CHAPTER III

CONFORMANCE TESTING METHODOLOGIES

1. INTRODUCTION

Currently, the main issue in protocol conformance testing that requires further research is combining the efforts in three major fields: formal description techniques (FDTs), formal test generation techniques and the conformance testing standards. These fields address the same problems in the area of communication protocols, namely assuring that products are implemented correctly and they will interoperate and deliver specified services to the users. Unfortunately, in the reported results, there are gaps to be filled, and inconsistencies to be resolved in some cases. This is primarily due to the trade-off between the ease of implementation and effectiveness in testing. It is well established that defining a complex protocol as a collection of relatively small communicating modules is preferable. However, if not designed carefully, formal specifications written as a set of communicating modules (or processes) may result in implementations with unpredictable external behavior; in this case, unpredictable behavior may cause insufficiencies in service delivery and make thorough testing almost impossible. In this chapter, we make an attempt to identify some of the problems regarding a formidable task: assimilation of various results in the fields of test generation techniques, formal description techniques and conformance testing standards.

In Section 2, we discuss the techniques for generating conformance tests in detail; the techniques that are reported in the literature can be classified into four major groups. The first technique is called the *transition tour method* as outlined by Sarikaya and Bochmann in *Some Experience with Test Sequence Generation for Protocols*. An input sequence (called a *transition tour*) is applied to an implementation to check whether the state transitions are implemented correctly. In *X.25 Conformance Testing – A Tutorial*, Sherif et al. applied the transition tour method to the X.25 Data-Link and Network Layer protocols. In *Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931*, Uyar and Dahbura introduce an optimization technique to minimize the length of the tour based on the graph theoretic concept called the *Chinese postman problem*. The second technique is called the *distinguishing sequences method* where the state of an implementation is identified by applying a set inputs and analyzing the outputs. Hennie introduced this method in his paper titled *Fault Detecting Experiments for Sequential Circuits*. Gönenç brings an algorithmic approach to the distinguishing sequences method in *A Method for the Design of Fault Detection Experiments*. In *Using Checking Sequences for OSI Session Layer Conformance Testing*, Hengeveld and Kroon report a case study that applies this method to generate test sequences for Session Layer protocol. For protocols that do not have distinguishing sequences, the *characterizing sequences method* can be used to identify the current state of an implementation. This method is discussed in Chow's paper titled *Testing Software Design Modeled by Finite-State Machines*. Fujiwara et al. propose a technique in *Test Selection on Finite State Machines* to shorten the final test sequence obtained by this method. The last method, called the *unique input/output sequences*, was developed by Sabnani and Dahbura is presented in *A Protocol Test Generation Procedure*. Compared to the distinguishing and characterizing sequences methods the unique input/output sequences method requires significantly fewer restrictions on protocol specifications than most techniques. Aho et al. apply the optimization technique introduced by Uyar and Dahbura into the unique input/output sequences method and present an algorithm to minimize the length of the test sequence in *An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours*. This technique uses a more general form of the Chinese postman problem, called the *rural Chinese postman problem*. Miller and Paul present an enhancement for the unique input/output sequences method in *Generating Minimal Test Length Test Sequences for Conformance Testing of Communication Protocols*. In the last paper of this section, titled *Fault*

Coverage of Protocol Test Methods, Sidhu and Leung present a study to assess the relative fault coverage of the above four test generation techniques.

The techniques described in Section 2 are primarily applicable to protocols that are specified as finite state machines (FSMs) which were defined in Chapter I. These techniques can be readily used to generate tests for the *control flow portion* of a protocol (e.g., establishing, maintaining and clearing connections). In Section 3, we briefly discuss testing the *data portion* of a protocol (e.g., once a connection is established, testing to assure that the information conveyed among entities over the connection and interactions between the input parameter values, context variables and output parameters are satisfactory). The analysis of testing the data portion of a protocol is not as precise as the control flow portion – partly because the results reported in the literature are applicable to special cases of extended FSM (EFSM) and FDT specifications (as opposed to general solutions), and partly because the topic is currently an open research problem. Synchronization issues discussed in Section 7 of Chapter II are not revisited here. Note that the same concerns regarding the synchronization of various entities in a test system remain valid for test generation aspects of the conformance testing problem.

In Section 3, we discuss applicability of the FSM-based models to the specifications that are defined by using EFSMs and FDTs. Algorithmic test generation methods can be used for the specifications based on EFSMs or FDTs provided that the specifications satisfy certain conditions such as avoiding spontaneous state transitions and transient states, and making the parameters and internal variables controllable. In general, this subject is an open research problem and test generation techniques are not necessarily readily utilizable at this point. However, we believe that the techniques that are discussed in Section 2 present a good starting point for further research. We discuss the advancements in test generation for FDT-based specifications which are described in *TESDL: Experience with Generating Test Cases from DSL Specifications* by Brömstrup and Hogrefe for SDL, in *LOTOS Specifications, Their Implementations and Their Tests* by Brinksma et al. for LOTOS, and in *A Test Design Methodology for Protocol Testing* by Sarikaya et al. for Estelle. In *Derivation of Test Cases for LAP-B from a LOTOS Specification*, Gueraichi and Logrippo give a case study for applying the unique input/output sequences technique to a certain class of specifications written in LOTOS. As a complementary study to those reported in this chapter, we also include the test generation techniques based on software engineering which are described by Ural in *A Test Derivation Method For Protocol Conformance Testing*.

In Section 4, we discuss the relationship between the principles defined by the conformance testing standard [1], which were briefly presented in Section 2 of Chapter II, and conformance test generation techniques: test purposes, abstract test cases and test methods are among the concepts that are evaluated in terms of conformance test generation techniques.

Section 5 summarizes the chapter, and Section 6 discusses some of the open research problems and gives references for further reading.

2. Algorithmic Procedures for Conformance Testing

In this section, we discuss the methodologies reported in the literature which bring algorithmic solutions to the conformance testing problem. In Section 2.1, we define conformance testing from the viewpoint of algorithmic procedures by introducing two important concepts, called *controllability* and *observability*. Four major techniques, called *transition tours method, distinguishing sequences method, characterizing sequences method* and *unique input/output sequences method* are discussed in Sections 2.2, 2.3, 2.4 and 2.5, respectively. In Section 2.6, we present a study addressing the fault coverage of these methods.

Note that the four techniques considered in this section assume that a directed graph representation of a protocol specification is strongly-connected. Recall from Section 4.1 in Chapter I that a directed graph is called strongly-connected if there is a path from any vertex to any other vertex. This property corresponds to protocol specifications (not implementations!) that do not have any deadlocks. We have to state the difference, however, the protocol specifications that consist of single or multiple communicating entities. For the latter case, the fact that all communicating entities have strongly-connected specifications does not guarantee that the global specification obtained by combining them is free of deadlocks; the communication among the entities may easily cause a deadlock such as waiting for

a message that will never arrive. In this case, a verification of the specification is needed. The scope of this book is limited to single entity (i.e., single finite state machine) specifications.

2.1. Conformance Testing Problem

The purpose of conformance testing is to check whether an implementation of a protocol behaves in accordance with its specification. Conformance testing is classified as a *black-box* approach if an external tester can only observe the outputs generated by the implementation upon receipt of inputs (as opposed to the tester having information about the internal design of an implementation). Since the black-box approach was introduced previously, we analyze this approach from a different point of view – conformance test generation.

Various techniques are available to make black-box testing more effective and efficient. They are classified into four major techniques: *transition tours*, *distinguishing sequences*, *characterizing sequences* and *unique input/output sequences*. Before we discuss each technique, let us define the purpose of testing a state transition of an FSM in more detail.

Recall that we denote a state transition from state s_i to s_j which is caused by an $input_k$ and generates $output_l$ by:

$$(s_i, s_j \; ; \; input_k/output_l)$$

Given an implementation of an FSM, referred to as an *implementation under test (IUT)*, there are three steps to test whether this state transition is realized correctly:

Step I: Bring the IUT into state s_i.

Step II: Apply the input called $input_k$ and observe that the IUT generates the output called $output_l$.

Step III: Verify that the final state of the IUT is s_j.

In the remainder of this chapter, we will refer to the above three steps as the *basic test procedure*. In general, the above steps of the basic test procedure are not trivial to realize for a given FSM implementation for two reasons: limitations on the controllability and the observability of the IUT. Throughout this chapter, we will use the same example FSM specification of PhoneDTE presented in Chapter I, which is replicated in Figure 3.1 for your ease of reference.

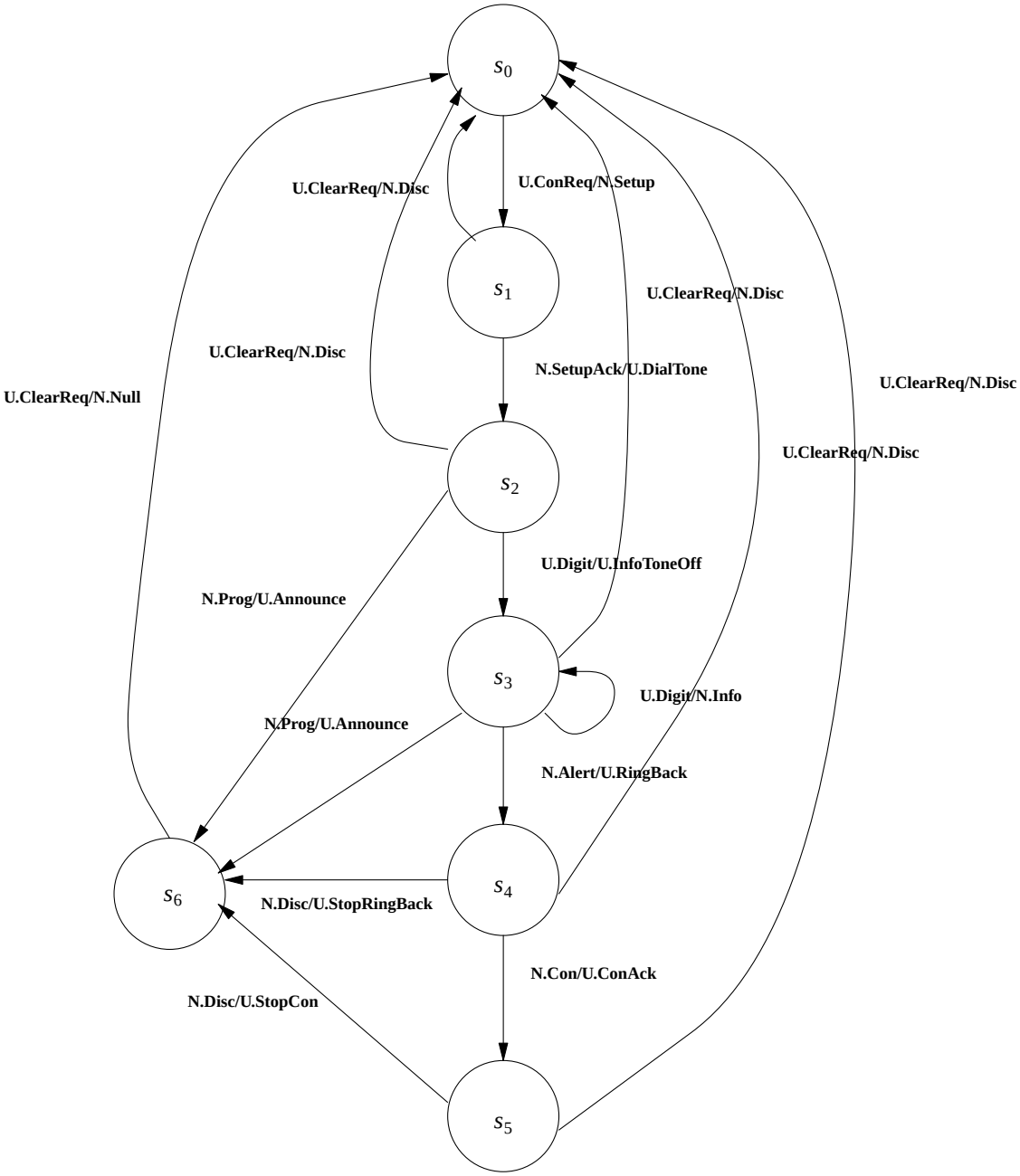

Figure 3.1. PhoneDTE specification as a directed graph.

Due to the limited *controllability* of an implementation, it is not possible to bring the implementation into state s_i in Step I of the above procedure without realizing several transitions. For example, for an implementation of the PhoneDTE specification, the tester has to apply three inputs to move the implementation from state s_0 to s_3, namely the inputs of *U.ConReq, N.SetupAck,* and *U.Digit*. The test sequence has to take advantage of the fact that once an implementation is brought into a particular state, as many aspects as possible should be tested without returning to the initial state. In complex real-life protocols, this task (i.e., to decide what to test next, once the implementation is in a given state) becomes very difficult. Unless an efficient solution is found to overcome this difficulty, the number of inputs required to test every specified edge of a real-life protocol may require an unacceptably large number of inputs, and consequently, the time required to realize such tests in a test laboratory may be prohibitively long.

