Formal Methods for Protocol Testing: A Detailed Study

DEEPINDER P. SIDHU, SENIOR MEMBER, IEEE, AND TING-KAU LEUNG

Abstract—**A protocol standard, in general, can lead to different implementations, which necessitates the need for conformance testing of an implementation to its standard. Testing is carried out with the help of a test sequence generated from a protocol specification. This paper presents a detailed study of four formal methods (T-, U-, D-, and W-methods) for generating test sequences for protocols. Applications of these methods to NBS Class 4 Transport Protocol are discussed. This paper also presents an estimation of fault coverage of four protocol test sequences generation techniques using Monte Carlo simulation. The ability of a test sequence to decide whether a protocol implementation conforms to its specification heavily relies upon the range of faults that it can capture. Conformance is defined at two levels, namely, weak and strong conformance. This study shows that a test sequence produced by T-method has a poor fault detection capability whereas test sequences produced by U-, D- and W-methods have comparable (superior to that for T-method) fault coverage on several classes of randomly generated machines used in this study. Also, some problems with a straightforward application of the four protocol test sequence generation methods to real-world communication protocols are pointed out.**

Index Terms—**Formal description technique, formal modeling, protocol conformance testing, protocol specification, protocol standards, protocol verification, state transition model.**

I. Introduction

THE modern advances in hardware technologies has been playing a key role in the rapid development of computer communications networks and distributed processing systems. Within the last decade, several successful networks have been designed and implemented. The success of the first generation experimental networks has given impetus to the development of several public, private and commercial computer communication networks.

The computer systems attached to a network communicate with each other using a common set of rules and conventions called protocols. The ISO/OSI Reference Model defines a seven layered protocol architecture [1] for communication systems. The layering concept was used to divide the communication functions into sets which can be specified separately. This allows independent development and implementation of standards at each layer. Several organizations are working on developing protocol standards for the different layers in the ISO/OSI model using formal description techniques.

A protocol, in general, is quite complex and takes a considerable effort to implement on a system. The implementation of a protocol is generally derived from a specification standard. A protocol standard, in general, can lead to several different implementations. This calls for testing each protocol implementation for conformance to the specification of the protocol standard. The complexity of protocols necessitates the use of automated tools to provide assistance in the specification, verification, implementation, and testing of communication protocols.

A test sequence for a protocol is a sequence of input–output pairs derived from the protocol specification. These inputs are applied to an implementation under test. The implementation is assumed to be a black box with an input port and output port. The inputs to this black box are given at its input port and the outputs can be observed at its output port. The outputs generated by the implementation are then compared with the corresponding outputs in the test sequence. If they match, such a protocol implementation is said to conform to the specification. Otherwise, the implementation is assumed to be faulty.

In Section II, we briefly discuss four test sequence generation techniques (T-, U-, D-, and W-methods) and their implementation as automated software tools. Section III discusses the fault coverage of the four protocol test methods. The ability of a test sequence to decide whether a protocol implementation conforms to its specification heavily relies upon the range of faults that it can capture. Section IV demonstrates an application of the automated protocol test sequence generators to the derivation of test sequences from the specification standard of NBS Class 4 Transport Protocol (NBS TP4). In Section V, we examine and compare certain aspects of the four protocol test sequence generation methods. Section VI contains some conclusions. The Appendixes include some test sequences for NBS TP4 used in estimating the fault detection capabilities of the test methods.

II. Test Sequence Generation Techniques

The specification of a protocol standard is, in general, a detailed document describing the interfaces and mechanism of the protocol. If the description technique used in specifying a protocol standard is not formal, it is possible that this protocol specification is ambiguous in some

Manuscript received December 19, 1986; revised January 4, 1988.

D. P. Sidhu is with the Department of Computer Science, University of Maryland—BC, Baltimore, MD 21228, and the Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

T.-K. Leung is with Nixdorf Computer Limited, Hong Kong.

IEEE Log Number 8826224.

sense. The presence of ambiguities in a protocol specification can lead to different implementations of the protocol. Two such implementations may not be able to communicate with each other using that protocol.

In spite of using a formal description technique for specifying a protocol standard, it is still possible that two implementations derived from the standard are not compatible. This can result due to incorrect implementation of some aspects of the protocol. This means that there is a need for testing each protocol implementation for conformance to its specification standard [2]–[6]. Testing is carried out by using test sequences.

In this section, we briefly describe four protocol test sequence generation techniques, namely, the T-method, U-method, D-method, and W-method. All four methods assume a Mealy machine model for protocol entity specifications. A Mealy machine is a finite state machine which produces an output upon each transition. Examples are given for the application of each test sequence generation method.

In the subsequent discussion, we use M to denote "Mealy machine M" and adopt the following notation.

$M|s$ ≡ machine M at state s.
$M|s(\alpha)$ ≡ the last output symbol on input string α to $M|s$.
$M|s<\alpha>$ ≡ the output string on input string α to $M|s$.

First we give some definitions which are needed for the following discussion.

Definition 1: A machine M is *minimal* if the number of states of M is less than or equal to the number of states of M' for any Mealy machine M' equivalent to M.

Definition 2: A machine M is *completely specified* if from each state it has a transition for each input symbol. M is *incompletely specified* if it is not completely specified.

Definition 3: A machine M is *strongly connected* if for each state pair (s_i, s_j) there is a transition path going from s_i to s_j.

Definition 4: A *transition table* of M is a table consisting of two subtables: an output subtable and a next-state subtable, each with rows and columns identified by the states and input symbols of M, respectively. An entry in the output (next-state) subtable specifies, corresponding to a state s and an input symbol A of M, the output (next-state) of $M|s$ on A.

Definition 5: A *test subsequence* for M is a sequence of input symbols for testing either a state or a transition of M.

Definition 6: A *β-sequence* for M is a concatenation of test subsequences for testing all transitions of M.

Definition 7: A *test sequence* for M is a sequence of input symbols which can be used in testing conformance of implementations of M against the specification of M.

