A FAMILY OF PROTOCOL TESTING TECHNIQUES

by

WENDY YUEN-LING CHAN

B.A.Sc., The University of British Columbia, 1987

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF ELECTRICAL ENGINEERING)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1989

Department of ELECTRICAL ENGINEERING

The University of British Columbia
Vancouver, Canada

Date 1989 AUGUST 10

ABSTRACT

This thesis developed three testing techniques that are applicable to the conformance testing of protocols: the UIOv-method, the eUIOv-method and the hybrid technique.

The UIOv-method is for testing simple protocols modelled by finite state machines (FSMs). Any protocol that passes its tests possesses an FSM skeleton that is identical to the specified FSM. The UIOv-method is extended to the eUIOv-method to test more complex protocols that can be modelled by extended FSMs (EFSMs). A data flow testing procedure (DFTP) based on static data flow analysis and FSM testing is developed in this thesis to test the flow of parameters and variables in a protocol. This procedure is augmented with the eUIOv-method to form the hybrid technique which is directly applicable to the testing of complex protocols implemented according to their Estelle specifications. The technique captures protocols with erroneous EFSM control structures that cannot be detected by existing testing methods developed by Sarikaya and Ural.

KEYWORDS: conformance testing, protocols, Estelle, finite state machines, unique input/outputs, extended finite state machines, data flow, fault coverage, protocol specification.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENT

I would like to thank Dr. Son Vuong for his guidance and for his constant encouragements that I publish my work. I would like to thank Dr. Mabo Ito for his careful reading of this thesis and for his help in improving my writing tremendously. I would like to thank Dr. Sam Chanson and Jianping Wu for many helpful discussions and for their moral support during difficult times. Also, a very special thank you to Dr. Gunther Schrack, without him, I would not have been able to become a graduate student.

I would like to thank the Natural Sciences and Engineering Council of Canada and Idacom Electronics Limited for their support in the form of an University and Industry Cooperative Research Grant.

Lastly, many heartfelt thanks to my parents for their immeasurable love and understanding; and especially to my brother, Ramsey, and my fiance, Dawson, for their continuous moral support and their immense faith in me; without them, this work would not have been possible.

During conformance testing, the protocol implementation under test (IUT) is typically considered as a "black box" since its source code is generally not accessible. Testing is carried out using test sequences. A test sequence is a sequence of input/output (I/O) pairs derived from the protocol specification. Test inputs are applied to the IUT via its input port. The outputs generated by the IUT are received via its output port and compared with the corresponding outputs in the test sequence. If they match, the IUT is said to conform to the specification.

The ability of a test to detect non-conforming or erroneous IUTs during testing depends solely on the test sequence used. Recently, test sequence generation has received much attention from the research community; in particular, sequences generated from the formal specification of protocols. The formal specification language of interest in this thesis is Estelle. In Estelle, the control structure of the protocol is modelled as an extended finite state machine (EFSM) while the data flow aspect of the protocol is described by a set of Pascal statements.

1.1 PAST TEST GENERATION EFFORT

Recent work on test sequence generation based on Estelle protocol specifications include those of Sarikaya [Sari87] and Ural [Ural88], both of whom emphasize the data flow aspect of protocols and both of whom work from normal form Estelle specifications [Sari86]. Other notable test sequence generation techniques that are applicable to Estelle protocol specifications include those developed for testing finite state machines (FSMs) [Kou87, Sari82, Sidh89]. These include the T-method [Nait81]; the W-method [Chow78, Sidh89], the D-method [Gone70], and the U-method [Sabn88, Aho88]. Their applications to protocol conformance testing were extensively studied in [Sidh89]. These methods examine only the control structure of protocols; they ignore interaction primitive parameters and may face executability problems when applied to Estelle specifications. However, these methods are directly applicable to the testing of simple protocols that can be modelled by FSMs.

1.2 THESIS GOAL

The goal of this thesis is to develop testing techniques for the conformance testing of protocols such that maximum fault coverage can be achieved. Fault coverage is defined in this thesis according to the specifications of

the protocols used. The following two issues are not of primary concern in here: test sequence optimization and test architecture.

Problems in the generation of conformance test sequences generally fall into two categories: coverage and optimization. While coverage deals with how to generate test sequences to achieve a particular fault coverage, optimization deals with how to optimize a given test sequence. Although optimization is necessary to reduce the cost of a test, it is secondary to coverage. Only when a desirable fault coverage is achieved should optimization follow. This thesis thus focuses on the more important coverage issue in test sequence generation. Some sample test sequences generated in this thesis are not optimized so that their intents may be easily followed.

In testing the IUT of a protocol that belongs to the Open System Interconnection (OSI) Reference Model [Zimm80], an upper tester and a lower tester are assumed to be available in this thesis to provide control and observation at the upper and lower interfaces of the IUT.

1.3 THESIS CONTRIBUTIONS

This thesis contributes to solving the problem of generating conformance test sequences for protocols from their specifications in the following ways.

This thesis found the original U-method to be inadequate and modified it so that it is now capable of detecting all I/O errors as well as state errors [Chan89b]. Its fault detection capability is now equivalent to those of the W- and D-method. The D-method and W-method are actually special cases of the modified U-method (UIOv-method) developed in this thesis where the UIOv-method enjoys the applicability advantage of the W-method and the length advantage of the D-method while providing full fault coverage.

This thesis introduced the concept of unique test sequences (UTSs). These are test sequences unique to the specified FSM from which they are generated. Any FSM that passes a test using an UTS possesses a FSM skeleton identical to the specified FSM.

This thesis extended the UIOv-method to produce the eUIOv-method for testing protocols that can be modelled by EFSMs.

This thesis developed a data flow testing procedure based on static data flow analysis where each definition and

usage of a variable are exercised during testing and, whenever possible, they are verified as well.

This thesis created a hybrid test sequence generation technique by augmenting the eUIOv-method with the data flow testing procedure. This hybrid technique is applicable to testing the EFSM control structure of a protocol as well as its data flow; hence, it is directly applicable to IUTs that are implemented according to their Estelle specifications. During testing, the IUT is checked for each transition in its Estelle specification. Each statement in the transition is exercised as well as verified whenever possible.

This thesis refined and reformatted the normal form of Estelle so that its transitions are in canonical form and its format explicitly brings out the underlying EFSM and data flow in an Estelle specification.

The hybrid technique was tailored for protocols with underlying EFSM control structures, this technique is thus capable of detecting faulty IUTs with erroneous EFSMs that would otherwise be missed by the techniques developed by Ural and Sarikaya. Since the hybrid technique also includes data flow testing, it is capable of detecting certain types of data flow errors in an IUT that would otherwise be missed by FSM testing techniques.

Test sequences generated by the hybrid technique are also applicable to IUTs not implemented according to their Estelle specifications. The fault coverage achieved would be limited to faults in the flow of data; sequencing faults may or may not be captured.

1.4 THESIS ORGANIZATION

The remainder of this thesis is organized as follows. Chapter 2 discusses FSM test sequence generation techniques and presents the UIOv-method as well as the UTS concept. In chapter 3, a brief introduction to EFSMs is followed by a discussion of extending the UIOv-method to the eUIOv-method for the testing of protocols modelled by EFSMs. Chapter 4 provides a brief review of static data flow analysis and shows how the data flow testing procedure is developed from it. In chapter 5, the hybrid technique is discussed and applied to Estelle using the Class 0 Transport Protocol as an example. An evaluation and a comparison of the hybrid technique with existing test sequence generation techniques applicable to Estelle protocol specifications are presented in Chapter 6. Chapter 7 concludes this thesis with a summary of its contributions and a discussion of possible future work in the area of conformance testing.

2 THE UIOV-METHOD

Test sequence generation techniques developed for finite state machines (FSMs) may be used to generate conformance test sequences for those protocols that can be modelled as FSMs. This chapter looks at a recent study done by Sidhu [Sidh89] on this approach to conformance test generation. Four notable test generation methods were compared in that paper: the T-method, the D-method, the W-method and the U-method. In general, the D-method is hampered by its limited applicability while the W-method is undesirable because of the lengthy test sequences it generates. The T-method, on the other hand, has neither one of the above problems; however, it generally cannot achieve full fault coverage. The most recent method, the U-method, is more widely applicable than the D-method and it generates shorter test sequences than those generated by the W-method. Fault coverage produced by the U-method was found to be full in [Sidh89]. The U-method thus seemed to be the best of the four methods.

This chapter shows that the U-method does not always achieve full fault coverage; it points out a shortcoming with the U-method [Chan89b] that sometimes hampers its fault detection capability. The U-method is then revised with the addition of a verification procedure in here to form the

UIOv-method [Chan89b] which corrects the problem and produces full fault coverage each time. The resulting UIOv-method is still more widely applicable than the D-method and it generally produces shorter test sequences than the W-method does.

All test sequences generated by the D-method, W-method and UIOv-method possess a property that is responsible for their full fault coverages. This property distinguishes these sequences to be unique test sequences (UTSs) [Chan89b], and it guarantees the detection of any erroneous IUT provided that the number of states in the IUT does not exceed that which is in the FSM specification. This concept of UTSs is also reviewed in this chapter.

2.1 THE FINITE STATE MACHINE MODEL

A finite state machine (FSM) can be represented as F={S,I,O,T,A} where S denotes the set of states, I denotes the set of inputs, O denotes the set of outputs, T denotes the state transition function which produces a new state based on the current state and the current input, and A denotes the action function which produces an output based again on the current state and the current input.

An FSM can also be represented by a collection of transitions each of which performs an output operation and a

state transition provided that an input event is correctly received at the appropriate starting state. This representation can be presented in a tabular or graphical format. In the tabular format, each column is labelled by a state to denote the starting state of a transition. Each row is labelled by an input operation. Each transition is denoted by a table entry identified by its output operation and the next state. The graphical format is denoted by G=(N,E), where N is a set of nodes representing the states of the machine and E is a set of arcs representing the set of transitions. Each arc starts at the starting state of the transition and ends at its final state. Each arc is identified by the input event that triggered the transition and the output event produced.

The FSM models considered in this thesis are assumed to be minimal and strongly connected Mealy machines that may or may not be completely specified. A Mealy machine, M, is an FSM whose set of outputs is dependent on the set of states as well as the set of inputs. The Mealy machine is minimal, or reduced, implies it does not have equivalent states; that is, it has the smallest number of states possible. Machine M is completely specified if an output is specified for each input in I for each of the states in S. A strongly

connected machine possesses an input sequence which traverses between any two states in the machine.

2.2 FSM TESTING TECHNIQUES

There are four notable test sequence generation techniques for FSMs: the T-method, the W-method, the D-method and the most recent method, the U-method.

2.2.1 The T-Method

The T-method [Nait81] is the simplest of the four methods. This method is applicable to any strongly connected FSM. It generates a test sequence, called a transition tour, by applying random inputs to an FSM until every transition is traversed at least once.

There are two disadvantages to using the T-method. One is the test sequences generated may contain redundant inputs. The other being the test only checks for the existence of transitions; it does not verify that the states in each transition are correct. As a result, the T-method may not detect state errors; it detects only output errors.

2.2.2 The W-Method

The original PW-method [Chow78] was slightly modified in [Sidh89] to form the W-method, which is a checking

experiment that uses the W-set for state identification. A checking experiment is a test procedure which verifies that each input specified for a state produces the expected output and takes the FSM to the correct next state. In the first part of a checking experiment, each state in the FSM is identified using a chosen characterizing I/O sequence. In the second part of the experiment, each I/O operation and the final state in each transition in the FSM are verified.

The W-set is also known as the characterization set. It is a set of input sequences which is composed of, for every pair of states in the FSM, at least one input sequence that can distinguish them. Every minimal, strongly connected and completely specified FSM possesses a W-set. Of the four methods, the W-method produces the longest test sequences. It is capable of detecting all faults in an FSM with a number of states not exceeding that which is in the specified FSM.

2.2.3 The D-Method

The D-method [Gone70] is a checking experiment which uses a set of distinguishing sequences (DSs) for state identification.

An DS can be thought of as a special case of a W-set where there exists only one sequence of inputs. This

sequence produces a different sequence of outputs for every different starting state. Not every completely specified, minimal and strongly connected FSM possesses an DS. This method, when applicable, is also capable of detecting all faults in an FSM with a number of states smaller than or equal to that in the specified FSM.

2.2.4 The U-Method

The U-method [Sabn88] is also a checking experiment with the exception that it uses a sequence of unique input/outputs (UIOs) to identify the states in each transition.

An UIO sequence (UIOS) for a state is an I/O behavior not exhibited by any other state in the FSM. Not every state in an FSM possesses an UIOS. In the absence of an UIOS, a state signature is used which is formed by concatenating a set of input sequences, each of which distinguishes the state from one other state in the FSM.

An FSM does not have to be completely specified in order to possess a set of UIOSs. In [Sidh89], it was found that the U-method possesses a fault detection capability equivalent to those produced by the W- and D-method for minimal, completely specified and strongly connected FSMs. The U-method generates test sequences whose lengths are

comparable to those generated by the D-method. Their lengths are generally shorter than those produced by the W-method.

Both the U-method and the W-method are generally applicable to completely specified, strongly connected and minimal FSMs, but being completely specified is a necessary condition for the W-method while it is sufficient, but not necessary, for the U-method [Sidh89].

2.2.5 Comments On The Four Methods

2.2.5.1 Applicability

In deciding on which test method to use, the first factor to be considered is applicability, then comes the achievable fault coverage, and finally, the length of the test sequence produced. If all T-, W-, D- and U-methods were applicable to test a particular FSM, then any one of the latter three methods is preferred over the T-method because of their better fault coverages. Among the three methods, the W-method is the most undesirable because of its lengthy test sequences. Since UIOSs generally occur more frequently than DSs do, the U-method seems to stand out from among the three.

In fact, DSs are special cases of UIOSs where the input sequences for all the UIOSs are identical. In the search algorithm presented in [Sabn88] for finding minimum UIOSs, if the resulting UIOSs have identical input sequences, then they would also be referred to as DSs.

The W-set is to DSs as the signatures are to UIOSs. The W-set possesses the same input constraint as that in the DSs. While signatures are used when UIOSs are absent, the W-set should be used only when the DSs are absent because of the lengthy test sequences that can result from using the W-method. From this viewpoint, the UIOSs or the signatures, having the least restrictive constraints, occur more frequently than the DSs or the W-set does in FSMs.

2.2.5.2 Fault Coverage

The four FSM test methods are directly applicable to the conformance testing of those protocols that can be modelled as FSMs. The ability of a test sequence to decide whether a protocol implementation under test (IUT) conforms to its specification solely relies upon the range of faults that it can detect.

The four methods work by checking for the existence of transitions and, with the exception of the T-method, by verifying that the states in each transition are correct.

In this sense, missing and erroneous states and I/Os in an IUT can be detected, but extra states and I/Os cannot. However, conformance merely requires that an IUT behaves as according to its specification. This implies as long as an IUT possesses a skeleton FSM that is identical to its specification FSM, it would be a conforming IUT independent of whether other states or transitions exist. As a result, the W-, D- and U-method suffice as conformance test methods for protocols. Their only limitation is that the IUT must not possess a number of states that exceeds that which is in the specified FSM since extra states no longer guarantee "what you see is what you get" during testing.

FSMs that model protocols may not be completely specified. Those edges that are specified are referred to as core edges [Sabn88]. For the unspecified state-input pairs, a Completeness Assumption can be used where the protocol machine is either assumed to produce a null output and remain in its current state or it is assumed to enter an error state following the generation of an error message. The Completeness Assumption allows an incompletely specified FSM to become completely specified. Conformance can thus be defined at two levels: strong and weak. An IUT has strong conformance to its specification if, for all test inputs, it generates the same outputs as those specified in its

specification. An IUT has weak conformance to its specification if the IUT has the same I/O behavior as its incompletely specified FSM specification. For the non-core edges, the IUT has unspecified behavior. This implies strong conformance testing can be carried out only for those IUTs whose specification FSMs are completely specified; otherwise, only weak conformance testing is possible.

The following summarizes the fault coverages that were reported in [Sidh89] of the four test methods. The fault coverage of the weak conformance test sequence for the U-method is better than that of the T-method since the T-method only checks the I/O operations at each transition while the U-method verifies the states of each transition as well. The fault coverages of the strong conformance test sequences for the W-, D- and U-method are identical and are better than that of the T-method. The latter is again based on the fact that the T-method does not verify the states during testing while the other three methods do. The former is based on all three methods are checking experiments except different characterizing I/O sequences are used for state identification. Sidhu claims that each of the three methods can detect all I/O errors as well as state transition errors. This, however, is not the case as the following section will show.

In general, a test sequence generated by any one of the four methods is not unique. The reason being an FSM has more than one transition tour, and it may have more than one set of UIOSs or DSs, or it may have more than one W-set. The choice of which characterizing sequence to use depends on the tester.

2.3 THE SHORTCOMING OF THE U-METHOD

The U-method appeared to be the ultimate test sequence generation method for FSMs where the applicability advantage of the W-method is combined with the length advantage of the D-method, and, at the same time, it possesses full fault detection capability. Unfortunately, this is not the case. It was found in [Chan89b] that the fault detection capability of the U-method in fact depends on the set of UIOSs chosen. This section is extracted from [Chan89b]. It compares the U-method with the D-method and it shows why the U-method does not have as strong fault coverage as the D-method does.

2.3.1 Assumptions

In the following discussion, each state in an IUT is assumed to have a reset input, r, which takes the IUT back to the initial idle state; no output is generated by the IUT

in response to r. An arc to denote this feature would be labelled as r/-; it would begin at a starting state and end at the initial idle state. Each r/- arc should be treated as a transition and exercised as well as verified during testing, but for simplicity, each arc will be assumed to be correct in the following discussion.

Sample test sequences generated in this section are not optimized to better illustrate the role of each subsequence. These sequences can be optimized by eliminating those test subsequences that are completely contained in other test subsequences.

2.3.2 The UIOS Problem

Testing of the simple FSM extracted from [Chan89b] and shown in Figure 2.1 shows the fault coverage for the U-method is not always identical to that of the D-method.