Due to the limited *observability* of an implementation, in Step III of the basic test procedure, it is typically not possible to directly verify that the implementation is in state s_j. For example, in Figure 3.1, consider the state transitions from s_1 to s_2 to s_0 realized by sending to the implementation the inputs of *N.SetupAck* and *U.ClearReq* and observing the outputs of *U.DialTone* and *N.Disc*, respectively. There is no way of knowing that the implementation actually makes these state transitions by only observing the outputs. Suppose an implementation error caused the FSM to stay in state s_1 after *N.SetupAck* is applied (instead of moving to state s_2 for the above example); in this case, the implementation would have generated the same output *N.Disc* when *U.ClearReq* is applied (since both s_1 and s_2 behave the same way for the input of *U.ClearReq*). Because of this implementation error, a tester would have mistakenly concluded that s_1 to s_2 state transition is implemented correctly, although it has not. Therefore, the issue of verifying that a state transition to a final state actually occurred is critical for improving the effectiveness of a test in terms of the ability to identify implementation errors.

All conformance test generation techniques that are reported in the literature try to present solutions to the observability and the controllability issues in testing. In the remainder of this section, we discuss four major techniques briefly and describe how each one addresses (or fails to address) these two issues.

2.2. Transition Tour Method

The transition tour method is the most straight-forward approach for conformance test generation. The state transitions defined in a protocol specification are exercised at least once by applying an input sequence to an implementation, starting from the initial state of the FSM. Such an input sequence is called a *transition tour* of the FSM. Recall from Section 4.1 of Chapter I that a tour in a directed graph is defined as a sequence of consecutive edges that starts and ends at the same vertex. Traditionally, the vertex corresponding to the initial state of an FSM is considered as the starting and ending vertex of a transition tour. In a transition tour, traversing an edge corresponds to sending the input defined for that edge to an IUT and observing the output generated by the IUT. Therefore, a sequence of edges in a transition tour is interpreted as exercising the inputs and outputs of an IUT in various states.

The transition tour method was first suggested by Naito et al. [2] for FSM-based representations of fully-specified, strongly-connected sequential circuits by a heuristic algorithm. The transition tour approach is first applied to the area of protocol conformance testing by Sarikaya and Bochmann in *Some Experience with Test Sequence Generation for Protocols*. The authors introduce the method as "the simplest approach" compared to other methods and apply the technique to Transport Layer protocol. The transition tour method, which is applicable to partially-specified protocols (as described by Sarikaya and Bochmann), has the major disadvantage that Step III of the basic test procedure defined in Section 2.1 is not part of the technique. In other words, the state verification step is omitted, severely limiting the fault detection capability of the technique.

In *X.25 Conformance Testing – A Tutorial*, Sherif et al. present one of the early applications of the transition tour method to X.25 Data-Link and Network Layer protocols. The specifications of these protocols are represented in a state transition table form, which are derived from the English text given in the standard. Although the new state verification step is not addressed, this paper was among the first to point out that conformance tests can be structured as valid, invalid and inopportune tests, thereby provided support from industry to the development of the conformance testing standards.

Uyar and Dahbura applied the transition tour approach to a certain class of protocols that have a special feature, called the *"status feature."* In a protocol specification with the status feature, there is a special input message defined in every protocol state, called *status_inquiry*; applying *status_inquiry* to an implementation generates an output that reports the current state of the implementation and the implementation stays in the same state. Formally stated, a protocol specification with the status feature has a state transition defined for every state s_i such that

$$(s_i, s_i \ ; \ status\_inquiry \ / \ report\_s_i \ )$$

Note that the names and formats of actual input and output messages may be different for a particular protocol. In their paper titled *Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931*, the authors incorporate Step III of the basic test procedure into the transition tour method by applying a *status_inquiry* input after traversing each new edge, therefore, verifying the new state of the implementation. In this case, the basic test procedure defined in Section 2.1 to test a state transition specified as (s_i, s_j ; $input_k$ / $output_l$) becomes:

Step I: Bring the implementation into state s_i.

Step II: Apply the input called $input_k$ and observe that the implementation generates the output called $output_l$.

Step III: Apply *status_inquiry* to the implementation and observe that the output is $report\_s_j$.

Uyar and Dahbura also showed that the length of the transition tour can be minimized by using the graph theoretic concept called the *Chinese postman problem* [3], where a postman is required to deliver the mail to every street of a town in such a way that the postman walks through each street a minimum number of times (the streets and the intersections of streets correspond to edges and vertices of a graph, respectively). For directed graphs, the Chinese postman problem is defined as finding a minimum-cost tour of the graph such that every edge of the graph is traversed at least once. Each edge of the graph is associated with an integer *cost* value representing the time and difficulty to realize the input and output operations of that edge. For example, an edge that represents an expiry of a long timer will have a higher cost value than an edge requiring a simple input and output. If all the edges of a graph have the same cost value, the minimum-cost tour that covers every edge corresponds to a minimum-length tour. The authors give an efficient algorithm for generating a minimum-cost transition tour of a protocol specification and apply the technique to ISDN Network Layer protocol, called Q.931 (see the paper for details).

Using this technique, a minimum-cost transition tour for PhoneDTE specification is given in Figure 3.2. We assumed that all edges defined for PhoneDTE has the same cost value, therefore, the tour is of minimum-length. Note that in the tour of Figure 3.2 Step III of the basic test procedure (i.e., verification of new state) for each state transition test is omitted.

TEST SEQUENCE TABLE

STEP	CURRENT STATE	NEXT STATE	MSG TO IUT	MSG FROM IUT
1<--	s_0	s_1	**U.ConReq**	**N.Setup**
2<--	s_1	s_2	**N.SetupAck**	**U.DialTone**
3<--	s_2	s_6	**N.Prog**	**U.Announce**
4<--	s_6	s_0	**U.ClearReq**	**N.Null**
5	s_0	s_1	U.ConReq	N.Setup
6	s_1	s_2	N.SetupAck	U.DialTone
7<--	s_2	s_3	**U.Digit**	**U.InfoToneOff**
8<--	s_3	s_3	**U.Digit**	**N.Info**
9<--	s_3	s_4	**N.Alert**	**U.RingBack**
10<--	s_4	s_5	**N.Con**	**U.ConAck**
11<--	s_5	s_6	**N.Disc**	**U.StopCon**

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
12	s_6	s_0	U.ClearReq	N.Null
13	s_0	s_1	U.ConReq	N.Setup
14	s_1	s_2	N.SetupAck	U.DialTone
15	s_2	s_3	U.Digit	U.InfoToneOff
16<--	s_3	s_6	**N.Prog**	**U.Announce**
17	s_6	s_0	U.ClearReq	N.Null
18	s_0	s_1	U.ConReq	N.Setup
19	s_1	s_2	N.SetupAck	U.DialTone
20	s_2	s_3	U.Digit	U.InfoToneOff
21	s_3	s_4	N.Alert	U.RingBack
22<--	s_4	s_6	**N.Disc**	**U.StopRingBack**
23	s_6	s_0	U.ClearReq	N.Null
24	s_0	s_1	U.ConReq	N.Setup
25	s_1	s_2	N.SetupAck	U.DialTone
26	s_2	s_3	U.Digit	U.InfoToneOff
27	s_3	s_4	N.Alert	U.RingBack
28	s_4	s_5	N.Con	U.ConAck
29<--	s_5	s_0	**U.ClearReq**	**N.Disc**
30	s_0	s_1	U.ConReq	N.Setup
31	s_1	s_2	N.SetupAck	U.DialTone
32	s_2	s_3	U.Digit	U.InfoToneOff
33	s_3	s_4	N.Alert	U.RingBack
34<--	s_4	s_0	**U.ClearReq**	**N.Disc**
35	s_0	s_1	U.ConReq	N.Setup
36	s_1	s_2	N.SetupAck	U.DialTone
37	s_2	s_3	U.Digit	U.InfoToneOff
38<--	s_3	s_0	**U.ClearReq**	**N.Disc**
39	s_0	s_1	U.ConReq	N.Setup
40<--	s_1	s_0	**U.ClearReq**	**N.Disc**

Figure 3.2. A minimum-length transition tour for PhoneDTE specification given in Figure 3.1. (New state verification for each state transition test is omitted.)

In Figure 3.2, the columns called *MSG TO IUT* and *MSG FROM IUT* represent the input message sent to the IUT and the expected output message generated by the IUT, respectively. The current and expected next state of the IUT are shown in the columns labeled as *CURRENT STATE* and *NEXT STATE*, respectively. During conformance testing, the inputs are applied to an IUT in the order denoted by the column called *STEP*. If the response of an IUT is not what is expected at any step of the tour, an error is detected in the implementation. In Figure 3.2, a step marked by an arrow "<--" indicates that a state transition denoted as (s_i, s_j ; $input_k$ / $output_l$) is being tested at that step. For example, in Step 1, the state transition defined as (s_0, s_1 ; $U.ConReq$ / $N.Setup$) is being tested. The steps that are not marked represent the intermediate steps to bring the IUT into a state where a state transition is to be tested. For example, Steps 12 through 15 are used to bring the IUT from state s_6 to state s_3 where *N.Prog/U.Announce* is to be tested.

Now, suppose the specification of Figure 3.1 has the status feature defined as $(s_i, s_i\ ;\ N.Status/N.Report\_s_i\ )$ for each state s_i. For example, state s_0 of Figure 3.1 would include a permissible input called $N.Status$ which would generate an output called $N.Report\_s_0$ and the FSM would stay in state s_0. In this case, every new edge to be tested can include a new state verification by using the status feature. Note that the status feature messages also need to be tested. A minimum-length test sequence is given in Figure 3.3. (We assumed that every state transition of PhoneDTE has the same cost value.)

TEST SEQUENCE TABLE

STEP	CURRENT STATE	NEXT STATE	MSG TO IUT	MSG FROM IUT
1<--	s_0	s_0	**N.Status**	**N.Report_s_0**
2	s_0	s_0	**N.Status**	**N.Report_s_0**
3<--	s_0	s_1	**U.ConReq**	**N.Setup**
4	s_1	s_1	**N.Status**	**N.Report_s_1**
5<--	s_1	s_1	**N.Status**	**N.Report_s_1**
6	s_1	s_1	**N.Status**	**N.Report_s_1**
7<--	s_1	s_2	**N.SetupAck**	**U.DialTone**
8	s_2	s_2	**N.Status**	**N.Report_s_2**
9<--	s_2	s_2	**N.Status**	**N.Report_s_2**
10	s_2	s_2	**N.Status**	**N.Report_s_2**
11<--	s_2	s_3	**U.Digit**	**U.InfoToneOff**
12	s_3	s_3	**N.Status**	**N.Report_s_3**
13<--	s_3	s_3	**N.Status**	**N.Report_s_3**
14	s_3	s_3	**N.Status**	**N.Report_s_3**
15<--	s_3	s_3	**U.Digit**	**N.Info**
16	s_3	s_3	**N.Status**	**N.Report_s_3**
17<--	s_3	s_4	**N.Alert**	**U.RingBack**
18	s_4	s_4	**N.Status**	**N.Report_s_4**
19<--	s_4	s_4	**N.Status**	**N.Report_s_4**
20	s_4	s_4	**N.Status**	**N.Report_s_4**
21<--	s_4	s_5	**N.Con**	**U.ConAck**
22	s_5	s_5	**N.Status**	**N.Report_s_5**
23<--	s_5	s_5	**N.Status**	**N.Report_s_5**
24	s_5	s_5	**N.Status**	**N.Report_s_5**
25<--	s_5	s_6	**N.Disc**	**U.StopCon**
26	s_6	s_6	**N.Status**	**N.Report_s_6**
27<--	s_6	s_6	**N.Status**	**N.Report_s_6**
28	s_6	s_6	**N.Status**	**N.Report_s_6**
29	s_6	s_0	U.ClearReq	N.Null
30	s_0	s_1	U.ConReq	N.Setup
31	s_1	s_2	N.SetupAck	U.DialTone
32<--	s_2	s_6	**N.Prog**	**U.Announce**
33	s_6	s_6	**N.Status**	**N.Report_s_6**
34	s_6	s_0	U.ClearReq	N.Null
35	s_0	s_1	U.ConReq	N.Setup
36	s_1	s_2	N.SetupAck	U.DialTone
37	s_2	s_3	U.Digit	U.InfoToneOff

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
38<--	s_3	s_6	**N.Prog**	**U.Announce**
39	s_6	s_6	**N.Status**	**N.Report_**s_6
40	s_6	s_0	U.ClearReq	N.Null
41	s_0	s_1	U.ConReq	N.Setup
42	s_1	s_2	N.SetupAck	U.DialTone
43	s_2	s_3	U.Digit	U.InfoToneOff
44	s_3	s_4	N.Alert	U.RingBack
45<--	s_4	s_6	**N.Disc**	**U.StopRingBack**
46	s_6	s_6	**N.Status**	**N.Report_**s_6
47<--	s_6	s_0	**U.ClearReq**	**N.Null**
48	s_0	s_0	**N.Status**	**N.Report_**s_0
49	s_0	s_1	U.ConReq	N.Setup
50	s_1	s_2	N.SetupAck	U.DialTone
51	s_2	s_3	U.Digit	U.InfoToneOff
52	s_3	s_4	N.Alert	U.RingBack
53	s_4	s_5	N.Con	U.ConAck
54<--	s_5	s_0	**U.ClearReq**	**N.Disc**
55	s_0	s_0	**N.Status**	**N.Report_**s_0
56	s_0	s_1	U.ConReq	N.Setup
57	s_1	s_2	N.SetupAck	U.DialTone
58	s_2	s_3	U.Digit	U.InfoToneOff
59	s_3	s_4	N.Alert	U.RingBack
60<--	s_4	s_0	**U.ClearReq**	**N.Disc**
61	s_0	s_0	**N.Status**	**N.Report_**s_0
62	s_0	s_1	U.ConReq	N.Setup
63	s_1	s_2	N.SetupAck	U.DialTone
64	s_2	s_3	U.Digit	U.InfoToneOff
65<--	s_3	s_0	**U.ClearReq**	**N.Disc**
66	s_0	s_0	**N.Status**	**N.Report_**s_0
67	s_0	s_1	U.ConReq	N.Setup
68	s_1	s_2	N.SetupAck	U.DialTone
69<--	s_2	s_0	**U.ClearReq**	**N.Disc**
70	s_0	s_0	**N.Status**	**N.Report_**s_0
71	s_0	s_1	U.ConReq	N.Setup
72<--	s_1	s_0	**U.ClearReq**	**N.Disc**
73	s_0	s_0	**N.Status**	**N.Report_**s_0

Figure 3.3. A minimum-length transition tour for PhoneDTE specification with the status feature.

