DESIGN AND IMPLEMENTATION OF A FERRY CLIP TEST SYSTEM

by

S.T. Chanson, B.P. Lee, N.J. Parakh, and H.X. Zeng

Technical Report 88-24

December 1988

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5 Canada

DESIGN AND IMPLEMENTATION OF A FERRY CLIP TEST SYSTEM†

by

S. T. Chanson, B. P. Lee, N. J. Parakh and H. X. Zeng
Technical Report 88-24
December 1988

Department of Computer Science
University of British Columbia
Vancouver, B.C.,
Canada V6T 1W5

---

† The work has been supported in part by the Natural Sciences and Engineering Research Council of Canada, and IDACOM Electronics Ltd. of Edmonton, Canada.

Abstract

The Ferry Clip concept can be used to build a Test System for protocol testing. By structuring the system into a set of modules, it is possible to minimize the effort required in using such a system to test different protocol implementations. In this paper we describe a method for structuring and implementing a Ferry Clip based Test System. Implementation issues encountered in building such a system under different environments are also discussed.

Abbreviations

ASP	Abstract Service Primitive
CPU	Central Processing Unit
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
LMAP	Lower Mapping Module
LT	Lower Tester
MPT	Multi-port Protocol Tester
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PT	Protocol Tester
PTE	Protocol Testing Environment
SAP	Service Access Point
SUT	System Under Test
TM	Test Manager
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester

1 Introduction

Open Systems Interconnection (OSI) requires conformance testing of protocol implementations. Conformance testing involves determining whether a given protocol implementation conforms to the specification as defined by the standards to which it purports to adhere. The purpose of conformance testing is to increase the probability that different protocol implementations can interwork together. Conformance testing can be done in test centers for certification purposes or by the implementors during protocol development for diagnostic purposes. Since diagnostic testing is performed by the vendor, all the available *service access points* (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).

The *International Standards Organization* (ISO) has defined a set of abstract test methods for the conformance testing of computer protocols [6]. The Ferry Clip concept [2] [3] [4] is a test approach to realize those test methods and is a generalization of the Ferry Concept as defined by Zeng [1]. The main idea is to transport test data transparently from the *system under test* (SUT) to the test system thus allowing both the *upper tester* (UT) and the *lower tester* (LT) to reside in the test system (see Figure 1). This simplifies the synchronization between the UT and the LT and reduces the amount of software that must reside in the SUT [1] [3]. A Ferry Clip system consists of two major components, an Active Ferry Clip which resides in the test system, and a Passive Ferry Clip which resides in the SUT (see Figure 1). The two ferries use the services of the *ferry control protocol* (FCP) to transfer test data between the *Test Manager* (TM) in the test system and an external IUT residing in the SUT. The FCP provides a standard interface on top of some existing protocol such as X.25 which actually transfers the test data, and which we shall call the *ferry transfer medium protocol* (FTMP). Once the FCP is standardized, the ferry clip test system 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.

One of our main goals was to study the design and implementation issues involved in building a *Ferry Clip based Test System* (FCTS) on various hardware and software architectures. Emphasis was placed on structuring the FCTS so as to minimize the

amount of code that would have to be rewritten to test different IUTs or use different FTMPs.

As of this date, we have implemented the system on three different environments - Unix, MPT and OSI-PTE. Unix is a popular operating system which runs on many different mainframes and workstations; MPT [12] 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. This paper will focus on the design and experience gained in the implementation for the MPT environment with a brief description of the other two. In this paper the terms Active Ferry and Active Ferry Clip as well as the terms Passive Ferry and Passive Ferry Clip are used interchangeably. Following the introduction, section 2 describes a general scheme for structuring the Active and Passive Ferry Clips. Section 3 briefly describes the role of the TM in an FCTS, and a modular scheme for constructing an *encoder/decoder* (E/D). Section 4 discusses the implementation of the FCTS in the MPT environment. Section 5 describes some of the protocols that have been tested by the FCTS and section 6 concludes the paper.

2 Structuring the Ferry Clips

2.1 The Active Ferry Clip

The functions performed by the Active Ferry are independent of the particular IUT being tested. Hence, changing the IUT has no effect on the Active Ferry. However, since the Active Ferry uses the services of an underlying FTMP, part of its code is FTMP specific. Nevertheless, a considerable portion of the Active Ferry's functions such as fragmenting, reassembly and buffering of test data are operations that are independent of the FTMP.

