CONFORMANCE TESTING OF OSI PROTOCOLS:

The Class 0 Transport Protocol As An Example

By

TIAN KOU

B.Sc., Northwest University, 1976
B.Sc., University of British Columbia, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1987

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.

Tian Kou

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date August, 19, 1987

Abstract

This thesis addresses the problem of conformance testing of communication protocol implementations. Test sequence generation techniques for finite state machines (FSM) have been developed to solve the problem of high costs of an exhaustive test. These techniques also guarantee a complete coverage of an implementation in terms of state transitions and output functions, and therefore provide a sound test of the implementation under test. In this thesis, we have modified and applied three test sequence generation techniques on the class 0 transport protocol. A local tester and executable test sequences for the ISO class 0 transport protocol have been developed on a portable protocol tester to demonstrate the practicality of the test methods and test methodologies. The local test is achieved by an upper tester residing on top of the implementation under test (IUT) and a lower tester residing at the bottom of the IUT. Tests are designed based on the state diagram of an IUT. Some methodologies of parameter variations have also been used to test primitive parameters of the implementation. Some problems encountered during the implementation of the testers and how they are resolved are also discussed in the thesis.

Contents

List of Figures

Acknowledgement

I would like to thank my supervisor Dr. Son Vuong for his guidance and advice on this thesis and Dr. Sam Chanson for his comments and careful reading of this thesis.

Many thanks go to Jean-Marc Serre who has supplied the implementation under test and has given me much helpful advice through electronic mail.

I would also like to thank Mark Giesbrecht for his many hours of careful reading of the manuscript and providing excellent criticism on writing and presentation.

Chapter 1

Introduction

1.1 Motivations and Contributions

Conformance testing is an important stage in the communication protocol development process. A new protocol implementation should be sufficiently tested for conformance with the specification.

In 1977, the International Organisation for Standardisation (ISO) developed a model for Open System Interconnection (OSI). The objective was to provide a common basis for the coordination for standards developed for the purpose of system interconnection. This model defines a seven layer protocol architecture for communication systems. The main goals of the layered approach were to develop complicated communication systems hierarchically and to decompose them into manageable parts. Each layer performs a related subset of functions required to communicate with another system. It relies on the next lower layer to perform more primitive functions and

to conceal the details of those functions by providing services to the next higher layer. Since it allows independent development and implementation of each layer, it gives a conceptual framework in which international teams of experts can work productively and independently on the development of standards for each layer. This approach has been widely accepted by most communication software developers and researchers. A detailed introduction for each of the seven layers can be found in [Tan81,Sta85].

The transport layer (layer 4), and above, of the ISO reference model are generally referred to as the higher layers. The transport layer ensures that data units are delivered error-free, in sequence, and with no losses or duplicates. The transport layer is the keystone of a computer communications architecture. It provides the basic end-to-end services of transferring data between users. Confidence in the transport layer implementation is crucial for the whole concept of computer communications. Testing is the most important element in developing this confidence, and hence is born the necessity for protocol implementation testing.

Protocol implementation testing is a major task in the development cycle for communication software. It is also a relatively new area in computer communications. The pioneering work can be traced back to the early 1980's when the National Physical Lab (NPL) in England [Ray82] and RHIN project in France [Ans82] did the initial work in testing. The

implementation under test is considered as a "black box" since the source code generally cannot be assumed to be accessible. Only the service primitives at the boundaries are available and observable to the testers. The ISO specifications of the layered approach to communication protocols are well advanced. Since each layer can be developed independently, the testing of that layer can be carried out without consideration of other layers, even the lower layers. This is commonly referred to as a local test [ISO86]. Local testing on a communication protocol implementation can be achieved by an upper tester residing on top of the implementation under test (IUT) and a lower tester residing at the bottom of the IUT. The control and observation points are at the service boundaries above and below the N-entity under test (single layer IUT). The test events are specified in terms of N-ASPs (Abstract Service Primitive) above the IUT, and (N-1) ASPs and N-PDUs (Protocol Data Unit) below the IUT.

The selection of test scenarios is the major problem for testing. Exhaustive testing is prohibitively costly. Techniques for a complete coverage of all possible cases of interactions, without an exhaustive approach, have been researched for many years. The PW method of [Cho78], the checking sequence method of [Gon70] and [Koh78], and the transition tour method of [NT81] were initially developed for circuit testing. They can be adopted for protocol implementation testing if the implementation is considered as

a black box and only the service primitives are observable. These testing techniques assume the specification can be described as a finite state machine (FSM). Testing can be designed based on the state diagram of an IUT. Test sequence generation techniques for FSMs solve the problems of high costs and basic intractibility of an exhaustive test. These techniques also guarantee a complete coverage of an implementation in terms of state transitions and output functions, and therefore provide sound confidence in the implementation under test.

We have modified the above algorithms for correctness and performance improvement. These modified test sequence generation techniques have been applied to an implementation of the ISO class 0 transport protocol on a portable protocol tester. Some methodologies of parameter variations have also been applied on primitive parameters of the implementation.

1.2 Thesis Outline

In Chapter 2, ISO active test methods are introduced. The functions, application areas, observation points and events of each test method are also discussed. Chapter 3 provides three modified test sequence generation techniques. The modified algorithms and examples are presented in an easy-to-understand style. The improvements on the performance of the modified algorithms are presented. In Chapter 4, the approaches used

in testing an implementation of the ISO class 0 transport protocol are presented. The details of the problems encountered and how they were solved can be found in this chapter too. Chapter 5 provides the test results, analysis, and remarks on the IUT. In Chapter 6, conclusions and future directions in conformance testing are indicated.

Chapter 2

Test Methods

In this thesis we are only concerned with the active abstract test methods. Passive test methods, methods which only observe the activities of the IUT, will not be discussed here. Active abstract test methods can be divided into categories according to the control and observation of events. This is because test suites are specified in terms of the control and observation of events. Once an abstract test method is decided, the points of control and observation are clearly defined. In this chapter, we will briefly review the active test methods and their functions and usages. A detailed discussion can be found in [ISO86].

2.1 Single Layer Test Method for Single Layer IUTs

This category can be further divided into the following sub-categories:

1. The Local Single layer Test Method (LS)

2. The Distributed Single layer Test Method (DS)

3. The Coordinated Single layer Test Method (CS)

4. The Remote Single layer Test Method (RS)

In the Local Single layer Test Method, test events are specified in terms of the N-ASPs above the single layer IUT, and the (N-1)-ASPs and N-PDUs below the IUT. In other words, the upper tester will be responsible for the N-ASPs and the lower tester will be responsible for the (N-1)-ASPs and the N-PDUs. Figure 4.1 shows a local test method. This is the test method used in this thesis. In our case, N is the transport layer and N-1 is the network layer. A detailed discussion will be given in Chapter 4.

2.2 Multi-layer Test Methods for Multi-layer IUTs

The assumption on this method is that the combined behaviours of the multi-layered IUTs must be know. Unlike the single layer test method, the multi-layer test methods test a multi-layer IUT as a whole and do not control and observe the inter-layer boundaries within the IUT.

Again, this category can be further divided into the following sub-categories:

1. The Local Multi-layer Test Method (LM)

2. The Distributed Multi-layer Test Method (DM)

3. The Coordinate Multi-layer Test Method (CM)

4. The Remote Multi-layer Test Method (RM)

2.3 Single-layer Testing for Multi-layer IUTs or SUTs (System Under Test)

Thess methods are generally referred to as the Embedded Methods. The ideas of these test methods are to test a multi-layered IUT layer by layer from top to bottom, but without specifying control or observation at service boundaries within the multi-layer IUT.

Similar to the above test methods, this category can be further divided into the following four sub-categories:

1. The Local Single-layer Embedded Test Method (LSE)

2. The Distributed Single-layer Embedded Test Method (DSE)

3. The Coordinated Single-layer Embedded Test Method (CSE)

4. The Remote Single-layer Embedded Test Method (RSE)

A detailed descriptions of these test methods can be found in [ISO86].

The test method used in this thesis is the local single-layer test method and is discussed in Chapter 4.

Chapter 3

Test Sequence Generation

3.1 The Finite State Machine Model

3.1.1 Definitions

A finite state machine can be represented as a quintuple:

$$M = \{S, I, O, T, A\} \tag{3.1}$$

where

S : a set of n states, $\{s_1, s_2, ...s_n\}$;

I : a set of m inputs, $\{i_1, i_2, ...i_m\}$;

O : a set of p outputs, $\{o_1, o_2, ...o_p\}$;

T : state transition function which takes a state and an input and produces a new state: $S \times I \longrightarrow S$;

A : action function which takes a state and an input and produces an output: $S \times I \longrightarrow O$.

3.1.2 Representation Of A Finite State Machine

The relationship between the inputs, outputs, and states can be described by state tables or a state diagram. The state tables of a finite state machine contain two tables: the next-state table and the action table. Each table has n rows, one for each state, and m columns, one for each input. The rows are numbered by the states and the columns are numbered by the inputs. The content of row i, column j of the tables are the next-state or the action of applying input i_j to state s_i of the machine. Figure 3.3 shows the state table of the alternating bit receiver's protocol. This protocol expects data 0 and 1 alternatively and acknowledges them accordingly. Two possible states of the machine are : expect1 and expect0. There are three possible input events : D0 (data 0), D1 (data 1), and Err (anything else). Three actions can be taken depending on the input and the current state. The actions are: send A0 (acknowledge D0), send A1 (acknowledge D1), and "no action" which is represented as a "/" in Figure 3.3 meaning the machine does not do any thing to react the input event. A finite state machine can also be represented by a state diagram. In the state diagram of a finite state machine there are n nodes, one for each state, and a maximum of m

arcs coming out of each node. In the models discussed in this thesis , there are exactly m arcs coming out of each node. A total of $m \times n$ arcs are assumed in our models. The arcs are marked with the inputs and outputs. They are pointing to the next states. In Figure 3.2 , a state diagram of the alternating bit receiver's protocol is given.

3.1.3 The Assumptions On The Finite State Machines

The finite state machine model considered in this thesis is assumed to have the following characteristics:

1. The machine is Mealy-type. This means the output is a function of the state and the input. It is dependent on both the current state and the input [Koh78].

2. The machine is reduced. A reduced Mealy machine has the smallest number of states. In other words, it does not possess two equivalent states. This condition is also known as minimal. [Sal69,Gil62]

3. The machine is completely specified. That is, all the possible receptions should be specified. In our case, at any state, any input interaction is possible. [Vuo83]

4. The machine is strongly connected, i.e. an input sequence exists which passes machine M from any state s_i to any given state s_j. Thus, a

strongly connected machine is a machine in which every state can be attained regardless of the past history of the machine. [Gil62]

3.2 The Transition Tour Method

A transition tour of a finite state machine is an input sequence which traverses every transition path of the machine at least once.

It is obvious that a transition tour can be obtained by applying random inputs to the FSM until every transition path is traversed. As a consequence, a transition tour generated by this method contains many redundant input sequences, and the generation of such a tour is costly and ungraceful.

In [NT81], Sachio Naito and Masahiro Tsunoyama provided an algorithm to generate transition tours and an algorithm to reduce the redundant inputs. In this section, we will introduce a modified version of the algorithm presented in [NT81]. In the modified version, the performance is improved as follows:

1. The procedure does not terminate before every transition path is traversed. In other words, the termination condition, which is the number of distinguishing events, works properly.