(New state verification is included for each state transition test.)

The notation used in the test sequence of Figure 3.3 is similar to the one in Figure 3.2. The test sequence is applied to an IUT in the order given by the *STEP* column. A step marked by an arrow "<--" indicates that a state transition is being tested. However, each state transition test of Figure 3.3 consists of two steps: the step that is marked by an arrow, and the step following it for verification of the new state. For example, at Steps 17 and 18, the state transition from s_3 to s_4 defined as *N.Alert/U.RingBack* is being tested. At the first step an input is applied to the IUT and the output is observed (i.e., Step II of the basic test procedure given in Section 3.1) as shown at Step 17 where *N.Alert* is sent to the IUT and *U.RingBack* is expected. The second step is the verification of the new state of the IUT (i.e., Step III of the basic test procedure) as given in Step 18 where the expected state of an IUT is verified as s_4 by sending *N.Status* to the IUT and expecting *N.Report_*s_4 as response.

Note that the test sequence of Figure 3.3 also checks the correctness of the status feature implementation. For example, Steps 1 and 2 test the implementation of *N.Status/N.Report_*s_0 at state s_0; similarly, Steps 5 and 6 test *N.Status/N.Report_*s_1 at state s_1.

2.3. Distinguishing Sequences Method

A *distinguishing sequence* of an FSM is an input sequence which generates an output sequence that identifies the state of an implementation before the input sequence is applied. In other words, when a distinguishing sequence is applied to an FSM implementation that is originally in state s_i, the output sequence generated by the implementation is different for each state s_i.

Distinguishing sequences were originally developed by researchers for testing sequential digital circuits (see for example [4],[5]). The so-called *checking experiments* (i.e., a sequence of input signals) are applied to a digital circuit to obtain information about the operation of an implementation, such as the state that the implementation is in before or after the experiment is started, or its operation is error-free, etc.

In his 1964 paper, titled *Fault Detecting Experiments for Sequential Circuits*, Hennie defines the fundamental concepts in checking experiments including the definitions of the synchronizing, homing and distinguishing sequences.

A *synchronizing sequence* of an FSM is an input sequence that, when applied to an implementation of the FSM, leaves the implementation in a certain state, independent of the state that the implementation was before the synchronizing sequence is applied. A *homing sequence* of an FSM is an input sequence that determines the final state of an implementation by observing the output sequence generated. For example, the input called *U.ClearReq* is a synchronizing sequence for the PhoneDTE specification given in Figure 3.1. This input is also a homing sequence since the output (either *N.Disc* or *N.Null*) determines that the final state is s_0. Applying the input of $U.ClearReq$ moves an implementation into state s_0 regardless of the output generated by the implementation (which can be either $N.Disc$ or $N.Null$,

depending on the state of an implementation). Also, every distinguishing sequence is a homing sequence, however, the converse is not true. Typically, a distinguishing sequence of an FSM specification is much longer than the homing and the synchronizing sequence of the specification since it provides more information about the FSM implementation than the latter two. For more details about checking experiments, the reader is referred to the studies by Kohavi [4] and Bhattacharyya [5].

A distinguishing sequence can uniquely identify the state of an implementation, and, therefore, can be used to realize Step III of the basic test procedure. For each state transition defined as (s_i , s_j ; $input_k$ / $output_l$) the basic test procedure defined in Section 2.1 becomes:

Step I: Bring the implementation into state s_i. Using the terminology defined by Hennie and Kohavi [4], this step can be expanded as follows:

I.a: Apply the synchronizing sequence to bring the implementation into the initial state.

I.b: Apply the *transfer sequence* for s_i to bring the implementation from initial state into a s_i.

Step II: Apply the input called $input_k$ and observe that the implementation generates the output called $output_l$.

Step III: Apply the distinguishing sequence for s_j and verify that the new state after Step II was indeed s_j.

One of the earlier and frequently referenced studies on distinguishing sequences is reported by Gönenç in *A Method for the Design of Fault Detection Experiments*. Gönenç brings an algorithmic approach to apply the distinguishing sequences method to test sequential machine implementations. There are algorithms in the paper to generate so-called $\alpha-$ and $\beta-$ *sequences*. A test sequence which tests that the states are implemented correctly (regardless of the state transitions) is called an $\alpha-$ *sequence*. The test designer should know how many states are implemented in an FSM-based design (which may be equal to or larger than the number of states defined in the specification), and test the existence of each state by using the α-sequences. The state transitions of an implementation is tested by a $\beta-$ *sequence*. Note that, for black-box testing of communication protocols, α-sequences do not have a practical use. The information about the details of an implementation (i.e., the number of states that are implemented) is simply unavailable to the test designer. Therefore, a test designer can only assume that the number of states that an implementation has equal to the number of states that are defined in the FSM specification. As a consequence, use of the β-sequences is sufficient for the protocol conformance testing (or, more realistically speaking, *has to be* sufficient).

One of the rare applications of the distinguishing sequences method is reported in *Using Checking Sequences for OSI Session Layer Conformance Testing* by Hengeveld and Kroon, where they apply this method to the OSI Session Layer protocol. The specification is defined within the standard in the form

of a state transition table. However, there are some variables used in the state transition table whose values influence the next state and the output; therefore, the specification is an EFSM. The paper gives step-by-step description of the process of converting the specification from the EFSM to FSM form. An interesting application of test effectiveness techniques from hardware testing (such as "stuck-at" faults) are applied to the resulting test sequences.

Although the distinguishing sequences method can be a useful tool for realizing the state verification step of the basic test procedure, there are several drawbacks that may limit the applicability of this method to many real-life protocols. As mentioned above, the distinguishing sequences method is originally introduced for sequential digital circuits that are fully-specified. However, communication protocol specifications are typically partially-specified since it may not be physically possible to generate every input at every state. Hence, most real-life protocols do not posses a distinguishing sequence. Furthermore, the length of a distinguishing sequence becomes a severe limitation for running the tests in a test laboratory since the length is in the order of n^{n+1} for an FSM with n-states. We should note that a distinguishing sequence may exist even for specifications that are not fully-specified. It is the algorithms reported in the literature that require this constraint.

For illustration, let us try to apply the distinguishing sequences method to the specification PhoneDTE given in Figure 3.1. Assume that a telephone corresponding to the PhoneDTE specification is the implementation to be tested. One can see from the specification of PhoneDTE that it does not have a distinguishing sequence since there is no input which is permissible at *every state* of the specification. Also note that not every real-life protocol can be fully-specified. Suppose the input called *U.ConReq* corresponds to picking up the handset before dialing. It is clear that this input can only be generated in the protocol states where the telephone handset is hung up (only in s_0 in Figure 3.1), but not in other states.

However, for the sake of illustrating the concept of distinguishing sequences, let us assume that PhoneDTE is fully-specified, where every unspecified input in Figure 3.1 is replaced with an input that generates an *N.Null* output and the FSM stays in its current state. Figure 3.4 gives the state transition table for this fully-specified version of PhoneDTE specification.

inputs / states	U.Con Req	N.Setup Ack	U.Digit	N.Alert	N.Con	N.Disc	U.Clear Req	N.Prog
s_0	N.Setup (s_1)	N.Null (s_0)	N.Null (s_0)	N.Null (s_0)	N.Null (s_0)	N.Null (s_0)	N.Null (s_0)	N.Null (s_0)
s_1	N.Null (s_1)	U.DialTone (s_2)	N.Null (s_1)	N.Null (s_1)	N.Null (s_1)	N.Null (s_1)	N.Disc (s_0)	N.Null (s_1)
s_2	N.Null (s_2)	N.Null (s_2)	U.Info ToneOff (s_3)	N.Null (s_2)	N.Null (s_2)	N.Null (s_2)	N.Disc (s_0)	U.Announce (s_6)
s_3	N.Null (s_3)	N.Null (s_3)	U.Info (s_3)	U.RingBack (s_4)	N.Null (s_3)	N.Null (s_3)	N.Disc (s_0)	U.Announce (s_6)
s_4	N.Null (s_4)	N.Null (s_4)	N.Null (s_4)	N.Null (s_4)	U.ConAck (s_5)	U.Stop RingBack (s_6)	N.Disc (s_0)	N.Null (s_4)
s_5	N.Null (s_5)	N.Null (s_5)	N.Null (s_5)	N.Null (s_5)	N.Null (s_5)	U.StopCon (s_6)	N.Disc (s_0)	N.Null (s_5)
s_6	N.Null (s_6)	N.Null (s_6)	N.Null (s_6)	N.Null (s_6)	N.Null (s_6)	N.Null (s_6)	N.Null (s_0)	N.Null (s_6)

Figure 3.4. State transition table for the fully-specified version of PhoneDTE specification given in Figure 3.1.

In the directed graph representation of PhoneDTE, this specification change corresponds to adding a self-loop for each unspecified input at every state as follows:

$$(s_i, s_i; unspecified\_input_j \,/\, N.Null)$$

For example, in vertex s_3 of Figure 3.1, there are a total of 4 self-loops defined as:

$$(s_3, s_3 ; U.ConReq \,/\, N.Null)$$

$$(s_3, s_3 ; N.SetupAck \,/\, N.Null)$$

$$(s_3, s_3 ; N.Con \,/\, N.Null)$$

$$(s_3, s_3 ; N.Disc \,/\, N.Null)$$

In this modified specification, a distinguishing sequence called *DS* can be found as:

$$DS = U.ConReq, N.SetupAck, U.Digit, N.Alert, N.Con, N.Disc, U.ClearReq$$

The outputs generated by a correct implementation as response to this distinguishing sequence is different for each state as shown in Figure 3.5.

states	Outputs generated by DS=con_req,setupack,digit,alert,conn,disc,clear_req
s_0	setup, dialtone, info_tone_off, ringback, hear, stop_hearing, null
s_1	null, dialtone, info_tone_off, ringback, hear, stop_hearing, null
s_2	null, null, info_tone_off, ringback, hear, stop_hearing, null
s_3	null, null, info, ringback, hear, stop_hearing, null
s_4	null, null, null, null, hear, stop_hearing, null
s_5	null, null, null, null, null, stop_hearing, null
s_6	null, null, null, null, null, null, null

Figure 3.5. Output sequence generated as response to the DS at each state by the modified PhoneDTE given in Figure 3.4.

Let us apply the distinguishing sequence method to the modified specification of PhoneDTE (Figure 3.4) by using the above distinguishing sequence. The transfer sequences for each state are:

$T(s_0)$ = *empty sequence*

$T(s_1)$ = *U.ConReq*

$T(s_2)$ = *U.ConReq, N.SetupAck*

$T(s_3)$ = *U.ConReq, N.SetupAck, U.Digit*

$T(s_4)$ = *U.ConReq, N.SetupAck, U.Digit, N.Alert*

$T(s_5)$ = *U.ConReq, N.SetupAck, U.Digit, N.Alert, N.Con*

$T(s_6)$ = *U.ConReq, N.SetupAck, N.Prog.*

A synchronizing sequence *S* is *U.ClearReq*.

In this case, the test sequence for testing the state transitions defined for state s_2 is given in Figure 3.6. The tests for the remaining states can be generated similarly.