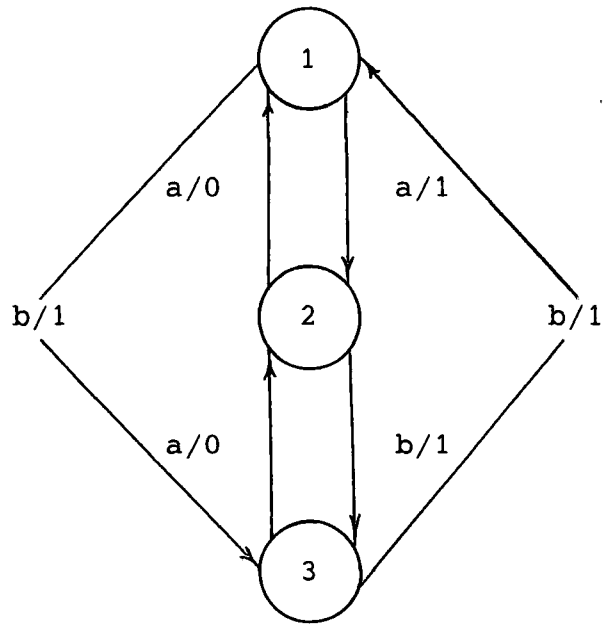

Figure 2.1: An FSM specification.

The test sequence generated by the U-method, shown in Table 2.1, cannot detect the faulty implementation, shown in Figure 2.2, of the FSM in Figure 2.1. The UIOSs chosen for states 1, 2 and 3 in that test sequence are a/1, a/0.a/1 and b/1.a/1 respectively. The resulting test sequence could not detect the erroneous tail state for the b/1 edge.

```
r/- a/1
r/- a/1.a/0.a/1
r/- a/1.b/1.b/1.a/1

r/- a/1.a/0.a/1
r/- b/1.b/1.a/1
r/- a/1.a/0.a/1
r/- a/1.b/1.b/1.a/1
r/- a/1.b/1.a/0.a/0.a/1
r/- a/1.b/1.b/1.a/1
```

Table 2.1: U-method test sequence for the FSM in Figure 2.1.

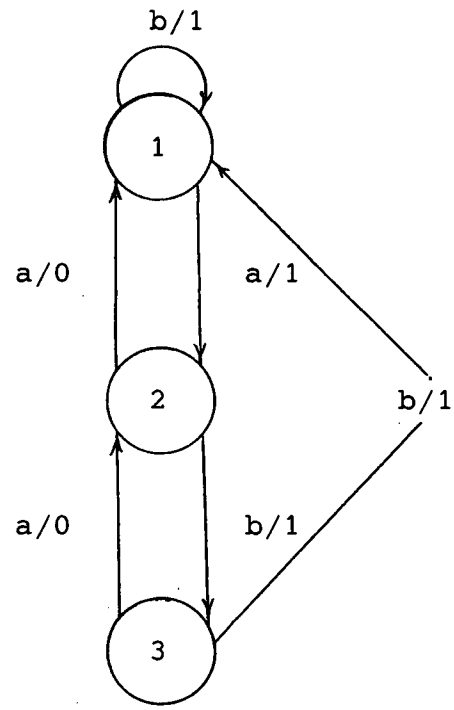

Figure 2.2: A faulty IUT of the FSM in Figure 2.1.

However, the test sequence generated by the D-method shown in Table 2.2 is capable of detecting this faulty IUT. The DSs chosen for states 1, 2 and 3 are respectively a/1.a/0, a/0.a/1 and a/0.a/0.

```
r/- a/1.a/0
r/- a/1.a/0.a/1
r/- a/1.b/1.a/0.a/0

r/- a/1.a/0.a/1
r/- b/1.a/0.a/0
r/- a/1.a/0.a/1.a/0
r/- a/1.b/1.a/0.a/0
r/- a/1.b/1.a/0.a/0.a/1
r/- a/1.b/1.b/1.a/1.a/0
```

Table 2.2: D-method test sequence for Figure 2.1.

The reason why the U-method test sequence could not detect the erroneous tail state for the b/1 edge is because the chosen set of UIOSs is unique in the specification FSM, but it is not unique in the faulty IUT in Figure 2.2. Both states 1 and 3 in the faulty IUT could generate the UIOS, b/1.a/1, chosen for state 3 in the test. This non-uniqueness is not detected during the testing of the IUT. As a result, the IUT could be in either state 1 or state 3 when b/1.a/1 is observed during testing. If, however, another set of UIOSs were chosen, for instance, a/1 for state 1, a/0.a/1 for state 2 and a/0.a/0 for state 3, then the faulty IUT would be detected as these UIOSs are also unique I/O behaviors in the faulty IUT.

The problem with the U-method is thus UIOSs are used based on the assumption that they are also UIOSs in the IUT when in fact they may not be. UIOSs are chosen based on the FSM specification. They are capable of identifying the states in the specification, but they cannot identify states in the IUT unless they are also UIOSs in the IUT. In a faulty IUT, UIOSs may not be unique. Hence, the U-method incorrectly assumes that if UIOSs are unique in the specification, then they must also be unique in the IUT. As a result, an erroneous state could escape detection if it is capable of producing exactly the same UIOS that belongs to another state.

2.3.3 The State Signature Problem

The uniqueness problem may also exist for state signatures. It may even be inherent since signatures are not required to be unique [Chan89b]. Figure 2.3 is a duplicate of Figure 4 in [Chan89b]. It shows an FSM whose state C has no UIOSs. A signature, 0/0.0/1.1/0, is generated for it. Note that the signature is not unique to state C; state B is also capable of generating this I/O sequence.

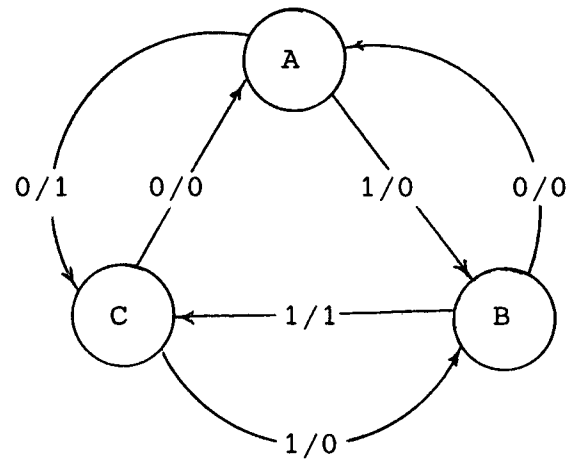

Figure 2.3: An FSM with no UIOS for state C.

A test sequence for this FSM generated by the U-method is shown in Table 2.3. The UIOSs chosen for states A and B are 0/1 and 1/1 respectively.

```
r/- 0/1
r/- 1/0.1/1
r/- 0/1.0/0.0/1.1/0

r/- 1/0.1/1
r/- 0/1.0/0.0/1.1/0
r/- 1/0.0/0.0/1
r/- 1/0.1/1.0/0.0/1.1/0
r/- 0/1.0/0.0/1
r/- 0/1.1/0.1/1
```

Table 2.3: U-method test sequence for FSM in Figure 2.3.

This test sequence cannot detect the faulty IUT shown in Figure 2.4. The edge 1/1 from state B incorrectly ended at state B in the faulty IUT, but this is not detected by the test sequence in Table 2.3. The reason is because the

state signature used to verify state C is not unique. As a result, when the state signature is observed, the state that generated it could have been either C or B. This problem, unlike the UIOSs, could be inherent in state signatures; that is, this non-uniqueness could also exist in the specification FSM. As a result, the erroneous final state also escapes detection during testing.

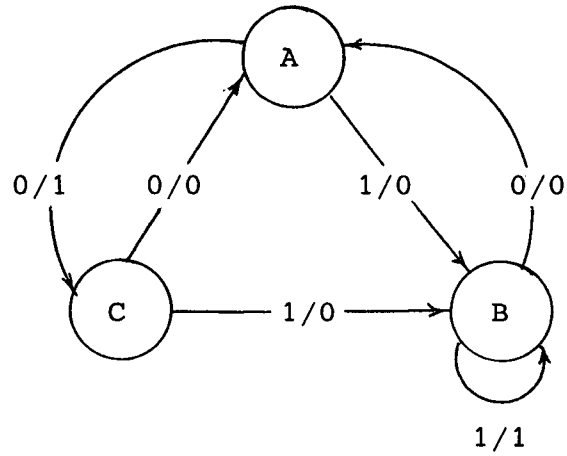

Figure 2.4: A faulty implementation of the FSM in Figure 2.3.

2.4 THE UIOV-METHOD

This section revises the U-method with the addition of a verification procedure to eliminate its uniqueness problems.

2.4.1 Uniqueness Problem Analysis

The uniqueness problem with the U-method concerning its UIOSs is not evident in the D-method or the W-method. The reason is because the input sequences for the DSs and the W-set are identical for all the states in a given FSM. During the first state identification procedure in a checking experiment generated by the D- or W-method, a state that responds with the correct output sequence implies two things: the state possesses the correct DS or W-set as according to the specification; as well, the state does not generate an DS or W-set that belongs to another state. This eliminates the possibility that more than one state in the IUT may produce the same characterizing I/O sequence.

In a checking experiment generated by the U-method, the first state identification procedure may have different input sequences for different states. As a result, a state which responds with the expected output sequence implies it possesses the expected UIOS as according to its specification, but it certainly does not imply that it cannot produce an UIOS that belongs to another state whose input sequence differs from that of its own UIOS.

2.4.2 ~Uv

The way to eliminate the uniqueness problem concerning the UIOSs is thus to ensure that the chosen set of UIOSs for a specified FSM is also a set of UIOSs in the IUT; that is, the uniqueness of the UIOSs also holds in the IUT. This can be achieved by verifying the uniqueness of the UIOSs prior to their uses during testing [Chan89b]. This verification procedure will be referred to as ~Uv while the usual state identification procedure in the U-method will be referred to as Uv from now on. The U-method with the addition of ~Uv will henceforth be referred to as the UIOv-method.

~Uv is required only for those states whose UIOSs have different input sequences; for the other states, ~Uv is implied in Uv [Chan89b]. During ~Uv, a partial input sequence may be sufficient. For instance, given two states A and B in an FSM, if the UIOS for A is a/1.b/1.a/0 and the response of B to the input sequence a.b.a is 1.0.0, then B can be tested for the absence of a/1.b/1.a/0 by merely observing that B generates the subsequence a/1.b/0 since this is enough to show that B cannot generate a/1.b/1.a/0. This partiality reduces the cost of the final test sequence.

In ~Uv, the emphasis is on states not generating UIOSs that belong to other states, hence, identical outputs generated by different states in response to the same inputs

are acceptable; whereas in Uv, this would not be allowed as it implies one UIOS is generated by more than one state. An example test sequence using the UIOv-method is produced in Table 2.4 for the FSM in Figure 2.1. This test sequence is equivalent to that shown in Table 2.1 with the exception that ~Uv is added. Note that at subsequence r/- b/1.a/0 the faulty IUT in Figure 2.2 fails since state 1 displays r/- b/1.a/1 instead; that is, state 1 generates the UIOS that belongs to state 3.

```
r/- a/1
r/- a/1.a/0.a/1
r/- a/1.b/1.b/1.a/1

r/- a/1.b/1.a/0
r/- a/1.b/1.a/0.a/0
r/- b/1.a/0
r/- a/1.b/1.a/0

r/- a/1.a/0.a/1
r/- b/1.b/1.a/1
r/- a/1.a/0.a/1
r/- a/1.b/1.b/1.a/1
r/- a/1.b/1.a/0.a/0.a/1
r/- a/1.b/1.b/1.a/1
```

Table 2.4: UIOv-method test sequence for the FSM in Figure 2.1.

2.4.3 IO(S,K)s

To correct the inherent uniqueness problem in state signatures, [Chan89b] proposed the use of a set of IO(S,K)s in place of a state signature. Each member in the set of IO(S,K)s is a sequence of I/Os that distinguishes the state,

S, to which the IO(S,K)s belongs, from at least one other state, K, in the FSM. This is somewhat similar to the W-set with the exception that the input sequences in the IO(S,K)s for different S states may be different and the number of sequences in each set of IO(S,K)s may also vary depending on state S. The size of the set of IO(S,K)s for a state S is the minimum required to distinguish S from all other states in the FSM.

Each set of IO(S,K)s must be unique to state S. Each set is treated as though it is an UIOS during testing; that is, each is verified in the same way UIOSs are verified in the ~Uv and Uv procedures prior to their uses during testing. During the transition testing, Tt, portion of the checking experiment, each I/O operation with tail state S will be tested a number of times equals to the number of I/O sequences in its set of IO(S,K)s. An example is shown in Table 2.5 for the FSM in Figure 2.3.

```
r/- 0/1
r/- 1/0.1/1
r/- 0/1.0/0
r/- 0/1.1/0

r/- 1/0
r/- 1/0
r/- 0/1
r/- 0/1.0/0
r/- 0/1.1/0
r/- 1/0.1/1
r/- 1/0.0/0
r/- 1/0.0/0

r/- 1/0.1/1
r/- 1/0.0/0.0/1
r/- 0/1.0/0.0/1
r/- 0/1.1/0.1/1
r/- 0/1.0/0
r/- 0/1.1/0
r/- 1/0.1/1.0/0
r/- 1/0.1/1.1/0
```

Table 2.5: UIOv-method test sequence for the FSM in Figure 2.3.

For state C, IO(C,A) is 0/0 and IO(C,B) is 1/0. These two sequences are checked for their absences in states A and B during $\tilde{U}v$. In state A, although 1/0 is present, 0/0 is absent. In state B, 0/0 is present but 1/0 is absent. Hence, IO(C,A) and IO(C,B) together can be used to uniquely identify state C. Each edge that ends at C, 1/0 from A and 1/1 from B, is checked twice using 1/0 first then 0/0 the second time. Note that the faulty IUT in Figure 2.3 possesses the same set of UIOSs and IO(C,K)s as that in the specified FSM. As a result, the erroneous final state for edge 1/1 is detected by IO(C,B).

While the DSs are special cases of the UIOSs, the W-sets can be viewed as special cases of the IO(S,K)s where the former has the additional constraints that the input sequences must be identical for all the states and the number of sequences in each W-set for each state must be identical. As well, the W-set must exist for every state in the FSM.

In the remainder of this thesis, when UIOSs or IO(S,K)s are referred to, it is assumed that these are verifiable UIOSs and verifiable IO(S,K)s.

2.4.4 Comparing The UIOv-Method With The Others

The fault coverage produced by the UIOv-method is better than that produced by the U-method. Erroneous tail states such as those illustrated in the previous sections are now detectable. The fault coverage of the UIOv-method is now identical to those of the W- and D-method. The proof is as follows.

First of all, the W-, D- and UIOv-method are all checking experiments with the exception that they use different characterizing I/O sequences for state verification. Secondly, the DSs and the W-set are special cases of the UIOSs and the IO(S,K)s respectively. Recall how the ~Uv procedure for a particular state is implied in

its Uv procedure when the input sequences for the UIOSs belonging to other states are identical to that of its own UIOS. In this sense, the state identification procedures of the checking experiments generated by the W- and D-method accomplish what both Uv and ~Uv accomplish in the UIOv-method. As a result, the fault coverages of the W-, D- and UIOv-method are identical.

The length of the test sequences produced by the UIOv-method are obviously longer than those produced by the U-method; however, they are generally shorter than the sequences derived by the W-method since the latter requires that each transition be tested a number of times equals to the fixed number of I/O sequences in the W-set. In comparison to the D-method, it is more difficult as the sequence lengths may differ according to different FSMs. The test subsequence corresponding to Uv and ~Uv is likely longer than that corresponding to the state identification procedure for the D-method. However, the subsequence for the Tt portion of the test is likely shorter in the UIOv-method because of less restrictive conditions in forming the UIOSs, which produce UIOSs that are generally shorter than DSs. If the UIOv- and D-method were both applicable to an FSM and the DSs are shorter than the UIOSs, then the D-

method should be used; otherwise, both methods need to be examined.

In terms of applicability, the UIOv-method is more widely applicable than both D- and W-methods. This is obvious due to the added constraints in the formation of the DSs and W-set. The likelihood that a set of DSs exists in a given FSM is generally lower than that for a set of UIOSs or W-set. In general, a minimal and strongly connected Mealy machine being completely specified is a sufficient but not necessary condition for the application of the D-, W- and UIOv-method. The necessary condition is that the FSM possesses an DSs set, a W-set or an UIOSs set. The same condition applies to incompletely specified machines: so long as such a machine possesses a completely specified FSM skeleton from which a set of DSs or a W-set can be produced, or from which stems a set of UIOSs, the respective D-, W- and UIOv-method would be applicable. Hence, all three methods are applicable to any FSM independent of whether it is completely or incompletely specified as long as the required characterizing I/O sequences exist in the FSM.

2.5 UNIQUE TEST SEQUENCES

All test sequences generated by the D-, W- and UIOv-methods achieve full fault coverage in the testing of FSMs.

A property that is common among these test sequences is captured in the concept of unique test sequences (UTSs) proposed in [Chan89b]. Full fault coverage in FSM testing in this thesis implies the detection of all erroneous and missing states and I/Os. An UTS is defined to be a test sequence that is unique to a specified FSM, M, if there does not exist any other FSM with the same number of states and transitions as that in M capable of producing an I/O sequence that is identical to the UTS of M. An IUT thus passes a test that uses an UTS if and only if the IUT possesses a FSM skeleton identical to the specified FSM. This implies UTSs are capable of detecting any faulty IUT provided that the number of states in the IUT is no greater than that which is in the FSM specification. This also implies an IUT that passes such a test may possess extra transitions in addition to those specified in the FSM specification, provided that the IUT has a number of states greater than or equal to that which is in the FSM specification.

An UTS can be formed by using a sequence of I/Os to accurately describe each transition in the FSM specification; that is, each I/O operation is described as is, their starting and final states are described by their characterizing I/O sequences. This implies that if there

exists a program which generates FSM graphs according to a given I/O sequence and a given number of states, then if the given I/O sequence were an UTS, then there would be one and only one graph that can be generated from the given sequence. An FSM may have more than one UTS depending on which characterizing I/O sequence is selected.

It was found in [Chan89b] that test sequences produced by the D-method and the UIOv-method were sequences of I/Os that accurately described each transition specified in the given FSM; that is, both methods produced UTSs. Since the W-method is a special case of the D-method and the UIOv-method, it is also capable of generating UTSs. Since the T-method describes only the I/O operations inside each transition without any reference to its states, this method generally does not produce UTSs in FSMs with the exception of the following type of simple FSMs.

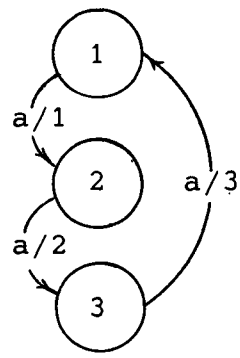

Figure 2.5: A simple FSM with an inherent UTS.

In the simple FSM, assuming that the test sequence must begin at the initial state, state 1, the shortest transition tour possible is a/1.a/2.a/3. This is not an UTS because there exists two other completely specified FSMs, each having three states and an input set I = {a}, that is capable of generating an I/O sequence identical to a/1.a/2.a/3. These two FSMs differ from that in Figure 2.5 by the tail state of the a/3 arc. One FSM has its a/3 arc end at state 2; the other has its a/3 arc end at state 3. However, the transition tour a/1.a/2.a/3.a/1 is an UTS since it bounds the tail state of a/3 to state 1, the only state that generates a/1.