2. Some distinguishing events will not be eliminated by the reduction procedure.

3. The length of a redundant input is shorter.

4. The possibility of the occurrence of repeated events is smaller.

The details of these advantages will be discussed after the modified algorithm is presented.

An example of the Alternating Bit Protocol will be given and the applications to the ISO class 0 transport protocol are discussed. The test sequences generated by using this algorithm are also provided.

3.2.1 Notation and Definitions

We represent a sequence of inputs by superscripting them. Therefore a sequence of k inputs can be represented as $(i^1, i^2, i^3, ..., i^k)$. The superscript indicates the sequence number of an input in a series of inputs.

We use a pair (s^j, i^j) to represent the application of input i^j to state s^j. A sequence of k (state, input) pairs $\begin{pmatrix} s^1, & s^2, & ..., & s^k \\ i^1, & i^2, & ..., & i^k \end{pmatrix}$ shows the behaviour of a machine with initial state s^1 and input sequence $(i^1, i^2, i^3, ..., i^k)$. For example, the following sequence is the behaviour of the alternating bit receiver's protocol when the input sequence: $(D_0, D_1, D_1, E, D_1, E, D_0)$ is applied to an initial state expect0:

$$\begin{pmatrix} expect0 & expect1 & expect0 & expect0 & expect0 & expect0 & expect0 \\ D_0 & D_1 & D_1 & Err & D_1 & Err & D_0 \end{pmatrix}.$$

To pursue further discussion, the following definitions are required:

diff-input :

Input i^k is called a diff-input for cell (s^k, i^k), if (state, input) pair (s^k, i^k) is not equal to (state, input) pair (s^a, i^a) for $a = 1, 2, \ldots$ and $k-1$. In other words, a diff-input does not repeat any previous (state, input) pair in the sequence.

rep-input :

Input i^k is called a rep-input for cell (s^k, i^k), if (state, input) pair (s^k, i^k) is equal to (state, input) pair (s^a, i^a) for $a = 1, 2, \ldots$ or $k-1$. That is, a rep-input repeats one of the previous input cells.

redun-sequence:

The subsequence $\begin{pmatrix} s^{r+1}, & s^{r+2}, & \ldots, & s^{r+l} \\ i^{r+1}, & i^{r+2}, & \ldots, & i^{r+l} \end{pmatrix}$ is called a **redun-sequence** if input i^{r+1} and i^{r+l} are both diff-input and none of i^{r+2}, i^{r+3}, ..., and i^{r+l-1} is a diff-input.

A redun-sequence is a 'would be' redundant sequence since i^{r+2}, i^{r+3}, ..., and i^{r+l-1} repeat some of the previous inputs and they are candidats for removal. The algorithm for reducing the redundant input sequences will be introduced in the next section.

3.2.2 The Redundant Sequence Reduction Algorithm

If $\begin{pmatrix} s^{i+1}, & s^{i+2}, & \ldots, & s^{i+l} \\ i^{i+1}, & i^{i+2}, & \ldots, & i^{i+l} \end{pmatrix}$ is a redun-sequence, the redundant inputs can be removed by the following algorithm:

The Algorithm:

Step 1: Set r to $i+1$;

Step 2: Set s to $i+l$;

Step 3: Compare state s^r with state s^s.

Step 3.1 If $s^r = s^s$, then remove the subsequence $\begin{pmatrix} s^r, & s^{r+1}, & \ldots, & s^{s-1} \\ i^r, & i^{r+1}, & \ldots, & i^{s-1} \end{pmatrix}$, set r to $s+1$, go to Step 6.

Step 3.2 else go to the next step;

Step 4: Let s be $(s-1)$;

Step 5: Compare s and r

Step 5.1 If $s > r$, then go to Step 3.

Step 5.2 else let r be $r+1$, go to the next step.

Step 6: Compare r and $i+l$

Step 6.1 If $r < i+l$, then go to Step 2.

Step 6.2 else stop.

3.2.3 The Algorithm for Generating Transition Tours

In the transition tour generation algorithm, the abbreviations in 3.1 are used:

H	:	redun-sequence head pointer
T	:	redun-sequence tail pointer
C	:	counter for termination condition

Figure 3.1: Abbreviations used in Transition Tour Algorithm

The following is the modified version of the transition tour generation algorithm

The algorithm:

Step 1 : Set $H = 0$, $T = 0$, and $C = 0$.

Step 2 : Let $H = H + 1$.

Step 3 : Let $T = T + 1$.

Step 4 : Generate an input x^T.

Step 5 : If x^T is a rep-input and not all the transition paths from the current state are traversed, then discard x^T, and go to step 4. (Try to avoid traversed transitions if possible.)

Step 6 : Apply x^T to machine M.

Step 7 : If x^T is a rep-input, then go to step 3. (Keep looking for a diff-input) Else, go to next step.

Step 8: If $H = T - 1$, then go to step 10. (x^T is a diff-input and no rep-input between the two diff-inputs)

Step 9: If $T <> 1$, then Reduce the redun-sequence $\begin{pmatrix} s^H, & s^{H+1}, & ..., & s^T \\ i^H, & i^{H+1}, & ..., & i^T \end{pmatrix}$ by the reduction procedure. (First time, we don't have a redun-sequence.)

Step 10: Let $C = C + 1$. (Increment counter for total number of diff-inputs.)

Step 11: If $C < m \times n$, then let $H = T$, go to step 3, else terminate (Only when the total number of diff-inputs $= m \times n$, i.e. all the transitions are traversed, will the algorithm stop).

The idea of this algorithm is to try to find a new (i.e. never traversed) transition path (if any). If a repeated path has to be traversed, then apply the reduction procedure to reduce a redundant sequence.

All the x^Ts, except the reduced inputs, construct a transition tour.

3.2.4 The Improvements Over Existing Algorithms

The following improvements have been achieved in the modified algorithm:

1. In the modified version as apposed to the algorithm used in [NT81], an additional variable C is added. It keeps track of the number of diff-inputs. It is totally independent from the head (H) and the tail (T) of the redun-sequence. After adding this variable, the program will not terminate before all the transition paths are traversed.

2. If we use H as the counter and increment H by 1 after the reduction procedure is called, as the original algorithm does, some diff-inputs may be eliminated unnecessarily and will produce incorrect results in most cases. In the modified version, H is reset to the position of the tail of the previous redun-sequence. It is the beginning of the subsequent traversing and the beginning of another redun-sequence if any.

3. Moreover, in the old version, the algorithm may terminate before every transition tour is traversed. This is because the reduction procedure will not reduce all the rep-inputs in the redun-

sequence due to the characteristics of a finite state machine. Generally speaking, the length of the transition tour of a FSM is longer than $m \times n$ in most cases, where m is the number of inputs and n is the number of states.

4. By adding step 5 in the modified algorithm, the chance of traversing a traversed arc is reduced. The change guarantees that all the diff-inputs are traversed first, and therefore the transition tour is shorter and the algorithm is more efficient.

5. Step 8 and the condition of step 9 reduce the unnecessary calling of the reduction procedure. They eliminate the cases in which only one diff-input exists ($T = 1$) or one diff-input immediately follows another diff-input ($H = T - 1$). In the modified algorithm, the reduction procedure is called only when it is necessary. The reduced number of unnecessary calls to the reduction procedure can be as high as $m \times n$, which is the case where the minimal length of a transition tour is $m \times n$.

3.2.5 Examples

Figure 3.2 is the state diagram of the alternating bit receiver protocol and Figure 3.3 is the state tables of the alternating bit receiver protocol.

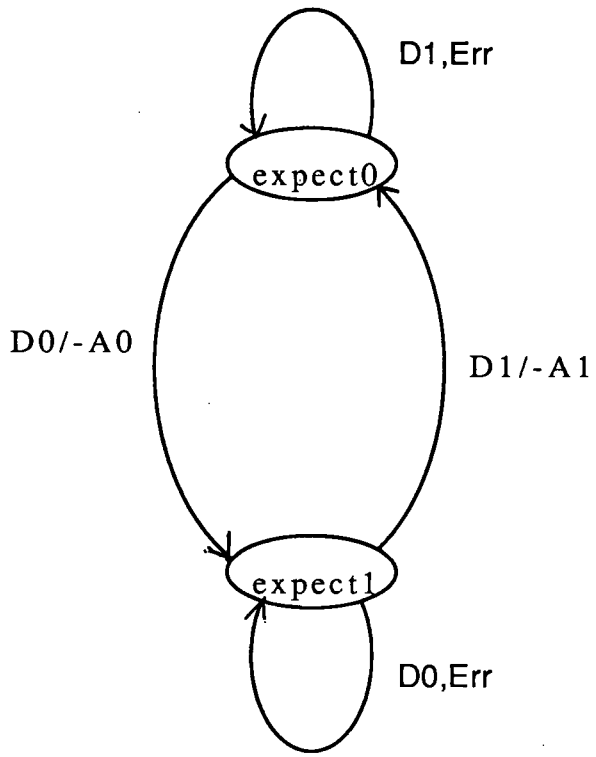


Figure 3.2: The State Diagram of The Alternating Bit Receiver Protocol

The following is the transition tour of the alternating bit protocol generated by the modified algorithm.

$$\begin{pmatrix} expect0 & expect1 & expect0 & expect0 & expect0 & expect1 & expect1 & expect1 \\ D_0 & D_1 & D_1 & Err & D_0 & D_0 & Err & \end{pmatrix}$$

In this transition tour there are a total of 7 inputs. This is the worst case which is only one input longer than the best case. In the best case, the length of the transition tour is $m \times n$, which is 6 in this

input / state	D0	D1	Err
expect0	1	0	0
expect1	1	0	1

a) next-state table of the alternating bit protocol

input / state	D0	D1	Err
expect0	-A0	/	/
expect1	/	-A1	/

b) action table of the alternating bit protocol

Figure 3.3: The State Tables of The Alternating Bit Protocol

protocol. If we let the transition tour end at the initial state, we can append one input event D1 to the end of the above test sequence. So, there will be 8 events in the tour as follows:

$$\begin{pmatrix} expect0 & expect1 & expect0 & expect0 & expect0 & expect1 & expect1 & expect1 \\ D_0 & D_1 & D_1 & Err & D_0 & D_0 & Err & D_1 \end{pmatrix}$$

In this sequence there are 4 subtours. A subtour is an input sequence which starts at the initial state and ends at the initial state [Sar85].

They are listed as follows,

S1: $\begin{pmatrix} expect0 & expect1 & expect0 \\ D_0 & D_1 & D_1 \end{pmatrix}$

S2: $\begin{pmatrix} expect0 & expect0 \\ D_1 & Err \end{pmatrix}$

S3: $\begin{pmatrix} expect0 & expect0 \\ Err & D_0 \end{pmatrix}$

S4: $\begin{pmatrix} expect0 & expect1 & expect1 & expect1 & expect0 \\ D_0 & D_0 & Err & D_1 & \end{pmatrix}$.

We have applied the algorithm to the class 0 transport protocol of the ISO standard. In Figure 3.4 we show the state diagram of the IUT which follows the class 0 transport protocol ISO standards. This state diagram has six states :

0-T-CLOSED
1-T-WFTRESP
2-T-WFNC
3-T-WFCC
4-T-OPEN
5-T-CLOSING

For each state there are a total of seventeen possible inputs. They are listed in Figure 3.5.