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
	Apply *S*:			
1	S_0	S_0	U.ClearReq	N.Null
	Apply *T*(S_2):			
2	S_0	S_1	U.ConReq	N.Setup
3	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
4<--	S_2	S_2	**U.ConReq**	**N.Null**
	Apply *DS*:			
5	S_2	S_2	U.ConReq	N.Null
6	S_2	S_2	N.SetupAck	N.Null
7	S_2	S_3	U.Digit	U.InfoToneOff
8	S_3	S_4	N.Alert	U.RingBack
9	S_4	S_5	N.Con	U.ConAck
10	S_5	S_6	N.Disc	U.StopCon
11	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
12	S_0	S_0	U.ClearReq	N.Null
	Apply *T*(S_2):			
13	S_0	S_1	U.ConReq	N.Setup
14	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
15<--	S_2	S_2	**N.SetupAck**	**N.Null**
	Apply *DS*:			
16	S_2	S_2	U.ConReq	N.Null
17	S_2	S_2	N.SetupAck	N.Null
18	S_2	S_3	U.Digit	U.InfoToneOff
19	S_3	S_4	N.Alert	U.RingBack
20	S_4	S_5	N.Con	U.ConAck
21	S_5	S_6	N.Disc	U.StopCon
22	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
23	S_0	S_0	U.ClearReq	N.Null
	Apply *T*(S_2):			
24	S_0	S_1	U.ConReq	N.Setup
25	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
26<--	S_2	S_3	**U.Digit**	**U.InfoToneOff**
	Apply *DS*:			
27	S_3	S_3	U.ConReq	N.Null
28	S_3	S_3	N.SetupAck	N.Null
29	S_3	S_3	U.Digit	N.Null
30	S_3	S_4	N.Alert	U.RingBack
31	S_4	S_5	N.Con	U.ConAck
32	S_5	S_6	N.Disc	U.StopCon
33	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
34	S_0	S_0	U.ClearReq	N.Null
	Apply *T*(S_2):			
35	S_0	S_1	U.ConReq	N.Setup
36	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
37<--	S_2	S_2	**N.Alert**	**N.Null**
	Apply *DS*:			

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
38	S_2	S_2	U.ConReq	N.Null
39	S_2	S_2	N.SetupAck	N.Null
40	S_2	S_3	U.Digit	U.InfoToneOff
41	S_3	S_4	N.Alert	U.RingBack
42	S_4	S_5	N.Con	U.ConAck
43	S_5	S_6	N.Disc	U.StopCon
44	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
45	S_0	S_0	U.ClearReq	N.Null
	Apply $T(S_2)$:			
46	S_0	S_1	U.ConReq	N.Setup
47	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
48<--	S_2	S_2	**N.Con**	**N.Null**
	Apply *DS*:			
49	S_2	S_2	U.ConReq	N.Null
50	S_2	S_2	N.SetupAck	N.Null
51	S_2	S_3	U.Digit	U.InfoToneOff
52	S_3	S_4	N.Alert	U.RingBack
53	S_4	S_5	N.Con	U.ConAck
54	S_5	S_6	N.Disc	U.StopCon
55	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
56	S_0	S_0	U.ClearReq	N.Null
	Apply $T(S_2)$:			
57	S_0	S_1	U.ConReq	N.Setup
58	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
59<--	S_2	S_2	**N.Disc**	**N.Null**
	Apply *DS*:			
60	S_2	S_2	U.ConReq	N.Null
61	S_2	S_2	N.SetupAck	N.Null
62	S_2	S_3	U.Digit	U.InfoToneOff
63	S_3	S_4	N.Alert	U.RingBack
64	S_4	S_5	N.Con	U.ConAck
65	S_5	S_6	N.Disc	U.StopCon
66	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			
67	S_0	S_0	U.ClearReq	N.Null
	Apply $T(S_2)$:			
68	S_0	S_1	U.ConReq	N.Setup
69	S_1	S_2	N.SetupAck	U.DialTone
	Test edge:			
70<--	S_2	S_0	**U.ClearReq**	**N.Disc**
	Apply *DS*:			
71	S_0	S_1	U.ConReq	N.Setup
72	S_1	S_2	N.SetupAck	U.DialTone
73	S_2	S_3	U.Digit	U.InfoToneOff
74	S_3	S_4	N.Alert	U.RingBack
75	S_4	S_5	N.Con	U.ConAck
76	S_5	S_6	N.Disc	U.StopCon
77	S_6	S_0	U.ClearReq	N.Null
	Apply *S*:			

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
78	s_0	s_0	U.ClearReq	N.Null
	Apply $T(s_2)$:			
79	s_0	s_1	U.ConReq	N.Setup
80	s_1	s_2	N.SetupAck	U.DialTone
	Test edge:			
81<--	s_2	s_2	**N.Prog**	**N.Null**
	Apply *DS*:			
82	s_2	s_2	U.ConReq	N.Null
83	s_2	s_2	N.SetupAck	N.Null
84	s_2	s_3	U.Digit	U.InfoToneOff
85	s_3	s_4	N.Alert	U.RingBack
86	s_4	s_5	N.Con	U.ConAck
87	s_5	s_6	N.Disc	U.StopCon
88	s_6	s_0	U.ClearReq	N.Null

Figure 3.6. Test sequence generated by the distinguishing sequences method for the state transitions defined for state s_2 of modified PhoneDTE specification given in Figure 3.4.

The format used in the test sequence table given in Figure 3.6 is similar to the one used in Figure 3.2. Before testing each state transition, the synchronizing sequence is applied to the implementation (for example, in Steps 1, 12, 23, in Figure 3.6) followed by the transfer sequence $T(s_2)$ to bring the implementation into state s_2 (for example, in Steps 2 and 3 in Figure 3.6). Then the input/output operation to be tested is performed, marked by an arrow "<--" in Figure 3.6. Finally, the distinguishing sequence is applied to the implementation to verify that the state transition to be tested moved the implementation into the correct state (for example, Steps 5 through 11 in Figure 3.6). During testing, the implementation must generate the outputs as expected, otherwise an error is detected.

The test sequences for the state transitions defined for the remaining states can be generated similarly. The length of the test sequence for the specification of Figure 3.5 is more than 500. Note that some of the sequences can be overlapped to shorten the final test sequence, however, the length of the test sequence still remains too long for most real-life protocols. For example, the synchronizing sequence chosen for the above example (i.e., $U.ClearReq$) happens to be included in the *DS* as the last input/output operation. In the test sequence of Figure 3.6, the FSM implementation should be in state s_0 after the *DS* is applied; therefore, the synchronizing sequence can be omitted in this case, and Steps 11 and 12, 22 and 23, etc. can be merged.

2.4. Characterizing Sequences Method

The characterizing sequences method was introduced for the FSM specifications that are fully-specified but do not posses a distinguishing sequence [4][5]. A *characterizing set* of a state s_i is a set of input sequences such that, when each sequence is applied to the implementation at state s_i, the set of output

sequences generated by the implementation uniquely identifies state s_i. Each sequence of the characterizing set of state s_i distinguishes state s_i from a group of states (i.e., acts like a partial distinguishing sequence). Therefore, applying all of the sequences in the characterizing set distinguishes state s_i from all other states.

For an FSM-based specification, a set that consists of characterizing sets of every state is called the *characterizing set* of the FSM. The sequences of the characterizing set of an FSM specification is called the *characterizing sequences* of the FSM. The characterizing sequences method is also referred to as *W-method* since, as discussed below, Chow calls the characterizing set as the "$W-set$" in *Testing Software Design Modeled by Finite-State Machines*.

We cannot use the PhoneDTE example of Figure 3.1 to illustrate the application of the characterizing sequences method to the protocol conformance testing, since the FSM specification is not fully-specified. The modified version of this FSM as given in Figure 3.4 has a distinguishing sequence, therefore, we cannot use the modified version either. Instead, let us consider the example FSM specification given in Figure 3.7 which is fully-specified and does not have a distinguishing sequence (as illustrated by Kohavi [4]).

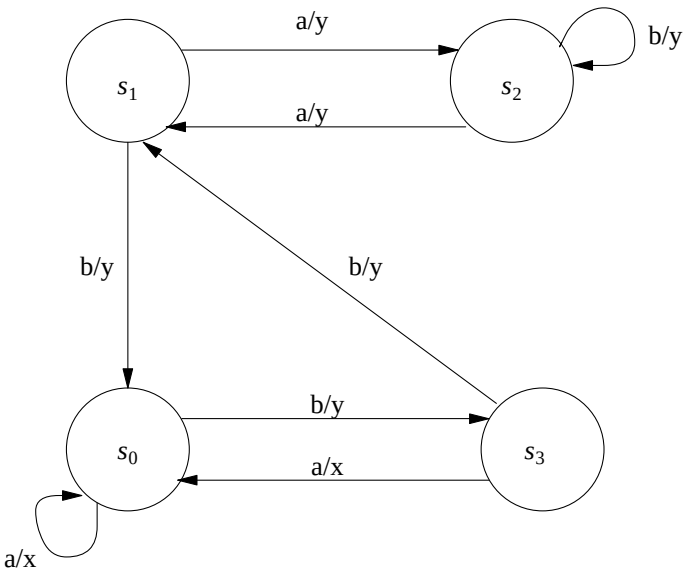

Figure 3.7. An FSM specification that does not posses a distinguishing sequence (initial state is s_0).

A $W-set$ for this FSM specification can be constructed by using the so-called *multiple-experiment* as defined by Kohavi [4]. Consider the input sequence called $W_1 = a, b, a$. The output sequences generated by applying W_1 for each state of the above FSM is shown in Figure 3.8.

starting states	Outputs generated by W_1 = **a, b, a**
s_0	x, y, x
s_1	y, y, y
s_2	y, y, x
s_3	x, y, x

Figure 3.8. Output sequences generated by the FSM of Figure 3.7 as response to W_1.

The output sequence generated by W_1 can identify whether the state of an implementation was either s_1 or s_2 before W_1 is applied, since the outputs for s_1 and s_2 are unique (as shown in Figure 3.8, the output sequences are *y,y,y* and *y,y,x*, respectively). However, W_1 cannot distinguish the state of an implementation if the FSM was initially at either s_0 or s_3 (the output sequence is *x,y,x* for both states as can be seen in Figure 3.8). Now, let us examine the response of implementation to the input sequence called W_2 = *b, a* for each state (Figure 3.9).

starting states	Outputs generated by W_2 = **b, a**
s_0	y, x
s_1	y, x
s_2	y, y
s_3	y, y

Figure 3.9. Output sequences generated by the FSM of Figure 3.7 as response to W_2.

The FSM implementation generates distinct output sequences as response to W_2 for the cases where the implementation was initially at s_0 and s_3 as given in Figure 3.9 (the output sequences are *y,x* and *y,y*, respectively). Therefore, the characterizing set for the FSM specification of Figure 3.7 consists of two input sequences: $W = \{ W_1 , W_2 \}$

A method to apply the characterizing sequences to designs that can be modeled as FSMs is reported in Chow's paper. Chow defines two sets of sequences, called $P-$ and $W-sets$. $P-set$ includes all paths required to reach every state (i.e., input sequences that start from the initial state of the FSM and bring it into each state). In other words, $P-set$ corresponds to the transfer sequences defined by Kohavi [4] and Bhattacharyya [5], and $W-set$ is the characterizing set of the FSM. Chow puts a special emphasis for estimating the number of states that an implementation contains. Recall that the number states in an implementation may be equal to or larger than the number of states defined in the specification; a $W-set$ is called a $Z-set$ if the implementation has extra states. Since the information that yields the number of states of a protocol implementation is not available to the test designer, the number of states in an implementation should be assumed to be the number of states defined in the specification. Any guesses on what the number of states in an implementation may be does not have any practical base simply because the implementation is a black-box.

The controllability problem that we discussed in Section 2.1 is addressed by introducing the partial paths (i.e., $P-set$) in the characterizing sequences method, but the resulting test sequence is potentially too long for most real-life protocols. $W-set$ is primarily designed to solve the observability problem. The main advantage of this method is that every minimal and fully-specified FSM specification (see Section 4.1 in Chapter I for definitions) has a $W-set$. This makes the characterizing sequences an alternative method if a specification does not have a distinguishing sequence. However, by definition, every distinguishing sequence is a characterizing sequence. Therefore, the disadvantages of the distinguishing sequences method are inherent in the characterizing sequences. First, many real-life protocols do not possess distinguishing sequences (partly because they are not always fully-specified). Second, the length of the final test sequence is large. In addition, since there is a set of input sequences to be applied to verify the state of an implementation (as opposed to a single distinguishing sequence), the test sequences generated by the characterizing sequences method are typically much longer than those generated by the distinguishing sequences.

At this point, we refer back to a paper cited in Chapter I titled *A Useful FSM Representation For Test Suite Design and Development* by Kanungo et al. In addition to giving an example of using state transition tables as the form of protocol specification, this paper also presents a case study for constructing the partial paths (i.e., *P-set*) defined in the characterizing sequences method. The protocol that the authors considered is called LAP-B, a Data-Link Layer protocol. The tests that are derived from the specification of LAP-B are presented by using the standardized TTCN graphic syntax (i.e., tables).

Applying the characterizing sequences method, the basic test procedure for testing a state transition (s_i, s_j ; $input_k$ / $output_l$) becomes:

Repeat the following procedure for each input sequence of $W - set$:

Step I: Bring the implementation into state s_i. Similar to the case of distinguishing sequences, this step can be expanded as follows:

I.a: Apply the synchronizing sequence.

I.b: Apply the transfer sequence for s_i.

Step II: Apply the input called $input_k$ and observe that the implementation generates the output called $output_l$.

Step III: Apply an input sequence from $W - set$ and verify that the output sequence is as expected.

For the example of Figure 3.7, the test sequence generated by characterizing sequences method is as follows:

Transfer sequences: $T(s_1) = b, b$; $T(s_2) = b, a$; $T(s_3) = b$

A synchronizing sequence is $S = b, a, b, a, b, a$

Characterizing set: $W - set = \{W_1 , W_2 \}$, where $W_1 = a, b, a$ and $W_2 = b, a$.

In this case, the input sequence for testing the state transitions defined for state s_3 is given in Figure 3.10.

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
	Apply $T(s_3)$			
1	s_0	s_3	b	y
	Test edge:			
2<--	s_3	s_0	**a**	**x**
	Apply W_1			
3	s_0	s_0	a	x
4	s_0	s_3	b	y
5	s_3	s_0	a	x
	Apply S			
6	s_0	s_3	b	y
7	s_3	s_0	a	x
8	s_0	s_3	b	y
9	s_3	s_0	a	x
10	s_0	s_3	b	y
11	s_3	s_0	a	x
	Apply $T(s_3)$			
12	s_0	s_3	b	y
	Test edge:			
13<--	s_3	s_0	**a**	**x**
	Apply W_2			

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
14	s_0	s_3	b	y
15	s_3	s_0	a	x
	Apply S			
16	s_0	s_3	b	y
17	s_3	s_0	a	x
18	s_0	s_3	b	y
19	s_3	s_0	a	x
20	s_0	s_3	b	y
21	s_3	s_0	a	x
	Apply $T(s_3)$			
22	s_0	s_3	b	y
	Test edge:			
23<--	s_3	s_1	**b**	**y**
	Apply W_1			
24	s_1	s_2	a	y
25	s_2	s_2	b	y
26	s_2	s_1	a	y
	Apply S			
27	s_1	s_0	b	y
28	s_0	s_0	a	x
29	s_0	s_3	b	y
30	s_3	s_0	a	x
31	s_0	s_3	b	y
32	s_3	s_0	a	x
	Apply $T(s_3)$			
33	s_0	s_3	b	y
	Test edge:			
34<--	s_3	s_1	**b**	**y**
	Apply W_2			
35	s_1	s_0	b	y
36	s_0	s_0	a	x

Figure 3.10. Test sequence obtained by the W-method for the state transitions defined for state s_3 of the FSM given in Figure 3.7.