UTSs apply to both completely and incompletely specified FSM specifications since the D-, W- and UIOv-method are applicable to any FSM that possesses a set of DSs, a W-set or a set of UIOSs respectively. This implies any IUT that possesses an FSM skeleton identical to the specification FSM, whether it be completely or incompletely specified, will pass a test using an UTS based on the specification.

One question that is often brought up is whether a generated test sequence is correct; that is, whether the test correctly represents the specification so that any IUT that passes that test must conform to the specification.

One way to verify that a given test is correct is to generate all possible FSMs that correspond to the given test. If the test uses an UTS, then there would exist only one FSM, with the same number of states and transitions as that in the specification, that can be generated.

2.6 FSM TESTING

Much can be learnt from FSM testing. In the T-method, transitions are only exercised during testing. This method generally achieves a partial fault coverage that is comparable to that produced by the branch coverage criterion in software testing, where each branch is exercised at least once. From the T-method, it becomes clear that merely exercising statements, or branches, do not constitute a sufficient test; that is, some errors may escape detection. These errors are those found in elements that are not directly observable in the I/O operations; for instance, the states in an FSM. The checking experiments for FSMs solve this problem by verifying each state with its characterizing I/O sequence. This is possible because the variable STATE takes on a finite set of possible values. As well, each different value can be characterized by a different I/O sequence. This, however, may not be possible in software testing where a set of possible values for a given variable

may be infinite. As well, there may not be a sequence of externally observable events that can differentiate among different values of a variable. Nevertheless, FSM testing does show that exercise along with verification gives improved fault coverage over that produced by exercise alone.

2.7 CHAPTER SUMMARY

In summary, checking experiments for testing FSMs are capable of detecting all states and I/O errors as according to an FSM specification. Before the advent of the UIOv-method, checking experiments based on DSs were hampered by their limited applicabilities while those based on W-sets were handicapped by their lengthy test sequences. With the advent of the UIOv-method, checking experiments became more applicable without the disadvantage of a lengthy test sequence.

The UIOv-method solves the uniqueness problem in the UIO-method with the use of a verification procedure. This solution is independent of which UIOS is non-unique in the IUT [Chan89c]; it solves the problem by attacking its cause. This solution permits any minimum length UIOS to be used provided that it is verifiable; hence, it does not

considerably add to the complexity of the search algorithm for UIOSs given in [Sabn88].

Checking experiments generate test sequences that are unique to the specified FSM. As a result, any IUT that passes such a test possesses an FSM skeleton that is identical to the specified FSM; however, it may have additional transitions that do not belong to the specification.

In terms of applicability, checking experiments are generally applicable to Mealy machines that are minimal and strongly connected. Whether a Mealy machine is completely or incompletely specified is not important, so long as it has a completely specified skeleton for which a W-set or a set of DSs can be generated, or from which stems a set of UIOSs, a checking experiment would be applicable.

One result from FSM testing is that mere exercising of transitions alone generally cannot uncover all faults. Exercise of transitions along with verification, whenever possible, produce a better fault coverage.

In the remainder of this thesis, the "UIOv-method" will be used to refer to the UIOv-method, the W-method, as well as the D-method for simplicity. This implies the term "UIOSs" will refer to UIOSs as well as DSs. This is possible since DSs are merely UIOSs whose input sequences

must be identical for all the states in an FSM. Similarly, the term "IO(S,K)s" will also refer to the W-sets since the latter are really IO(S,K)s whose input sequences are identical for all the states and whose members must be consistently numbered among the states in an FSM.

3 TESTING EXTENDED FINITE STATE MACHINES

Techniques for testing FSMs are applicable to testing simple protocols that can be modelled by FSMs. These models, however, are impractical for complex protocols which may require a very large number of states. For instance, the use of sequence numbers in a protocol introduces a different state for each possible value. This is known as the state space explosion problem.

A remedy to the state space explosion problem is by using an extended finite state machine (EFSM) model for the specification of more complex protocols. When FSM testing techniques are applied to such models, executabilities of the resulting test sequences are no longer guaranteed. This chapter discusses how the UIOv-method can be extended to the testing of protocols modelled by EFSMs so that executability of the final test sequence is guaranteed and maximum fault coverage is achieved within the limitations of testing.

3.1 BACKGROUND

An EFSM is an FSM with the addition of minor state variables. These variables form additional enabling conditions in the transitions to reduce the number of states required in the underlying FSM. As a result, different transitions may occur in response to the same combination of

input event and starting state in an EFSM. A transition in an EFSM may be triggered by three types of enabling conditions: the input event, the current state and a boolean expression involving minor state variables. Each transition now consists of three operations: the output operation, the state transition and operations that alter values of the minor state variables. The "state" of an EFSM no longer refers to the value belonging to the STATE variable alone but it refers to the values belonging to the minor state variables as well. The "state" of an EFSM will be referred to as its total system state (TSS) from now on.

3.2 THE EFSM TABLE

Test sequences generated from FSMs are generally done according to their directed graphs. A similar graph can be constructed for EFSMs; however, since minor state variables need to be considered as well in EFSMs to ensure the final test sequences are executable, when these variables are included in an EFSM directed graph, the graph can become very cluttered and difficult to use. The tabular format is thus better suited for representing EFSMs for testing purposes and is used in here as a tool for the test sequence generation procedure.

Columns in the EFSM table are labelled by the starting TSSs of the EFSM transitions. Rows in the EFSM table are labelled by the final TSSs of the EFSM transitions. Each table entry records a transition and can be labelled by either the transition identifier, t*, or the input and output operations within the transition. The EFSM table thus contains information on the semantics and syntax of an EFSM. By considering both, instead of syntax alone, executability is taken into consideration during the test sequence generation process.

Table 3.1 shows a simple example of an EFSM table. There is only one minor state variable, c, in the example. When c is absent at the current TSS, C.TSS, or at the next TSS, N.TSS, it indicates c can assume any value.

C.TSS / N.TSS	s1 c<2	s1	s1 c=2	s2	s2 c<2	s2 c=2
s1 c:=c+1	a/0					
s1		b/-				
s2 c:=0			a/3			
s2				a/-		
s2 c:=c+1					b/0	
s1 c:=0						b/3

Table 3.1 An EFSM table.

3.3 EXTENSION OF THE UIOV-METHOD

The UIOv-method is extended in this section to be applied to EFSM testing. An EFSM is an FSM with additional minor state variables, hence, the UIOv-method is directly applicable to testing its FSM portion. The UIOSs and preambles, however, must be carefully selected to prevent executability problems in the final test sequence. In addition, since minor state variables contribute to the TSS, they must also be verified the same way STATE variables are in FSM testing.

3.3.1 UIOSs Selection In An EFSM

For simplicity, those transitions in an EFSM whose starting TSSs involve p-uses of minor state variables will be referred to as "e.transitions" in the subsequent discussion. Those transitions that do not will simply be referred to as "transitions." For example, referring to Table 3.1, those transitions with the following I/Os are e.transitions: a/0, a/3, b/0 and b/3.

Executability problems may exist in the chosen UIOSs if they contain e.transitions and their enabling conditions are not part of the UIOSs. An uncomplicated way to prevent executability problems in the UIOSs chosen is to "remove" all the e.transitions in the EFSM table to form a table representing a pure FSM. The search algorithm in [Sabn88] for UIOSs is now directly applicable to the table. E.transitions are all the transitions found under the columns labelled with TSS variables consisting of minor state variables. In practice, the e.transitions are not physically removed but are simply ignored during the search. Although this method may not produce minimum UIOSs, its advantage is that executability is guaranteed within the UIOSs. This implies no special preamble is required to enable any UIOS. As will be seen later on, special preambles can considerably increase the final length of the

test sequence and may be problematic in the testing of e.transitions.

If the resulting FSM table produces only non-verifiable UIOSs which are UIOSs whose input sequences are not applicable to other states for verification, then ~Uv must involve e.transitions. This may require special preambles to permit executability during ~Uv. The tail states of these preambles must be verified using their respective UIOSs to ensure ~Uv is performed for the correct state. These preambles are formed only for the purpose of an executable ~Uv procedure. The Uv and ~Uv procedures during testing will use these special preambles.

If a state in the FSM table does not have an UIOS, then the EFSM table has to be used to find an UIOS for that state. The e.transitions in the resulting UIOS require the following attention. If within the UIOS the p-use of the minor state variable in an e.transition is enabled by a transition that is also within the UIOS, then the UIOS is executable by itself and does not require a special preamble. If, however, the enabling transition, et, is not within the UIOS, then the preamble for the state, s, to which the UIOS belongs must include et. Two possibilities can occur. The et may have s as its ending state. This is a problem if there exists more than one incoming arc to s.

This implies the UIOS cannot be used to verify the tail states of the other incoming arcs because the UIOS would not have been enabled. Another UIOS or a set of IO(s,k)s would have to be used for state s. If, however, there is only one incoming arc to s and it enables the UIOS, then the UIOS may be used for s. The other possibility is that the et ends at a state other than s, in which case a partial preamble can be formed from the initial state of the EFSM to the et, and def-free paths joining this partial preamble to each of the starting states for the incoming arcs to s complete the preambles. The UIOS for s can be used provided these preambles exist for all the incoming arcs to s. Each of the preambles must have its tail state checked to ensure it arrives at the correct state before it is used in the Tt procedure.

Using special preambles and having to verify their ending states is a penalty to pay when UIOSs have executability problems. The resulting test sequence, as well, may not be minimal. In addition, if the transition to be tested is an e.transition which requires an et to be included in the preamble, then the preamble needs to enable both the e.transition and the UIOS. Such a preamble may not exist, in which case another UIOS has to be found. There may perhaps be an executable set of IO(s,k)s for state s

that would reduce the number of special preambles required and produce a shorter test sequence. An optimization technique such as that used in [Shen89] would find the best alternative.

For the example in Table 3.1, transitions a/- and b/- constitute UIOSs for the states s2 and s1 respectively. Since they are not e.transitions, they do not require special preambles to enable them.

3.3.2 Verification Of TSS Variables

The UIOv-method verifies a state in an FSM with its UIOS. Similarly, in an EFSM, values of minor state variables that contribute to the TSS should also be verified so that the TSS is verified, not just the STATE variable in the EFSM.

In the same way that a state is verified by the UIOS that is unique to it, an TSS can also be verified by the I/O sequence that is unique to it. However, it may not be possible to verify the whole TSS as a unit. For instance, there may not exist a transition whose C.TSS is identical to the N.TSS to be verified so that an I/O sequence that begins at that C.TSS transition can perform the verification. The reason is as follows. The C.TSS for a particular transition may not involve all the minor state variables that appear in

all the C.TSSs in the EFSM. Unspecified variables are allowed to assume any value. As a result, an N.TSS for a transition may not be verifiable as a whole if there does not exist a transition with an C.TSS which contains p-uses of all the variables that contributed to the N.TSS. An example would be the transition a/- in Table 3.1. This transition cannot be used to verify the N.TSS for the transition a/3 which has the variable c reset to 0. The C.TSS for a/- merely requires that the STATE variable be at s2, it says nothing about the variable c. Hence, exercising a/- after a/3 cannot verify c although it can verify that the final STATE is at s2. The c variable thus has to be verified separately. For C.TSSs, each variable involved may also have to be verified separately.

3.3.2.1 Verification of N.TSSs

Verification of final TSSs, or N.TSSs, may be achieved using I/O sequences. Each definition of a minor state variable that contributes to the N.TSS as well as the definition of the STATE variable has to be verified. The STATE variables can be verified the same way they are verified in the UIOv-method. For the minor state variables, an I/O path is used to lead the defined variable to a transition whose C.TSS contains a predicate which uses that

variable definition. The definition of the minor state variable can then be verified using an I/O sequence that begins at that C.TSS.

Referring back to the final TSS for the a/3 transition where the STATE variable is set to s2 and c is reset to 0. None of the C.TSSs is identical to it; however, the C.TSS s2 AND c<2 can verify that c is not at 2 and STATE is at s2; as well, the sequence b/0.b/0.b/3 beginning from that C.TSS can verify that c was indeed reset to 0. Each execution of b/0 increments c by 1 and transition b/3 is not possible unless c has reached 2 after two increments. As a result, the subsequence b/0.b/0 would have incremented c to 2 if it were at 0 to begin with, and the I/O b/3 would verify that c had been incremented to 2 correctly. Hence, the sequence can verify that c was reset to 0 correctly. Similarly for the N.TSS s2 and c=c+1. The variable c can be verified that it is incremented correctly by initially setting it to 0, then to 1. When it is at 0 and STATE is at s2, the situation is identical to that mentioned above and the sequence b/0.b/0.b/3 verifies that STATE is at s2 and that c has been both set to 0 correctly and incremented correctly. When c is at 1 and STATE is at s2, the sequence b/0.b/3 verifies that STATE is at s2 and c is incremented correctly. The b/0 transition increments c to 2, and the b/3 transition checks

that c is correctly at 2. Hence, c must have been at 1 to begin with.

3.3.2.2 Verification of C.TSSs

Verification of the starting TSS, or C.TSS, for a transition requires verifying that the STATE variable is correct and any predicate involving a minor state variable is also correct. Verification of the STATE variable is again identical to the Uv and ~Uv procedures in the UIOv-method. Verification of the predicate is somewhat different. Two things have to be checked in the verification of a predicate: effect and correctness. The effect of the predicate has to be checked, whenever possible, to distinguish it from other predicates involving the same minor state variables. This is similar to the STATE variables being distinguished from one another. The distinguishing procedure is also an identification process that identifies the STATE or the predicate by means of an I/O sequence that characterizes the value of the STATE variable or the presence of the predicate. For instance, referring to the example in Table 3.1, the predicate for transition a/0 can be distinguished from that for transition a/3 by setting c to 0 or 1 and inputting the sequence a.a.a or a.a respectively. The predicate for transition a/0 would

prompt the response 0.0.3 or 0.3 respectively. The predicate for transition a/3 can be distinguished from that for a/0 by setting c to 2 and then inputting a.a.a or a.a to the IUT. The response of the IUT would be 3.-.- or 3.- if the predicate were correct. Note that both predicates have to be verified separately the same way that all STATE values have to be checked separately. Simply checking one does not imply the other is correct. Checking the predicate for transition a/3 can also employ only the I/O sequence a/3. The I/O sequences used are thus UIOSs for the predicates.

The other item a predicate has to be checked for is its correctness. For instance, if s1 AND c<4 constitute an C.TSS that enables transition x, then this implies s1 AND c=0, s1 AND c=1, s1 AND c=2 and s1 AND c=3 are all capable of enabling x. To check that the predicate has been implemented correctly, one way is to check all these possibilities. However, this could result in an extremely lengthy test sequence if the predicate were c<100 instead of 4. Instead, the boundary-interior value criterion can be used here and the predicate c<4 can be checked by setting c to 0, 1, then 3 so that the domain of the predicate can be established. The I/O sequence which characterizes C.TSS s1 AND c<4 can then be applied to verify the response of the IUT in each case. Checking for the correctness of the

predicate has to be considered here in the test sequence generation procedure because setting the TSS variables to particular values may require certain sequences of I/Os to be implemented or certain values be used at the time of input. Leaving the verification procedure until the test data selection process may be too late since the I/O sequences have already been determined by then.

3.3.2.3 UIOSs for TSSs

In the same way UIOSs are used to verify states, I/O sequences are used to verify minor state variables. If the variables are "completely specified," then these I/O sequences must be unique, as discussed above, so that one predicate can be distinguished from another involving the same minor state variables. For instance, referring to Table 3.1, since starting TSSs s1 AND c<2, s1 AND c=2 and s1 are all specified, if s1 AND c=2 were used to verify a final TSS, then the I/O sequence that begins at that TSS must not be also producible at the TSSs s1 AND c<2 and s1; otherwise, when the I/O sequence is observed, the c variable at the IUT could have been at 2, less than 2 or any value. The I/O sequence used must thus be unique to the starting TSS. In Table 3.1, this uniqueness exists for all TSSs; that is, at TSS s1 and c<2, a/0 is produced and it is not generated by

any other TSS. This implies the uniqueness of a/0 should be checked, prior to its use, the same way ~Uv checks the UIOSs for the STATE variables, to ensure its uniqueness also exists in the IUT. If TSSs s1 AND c=2 and s1 did not exist, then a/0 would not have to be checked since it does not have to distinguish TSS s1 AND c<2 from s1 AND s1 AND c=2.

3.3.3 The eUIOv-Method

The extended UIOv-method, or eUIOv-method, is the UIOv-method taking executability and the TSS into consideration. It consists of the following procedures.

Uv = preamble(si)@UIOS(si)

~Uv = preamble(si)@~UIOSs(si)

where preamble(si) denotes the preamble that takes the EFSM from its initial state to state si; "@" denotes concatenation; UIOS(si) is the UIOS belonging to si and ~UIOSs(si) are UIOSs that belong to other states in the FSM which should not be producible by si. Uv and ~Uv are done for each STATE in the EFSM. Both procedures together determine the minimum number of STATEs in the EFSM. The choice of preamble(si) for each si must ensure that the subsequent UIOS(si) and ~UIOSs(si) are all executable. If no such preamble(si) exists, then either another set of UIOSs should be used or a set of IO(si,k)s should be used.

Uv and ~Uv must both be done because of the uncertainty of UIOSs without their verifications as discussed in Chapter 2. If UIOSs are used for verifying TSS variables, then their uniquenesses must also be verified in these procedures.

The next procedure in the eUIOv-method is transition testing, or Tt. The testing of e.transitions will be referred to as eTt.

```
Tt = (1) t.preamble(si)@transition(si,sf)@UIOS(sf)
     + (2) t.preamble(si)@UIOS(si)
eTt = (1) par.preamble(sj)@def-free path(sj,si)@
      e.transition(si,sf)@UIOS(sf)
      + (2) par.preamble(sj)@def-free path(sj,si)@
        UIOS(si)
```

If t.preamble(si) is equal to preamble(si) used in Uv and ~Uv, then (2) is not required to verify its tail state to be si again; otherwise, verification is required as indicated by (2). Transition(si,sf) denotes a transition that begins at si and ends at state sf. Par.preamble(sj) is a path that begins at the initial state and ends at state sj. Def-free path(sj,si) is a path that leads from a transition that starts at sj to si. That transition at sj houses the def of the p-use appearing in e.transition. The total preamble, which is the partial path concatenated with the def-free path, must also have its tail state verified as indicated in

(2). This in essence is checking the starting state for e.transition(si,sf). If the UIOSs consist of e.transitions, then embedded within the preambles must be defs that enable those e.transitions. This is why it is preferable to use slightly longer UIOSs if they do not have e.transitions involved. During eTt, a preamble that enables UIOS(sf) as well as e.transition(si,sf) may not exist.