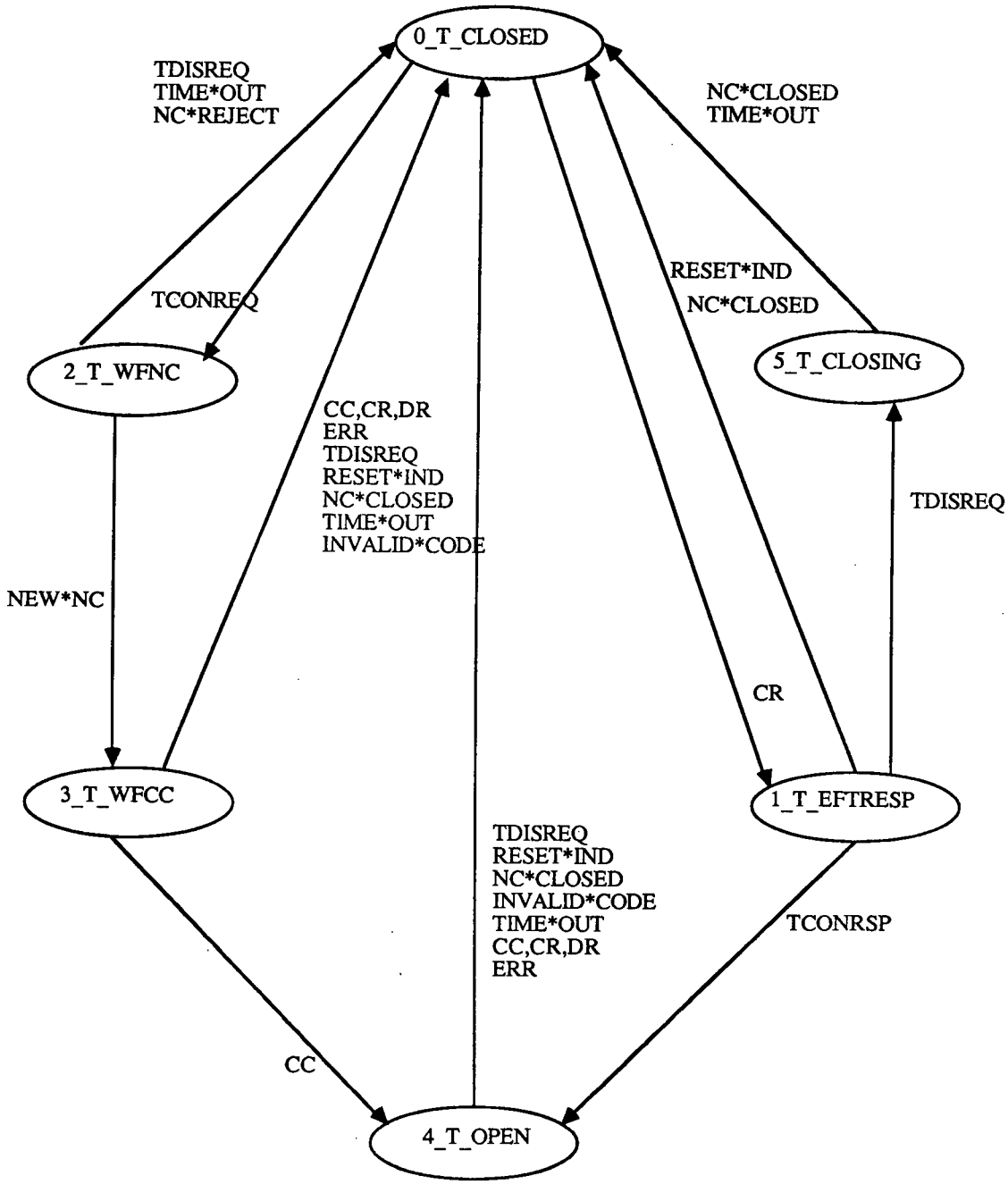


Figure 3.4: State diagram of Class 0 Transport Protocol

TCONREQ	CR*CODE	RESET*IND	NEW*NC
TCONRSP	CC*CODE	NC*CLOSED	EVALUATE*FLOW
TDATREQ	DT*CODE	TIME*OUT	TEVALFLOW
TDISREQ	DR*CODE	NC*REJECT	INVALID*CODE
ERR*CODE			

Figure 3.5: Inputs for Class 0 Transport Protocol

Note that all the "self-loops" are not shown in Figure 3.4. A self-loop is a transition which goes to the same state in response to an input. In Figure 3.4, the self-loops for each state contains all the remaining inputs which do not appear in the figure for that particular state.

Figure 3.6 shows the transition tour generated by the algorithm presented in the last section. The numbers are the states and the input below each state number is the input applied to that particular state. Self-loops are not shown in this sequence and they are applied to each state when it first appears in the transition tour.

The total length of this transition tour is 143 and there are 40 subtours.

If this state diagram was a *Eulerian* graph in which the number of outgoing arcs is equal to the number of incoming arcs, then the length of the transition tour would be $m \times n$ which is 102. The current length of the transition tour is 41 more than the minimal length.

It is also the shortest transition tour which can be generated

```
0          2          3          4          1          4
TCONREQ    NEW*NC     TDISREQ    CR*CODE    TCONRSP    TDISREQ

0          2          0          1          5          0
TCONREQ    TDISREQ    CR*CODE    TDISREQ    NC*CLOSED  TCONREQ

2          0          1          0          2          0
TIME*OUT   CR*CODE    RESET*IND  TCONREQ    NC*REJECT  CR*CODE

1          0          2          3          0          1
NC*CLOSED  TCONREQ    NEW*NC     CR*CODE    CR*CODE    TCONRSP

4          0          1          5          0          2
CR*CODE    CR*CODE    TDISREQ    TIME*OUT   TCONREQ    NEW*NC

3          0          2          3          0          2
DT*CODE    TCONREQ    NEW*NC     DR*CODE    TCONREQ    NEW*NC

3          0          2          3          0          2
ERR*CODE   TCONREQ    NEW*NC     INVALID*CODE TCONREQ  NEW*NC

3          0          2          3          0          2
RESET*IND  TCONREQ    NEW*NC     NC*CLOSED  TCONREQ    NEW*NC

3          0          1          4          0          1
TIME*OUT   CR*CODE    TCONREQ    CC*CODE    CR*CODE    TCONREQ

4          0          1          4          0          1
DR*CODE    CR*CODE    TCONREQ    ERR*CODE   CR*CODE    TCONREQ

4             0       1          4          0          1
INVALID*CODE CR*CODE  TCONREQ    RESET*IND  CR*CODE    TCONREQ

4          0          1          4          0
NC*CLOSED  CR*CODE    TCONREQ    TIME*OUT
```

Figure 3.6: Transition Tour Generated for Class 0 Transport Protocol

by the modified algorithm and it can be easily expanded to 152 without any redun-sequences. This is because in order to traverse the transitions form state 4-T-OPEN to state 0-T-CLOSED, two paths can be taken. The first one is $(0-\mathrm{T}-\mathrm{CLOSED} \rightarrow 2-\mathrm{T}-\mathrm{WFNC} \rightarrow 3-\mathrm{T}-\mathrm{WFCC} \rightarrow 4-\mathrm{T}-\mathrm{OPEN})$ and the second one is $(0-\mathrm{T}-\mathrm{CLOSED} \rightarrow 1-\mathrm{T}-\mathrm{WFTRESP} \rightarrow 4-\mathrm{T}-\mathrm{OPEN})$. We have chosen the second one in our transition tour, so it is 9 transitions shorter than choosing the first path.

3.3 The PW-Method

The PW-method is a testing strategy which is both valid and reliable. It is guaranteed to reveal any errors in the control structure. This method is based on a finite-state machine model and uses the same assumptions as the transition-tour method. There are three major steps in the PW-method [Cho78]. They are:

1. Construct P
   (a) Generate Testing Tree T.
   (b) Get P, which is the set of all the partial paths of T.
2. Construct W, the characterisation set (see below).
3. Construct $Z = W \cup P \cdot W \cup P^2 \cdot W \cup ... \cup P^{m-n} \cdot W$

An algorithm based on [Cho78] has been developed and will be presented in the following section.

An example of the Alternating Bit Protocol will be given and the applications on the class 0 transport protocol are also discussed. The test sequences generated by using this algorithm are also provided in the subsequent section .

3.3.1 The Algorithm

In order to generate set P, we construct the testing tree T first. In Figure 3.7, we introduce the abbreviations used in the algorithm for constructing the testing tree.

VS : visited-state pool
LL : last-level in the tree
NL : next-level in the tree

Figure 3.7: Abbreviations used in Testing Tree Construction Algorithm

The following is the algorithm for constructing the testing tree.

Step 1: Let $VS = \emptyset$.

Step 2: Create a node labeled with init-state. Let LL = init-state.

Step 3: Let $NL = \emptyset$

Step 4: For each node $n \in LL$, if n not $\in VS$, do step 4.1.

Step 4.1: For each input i, do the following

Step 4.1.1: Attach an edge to node n, label it i.

Step 4.1.2: Attach a node to edge i, label it ns which is the next-state of n when input i is applied.

Step 4.1.3: Add ns to NL.

Step 4.1.4: Add n to VS.

Step 4.2: Else go to next step.

Step 5: Let $LL = NL$

Step 6: If all the states are in VS, then stop; else go to step 4.

The idea of this algorithm is to build a tree starting at the initial state and attach nodes to it according to the state diagram. A node is terminated when it has appeared in the higher level of the tree. It is easy to see that there are a total of n (which is the number of states) internal nodes in the tree.

After constructing the tree, set P can be obtained by taking all the partial paths in the tree. It is also easy to see that there is a total of $m \times n$ paths in the tree.

The next step is to construct W, the characterisation set. A characterisation set is a set of input sequences which can identify the states of the machine by the reactions of the machine to the input sequences. They are also referred to as identifying sequences [Koh78]. In other words, the characterisation set consists of input sequences that can distinguish between the behaviours of every pair of states in a minimal automata. [Cho78]

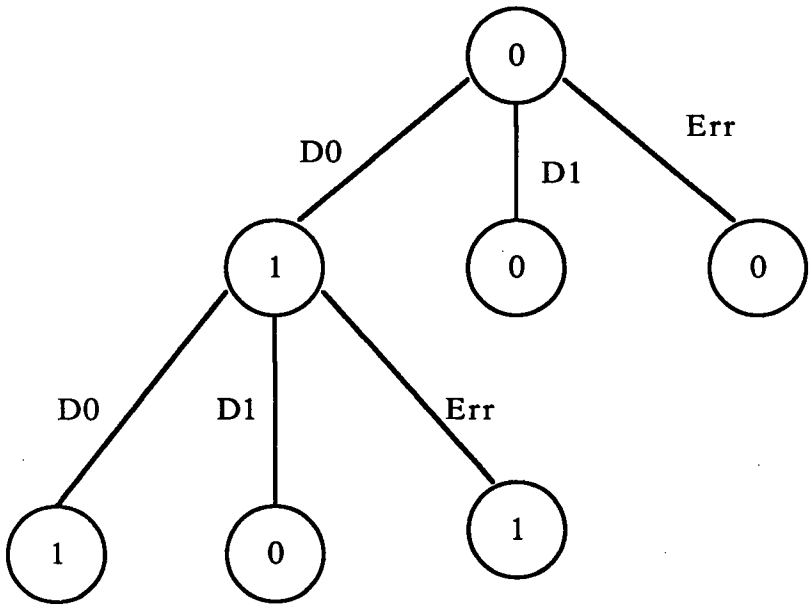
After both P and W are constructed, test suites are obtained through the equation:

$$Z = W \cup P \cdot W \cup P^2 \cdot W \cup ... \cup P^{m-n} \cdot W \tag{3.2}$$

In our case, the number of states in the implementation and the number of states in the specification are the same, i.e. $m = n$ in the above formula. So the formula for generating Z is reduced to:

$$Z = W \cup P \cdot W \tag{3.3}$$

3.3.2 Examples



a)Testing Tree of the alternating bit ptotocol

{D0}

D0 * {D0}

D1 * {D0}

Err * {D0}

D0 * D0 * {D0}

D0 * D1 * {D0}

D0 * Err * {Do}

b) Test sequences of the alternating bit protocol using PW-method.