In Figure 3.10, we use the same format as in Figure 3.2. In order to test the state transitions defined for state s_3 of the FSM specification of Figure 3.7, the tester first applies a transfer sequence (assuming that the implementation is initially at state s_0) to bring the implementation into state s_3 (for example, in Step 1 of Figure 3.10). Then the state transition to be tested at state s_3 is performed, which is denoted by an arrow "-->" in Figure 3.10. W_1 is applied to check the state of the implementation (for example, Steps 3, 4 and 5 in Figure 3.10). At this point, the state transition is only partially tested since W_1 is not enough to identify the state of an implementation. The synchronizing sequence (for example, Steps 6 through 11) followed by the transfer sequence of s_3 are then applied to bring the implementation into the initial state and into state s_3, respectively. The test is repeated for the same edge by using W_2. If all outputs received from the implementation are as defined by the specification, the state transition test is completed successfully.

The remaining state transitions of the FSM specification are tested in a similar manner. Note that the total length of the test sequence for the simple example specification of Figure 3.7 exceeds 140 input/output operations.

In *Test Selection Based on Finite State Models*, Fujiwara et al. gives a version of *W*-method, called the "*partial W-method*" where the length of the total test sequence is shortened. The authors propose the savings in the final test length by shortening the $W - set$. The inputs are excluded from the $W - set$ if they are not generating different outputs at different states (hence they are not giving any useful information).

2.5. Unique Input/Output Sequences Method

A *unique input/output (UIO) sequence for a state* s_i is an input sequence such that its output sequence uniquely identifies state s_i. In a given FSM specification with n states, there are at least n UIO sequences, one for each specified state (some states may have more than one UIO sequence) if the specification is strongly-connected and minimal (see the formal definitions in Section 2 of Chapter I).

Before discussing the UIO sequences method, let us consider the main difference between the UIO sequences and the distinguishing and the characterizing sequences. Although the goal of the distinguishing, characterizing and UIO sequences methods is to address the observability problem in testing (by identifying the current state of an implementation), the UIO sequences method approaches the problem from a fundamentally different perspective.

During Step III of the basic test procedure for a state transition of $(s_i, s_j; input_k / output_l)$, the distinguishing and the characterizing sequences methods identify the new state of the implementation without using the knowledge that the new state is expected to be s_j. On the other hand, the UIO sequences method takes advantage of the fact that the expected state of the implementation (i.e., state s_j) is known by the test designer. From the conformance testing point of view, during Step III of the basic test procedure, all a test designer needs to know is whether the new state of the implementation is s_j. The UIO sequences are designed such that they can only identify that the new state is the one expected. If an implementation is not in the expected state, this method does not give any further information about the state of the implementation, other than declaring the test verdict as fail. However, in the cases of the distinguishing and the characterizing sequences methods, test designer knows the new state of the implementation – expected or not.

The UIO sequences technique was introduced by Sabnani and Dahbura in *A Protocol Test Generation Procedure* where the authors show that every minimal FSM specification has UIO sequences for each state defined in the specification (at least one UIO sequence per state). For example, the UIO sequence of state s_2 of the specification PhoneDTE of Figure 3.1 is $U.Digit/U.InfoToneOff$. We can verify from the PhoneDTE specification that no other state generates the output of $U.InfoToneOff$ as response to

input $U.Digit$, but only state s_2. Therefore, to verify that an implementation is in state s_2, tester applies the UIO sequence of state s_2 to the implementation. If the output is $U.InfoToneOff$, it is concluded that the implementation was in state s_2 before the UIO sequence is applied. As noted earlier, if the implementation generates a different output, tester cannot know what the state of the implementation was and declares the test as a fail. The UIO sequences for the states of PhoneDTE are given in Figure 3.11.

starting state	**UIO sequences**
s_0	U.ConReq/N.Setup
s_1	N.SetupAck/U.DialTone
s_2	U.Digit/U.InfoToneOff
s_3	U.Digit/N.Info
s_4	N.Con/U.ConAck
s_5	N.Disc/U.StopCon
s_6	U.ClearReq/N.Null

Figure 3.11. UIO sequences for the PhoneDTE specification of Figure 3.1.

By using the UIO sequences method, the basic test procedure for testing a state transition implementation specified as $(s_i, s_j; input_k / output_l)$ becomes:

Step I Bring the implementation into state s_i.

Step II Apply the input called $input_k$ and observe that the implementation generates the output called $output_l$.

Step III Apply the UIO sequence of state s_j and verify that the output sequence is as expected.

The UIO sequences approach, as opposed to the distinguishing and characterizing sequences methods, does not require a fully-specified protocol specification. It also results in much shorter sequences since only a specific state information is provided. The main disadvantage is that tester does not know the state of the implementation in case of a failed test.

Aho et al. introduced an optimization technique to minimize the length of the test sequence in *An*

Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. This technique, based on the approach introduced by Uyar and Dahbura, uses the graph theoretic concept called the *rural Chinese postman problem* – a more general form of the Chinese postman problem, where the postman is required to deliver the mail to only certain streets in a town which have mail-boxes (as opposed to every street) by walking through each street with a mail-box a minimum number of times. This approach is adopted to protocol conformance testing by modeling a specification as a directed graph with two different sets of edges: one set representing the original state transitions defined for by specification (optional edges) and the other set representing the combination of the original edges and the UIO sequences (mandatory edges). In this case, a rural Chinese postman tour traverses every mandatory edge of the graph with the minimum cost (cost is an integer value associated with each edge as defined in the Chinese postman problem).

As reported by Aho et al., the rural Chinese postman algorithm is applicable to specifications that satisfy two sufficient, but not necessary, conditions: the specification has at least one self-loop (see Section 4.1 in Chapter I) defined for every state, and the so-called *reset* feature. The reset feature is defined in the paper as an input/output operation defined for every state (not necessarily with the same input or output message) that brings the protocol into its initial state. A minimum-length test sequence for the PhoneDTE specification of Figure 3.1 generated by using the rural Chinese postman algorithm and the UIO sequences of Figure 3.11 is shown in Figure 3.12. The format of the test sequence is the same as in Figure 3.3. The steps where a state transition is tested are marked by arrows. Each state transition test consists of two consecutive steps. The first one corresponds to sending and receiving the input and output messages, and the second verifying the new state of the implementation. For example, in Figure 3.12, Steps 11 and 12 test the state transition of (s_4, s_5 ; $N.Con$ / $U.ConAck$). At Step 11, the tester sends *N.Con* to the implementation and expects *U.ConAck* as response. After Step 11, the implementation is expected to be in state s_5, which is verified at Step 12 by applying the UIO sequence of s_5 to the implementation (i.e., *N.Disc/U.StopCon* from Figure 3.11).

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
1 <--	s_0	s_1	**U.ConReq**	**N.Setup**
2	s_1	s_2	**N.SetupAck**	**U.DialTone**
3 <--	s_2	s_0	**U.ClearReq**	**N.Disc**
4	s_0	s_1	**U.ConReq**	**N.Setup**
5 <--	s_1	s_0	**U.ClearReq**	**N.Disc**
6	s_0	s_1	**U.ConReq**	**N.Setup**
7 <--	s_1	s_2	**N.SetupAck**	**U.DialTone**
8	s_2	s_3	**U.Digit**	**U.InfoToneOff**
9 <--	s_3	s_3	**U.Digit**	**N.Info**
10	s_3	s_4	**N.Alert**	**U.RingBack**
11<--	s_4	s_5	**N.Con**	**U.ConAck**

TEST SEQUENCE TABLE				
STEP	**CURRENT STATE**	**NEXT STATE**	**MSG TO IUT**	**MSG FROM IUT**
12	s_5	s_6	**N.Disc**	**U.StopCon**
13<--	s_6	s_0	**U.ClearReq**	**N.Null**
14	s_0	s_1	**U.ConReq**	**N.Setup**
15	s_1	s_2	N.SetupAck	U.DialTone
16<--	s_2	s_3	**U.Digit**	**U.InfoToneOff**
17	s_3	s_4	**N.Alert**	**U.RingBack**
18<--	s_4	s_6	**N.Disc**	**U.StopRingBack**
19	s_6	s_0	**U.ClearReq**	**N.Null**
20	s_0	s_1	U.ConReq	N.Setup
21	s_1	s_2	N.SetupAck	U.DialTone
22	s_2	s_3	U.Digit	U.InfoToneOff
23<--	s_3	s_4	**N.Alert**	**U.RingBack**
24	s_4	s_5	**N.Con**	**U.ConAck**
25<--	s_5	s_6	**N.Disc**	**U.StopCon**
26	s_6	s_0	**U.ClearReq**	**N.Null**
27	s_0	s_1	U.ConReq	N.Setup
28	s_1	s_2	N.SetupAck	U.DialTone
29	s_2	s_3	U.Digit	U.InfoToneOff
30<--	s_3	s_6	**N.Prog**	**U.Announce**
31	s_6	s_0	**U.ClearReq**	**N.Null**
32	s_0	s_1	U.ConReq	N.Setup
33	s_1	s_2	N.SetupAck	U.DialTone
34	s_2	s_3	U.Digit	U.InfoToneOff
35	s_3	s_4	N.Alert	U.RingBack
36	s_4	s_5	N.Con	U.ConAck
37<--	s_5	s_0	**U.ClearReq**	**N.Disc**
38	s_0	s_1	**U.ConReq**	**N.Setup**
39	s_1	s_2	N.SetupAck	U.DialTone
40	s_2	s_3	U.Digit	U.InfoToneOff
41	s_3	s_4	N.Alert	U.RingBack
42<--	s_4	s_0	**U.ClearReq**	**N.Disc**
43	s_0	s_1	**U.ConReq**	**N.Setup**
44	s_1	s_2	N.SetupAck	U.DialTone
45	s_2	s_3	U.Digit	U.InfoToneOff
46<--	s_3	s_0	**U.ClearReq**	**N.Disc**
47	s_0	s_1	**U.ConReq**	**N.Setup**
48	s_1	s_2	N.SetupAck	U.DialTone
49<--	s_2	s_6	**N.Prog**	**U.Announce**
50	s_6	s_0	**U.ClearReq**	**N.Null**

Figure 3.12. A minimum-length test sequence for PhoneDTE by using the rural Chinese postman tours and the UIO sequences of Figure 3.11.

There are recent developments on the UIO sequences method to further reduce the length of the test sequences. For example, Shen et al. [6] show that if there is more than one UIO for a state, these additional UIO sequences can be used to obtain test sequences shorter than obtained by the technique given by Aho et al. (which generates minimum-cost tours by using a single UIO sequence per state). Chen et al. [7] introduce a heuristic method to reduce the length of the test sequences by exploiting the fact that the state transitions of a UIO sequence which implement Step 3 of the basic test procedure can be also utilized (where possible) as Step 2 of the procedure. In other words, for a given set of state transition tests, the input/output messages required in Steps 2 and 3 of the basic test procedure may overlap for several state transition tests. The heuristic algorithm of Chen et al. tries to reduce the possible redundancies in the final test sequence.

In *Generating Minimal Length Test Sequences for Conformance Testing of Communication Protocols*, Miller and Paul further investigate the shortening of the test sequence length by combining several techniques. They use multiple UIO sequences and segment overlapping to obatin a shorter final sequence. Also, the computational complexity analysis of their heuristic is noteworthy.

The reader is encouraged to study these techniques since each enhancement is a step towards compacting the length of test sequences, which is one of the major concerns in testing complex protocols with up to thousands of features to test.

2.6. Overview of Test Generation Techniques:

In Chapter I, we identified several aspects of the problem of testing implementations for their conformance to their specifications. Even our simplistic calculator example (Section 2 of Chapter I) is enough to point out the difficulties of this problem. When there are several different techniques for conformance test generation a question that naturally comes to mind is whether one technique is better than another. Defining effectiveness of each technique (at least relative to each another), however, requires the formidable task of defining a model for *fault coverage*.

Several studies are reported in the literature addressing the fault coverage of the various test generation techniques. In *Fault Coverage of Protocol Test Methods*, Sidhu and Leung present a fault model based on Monte Carlo simulation technique for estimating the fault coverage of several test generation methods. They note that a specification with n states, i inputs, and o outputs can have $(n.o)^{(n.i)}$ different implementations; therefore, examining all such implementations is impossible. The authors introduce ten different *classes* of randomly faulty specifications, each can be obtained by random alteration of a given specification. For example, *Class 1* faults consist of randomly altering an output operation in a given specification. The transition tour, distinguishing, characterizing and UIO sequences methods are

compared for their ability to detect these ten classes of faulty specifications. For partially-specified protocols, the authors conclude that all methods, except for the transition tour method, can detect all single faults (as opposed to several faults) introduced in a given specification. It is also shown in the paper that (however, without a formal proof) distinguishing, characterizing and UIO sequences methods have the same fault detection capability. Another study, similar to the one by Sidhu and Leung, is reported by Dahbura and Sabnani for the UIO sequences method [8].