For the U-, D-, and W-methods, test sequences consist of concatenation of test subsequences of a β-sequence after suitable optimizations.

Definition 8: An *optimized test sequence* is a test sequence such that no subsequence of it is completely contained in any other subsequence.

It is understood throughout this paper that I, O, and s (possibly with subscripts or superscripts) will denote input symbol I, output symbol O, and state s, respectively, for a machine. We use λ to denote the null output, and r (or Rset) for reset input symbol. The reset input takes a machine M to its initial state from any state of the machine.

In Fig. 1 and Table I, we give the transition diagram and the transition table for a Mealy machine M used as an example for generating test sequences by the four methods mentioned above. In this example, machine M has two inputs which are A and B. M has transitions for both inputs from all states except for state 0. Machine M is made fully specified by adding a self loop B/λ to state 0, where the symbol λ represents null output. Transitions on reset input r for each state in M are not shown in Fig. 1 as they are not part of the original machine. But adding an edge r/λ from each state to the initial state 0 guarantees that M is strongly connected. Also, it is easy to check that M is minimal.

A. The T-Method

The T-method [7] is relatively simple, compared to the other three methods discussed. This method assumes a minimal, strongly connected, and completely specified Mealy machine model. A test sequence (called a transition-tour sequence) can be generated by simply applying random inputs to a fault-free machine until the machine has traversed every transition at least once. However, the sequence generated may contain many redundant inputs which in turn generate loops in the transition tour. These redundant inputs are removed using a reduction procedure.

In our implementation of the T-method, a machine does not need to be completely specified but it must be strongly connected as it is a necessary condition for producing a transition tour. For an incompletely specified machine, the transition tour is obtained by traversing edges in the original machine.

It is obvious that a test sequence generated by the T-method only checks for the existence of transitions and does not test the tail states of the transitions.

Example: A transition-tour sequence for machine M in Fig. 1 is shown below (together with the corresponding state sequence in the line below):

B A B A B A A A A A A A B B
0 0 3 3 4 0 3 4 2 1 4 2 1 2

Note that the above test sequence visits the transition from state 0 to state 0, which is not part of the original machine.

B. The U-Method

The U-method [8] assumes a minimal, strongly connected, and completely specified Mealy machine M. This

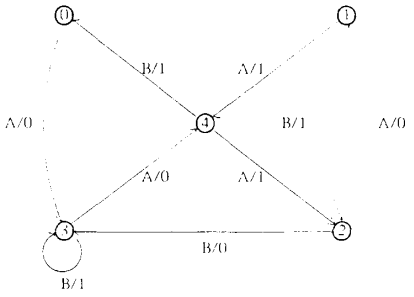


Fig. 1. A transition diagram for a machine M.

TABLE I
A TRANSITION TABLE FOR MACHINE IN FIG. 1

input state	output A	output B	next-state A	next-state B
0	0	λ	3	0
1	1	1	4	2
2	0	0	1	3
3	0	1	4	3
4	1	1	2	0

method involves deriving a *unique input/output* (UIO) sequence for each state of M. A UIO sequence for a state of M is an I/O behavior that is not exhibited by any other state of M.

A β-sequence is constructed by concatenating the test subsequence for each transition. For each state transition edge (s_i, s_j) in machine M, we generate its test subsequence as follows: 1) apply reset input r to M so that M is reset to the initial state 0, 2) find the shortest path $SP(s_i)$ from state 0 to state s_i, 3) apply an input symbol such that M makes a state transition to state s_j, 4) apply the UIO for state s_j.

In our implementation of the U-method, a machine does not need to be completely specified for the generation of UIO sequences for the states of the machine.

Example: As an application of this method, we use the machine M shown in Fig. 1 and Table I. Table II shows a set of UIO sequences for states of M. A state can be uniquely identified by observing the output string produced by the application of the input string from its UIO sequence. Thus, if the input string is AA and the output string is 11, we know we were at state 1 before the application of the string.

The β-sequence generated by the application of U-method for M in Fig. 1 is given below.

β-sequence:

r A B B
r B B
r A A A A A A A
r A A A A B B
r A A A A A A
r A A A B B B
r A A A A
r A B B B
r A A A B
r A A B B

TABLE II
UIO SEQUENCES FOR M IN FIG. 1

state	UIO
0	B/λ
1	A/1 A/1
2	B/0
3	B/1 B/1
4	A/1 A/0

An optimized test sequence constructed from the above test subsequences is:

rAAAAAAArAAAABBrAAABBBrAABBrABBBrBB

C. The D-Method

The D-method [10] assumes a Mealy machine which is minimal, strongly connected, completely specified and possesses a distinguishing sequence (DS). An input string x is said to be a *distinguishing sequence* of a machine M if the output string produced by M in response to x is different for each starting state. The key idea of this method is to compute a DS (if it exists) for a machine M. This can be done by constructing the "distinguishing tree" inductively on tree levels [11].

The construction of β-sequence for the D-method follow the same procedure as for the U-method but with every occurrence of a UIO sequence for a state replaced with the DS. But unlike the U-method, our implementation of this method requires a completely specified machine for the generation of a DS.

Example: We use the machine M in Fig. 1 and Table I to show an application of the D-method. It is easy to check that BB is the shortest DS for this machine. Table III shows the output string obtained by applying this DS to each state of M. Thus, if the output string is 10 on applying DS, we know we were at state 1 before the application of DS.

The β-sequence generated by the application of D-method for M in Fig. 1 is given below.