Figure 3.8: Alternating Bit Protocol Testing Tree and Test Sequences

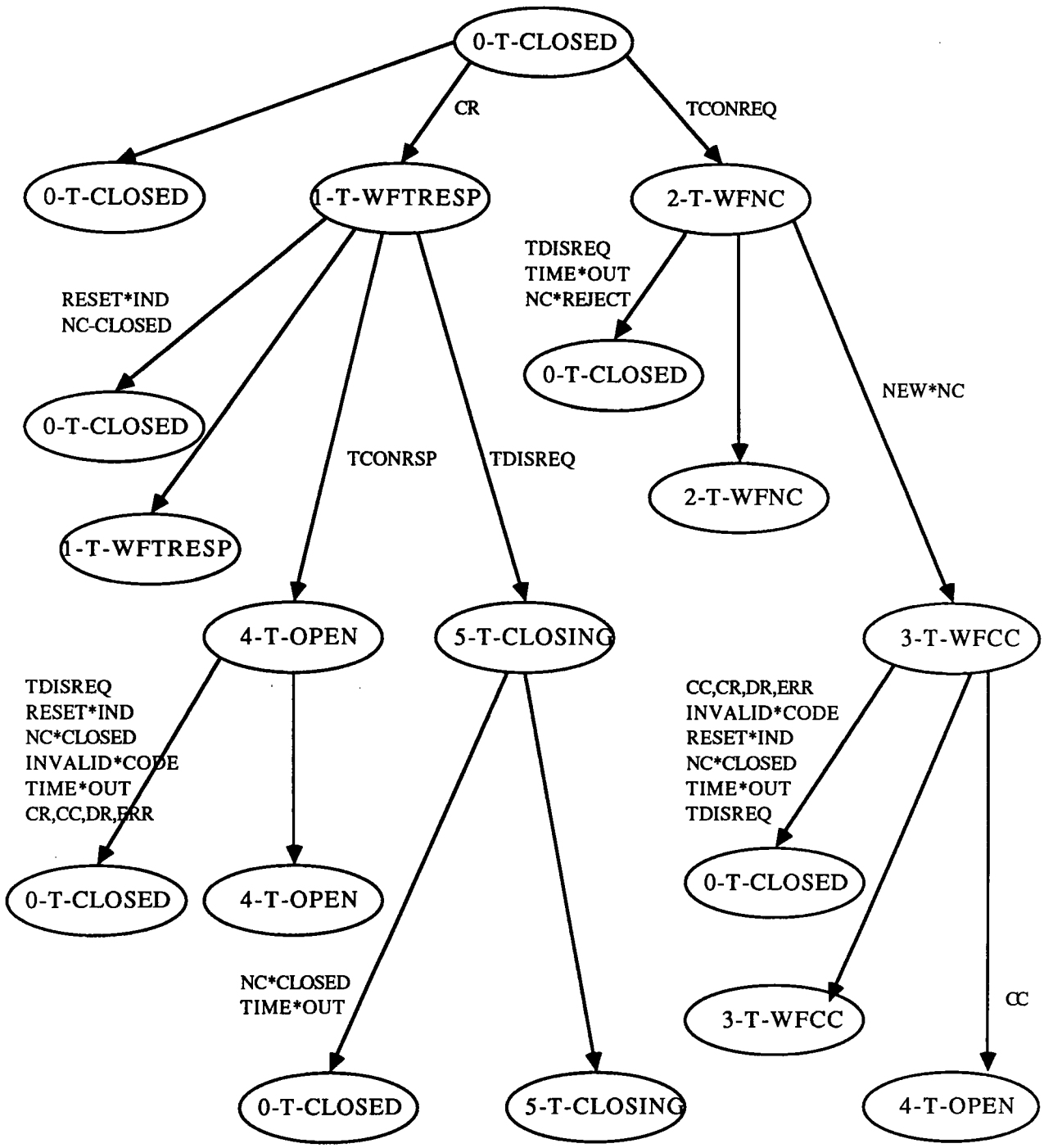


Figure 3.9: Testing Tree for Class 0 Transport Protocol

The testing tree generated for the alternating bit receiver's protocol is given in Figure 3.8. It is clear that the characterisation set for this protocol is either D0 or D1. In Figure 3.8, test sequences with characterization set D0 are also listed.

The application of the test tree algorithm on the ISO class 0 transport protocol has also been completed. The testing tree is shown in Figure 3.9. Self-loops are shown with arcs without labels, which contain all the inputs that do not appear in other arcs emerging from that particular state. For example, the arc goes from state 0-T-CLOSED to 0-T-CLOSED contains all the inputs except TCONREQ and CR*CODE. That arc is an abbreviation of 15 arcs since they all go to the same state. Set P of the testing tree of class 0 transport protocol is listed as follows:

TCONREQ
CR*CODE

TCONRSP
TDATREQ
TDISREQ
CC*CODE
DT*CODE
DR*CODE
ERR*CODE
INVALID*CODE
RESET*IND

NC*CLOSED
TIME*OUT
NC*REJECT
NEW*NC
EVALUATE*FLOW
TEVALFLOW

CR*CODE, RESET*IND
CR*CODE, NC*CLOSED
CR*CODE, TCONRSP
CR*CODE, TDISREQ

CR*CODE, TCONREQ
CR*CODE, TDATREQ
CR*CODE, CR*CODE
CR*CODE, CC*CODE
CR*CODE, DT*CODE
CR*CODE, DR*CODE
CR*CODE, ERR*CODE
CR*CODE, INVALID*CODE
CR*CODE, TIME*OUT
CR*CODE, NC*REJECT
CR*CODE, NEW*NC
CR*CODE, EVALUATE*FLOW
CR*CODE, TEVALFLOW

TCONREQ, TDISREQ
TCONREQ, TIME*OUT
TCONREQ, NC*REJECT
TCONREQ, NEW*NC

TCONREQ, TCONREQ
TCONREQ, TCONRSP
TCONREQ, TDATREQ
TCONREQ, CR*CODE
TCONREQ, CC*CODE
TCONREQ, DT*CODE
TCONREQ, DR*CODE
TCONREQ, ERR*CODE
TCONREQ, INVALID*CODE
TCONREQ, RESET*IND
TCONREQ, NC*CLOSED
TCONREQ, EVALUATE*FLOW
TCONREQ, TEVALFLOW

CR*CODE, TCONRSP, TDISREQ
CR*CODE, TCONRSP, CR*CODE
CR*CODE, TCONRSP, CC*CODE
CR*CODE, TCONRSP, DR*CODE
CR*CODE, TCONRSP, ERR*CODE
CR*CODE, TCONRSP, INVALID*CODE
CR*CODE, TCONRSP, RESET*IND
CR*CODE, TCONRSP, NC*CLOSED
CR*CODE, TCONRSP, TIME*OUT

CR*CODE, TCONRSP, NC*REJECT
CR*CODE, TCONRSP, NEW*NC
CR*CODE, TCONRSP, EVALUATE*FLOW
CR*CODE, TCONRSP, TEVALFLOW
CR*CODE, TCONRSP, TCONRSP

CR*CODE, TDISREQ, NC*CLOSED
CR*CODE, TDISREQ, TIME*OUT
CR*CODE, TDISREQ, TCONREQ
CR*CODE, TDISREQ, TCONRSP
CR*CODE, TDISREQ, TDATREQ
CR*CODE, TDISREQ, TDISREQ
CR*CODE, TDISREQ, CR*CODE
CR*CODE, TDISREQ, CC*CODE
CR*CODE, TDISREQ, DT*CODE
CR*CODE, TDISREQ, DR*CODE
CR*CODE, TDISREQ, ERR*CODE
CR*CODE, TDISREQ, INVALID*CODE
CR*CODE, TDISREQ, RESET*IND
CR*CODE, TDISREQ, NC*REJECT
CR*CODE, TDISREQ, NEW*NC
CR*CODE, TDISREQ, EVALUATE*FLOW
CR*CODE, TDISREQ, TEVALFLOW

TCONREQ, NEW*NC, TDISREQ
TCONREQ, NEW*NC, CR*CODE
TCONREQ, NEW*NC, CC*CODE
TCONREQ, NEW*NC, DT*CODE
TCONREQ, NEW*NC, DR*CODE
TCONREQ, NEW*NC, ERR*CODE
TCONREQ, NEW*NC, INVALID*CODE
TCONREQ, NEW*NC, RESET*IND
TCONREQ, NEW*NC, NC*CLOSED
TCONREQ, NEW*NC, TIME*OUT
TCONREQ, NEW*NC, TCONREQ

TCONREQ, NEW*NC, TCONRSP
TCONREQ, NEW*NC, TDATREQ
TCONREQ, NEW*NC, NC*REJECT
TCONREQ, NEW*NC, NEW*NC
TCONREQ, NEW*NC, EVALUATE*FLOW

TCONREQ, NEW*NC, TEVALFLOW

One characterization set for TRC0 can be $\{\text{TCONREQ}, \text{TCONRSP}, \text{TDATREQ}, \text{NEW} * \text{NC}, \text{CR} * \text{CODE}\}$, since according to the output of this set of inputs, we can identify all the states. However, the set $\{\text{CR} * \text{CODE}, \text{NEW} * \text{NC}\}$ is also a characterisation set. It is the shortest one for this protocol. Therefore, it is chosen as the characterisation set for the thesis.

To get the set Z, use equation 3.3 to concatenate each path of the above listed paths with the characterization set. As we mentioned in the last section, there are a total of $m \times n$ paths in P, so there are $1 + m \times n$ test sequences in Z. On average, the length of a path is $n \div 2$. Therefore the average number of test events is $n \div 2 + m \times n^2 \div 2$. Compared with the Transition Tour technique, the PW-method has a higher complexity.

In the case of ISO class 0 transport protocol, there are a total of 103 test sequences and the average length of each path is 3. The total length of the test events is 309 in the average case.

3.4 The Checking Sequence

The main ideas of the checking sequence approach to fault detection experiments are to identify all the states and to check all the transitions. The checking sequence consists of three concatenated parts [Gon70],

1. The initial sequence

   This sequence brings the machine to a particular state called the starting state. Usually this state is a stable state from where the rest of the checking sequence can be applied. In the example of the class 0 transport protocol, the starting state is 0-T-CLOSED.