The last procedure in the eUIOv-method is the verification of minor state variables in the TSSs in each transition .

msvv = t.preamble(si)@transition(si,sf)@
def-free path(sf,sk)@I/Os(msvf)
OR par.preamble(sj)@def-free path(sj,si)
@e.transition(si,sf)@def-free
path(sf,sk)@I/Os(msvf)

where I/Os(msvf) is the sequence of I/Os to verify the definition of one or more minor state variables in transition(si,sf) or e.transition(si,sf). The def-free path(sf,sk) leads from sf to the e.transition starting at sk which houses the predicate involving the definition in transition(si,sf) or e.transition(si,sf). If I/Os(msvf) has to be unique, then the verification procedure identical to Uv and ~Uv has to be carried out for it to check its uniqueness. Predicates in the C.TSSs are also verified in

this procedure using the methodology discussed in the previous section.

To return to the initial idle state, if the reset feature r/- is not available, then a sequence of I/Os has to be used to bring the machine back to the initial state. These sequences of I/Os should be verified prior to their uses to ensure they end at the initial state. Their verification can be done as follows for a state si in the EFSM.

preamble(si)@postamble(si)@UIOS(idle)

Parameter variation methods such as the boundary-interior value criterion or the most commonly used values can be used during the test sequence generation procedure to achieve more thorough testing, in such cases, test sequences may be repeated to accommodate the various values selected.

3.3.4 An Example

The eUIOv-method is applied to the EFSM in Table 3.1 as an example. The initial state is taken to be s1 and the c variable is assumed to be initialized to 0 at the beginning of each implementation. The reset feature r/- is assumed to exist to bring each state in the EFSM back to s1. Its verification is assumed to be correct.

The UIOSs for s1 and s2 are b/- and a/- respectively. The UIOSs for verifying c<2 at STATE s1 starts with a/0 and for c=2 at s1 is a/3. Their uniqueness need not be verified separately in the ~Uv procedure since they share identical inputs. The UIOSs for verifying c<2 at s2 starts at b/0 and for c=2 at s2 is b/3; again, their uniqueness are already confirmed in the Uv procedure. The UIOSs for verifying different c values at the same STATEs thus need be verified only in Uv. No preamble is required to go to s1. The preamble for s2 is a/0.a/0.a/3. Transitions a/3, b/0 and b/3 are e.transitions and they require the following preambles to ensure their executabilities. The preambles are found by traversing backwards from each e.transition to the initial s1 state. The predicates in the e.transition are considered and those transitions whose N.TSSs enable these predicates are tracked and included in the preambles.

preamble(a/3) = a/0.a/0

preamble(b/0) = a/0.a/0.a/3

preamble(b/3) = a/0.a/0.a/3.b/0.b/0

The following test subsequence constitute the Uv and ~Uv procedure for the EFSM in Table 1.

Uv: b/-

a/0.a/0.a/3.a/-.r/-

a/0.r/-

a/0.a/0.a/3.r/-

a/0.a/0.a/3.b/0.r/-

a/0.a/0.a/3.b/0.b/0.b/3

~Uv: a/0.r/-

a/0.a/0.a/3.b/0.r/-

{The following verifies the special preambles.}

a/0.a/0.b/-.r/-

a/0.a/0.a/3.a/-.r/-

a/0.a/0.a/3.b/0.b/0.a/-.r/-

The following test subsequence constitutes the Tt, eTt and msvv procedures. The information in the parenthesis indicate the N.TSS to be verified in the transition under test. In this particular example, the msvv procedure to verify TSSs is carried out with the eTt procedure whenever possible to reduce the length of the final test sequence. Since all the preambles used have already been verified in the Uv and ~Uv procedure, they do not have to be verified here again.

Tt: b/-(s1).b/-

a/0.a/0.a/3.a/-(s2).a/-.r/-

eTt & msvv: a/0(s1 AND c=1).b/-.a/0{c=2}.

a/3.r/-

a/0.a/0.a/3(s2 AND c=0).a/-.b/0{c=1}.

b/0{c=2}.b/3

a/0.a/0.a/3.b/0(s2 AND c=1).a/-.b/0{c=2}.

b/3

a/0.a/0.a/3.b/0.b/0.b/3(s1 AND c=0).b/-.

a/0{c=1}.a/0{c=2}.a/3.r/-

Although the UIOSs b/- and a/- for STATEs s1 and s2 are used to verify the STATEs indicated in the parentheses above, the STATEs can also be verified at the same time the c variables are verified in this particular case since the UIOSs for the c variables are capable of distinguishing among different c variables as well as among different STATEs. However, if these UIOSs are used for this purpose, they should first be verified in procedure ~Uv to ensure they indeed have these capabilities. Values of the c variables that have to be verified are recorded in braces to track the verification procedure. For instance, {c=2} tracks the c variable while it is being incremented by one. The value of the increment, in turn, is verified by the transition that is enabled when c=2. Verification of the effect of the predicates is implied in this example in the verification of the definitions of the corresponding variables. Domains of the predicates are also implicitly verified in the eTt and msvv procedure above where c was set to 0 first, then to 1.

The final test sequence may be reduced by eliminating those test subsequences that are completely contained in other subsequences.

The conclusion one can draw from a test that uses the test sequence generated above is that if the EFSM has only two states, then an IUT that passes the above test possesses an EFSM skeleton identical to the specified EFSM. The number of states must again be restricted to be equal to that in the specification. This again implies there may exist extra transitions in the IUT that are not in the EFSM specification.

3.3.5 Summary Of The eUIOv-Method

The eUIOv-method is essentially a checking experiment that checks an IUT for each EFSM transition given in the specification. The Uv and ~Uv procedures in the eUIOv-method establish the minimum number of states in the EFSM and they verify the uniqueness of the UIOSs. The Tt and eTt procedures check each transition specified in the EFSM by observing its current and final STATE values as well as its I/O operations. The msvv procedure checks the definitions of minor state variables that contribute to the N.TSS of a transition. Whenever possible, these definitions are verified to be correct. The minor state variables that

contribute to the C.TSS of a transition are also checked for their correctness and their effects. Their effects are checked by distinguishing predicates involving identical minor state variables and STATE variables from one another. Their correctness are checked using two boundary values and an interior value to establish the domains of the predicates whenever necessary.

The eUIOv-method requires that values of minor state variables that contribute to C.TSSs are tracked during the test sequence generation procedure in order to ensure executability of the final test sequence and chosen values for testing predicate domains are selected. The tracking procedure may require some computations as well as pre-selection of test data. The eUIOv-method thus couples the test sequence selection process with the test data selection process whenever necessary. These two processes are traditionally carried out separately in software testing where the test sequences generated may face executability problems.

3.4 FAULT COVERAGE

The eUIOv-method is based on the UIOv-method; hence, it produces a fault coverage that includes that which is produced by the UIOv-method. This implies the eUIOv-method

is also capable of detecting all I/O errors and STATE errors in an EFSM provided its set of STATES is no greater than that which is in the specification. This also implies extra I/Os present in the IUT will not be detected. However, referring to the term "conformance," and given the limitations of testing, this is acceptable. As for the testing of the TSS variables, again, extra statements cannot be detected. The effects of extra statements are discussed in greater detail in the next chapter. If the predicates constituting the C.TSSs involve mathematical expressions, or if variables in the N.TSSs are assigned new values with mathematical expressions, then their correctness may not be guaranteed even though they may display correct values during testing. This issue will also be discussed in greater details in the next chapter.

4 DATA FLOW TESTING

While simple protocols can be modelled by FSMs, the more complex ones require EFSMs to accurately describe their functions and behaviors. EFSM models alone, however, cannot detail the flow of information that often takes place from one control phase to another. For instance, input primitives often have accompanying parameters that are input at one control phase and stored by the IUT in the form of a variable for usage or reference in another control phase. Instead, the functions of these parameters and variables can be separately described in a data flow transition embedded within the appropriate EFSM transition. These parameters and variables should also be checked during testing to ensure their functions are correctly carried out and the output primitives are accompanied by the correct parameters. This checking process can be accomplished by means of a data flow testing procedure.

An overview of the data flow testing procedure gives a process similar to the msvv procedure in the eUIOv-method, except here, instead of a path tracking the definition of a TSS minor state variable to its usage, each data path tracks the definition of a parameter or a variable to its usage. Each path is exercised and verified whenever possible during testing to ensure the definition, or usage, under test is

correct. Any predicate that appears in the data flow transitions also has its effect and correctness verified the same way they are verified in the msvv procedure. The data flow testing technique is based on static data flow analysis. It is also based on the lesson learnt from FSM testing that transition exercising alone is not enough, whenever it is possible, elements that are not directly observable in each transition should be verified as well.

4.1 STATIC DATA FLOW ANALYSIS

Static data flow analysis originated in compiler optimization [Hech77]. It was applied to software testing in [Rapp85] and is now generally accepted as a structural testing strategy [Weis85].

Essentially, static data flow analysis tracks each input variable through a program until it is ultimately used to produce output values. Associations between the definition of, or the assignment of a value to, a variable and the usage of that variable are identified and examined during testing. The concept is based on each defined variable must reach a use while each variable to be used must have been previously defined.

The following definitions are introduced to enhance subsequent discussions. In static data flow analysis

terminology, the process of assigning a value to a variable is called the definition, or def, of the variable. The use of the variable in a predicate is called a p-use. The use of the variable in determining the def of another variable is called a c-use [Rapp85].

Ural [Ural88] was the first to apply static data flow analysis to the conformance testing of protocols. The result was a method of generating test sequences based on rigorously-defined data flow relationships which allow easier interpretation of test results. In comparison with the method proposed by Sarikaya [Sari87], this method generates more comprehensive tests in terms of structural coverage; and in terms of functional coverage, Sarikaya's was more complete [Ural88].

4.2 DATA FLOW PATHS

In a protocol, a data flow path for a particular parameter or variable refers to a sequence of data flow transitions where the first transition contains a definition of the parameter or variable and the last transition contains a usage of that parameter or variable. These two transitions are connected by a sequence of transitions based on the underlying EFSM modelling the control structure of the protocol.

A data flow path of length one is a path embodied within the same data flow transition; that is, the usage of the variable or parameter immediately follows its definition. This type of flow path is exercised whenever the EFSM transition is executed.

Data flow paths which extend from one EFSM transition to another require that the particular sequence of transitions be examined during testing. This sequence may or may not already be covered during EFSM testing. This type of flow path is referred to as "inter-transitional" [Chan89a] and requires identifying the transition at which the definition first occurs, the transition at which the usage occurs, and connecting these two transitions with a path based on the control structure. The path must be executable and it must be free of any re-definition of the variable or parameter; that is, the definition is "live" in the connecting path.

4.3 DATA FLOW TESTING

In testing the data flow of a protocol, the variables and parameters that appear in the data flow transitions embedded within the EFSM transitions are tracked from the moment they are defined until they are used either for computational purposes or as predicates. When computations

eventually lead to the definition of an output primitive parameter, the path is further extended to that parameter in order that the computations may be verified to be correct. If verification is not possible, the flow paths would only be exercised; they stop at the usage of the variable or parameter. For predicate uses, the variable or parameter involved may not be verifiable in terms of externally observable events, in which case the flow path again would only be exercised and stops at the transition containing the predicate. If verification is possible, where the actions enabled by different p-uses differ and are manifested in externally observable events, then these events would be checked to verify the p-uses. For instance, a different value assigned to an output parameter depending on the predicate constitutes an externally observable event. Usages are thus again verified according to the effects they have on determining values for the I/Os or the order in which they appear. The correctness of these usages may be verified during the test data selection procedure by selecting appropriate values for test inputs. The boundary-interior value criterion can again be used for the process. However, if special preambles are required so that the predicates, with selected values assigned to the variables involved, may be enabled, then they must be considered again

during the sequence generation procedure to ensure the final test sequence is executable.

It should be noted that in a protocol specification, not all defs are necessarily used in which case flow paths do not exist for those defs. An example of this type of def is a def for the purpose of initializing a variable where the initialized value is not used.

While there may be more than one def of a variable, there may also be more than one usage of it. Each def should be exercised and verified at least once; the same applies for each use.

More than one usage may exist for a single def of a variable; hence, some usages may not be tested if paths were formed from the defs alone since the number of paths would then be limited by the number of defs. Hence, usages that have been missed should be tracked back to their appropriate defs to form additional flow paths so that they can be verified as well.

When the usages are p-uses, they require that the defs produce specific values corresponding to those in the p-uses. In these cases, the search for a def of the variable must take into consideration the value that is assigned to the variable. Similarly, if a p-use were to verify a def statement, the value of the def must be considered to ensure

executability. When the usages are c-uses, the values that arise from the def statements are not critical in the test sequence generation process.

4.4 THE DATA FLOW TESTING PROCEDURE

As previously mentioned, the procedure for testing data flows (DFTP) is similar to the msvv procedure in the eUIOv-method except all parameters and variables are considered. All flow paths that begin at transition(si,sf), for testing the defs of variables in that transition, are appended to the appropriate preambles to form a test subsequence in the following manner. The "def-free DF path" indicates the path that brings the transition, which houses the def, to the transition that uses or verifies, if possible, the variable. The latter transition is included in the path.

```
DFTP = preamble(si)@transition(si,sf)@def-free DF
         path(transition)@postamble
       OR
       par.preamble(sj)@def-free path(sj,si)@
         e.transition(si,sf)@def-free DF path
         (e.transition)@postamble
```

The same applies to the flow paths that test the usages of variables in transition(si,sf) or e.transition(si,sf). The test subsequences they form are in the following format.

```
DFTP = preamble(sk)@def-free DF path(sk,si)
         @transition(si,sf)@postamble(sf)
       OR
       par.preamble(sj)@def-free path(sj,sk)@
         @def-free DF path(sk,si)@e.transition(si,sf)
         @postamble(sf)
```

The "def-free DF path" here is the def-free path that includes the transition housing the def of the variable, and which connects the transition to the transition where the usage is, transition(si,sf). The paths to transition(si,sf) and e.transition(si,sf) are preferably those previously used in the Tt and eTt procedures. If new paths are used, perhaps due to executability problems in the def-free DF paths or due to usage requirements, then the tail states of these new "preambles" to transition(si,sf) or e.transition(si,sf) must be verified to ensure they end at the correct state so that data flow paths used to verify defs of variables (def-itDFPs) are spanned from the correct transition, and paths used to verify uses of variables (use-itDFPs) are verifying uses situated at the correct transitions.

The EFSM table used in the eUIOv-method is augmented with enabling conditions for the data flow transitions to aid in determining executable flow paths between two

selected data flow transitions. If the data flow transition is preceded with a predicate involving minor state variables, then their p-uses and defs would be extracted to form new TSSs in the EFSM table. New columns or rows may thus be added in the augmented table along with new table entries. Those predicates that involve input parameters would be indicated in the appropriate table entries to reflect specific values are required at the time of input during testing.

4.5 AN EXAMPLE

The following is a simple example to illustrate the DFTP. Referring back to the EFSM table in Table 3.1, let there be data flow transitions added to the EFSM transitions as follows. The data flow transitions are denoted by DF and are enclosed in a block delimited by BEGIN ... END. Each EFSM transition is identified by a number.

```
transition 1:  IN: a(a.addr)
               C.TSS: s1 AND c<2
               N.TSS: s1 AND c:=c+1
               OUT: 0(a.addr)
transition 2:  IN: b(b.addr)
               C.TSS: s1
               N.TSS: s1
```

```
                OUT: -
transition 3:   IN: a(a.addr)
                C.TSS: s1 AND c=2
                N.TSS: s2 AND c:=0
                DF: BEGIN  address:=a.addr  END
                OUT: 3(a.addr)
transition 4:   IN: a(a.addr)
                C.TSS: s2
                N.TSS: s2
                OUT: -
transition 5:   IN: b(b.addr)
                C.TSS: s2 AND c<2
                N.TSS: s2 AND c:=c+1
                OUT: 0(b.addr)
transition 6:   IN: b(b.addr)
                C.TSS: s2 AND c=2
                N.TSS: s1 AND c:=0
                DF:  BEGIN  3.addr:=b.addr@address  END
                OUT: 3(3.addr)
```

The flow of each a.addr and b.addr to an output parameter in transitions 1, 3 and 5 can be verified by simply executing the transitions and observing that the output parameters enclosed in parenthesis are correct during the EFSM testing procedure. In transition 3, a.addr had to

be remembered by the address variable so that it may be used later in transition 6 to be appended to b.addr to form the output parameter 3.addr. In EFSM testing, this flow of information may have been ignored since it requires a particular sequence of transitions to occur. The DFTP is thus applied here to check this flow of data. A data flow path which connects transition 3 to transition 6 is 3.5.5.6. or, in terms of an I/O sequence, a/3.b/0.b/0.b/3. This path tracks the def of the address variable in transition 3 to its usage in transition 6. Since the variable is then used to compute the output parameter 3.addr, the def of the variable can be verified by observing the value of the variable, 3.addr. If this data flow path were not included in the EFSM testing process, then it would be tested separately in the DFTP by building a preamble that brings the EFSM from its initial s1 state to the TSS that enables t3, and concatenating the preamble to the data flow path to form the following test subsequence.

DFTP: a/0.a/0.a/3.b/0.b/0.b/3

The above test subsequence tests the correctness of the def of the variable address; it also tests the correctness of its usage. In addition, from this test subsequence, it can be seen that the input parameters a.addr and b.addr are used correctly to form the output parameter 3.addr.

4.6 CHAPTER SUMMARY

The data flow testing procedure is based on static data flow analysis and the lessons learnt from FSM testing. It is different from static data flow analysis because a def is not merely traced to its usage, but it is verified as well whenever possible. Similarly, a usage is not just exercised but its effect also verified whenever possible. When the usage is a p-use, there are two things that can be checked: its effect and its correctness. Its correctness can be verified by applying parameter variation methods as discussed in the previous chapter to establish the domain of the predicate and then by observing its effects, via its characterizing I/O sequence, to verify its existence in the IUT. When the usage is a c-use and if it involves mathematical expressions or other parameters or variables, then its correctness is more difficult to check. This is discussed in greater details in the next section.