Although we cannot give a widely-accepted model for measuring the fault coverage, we can at least classify the formal test generation techniques discussed in Sections 2.2 through 2.5 based on their ability to address the controllability and observability issues. Recall that the controllability issue deals with bringing an IUT into the desired state where a test is to be conducted, whereas the observability issue is the identification of the state of an IUT after a state transition takes place (i.e., Steps I and III of the basic test procedure given in Section 2.1, respectively). In general, the transition tour approach mainly deals with the controllability problem, unless the protocol has special *status* feature. The distinguishing, characterizing and UIO sequences techniques address the observability issue. The optimization techniques introduced by Uyar-Dahbura and Aho et al. try to close the gap between the observability and controllability problems since they take into account both the state verification and the total length of the test sequence.

3. Test Generation Based on Formal Description Techniques

Formal description techniques (FDTs) are defined by the standards organizations to achieve several goals: minimizing the ambiguities in protocol specifications, automating the implementation process, and, consequently, automating the test generation for such specifications. Ideally, these goals promote the fulfillment of several difficult tasks simultaneously:

- Formal specifications that are precise enough to be directly implemented can significantly simplify the implementation (almost all FDTs have compilers to generate executable code).
- Products can be implemented by different manufacturers to achieve the principles of an Open Systems Interconnection environment;
- Interoperability of products that are implemented by different manufacturers can be enhanced;
- Protocol engineers can meet the demands of versatile user services by specifying protocols as complex as necessary (after all, they can be thoroughly tested);

Typically, the task of defining formal specifications that can be efficiently implemented requires that a specification consists of a number of relatively small independent modules, each of which is easy to implement, to modify and to test. These modules interact with each other to deliver the services defined by the specification. However, if the specification is not designed cautiously, once such modules are placed within a single implementation, the interactions among these modules can make the behavior of

the entire implementation uncontrollable and unobservable under certain conditions. Although conformance testing uses the black-box approach (hence only the externally observable and controllable behavior is testable), some portions of the protocol may not be testable despite the fact that they are formally specified.

Another important issue regarding the specifications that are written in the form of several communicating processes is to obtain the combined behavior of these processes. The global process that represents the overall observable behavior of the specification may be extremely large (for some constructs defined in the FDTs, infinitely large). The problem of verification of the correctness of a specification (i.e., the absence of logical errors, deadlocks, livelocks, inconsistencies, etc.) can become intractable for specifications using even moderately small number of communicating processes [9]-[14]. Therefore, when defining specifications, protocol engineers have to keep in mind the verification issue as well as the testability.

The trade-off between the efficient implementation and effective testing is reflected in the respective results that are reported in the literature. Often, the papers that address one aspect ignore the other. In this section, we give examples of test generation methods based on the FDTs; we refrain from any comparison of the results since most of the techniques are experimental and the general solution is an open research problem at this point. We, however, encourage the reader to study the references given in each paper for insightful information in this field.

The test generation techniques discussed in Section 2 are mainly targeted for the specifications that are written in the form FSMs. These techniques can be also *directly* applicable to EFSMs and FDTs if certain limitations are imposed on the specifications. The main issue is the potential loss of controllability and observability of an implementation because of the use of EFSMs and FDTs, which can make testing extremely difficult. The use of spontaneous transitions (i.e., the state transitions that take place without external stimuli and, therefore, cannot be controlled externally by the tester) and transient states (i.e., the states where an implementation stays only a relatively short amount of time and moves to another state without an external stimulus) can make an implementation untestable.

Below, we discuss several papers regarding the test generation based on FDTs, all of which address the difficulty of test generation based on the FDTs. Almost all the authors use limited versions of the current FDTs to demonstrate the advances made towards a solution of automated testing of formal specifications. More research is needed for a general solution to test implementations that are specified by using all capabilities of the current FDTs. Recently, ISO and CCITT have accelerated their efforts by forming study groups and committees on formal methods in conformance testing, which emphasizes the importance and urgency of this problem.

In *TESDL: Experience with Generating Test Cases from SDL Specifications*, Brömstrup and Hogrefe present a heuristic algorithm to derive the global behavior of a protocol as a tree, called an Asynchronous

Communication Tree (ACT), which is based on a restricted set of SDL diagrams. The ACT is the global system description as obtained by reachability analysis by perturbation. In ACT, the nodes (i.e., vertices) represent a global states of the protocol. A global state contains information about the states of all processes in the specification. Tests are derived from the ACT of a specification by a software tool, called TESDL.

A theory for defining the conformance testing for systems that are defined in LOTOS (or, in general, labeled transition systems [15][16]) is given in *LOTOS Specifications, Their Implementations and Their Tests* by Brinksma et al. The authors define a notion called a "*canonical tester*" which detects the discrepancies between a given specification and its implementation. An algorithmic way to construct the canonical testers is an open problem. This paper represents a formal framework for defining various fundamental concepts in the field of conformance testing.

A more practical approach for LOTOS specifications can be found in *Derivation of Test Cases for LAP-B from a LOTOS Specification* by Gueraichi and Logrippo. The paper reports a case study for applying the UIO sequences method to a real-life protocol specification in LOTOS where a state-oriented approach is used. The authors derive the "*execution trees*" from the LOTOS specification by removing the LOTOS operators such as parallel composition and disable. The execution trees represent all possible execution sequences, some of which may be invalid (e.g., they may have incompatible or conflicting variable values). Resulting conformance tests for LAP-B (a Data-Link Layer protocol) are comparable to those generated by state transition tables. Although the authors note that the test selection algorithm is based on a heuristic and may not be feasible for more complex and detailed specifications, this paper is one of the encouraging examples of the effort of combining a formal specification and test generation techniques.

In *A Test Design Methodology for Protocol Testing* by Sarikaya et al., protocols specifications that are defined in Estelle are considered. An Estelle specification is first translated into a *normalized form* where procedure/function calls and conditional statements are replaced with equivalent with in-line code constructs. Based on the normalized specification the control and data flow graphs are obtained. The control graph represents the state changes based on input values and spontaneous transitions. The data flow represents the relationships among the inputs, context variables, functions and outputs. Each node (vertex) in the data flow graph represents either an input, variable, function or output, and the edges correspond to the information flow among those nodes. The authors show that the control graph can be tested by using various techniques that we discussed in Section 2 of this chapter. The data flow, on the other hand, is first partitioned into a set of "blocks" each of which shows the flow for a single variable; the blocks that are dependent (e.g., the variables that are related to the same inputs) are then merged into larger blocks to shorten the test sequences. The resulting data flow graph can be tested by using the techniques of Section 2 if the test designer can supply necessary parameter values for each path of the

data flow graph. A similar study for Estelle specifications also exists [17].

In *A Test Derivation Method For Protocol Conformance Testing,* Ural applies software engineering techniques to protocol conformance testing. The technique that Ural describes is applicable to the protocol specifications given in Estelle as a "*normal form specification*" (NFS), an easy-to-analyze, single-module form of Estelle (see the paper for the formal definition of NFS). The NFS is similar to the *normalized form* given in *A Test Design Methodology for Protocol Testing* by Sarikaya et al. The NFS of a protocol is translated into a graph: the vertices are called *s-*, *i-*, and *t-nodes* representing the states, inputs and the statement blocks of the NFS, respectively; the edges among these vertices are defined as *si-*, *it-*, *ts-*, and *st-* edges. Covering all nodes, edges and paths of this graph correspond to statement, branch and path coverage, respectively, as defined in software engineering. Test derivation technique discussed in the paper is based on selecting the paths covering the variables used in predicates or computational statements. Ural suggests that this method is a complementary to those discussed in Section 2 and this section.

4. Application of Formal Test Generation Methods to OSI Standards

OSI conformance testing standards focus on the issues related to the various methods for conducting conformance testing as outlined in Section 2 of Chapter II. The assumption of the standardization effort is that the test designer has already developed a set of test purposes (usually by manual methods) to perform on an IUT. On the other hand, formal techniques for test generation do not concentrate on the issues regarding the realization of tests in a test laboratory. This section points out the applicability of the concepts developed by the OSI conformance standards to the formal test generation methodologies, and vice versa.

In Section 4.1, we discuss the definitions of test purposes and corresponding abstract test cases which specify the ASP/PDU exchanges between an IUT and upper/lower testers that are needed to implement test purposes. Then the test purposes for our example protocol of Figure 3.1 are defined based on its formal specification. TTCN graphic syntax (i.e., tables) is used to describe the abstract test cases. For more information regarding TTCN, we suggest the paper included in Chapter II by Probert and Monkewich entitled *TTCN: The International Notation for Specifying Tests of Communications Systems*. While we focus on behavior tests, the reader should keep in mind that basic interconnection and capability tests are subsets of behavior tests. In Section 4.2, test system methods defined by the standards are studied in terms of their effect on the abstract test suite. In Section 4.3, the applicability of test generation techniques to improve execution time and effectiveness of conformance tests is discussed.

4.1. Abstract Test Cases

Recall our discussion in Chapter II regarding various types of tests defined by the conformance testing standard. Among these tests, the behavior tests constitute the major part of conformance tests. They

include the tests for valid, inopportune and invalid PDUs, timers, etc. Although effectiveness in terms of error detection capability and run-time efficiency may vary for each technique, all of the formal test generation techniques previously discussed in this chapter are applicable to behavior tests if the behavior of the protocol can be expressed in the model that the formal techniques require (with the possible exception of tests for invalid PDUs). Basic interconnection and capability tests are subsets of the behavior tests, and, therefore, can be combined within the model for the behavior tests.

Specifications written as FSMs are natural candidates for the formal test generation techniques. They are also applicable to SDL- and Estelle- based specifications, which either consist of a single process or can be combined into a single global process, and EFSM-based specifications if the specification does not include transient states and spontaneous transitions (as defined in Chapter I). Such restrictions are due to the testers' need to control variables and parameter values that influence the external behavior of an implementation. State-oriented LOTOS specifications are also possible candidates for the formal test generation techniques. Although the discussions in this section are primarily based on specifications written as FSMs, they are also applicable to all specifications satisfying the above-mentioned criteria.

As stated in the conformance testing standard, before constructing abstract test cases for behavior tests, test purposes need to be defined based on the protocol specification. Test purposes are written to define the objective of a test without describing the specific actions to be taken by the lower or upper testers or the IUT. In general, each input/output action constitutes an edge in a directed graph representing the behavior of the IUT in response to those inputs. Here are two examples of test purposes defined for our example specification of Figure 3.1:

Test purpose for the edge of $(s_1, s_2; N.\mathit{Setupack}/U.\mathit{DialTone})$ *:* Verify that IUT generates U.DialTone at the user interface as response to N.SetupAck from the network interface at state s_1; the IUT is expected to move to state s_2.

Test purpose for the edge of $(s_3, s_4; N.\mathit{Alert}/U.\mathit{RingBack})$ *:* Verify that IUT generates U.RingBack at the user interface as response to N.Alert from the network interface at state s_3; the IUT is expected to move to state s_4.

The following assumptions are made for a hypothetical test system for the purposes of our discussion:

- An implementation of the PhoneDTE specification is the IUT;
- An upper tester assumes the role of the user interface (i.e., every input or output with a prefix of U is handled by the upper tester);
- A lower tester assumes the role of the network interface (i.e., every input or output with a prefix of N is handled by the lower tester);

Once test purposes are defined, abstract test suite designers construct the *abstract test cases* each of

which contains a *preamble*, a *test body* and a *postamble* based on the test purposes. Recall from Chapter II that the notions of preamble, test body and postamble constitute a conceptual framework where the preambles and postambles are optional. Each notion is useful in organizing a test suite into hierarchically structured fragments of text which may be grouped by test purposes and may be used in more than one test case.

TTCN defines an operator called *attach* (denoted as "+" in the tabular representation) that may be considered as a recursive "include" or "copy" directive to incorporate the body of the named text at the point the *attach* operator appears in the text (see Chapter II for more information in TTCN). Thus, a named set of actions by an IUT or a test entity (e.g., a preamble, postamble or verification routine) may be included within a test body via an attach directive. An example for the use of *attach* operator can be seen in Figure 3.15.

A preamble consists of a set of actions (i.e., ASP/PDU exchanges between an IUT and upper/lower testers) to bring an IUT into a desired state where the test will be conducted. For example, the preamble for state s_1 of PhoneDTE is that the user interface sends a *U.ConReq* to the IUT, the network interface receives a *N.Setup* and sends a *N.SetupAck* to the IUT, and then the user interface receives a *U.DialTone* from the IUT. The TTCN description of this preamble is shown in Figure 3.13. An upper tester and a lower tester (denoted as *L* and *U*) are used to test the network and user interfaces of an implementation, respectively. In TTCN, "!" and "?" denote sending and receiving a message (ASP or PDU) to/from an IUT, respectively. In this example, a timer (called *T_operator*) is defined to limit the time to wait for a particular message that an IUT is expected to send. The receipt of an unexpected message is shown as *L?OTHERWISE* and *U?OTHERWISE* in Figure 3.13. Expecting no messages from an IUT (i.e., *?TIMEOUT T_operator* in Figure 3.13) is considered to be *inconclusive* and denoted as *(I)* in the verdict column, since the conformance testing standard allows for *fail* verdicts only in a test body, not in a preamble or postamble.