2. The $\alpha - Sequence$

   This sequence recognizes all the states before applying DS (the distinguishing sequence) and all the states after applying DS.

3. The $\beta - Sequence$

   This sequence checks all the transitions in the state tables.

Since we always start at state 0-T-CLOSED of our communication protocol, the initial sequence can be omitted. In the following sections, we first introduce some definitions used and then give detailed descriptions of the $\alpha - sequence$ and $\beta - sequence$, and algorithms

to generate these sequences. The alternating bit protocol and ISO class 0 transport protocol will be discussed as examples, and the test sequences generated by using the checking sequence technique are presented.

3.4.1 Definitions and Notation

The following definitions and notation are used in the description of the checking sequence generation algorithm. This section may be skipped initially and used as a reference when reading the algorithm.

In the following discussion, the notation is the same as introduced in Section 3.1.1 except X is used here as the input set and $x_i's$ are used as individual inputs. Besides the assumptions described in Section 3.1.3, the finite state machine is assumed to possess distinguishing sequences (DS) [Gon70].

Distinguishing Sequence(DS)

A distinguishing sequence (DS) is an input sequence which can identify the state in which the DS is applied by the output of the machine [Koh78,Gil62,Sal69] . This is the same as the characterisation set defined in section 3.3.1.

X-diagram

An X-diagram of a finite-state machine is a diagram in which the states are the same states as the original finite state machine, and the arc (transition) from state S_i to state S_j, (S_i, S_j), indicates that state S_j is the next-state after applying input sequence X to state S_i.

Source

A source for an input sequence X is a state which is not the next-state of any state under input X. In other words, it is a state which will not be reached by applying X to the machine. In terms of the X-diagram, a source of an X-diagram is a state which does not have any incoming arcs in the X-diagram.

T-sequence

A T-sequence from state S_i to state S_j is the shortest transition (input) sequence which brings the machine from state S_i to state S_j.

Recognized

A state S_i is said to be recognized if from the knowledge of the output, S_i can be identified with a given state.

Note that S_i can be the previous state or the next-state in the application of an input sequence X.

α – Diagram

An α – *diagram* of a finite state machine is an X-diagram in which the input sequence X is the shortest DS of the machine.

Note there are only n arcs in an α – *diagram* since there are a total of n states in the finite state machine.

β – Diagram

A β – *diagram* of a finite state machine is an X-diagram in which the input sequence X is the concatenation of each input $x_i(i = 1, 2, ...m)$ and the shortest DS of the machine.

Note there are a total of $m \times n$ arcs in the β – *diagram* since there are m inputs and n states in the finite state machine.

Modified β – *diagram*

A modified β – *diagram* is a β – *diagram* without the arcs which have been recognized in the α – *sequence*. That means if input x_i has been applied to state S_j in the α – *sequence*, then the arc, labeled with $x_i\dot{D}S$, coming out of state S_i will not appear in the modified β – *diagram*.

Isthmus

A isthmus is an arc such that deletion of it divides the graph into two components.

3.4.2 Major Steps

There are several major steps in the procedure of deriving the checking sequence of a finite state machine. They are listed below:

1. Find the shortest DS and name it as X_d.
2. Find T-sequences.
3. Construct $\alpha - diagram$.
4. Identify sources if any.
5. Derive $\alpha - sequence$ using the algorithm presented in section 3.4.3.
6. Construct $\beta - diagram$.
7. Construct the modified $\beta - diagram$ (optional).
8. Derive $\beta - sequence$ using the algorithm presented in section 3.4.4.

3.4.3 The $\alpha - Sequence$

As we mentioned before that $\alpha - sequence$ recognizes all the states before applying DS (the distinguishing sequence) and all the states after applying DS, so $\alpha - sequence$ should be such that X_dX_d is applied to each state at least once. This is because every state can be

recognized as the result state of applying X_d and it also can be recognized as the start state of applying X_d to it. Therefore X_dX_d should be applied to each state at least once.

The following is the algorithm to derive an $\alpha - sequence$.

Step 1: Choose the initial state as the present state.

Step 2: Apply X_d.

Step 3: Let S_n be the state to which the $\alpha - diagram$ goes after the application of X_d to the present state.

Step 3.1: If S_n is a state not yet recognized, let S_n be the present state and go to step 2.

Step 3.2: Otherwise go to next step.

Step 4: Apply X_d. (Apply X_d to a already recognized state.)

Step 5: Check weather there is a source not yet recognized,

Step 5.1; If yes, go to step 6.

Step 5.2: Otherwise go to step 7.

Step 6: Apply a T-sequence from the present state to one of the sources not yet recognized, then let that source be the present state and go to step 2.

Step 7: Check whether there is any state not yet recognized,

Step 7.1: If yes, apply a T-sequence from the present state to the state not yet recognized, then let that state be the present state and go to step 2.

Step 7.2: Otherwise Terminate.

3.4.4 The $\beta - Sequence$

To derive a $\beta - Sequence$, the first thing done is to construct the $\beta - diagram$. Each node S_i in the $\beta - diagram$ is labelled with the value of (# of outgoing arcs - # of incoming arcs) and call it d_i. Divide all the nodes into three categories according to their d_i values.

1. $E = \{S_i \subset S | d_i = 0\}$
2. $G = \{S_i \subset S | d_i > 0\}$
3. $L = \{S_i \subset S | d_i < 0\}$

Notice that E contains all the states which have an equal number of outgoing arcs and incoming arcs. G consists of states which have more outgoing arcs than incoming arcs, while L has all the states which have fewer outgoing arcs than incoming arcs.

The purpose of dividing the nodes of a $\beta - diagram$ in this fashion is to find those states which can be jumped to, and so produce a minimal

β – *sequence*. The states which can be jumped to are contained in set G.

The following is the algorithm to produce a minimal β – *sequence*.

Step 1: Choose the initial state as the present state.

Step 2: Check if an arc which is not an isthmus can be found from the present state.

Step 2.1: If yes, follow it and then erase it. Update the present state and go to Step 3.

Step 2.2: Otherwise, find a T-sequence which goes to a state in G. Let that state be the present state and go to Step 3.

Step 3: If all the arcs are erased, stop. Otherwise, go to step 2.

3.4.5 Examples

In Figure 3.10, the α – *diagram*, the β – *diagram*, the α – *sequence*, the β – *sequence*, and the checking sequence of the alternating bit protocol are listed.

We have applied the checking sequence technique to the class 0 transport protocol as well. In Figure 3.11, the α – *diagram* of the class 0 transport protocol is presented. It is clear that there are only six

arcs in this diagram since there are a total of six states in this finite state machine. Using the algorithm presented on page 42, the shortest $\alpha - sequence$ is as follows,

State:	0,	1,	1,	$\rightarrow$ 4,	0,
Input:	X_d,	X_d,	TCONRSP,	X_d,	TCONREQ

State:	$\rightarrow$ 2,	3,	0,	$\rightarrow$ 5,	5
Input:	X_d,	X_d,	CR*CODE, TDISREQ,	X_d	

The symbol $\rightarrow$ represents a T-sequence and its content is listed in the 'Input' line.

Since X_d is $\{CR * CODE, NEW * NC\}$, the total length of this $\alpha - sequence$ is 16.

Figure 3.12 is the corresponding $\beta - diagram$. There are a total of 102 arcs in the diagram. For simplicity's sake, not all the inputs are marked in the diagram. Arcs without labels represent the transitions caused by the rest of the input set plus DS.

Figure 3.13 gives a summary of the $\beta - diagram$. From this information, we can see that

$$\begin{aligned} \text{E} &= \{\emptyset\} \\ \text{G} &= \{2,3,4,5\} \\ \text{L} &= \{0,1\} \end{aligned}$$

The following is the $\beta - sequence$ for ISO class 0 transport protocol,

TCONREQ+DS,
TCONRSP+DS,
DT*CODE+DS,
RESET*IND+DS,
TEVALFLOW+DS,
CR*CODE+DS,
ERR*CODE+DS,
TIME*OUT+DS,
TEVALFLOW+DS,
TCONREQ+DS,
TDISREQ+DS,
TCONRSP,
NC*REJECT+DS,
DR*CODE+DS,
TCONRSP,
TIME*OUT+DS,
RESET*IND+DS,
RESET*IND,
TCONREQ,
TCONREQ+DS,
TCONRSP+DS,
TDATREQ+DS,
CR*CODE+DS,
CC*CODE+DS,
DT*CODE+DS,
DR*CODE+DS,
ERR*CODE+DS,
TCONREQ,
RESET*IND,
NC*CLOSED+DS,
CC*CODE,
EVAL*FLOW + DS,
TCONREQ,
RESET*IND,
CR*CODE+DS,
CC*CODE,
NEW*NC,
TCONREQ,
RESET*IND,
TIME*OUT+DS,
TDATREQ+DS,
NEW*NC,
TCONREQ,
RESET*IND,
TEVALFLOW+DS,
NEW*NC+DS,
NEW*NC,
TCONREQ,

CC*CODE+DS,
TDATREQ+DS,
DR*CODE+DS,
NC*REJECT+DS,
NC*CLOSED+DS,
CC*CODE+DS,
INVALID*CODE+DS,
NC*REJECT+DS,
TCONRSP+DS,
TDATREQ+DS,
TCONRSP,
DT*CODE+DS,
DT*CODE+DS,
TCONRSP,
TEVALFLOW+DS,
RESET*IND,
RESET*IND,
TCONREQ,
NC*REJECT+DS,
DR*CODE+DS,
ERR*CODE+DS,
TDISREQ+DS,
RESET*IND+DS,
NC*CLOSED+DS,
TIME*OUT+DS,
INVALID*CODE+DS,
CC*CODE,
INVALID*CODE+DS,
TCONREQ,
RESET*IND,
INVALID*CODE+DS,
CC*CODE,
TEVALFLOW+DS,
TCONREQ,
RESET*IND,
DR*CODE+DS,
CC*CODE,
NEW*NC,
TCONREQ,
RESET*IND,
NC*REJECT+DS,
CR*CODE+DS,
NEW*NC,
TCONREQ,
RESET*IND,
CR*CODE,
EVAL*FLOW+DS,
NEW*NC,

TDISREQ+DS,
CR*CODE+DS,
ERR*CODE+DS,
NEW*NC+DS,
TCONREQ+DS,
DT*CODE+DS,
RESET*IND+DS,
NEW*NC+DS,
TCONRSP+DS,
TCONRSP,
TDATREQ+DS,
CC*CODE+DS,
TCONRSP,
EVAL*FLOW + DS,
INVALID*CODE+DS,
TCONREQ,
TCONREQ,
TIME*OUT+DS,
RESET*IND,
RESET*IND,
RESET*IND,
RESET*IND,
RESET*IND,
RESET*IND,
RESET*IND,
RESET*IND,
TDISREQ+DS,
CC*CODE,
RESET*IND+DS,
TCONREQ,
RESET*IND,
TIME*OUT+DS,
CC*CODE,
NEW*NC,
TCONREQ,
RESET*IND,
ERR*CODE+DS,
TCONREQ+DS,
NEW*NC,
TCONREQ,
RESET*IND,
NEW*NC+DS,
DR*CODE+DS,
NEW*NC,
TCONREQ,
RESET*IND,
CR*CODE,
TEVALFLOW+DS

TCONREQ+DS,
CC*CODE+DS,
INVALID*CODE+DS,
EVAL*FLOW + DS,
TDATREQ+DS,
DR*CODE+DS ,
NC*CLOSED+DS,
EVAL*FLOW+DS,
TCONRSP,
TCONRSP+DS,
CR*CODE+DS,
TCONRSP,
NEW*NC+DS,
ERR*CODE+DS,
TDISREQ,
NEW*NC+DS,
TDISREQ+DS,
RESET*IND,
TCONREQ,
TCONREQ,
TCONREQ,
TCONREQ,
TCONREQ,
TCONREQ,
TCONREQ,
TCONREQ,
RESET*IND,
RESET*IND+DS,
CC*CODE,
NC*CLOSED+DS,
TCONREQ,
RESET*IND,
CC*CODE+DS,
CC*CODE+DS,
NEW*NC,
TCONREQ,
RESET*IND,
NC*CLOSED+DS,
TCONRSP+DS,
NEW*NC,
TCONREQ,
RESET*IND,
EVAL*FLOW+DS,
NC*REJECT+DS,
NEW*NC,
TCONREQ,
RESET*IND,

The length of the $\beta - sequence$ is 395. By adding this to the $\alpha - sequence$ and a $T - sequence$ which brings the machine to initial state after applying $\alpha - sequence$, the total length of the checking sequence is 412.