β-sequence:

r A B B
r B B B
r A A A A A B B
r A A A A B B B
r A A A A B B
r A A A B B B
r A A B B
r A B B B
r A A A B B
r A A B B B

An optimized test sequence constructed from the above test subsequences is:

rAAAAABBrAAAABBBrAAABBBrAABBBrABBBrBBB

D. The W-Method

The W-method [12] assumes a minimal, strongly connected, and completely specified Mealy machine. It in-

TABLE III
OUTPUTS ON DS FOR M IN FIG. 1

state	Mls<DS>
0	λλ
1	10
2	01
3	11
4	1λ

volves deriving a characterization set W of the FSM. A *characterization set* W for M is a set consisting of input strings $\alpha_1, \cdots, \alpha_k$ such that the last output symbols observed from the application of these strings (in a fixed order) are different at each state of M, i.e., $M|s_1(\alpha_1, \cdots, \alpha_k) \neq M|s_2(\alpha_1, \cdots, \alpha_k)$, where s_1 and s_2 are any two different states of M and $M|s_i(\alpha_1, \cdots, \alpha_k) = (M|s_i(\alpha_1), \cdots, M|s_i(\alpha_k))$, $i = 1, 2$.

Our modification to the W-method uses an idea similar to U-method in the generation of test sequences. Here the W set plays the role of identifying a state of a machine M. Test sequences for M are generated as follows:

1) Construct the characterization set $W = \{\alpha_1, \alpha_2, \cdots, \alpha_k\}$ as in [12], where α_i, $1 \leq i \leq k$, is an input string.

2) Generate a β-sequence as for the U-method except replace UIO(s) for state s with W with the understanding that

$$\begin{aligned} \mathrm{SP}(s) \,@\, W &= \mathrm{SP}(s) \,@\, \{\alpha_1, \alpha_2, \cdots, \alpha_k\} \\ &= \{\mathrm{SP}(s) \,@\, \alpha_1, \mathrm{SP}(s) \,@\, \alpha_2, \cdots, \\ &\quad \mathrm{SP}(s) \,@\, \alpha_k\} \end{aligned}$$

where SP(s) is the shortest path from the initial state to state s.

Like the D-method, our implementation of W-method requires a completely specified machine for the generation of a characterization set W.

Example: It is easy to check that $\{A, AA, B\}$ is a characterization set W for machine M in Fig. 1. Table IV shows the last output symbols from characterization set W applied to the states of M in Fig. 1. By observing the output sequences, we can easily identify each state of M. Thus, if the output produced on applying the characterization set W is 101, we know machine M was in state 4 before W was applied.

The β-sequence generated by the application of W-method for M in Fig. 1 is given below.

β-sequence:

r A A
r A A A
r A B
r B A
r B A A
r B B
r A A A A A A
r A A A A A A A
r A A A A A B
r A A A A B A
r A A A A B A A
r A A A A B B
r A A A A A
r A A A A A A
r A A A A B
r A A A B A
r A A A B A A
r A A A B B
r A A A
r A A A A
r A A B
r A B A
r A B A A
r A B B
r A A A A
r A A A A A
r A A A B
r A A B A
r A A B A A
r A A B B

TABLE IV
LAST OUTPUT SYMBOLS ON W FOR M IN FIG. 1

state	Mls(A)	Mls(AA)	Mls(B)
0	0	0	λ
1	1	1	1
2	0	1	0
3	0	1	1
4	1	0	1

An optimized test sequence constructed from the above test subsequences is:

rAAAAAAArAAAAABrAAAABAArAAAABBrAAABAArAAABBrAABAArAABBrABAArABBrBAArBB

E. Software Tools for Generating Test Sequences

All four protocol test sequence generation techniques (T-, U-, D-, and W-methods) have been implemented in C language [13] and all run on the VAX/UNIX system. Each implementation of these methods accepts a machine input in the form

$$< \mathrm{cs}, \mathrm{ns}, \mathrm{i}, \mathrm{o} >$$

where cs, ns, i, and o are integer representations for the current state, next state, input, and output, respectively, for each transition in the formal specification of a protocol. It is assumed that the state and input numberings are consecutive and each starts from 0. This speeds up the implementations by embedding state and input numbers into the array indexes as arrays are used to store information about transitions in the FSM. To generate test sequences for a protocol using these software tools, two constants, NoOfStates and NoOfInputs, need to be provided to define the numbers of states and input symbols in the protocol machine.

The implementation of the T-method is straightforward and directly follows the algorithms given in [7]. Only one output file, Tseq, containing the test sequence is generated.

For the U-, D-, and W-methods, five output files named below are generated.

1) UIO or DS or Wset: The file UIO contains the UIO sequences for the U-method, file DS contains the distinguishing sequence for the D-method, and file Wset contains the characterization set for the W-method.

2) Bseq: This file contains the β-sequence for testing the transitions. Each line of the file represents a test subsequence for a transition.

3) Bopt: This file contains the optimized test sequence for testing a machine. The optimization is done by eliminating all of the test subsequences in the file Bseq which are completely contained in other test subsequences in that file.

In our implementation of the software tools for the D- and W-methods, the procedures described in [10] and [12] for the derivation of test sequences have been modified. For the D-method, we use the algorithm for the computation of distinguishing sequence given in [10] but follow the U-method for the generation of the test sequence. The same is done to the W-method. Thus, a characterization set is derived as given in [12] while the test sequence is generated as in the U-method.

III. Fault Coverage of Protocol Test Methods

The ability of a test sequence to decide whether a protocol implementation conforms to its specification solely relies upon the range of faults or errors that it can detect. To evaluate the fault coverage of a test sequence, we must compute the class of FSM's which are not equivalent to the specification FSM but will produce the same outputs as the specification when the test sequence is applied. Such machines are the ones whose nonequivalence with the specification would not be detected by the test sequence. This arises from the fact that not all faults may be detected by a test sequence.

The following definitions are introduced to enhance subsequent discussion of this section.

A protocol entity is modeled as a deterministic finite state machine, which is represented as a directed graph. The vertices of the graph are the states of the FSM. Each edge is a possible state transition in the finite state machine, and has a label containing an input and output operation. For a real protocol, however, it may not cover all possible state-input combinations in the finite state machine. The edges in a given protocol machine are referred to as *core edges. For the unspecified state-input combinations, we assume that the protocol entity produces null output and remains in its present state.* These edges are referred to as *noncore edges.*

As explained above, a test sequence for a protocol is generated to determine whether the input/output behavior of an implementation conforms to its specification. Since most protocols are not completely specified, conformance is defined at two levels, namely, weak and strong conformance [23].