A postamble is defined as the actions to bring an IUT into a pre-defined initial state after a test is conducted. For the example specification of PhoneDTE, sending a *U.ClearReq* to an IUT will put the implementation in state s_0, therefore, constituting a postamble applicable for every state. The TTCN representation of this postamble is given in Figure 3.14. Note that the response of an IUT to *U.ClearReq* in any state (except s_6) is *N.Disc* which is different from the response when the IUT is in state s_6 (*N.Null*). In Figure 3.14, both responses are shown as acceptable. Unexpected responses by the IUT also cause an inconclusive verdict in the postamble as illustrated in Figure 3.14.

A test body consists of two logical parts. The first part specifies the ASP/PDU exchanges (test events) required to satisfy the test purpose. The second part is the procedure to verify the new state of the IUT. The test body in Figure 3.15 corresponds to the first example test purpose defined above for state s_1. Unexpected responses cause the IUT to fail this test case (shown as *(F)* in the verdict column of Figure

3.15). After sending U.DialTone to the upper tester, the attachment of the fragment of text named *VERIFICATION_S2* consists of the ASP/PDU exchanges used to verify that the new state of the IUT is s_2. If the verification routine completes successfully, a *pass* verdict is assigned for this test case (shown as *(P)* in Figure 3.15). For verification, we use the UIO sequence of state s_2 (given in Figure 3.11) in TTCN format as shown in Figure 3.16.

Dynamic Behavior				
Reference	PHONE_DTE/PREAMBLE_S1			
Identifier	PREAMBLE_S1			
Purpose	Necessary steps to place the IUT in test state s_1.			
Default Reference				
Behavior Description	Label	Constraint Reference	Verdict	Comments
PREAMBLE_S1				
U!U.ConReq				Upper Tester sends a ConReq
START T_operator				A timer is started to wait for N.Setup
L?N.Setup				
L!N.SetupAck (CANCEL T_operator)				Lower Tester sends a N.SetupAck
START T_operator				A timer is started to wait for U.DialTone
U?U.DialTone				Upper Tester receives dial tone
?TIMEOUT T_operator			(I)	No output is inconclusive
U?OTHERWISE			(I)	Any other output to Upper Tester is inconclusive
L?OTHERWISE			(I)	Any other output to Lower Tester is inconclusive
?TIMEOUT T_operator			(I)	No output is inconclusive
L?OTHERWISE			(I)	Any other output to Lower Tester is inconclusive
U?OTHERWISE			(I)	Any other output to Upper Tester is inconclusive

Figure 3.13. TTCN representation of state s_1 preamble for PhoneDTE specification.

Dynamic Behavior				
Reference	PHONE_DTE/POSTAMBLE			
Identifier	POSTAMBLE			
Purpose	Necessary steps to bring the IUT in test state s_0 after a test body is run.			
Default Reference				
Behavior Description	Label	Constraint Reference	Verdict	Comments
POSTAMBLE				
U!U.ClearReq				Upper Tester sends a ClearReq
START T_operator				A timer is started to wait for N.Disc or N.Null
L?N.Disc				Lower Tester receives a Disc
?TIMEOUT T_operator				No output is N.Null
L?OTHERWISE			(I)	Any other output to Lower Tester is inconclusive
U?OTHERWISE			(I)	Any other output to Upper Tester is inconclusive

Figure 3.14. TTCN representation of the postamble for PhoneDTE specification.

Dynamic Behavior				
Reference	PHONE_DTE/STATE_TEST/TEST_S1_1			
Identifier	TEST_S1_1			
Purpose	Verify that IUT generates U.DialTone at the user interface as response to N.SetupAck from network interface at state s_1; the IUT is expected to move to state s_2.			
Default Reference				
Behavior Description	Label	Constraint Reference	Verdict	Comments
TEST_S1_1				
+PREAMBLE_S1				Move IUT to state s_1
L!N.SetupAck				Lower Tester sends a N.SetupAck
START T_operator				Upper Tester starts a timer
U?U.DialTone				Upper Tester receives U.DialTone
+VERIFICATION_S2			(P)	Verify new state is s_2
+POSTAMBLE				Move IUT to state s_0
U?OTHERWISE			(F)	Any other output to Upper Tester is a Fail
L?OTHERWISE			(F)	Any other output to Lower Tester is a Fail
?TIMEOUT T_operator			(F)	No output is a Fail

Figure 3.15. TTCN representation of a test case for PhoneDTE specification.

Dynamic Behavior				
Reference	PHONE_DTE/VERIFICATION_S2			
Identifier	VERIFICATION_S2			
Purpose	Verify that IUT is in state s_2.			
	This example implements the UIO sequence of s_2 from Figure 3.9			
Default Reference				
Behavior Description	Label	Constraint Reference	Verdict	Comments
VERIFICATION_S2				
U!U.Digit				Upper Tester sends U.Digit
START T_operator				A timer is started
U?U.InfoToneOff				Upper Tester receives U.InfoToneOff
U?OTHERWISE			(F)	Any other output to Upper Tester is a Fail
L?OTHERWISE			(F)	Any other output to Lower Tester is a Fail
?TIMEOUT T_operator			(F)	No output is a Fail

Figure 3.16. TTCN representation of state $s_1$verification of PhoneDTE specification. This example implements the UIO sequence of s_2 from Figure 3.9.

The implementation options may generate different outputs for a given input at a given state. TTCN allows for different responses for an input as shown in the example of the postamble for PhoneDTE specification (Figure 3.14), where the IUT response depends on its current state.

4.2. OSI Abstract Test Methods

The abstract test methods described in the conformance testing standard [1] (i.e., local, distributed, coordinated, and remote methods) are based on the availability of the abstract service primitives (ASPs) at a given point of control and observation (PCO) (see Section 2 in Chapter II). Therefore, the abstract test methods define the model to be used to represent an IUT for the formal test generation methods. Restrictions or additional features are imposed on the IUT model based on the capabilities of the test method used. For example, the remote method requires only the interactions with the lower tester to be modeled, whereas, in the case of distributed method, the model for the same IUT needs additional features to represent the interactions with the upper tester. Therefore, the model representing an IUT to be tested in a remote testbed is different from the model for the same IUT to be tested in a distributed testbed; the latter is a superset of the former. In general, the local method requires the most detailed model since it assumes that all ASPs and PDUs of an IUT can be controlled and observed. The simplest model is for the remote testbed which is the most restrictive (from a tester's point of view) among the abstract test methods.

Let us examine the specification of PhoneDTE to check which test method can be applicable for testing

its implementations. If remote testing method is adopted to test PhoneDTE implementations, as can be seen from Figure 3.1, any state transition that requires interactions with a user interface cannot be tested. Although we could use an operator as an upper tester to generate the inputs sent by the user interface, the operator would not be able to analyze the responses sent to the user interface by the IUT. Recall that an upper tester is a software system capable of sending stimuli to an IUT and analyzing the responses at upper layer interface (i.e., the user interface of PhoneDTE). Therefore, for the PhoneDTE example, the local, distributed or coordinated methods are the only choices. As a consequence, manufacturers of PhoneDTE, however, must provide an accessible interface for the upper tester.

It can be seen from the test case example of Figure 3.15 that a coordination mechanism is required to run the test cases successfully. In Figure 3.15, the upper tester should know when to expect the *U.DialTone* response from the IUT. A coordination protocol that can notify the upper tester after lower tester sends *N.SetupAck* is required so that the upper tester can start *T_operator* timer and wait for *U.DialTone* from the IUT.

4.3. Application of Test Generation Techniques to OSI Test Methods

The concept of controllability found in the formal test generation techniques (e.g., homing sequences, transfer sequences, etc.) corresponds to the existence of the preambles and postambles in the OSI standards. As described in Section 4.1, a preamble brings the IUT into the desired state of a test case from a stable state, and a postamble puts the IUT back into a stable state after the test body is run. Typically, the initial state is defined (or interpreted) as the stable state in an abstract test suite. The solution brought to the controllability problem by the formal test generation methods is the use of various test sequencing methods as discussed in Section 2. Therefore, the test generation methods (for example, optimization techniques based on the Chinese or the rural Chinese postman problems or the test sequencing presented by Sarikaya and Bochmann) can eliminate the overhead of preambles and postambles where applicable.

It is stated in Part 2 of the conformance testing standard that every test case should have a preamble(s) and postamble(s) [1]. While it is possible to execute each test case individually by using a preamble and postamble, the standard does not prohibit the concatenation of *test bodies* (i.e., not utilizing the preamble and postamble of a test case all the times) nor optimizing the organization of a test suite. The standard also does not mandate these options be part of a test suite. For example, if the ending state of a test body is the initial state, there is no need to run a postamble. Therefore, if a test sequencing method is used to order the *test bodies* in a tour manner (i.e., a non-empty sequence of test bodies), the time to run test cases can be significantly reduced. In this case, preambles and postambles are necessary only when an individual test case is to be run (as opposed to a number of them, one after another) or a test case fails and the sequence of test bodies is broken.

In order to incorporate the controllability solution of the formal test generation methods into the abstract

test case definitions, the preambles and postambles can be defined such that they are executed conditionally. For example, each preamble may contain a boolean variable which can be defined as a test suite variable to allow the preambles run conditionally. This variable can be set to bypass a preamble when the previous test run in a test suite results in a "*pass*" verdict. Similarly, the same variable can skip the execution of a postamble if the verdict of a test case is "*pass*."

Suppose we consider applying such a test sequencing technique for the PhoneDTE example. Let us define a test suite parameter called *proceed_tour* and a test suite boolean variable called *prev_verdict*. When a test designer wants to use test sequencing *proceed_tour* is set to *true*; *prev_verdict* is set to *true* if the previous test case in the tour of test bodies results in a *pass* verdict, otherwise to *false*. We have to modify the preambles and postambles originally defined for PhoneDTE to accommodate *proceed_tour* and *prev_verdict*. In Figure 3.17, the modification is given for the preamble of state s_1 (originally given in Figure 3.13). This modified preamble is executed when a test designer (or laboratory) wants to use a test sequencing approach (i.e., *proceed_tour=true*) and the previous test was resulted in a *pass* verdict (i.e., *prev_verdict=true*); if any of these variables are set to *false*, the preamble is executed. The same argument is also valid for the postambles.

Dynamic Behavior				
Reference	PHONE_DTE/PREAMBLE_S1			
Identifier	PREAMBLE_S1			
Purpose	Necessary steps to place the IUT in test state s_1. This preamble will be run only if the previous verdict in the tour is a fail or tester wants to run an individual test.			
Default Reference				
Behavior Description	Label	Constraint Reference	Verdict	Comments
PREAMBLE_S1				
[(proceed_tour) AND (prev_verdict)]				
U!U.ConReq				Upper Tester sends a ConReq
START T_operator				A timer is started to wait for N.Setup
L?N.Setup				
L!N.SetupAck (CANCEL T_operator)				Lower Tester sends a N.SetupAck
START T_operator				A timer is started to wait for U.DialTone
U?U.DialTone				Upper Tester receives dial tone
?TIMEOUT T_operator			(I)	No output is inconclusive
U?OTHERWISE			(I)	Any other output to Upper Tester is inconclusive
L?OTHERWISE			(I)	Any other output to Lower Tester is inconclusive
?TIMEOUT T_operator			(I)	No output is inconclusive
L?OTHERWISE			(I)	Any other output to Lower Tester is inconclusive
U?OTHERWISE			(I)	Any other output to Upper Tester is inconclusive
[NOT((proceed_tour) AND (prev_verdict))]				

Figure 3.17. TTCN representation of state s_1 preamble modified for test sequencing.

For the observability problem (i.e., verifying the state of an IUT), the test generation methods discussed in Section 2 introduce the concepts of the distinguishing, characterizing and UIO sequences. Any of these techniques can be used for the state verification routines in test bodies. In the above example, we have already used the UIO sequences in one of the test cases for the PhoneDTE specification.

Continuing with this example, we can obtain an order to run the test bodies so that the run-time is minimized, if all test bodies result in *pass* verdicts. Any of the optimization techniques discussed in Section 2 can be used for ordering the test bodies which have the preambles and postambles with the conditional test suite parameters (Figure 3.17). Figure 3.18 presents a tour of test bodies generated by the technique of Aho et al. using the rural Chinese postman algorithm. In fact, the test sequence of Figure 3.12 and the tour of test bodies in Figure 3.18 are equivalent, except that the latter is presented in

the terminology adopted by the standards organizations. The UIO sequence of each state s_x is referred to as *VERIFY_sx*. In Figure 3.18, the column called *TEST BODY* identifies which test body is to be run (including the verification routine, but excluding the preamble and postamble of the test case) at that step of the tour. The columns called *PREAMBLE* and *POSTAMBLE* defines the preambles and postambles, respectively, that are needed if a test fails and the tour is broken. The brackets around each preamble and postamble emphasize that they are run conditionally (except for the first preamble, which has to be run to bring the IUT into the initial state at the beginning of the tour). If any of the test cases results in a *fail* verdict, the next step of the tour has to run the preamble to continue the tour. Similarly, a test designer (or laboratory) can set *proceed_tour* to be *false* to run each test case individually (test body together with the preamble and postamble). There are several procedures used in Figure 3.18 (labeled as *proc*) in order to link the test bodies in a minimum-length tour. These procedures are defined based on the specification of PhoneDTE as follows:

proc(s0_to_s1): U.ConReq/N.Setup

proc(s1_to_s2): N.SetupAck/U.DialTone

proc(s2_to_s3): U.Digit/U.InfoToneOff

proc(s3_to_s4): N.Alert/U.RingBack

proc(s4_to_s5): N.Con/U.ConAck

Note that during the execution of the above procedures a *pass/fail* verdict cannot be assigned for an IUT since the procedures are not the part of a test body. They can be viewed as different preambles for a given state to minimize the run-time of the test bodies.