The effects of usages are verified as follows. When a variable is used as a c-use and it defines an output primitive parameter, the parameter would be tracked and observed to verify the correctness of the def and the c-use of the variable. When the usage is a p-use, then the effects among different p-uses of the same variable at the same TSS are compared and whenever possible, these effects

are tracked to verify the effect of the p-use. For instance, if the predicate x=2 enables the assignment statement y:=2, and the predicate x<>2 enables the assignment statement y:=4, then if the variable y were used to compute an output parameter in another transition, this output parameter would be tracked and its value checked to ensure that y was assigned the correct value corresponding to the appropriate predicate involving the variable x. If, however, the variable y is itself used in a p-use, then the different p-uses of y and their different effects would again be tracked until they are externally observable or until they are proved to be non-verifiable by looping back to the initial p-uses for the variable x, in which case, the p-use of x would only be exercised.

The DFTP employs the same method of generating test subsequences as that used in the msvv procedure in the eUIOv-method. The result is that a def that was not verified in the EFSM testing procedure is now verified in data flow testing and the preamble ensures that it is located at the expected transition. The same applies to the usages that were missed by the eUIOv-method. The preambles in their test subsequences ensure that the usages are situated at the correct transitions.

4.7 COMMENTS ON THE DFTP

The DFTP can detect errors in the defs and usages of variables provided they are externally observable or verifiable and no extra def or usage statements exist in the IUT. Erroneous defs or usages of variables or parameters that cannot be externally observed and verified may escape detection.

The possible effects of extra def statements in an IUT are as follows. As an example, consider a def-free path t1.t2.t3.t4 where a def of a variable in t1 is to be verified by its use in t4. According to the specification, if the IUT were correctly implemented, then the def is indeed in t1 and the use in t4 indeed verifies it. However, if t3 erroneously redefined the variable in the IUT, which is unknown to and unexpected by the tester, the usage in t4 would have verified the def in t3 instead of the one in t1 as expected. If the verification result were positive and if the def in t1 were in fact erroneous, then the error would have gone unnoticed. If the verification result were negative, then the def in t1 would be suspected of being erroneous when in fact it may be correct. As a result, extra statements may distort test results and may not be detected.

If the uses were p-uses, then it is generally assumed that all possible conditions are considered since this only constitutes good practice. For instance, if x>3 appears in an enabling condition, then x<3 and x=3, or a combination of the two, should also be present; otherwise, testing x>y would require that x be set to 3 and a value less than 3 as well to ensure that the enabling condition is indeed correct. However, this would not be possible if the specification did not specify what the IUT is to do when x<3 and x=3. If the specification did specify them, then if the IUT erroneously missed the condition x>y altogether so that x could have assumed any value, then testing x<3 and x=3 would have detected the missing p-use provided the effect of the p-uses are verifiable. Hence, the best alternative is to specify all possibilities.

If the def statements involved mathematical expressions, then the value used as test input could be just as important as the sequence of test inputs to be used. A well known example is the expression y:=2+x. If x were chosen to be 2, then if y were erroneously set to 2*x, the erroneous usage of x, or the def of y, would have gone unnoticed. For this type of defs, it seems the most appropriate method of selecting test data is by knowing ahead of time what could possibly go wrong. However, this

is impossible because everything "under the sun" would have to be taken into consideration. Instead, one way to select test data could be with the boundary-interior value method, where extreme values and middle ranged values are used. A more practical approach would be to select as test data all those values that are frequently used in a real implementation of the protocol. This approach can also be extended to apply to test sequence selection; that is, select those sequences that are most likely used in a real implementation to be tested. This, however, requires close cooperation between the testers and the users.

5 THE HYBRID TECHNIQUE

When the eUIOv-method is augmented with the DFTP, a hybrid technique is formed which is essentially a checking experiment that is applicable to testing complex protocols that are modelled with EFSMs augmented with descriptions detailing the functions of their parameters and variables.

The hybrid technique is also directly applicable to the testing of protocols that are implemented according to their Estelle specifications. The EFSM portion of the protocol, modelling its control flow, is checked using the eUIOv-method in the hybrid technique. The Pascal statements in the Estelle specification which describe the data flow aspect of the protocol can be used to guide the DFTP within the hybrid technique to exercise and verify, whenever possible, the input and output primitive parameters in the IUT as well as its minor state variables. The hybrid technique is believed to be adequate for testing Estelle specified protocols based on the following observations.

In FSM testing, the UIOv-method tests each transition by observing its I/O operations and verifying that its starting and final states are correct. As a result, all elements in an FSM transition are exercised and verified. In EFSM testing, the eUIOv-method tests each transition by observing its input and output primitives and verifying that

its starting and final TSS variables are correct. As a result, the eUIOv-method also exercises and verifies each element in an EFSM transition. When it comes to testing an Estelle specification, since it houses EFSM transitions and data flow transitions, its EFSM transitions can be tested using the eUIov-method. The data flow transitions can be tested by exercising and verifying, whenever possible, statements involving the input and output primitive parameters as well as the minor state variables. Once all these parameters and variables are also tested, all elements in an Estelle transition would have been exercised and verified. The hybrid technique can thus be viewed as an extension to the eUIOv-method where, not only minor state variables are examined, but all other variables as well as parameters contributing to data flow are examined as well.

The hybrid technique is thus directly applicable in the testing of protocols that are implemented according to their Estelle specifications. During testing, the IUT is checked for each transition in the Estelle specification one at a time, exercising and verifying each statement in the transition whenever possible.

This chapter provides an example of how the hybrid technique can be used to generate a conformance test sequence from an Estelle specification to test an IUT

generated from the specification. The Class 0 Transport Protocol is used as the sample protocol in here.

5.1 BACKGROUND

This section provides a brief introduction to Estelle and its normal form.

5.1.1 Estelle

Estelle is a formal description technique developed by ISO [ISO88] for the specification of, but not limited to [Vuon88b], communication protocols and services.

Estelle is based on an EFSM model for specifying the control structure of a protocol and a set of Pascal statements for specifying its data flow. Estelle protocol specifications are implementation oriented and compilers have been developed to generate automatic protocol implementations from their Estelle specifications [Vuong88a].

Estelle defines a system using a hierarchy of modules. Parent modules may dynamically create and destroy child modules. Modules contain abstract message interfaces or interaction points that can be connected to form communication channels. Selected sets of messages or

interaction primitives may be exchanged across these channels to facilitate communication among modules.

A protocol can be specified using one or more modules. The actions of each module, corresponding to the behaviors of the protocol, are defined by a set of EFSM transitions. The complete "state" of the module, or the protocol machine, is captured by the STATE variable as well as other minor state variables if they exist. Each transition in the module is composed of a set of operations and a set of enabling conditions. The enabling conditions may or may not include an input event, the current state and a boolean expression involving minor state variables or input primitive parameters. Each of these conditions is optional and, if absent, is assumed to be satisfied. The enabling condition involving minor state variables generally indicates the circumstance under which different transitions are triggered in response to the same pair of input event and starting state. Once the enabling conditions in a transition are satisfied, a set of operations may be carried out. These may include an output operation, a state transition, and a set of operations, specified in Pascal, involving minor state variables or interaction primitive parameters. The Pascal statements may include procedure and function calls as well as conditional and loop statements.

Conditional statements permit a choice of output operations to be performed within a single transition.

5.1.2 Normal Form Estelle Specifications

An Estelle specification may be simplified to a form that is more easily represented by directed graphs. This simplified form of an Estelle specification is known as its normal form specification (NFS) [Sari86].

Modules in the Estelle specification are combined using symbolic execution to form a single module in the NFS. States that have been grouped into state sets in the Estelle specification are expanded into the individual states they represent in an NFS. This is done by replicating the appropriate transition a number of times equals to the number of states in the state set. Conditional and loop statements are also eliminated in the NFS by replicating the transition which houses those statements a number of times equals to the number of branches or the size of the loop; this assumes loops do not have variable bounds. Procedure and function calls are eliminated within transition bodies using symbolic execution. Procedures are assumed to be non-recursive and do not contain loops with variable bounds.

The overall appearance of an NFS is that of a giant EFSM with Pascal assignment statements embodied within the

EFSM transitions involving minor state variables and interaction primitive parameters. An NFS displays the two types of flows in an Estelle specification more explicitly. Control flow is marked by changes in the STATE variable and the input and output operations involving the interaction primitives. Data flow is identified by the operations, specified in Pascal, on the minor state variables and interaction primitive parameters.

An Estelle specification henceforth refers to the original form of the specification in subsequent discussions.

5.1.2.1 Example of an NFS

[Ural88] provides an example of an NFS of the Class 0 Transport Protocol. This NFS is polished and reproduced here in Appendix A. All redundant assignment statements are eliminated; details that have been left out are added back in.

Each transition in the NFS is identified with a number. The current state or the starting state of each transition is denoted with FROM. The final state or the next state of each transition is denoted with TO. The PROVIDED clause expresses the enabling condition involving minor state variables or input primitive parameters. In an Estelle

specification, the PROVIDED clause denotes only the conditions under which different state transitions occur for the same combination of input event and starting state. In an NFS, the PROVIDED clause also houses branching conditions responsible for branch paths within transitions. An example of this is transitions t3 and t4. The branching conditions "cr.in.max.tpdu.size <> nil" and "cr.in.max.tpdu.size = nil" have been extracted from a single transition in the Estelle specification; that transition was replicated twice to form transitions t3 and t4 in the NFS. Each input interaction primitive is preceded by the keyword WHEN. A transition may not have an input interaction primitive. An example is transition t14. This type of transition is known as spontaneous transitions. Each output interaction primitive is preceded by OUT. Each output operation resides inside a block in the body of the transition delimited by BEGIN ... END. Operations to be performed at each transition involving minor state variables and interaction primitive parameters are expressed in Pascal and are also enclosed in the block delimited by BEGIN ... END.

The interaction primitive parameters are enclosed in parenthesis following their respective primitives. These parameters, when referred to inside the transitions, are preceded by the primitive names and either "in" or "out" to

indicate whether they belong to an input or output primitive respectively. When an output primitive parameter is assigned a constant value, the constant is directly placed in the appropriate position in the OUT statement. An example is in t1, where "normal" is directly placed at the output parameter list. When the output primitive parameter is assigned a value identical to that of a minor state variable, the variable name is directly placed in the appropriate position at the corresponding OUT statement. An example is in t3, where "qts.estimate" replaces the last parameter name for the primitive tcind. These direct placements eliminate redundant assignment statements.

5.2 REFINING THE NFS

An Estelle specification in its normal form provides a graphical description of a protocol in terms of an "enriched" EFSM. The NFS is further refined and reformatted in this section to obtain canonical transitions and to separately bring out the underlying EFSM and data flow for testing purposes.

5.2.1 Canonical Transitions

In order to form canonical transitions in an NFS, expressions ORed in a PROVIDED clause should be extracted

and the transition replicated accordingly. The reason is because OR is merely a way of combining different system starting states which trigger the same transition when a particular input is received. An example of this kind of combination is the state set feature in Estelle. State sets combined different starting STATE variables that trigger the same transition upon receiving the same input into one set. In forming the NFS, each state in the set would be extracted and the transition replicated for each member in the set. The same should thus be done for all the variables responsible for the TSS.

For example, if transition t3 in Appendix A had OR in its PROVIDED clause instead of AND, then t3 would be replicated as follows:

```
WHEN cr(...)
FROM idle
PROVIDED cr.in.max.tpdu.size <> nil
TO wftr
t3:  BEGIN
     remote.refer := ...;
     ...
     END;

WHEN cr(...)
FROM idle
PROVIDED cr.in.option = ok
TO wftr
t3i: BEGIN
     remote.refer := ...;
     ...
     END;
```

5.2.2 Reformatting the NFS

Since Estelle uses an EFSM to model the control structure of a protocol and a set of Pascal statements to describe its data flow, conformance testing of a protocol implemented according to its Estelle specification should have its control structure checked against the specified EFSM and its data flow checked against the Pascal descriptions [Chan89a]. A test sequence generation technique for an Estelle specification should thus consider both its EFSM model as well as its data flow statements. The NFS is reformatted here to separate and bring out the underlying EFSM and data flow to aid this test sequence generation process. Factors that contribute to executability problems are also brought out in the reformatted NFS.

5.2.2.1 Executability Problems

There are two types of enabling conditions in an Estelle specification: those that contribute to executability problems and those that do not. Conditions that involve input primitive parameters belong to the latter category because appropriate values can always be input at the time of testing to permit executability. On the other hand, conditions involving minor state variables could

contribute to the executability problem as these variables are not directly accessible. Their values may depend on previous input primitive parameters or they may require that a particular sequence of I/Os occur before they do. The executability problem thus surfaces if these variables are not considered during the selection of test sequences. The roles of these variables in the EFSM and data flow are made explicit in the reformatted NFS to aid executability.

5.2.2.2 The Reformatted NFS

The major difference between an NFS and a reformatted NFS is that the latter separates and brings out the EFSM and data flow from an Estelle specification.

The following features in an Estelle specification constitute an EFSM. Minor state variables that appear at the PROVIDED clause together with the STATE variable preceded by FROM constitute the current TSS in the EFSM. The values obtained from their alterations in the body of a transition and the new STATE variable indicated by the keyword TO constitute the final TSS in the EFSM. Input primitive parameters found at the PROVIDED clauses in an Estelle specification together with their input interaction primitives denoted by WHEN constitute the set of inputs in the EFSM. The output set is composed of the output

primitives indicated by OUT. The remaining statements in the Estelle specification belong to its data flow.

An EFSM can be extracted from either an Estelle specification or its NFS by gathering statements that correspond to the features mentioned above. In an NFS, this requires that some enabling conditions in the PROVIDED clause be separated and some Pascal statements corresponding to data flow be duplicated.

5.2.2.3 The Enabling Conditions

Enabling conditions in the PROVIDED clause in an NFS that enable different transitions must be separated from those branching conditions that affect different paths within a single transition. In an Estelle specification, the first set of conditions is expressed in the PROVIDED clause and contributes to the EFSM; the second set of conditions is found in the IF clause or the CASE clause inside a transition and contributes to the flow of data. Both types of conditions may involve minor state variables as well as input interaction primitive parameters.

5.2.2.4 The Def Statements

The other major change required of the NFS in its reformation is that statements with defs of variables that

have p-uses in the PROVIDED clause in the Estelle specification must be gathered into one block, rest of the statements with defs of variables or parameters that have future c-uses or p-uses responsible for paths within a transition are gathered into another block. The first block determines the next TSS in the EFSM. The second block brings out the data flow aspect. For those variables that have uses that belong to both blocks, their def statements are duplicated and placed in both blocks.

5.2.2.5 Format of the rNFS

The following shows the general form of the reformatted NFS (rNFS).

```
t*:  WHEN i.i.p. {i.i.p.p. p-uses}
     FROM c.state AND {m.s.v. p-uses}
     TO n.state AND BEGIN
                    {m.s.v. defs}
                    ...
                    END;
     OUT o.i.p.
     PROVIDED ({m.s.v. or i.i.p.p. p-uses})
     BEGIN
     {m.s.v. defs}
     {i.i.p.p. c-uses}
     {m.s.v. c-uses}
     {o.i.p.p. defs}
     ...
     END;
```

where t*: rNFS transition label

i.i.p.: input interaction primitive

i.i.p.p.: input interaction primitive parameter

c.state: current state

m.s.v.: minor state variable

n.state: next state

o.i.p.: output interaction primitive

o.i.p.p.: output interaction primitive parameters

An rNFS transition is composed of an EFSM transition and a data flow transition. The group of statements from the WHEN clause to the OUT clause describes the EFSM transition; rest of the statements starting with the PROVIDED clause describe the data flow. All output primitives are expressed in the OUT clause. If in the original specification, there exists two possible choices of output primitives within a single transition, then its corresponding EFSM portion would be replicated and these output primitives extracted and put into each EFSM transition. The predicate that determines which output primitive to use would also be extracted and put in the FROM clause if it involves a variable; otherwise, it would be put in the WHEN clause. Variables in the predicate are thus treated as TSS variables. If the predicate also affects data flow, then it would be replicated and placed at the PROVIDED clause as well to indicate the data flow paths.

The NFS of the Class 0 Transport Protocol in Appendix A is reformatted into an rNFS shown in Appendix B. Note that

when two rNFS transitions share an identical EFSM transition, their labels indicate them as "tea" and "teb" where "te" labels the rNFS transition and "a" and "b" label its two data flow paths. This type of rNFS transition corresponds to a single Estelle transition with two branch paths within.

The reformatted NFS can be easily generated from an Estelle specification by following the same procedure as that for generating the NFS with the exception that the enabling conditions are kept separate. As well, minor state variables that appear in the PROVIDED clauses are identified and their def statements are extracted and duplicated whenever necessary. If more than one output primitive exists in a transition, they would also be extracted as discussed above.

5.3 ESTELLE EFSM TESTING

This section shows how an Estelle EFSM may be tested using the eUIOv-method. Table 5.1 is an EFSM table representing the EFSM of the Class 0 Transport Protocol extracted from its rNFS in Appendix B.

C.TSS / F.TSS	IDLE	WFCC	WFTR	DATA	DATA o.b.<>0	DATA i.b.<>0
WFCC	tcreq (qr=ok) /cr					
IDLE	tcreq (qr<>ok) /tdind cr (op<>ok) /dr	dr/ ndreq, tdind	tcres (qr>qe) /dr, tdind tdreq/ dr	tdreq /ndreq ndind /tdind nrind /tdind		
WFTR q.e.=...	cr (op=ok) /tcind					
DATA i.b.=0 o.b.=0 q.e.=...		cc/ tccon				
DATA i.b.=0 o.b=0			tcres (qr<=qe) /cc			
DATA o.b.<>0				tdatr /-		
DATA o.b.=0					-/dt	
DATA i.b.<>0				dt/-		
DATA i.b.=0						-/ tdati

Table 5.1 EFSM table for the Class 0 Transport Protocol.

5.3.1 Spontaneous transitions

Spontaneous transitions are Estelle transitions that do not require inputs to be enabled. These transitions are enabled as long as their TSSs are satisfied. A spontaneous transition thus verifies all or part of a final TSS for another transition when it is enabled and its output is observed during testing. In addition, its absence is capable of verifying all or part of a final TSS for another transition whose TSS values do not equal to those of the starting TSS for the spontaneous transition. For instance, if a spontaneous transition has the following starting TSS and output:

```
FROM s1 AND x = y
OUT o1
```

then when -/o1 is observed, it verifies that the STATE variable was at s1 and the x variable was equal to y. When -/o1 is not observed, then either STATE was not at s1 or x was not equal to y. Hence, the absence of a spontaneous transition output is equally significant during testing.

In summary, the presence of a spontaneous transition output verifies an N.TSS identical to the spontaneous transition C.TSS. Its absence verifies an N.TSS that is different from its C.TSS.