3.4.6 Analysis

In this section a number of proofs and some analyses are provided, verifying the correctness of the checking sequence algorithm introduced above.

It is easy to show that the length of an $\alpha - sequence$, $l(\alpha)$, satisfies the following equation:

$$l(\alpha) \leq 2nL + (n-1)^2 \tag{3.4}$$

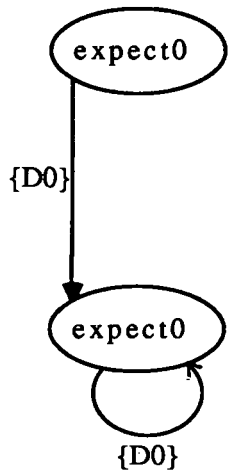
and the length of a $\beta - sequence$, $l(\beta)$, satisfies the following equation:

$$l(\beta) \leq q(n-1)^2 + qn(L+1), \tag{3.5}$$

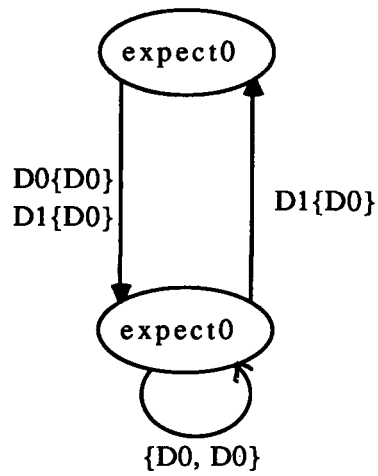
where L is the length of DS, q is the number of inputs and n is the number of states. So the total length of a checking sequence, $l(\alpha+\beta)$, is as follows,

$$l(\alpha+\beta) \leq (1+q)(n-1)^2 + 2nL + qn(L+1). \tag{3.6}$$

This result is shorter than both [Hen84,Kim66]. In our testers, $L = 2$, $q = 17$, and $n = 6$. The total length of the checking sequence is thus at most 780.



a) a-diagram



b) b-diagram

{D0, D0}

c) a-sequence

{D0, D0, D1, D0, D1, D0, D1, D0, D0, D0}

d) b-sequence

{D0, D0, D1, D0, D0, D1, D0, D1, D0,D0, D0}

d) checking sequence

Figure 3.10: The Checking Sequence of the Alternating Bit Protocol

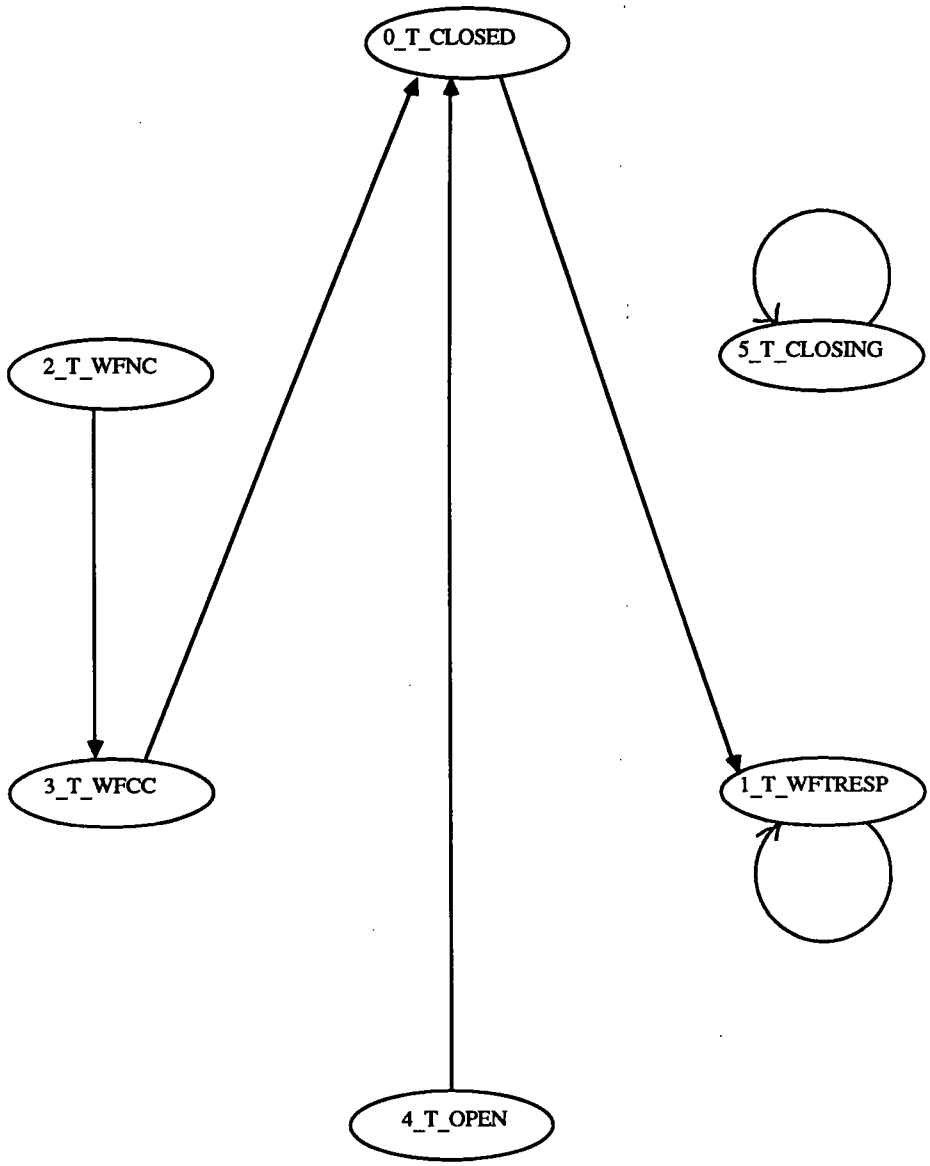


Figure 3.11: The α – *diagram* of the ISO Class 0 Transport Protocol

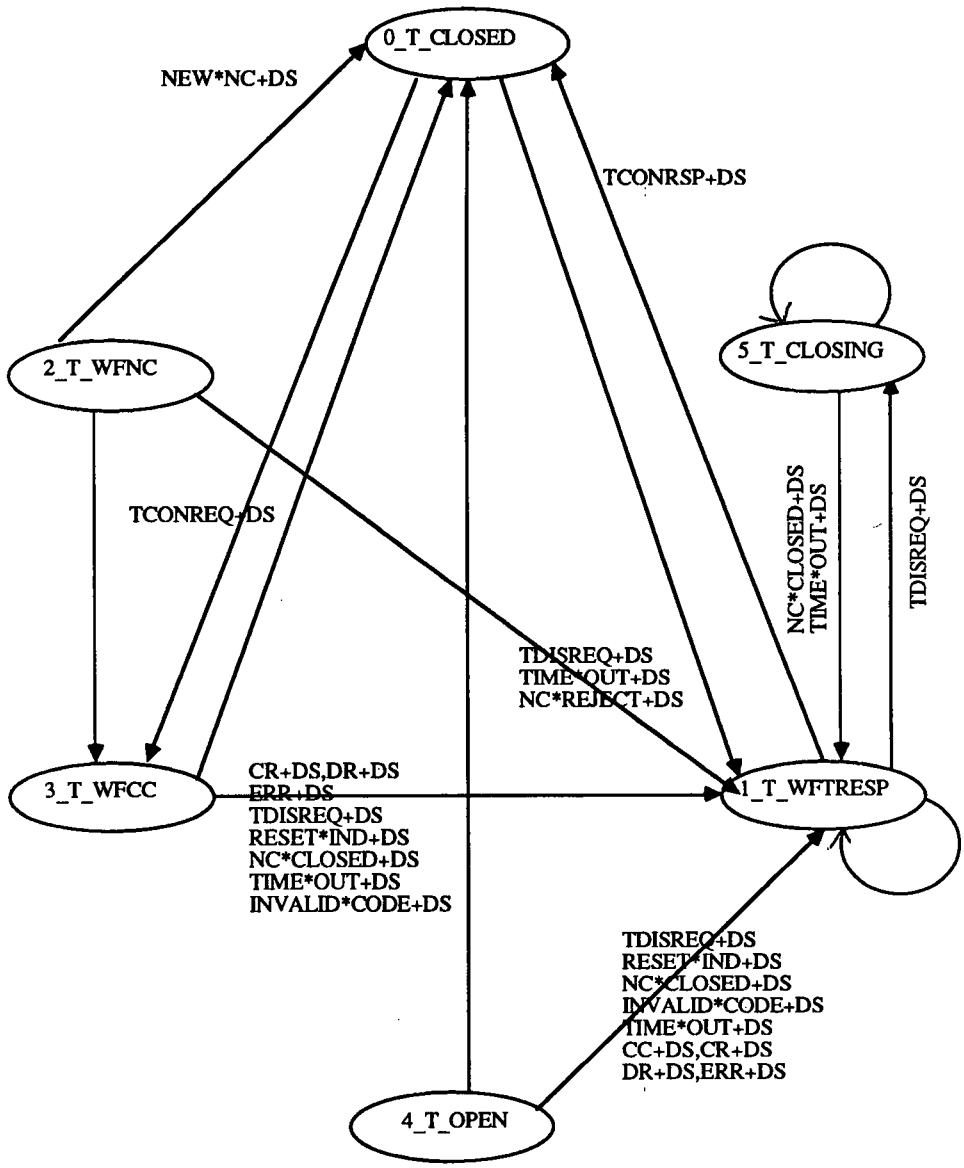


Figure 3.12: The $\beta - diagram$ of the ISO Class 0 Transport Protocol

state	# of incoming arcs	# of outgoing arcs	di
0-T-CLOSED	19	17	-2
1-T-WFTRESP	53	17	-36
2-T-WFNC	0	17	17
3-T-WFCC	14	17	3
4-T-OPEN	0	17	17
5-T-CLOSING	16	17	1
Total	102	102	0

Figure 3.13: Information of the $\beta - diagram$ of the TRC0

Chapter 4

Conformance Testing of a TPC0 Implementation

4.1 Test Method

A local tester of the ISO class 0 transport protocol has been developed on an MPT, a portable tester. This is achieved by an upper tester residing on top of the implementation under test (IUT) and a lower tester residing at the bottom of the IUT. The control and observation points are at the service boundaries above and below the IUT. The test events are specified in terms of the transport ASPs above the IUT, and the network ASPs and the transport PDUs below the IUT.