Definition 11: An implementation has *strong conformance* to the specification if both generate the same outputs for all input sequences.

Definition 12: An implementation has *weak conformance* to a specification if the implementation has the same input/output behavior as the protocol specification consisting of core edges only. But it has unspecified behavior for the input-state combinations specified by those noncore edges.

The test sequences for a protocol generated by the T-, U-, D-, and W-methods are expected to detect, in general, different combinations of faults in a protocol implementation. The following theorems can be stated for the fault detection capabilities of the weak and strong conformance test sequences generated by the four test methods.

Theorem 1: The fault coverage of the weak conformance test sequence for the U-method is better than the fault coverage of the weak conformance test sequence for the T-method.

The proof of this theorem is based on the observation that the weak conformance test sequence for the U-method tests edge labels as well as tail states of transitions, while the weak conformance test sequence for the T-method tests edge labels of transitions only.

Theorem 2: The fault coverage of the strong conformance test sequence for the U-, D-, and W-methods is better than the fault coverage of the strong conformance test sequence for the T-method.

The proof of this theorem is similar to the proof in Theorem 1.

Theorem 3: The fault coverage of strong conformance test sequences for the U-, D-, and W-methods are the same.

The proof of this theorem is based on 1) the machine under test is complete, 2) all U-, D-, and W-method test sequences are derived in the same manner except different characterizing entities for testing state are used, 3) each of U-, D-, and W-method test sequence can detect errors in the edge labels as well as errors in the tail states of transitions.

A. Procedure to Estimate Fault Coverage

Estimation of fault coverage of a test sequence is a difficult task because the number of machines that must be examined is very large. For example, a specification machine with n states, m inputs, and p outputs can have $(np)^{(mn)}$ possible implementations [14], [15]. It is obviously impossible to examine all of these machines. Instead, we confine to testing random machines that are marginally different from the specification machine.

Random faulty machines are generated by changing the tail state(s) and/or the output(s) of one or more edges of the specification machine and can be categorized into the following classes:

Class 1: This class is formed by altering the output of a random edge from the specification machine.

Class 2: Same as class 1 except the tail state of one random edge is modified.

Class 3: The outputs of two random edges in the specification machine are changed to form this class of machines.

Class 4: Same as class 3 except the tail states of two random edges are altered.

Class 5: The class of random machines is generated by changing the tail state of one random edge and the output of another edge of the specification machine.

Class 6: The tail state and the output of a random edge in this class of machine are different from the specification machine.

Class 7: The tail states and the outputs of two random edges are changed to obtain this class of random machines.

Class 8: This class of machines is obtained by changing the tail state of one random edge and the outputs of another two random edges of the specification machine.

Class 9: The tail states of two random edges and the outputs of another two random edges of this class of random machines differ from the specification machine.

Class 10: The tail states of three random edges and the outputs of another two random edges are modified to generate this class of random machines.

To ensure fairness, edge(s), new values for the tail state(s), and the output(s) are taken from independent pseudo-random sequences [16].

Because most real protocols are incompletely specified, conformance testing is studied at two levels using weak and strong conformance test sequences. Thus, we need to estimate fault coverage of two test sequences for a specification machine. For deriving weak conformance test sequence using the methods described in Section II, however, we may need to add artificial edge(s) to the machine to satisfy the necessary condition(s) of the method. For strong conformance test sequence derivation, we must add those missing edges to the original machine so that it becomes fully specified. *These artificial edges will produce* λ *(null) as outputs and machine remains in the same state (self loop).*

To assess the reliability of a test sequence, we must have a way to determine whether those machines which pass the test sequence test (i.e., produce the same outputs as the specification machine) actually conform to the specification machine. The following algorithm [24] can be employed for this purpose.

```
is1 := initial state for machine F1;
is2 := initial state for machine F2;

Set1 := ∅;
Set2 := {(is1, is2)};

while (Set2 ≠ ∅)
   begin
   pick an element (s1, s2) ∈ Set2;
   for (each outgoing edge e1 from s1)
      if (there exists an outgoing edge e2 from s2
         with the same label as e1)
         begin
         tuple := {(tail(e1), tail(e2))};
         if (¬ (tuple ∈ Set1) and ¬ (tuple ∈ Set2))
            Set2 := Set2 ∪ tuple;
         Set2 := Set2 − {(s1,s2)};
         Set1 := Set1 ∪ {(s1,s2)};
         end;
      else
            F2 does not conform to F1;
   end;
F2 conforms to F1;
```

The following summarizes the essential steps in estimating the fault coverage of a test sequence.

1) The specification finite state machine is read in.

2) The test sequence generated by a test sequence generation method is read in.

3) Random machines which are marginally different from the specification finite state machine are generated as described above.

4) The test sequence is applied to each of the machines generated in step 3) to check if they produce the same output as the specification machine.

5) Machines that passed the test in step 4) are checked if they actually conform to the specification machine.

Our procedure for studying fault coverage of a protocol test sequence generated by the four methods discussed above is similar to a procedure used in [23] for a similar study for sequences generated by the U-method. Similar ideas have also been used in the testing of digital circuits [11].

B. Fault Coverage Estimation Using an Example

Fig. 1 and Table I in Section II show the transition digram and the transition table for a Mealy machine to be used for the estimation of fault coverage of test sequences produced by the four methods above. This machine is typical of a machine for a real-world protocol standard which is generally more complex. In Section IV, we discuss test sequences and their fault coverage for a subset of NBS Class 4 Transport Protocol. This protocol subset has 15 states and 27 inputs.