5.3.2 Testing the EFSM in the COTP

As an example, the eUIOv-method is applied to the EFSM table in Table 5.1 for the Class 0 Transport Protocol (COTP). The EFSM had to be completed with the Completeness Assumption to generate a set of UIOSs in this case. The Completeness Assumption used in the example is that unspecified inputs are ignored and the IUT remains in its current state. The additional transitions arising from the assumption are not tested in here, but they would be if they had been part of the specification.

The UIOSs chosen for the states in the EFSM are as follows:

UIOS(IDLE) = tcreq(qts.req = ok)/cr

UIOS(WFCC) = dr/ndreq,tdind

UIOS(WFTR) = tdreq/dr

UIOS(DATA) = tdreq/ndreq

This is done by selecting I/O sequences that are unique to the states from the first four columns of the table. These columns denote TSSs determined solely by the STATE variables; hence, their transitions are not e.transitions. Since the reset feature does not exist in the Class 0 Transport Protocol, each of the following I/O sequences takes a state back to the initial IDLE state:

postamble(WFCC) = dr/ndreq,tdind

postamble(WFTR) = tdreq/dr

postamble(DATA) = tdreq/ndreq

The following I/O sequences are chosen as preambles to take the EFSM from its initial IDLE state to each of the other states. No special requirement exists for the preambles since the UIOSs are themselves executable.

preamble(WFCC) = tcreq(qts.req = ok)/cr

preamble(WFTR) = cr(option = ok)/tcind

preamble(DATA) = tcreq(qts.req = ok)/cr.cc/tccon

The following test subsequences verify the postambles to ensure they do end at the initial IDLE state. The procedure can be shortened by verifying only those postambles that are actually used.

tcreq(qts.req=ok)/cr.dr/ndreq,tdind.tcreq(qts.req=ok)
/cr.dr/ndreq,tdind (for WFCC)

cr(option=ok)/tcind.tdreq/dr.tcreq(qts.req=ok)/cr.
dr/ndreq,tdind (for WFTR)

tcreq(qts.req=ok)/cr.cc/tccon.tdreq/ndreq.
tcreq(qts.req=ok)/cr.dr/ndreq,tdind (for DATA)

The following test subsequences constitute the Uv and ~Uv procedures for the EFSM.

Uv: tcreq(qts.req=ok)/cr dr/ndreq,tdind

tcreq(qts.req=ok)/cr.dr/ndreq,tdind

cr(option=ok)/tcind.tdreq/dr

tcreq(qts.req=ok)/cr.cc/tccon.tdreq/ndreq

~Uv: dr/-

tdreq/-

tcreq(qts.req=ok)/cr.tcreq/-.dr/ndreq,tdind

tcreq(qts.req=ok)/cr.tdreq/-.dr/ndreq,tdind

cr(option=ok)/tcind.tcreq/-.tdreq/dr

cr(option=ok)/tcind.dr/-.tdreq/dr

tcreq(qts.req=ok)/cr.cc/tccon.tcreq/-.tdreq/ndreq

tcreq(qts.req=ok)/cr.cc/tccon.dr/-.tdreq/ndreq

Note that since WFTR and DATA both have an UIOS with tdreq as input, the ~Uv procedure is shortened. IDLE and WFCC testing for tdreq/- verifies the absences of both UIOSs. WFTR testing for tdreq/dr in Uv verifies the absence of tdreq/ndreq, the UIOS for DATA. DATA testing for tdreq/ndreq in Uv verifies the absence of tdreq/dr, the UIOS for WFTR. Duplicated test subsequences can be eliminated to optimize the final test sequence.

The next procedure in the eUIOv-method is Tt; it tests and verifies all transitions. With the exception of e.transitions t11 and t13, all the transitions or table entries are tested in this procedure. The following test subsequences are derived for this procedure.

Tt: tcreq(qts.req=ok)/cr.dr/ndreq,tdind

tcreq(qts.req<>ok)/tdind.tcreq(qts.req=ok)/cr.

dr/ndreq,tdind

cr(option<>ok)/dr.tcreq(qts.req=ok)/cr.dr/ndreq,tdind

cr(option=ok)/tcind.tdreq/dr

tcreq(qts.req=ok)/cr.dr/ndreq,tdind.

tcreq(qts.req=ok)/cr.dr/ndreq,tdind

tcreq(qts.req=ok)/cr.cc/tccon.tdreq/ndreq

cr(option=ok)/tcind.tcreq(qts.req>qts.estimate)/

dr,tdind.tcreq(qts.req=ok)/cr.dr/ndreq,tdind

cr(option=ok)/tcind.tdreq/dr.tcreq(qts.req=ok)/cr.

dr/ndreq,tdind

cr(option=ok)/tcind.tcres(qts.req<=qts.estimate)/

cc.tdreq/ndreq.dr/ndreq,tdind

tcreq(qts.req=ok)/cr.cc/tccon.tdreq/ndreq.tcreq(qts.req

=ok)/cr.dr/ndreq,tdind

tcreq(qts.req=ok)cr.cc/tccon.ndind/tdind.tcreq(qts.req

=ok)/cr.dr/ndreq,tdind

tcreq(qts.req=ok)cr.cc/tccon.nrind/tdind.tcreq(qts.req

=ok)/cr.dr/ndreq,tdind

tcreq(qts.req=ok)/cr.cc/tccon.tdatr/-.-/dt.tdreq/ndreq

tcreq(qts.req=ok)/cr.cc/tccon.dt/-.-/tdati.tdreq/ndreq

Note that the last two subsequences contain the spontaneous transitions -/dt and -/tdati. These are included because their TSSs are enabled following tdatr/- and dt/- respectively. Their outputs should thus be observed during

testing; hence, they are included in the final sequence. This implies tdatr/- and dt/- do not require UIOS(DATA) to verify their final states. Their final states are automatically verified by the presences of -/dt and -/tdati. This type of spontaneous transitions can be directly appended to their enabling transitions to denote their lack of inputs and when they are enabled. In this example, the EFSM table entry for tdatr/- can be changed to tdatr/-.-/dt; dt/- can be changed to dt/-.-/tdati. The columns and rows for -/dt and -/tdati can be removed. The row where tdatr/-.-/dt appears can be relabelled as data,out.buffer=0; where dt/-.-/tdati appears can be relabelled as data,in.buffer=0. Concatenating the spontaneous transitions removes all the e.transitions in this case.

The eTt procedure is not applicable here since the only e.transitions are spontaneous transitions. However, if they had not been spontaneous transitions, the procedure to form their test subsequences would be as follows. Extract the e.transition, in this example, take t11 or -/dt. According to its column label, its starting TSS is at DATA and out.buffer <> 0. Travel down the rows and find the one whose label denotes a final TSS identical to that. The search would stop at DATA and out.buffer = tdatr.in.tsdu.fragment, denoted as DATA, o.b.<>0 in the

table. Any entry at this row is an enabling condition for t11. There is only one entry in this example, namely transition t10 or tdatr/-. Hence, tdatr/- must precede -/dt to enable the latter. To test -/dt, a preamble must be formed which begins at IDLE and ends with tdatr/-. A partial preamble tcreq(qts.req=ok)/cr.cc/tccon takes the EFSM from idle to DATA, then tdatr/- is concatenated to it to form the complete preamble. The following test subsequences would be used to test -/dt:

eTt: (1)tcreq(qts.req=ok)/cr.cc/tccon.tdatr/-.tdreq/ndreq

(2)tcreq(qts.req=ok)/cr.cc/tccon.tdatr/-.-/dt.

tdreq/ndreq

where subsequence (1) first verifies the preamble to ensure it ends at the correct state, thus verifying the starting state of the e.transition to be tested. Subsequence (2) then tests the e.transition, -/dt, and verifies its final state with tdreq/ndreq. Since -/dt is really a spontaneous transition, subsequence (1) is not possible. Subsequence (2) is identical to that for testing the transition tdatr/-, hence, it is already included.

The last procedure is msvv to verify the starting and final TSSs at each transition. The only two starting TSSs that involve minor state variables are DATA AND o.b.<>0 and DATA AND i.b.<>0. However, since they enable spontaneous

transitions, the effects of these predicates have already been verified in the Tt procedure when the outputs from these transitions are observed. The final TSSs that have to be verified here are identified by those transitions whose row labels include defs of minor state variables. In the example, these are transitions t3, t5, t7, t10, t11, t12 and t13. It should be noted that while an use must be preceded by a def of the variable, not every def will necessarily be used. These defs can easily be identified by comparing the column labels with the row labels. For instance, the defs in.buffer = 0 and out.buffer = 0 do not appear at the column labels. This implies these two defs are never used. However, since there exists two spontaneous transitions which are enabled when state is at DATA and in.buffer or out.buffer are <> 0, these defs can implicitly be verified by observing that those two spontaneous transitions do not occur! This also verifies that the predicates using i.b. and o.b. are correct. Since the domains of these predicates are either 0 or not 0, they do not require the boundary-interior value criterion, but the criterion can be used to select very large and very small values to verify the domains when they are not at 0. The other defs to be verified include qts.estimate = ..., out.buffer = tdatr.in.tsdu.fragment (o.b.<>0) and in.buffer =

dt.in.user.data (i.b.<>0). For qts.estimate = ..., transition t3 or cr(option=ok)/tcind enables it. Hence, the column labels can be searched for a p-use of qts.estimate. None is found in the example, however, for state WFTR, transition t8 requires its input parameter qts.req to be > qts.estimate, hence, t8 can be used to verify the def in t3. For the second qts.estimate = ... defined in the fourth row, transition t5 enables it. A search of the column labels again turns up transition t8. Hence, if there exists a path that is free of a def of qts.estimate, then t8 can also be used to verify the def in t5. From t5, the only transitions that would exit the state DATA are t14, t15 and t16 which all lead to IDLE. From IDLE, transition t1 leads to WFCC, but transitions from WFCC either takes the EFSM back to IDLE or to DATA again, the latter introducing a new def of qts.estimate. The other path from IDLE that does not loop back to IDLE is via t3 which also enables a new def of qts.estimate. Hence, it can be concluded that there is no def-free path that would bring t5 to t8, or to any transition generated from WFTR which contains a p-use of qts.estimate; hence, the def of qts.estimate could never lead to the p-use of it in t8, or in the other transition from WFTR, t7. The remaining defs to be verified are o.b. <> 0 and i.b. <> 0. For both of them, the only starting

TSSs that make use of them belong to spontaneous transitions. Hence, their verifications are automatic and implicit when those spontaneous transitions are enabled and observed. Hence, for the msvv procedure, there is only one test subsequence.

```
msvv: cr(option=ok)/tcind.tcres(qts.req>qts.estimate)/
        dr,tdind
```

The minor state variable qts.estimate used in the predicate (qts.req > qts.estimate) in t8 can be made more visible by changing its starting TSS from WFTR to "WFTR, qts.estimate def'd". However, this is generally not necessary as minor state variables are typically initialized to some initial values at the beginning of the specification.

The final unoptimized test sequence for the EFSM of the Class 0 Transport Protocol has 98 test inputs. If duplicate test subsequences are removed, the optimized test sequence has 76 test inputs.

5.4 ESTELLE DATA FLOW TESTING

This section discusses how the data flow portion of an IUT generated according to its Estelle specification can be tested.

In testing the data flow of a protocol based on its Estelle specification, the variables and parameters that appear in the data flow portion of the rNFS are tracked from the moment they are defined until they are used either for computational purposes or as predicates. Verification for the defs, the effects of the uses as well as the correctness of p-uses are applied whenever possible as mentioned in the previous chapter.

Enabling conditions in spontaneous transitions that involve minor state variables again may aid in the process of verification in the same way they do in the eUIOv-method; that is, their absences also verify those minor state variables whose assigned values are different from those of their C.TSSs to be indeed different.

The EFSM table used in the eUIOv-method is augmented with enabling conditions for the data flow transitions to aid in determining executable flow paths between two selected rNFS transitions. If the data flow portion in a rNFS transition is preceded with a PROVIDED clause involving minor state variables, then their p-uses and defs would be extracted to form new TSSs in the EFSM table. New columns or rows may thus be added in the augmented table along with new table entries. Those PROVIDED clauses that involve input parameters would be indicated in the appropriate table

entries to indicate that specific values are required at the time of input during testing.

One of the advantages of separating the EFSM transitions from the data flow transitions in an rNFS surfaces here: during EFSM testing, replicated transitions due to data flow paths are not unnecessarily tested twice; whereas, if the NFS were used, EFSM testing would contain redundant test subsequences.

5.4.1 Data Flow Testing for the COTP

This section provides an example of data flow testing using the rNFS of the Class 0 Transport Protocol.

The EFSM table shown in Table 5.1 is first augmented with the necessary data flow portions to form the table shown in Table 5.2. Note that in the rNFS, there are two data flow paths in each of transition t3, t5 and t6. However, since their PROVIDED clauses involve only input parameters, no rows or columns needed to be added in.

The next step is to identify and extract all the transitions that span inter-transitional data flow paths (def-itDFPs). Each transition in the rNFS is examined one at a time. To begin with, transition t1 does not have any def-itDFP. All the definitions in that transition are used there; hence, exercising the transition alone exercises as

well as verifies all the defs in the transition since the usages are for defining output parameters. The same goes for transitions t2, t4, t14, t15 and t16. In transitions t10 and t12, recall that spontaneous transitions t11 and t13 may be appended to them because their execution indicate t11 and t13 would also be executed. As a result, the def-itDFPs in t10 and t12 are "automatic" and have already been included in the Tt procedure during EFSM testing. Hence, these two transitions do not have to be considered here again.

C.TSS / F.TSS	IDLE	WFCC	WFTR	DATA	DATA o.b.<>0	DATA i.b.<>0
WFCC	tcreq (qr=ok) /cr					
IDLE	tcreq (qr<>ok) /tdind cr (op<>ok) /dr	dr (dr=ui) /ndreq, tdind dr (dr<>ui) /ndreq, tdind	tcres (qr>qe) /dr, tdind tdreq/ dr	tdreq /ndreq ndind /tdind nrind /tdind		
WFTR q.e.=...	cr (op=ok, mts<>=0) /tcind cr (op=ok, mts=0) /tcind					
DATA i.b.=0 o.b.=0 q.e.=...		cc (mts<>0) /tccon cc (mts=0) /tccon				
DATA i.b.=0 o.b=0			tcres (qr<=qe) /cc			
DATA o.b.<>0				tdatr /-		
DATA o.b.=0					-/dt	
DATA i.b.<>0				dt/-		
DATA i.b.=0						-/ tdati

Table 5.2 Augmented EFSM table for Class 0 Transport Protocol.

Each of the remaining transitions in the rNFS contains def-itDFPs: t3a, t3b, t5a, t5b and t7. The def statements that span the itDFPs in these transitions are as follows.

```
t3a: remote.refer := cr.in.source.ref
     tpdu.size := cr.in.max.tpdu.size;
     (cr.in.max.tpdu.size <> nil)
t3b: remote.refer := cr.in.source.ref
     tpdu.size := ...;
     (cr.in.max.tpdu.size = nil)
t5a: in.buffer := nil;
     out.buffer := nil;
     tpdu.size := ...;
     {out.buffer.set.max.size (tpdu.size)}
     (cc.in.max.tpdu.size <> nil)
t5b: in.buffer := nil;
     out.buffer := nil;
     tpdu.size := ...;
     {out.buffer.set.max.size (tpdu.size)}
     (cc.in.max.tpdu.size = nil)
t7:  in.buffer := nil;
     out.buffer := nil;
```

Transitions t7, t8 and t9 contain usages of variables or parameters whose definitions do not reside within the same transitions. The usages are listed as follows:

```
t7:  cc.out.dest.refer := remote.refer
     cc.out.calling.t.addr := calling.t.addr;
     cc.out.called.t.addr := called.t.addr;
     cc.out.max.tpdu.size := tpdu.size;
     out.buffer.set.max.size(tpdu.size)
t8:  dr.out.dest.refer := remote.refer;
t9:  dr.out.dest.refer := remote.refer;
t6a: {tdind.out.ts.user.reason := dr.in.add.clr.reason;}
     (dr.in.disc.reason = "user.init")
t6b: (dr.in.disc.reason <> "user.init")
```

In transition t6a and t6b, although both neither span def-itDFPs nor require use-itDFPs, they contain p-uses of the variable dr.in.disc.reason. The effects of these p-uses can be verified by observing that when the parameter is equal to "user.init," an additional output parameter exists for the output primitive tdind. Both transitions t6a and t6b must be exercised to test and verify the p-uses within. They are thus included in the above list. The correctness of these p-uses are automatically verified when both transitions t6a and t6b are exercised during testing since the p-uses are completely specified. Domain testing using boundary-interior values is not applicable here.

For each of the variables that have been defined but not used, a def-free data flow path (def-itDFP) has to be

generated to connect it to its usage and, if possible, to extend it to a transition that can verify its correctness. Similarly for the uses of those variables whose definitions do not reside in the same transition. A def-free path has to connect a def of each variable to its usage (use-itDFP) and then, if possible, to a transition which can verify its correctness. The following lists the transitions containing the defs and uses of each variable listed above. Those variables defined without a use in the same transition are listed with "def" first; the "use" list that follows is extracted from all the transitions in the rNFS that contain uses of the variable. Those variables used without a def in the same transition are listed with "use" first; the "def" list that follows is also extracted from the entire rNFS. The "d" in the notation (d,u) indicates that the usage is directly preceded by a def of the variable in the same transition; hence, no def-free path exists for any def of that variable occurring in another transition in order to reach that use. The "u" in the notation indicates that it is an unobservable use.

```
remote.refer:  def - t3a, t3b
               use - t7, t8, t9
tpdu.size   :  def - t3a, t3b, t5a, t5b
               use - t5a(d,u), t5b(d,u), t7
```

```
in.buffer    :  def - t5a, t5b, t7
                use - t12.t13(d)
out.buffer   :  def - t5a, t5b, t7
                use - t10.t11(d)
remote.refer    : use - t7, t8 t9
                : def - t3a, t3b
calling.t.addr : use - t7
                  def - t3a,t3b
called.t.addr   : use - t7
                  def - t3a,t3b
tpdu.size       : use - t7
                  def - t3a, t3b, t5a, t5b
```

For the predicates, they are not included in the above list but they are extracted and the transitions that they enable are compared and the differences are listed as follows. By observing the differences, when possible, the p-uses can be verified.

```
cr.in.max.tpdu.size : <> nil ->
  tpdu.size := cr.in.max.tpdu.size
                    : = nil ->
  tpdu.size := ...;
cc.in.max.tpdu.size : <> nil ->
  tpdu.size := cc.in.max.tpdu.size
                    : = nil ->
```

```
    tpdu.size := ...;
  dr.in.disc.reason : = "user.init" ->
    tdind.out.ts.user.reason := dr.in.add.clr.reason
                    : <> "user.init" ->
    ~(tdind.out.ts.user.reason)
```

As can be seen above, the last p-use can be verified externally while the first two are verifiable only if the def of the variable "tpdu.size" can be verified externally.