To minimize the change to the Active Ferry when a different FTMP is used, it was decided to structure the Active Ferry into two modules (see Figure 2(a)) as follows.

1. The Active Ferry Finite State Machine Module (Active Ferry FSM)

   All functions of the Active Ferry that are independent of the FTMP are incorporated into this module. The main function of this module is to implement the Active Ferry's protocol state machine - hence its name. Specifically, fragmenting

and reassembly of test data packets as well as buffering of test data to be sent to the Passive Ferry are performed by this module.

The Active Ferry FSM interacts with the Test Manager module and the *Lower Mapping Module* (LMAP) through single interfaces by means of *abstract service primitives* (FM-ASPs and FT-ASPs respectively). These are discussed in [4] and listed in Appendices C and D. The state table of the Active Ferry is given in Appendix A.

2. The Active Ferry Lower Mapping Module (Active Ferry LMAP)

   This module contains all the code that is dependent on the particular FTMP. Specifically, it maps the Active Ferry ASPs (namely the FT-ASPs) into the ASPs or commands specific to the FTMP being used. The complexity of this mapping and hence the corresponding size of the LMAP module depends on the FTMP as well as the interface it provides.

   By localizing the code specific to the FTMP in this module, it is possible to change the FTMP by simply rewriting this module. No change to any other part of the Active Ferry is required. A library of LMAP modules corresponding to different FTMPs can be set up. Thus, the problem of configuring the Active Ferry to use a particular FTMP supported by the SUT is reduced to simply selecting an appropriate Active Ferry LMAP module from the library.

2.2 The Passive Ferry Clip

The chief goal in designing a Passive Ferry is to keep it small and compact so that it may be possible to implement it in a SUT with memory limitations. The Passive Ferry's code depends on the IUT as well as on the FTMP being used. Once again, our aim was to structure the Passive Ferry to facilitate the replacement of both the FTMP and the IUT. Hence, it was decided to structure the Passive Ferry into three modules (see Figure 2(b)) as follows.

1. Passive Ferry Finite State Machine Module (Passive Ferry FSM)

   This module contains all the functions of the Passive Ferry that are independent of the IUT and FTMP. It implements the Passive Ferry protocol state machine as described in [4] and listed in Appendix B. Fragmentation, reassembly and the buffering of test data packets (see section 4.1.3) are performed by this module.

The Passive Ferry FSM interacts with the FTMP via a set of Ferry ASPs (FT-ASPs) and with the IUT Interface module (described below) via another set of ASPs (FD-ASPs) as described in [4] and listed in Appendix E.

2. Passive Ferry Lower Mapping Module (Passive Ferry LMAP)

   This module contains the code for all the Passive Ferry functions that are specific to a particular FTMP, but independent of the IUT. It maps the Passive Ferry ASPs (FT-ASPs) into the ASPs specific to the FTMP being used. Hence, it is the only module that needs to be modified in the Passive Ferry when the FTMP is replaced by a new one.

3. IUT Interface Module

   This module contains all the code specific to the IUT. The Passive Ferry interfaces with the IUT through its upper and/or lower SAPs. Some data conversion is usually necessary to convert the data received by the Passive Ferry in a ferry *protocol data unit* (PDU) into a format that is accepted by the IUT. This format is usually unique to each IUT and hence this code needs to be rewritten or modified for each new IUT.

2.3 The Ferry Transfer Medium Protocol

In order to make the FCP simple and easy to implement, certain requirements have to be placed on the FTMP. Since the FCP cannot handle lost, mangled or out of sequence packets it is necessary that the FTMP guarantees end-to-end error free delivery of data and does not resequence data packets. The protocol used as the FTMP may be connection or connectionless oriented, and either stream (eg. Unix INET stream sockets [5]) or packet (eg. X.25 [8]) oriented.

3 The Test Manager and Encoder/Decoder

3.1 The Test Manager

The TM is that component of the FCTS that oversees the operation of the system. It reads and executes the test script 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.

The TM communicates with the Active Ferry through the E/D module (see Figure 1). Communication between the TM and the Active Ferry is via the set of FM-ASPs (see Appendix D). The test data is encoded by the E/D module so as to make it easy for the IUT Interface to convert the test data into a form that the IUT accepts. Similarly, data received by the Active Ferry is sent to the TM through the E/D module so that it can be converted into a format the TM understands.

With the Ferry Clip approach, events for both SAPs of an IUT can be specified in the same test script. Furthermore, since both the UT and LT can be merged together within the TM on the test system, the synchronization problems between the UT and LT do not occur in a FCTS.