To estimate the fault coverage of various test sequences, ten classes of random machines are constructed as described in the previous section. For each class, one million randomly generated machines are subjected to weak and strong conformance test sequences generated by the test sequence generation techniques. The results are presented below.

1) Weak Conformance Test: Although the machine in Fig. 1 is not fully specified, it is strongly connected. Thus our implementation of the T-method is able to generate a test sequence for doing the weak conformance test against the specification machine. The optimized weak confor-

mance test sequence generated is:

AAAAABABAABAAAB

and the results of its fault coverage are given in Table V.

The entries in Table V show that the T-method is able to detect one or more faults in output labels (class 1 and class 3) but not in tail states of transitions (class 2 and class 4). This is because the T-method test sequence does not test whether the tail state of a transition is correct or not. Thus, combination of faults in one or more edges involving an error in the tail state of a transition (classes 5–10) cannot all be detected by a T-method test sequence.

As for the T-method, our implementation of the U-method is able to generate a UIO sequence for each state even though the machine is not fully specified. The optimized test sequence produced by the method for the example machine in Fig. 1 is:

rAAAAAAArAAAABBrAAABBBrAABAArABBB

and the results of its fault coverage are given in Table VI.

Unlike the T-method test sequence, however, the results of Table VI show that the U-method test sequence is able to detect all single faults, multiple faults of the same kind, and combination of faults in one single edge (class 1–4 and class 6). But it is not the case for the combination of faults in multiple edges as in machines for class 5 and classes 7–10. A simple example described in Section V illustrates this point.

Because our implementations of the D-method and the W-method assume a fully specified machine, they are unable to generate a test sequence for doing weak conformance test. Therefore, we do not have the results of fault coverage of the D-method and W-method for weak conformance testing.

2) Strong Conformance Test: To generate strong conformance test sequences, we add self loops for those unspecified inputs and treat the loops as core edges. In our example machine, one self loop with label B/λ is added to state 0 for completing the machine. To estimate the fault coverage of strong conformance test sequences generated by the four methods, the same procedure is carried out as in the estimation of weak conformance test sequences except we apply the strong conformance test sequence to the faulty machines. The results of the strong conformance test sequences are given below.

Because the original machine is strongly connected, our implementation of the T-method has no problem in generating the strong conformance test sequence. An optimized test sequence is:

AAAAABBABAABAAAB

and the results of its fault coverage is shown in Table VII.

The T-method strong conformance test sequence, like its weak conformance counterpart, is only able to detect faults in output labels but not others, as shown by the results in Table VII. Again, this arises from the fact that the T-method test sequence does not test the tail state of a transition. Thus, the T-method will not be able to detect all faults involving errors in the tail states of transitions.

Because of changes in the machine, different UIO sequences for some states of the machine are generated. The optimized strong conformance test sequence produced by our implementation of the U-method is:

rAAAAAAArAAAABBrAAABBBrAABBrABBBrBB.

Since the machine becomes fully specified, our implementation of the D-method is therefore able to generate an optimized strong conformance test sequence as given below:

rAAAAABBrAAAABBBrAAABBBrAABBBrABBBrBBB

For the W-method, our implementation generates an optimized strong conformance test sequence for the modified machine as shown below:

rAAAAAAArAAAAABrAAAABAArAAAABBrAAABAArAAABBrAABAArAABBrABAArABBrBAArBB.

The results of fault coverage for U-, D-, and W-methods are given in Table VIII since they all have the same performance.

The results of Table VIII show that U-, D-, and W-method strong conformance test sequences are able to de-

TABLE V
WEAK CONFORMANCE TEST FOR T-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test	No. Equivalent to Specification Machine
1	1000000	531770	531770	468634
2	1000000	367325	257976	202667
3	1000000	309725	309725	246883
4	1000000	156320	83495	58779
5	1000000	195805	136331	94491
6	1000000	214588	160360	97224
7	1000000	60864	35024	19453
8	1000000	115238	79787	50183
9	1000000	50380	26023	14519
10	1000000	24595	9773	4837

TABLE VI
WEAK CONFORMANCE TEST FOR U-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test	No. Equivalent to Specification Machine
1	1000000	531770	531770	468634
2	1000000	257976	257976	202667
3	1000000	309725	309725	246883
4	1000000	83495	83495	58779
5	1000000	138716	136331	94491
6	1000000	160360	160360	97224
7	1000000	37156	35024	19453
8	1000000	82175	79787	50183
9	1000000	27462	26023	14519
10	1000000	10492	9773	4837

TABLE VII
STRONG CONFORMANCE TEST FOR T-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test
1	1000000	374685	374685
2	1000000	295799	202667
3	1000000	165058	165058
4	1000000	104785	58779
5	1000000	111266	75750
6	1000000	122495	75793
7	1000000	25816	13696
8	1000000	49083	33521
9	1000000	17530	9635
10	1000000	7178	3376

TABLE VIII
STRONG CONFORMANCE TEST FOR U-, D-, AND W-METHODS

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test
1	1000000	374685	374685
2	1000000	202667	202667
3	1000000	165058	165058
4	1000000	58779	58779
5	1000000	75750	75750
6	1000000	75793	75793
7	1000000	13696	13696
8	1000000	33521	33521
9	1000000	9635	9635
10	1000000	3376	3376

tect all faults. This is because for a strong conformance test of a completely specified machine, each transition in the machine along with its tail state are tested and therefore the test sequences capture all faults. As the test sequences for the three methods are generated in the same manner except for using different characterizing entities (UIO for U-method, DS for D-method and W set for W-method), it is expected that the three methods give the same performance for the strong conformance test.

C. Analysis of Results

For weak conformance testing, the test sequence for the T-method is not able to detect all of the single faults. In fact, it is perfect for detecting errors in the output labels but not in the tail states of transitions. For the U-method, the weak conformance test sequence is perfect in detecting single faults. But for a combination of faults such as errors in the output of one edge and tail state of another edge (class 5 and class 7), it is not able to detect all of them. For the D- and W-methods, we are not able to estimate their fault coverage since our implementations are not able to generate test sequences from a partially specified machine.