For each of the listed variables, a def-free path has to be formed to bring its def to a use. Note that for the variable remote.refer, since its unverified usages appear only in t7, t8 and t9 and its unverified defs appear in only t3a and t3b, only three def-free paths are required connecting t3a to t7, t3b to t8 and t3a to t9 in order to verify all the defs and the uses of that variable. The reason is because a use that is used to verify a def during testing is itself verified in the process! The following def-free paths are formed, based on the augmented EFSM table, for each of the variables listed above.

```
remote.refer : (def- & use-itDFP) t3a.t7
               (def- & use-itDFP) t3b.t8
               (use-itDFP) t3a.t9
tpdu.size : (def- & use-itDFP) t3a.t7
            (def- & use-itDFP) t3b.t7
```

calling.t.addr : (use-itDFP) t3a.t7

called.t.addr : (use-itDFP) t3a.t7

For the tpdu.size variable, its defs in transitions t5a and t5b cannot reach the use in t7 without being re-defined first; hence, the defs in these transitions cannot be externally observed and verified. This also implies the effect of the p-uses of the input parameter cc.in.max.tpdu.size at transitions t5a and t5b also cannot be externally observed and verified. Nevertheless, since the variable tpdu.size. is used in the same transition that it is defined in, its use can still be exercised during testing by simply executing the transition. The converse is true for the tpdu.size variable defined in transitions t3a and t3b. These defs can be defined by transition t7. As a result, the p-uses of cr.in.max.tpdu.size in these transitions are also verified.

The defs of the variables in.buffer and out.buffer are not used; hence, they cannot be verified. However, since their defs assign them values that do not satisfy the enabling conditions for the spontaneous transitions, the absences of the outputs produced by these spontaneous transitions during testing indicate the defs are correct.

Whenever necessary, the def-itDFPs are selected to be identical to the use-itDFPs so that the final test sequence

may be reduced. The resulting itDFPs to be verified in the data flow testing procedure are t3a.t7, t3b.t8, t3a.t9 and t3b.t7; that is,

cr(option=ok, max.tpdu.size<>nil)/tcind.
tcres(qts.req<=qts.estimate)/cc (def- & use-itDFP)
cr(option=ok, max.tpdu.size=nil)/tcind.
tcres(qts.req>qts.estimate)/dr, tdind
(def- & use-itDFP)
cr(option=ok, max.tpdu.size<>nil)/tcind.
tdreq/dr (use-itDFP)
cr(option=ok, max.tpdu.size=nil)/tcind.
tcres(qts.req<=qts.estimate)/cc (def- & use-itDFP)

For the def-itDFPs, their data flow test subsequences are formed by using previously used preambles to bring the EFSM to each of their first transitions, then a postamble is used to bring the EFSM back to IDLE each time. If any of the itDFPs includes transitions that are e.transitions, a special preamble need to be formed to ensure executability. This preamble, if not used before in the EFSM testing procedure, should have its tail state verified prior to its use. This is to ensure that the def-itDFP indeed originated at the correct transition.

For the use-itDFPs, their test subsequences are formed as follows. For each use-itDFP, the EFSM is brought to its

first transition again using a previously used preamble whenever possible, then the use-itDFP is concatenated to it. The EFSM is then brought back to the IDLE state using a postamble. The path formed by the preamble and the use-itDFP, excluding the portion that begins at the transition housing the usage to be examined, must also have its tail state verified to ensure the use examined indeed resides at the correct transition.

The following data flow test subsequences are formed from the above four itDFPs.

```
DFTP: cr(option=ok, max.tpdu.size<>nil)/tcind.
        tcres(qts.req<=qts.estimate)/cc.tdreq/ndreq
      cr(option=ok, max.tpdu.size<>nil)/tcind.
        tcres(qts.req<=qts.estimate)/cc.tdreq/ndreq
      cr(option=ok, max.tpdu.size=nil)/tcind.
        tcres(qts.req>qts.estimate)/dr, tdind
      cr(option=ok, max.tpdu.size=nil)/tcind.
        tcres(qts.req>qts.estimate)/dr, tdind
      cr(option=ok, max.tpdu.size<>nil)/tcind.
        tdreq/dr
      cr(option=ok, max.tpdu.size=nil)/tcind.
        tcres(qts.req<=qts.estimate)/cc.tdreq/ndreq
      cr(option=ok, max.tpdu.size=nil)/tcind.
        tcres(qts.req<=qts.estimate)/cc.tdreq/ndreq
```

Since all the def-itDFPs begin at IDLE, a preamble is not required in this example. Since the first transition at each use-itDFP has already been used as a preamble from IDLE to WFTR in testing the EFSM portion, their tail states need not be verified again. Some of the sequences are duplicated because they represent a def-itDFP as well as a use-itDFP. All the paths end either at IDLE or DATA. For the latter ones, the postamble tdreq/ndreq is used to bring the EFSM from DATA back to IDLE. Added to the above test subsequences are the following two subsequences for testing transitions t6a and t6b.

```
tcreq(qts.req=ok, disc.reason="user.init")/cr.
  dr/ndreq,tdind
tcreq(qts.req=ok, disc.reason<>"user.init")/cr.
  dr/ndreq,tdind
```

Length of the test sequence for the data flow testing procedure is 22. After optimization in which duplicate subsequences are removed, the length is reduced to 14 test inputs. If these subsequences were compared to those in the EFSM testing procedure and the duplicates were removed, the number of test inputs in this procedure would be further reduced to 7.

5.5 CHAPTER SUMMARY

This chapter shows how the hybrid technique can be applied to the testing of protocols implemented according to their Estelle specifications.

In the testing procedure, the spontaneous transitions are appended to those input transitions that enable them. This is possible since their spontaneity still require that their starting TSSs be satisfied, which in turn are enabled by the appropriate input transitions to which they are appended. If the spontaneous transitions do not have C.TSSs that can be enabled by any input transition, then they should not be appended to any input transition.

For the Class 0 Transport Protocol example, the hybrid technique would generate an optimized test sequence of length 83, where 76 test inputs are for testing the EFSM structure and 7 are for testing the flow of data. The length of the final test sequence varies according to the UIOSs chosen in the EFSM testing procedure.

6 EVALUATION AND COMPARISON OF THE HYBRID TECHNIQUE

The hybrid technique is a conformance test sequence generation method that is applicable to testing protocols implemented according to their Estelle specifications. The principles behind the method arise from lessons learnt from FSM testing, where merely exercising the features in an IUT according to its specification is not enough; whenever possible, those features that are not externally observable should be verified by other means. The interactive nature of protocols lend themselves well to this method of testing.

This chapter provides a general evaluation of the hybrid technique as well as a general comparison of this method to other notable test sequence generation techniques.

6.1 EVALUATION

6.1.1 Fault Coverage

Since the hybrid technique is a combination of two testing methods, its fault detection capability or fault coverage is also a combination of the fault coverages produced by the two methods.

Fault coverage of the hybrid technique can be expressed in terms of an erroneous Estelle specification. Provided no extra STATE and statement exists in the erroneous Estelle

specification, the hybrid technique is capable of detecting all STATE errors, I/O primitive errors, errors in those def and usage statements whose effects are externally observable and verifiable, and possibly errors in those def and usage statements whose effects are not verifiable or they contain mathematical expressions. These errors are detectable in all erroneous IUTs that can be described by the erroneous Estelle specification.

6.1.2 Executability

In the hybrid approach, since enabling conditions are taken into consideration in the process of generating test sequences, the resulting test sequences are all executable. Ensuring executability implies some test data selection has to be performed during the sequence generation procedure. The disadvantage is that complexity is added to the process of sequence generation and at times, computation may be required. Executability requires that the presentation of the resulting sequence of test inputs be augmented by any special input value required. Test data selection can again be based on the boundary-interior value criterion or done by choosing the most commonly used values in a real implementation.

6.1.3 Applicability

The test sequences generated by the hybrid technique are tailored for detecting errors in an IUT which can be described by an erroneous Estelle specification. The test sequences are especially suited to IUTs implemented according to Estelle specifications, but they can also be applied to any IUT independent of the type of specification from which it is generated.

For IUTs that are not implemented according to Estelle specifications, the test sequences are still capable of detecting errors in the data flow of the protocol. These include flow paths that would exist within a given Estelle transition as well as those that would be across sequences of Estelle transitions. What the test sequences may not detect is an erroneous sequence of I/Os executed in the IUT. The reason is as follows. Recall that in a protocol, I/Os not only assume certain values (data flow), they also assume certain sequences (control flow). If the control structure of an IUT had been implemented as a tree, then an erroneous branch of the tree may not be detected by sequences generated by the hybrid technique. The hybrid technique takes advantage of the EFSM control structure of a protocol and applies FSM-based testing techniques to check this control structure. Checking the control structure of such

an IUT thus requires less work; however, it does require knowing that the structure is modelled by an EFSM.

An alternative to testing IUTs whose control structures are unknown would be to span a tree according to the FSM in the Estelle specification and test the IUT for each branch in the tree. However, when loops are encountered or if the FSM is of considerable size, the resulting test sequence could be horrendously long. Hence, knowing that the control structure is modelled by an FSM and taking advantage of it can considerably reduce a test sequence without reducing the fault coverage producible.

6.1.4 Test Data Selection

The test sequences generated simply indicate the sequences of I/Os that should be considered during testing. Any special value required for a test input is also indicated in a test sequence generated by the hybrid technique. During test data selection, a test subsequence may need to be duplicated since more than one test value may be used in order to test the correctness of variable usages, rather than their effects. The selection of input values could either be based on boundary-interior values or a chosen set of most frequently used values in a real implementation. If the latter is not available, the former

can be used; else, the latter should take precedence. For p-uses, since it can generally be assumed that all possibilities are completely specified, each possibility would already be included in the test sequence and need not be fussed over during data selection. This also applies to conditions with the keyword AND. If "a AND b" were specified, then, "NOT b" and "NOT a" would also be specified and test data selection does not have to take these alternatives into consideration.

6.1.5 Testability

A protocol can be made more easily testable with the addition of "status" messages [Aho88, Sari84]. The "read state" message is an input which requires the IUT to report the STATE that it is currently at. The addition of this feature implies UIOSs will always exist, each of which is of length one. In addition, if synchronization were a problem [Sari84], then these status messages can be used to aid synchronization in the testing of OSI protocols by requiring that the IUT report its current state to both upper and lower testers as a means of synchronizing the two.

Another way of increasing testability is by completing an incompletely specified specification [Sari84] with one of the Completeness Assumptions. A "DEFAULT" feature can be

added to Estelle similar to the "PROVIDED otherwise" feature to specify a Completeness Assumption. For instance, if "otherwise" is used to group all the unspecified input primitives together, and "ANY_STATE" groups all the states together, then

```
DEFAULT:
  WHEN otherwise
  FROM ANY_STATE
  TO SAME
  BEGIN
  END;
```

specifies the assumption that any input not specified for a state is ignored and the machine remains at its current state and no output is generated. On the other hand,

```
DEFAULT:
  WHEN otherwise
  FROM ANY_STATE
  TO ERROR
  BEGIN
    OUTPUT input_error();
  END;
```

the above default sends the protocol to the ERROR state after it outputs a message complaining of an input error [Sari84] upon reception of an unspecified input. At the

ERROR state, all further inputs would be ignored until a disconnect or reset input is received, at which time the protocol will return to the initial IDLE state. These DEFAULT transitions are defaults and should be specified as the last transitions in an Estelle specification, similar to where "PROVIDED otherwise" is situated in a group of related PROVIDED clauses. If any modification to the specification is done so that an unspecified input becomes a specified one, no modification would be required at the DEFAULT transitions.

6.2 COMPARISON

The hybrid technique is compared to the FSM testing techniques and those developed by Ural [Ural88] and Sarikaya [Sari87] in this section. These methods are compared based on their applications to testing IUTs implemented according to Estelle specifications.

In comparing the hybrid technique with FSM testing techniques, the hybrid technique is obviously more comprehensive in terms of fault coverage because FSM testing techniques examine only the control structure of protocols while the hybrid technique also examines its data flow. In addition, the test sequences generated by the FSM techniques

may not be executable while those generated by the hybrid technique are.

In comparing the hybrid technique with that of Ural, since the data flow portion of the protocol is also tested in the hybrid technique in addition to the control structure of the IUT, the hybrid technique also achieves better coverage than that produced by Ural. An example is as follows using the Class 0 Transport Protocol. If the control structure of an erroneous implementation of the Transport Protocol were modelled by an EFSM with only three STATEs, for instance, the WFTR and WFCC states were erroneously merged into a single STATE; then Ural's approach would not have been able to detect this error. The reason is because the erroneous EFSM spans a tree part of which contains the tree spanned by the correct EFSM. As a result, all the data flow paths in a correct implementation of the protocol would also be present in the erroneous IUT. Since Ural's approach examines only the data flow paths, it would not have been able to detect the state error, but the hybrid approach would. Merging STATEs WFCC and WFTR is an error because, among the many extra branches the erroneous EFSM spans, one of them implies the IO sequence cr/tcind.cc/tccon is valid which is obviously incorrect. If this error were not detected, the IUT could go to its data transfer state

upon receiving a connect request and a connect confirmation one after another from the peer protocol without any response from its user. The merged STATE is thus an error and should be captured.

In terms of executability, since the hybrid technique takes executability into consideration during its sequence generation process, the sequences it generates would be executable while those generated by Ural may not be.

In comparing the hybrid technique to that of Sarikaya, the same erroneous Transport Protocol example can be used. Sarikaya's method again would not detect the underlying erroneous EFSM because it spans a tree that includes the one spanned by the correct EFSM. As a result, all the subtours present in the correct EFSM would also be present in the erroneous EFSM; and all the data flows present in the corresponding correct IUT would also be present in the erroneous IUT. Hence, the hybrid technique provides better coverage in this sense.

In terms of test derivation complexity, the hybrid technique is more complex than those for testing FSMs and that of Ural because it has to do more than any one of them does and, as well, it has to take executability into consideration. However, in comparison to the approach taken by Sarikaya, the hybrid technique seems simpler because it

does not have to generate the complex data flow graphs. The technique, however, does require taking values into consideration to ensure executability and test data may have to be selected during the sequence generation procedure. In addition, in the verification of p-uses, differences among the effects of the p-uses have to be identified so that they may be used to verify the p-uses. Hence, the hybrid technique demands that more attention be given to the semantics of the specification than Sarikaya's does. In this sense, the hybrid technique is more complex than that of Sarikaya.

7 CONCLUSIONS

This section brings this thesis to a close with a summary of its achievements and a discussion of possible future work in the area of conformance testing.

7.1 THESIS SUMMARY

Contributions of this thesis lie in three main areas: the development of the UIOv-method, the developments of the eUIOv-method and the data flow testing procedure, the formation of the hybrid technique and its application to Estelle.

This thesis began by examining notable FSM testing techniques that can be applied to the conformance testing of protocols modelled by FSMs. It found, contrary to a previous claim [Sidh89], that the most recently developed UIO-method (U-method) does not provide full fault coverage in the testing of completely specified protocols. It proposed adding a verification procedure, ~Uv, to the U-method so that the revised U-method, or UIOv-method, achieves full fault coverage in strong conformance tests. In weak conformance tests, it is also capable of detecting faulty IUTs provided the number of states in these IUTs do not exceed that which is in the specified FSM. The D-method

and the W-method were found to be special cases of the UIOv-method.

This thesis also proposed the concept of Unique Test Sequences (UTSs) to capture the property a test sequence should have in order to achieve full fault coverage. An UTS is a sequence of I/Os that is unique to the FSM from which it is generated so that any IUT that passes a test using an UTS contains a skeleton FSM that is identical to the specified FSM. As a result, if the number of states in the IUT does not exceed that which is in the specification FSM, then all faulty IUTs can be captured by tests employing UTSs. The UIOv-method generates such tests.

This thesis extended the UIOv-method to an eUIOv-method to generate test sequences for testing protocols that can be modelled by EFSMs. The generated test sequences are executable since the total state of the system is taken into consideration during the generation process. The total system state is verified the same way a state is in an FSM. Hence, all elements in an EFSM transition are exercised and verified, whenever possible, during testing.

This thesis developed a data flow testing procedure based on static data flow analysis and lessons learnt from FSM testing. The procedure extends the basic exercising of def-use data flow paths to verifying, whenever possible,

that each def is correct, each use is correct and each p-use is within the correct domain.

This thesis refined and reformatted the normal form of an Estelle specification so that its transitions are in true normal form and its new format, the rNFS, separately brings out the underlying EFSM and data flow from an Estelle specification. During EFSM testing, redundant EFSM transitions that appear in an NFS are eliminated.

The data flow testing procedure is combined with the eUIOv-method to form a hybrid technique that is applicable to generating conformance test sequences for protocols implemented according to their Estelle specifications. The resulting test sequences detect faulty IUTs that either have erroneous EFSM control structures or erroneous IO operations. The fault coverage achieved by the hybrid technique is a combination of those achieved by the eUIOv-method and the data flow testing procedure.

A correct sequence of I/Os is equally important in a protocol as its I/O values are. Testing of a protocol should thus consider both of these aspects. FSMs can model the control structure of some protocols effectively. Since these models lend themselves well to testing, this should be taken advantage of when testing such protocols so that its control structure as well as its data flow aspect can both

be examined during testing. The hybrid technique was developed to do just that for this type of protocols.

The hybrid technique was tailored for protocols with underlying EFSM control structures, this technique is thus capable of detecting faulty IUTs with erroneous EFSMs that would otherwise be missed by the techniques developed by Ural and Sarikaya. Since the hybrid technique also includes data flow testing, it is capable of detecting certain types of data flow errors in an IUT that would otherwise be missed by FSM testing techniques.

The hybrid technique is more complex than that of Ural and those for testing FSMs; however, it guarantees that the resulting test sequences are executable and it is generally less complex than that developed by Sarikaya.

Test data selection for the hybrid technique either employs a finite set of most frequently used values for the input parameters or, if unavailable, boundary-interior values would be chosen. To ensure executability, test data selection is sometimes combined with test sequence selection.

Test sequences generated by the hybrid technique are also applicable to IUTs not implemented according to their Estelle specifications. The fault coverage achieved would

be limited to faults in the I/Os; sequencing faults may or may not be captured.

7.2 FUTURE WORK

A tool is being developed in another study to implement the hybrid technique developed in this thesis so that it can be easily applied to more complex Estelle protocol specifications. Existing implementation techniques for the UIO-method and static data flow analysis can be coupled with symbolic execution to implement the hybrid technique.