The Ferry Clip concept could be extended to incorporate multiple Ferry Clips inside the SUT. Interaction and synchronization between different IUTs could be specified in a single test script. This could prove to be a useful feature for testing a protocol stack, allowing observation of the protocol exchanges at various layer boundaries.

3.2 The Encoder/Decoder

The E/D has to be rewritten for each IUT. To facilitate its replacement, the interface it provides to the TM and Active Ferry modules should be clear and concise and the E/D itself should be well structured.

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.

The primitive specification part defines the primitives available and their parameters. It describes what the TM is allowed to send and receive. 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 primitive specification part should be defined as completely as possible. It should include all possible primitives and all the parameters for the primitives, even those that are not supported by the particular IUT. 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 IUT has to be tested, only the encoding/decoding part needs to be changed.

To minimize the effort in replacing the E/D module, encoding and decoding formats could be specified externally instead of being coded into the E/D module. One possible solution is to build an interpreter that would accept PDU specifications in some format similar to that of the PDU definition part of TTCN-GR [14]. The primitives, parameter names for each primitive, parameter length and the allowable range for each parameter could be listed in the PDU definition part. Each IUT would now have its own PDU definition part from which its E/D could be automatically generated.

4 Implementation of the Active Ferry Clip

One of our main goals was to study the issues involved in implementing a FCTS in different environments. Three different environments were chosen. Interesting design and implementation issues were encountered in all the environments. Certain parts of the system such as the buffering scheme used were dictated by the environment. However, the proposed scheme for structuring the ferries (described in sections 2 and 3) was used consistently and proved to be extremely useful in all the environments in which the FCTS was built.

4.1 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 and contains three Motorola 68000 CPUs, two of which are available for implementing the FCTS. Even though memory is partitioned between the three CPUs, a CPU can access another CPU's memory partition. Communication between the CPUs is via inter-CPU messages. The operating system is event driven. Events are triggered by an incoming frame, a keyboard entry, a timer expiration or an inter-CPU message.

Two protocol testers were used - one for the test system and the other for the SUT. The two PTs were connected via an RS-232C serial cable. The TM, E/D and Active Ferry were implemented as a single process on one CPU of the test system. The Passive Ferry and IUT reside on different CPUs in the SUT.

4.1.1 The Test Manager and Encoder/Decoder

Keyboard events are processed immediately. Frame events are passed to the Active Ferry for further processing by the Active Ferry LMAP module. Timeout events as well as decoded test data packets received from the Passive Ferry are placed in the TM's event queue.

The test language used is an enhanced version of *Idacom test language* (ITL) [13]. It is a state-based language which has the control and expression evaluation features of Forth. Each state defined in the test script is associated with a list of events expected to be received, the corresponding action to be taken for each event, and an optional state change command. An example of a test script written in ITL and the corresponding test log generated are given in Figures 3(a) and 3(b) respectively.

A Forth procedure is defined in the E/D for each primitive that can be sent. Invoking this procedure either from the keyboard or from the test script would cause a 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 will invoke the FTMP to clear the output packets if the buffers of the Active Ferry become full.

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

4.1.2 The Active Ferry Clip

The Active Ferry was structured into two modules - Active Ferry FSM and LMAP. The Active Ferry FSM module maintains a finite set of buffers. The buffers are used to store packet fragments before they are accepted by the FTMP for transmission to the Passive Ferry.

Fragmentation and reassembly of data packets is required since the FTMP places a limit on the maximum size of a packet it will accept. The Active Ferry accepts data packets of arbitrary size from the TM and fragments it so that each fragment can fit into an FTMP packet. The packet fragments are reassembled at the Passive Ferry. The Passive Ferry performs similar fragmentation of data packets received from the IUT

before transmitting them to the Active Ferry.

The underlying FTMP could refuse to accept data from the Active Ferry FSM once the FTMP's local buffers fill up. Since the Active Ferry buffers have finite capacity, some mechanism is needed to inform the Test Manager when its local buffers are full. The scheme used is described below.

When the Test Manager sends a packet to the Active Ferry for transmission to the Passive Ferry, the Active Ferry FSM calculates how many packet fragments will result when the data packet is fragmented. If there are enough buffers to store all the fragments, the data packet is accepted, fragmented and stored in the Active Ferry FSM buffers for later transmission to the Passive Ferry. Otherwise, the Test Manager is informed that the packet cannot be accepted in which case the Test Manager simply retries at a later time. The Active Ferry FSM attempts to send as many data fragments as it can to the Active Ferry LMAP module, thereby clearing the buffers as soon as possible for more packets from the Test Manager. In this fashion, a fast Test Manager will not swamp a slow Active Ferry with data.