For strong conformance testing, the T-method test sequence again is not able to detect all single faults. All the other methods are perfect in detecting faults in all classes of machines and have the same performance. This is because strong conformance test sequences for these methods test all edges, including the tail states of these edges, of a machine.

Based on the results in the tables above, we can conclude that the fault detection capabilities of test sequences for the U-, D-, and W-methods are better than that for the T-method. Also the fault detection capabilities of the test sequences for the U-, D-, and W-methods are the same for strong conformance testing. Similar conclusions are drawn for the fault coverage of protocol test sequence generation methods applied to a subset of NBS Class 4 Transport Protocol discussed in Section IV.

IV. TEST SEQUENCES FOR NBS CLASS 4 TRANSPORT PROTOCOL

We now describe an application of our protocol test sequence generators based on the four methods (T-, U-, D-, and W-methods) discussed in Section II to an actual communication protocol. To actually test a protocol implementation for conformance to its specification, an adequate test architecture needs to be provided [2], [5]. Research in developing suitable test architectures is being conducted in several countries. A general organization of a testing architecture consists of a tester which is presumably a correct implementation of the protocol and a responder which contains an implementation under test.

A. NBS Class 4 Transport Protocol

The National Bureau of Standards has developed a set of standards for the transport protocol layer which is divided into several classes [17]–[19]. Each class provides the same service to the session layer, but is built on different types of service provided by the network layer. The class 2 transport protocol assumes a reliable service offered by the network layer which is responsible for error-free transmission of data, while the class 4 protocol assumes an unreliable network service and takes care of all error detection and recovery. The class 4 transport protocol is the most complex of all the NBS protocol classes [20]. It uses a large number of timers (12) to ensure that data arrives at destinations in order. The transport protocol has five types of service primitives which are: request, indication, response, confirm, and cancel. The request primitive is an initial request from the user for a service. The indication primitive signals the corresponding user of the connection establishment attempt. The response and confirm primitives are used only in connection establishment and represent the request (response) and indication (confirm) of the entity that, having received initial connection request, wishes to establish the connection. Cancel is used to inform the operating system that a timer is no longer needed that had previously been requested. NBS TP4 defines ten Transport Protocol Data Units (TPDU's) which are CR, CC, DR, DC, GR, ERR, DT, XPD, AK, and XAK.

B. Test Sequences for NBS TP4 from the Four Methods

We applied protocol test sequence generators (based on T-, U-, D-, and W-methods) to a subset of the NBS TP4 shown in Fig. 2. This subset excludes transitions for unit data, expedited data, data transfer, and close request service primitives. It is clear from Fig. 2 and Table IX that the FSM for this protocol subset is incomplete. In fact, for each of the 15 states in this machine, no more than 6 out of 27 possible outgoing edges are specified. Thus, the number of noncore edges is about six times as many as the number of core edges in the machine. Table IX lists the names and abbreviations for the inputs to the FSM across the user (U), network (N) and system interfaces (S). Table X gives the abbreviated forms of the output primitives used in the analysis of this protocol.

In applying our tools to generate test sequences, several things need to be done. For the D- and W-methods, we must complete the FSM in order to produce a distinguishing sequence and a W set, respectively. There are many ways to complete an FSM, we chose to complete the FSM such that for each unspecified input at each state a transition edge is added for that input with null output. For the U-method, completion of this particular subset of TP4 FSM is not necessary. Complete specification is only a sufficient condition (not necessary condition) in applying

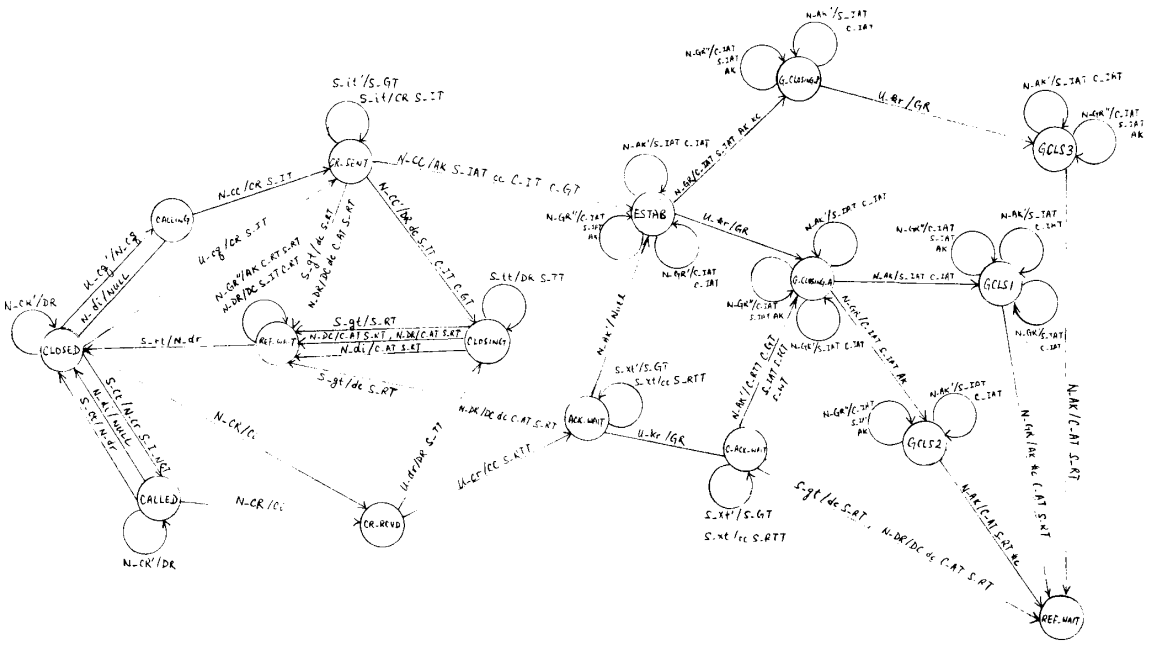


Fig. 2. A subset of NBS Class 4 transport protocol.