In figure 4.1, we give the structure of our testers. Although this is a local test, the upper tester and the lower tester reside on different CPUs. On CPU1, the upper tester sits on top of the IUT and the com-

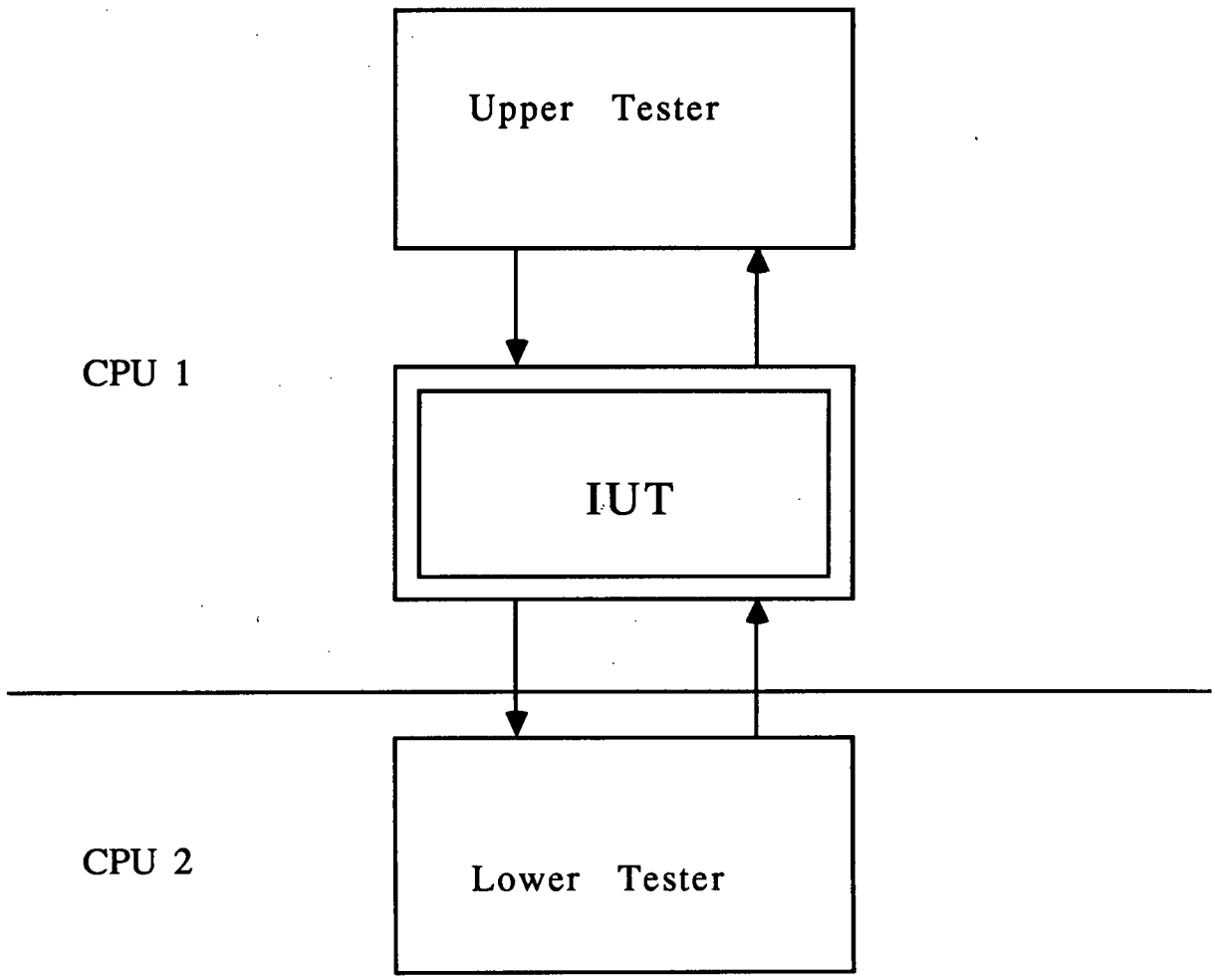


Figure 4.1: Test Method Used for Transport Layer Class 0

munication is through the service access points (SAPs). On CPU2, however, the lower tester communicates with the IUT through inter-cpu communications. The coordination between the upper tester and the lower tester is embedded in the test suites. For each test suite, there is one sub-test-sequence for the upper tester and another sub-test-sequence for the lower tester. This is because the test suite consists of events which can be obtained by the IUT from both the upper boundary and the lower boundary of the IUT. In other words, a test suite consists of the events of the transport ASPs, the network ASPs, and the transport PDUs.

4.2 Test Sequence Generation

4.2.1 Test Sequence Generation Techniques

In Chapter 3, we have discussed the test sequence generation techniques. We have modified three algorithms and have applied the modified algorithms to the ISO class 0 transport protocol. The abstract test suites generated from these techniques are listed on Page 26, Page 33, and Page 47.

We have developed executable test suites for the IUT according to the abstract test suites. There is a one-to-one relationship between the

abstract test suites and the executable test suites. The executable test suites are considerably more complicated than their abstract equivalents. This is caused by adding parameters for the service primitives or PDU fields for transport data. For each executable test suite, there are a number of different versions for different parameter variations. They follow the guidelines listed in the next section.

4.2.2 Test Sequence Divisions

We have divided the test sequence into the following groups according to their functions:

1. Normal Operations

   These test sequences are further divided into the following categories according to the protocol phases:

   (a) Connection Establishment

   (b) Data Transfer

   (c) Connection Closure

2. Abnormal Operations

   These test sequences are further divided into the following subcategories:

   (a) Data transfer before connection establishment

(b) Data transfer after connection closure

(c) Connection Closure before Connection Establishment

4.2.3 Parameter Variations

We have applied parameter variations on the parameters of the transport services primitives and the fields of transport protocol data units. We have divided them into the following categories according to their characteristics [Dat86],

A field may:

1. always be present or only be present sometimes
2. have fixed length or variable length
3. be historical dependent or historical independent
4. have a permanent value or a fixed value only for one connection
5. have a default value or have no default value

For fields which have ranges we applied the two extreme values, middle values, and illegal values. For fields which can only have a single value, we used the legal value and the illegal values. For fields which have one limit, for example "x op value", where op is a relational operator and x is a variable, we apply values according to the relational operator as follows:

1. If op is >, then we use
   (a) $x > value$ (correct operation)
   (b) $x = value$ (error report)
   (c) $x < value$ (error report)
2. If op is <, then we use
   (a) $x < value$ (correct operation)
   (b) $x = value$ (error report)
   (c) $x > value$ (error report)
3. If op is =, then we use
   (a) $x = value$ (correct operation)
   (b) $x > value$ (error report)
   (c) $x < value$ (error report)
4. If op is >=, then we use
   (a) $x > value$ (correct operation)
   (b) $x = value$ (correct operation)
   (c) $x < value$ (error report)
5. If op is <>, then we use
   (a) $x > value$ (correct operation)
   (b) $x < value$ (correct operation)

(c) $x = value$ (error report)

6. If op is <=, then we use

   (a) $x < value$ (correct operation)

   (b) $x = value$ (correct operation)

   (c) $x > value$ (error report)

4.3 The Upper Tester

The major tasks of our upper tester are the following:

1. sending the transport ASPs defined in the ISO standards
2. receiving the transport ASPs defined in the ISO standards
3. handling all the events specified in the test suites for the upper tester
4. automatic execution of the upper tester test suites
5. providing information on what it has been observed at the transport SAPs
6. making conclusions based on its observations

Here, the ASPs defined in the ISO standards include:

TCONREQ	TCONIND	TCONRSP	TCONCON
TDATREQ	TDATIND	TEXDATREQ [1]	TEXDATIND [2]
TDISREQ	TDISIND		

Sending the transport ASPs defined in the ISO standards is achieved by procedure calls, while receiving the ASPs is done by the upper tester checking some pre-arranged memory locations to get the information. Control messages, such as TCONREQ, TCONIND, TDISREQ, TDISIND etc, take higher priorities than data messages.

If the IUT passes a test sequence, specific conclusions of that particular test sequence will be given. The conclusions include a statement of the purposes of that particular test sequence and a confirmation of the successful test results of that test sequence. If the IUT fails a test before or after all the test events in the test sequence are exhausted, a diagnosis on the reasons of failure will be given.

Every interaction between the upper tester and the IUT is printed on the screen for the convenience of trace and observation.

The upper tester will automatically execute the test sequence selected for a particular test purpose until the test sequence is completed or the test sequence is interrupted by an abnormal behaviour of the IUT.

[1] This is not implemented in the IUT.
[2] This is not implemented in the IUT.

The control structure of our upper tester is as follows. The upper tester first takes an executable test event from an executable test sequence which is the transformation of an executable test sequence and the values of the parameters. Then the upper tester sends the test events to the IUT through a procedure call which is one of the service primitives defined in the ISO standards. After this, the control is passed to the IUT. The IUT operates accordingly and sends some information to the lower tester if necessary. Eventually, the IUT sends back the reaction to the upper tester. Once the upper tester receives this reaction from the IUT through the session layer receive services, it checks the reaction with the expected reaction stored in the upper tester. If the received event is the same as the expected events, testing will continue. Otherwise the execution is stopped and a diagnostic error message is printed.

4.4 The Lower Tester

The lower tester simulates the network service access points and 'feeds' the IUT with test events specifically designed for the lower tester. It also makes conclusions on what it has observed. The major tasks carried on by the lower tester are very similar to the upper tester's. The upper tester and the lower tester reside on different

CPUs, so they can both take instructions from human operators. Both of them can be the initiator or the responder of a connection request, data request, or a disconnection request. Therefore they both can be active or passive in a test event. The major differences between the tasks of the upper tester and the tasks of the lower tester are that for the the upper tester, ASPs are of the transport layer, whereas for the lower tester, the ASPs are defined for the network layer.

The following are the major tasks of the lower tester:

1. sending the network ASPs defined in the ISO standards
2. receiving the network ASPs defined in the ISO standards
3. sending the transport PDUs defined in the ISO standards
4. receiving the transport PDUs defined in the ISO standards
5. handling all the events specified in the test suites for the lower tester
6. automatic execution of the lower tester test suites
7. providing information on what it has observed
8. making conclusions based on its observations

4.5 The Coordination Between The Upper Tester and The Lower Tester

The coordination between the upper tester and the lower tester is embedded in the test suites. For each abstract test sequence, there is one executable sub-test-sequence for the upper tester and another executable sub-test-sequence for the lower tester. This is because the abstract test sequence consists of events which can be obtained by the IUT from both its upper boundary and its lower boundary. In other words, an abstract test sequence consists of the events of the transport ASPs, the network ASPs, and the transport PDUs. They will be sent to the IUT through either the upper boundary or the lower boundary of the IUT. Assigning the test events to the upper tester or the lower tester requires careful planning.