A special ASP called FM-FLUSH was introduced. Its specific purpose is for the Test Manager to call the Active Ferry periodically, instructing it to send any waiting data thereby clearing the Active Ferry FSM buffers.

From our experience in using Unix INET stream sockets and X.25 as the FTMP, we observed that the flow control schemes used by different protocols and the methods for accessing them vary a great deal. For example, with Unix INET domain sockets there was no convenient way to query the protocol for the status of the socket buffers. A "write" operation to the socket would have to be made. The operation would block if the socket buffers were full (it would fail if the "write" operation was made non-blocking). In our version of X.25 Packet Layer, it was possible to query the status of its buffers before a "write" operation was issued.

For uniformity and to minimize the code that needs to be rewritten for different FTMPs, it was decided to implement a simple flow control scheme at the Ferry protocol level. The scheme is independent of the flow control facilities offered by the underlying FTMP. The tradeoff is that the functionality of the Active and Passive Ferries increases thereby increasing the size of their code. Hence, the flow control scheme should be simple and compact.

The scheme used was as follows. The "reserved bits" in the control field of a FY-CNTL PDU [4] were used to implement two additional control functions, namely

"flow control on" and "flow control off". When the Passive Ferry discovers its local buffers have filled past a certain "high-water mark" it sends a "flow control on" control message to the Active Ferry. The Active Ferry then ceases to send any further data to the Passive Ferry. However, it continues to accept data packets sent to it by the Passive Ferry, thereby allowing the Passive Ferry to clear its buffers. When the Passive Ferry's local buffers clear below a "low-water mark", it sends a "flow control off" control message to the the Active Ferry, informing it to resume sending data packets.

The "high-water mark" must be chosen carefully. This is because before the "flow control on" message gets to the Active Ferry, data may still be sent to the Passive Ferry which must be ready to accept them. Hence, the "high-water mark" must be chosen so that there will still be enough space in the Passive Ferry's local buffers to accept data until the "flow control on" message gets to the Active Ferry. The additional buffers required can usually be estimated quite easily as follows:

E = Maximum packet delay from Passive to Active Ferry.
B = Baud rate of the line.
P = Maximum number of bits in a packet fragment.
N = Number of buffers required above the "high-water mark".

$$N = \lceil B \times E/P \rceil$$

Generally, there is no problem in estimating "N". Only in the case of long haul networks where the end-to-end delay has a high degree of variance does "E" become tricky to estimate. Overestimating "E" simply results in some buffer space not being used sometimes, but underestimating it could result in overflowing the Passive Ferry's local buffers.

4.1.3 The Passive Ferry

The Passive Ferry was implemented on one of the two available CPUs in the SUT. The other available CPU was used to run the IUT. The structuring was done in this manner so that the Passive Ferry would not interfere with the operation of the IUT. The Passive Ferry was structured into three modules - Passive Ferry FSM, LMAP and the IUT Interface module. The IUT Interface module resided on the same CPU as the IUT. Communication between the Passive Ferry FSM and the IUT Interface modules was by means of inter-CPU messages.

Three sets of buffers were maintained in the Passive Ferry. The first set was to store data packets received from the Active Ferry before they could be worked on. The other two sets of buffers were used to store packets to be sent to the Active Ferry - one set for data packets and the other for control packets. The reason for separating the "data" and "control" buffers was so that higher priority could be given to packets in the control buffer. This would ensure that a "flow control on" control message could be sent before any waiting data packets.

4.1.4 The Lower Mapping Module

The LMAP module resided in the same CPU as the Passive Ferry FSM module. Communication between these two modules was achieved via the FT-ASPs (Appendix C).

In the MPT version of X.25, the user specifies the frames and packets to be generated. For example, a "SABM" command would send across a SABM frame, and a "CALL" would send a "call request" packet to the peer entity. It is LMAP's responsibility to map the FT-ASP's into the MPT X.25 commands and vice-versa. For example, an FT-CONNreq requires the Active Ferry LMAP module to send a SABM frame and wait for a UA frame after which it must send a CALLreq packet and wait for a CALLcnf packet before returning a FT-CONNcnf to the Active Ferry FSM module. This is complicated further by the fact that other frames and packets could be received by the Active Ferry LMAP module during the process of setting up the link. These must be handled by the LMAP module. With this interface, the mapping between the Ferry ASPs and the FTMP commands is not trivial. To best realize this mapping it was decided to implement the LMAP module as a finite state machine.

