: LIB_otherwise[I]
BEGIN
    T!SUBreq                SUBreq_17
    I!Start          AFNR-Tmr
            T?DELind         DELind_94
                T?DELind     DELind_95
                I?Timeout    AFNR-Tmr          fail
            T?DELind         DELind_95
                T?DELind     DELind_94
                I?Timeout    AFNR-Tmr          fail
            I?Timeout        AFNR-Tmr          fail
END
```

Appendix C

Primitive Specification for TP/0

SEND PRIMITIVES	RECEIVE PRIMITIVES	PARAMETERS
N_CONN_IND	N-CONN-REQ	LI
N_CONN_CFM	N-CONN-RSP	CRDT
N_DATA	N-DATA-REQ	SRC
N_DACK_IND	N-DACK-REQ	DEST
N_EXPD_IND	N-EXPD-REQ	CLASS
N_RSET_IND	N-RSET-REQ	OPT
N_RSET_CFM	N-RSET-RSP	DREASON
N_DISC_IND	N-DISC-REQ	RCAUSE
T_CONN_REQ	T-CONN-IND	EOT
T_CONN_RSP	T-CONN-CFM	NR
	T_DISC-IND	ID
T_DATA_REQ	T-DATA-IND	
T_EXPD_REQ	T-EXPD-IN	
CR	TPDU_CR	
CC	TPDU_CC	
DR	TPDU_DR	
DC	TPDU_DC	
DT	TPDU_DT	
ED	TPDU_ED	
AK	TPDU_AK	
EA	TPDU_EA	
RJ	TPDU_RJ	
ER	TPDU_ER	

Appendix D

Sample Input to PDU Parser

```
IM-UAPDU_1_1
REDEFINE P3.DeliverEnvelope::=
  SET{[0] IMPLICIT P1.ContentType,
      original [1] IMPLICIT P1.EncodedInformationTypes,
      [2] IMPLICIT P3.DeliveryFlags,
      thisRecipient [4] IMPLICIT P1.ORName,
      submission [7] IMPLICIT P1.Time
     }
PDU BASE_IM-UAPDU_1
REPLACE
   BASE_IM-UAPDU_1_Body
BY
   SEQUENCE OF{
       SEQUENCE{SET{
                 SET{
                 ContentType INTEGER      [2]                 /*p2*/
                 original SET{BITSTRING ['20'H]}              /*{iA5Text}*/
                 DeliveryFlags
                   BITSTRING              ['40'H]             /*conversion*/
                                                              /*Prohibited*/
                 thisRecipient P1.ORName [TSP_ORName_I_1]
                 submission Time}        [TSP_UTCTime_1]
                   }
                IM-UAPDU                 [L_IM-UAPDU_1]
               }
              }

IM-UAPDU_0_1 PARTIAL DEFINITION {
IM-UAPDU.Heading.originator
  ORName [TSP_ORName_I_1]
IM-UAPDU.Heading.authorizingUsers
  1 ORDescriptor [L_ORDescriptor_4]
}
```

Appendix E

Sample Code to access the Synchronous Serial Interface

```
main (argc, argv)
register char **argv;
int argc;
{
    register int s;

    /* Open socket connection to serial interface */
    if ((s = socket (AF_CCITT, SOCK_STREAM, 0)) < 0)
        syserr ("socket");

    /* Get interface name from command line */
    argv++;
    ifname = *argv;

    /* Get interface flags */
    strcpy (ifr.ifr_name, ifname);
    if (ioctl (s, SIOCGIFFLAGS, (char *)&ifr) < 0)
        syserr ("ioctl (SIOCGIFFLAGS)");
    ifflags = ifr.ifr_flags;

    /* Set default options */
    strcpy (ifr.ifr_name, ifname);
    if (ioctl (s, SIOCGIFADDR, (char *)&ifr) == 0)
        bcopy((char*)&ifr.ifr_addr, (char *)&x25conf, sizeof(x25conf));

    /* Display status */
    if (argc == 2) {
            status ();
            exit (0);
    }
}
```

Bibliography

[1] H. X. Zeng and D. Rayner, *The impact of the ferry concept on protocol testing*, in Diaz, M. (ed.), Protocol Specification, Testing, and Verification V, p.533-544, North-Holland, 1986.

[2] H. X. Zeng, X. F. Du and C. S. He, *Promoting the "Local" Test Method with the New Concept "Ferry Clip"*, Proceedings of the 8th IFIP Symposium on Protocol Specification, Testing and Verification, Atlantic City, June 1988.

[3] H. X. Zeng. Q. Li, X. F. Du and C. S. He, *New Advances in Ferry Testing Approaches*, Journal of Computer Networks and ISDN Systems, 15,1 (1988).

[4] H. X. Zeng, S. T. Chanson and B. R. Smith, *On Ferry Clip Application in Protocol Testing*, Journal of Computer Networks and ISDN Systems, Vol. 17, July 1989.

[5] S. Sechrest, *An Introductory 4.3BSD Interprocess Communication Tutorial*, MT XINU Manual, 4.3BSD with NFS, Programmer's Supplementary Documents, Volume 1, PS1, 1986.

[6] ISO/TC 97/SC 21 N, 2nd DP 9646, *Conformance Testing Methodology and Framework*, 1987.

[7] CCITT Draft Recommendation X.200, *Reference Model of Open System Interconnection for CCITT Applications*, 1988.

[8] CCITT Draft Recommendation X.25, *Interface Between DTE and DCE Terminals Operating in Packet Mode*, 1988.

[9] CCITT Draft Recommendation X.213, *Network Service Definition for OSI for CCITT Applications*, 1988.

[10] CCITT Draft Recommendation X.223, *Use of X.25 to Provide OSI Connection-Mode Network Service*, 1988.

[11] G. V. Bochmann and C. S. He, *Ferry Approaches to Protocol Testing and Service Interfaces*, Proceedings of the 2nd International Symposium on Interoperable Information Systems, Tokyo, Japan, November 1988.

[12] IDACOM Electronics Ltd., *MPT368.2 User Manuals - Forth Programming*, November 1987.

[13] B. R. Smith *ITL - IDACOM Test Language - Language Specification* , Version 1.0, UBC-IDACOM Project Documentation, 7 October 1988.

[14] ISO Working Document DP 9646-3, *The Tree and tabular Combined Notation*, 12 July 1988.

[15] R. I. Chan, *OSI PT Environment*, Version 1.32, UBC-IDACOM Project Documentation, 21 September 1988.

[16] R. I. Chan et el., *A Software Environment for OSI Protocol Testing Systems*, Proceedings of the 9th IFIP Symposium on Protocol Specification, Testing and Verification, Enschede, The Netherlands, June 1989.

[17] N. J. Parakh, *The Implementation of a Ferry Clip Test System*, M.Sc. thesis, Department of Computer Science, University of British Columbia, 1989.

[18] J. Wu and S. T. Chanson, *Test Sequence Derivation Based on External Behavior Expression*, Proceedings of the Second International Workshop on Protocol Test

Systems, W. Berlin, Germany, October 1989.

[19] CCITT Recommendation X.409, *Message Handling Systems: Presentation Transfer Syntax and Notation"* 1984.

[20] A. Boshier, A. McKie, D. Dwyer, *ETTCN - an executable test language"*, position statement, Proceedings of the First International Workshop on Protocol Test Systems, 1988.

[21] ISO 8824:1987, *Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*, 1987.