ORDER OF EXECUTION FOR TEST CASES			
STEP	PREAMBLE	TEST BODY	POSTAMBLE
1	+preamble_s0	U.ConReq/N.Setup; verify_s1	[+postamble]
2	[+preamble_s2]	U.ClearReq/N.Disc; verify_s0	[+postamble]
3	[+preamble_s1]	U.ClearReq/N.Disc; verify_s0	[+postamble]
4	[+preamble_s1]	N.SetupAck/U.DialTone; verify_s2	[+postamble]
5	[+preamble_s3]	U.Digit/N.Info; verify_s3	[+postamble]
6	[+preamble_s4]	N.Con/U.ConAck; verify_s5	[+postamble]
7	[+preamble_s6]	U.ClearReq/N.Null; verify_s0	[+postamble]
			+proc(s1_to_s2)
8	[+preamble_s2]	U.Digit/U.InfoToneOff; verify_s3	[+postamble]
9	[+preamble_s4]	N.Disc/U.StopRingBack; verify_s6	[+postamble]
			+proc(s0_to_s1)
			+proc(s1_to_s2)
			+proc(s2_to_s3)
10	[+preamble_s3]	N.Alert/U.RingBack; verify_s4	[+postamble]
11	[+preamble_s5]	N.Disc/U.StopCon; verify_s6	[+postamble]
			+proc(s0_to_s1)
			+proc(s1_to_s2)
			+proc(s2_to_s3)
12	[+preamble_s3]	N.Prog/U.Announce; verify_s6	[+postamble]
			+proc(s0_to_s1)
			+proc(s1_to_s2)
			+proc(s2_to_s3)
			+proc(s3_to_s4)
			+proc(s4_to_s5)
13	[+preamble_s5]	U.ClearReq/N.Disc; verify_s0	[+postamble]
			+proc(s1_to_s2)
			+proc(s2_to_s3)
			+proc(s3_to_s4)
14	[+preamble_s4]	U.ClearReq/N.Disc; verify_s0	[+postamble]
			+proc(s1_to_s2)
			+proc(s2_to_s3)
15	[+preamble_s3]	U.ClearReq/N.Disc; verify_s0	[+postamble]
			+proc(s1_to_s2)
16	[+preamble_s2]	N.Prog/U.Announce; verify_s6	[+postamble]

Figure 3.18. Sequencing test bodies to minimize the run-time.
(This tour is equivalent to the one given in Figure 3.12.)

5. Summary

In this chapter, we introduce the four major techniques reported in the literature for the algorithmic generation of protocol conformance tests based on the extended or basic FSM models . We also discuss the test generation techniques based on the FDTs. The techniques developed for the FSM models (basic or extended) can be also applicable to the models based on FDTs with some restrictions. We present several studies as examples of such efforts. The concepts defined by the conformance testing standard are discussed in terms of their applicability to the conformance test generation methods.

The list of full papers that we include at the end of this chapter is as follows (in the order of appearance):

- B. Sarikaya and G.v. Bochmann, "*Some Experience with Test Sequence Generation for Protocols*," **Proc. Protocol Specification, Testing and Verification II**, C. Sunshine (ed.), North-Holland, 1982, pp.555-567.
- M.H. Sherif, G.L. Hoover and R.P. Wiederhold, "*X.25 Conformance Testing – A Tutorial*," **IEEE Communications Magazine**, Vol. 24, No. 1, Jan. 1986, pp. 16-27.
- M.U. Uyar and A.T. Dahbura, "*Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931*," **Proc. IEEE Global Communications Conf.**, Dec. 1986, pp. 68-72.
- F.C. Hennie, "*Fault Detecting Experiments for Sequential Circuits*," **Proc. Fifth Ann. Symp. on Switching Circuit Theory and Logical Design**, Nov. 1964, pp. 95-110.
- G. Gonenc, "*A Method for the Design of Fault Detection Experiments*," **IEEE Trans. on Computers**, Vol. COM-19, No. 6, June 1970, pp. 551-558.
- W. Hengeveld and J. Kroon, "*Using Checking Sequences for OSI Session Layer Conformance Testing*," **Proc. Protocol Specification, Testing and Verification VII**, H. Rudin and C.H. West (eds.), North-Holland, 1987, pp.435-449.
- T.S. Chow, "*Testing Software Design Modelled by Finite-State Machines*," **IEEE Trans. on Software Engineering**, Vol. SE-4, No. 3, May 1978, pp. 178-187.
- S. Fujiwara, G. v. Bochmann, F. Khendek, M.Amalou, and A. Ghedamsi, "*Test Selection Based on Finite State Models*, **IEEE Trans. on Software Engineering**, Vol. 17, No. 6, June 1991, pp. 591-604.
- K.K. Sabnani and A.T. Dahbura, "*A Protocol Test Generation Procedure*," **Computer Networks**, Vol. 15, No. 4, 1988, pp.295-297.
- A.V. Aho, A.T. Dahbura, D.Lee and M.U. Uyar, "*An Optimization Technique For Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours*," **IEEE Trans. on Communications**, Vol. 39, No. 11, Nov. 1991, pp. 1604-1615.
- R. E. Miller and S. Paul, "*Generating Minimal Length Test Sequences for Conformance Testing of Communication Protocols*, **Proc. IEEE INFOCOM'91**, 1991, pp. 8D.4.1-8D.4.10.
- D. Sidhu and T. Leung, "*Fault Coverage of Protocol Test Methods*," **Proc. IEEE INFOCOM'88**, 1988, pp. 80-85.
- L. Bromstrup and D. Hogrefe, "*TESDL: Experience with Generating Test Cases from DSL Specifications*," **SDL'89: The Language at Work**, North-Holland, 1989, pp. 267-279.
- E. Brinksma, G. Scollo and G. Steenbergen, "*LOTOS Specifications, Their Implementations and Their Tests*," **Proc. Protocol Specification, Testing and Verification VI**, B. Sarikaya and G.v. Bochmann (eds.), North-Holland, 1986, pp.349-360.
- B. Sarikaya, G.v. Bochmann and E. Cerny, *"A Test Design Methodology for Protocol Testing*," **IEEE Trans. on Software Engineering**, Vol. SE-13, No. 5, May 1987, pp. 518-531.
- D. Gueraichi and L. Logrippo, "*Derivation of Test Cases for LAP-B from a LOTOS Specification*," **Proc. of Second Int'l. Conf. on FDTs for Distributed Systems and Communication Protocols**, 1989, pp. 489-508.
- H. Ural, "*A Test Derivation Method For Protocol Conformance Testing*," **Proc. Protocol Specification, Testing and Verification VII**, H. Rudin and C.H. West (eds.), North-Holland, 1987, pp.347-358.

6. Open Research Problems and Further Reading

Algorithms to generate test sequences for protocol implementations constitutes one of the largest bodies of literature in the field of communication protocols. The task of designing algorithms becomes even more challenging when FDTs are utilized, which are the recent solution for writing unambiguous specifications. Currently, even the definition of test purposes is not agreed upon (i.e., what to test in an implementation for its conformance to the specification).

Active research has been pursued over many years for generating test sequences for the specifications that are in the form of an FSM. The results are directly applicable to EFSM- and FDT-based

specifications if certain limitations are imposed. For example, the specifications that avoid spontaneous transitions, and use parameters and variables that can be externally controlled and observed allow the direct application of these techniques. However, one may argue that these restrictions may be too limiting for specifying the ever-growing complexity of the services demanded by users and a range of implementation options. For example, allowing *null* in the *permissible input set* accommodates spontaneous transitions. A *null* input can be interpreted as the action of a test system is to set a timer and wait for a well-defined action and related output (for example, retransmission of a PDU with an expected sequence number). Indeed, the example of retransmission after expiry of an internal timer within the IUT is characteristic of a large class of spontaneous transitions. Thus, excluding spontaneous transitions or transient states fails to address a real-life problem. However, there is no literature which categorizes the use of spontaneous transitions and transient states, and which suggests realistic means to address the related issues of synchronization and control. This topic is an open research problem to be addressed in a framework of algorithmic test generation techniques – although this problem is routinely faced by designers of test systems and test suites, and solved by their intuition and insight.

For the protocol designer, the trade-off is specifying the protocols formally so that they can be precise, and that (at the same time) they can be thoroughly tested. The well-studied techniques of FSM testing are one of the best starting points to be explored and extended to cover the gap between formal specification and testing. Examples of these efforts are discussed in Section 3.

Designing specifications for testability is a major concern. As discussed in Section 2, special features such as *status* significantly reduce the complexity of the testing problem since such a feature is a built-in capability for the state verification step of the basic test procedure defined in Section 2.1. Another issue that needs attention during the definition of specifications is avoiding unsynchronizable state transitions which we have already discussed in Sections 7 and 9 of Chapter II.

One of the open questions that will be answered in time is whether the full capabilities of the current FDTs may make specifications very difficult to verify and to test, hence, threatening the interoperability of implementations – the main purpose of the Open Systems framework. Therefore, it is not surprising that most of the research reported in the literature uses only a small subset of the capabilities offered by the FDTs. Specific examples of this limited use include monolithic models (i.e., specifications that have a single process as opposed to several communicating processes), normal form specifications (somewhat limited versions of Estelle-based specifications), dialects and styles of FDTs convenient to testing (e.g., state-oriented specification style of LOTOS). Only with such simplifications, small contributions can be made towards developing algorithmic test generation procedures. However, we believe that only a combined effort from researchers, developers and standards organizations will eventually bring effective solutions for algorithmic techniques to test real-life protocols that are specified in formal languages.

7. References for Chapter II

[1] *Information Processing Systems – OSI Conformance Testing Methodology and Framework*, ISO/IEC JTC 1, IS 9646, Parts 1-5, 1991.

[2] S. Naito and M. Tsunoyama, "*Fault Detection For Sequential Machines by Transition Tours*," **Proc. 11th IEEE Fault Tolerant Computing Symp.**, 1981, pp. 238-243.

[3] M. K. Kuan, "*Graphics Programming Using Odd or Even Points*," **Chinese Mathematics**, Vol. 1, 1962, pp. 273-277.

[4] Z. Kohavi, "*Switching and Finite Automata Theory*," McGraw Hill, New York, 1978.

[5] A. Bhattacharyya, *Checking Experiments in Sequential Machines*, John Wiley & Sons, New York, 1989.

[6] Y. N. Shen, F. Lombardi and A. T. Dahbura, "*Protocol Conformance Testing by Multiple UIO Sequences*," **Proc. Protocol Specification, Testing, and Verification IX**, E. Brinksma, G. Scollo and C. Vissers (eds.), North-Holland, 1989.

[7] M.-S. Chen, Y. Choi and A. Kershenbaum, "*Minimal Length Test Sequences for Protocol Conformance*," **Proc. of the 1st Network Management and Control Workshop**, Polytechnic University, New York, Sept. 1989.

[8] A.T. Dahbura and K.K Sabnani, "*An Experience in Estimating Fault Coverage of a Protocol Test*," **Proc. of IEEE INFOCOM'88**, 1988, pp. 71-79.

[9] G.J. Holzmann, "*A Theory for Verification*," **IEEE Trans. on Computers**, Vol. C-31, No. 8, Aug. 1982, pp.730-738.

[10] C.H. West, "*An Automated Technique of Communications Protocol Validation*," **IEEE Trans. on Communications,** Vol. COM-26, No. 8, Aug. 1978, pp. 1271-1275.

[11] K.K. Sabnani, A.M. Lapone and M.U. Uyar, "*An Algorithmic Procedure for Checking Safety Properties of Communication Protocols*," **IEEE Trans. on Communications**, Vol. COM-37, No. 9, Sept. 1989, pp. 940-948.

[12] R.E. Miller and G.M. Lundy, "*An Approach to Modeling Communication Protocols Using Finite State Machines and Shared Variables*," **Proc. IEEE Global Communications Conf.**, Dec. 1986, pp. 3.8.1-3.8.5.

[13] M.G. Gouda, "*Closed Covers: To Verify Program of Communicating Finite State Machines*," **IEEE Trans. on Software Engineering,** Vol. SE-10, No. 6, Nov. 1984, pp. 846-855.

[14] S.T. Voung, D.D. Hui and D.D. Cowan, "*VALIRA: A Tool for Protocol Validation via Reachability Analysis*," **Proc. Protocol Specification, Testing, and Verification VI**, B. Sarikaya and G.v. Bochmann (eds.), North-Holland, 1986, pp. 35-42.

[15] R. Milner, "*A Calculus of Communicating Systems*," **Lecture Notes in Computer Science**, Vol. 92, New York: Springer-Verlag, 1980.

[16] C.A.R. Hoare, "*A Communicating Sequential Processes*," Prentice-Hall, Englewood, New Jersey, 1985, pp. 111-117. Computer Science, Vol. 92, New York: Springer-Verlag, 1980.

[17] J.P. Favreau and R.J. Linn, "*Automatic Generation of Test Scenario Skeletons from Protocol Specifications Written in Estelle*," **Proc. Protocol Specification, Testing, and Verification VI**, B. Sarikaya and G.v. Bochmann (eds.), North-Holland, 1986, pp. 191-202.