In the OSI-PTE version of the Test System, the X.25 used as the FTMP provides the user with a set of X.213 ASPs [9]. As an example, the user could invoke the N_CONNreq ASP to set up a network connection. If the peer entity accepts the connection, a N_CONNcnf ASP would be returned to the user. With this interface, there is a one-to-one mapping between the Ferry and FTMP ASPs. In this case, the LMAP module is very simple, and constitutes only a small portion of the overall size and complexity of the Active and Passive Ferries. An example of the mapping between the connection oriented Network service primitives (X.213) and the Ferry service primitives can be found in [4].

4.1.5 The IUT Interface Module

In the MPT environment, the IUT interface was placed together with the IUT on a single CPU. The IUT Interface was compiled on top of the IUT, and was divided into three parts: the Passive Ferry interface, an input handler and an output handler for each layer.

The Passive Ferry interface is essentially an arbiter for incoming events. Several handlers are installed in the interface for event handling. Simple message passing IPC is used to communicate between the Passive Ferry FSM and the IUT interface. The Passive Ferry FSM can send two type of messages to the IUT interface, namely initialize-IPC and incoming-packet.

When the Passive Ferry has a packet for the IUT, it sends the connection point ID and a buffer address with the incoming-packet message. The handler specified in the connection point ID is then called to send the packet to the correct SAP of the IUT. In this implementation, only one handler, the input handler was installed. It simply puts the packet into the IUT's buffer, and frees the buffer allocated by the Passive Ferry by replying to the incoming message.

Whenever the IUT has output, the output handler is invoked. It simply allocates a common buffer and messages the Passive Ferry to handle the packet.

The entire IUT interface was implemented in about 150 lines of Forth code.

In the three environments in which the Ferry Clip Test Systems was built, the test system and the SUT had the same CPU architecture. Hence, the basic data types had the same representation on both the test system and the SUT. A more general situation would be where the test system and the SUT have different architectures. In such an environment the representation of even the basic data types could differ. It would be useful to adopt some standard data representation such as ASN.1 [11]. This would save considerable amount of work which would otherwise have to be performed to convert between different data types.

4.2 The Unix Environment

Unix is a powerful and widely used operating system. The Unix environment used to develop the FCTS consisted of a pair of SUN 3/50 workstations connected via an ethernet local area network. Each workstation ran a version of Unix BSD 4.2 (Sun/OS 3.2).

The FCTS was implemented as a set of three processes. The Test Manager, E/D and Active Ferry were structured into a single process running on one workstation. The Passive Ferry and IUT were implemented as two processes running on the other workstation.

Communication between the ferries was achieved using Unix stream sockets in the internetwork domain [5]. This particular type of socket was chosen since it best fit the FTMP requirements (see section 2.3).

4.3 The OSI-PT Environment

The OSI-PTE [15] [16] is a new environment for the implementation and testing of computer protocols designed to run on an Idacom PT. It is a realization of the OSI Reference Model [7] within a single operating system process for efficiency. Besides providing an operating environment which is close to the OSI reference model, the OSI-PTE also allows the incorporation of a Test Manager into the test system. The system supports all test methods suggested by ISO as well as passive monitoring, logging and analysis capabilities.

The OSI-PTE is an event driven system. Each protocol is structured as a single or a group of protocol entities. Communication between protocol entities is through an event-posting scheme whereby one protocol entity posts an event to another protocol entity. The important events are incoming ASPs from the upper or lower SAP and timer expiry events.

When a protocol entity receives an ASP event, it also receives the parameters that correspond to that ASP. Overall the system resembles the OSI Reference Model much more closely than the other two environments discussed.

Only the Passive Ferry was built in this environment. The IUT Interface was subdivided into a "lower" and an "upper" IUT Interface which performed the mappings for the lower and upper SAPs of the IUT respectively. The Passive Ferry consisted of three entities, namely the Passive Ferry FSM, the Lower and the Upper IUT Interfaces. The FTMP (X.25) and the IUT were in turn independent protocol entities.

5 IUT

5.1 The NULL IUT

The simplest IUT possible is a protocol which returns data sent to it without altering the data in any way. Our version of the NULL IUT simply returns test data received by its upper SAP through its lower SAP and vice versa.