This thesis has emphasized on the fault coverages achieved by the test sequences; more research is required to optimize the resulting test sequences such that their fault detection capabilities are maintained and the UTSs remain unique.

For test data selection, a list of commonly used parameter values should be compiled so that they may be used as inputs during testing.

In the testing of OSI protocols, more than one protocol layer may have been implemented as a single unit so that an IUT may be embedded within a large implementation. How such an IUT can be tested still needs to be investigated. Perhaps the Estelle specification for each of the layers can be combined in a manner similar to that of an NFS so that a

single specification exists for the multi-layer implementation. This specification may then be used to develop test sequences to test the multi-layer implementation either as a whole or one layer at a time.

BIBLIOGRAPHY

[Aho88] Aho, A.V., Dahbura, A.T., Lee, D. and Uyar, U.M., "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours," *Proc. Eighth International Symposium on Protocol Specification, Testing, and Verification*, Atlantic City, N.J., 1988.

[Chan89a] Chan, W.Y.L., Vuong, S. and Ito, M.R., "On Test Generation for Protocols," *Proc. Ninth International Symposium on Protocol Specification, Testing and Verification*, The Netherlands, June 1989.

[Chan89b] Chan, W.Y.L., Vuong, S.T. and Ito, M.R., "An Improved Protocol Test Generation Procedure Based on UIOs," *Proc. ACM Sigcomm '89 Symposium, Communications Architectures and Protocols*, Austin, Texas, September 1989

[Chan89c] Chan, W.Y.L., Vuong, S.T. and Ito, M.R., "The UIOv-Method For Protocol Testing," Proc. Second International Workshop on Protocol Testing, Berlin, Germany, October 1989.

[Chow78] Chow, T., "Testing Software Design Modelled by Finite State Machines," *IEEE Transactions on Software Engineering*, vol SE-4, no. 3, 1978, pp. 178-187.

[Gone70] Gonenc, G., "A Method for the Design of Fault Detection Experiments," *IEEE Transactions on Computers,* vol. C-19, no. 6, June 1970.

[Hech77] Hecht, M.S., Flow Analysis of Computer Programs, New York: North-Holland, 1977.

[ISO88] ISO/TC97/SC21/WG1/Subgroup B, "Estelle - A Formal Description Technique Based on an Extended State Transition Model," IS 9074, 1988.

[Kou87] Kou, T., "Conformance Testing of OSI Protocol: The Class 0 Transport Protocol As An Example," Master Thesis, Dept. of Computer Science, the University of British Columbia, August 1987.

[Nait81] Naito, S. and Tsunoyama, M., "Fault Detection for Sequential Machines by Transition Tours," *Proc. of IEEE Fault Tolerant Computing Conference*, 1981.

[Rapp85] Rapps, S. and Weyuker, E.J., "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 4, April 1985, pp. 367-375.

[Rayn86] Rayner, D., "Standardizing Conformance Testing for OSI," *Protocol Specification, Testing and Verification,* V, M.Diaz, Ed., North-Holland, Amsterdam, The Netherlands, 1986.

[Sabn88] Sabnani, K.K. and Dahbura, A.T., "A Protocol Test Generation Procedure," *Computer Networks*, vol. 15, no. 4, 1988, pp. 285-297.

[Sari82] Sarikaya, B. and Bochmann, G.v., "Some Experience with Test Sequence Generation for Protocols," *Protocol Specification, Testing and Verification*, C. Sunshine, Ed., North-Holland, 1982, pp. 555-567.

[Sari84] Sarikaya, B. and Bochmann, G.v., "Synchronization and Specification Issues in Protocol Testing," *IEEE Transactions on Communications*, vol. COM-32, no. 4, April 1984, pp. 389-395.

[Sari86] Sarikaya, B. and Bochmann, G.v., "Obtaining Normal Form Specifications for Protocols," *Computer Network Usage: Recent Experiences*, L. Csaba, K. Tarnay and T. Szentivanyi (Eds.,), North-Holland, 1986, pp. 601-612.

[Sari87] Sarikaya, B., Bochmann, G.v., and Cerny, E., "A Test Design Methodology for Protocol Testing," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 5, May 1987, pp. 518-531.

[Sidh88] Sidhu, D. and Leung, T., "Experience with Test Generation for Real Protocols," *Proc. ACM Sigcomm '88 Symposium, Communications Architectures & Protocols*, Stanford, CA., August 1988, pp. 257-261.

[Sidh89] Sidhu, D.P. and Leung, T.K., "Formal Methods for Protocol Testing: A Detailed Study," *IEEE Transactions on Software Engineering* vol. 15, no. 4, April 1989, pp. 413-426.

[Ural88] Ural, H., Yang, B. and Probert, R.L., "A Test Sequence Selection Method for Protocols Specified in Estelle," Technical Report, TR-88-18, The University of Ottawa, June 1988.

[Vuon88a] Vuong, S.T., Chan, R.I. and Chan, W.Y.L., "An Estelle-C Compiler for Automatic Protocol Implementation," *Proc. Eighth International Symposium on Protocol Specification, Testing, and Verification*, Atlantic City, New Jersey, USA, June 1988.

[Vuon88b] Vuong, S.T. and Chan, W.Y.L., "Validation Of The Ferry Clip Local Testing System Using An Estelle-C Compiler," *Forte '88, First International Conference on Formal Description Techniques*, the University of Stirling, September 1988.

[Weiss85] Weiser, M.D., Gannon, J.D. and McMullin, P.R., "Comparison of Structural Test Coverage Metrics," *IEEE Software*, March 1985, pp. 80-85.

[Zimm80] Zimmermann, H., "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communication*, vol. COM-28, no. 4, April 1980.

```
WHEN tcreq(to.t.addr, from t.addr, qts.req)
FROM idle
PROVIDED tcreq.in.qts.req = ok
TO wfcc
t1:  BEGIN
     local.refer := ...;
     tpdu.size := ...;
     cr.out.calling.t.addr := tcreq.in.from.t.addr;
     cr.out.called.t.addr := tcreq.in.to.t.addr;
     OUT cr(local.refer, "normal", calling.t.addr,
            called.t.addr, tpdu.size);
     END;

-----

WHEN tcreq(to.t.addr, from.t.addr, qts.req)
FROM idle
PROVIDED tcreq.in.qts.req <> ok
TO idle
t2:  BEGIN
     tdind.out.ts.disc.reason := ...;
     tdind.out.ts.user.reason := ...;
     OUT tdind(ts.disc.reason, ts.user.reason);
     END;

-----

WHEN cr(source.ref, option, calling.t.addr,
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.max.tpdu.size <> nil AND
         cr.in.option = ok
TO wftr
t3:  BEGIN
     remote.refer := cr.in.source.ref;
     tpdu.size := cr.in.max.tpdu.size;
     calling.t.addr := cr.in.calling.t.addr;
     called.t.addr := cr.in.called.t.addr;
     qts.estimate := ...;
     OUT tcind(called.t.addr, calling.t.addr,
               qts.estimate);
     END;

-----
```

```
WHEN cr(source.ref, option, calling.t.addr,
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.max.tpdu.size = nil AND
         cr.in.option = ok
TO wftr
t4:  BEGIN
     remote.refer := cr.in.source.ref;
     tpdu.size := ...;
     calling.t.addr := cr.in.calling.t.addr;
     called.t.addr := cr.in.called.t.addr;
     qts.estimate := ...;
     OUT tcind(called.t.addr, calling.t.addr,
               qts.estimate);
     END;

-----

WHEN cr(source.ref, option, calling.t.addr,
        called.t.addr, max.tpdu.size)
FROM idle
PROVIDED cr.in.option <> ok
TO idle
t5:  BEGIN
     dr.out.dest.refer := cr.in.source.ref;
     dr.out.disc.reason := ...;
     OUT dr(dest.refer, disc.reason);
     END;

-----

WHEN cc(dest.ref, source.ref,calling.t.addr,
        called.t.addr,max.tpdu.size)
FROM wfcc
PROVIDED cc.in.max.tpdu.size <> nil
TO data
t6:  BEGIN
     tpdu.size := cc.in.max.tpdu.size;
     qts.estimate := ...;
     in.buffer := nil;
     out.buffer := nil;
     out.buffer.set.max.size(tpdu.size);
     OUT tccon(qts.estimate);
     END;

-----
```

```
WHEN cc(dest.ref, source.ref, calling.t.addr,
        called.t.addr,max.tpdu.size)
FROM wfcc
PROVIDED cc.in.max.tpdu.size = nil
TO data
t7:  BEGIN
     tpdu.size := ...;
     qts.estimate := ...;
     in.buffer := nil;
     out.buffer := nil;
     out.buffer.set.max.size(tpdu.size);
     OUT tccon(qts.estimate);
     END;

-----

WHEN dr(disc.reason, add.clr.reason)
FROM wfcc
PROVIDED dr.in.disc.reason = "user.init"
TO idle
t8:  BEGIN
     ndreq.out.disc.reason := dr.in.disc.reason;
     tdind.out.ts.disc.reason := dr.in.disc.reason;
     tdind.out.ts.user.reason := dr.in.add.clr.reason;
     OUT ndreq(disc.reason);
     OUT tdind(ts.user.reason, ts.disc.reason);
     END;

-----

WHEN dr(disc.reason, add.clr.reason)
FROM wfcc
PROVIDED dr.in.disc.reason <> "user.init"
TO idle
t9:  BEGIN
     ndreq.out.disc.reason := dr.in.disc.reason;
     tdind.out.ts.disc.reason := dr.in.disc.reason;
     OUT ndreq(disc.reason);
     OUT tdind(ts.disc.reason);
     END;

-----
```

```
WHEN tcres(qts.req)
FROM wftr
PROVIDED tcres.in.qts.req <= qts.estimate
TO data
t10: BEGIN
     local.refer := ...;
     in.buffer := nil;
     out.buffer := nil;
     out.buffer.set.max.size(tpdu.size);
     OUT cc(remote.refer, local.refer, calling.t.addr,
            called.t.addr, tpdu.size);
     END;

-----

WHEN tcres(qts.req)
FROM wftr
PROVIDED tcres.in.qts.req > qts.estimate
TO idle
t11: BEGIN
     dr.out.disc.reason := ...;
     dr.out.add.clr.reason := ...;
     tdind.out.ts.disc.reason := ...;
     OUT dr(remote.refer; disc.reason, add.clr.reason);
     OUT tdind(ts.disc.reason);
     END;

------

WHEN tdreq(ts.user.reason)
FROM wftr
TO idle
t12: BEGIN
     dr.out.disc.reason := ...;
     dr.out.add.clr.reason := tdreq.in.ts.user.reason;
     OUT dr(remote.refer, disc.reason, add.clr.reason);
     END;

-----

WHEN tdatr(tsdu.fragment)
FROM data
TO data
t13: BEGIN
     insert(out.buffer, tdatr.in.tsdu.fragment);
     END;

-----
```

```
FROM data
PROVIDED out.buffer <> nil
TO data
t14: BEGIN
     remove(out.buffer, dt.out.user.data);
     OUT dt(user.data);
     END;

-----

WHEN dt(user.data)
FROM data
TO data
t15: BEGIN
     insert(in.buffer, dt.in.user.data);
     END;

-----

FROM data
PROVIDED in.buffer <> nil
TO data
t16: BEGIN
     remove(in.buffer, tdati.out.tsdu.fragment);
     OUT tdati(tsdu.fragment);
     END;

-----

WHEN tdreq(ts.user.reason)
FROM data
TO idle
t17: BEGIN
     ndreq.out.disc.reason := tdreq.in.ts.user.reason;
     OUT ndreq(disc.reason);
     END;

-----

WHEN ndind()
FROM data
TO idle
t18: BEGIN
     tdind.out.ts.disc.reason := ...;
     OUT tdind(ts.disc.reason);
     END;

-----
```

```
WHEN nrind()
FROM data
TO idle
t19: BEGIN
     tdind.out.ts.disc.reason := ...;
     OUT tdind(ts.disc.reason);
     END;
```

APPENDIX B RNFS OF CLASS 0 TRANSPORT PROTOCOL

```
t1:  WHEN tcreq(qts.req = ok)
     FROM idle
     TO wfcc
     OUT cr
     BEGIN
       local.refer := ...;
       cr.out.source.refer := local.refer;
       cr.out.option := "normal";
       tpdu.size := ...;
       cr.out.max.tpdu.size := tpdu.size;
       cr.out.calling.t.addr := tcreq.in.from.t.addr;
       cr.out.called.t.addr := tcreq.in.to.t.addr;
     END;

-----

t2:  WHEN tcreq(qts.req <> ok)
     FROM idle
     TO idle
     OUT tdind
     BEGIN
       tdind.out.ts.disc.reason := ...;
       tdind.out.ts.user.reason := ...;
     END;

-----

t3a: WHEN cr(option = ok)
     FROM idle
     TO wftr AND qts.estimate := ...;
     OUT tcind
     PROVIDED (cr.in.max.tpdu.size <> nil)
     BEGIN
       remote.refer := cr.in.source.ref;
       tpdu.size := cr.in.max.tpdu.size;
       calling.t.addr := cr.in.calling.t.addr;
       tcind.out.from.t.addr := calling.t.addr;
       called.t.addr := cr.in.called.t.addr;
       tcind.out.to.t.addr := called.t.addr;
       qts.estimate := ...;
       tcind.out.qts.pro := qts.estimate;
     END;

-----
```

```
t3b: WHEN cr(option = ok)
     FROM idle
     TO wftr AND qts.estimate := ...
     OUT tcind
     PROVIDED (cr.in.max.tpdu.size = nil)
     BEGIN
       remote.refer := cr.in.source.ref;
       tpdu.size := ...;
       calling.t.addr := cr.in.calling.t.addr;
       tcind.out.from.t.addr := calling.t.addr;
       called.t.addr := cr.in.called.t.addr;
       tcind.out.to.t.addr := called.t.addr;
       qts.estimate := ...;
       tcind.out.qts.pro := qts.estimate;
     END;

-----

t4:  WHEN cr(option <> ok)
     FROM idle
     TO idle
     OUT dr
     BEGIN
       dr.out.dest.refer := cr.in.source.ref;
       dr.out.disc.reason := ...;
     END;

-----

t5a: WHEN cc
     FROM wfcc
     TO data AND BEGIN
                   in.buffer := nil;
                   out.buffer := nil;
                   qts.estimate := ...;
                 END;
     OUT tccon
     PROVIDED (cc.in.max.tpdu.size <> nil)
     BEGIN
       qts.estimate := ...;
       tccon.out.qts.estimate := qts.estimate;
       in.buffer := nil;
       out.buffer := nil;
       tpdu.size := cc.in.max.tpdu.size;
       out.buffer.set.max.size(tpdu.size);
     END;
```

```
-----

t5b: WHEN cc
     FROM wfcc
     TO data AND BEGIN
                  in.buffer := nil;
                  out.buffer := nil;
                  qts.estimate := ...;
                END;
     OUT tccon
     PROVIDED (cc.in.max.tpdu.size = nil)
     BEGIN
       qts.estimate := ..;
       tccon.out.qts.res := qts.estimate;
       in.buffer := nil;
       out.buffer := nil;
       tpdu.size := ...;
       out.buffer.set.max.size (tpdu.size);
     END;

-----

t6a: WHEN dr
     FROM wfcc
     TO idle
     OUT ndreq, tdind
     PROVIDED (dr.in.disc.reason = "user.init")
     BEGIN
       ndreq.out.disc.reason := dr.in.disc.reason;
       tdind.out.ts.disc.reason := dr.in.disc.reason;
       tdind.out.ts.user.reason :=
         dr.in.add.clr.reason;
     END;

-----

t6b: WHEN dr
     FROM wfcc
     TO idle
     OUT ndreq, tdind
     PROVIDED (dr.in.disc.reason <> "user.init")
     BEGIN
       ndreq.out.disc.reason := dr.in.disc.reason;
       tdind.out.ts.disc.reason :=
         dr.in.disc.reason;
     END;
```

```
-----

t7:  WHEN tcres(qts.req <= qts.estimate)
     FROM wftr
     TO data AND BEGIN
                  in.buffer := nil;
                  out.buffer := nil;
                END;
     OUT cc
     BEGIN
       local.refer := ...;
       cc.out.dest.refer := remote.refer;
       cc.out.source.refer := local.refer;
       cc.out.calling.t.addr := calling.t.addr;
       cc.out.called.t.addr := called.t.addr;
       cc.out.max.tpdu.size := tpdu.size;
       in.buffer := nil;
       out.buffer := nil;
       out.buffer.set.max.size (tpdu.size);
     END;

-----

t8:  WHEN tcres(qts.req > qts.estimate)
     FROM wftr
     TO idle
     OUT dr, tdind
     BEGIN
       dr.out.dest.refer := remote.refer;
       dr.out.disc.reason := ...;
       dr.out.add.clr.reason := ...;
       tdind.out.ts.disc.reason := ...;
     END;

-----

t9:  WHEN tdreq
     FROM wftr
     TO idle
     OUT dr
     BEGIN
       dr.out.disc.reason := ...;
       dr.out.add.clr.reason := tdreq.in.user.reason;
       dr.out.dest.refer := remote.refer;
     END;

-----
```

```
t10: WHEN tdatr(tsdu.fragment)
     FROM data
     TO data AND
              insert(out.buffer,tdatr.in.tsdu.fragment)
     BEGIN
       insert(out.buffer,tdatr.in.tsdu.fragment);
     END;

-----

t11: FROM data AND out.buffer <> nil
     TO data AND remove(out.buffer,dt.out.user.data)
     OUT dt
     BEGIN
       remove(out.buffer, dt.out.user.data);
     END;

-----

t12: WHEN dt
     FROM data
     TO data AND insert(in.buffer, dt.in.user.data)
     BEGIN
       insert(in.buffer, dt.in.user.data);
     END;

-----

t13: FROM data AND in.buffer <> nil
     TO data AND
              remove(in.buffer,tdati.out.tsdu.fragment)
     OUT tdati
     BEGIN
       remove(in.buffer, tdati.out.tsdu.fragment);
     END;

-----

t14: WHEN tdreq
     FROM data
     TO idle
     OUT ndreq
     BEGIN
       ndreq.out.disc.reason:= tdreq.in.ts.user.reason;
     END;

-----
```

```
t15: WHEN ndind
     FROM data
     TO idle
     OUT tdind
     BEGIN
       tdind.out.ts.disc.reason := ...;
     END;

-----

t16: WHEN nrind
     FROM data
     TO idle
     OUT tdind
     BEGIN
       tdind.out.ts.disc.reason := ...;
     END;
```