TABLE IX
ABBREVIATED FORMS OF THE INPUT PRIMITIVES USED IN THE ANALYSIS OF THE NBS TP4 SUBSET

N_CR	from N:N_DATA.indication(connection_request)
N_CR'	from N:N_DATA.indication(bad CR)
N_CC	from N:N_DATA.indication(connection_confirm)
N_CC'	from N:N_DATA.indication(bad CC)
N_DR	from N:N_DATA.indication(disconnect_request)
N_DC	from N:N_DATA.indication(disconnect_confirm)
N_AK	from N:N_DATA.indication(acknowledgement)
N_AK'	from N:N_DATA.indication(AK and AK_ok)
N_GR	from N:N_DATA.indication(graceful_close_request and GR_arrived)
N_GR'	from N:N_DATA.indication(graceful_close_request and not GR_arrived and $cond_1$)
N_GR''	from N:N_DATA.indication(graceful_close_request and $cond_2$)
U_cq	from U:CONNECT.request
U_cq'	from U:CONNECT.request and nc_required
U_cr	from U:CONNECT.response
U_dr	from U:DISCONNECT.request
N_ci	from N:CONNECT.indication
N_cc	from N:CONNECT.confirm
N_di	from N:DISCONNECT.indication
S_tt	from S:S_TIMER.response(terminate_timer)
S_it	from S:S_TIMER.response(initiate_timer)
S_it'	from S:S_TIMER.response(initiate_timer) and $cond_3$
S_gt	from S:S_TIMER.response(giveup_timer)
S_rt	from S:S_TIMER.response(reference_timer)
S_xt	from S:S_TIMER.response(retransmit_CC_timer)
S_xt'	from S:S_TIMER.response(retransmit_CC_timer) and $cond_3$
S_ct	from S:S_TIMER.response(incoming_nc_timer)

$cond_1$ is [from N:TPDU.tpdu_nr] ≤ rcv_window.
$cond_2$ is [from N:TPDU.tpdu_nr] < rcv_nxt or ≥ rcv_window.
$cond_3$ is [from S:Datum] ≥ rcount.

TABLE X
ABBREVIATED FORMS OF THE OUTPUT PRIMITIVES USED IN THE ANALYSIS OF THE NBS TP4 SUBSET

Output Label	Output Events
00	CR_TPDU (CR)
01	CC_TPDU (CC)
02	DR_TPDU (DR)
03	DC_TPDU (DC)
04	acknowledgment (AK)
09	set reference timer (S_RT)
10	set initiate timer (S_IT)
11	set giveup timer (S_GT)
12	set retransmit timer (S_RTT)
13	set terminate timer (S_TT)
14	cancel all timer (C_AT)
15	null output (Null)
25	N_connect_request (N_cq)
26	N_disconnect_request (N_dr)
27	N_connect_response (N_cr)
28	set incoming_nc timer (S_I_NCT)
51	connect_indication (ci)
61	connect_confirm (cc)
71	disconnect_confirm (dc)
80	cancel initiate timer (C_IT)
81	cancel giveup timer (C_GT)
82	cancel inactivity timer (C_IAT)
83	set inactivity timer (S_IAT)
84	set flow_control timer (S_FCT)
85	set window timer (S_WT)
86	cancel retransmit timer (C_RTT)
87	cancel reference timer (C_RT)
91	close_confirm (kc)

it. Also, we added a reset edge for each state to the initial state for applying the U-, D-, and W-methods.

Since the TP4 machine is not completely specified, only the T- and U-methods are able to generate weak conformance test sequences for TP4 and they are included in Appendixes A and B, respectively.

By completing the TP4 machine with self-loops, all of the four methods are able to generate strong conformance test sequences for TP4. The T-, U-, and W-method strong

conformance test sequences are too long to be included in this paper. The strong conformance test sequences for the T-, U-, and D-methods can be found in [26] in Appendixes C, D, and E, respectively. The characterization set W for the TP4 subset discovered by this method is

$$\{N\_CR\ N\_CC\ N\_DR\ N\_AK'\ U\_cq\ U\_cr\ N\_AK\ N\_GR\}.$$

Note that in each Appendix, the test sequence from each method is obtained by concatenating each line (test subsequence) in the given Appendix. For the T-method, there is no particular significance for each line. However, for the U-, D-, and W-methods, a line may either be a test subsequence for testing a particular state or a test subsequence for testing a particular transition edge for this TP4 FSM.

C. Fault Coverage of Test Sequences for NBS TP4

To estimate the fault coverage of the NBS TP4 test sequences generated by the four methods, ten classes of random machines were constructed as described in Section III-A. Because TP4 is such a large machine, only 10 000 random machines for each class were generated and subjected to weak and strong conformance test sequences of each method produced by our software tools. The results are presented below.

1) Weak Conformance Test: Since our implementations of the D- and W-methods require a fully specified machine, we are only able to do weak conformance testing for the T- and U-methods. The results of fault coverage of the T- and U-method weak conformance test sequences are given in Table XI and Table XII, respectively.

The results of Table XI are similar to those obtained for the T-method weak conformance test sequence for the example machine in Section III. The only differences are the entries for class 6 machines in the third and fourth columns of this table. A nonequivalent machine in class 6 passes the test sequence test only if one of its edges has the same output label but different tail state as the same edge in the specification machine. But since the number of core edges is so small compared to the number of noncore edges, it is expected that faults are introduced to noncore edges more than to the core edges in the 10 000 random machines generated for TP4. Therefore, it is possible that no such nonequivalent random machines are generated and we have same number of machines pass the test sequence and machine equivalence tests.