The NULL IUT allows the IUT interface to be tested. This is not the same as using the ferry *loopback mode* which only tests the Active and Passive Ferries as no data is sent to the IUT. Hence, it is useful to use a NULL IUT to confirm that the IUT Interface is functioning correctly before testing the actual IUT.

5.2 Some IUTs That Were Tested

The FCTS is not restricted by the fact that a particular IUT may only present the tester with a single SAP. Greater control and observation of the IUT could be achieved if both SAPs are accessible, but the IUT can still be tested through a single SAP.

Two versions of the X.25 packet layer have been used as the IUT, one in the Unix environment and the other in the MPT environment.

The Unix version of the X.25 packet layer was a single Unix process that allowed the tester to connect to it through a Unix socket. The IUT Interface was fairly trivial since little format change was required to bring the data in a FY-DATA PDU into the form the IUT required. Access was possible to both the upper and lower SAPs of the IUT.

In the MPT environment, the X.25 packet layer IUT ran on a single CPU in the MPT . The IUT Interface was once again quite simple since the E/D module encoded the test data in a format that the IUT could accept. However, the Packet layer IUT for the MPT does not have a clearly defined upper SAP, since it was taken from a reference emulation of X.25 designed to work with a test responder. We chose not to test the upper SAP at all, and were able to adapt existing test scripts with minimal modifications. A sample test script and the corresponding test log are given in Figure 3.

6 Conclusions

This paper has presented some interesting issues encountered in building a Ferry Clip based Test System (FCTS). A method was presented for structuring the Active and Passive Ferry Clips so as to minimize the effort required to test different IUTs as well as to use different FTMPs. An FTMP independent flow control scheme as well as the use of a standard data representation scheme such as ASN.1 was also suggested to increase the portability of the FCTS.

The FCTS has been developed on three different environments as part of the UBC/Idacom research project to develop the next generation of protocol test systems. Our experience has convinced us that the Ferry Clip approach is a powerful and useful technique for protocol testing. This technique has also been submitted to ISO and is currently under study to replace the Ferry Concept in Annex B of ISO DP9646-4 [6].

Acknowledgement

The authors would like to acknowledge the financial support from the Natural Sciences and Engineering Research Council of Canada and Idacom Electronics in this work, and useful discussions with Dr. Dave Rayner.

References

[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*, Submitted for publication, June 1988.

[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*, Submitted for publication, January 1989.