Synchronization between the two testers is crucial for correctly handling the observations and the responses. After the upper tester (or the lower tester) sends its reaction to the IUT, the IUT processes the information and sends it to the lower tester (or the upper tester). Then the control is passed back to the upper tester (or the lower tester) since the upper tester and the lower tester reside on different CPUs. The upper tester (or the lower tester) cannot continue

executing another test event in the test sequence since it has not received the response from the lower tester (or the upper tester) yet. The upper tester (or the lower tester) should wait for the IUT and the lower tester (or the upper tester) to complete their responses and then continue the execution of another test event in the test sequence.

4.6 The IUT Used

The implementation under test used in this thesis is implemented by Jean-Marc Serre. It is written in Forth and designated for a portable tester MPT produced by Idacom Electronics.

Chapter 5

Test Results

Although during the IUT development, the developer had tested the implementation intensively, due to the limitation of the test method used, some mistakes were not detected at that stage. They are detected when we apply our testers to the implementation. In the following section, we will present the test results and how the mistakes are discovered and some conclusions.

5.1 Problems Detected in the IUT

As mentioned in the previous chapters, an upper tester and a lower tester have been developed for the testing of the IUT. Both testers have the abilities of sending, observering, and receiving information and PDUs from the IUT. The following problems are discovered in the IUT during our local test.

1. We have used the lower tester to send the transport PDUs to the IUT. We also initiated the connection from the peer entity. This simulates the situation where the remote entity issues a connection request. In the development process of the IUT, this is very difficult to do, and therefore it is normally ignored. After we have applied the local test on the IUT, the lower tester can send a transport PDU to the IUT. In this case, we have discovered that the IUT could not handle this situation very well. As a consequence, the IUT could not interpret the transport calling address correctly. After consulting the source code, we found that the routine which decodes the calling address had made a slip. This mistake was not discovered during the development stage since the lower layer (network layer) used for development was not fully tested. That is, the situation in which the peer entity initiates a connection request could not be tested without a lower tester. Another reason for this mistake is that it is not appropriate to simulate the peer entity by the local entity. This is because if the local entity and the remote entity reside on the same machine, they share the same memory locations. They use same global variables and same system services, and therefore a lot of unexpected errors could occur.

2. As we mentioned in Chapter 4, we have used the guidelines presented in last chapter for parameter variations. We have discovered that the IUT cannot handle illegal addresses in service primitives TCONREQ and TCONRSP properly. If an address with a smaller number of delimiters appears in the parameter field, the system will crash. The system goes into an infinite loop when parsing the address. After consulting the source code, we found out that this was caused by the string parsing package which could not handle an abnormal address representation.

3. In the test of timing and timer variations of the IUT, we have tested different time periods. The IUT times out correctly. However, after a time out, the network requests a disconnect. This only closes the entity which initiates the connection. If the entity is the passive entity in a connection, a called entity, it is left open after a network disconnect request. The state of the requested transport connection is T-WFRESP, rather than T-CLOSED. The consequences of this mistake are that some transport connections cannot be re-used and eventually all the transport connections will be occupied by nonexistent 'users'.

4. During the tests of inopportune behaviour, we sent data after a connection was closed, the data was sent without any complaints.

However, if the data is sent before a connection is established, the IUT will reject the data transfer request. This indicates that the IUT does not handle the connection closure properly. It left the connection partly open although the state of the connection had been changed to 0-T-CLOSED.

5.2 Remarks

Some crucial problems of the IUT are detected by using an upper tester and a lower tester even after the implementor had tested the implementation intensively using other techniques. In general, we conclude that the method of local test is very effective, since it simulates the real environment of an IUT. The complete coverage of the test suites also provides sound confidence in the behaviour and capabilities of an IUT. Untested lower layers cannot be used in the development and testing stages. They may introduce mistakes and confuse the judgement on the implementation under test.

Chapter 6

Conclusions

6.1 Summary

This thesis has described the problem of conformance testing of communication protocol implementations. Local testing on class 0 transport protocol of the ISO standard has been conducted through an upper tester and a lower tester. Test sequence generation techniques have been developed and applied to the ISO class 0 transport protocol.

As one of the major problems, the selection of test scenarios, techniques for complete coverage in terms of state transitions and output functions without an exhaustive approach, have been researched and applied to the IUT. The PW-method of [Cho78], checking sequence method of [Gon70] and [Koh78], and TT-method of [NT81] are mod-

ified, developed, and part of the test suites are implemented for the ISO class 0 transport protocol.

The method used in this thesis is a local model which shows the effectiveness of this testing method for the development of communication protocol implementations. Some mistakes in the IUT are detected using an upper tester and a lower tester. These mistakes would not be detected without the two testers since without the lower tester, the interaction between the transport layer and the network layer cannot be simulated and therefore errors related to this matter in the implementation under test will not be uncovered. In general, we conclude that the method of local testing is very effective, especially when reliable lower layers are not available.

The transformation of an abstract test suite into an executable test suite is an implementation matter which requires a large amount of effort, especially when the coordination between the upper tester and the lower tester is not direct.

6.2 Future Work

The area of conformance testing of communication protocol implementations is still a most active one, both from a theoretical and

from an implementation standpoint.

Within the area of the theory of conformance testing, the following present interesting and open problems,

- At present, the variable variations applied in conformance testing of communication protocol implementations are at the stage of software testing, such as using extreme values, mid values, and illegal values. The complexity of the data flow graphs of [Sar85] is quite high. Further research on variation of variable context is urgently needed.
- The communication between the upper tester and the lower tester is also an active research issue. How this communication can be achieved for the case, in which the upper tester and the lower tester reside on different systems (coordinated teting method) and the lower layers are not reliable, is not thoroughly resolved.

Within the area of implementation, work in the following areas is needed,

- To transform an abstract test suite into an executable test suite is a complex procedure since it is not a one-to-one transformation between two forms. It involves service access points, and the

distribution of responsibilities between the upper tester and the lower tester.

- Communications between the upper tester and the lower tester can be very complex. It is also an implementation issue. The coordination between the two testers has to be established on top of reliable lower layers. When reliable lower layers are not available, this coordination has to be achieved by other means.
- The interface between the lower layer and the IUT, and the interface between the upper layer and the IUT are also important implementation matters. They are application and machine dependent. Automation of such task would be worthwhile.
- The techniques of test sequence generation are well defined. Therefore, software packages which generate test suites from a protocol specification in some format, such as Estelle and Lotos would be very useful. Expert systems for parameter variation and test suite development are also needed.

Bibliography

[Ans82] J. P. Ansart. A protocol independent system for testing protocol implementation. In C. Sunshine, editor, *Protocol Specification, Testing and Verification*, pages 523–528, Elsevier Science Publishers B.V. (North-Holland), 1982.

[BCMS83] G.V. Bochmann, E. Cerny, M. Maksu, and B. Sarikaya. Testing transport protocol implementations. In *Canadian Information Processing Society*, 1983.

[BEG86] H. J. Burkhardt, H. Eckert, and A. Giessler. Testing of protocol implementations - A systematic approach to derivation to test sequence. In *Protocol Specification, Testing and Verification*, pages 461–481, Elsevier Science Publishers B.V. (North-Holland), 1986.

[Bri86] Lonc Brigitte. Genepix : A portable version under unix of the OSI protocol tester GENEPI. In *Protocol Specification, Testing and Verification*, pages 507–519, Elsevier

Science Publishers B.V. (North-Holland), 1986.

[Cho78] Tsun S. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[CS85] R. Castanet and R. Sijelmassi. *Local Tests for Distributed Testing.* Technical Report, University of Bordeaux, 1985.

[Dat86] Rajendra R. Datar. Test sequence generation for network protocol. 1986.

[DB86] Rachida Dssouli and Gregor V. Bochmann. Conformance testing with multiple observers. In *Protocol Specification, Testing, and Verification*, 1986.

[EBM*83] E.Cerny, G.V. Bochmann, M. Maksud, A. Leveille, J. M. Serre, and B. Sarikaya. *Experiments in Testing Communication Protocol Implementations.* Technical Report 492, University of Montreal, 1983.

[Gil62] Arthur Gill. *Introduction to the Theory of Finite-State Machine.* McGraw-Hill Book Company, Inc, 1 edition, 1962.

[Gon70] Guney Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computers*, 551–558,

June 1970.

[Hen84] F. C. Hennie. Fault detecting experiments for sequencial circuits. In *Proc. 5th Ann. Symp on Switching Theory and Logical Design*, pages 95–110, November 1984.

[ISO86] *OSI Conformance Testing Methodology and Framework.* 1986.

[Kim66] C. R. Kime. *A Failure Detection Method for Sequential Circuits.* Tech. Rept. 66-13, Dept. of Elec. Engi., University of Lowa, January 1966.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Company, 2 edition, 1978.

[LJ83] R.J. Linn and J.S.Nightingale. Some experience with testing tools for OSI protocol implementations. In *Protocol Specification, Testing, and Verification*, 1983.

[LW83] R.J. Linn and W.H.McCoy. Producing tests for implementations of OSI protocols. In *Protocol Specification, Testing, and Verification*, 1983.

[NT81] Sachio Naito and Masahiro Tsunoyama. Fault detection for sequential machines by transition tours. *IEEE*, 238–243, 1981.

[PFM83] S. Palazzo, P. Fogliata, and G. L. Moli. A layer-independent architecture for a testing system of protocol implementations. In C. S. West, editor, *Protocol Specification, Testing and Verification*, pages 393–406, Elsevier Science Publishers B.V. (North-Holland), 1983.

[Raf85] O. Rafiq. Tools and methodology for testing OSI protocol entities. In *The International Symposium on Fault Tolerant Computing, IEEE*, 1985.

[Ray82] D. Rayner. A system for testing protocol implementations. In C. Sunshine, editor, *Protocol Specification, Testing and Verification*, pages 539–554, Elsevier Science Publishers B.V. (North-Holland), 1982.

[Ray86] D. Rayner. Towards standardized OSI conformance tests. In M. Daiz, editor, *Protocol Specification, Testing and Verification*, pages 441–460, Elsevier Science Publishers B.V. (North-Holland), 1986.

[RCR86] O. Rafig, C.Chraibi, and R.Castanet. *Experimental Testing of Transport Protocol.* Technical Report, University de Bordeaux, 1986.

[Sal69] Arto Salomaa. *Theory of Automata.* Pergamon Press, 1

edition, 1969.

[Sar85] B. Sarikaya. *Test Design for Computer Network Protocols.* PhD thesis, University of Montreal, 1985.

[SB82] B. Sarikaya and G. V. Bochmann. Some experience with test sequence for protocols. In C. Sunshine, editor, *Protocol Specification, Testing and Verification*, pages 555–567, Elsevier Science Publishers B.V. (North-Holland), 1982.

[Sta85] William Stallings. *Data and Computer Communications.* Macmillan Publishing Company, Collier Macmillan Publishers, 1985.

[Tan81] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc, 1981.

[UP83] Hasan Ural and Robert L. Probert. User-guided test sequence generation. In *Protocol Specification, Testing, and Verification*, 1983.

[Vuo83] Son T. Vuong. *Valira - A Tool for Protocol Validation Via Reachability Analysis.* 1983.

[ZR86] H. X. Zeng and D. Rayner. The impact of the ferry concept on protocol testing. In *Protocol Specification, Testing and Verification*, pages 519–531, Elsevier Science Publish-

ers B.V. (North-Holland), 1986.