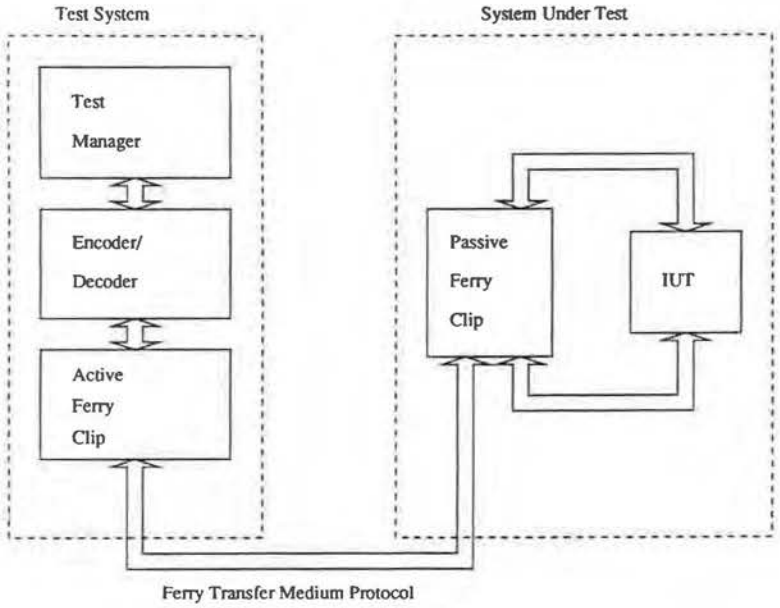

Figure 1 - Schematic Diagram of Ferry Test System

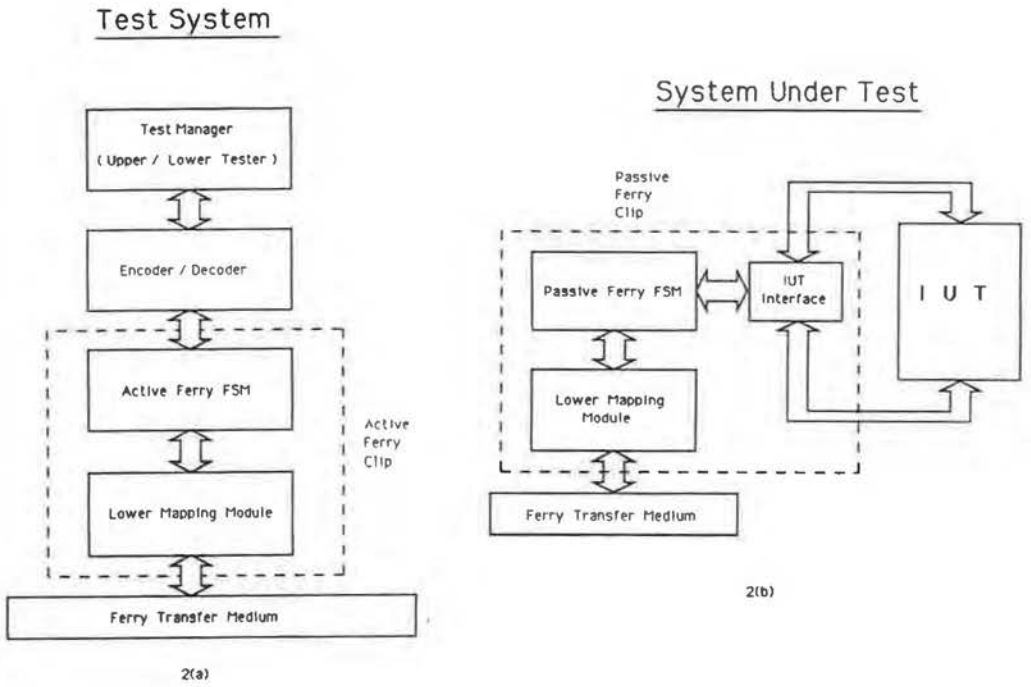

Figure 2 - The Ferry Clips

```
( SAMPLE FERRY CLIP ITL TEST SCRIPT                                    )
(                                                                      )
( PURPOSE : Send a Restart Request, expect a Restart Confirm.          )
(           Send a Call Request, expect a Call Confirm.                )
(           Send a Clear Request, expect a Clear Confirm.              )
(                                                                      )
( ENTER FUNCTION KEY CF1 TO START THE TEST                             )

( Initialize Test Manager )
TCLR

( State 0 : Wait for CNTL-F1 to be pressed, then send Ferry Connect Request )
0 STATE{ FK_CF1 ACTION{
                    PRINT_TIME
                    " TEST STARTING" BTYPE WCR 20 COUNTER1 !
                    F_CONN 1 NEW_STATE
                }ACTION
  }STATE

( State 1 : Wait for Ferry Connect Confirm, send Restart Request )
1 STATE{  F_CONN_CONF 1 ?RX ACTION{
                                PKT:RESTART
                                2 NEW_STATE
                            }ACTION
          OTHER_EVENT       ACTION{
                                " VERDICT : " RTYPE " INCONCLUSIVE" YTYPE WCR
                                TM_STOP
                            }ACTION
  }STATE

( State 2 : Wait for Restart Confirm, send Call Request )
2 STATE{ RESTARTcnf 1 ?RX ACTION{
                              PKT:CALL 3 NEW_STATE
                          }ACTION
         OTHER_EVENT      ACTION{
                              " VERDICT : " RTYPE " FAILED" RTYPE WCR
                              F_DISC TM_STOP
                          }ACTION
  }STATE

( State 3 : Wait for Call Confirm, send Clear Request )
3 STATE{ CALLcnf 1 ?RX ACTION{
                           PKT:CLEAR
                           4 NEW_STATE
                       }ACTION
         OTHER_EVENT   ACTION{
                           " VERDICT : " RTYPE " FAILED" RTYPE WCR
                           F_DISC TM_STOP
                       }ACTION
  }STATE

( State 4 : Wait for Clear Confirm, send Ferry Disconnect Request )
4 STATE{ CLEARcnf 1 ?RX ACTION{
                            F_DISC
                            " VERDICT : " RTYPE " PASSED" BTYPE WCR
                            " TEST FINISHED" BTYPE WCR TM_STOP
                        }ACTION
         OTHER_EVENT    ACTION{
                            F_DISC
                            " VERDICT : " RTYPE " FAILED" RTYPE WCR
                            " TEST FINISHED" BTYPE WCR TM_STOP
                        }ACTION
}STATE
```

Figure 3(a) - Sample ITL Test Script

```
PRESS CF1 TO START
OCTOBER 13 1988                    17 : 39 : 45
TEST STARTING
SEND FERRY CONNECT
RECV FERRY CONNECT CONFIRM
SEND LOWER SAP PKT:RESTARTreq
RECV LOWER SAP PKT:RESTARTcnf
SEND LOWER SAP PKT:CALLreq
RECV LOWER SAP PKT:CALLcnf
SEND LOWER SAP PKT:CLEARreq
RECV LOWER SAP PKT:CLEARcnf
SEND FERRY DISCONNECT
VERDICT : PASSED
TEST FINISHED
```

Figure 3(b) - Test Log

State	Event	Action	Next State
idle	FM-CONN req FM-DISC req	FT-CONN req none	connecting idle
	FM-CNTL req FD-DATA req	FM-DISC ind	idle
	FT-DISC ind FT-ERROR ind	none FT-DISC req	idle
connecting	FM-DISC req	FT-DISC req	idle
	FT-CONN cnf FT-DISC ind FT-ERROR ind	FM-CONN cnf FM-DISC ind EXCEPTION	connected idle idle
connected	FM-DISC req	FT-DISC req	idle
	FM-CNTL req FD-DATA req	FT-DATA req(FY-CNTL) FT-DATA req(FY-DATA)	connected
	FT-DATA ind	P1: FD-DATA ind P2: control actions	connected
	FT-DISC ind FT-ERROR ind	FM-DISC ind EXCEPTION	idle

Notes:

- In any state, the action and transition taken for events not listed in the table are the same as those listed for the FT-ERROR ind event, *e.g.* if an FM-CNTL req is received in the *connecting* state, the EXCEPTION action should be taken and the state should change to *idle*.
- EXCEPTION indicates the dual actions FT-DISC req and FM-DISC ind.
- P1 (predicate 1) – the FT-DATA received is an FY-DATA PDU.
- P2 (predicate 2) – the FT-DATA received is an FY-CNTL PDU; action is to process FY-CNTL flag bits and generate FM-CNTL cnf.

Appendix A - State Table for Active Ferry

FT-CONNECT request	(FT-CONN req)	[test system only],
FT-CONNECT indication	(FT-CONN ind)	[SUT only],
FT-CONNECT response	(FT-CONN rsp)	[SUT only],
FT-CONNECT confirm	(FT-CONN cnf)	[test system only],
FT-DISCONNECT request	(FT-DISC req),	
FT-DISCONNECT indication	(FT-DISC ind),	
FT-DATA request	(FT-DATA req),	
FT-DATA indication	(FT-DATA ind), and	
FT-ERROR indication	(FT-ERR ind).	

Appendix C - FT-ASPs

State	Event	Action	Next State
idle	FT-CONN ind	P1: FT-CONN rsp P2: FT-DISC req	connected idle
	FT-DISC ind FT-ERROR ind	none FT-DISC req	idle
	FD-DATA req	none	idle
connected	FT-DISC ind FT-ERROR ind	none FT-DISC req	idle
	FT-DATA ind	P3: FT-DATA req (loop back) P4: FD-DATA ind P5: control actions	connected
	FD-DATA req	P6: FT-DATA req(FY-DATA) P3: none	connected

Notes:

- In any state, the action and transition taken for events not listed in the table are the same as those listed for the FT-ERROR ind event.
- P1 (predicate 1) – the incoming FT-CONN ind is acceptable.
- P2 – the incoming FT-CONN ind is unacceptable.
- P3 – the passive ferry clip is in loop-back mode.
- P4 – received data is FY-DATA PDU and the passive ferry clip is not in loop-back mode.
- P5 – received data is FY-CNTL PDU and the passive ferry clip is not in loop-back mode. Perform appropriate control actions and generate FY-CNTL (using FT-DATA req) back to active ferry.
- P6 – the passive ferry clip is not in loop-back mode.

Appendix B - State Table for Passive Ferry

FM-CONNECT request	(FM-CONN req),
FM-CONNECT confirm	(FM-CONN cnf),
FM-DISCONNECT request	(FM-DISC req),
FM-DISCONNECT indication	(FM-DISC ind),
FM-CONTROL request	(FM-CNTL req), and
FM-CONTROL confirm	(FM-CNTL cnf).

Appendix D - FM-ASPs

FD-DATA request	(FD-DATA req), and
FD-DATA indication	(FD-DATA ind).

Appendix E - FD-ASPs