The results of Table XII have a similar pattern to the results obtained by the U-method weak conformance test sequence for the example machine in Section III. The difference is that the test sequence for TP4 is able to detect more combinations of faults such as classes 7–10. This, again, can be explained by the fact that faults are introduced to noncore edges more than to the core edges. Hence, fewer nonequivalent machines pass the test sequence test.

2) Strong Conformance Test: By completing the TP4 machine, our software tools are able to generate strong conformance test sequences of the four methods. To estimate fault coverage of these test sequences, the same procedure is carried out as in the estimation of weak conformance test sequences and their results are given below.

TABLE XI
WEAK CONFORMANCE TEST FOR T-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test	No. Equivalent to Specification Machine
1	10000	8523	8523	60
2	10000	8579	8546	726
3	10000	7227	7227	1
4	10000	7350	7296	61
5	10000	7280	7250	4
6	10000	8469	8469	6
7	10000	7136	7134	0
8	10000	6230	6200	0
9	10000	5317	5273	0
10	10000	4554	4505	0

TABLE XII
WEAK CONFORMANCE TEST FOR U-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test	No. Equivalent to Specification Machine
1	10000	8523	8523	60
2	10000	8546	8546	726
3	10000	7227	7227	1
4	10000	7296	7296	61
5	10000	7250	7250	4
6	10000	8469	8469	6
7	10000	7134	7134	0
8	10000	6201	6200	0
9	10000	5273	5273	0
10	10000	4505	4505	0

TABLE XIII
STRONG CONFORMANCE TEST FOR T-METHOD

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test
1	10000	218	218
2	10000	947	726
3	10000	11	11
4	10000	99	61
5	10000	20	15
6	10000	21	19
7	10000	0	0
8	10000	0	0
9	10000	0	0
10	10000	0	0

Table XIII shows results for the strong conformance test sequence of T-method for TP4 similar to those obtained by the T-method strong conformance test sequence for the example machine in Section III. Because the TP4 is such a large machine and no fewer than three faults are introduced in classes 7–10, it is therefore unlikely to generate machines that are equivalent to the specification machine. Thus, we have no random machines in these four classes pass the test sequence test.

The results of fault coverage for the U-, D-, and W-methods strong conformance test sequences generated for TP4 are given in Table XIV since they all have the same performance.

Like the results obtained by the strong conformance test sequences of the three methods for the example machine in Section III, Table XIV shows that the U-, D-, and W-method strong conformance test sequences are able to de-

TABLE XIV
STRONG CONFORMANCE TEST FOR U-, D-, AND W-METHODS

Test for Class	No. of Randomly Generated Machines	No. Passing Test Sequence Test	No. Passing Machine Equivalence Test
1	10000	218	218
2	10000	726	726
3	10000	11	11
4	10000	61	61
5	10000	15	15
6	10000	19	19
7	10000	0	0
8	10000	0	0
9	10000	0	0
10	10000	0	0

tect all faults. As explained above, it is unlikely to generate machines in classes 7–10 that are equivalent to the specification machine and hence, we have no machine pass the test sequence tests.

V. COMMENTS ON PROTOCOL TEST METHODS

In this section, we discuss the assumptions, applicability, fault detection capability, lengths of test sequences, and nonuniqueness of test sequences and test subsequences generated by the T-, U-, W-, and D-methods discussed in the previous sections.

1) Assumptions: All of these four methods assume minimal, strongly connected, and completely specified Mealy machine models of protocol entities.

Minimality provides an effective standard that enables one correctly implemented machine to test against others.

The assumption of strong connectivity guarantees that a machine can reach other states from any state. In the case of T-method, this guarantees that a transition-tour sequence can be generated. This assumption is not explicitly required for the generation of UIO sequences, distinguishing sequence DS or characterization set W for the U-, D-, and W-methods, respectively.

A completely specified machine may not be needed for generating a weak conformance test sequence by the T-method but it is necessary for generating strong conformance test sequences by all the methods. However, the completely specified machine assumption is rarely met in the specification of a protocol machine for a real-world protocol such as the NBS class 4 transport protocol discussed in Section IV.

2) Applicability: A completely specified protocol FSM is a necessary and sufficient condition for generating a distinguishing sequence (if it exists) and a W set for the D- and W-methods, respectively. But for the U-method, it is only a sufficient condition for producing a set of UIO sequences. Since real protocols are seldom complete, the completion of the protocol FSM, in general, introduces a large number of artificial transition edges, which, in turn, result in a test sequence which has intolerably long length. For the T-method, it may not be able to generate a test sequence for a machine that is not strongly connected.

Except for the D-method, the other three methods guarantee the existence of a testing sequence for an FSM which satisfies the assumption of minimality, strong connectivity, and complete specification. The D-method requires further that a distinguishing sequence be existent [11]. However, it is not certain how to provide a "dressing up" procedure for a machine so that it possesses a distinguishing sequence as is done for machines that are incompletely specified or not strongly connected. The D-method may thus not be applicable for all protocols.

3) Fault Detection Capability: For incompletely specified machines, T-method weak conformance test sequences are able to detect faults in output labels but not in tail states of transitions. In fact, they are unable to detect any combination of faults involving errors in the tail states of transitions. The main reason is that the T-method only checks for the existence of transitions regardless of their starting states and ending states.

As the results in Section III show, U-method weak conformance test sequences are able to detect single faults but not two or more faults. It is not possible to assess fault detection capabilities of the D- and W-methods for weak conformance testing since we are unable to generate weak conformance test sequences from these methods.

For completely specified finite state machines, test sequences generated by the four methods are able to detect faults in output labels of transitions. The same is true for all but the T-method for detecting faults in tail states of transitions. T-method strong conformance test sequence is not able to detect faults in tail states for the same reason explained above for T-method weak conformance test sequences.

Besides capturing single faults, the strong conformance test sequences for the U-, D-, and W-methods are capable of detecting all other faults. This is because all edges along with their tail states in a completely specified finite state machine are tested. Since test sequences of these three methods are generated in the same manner except different characterizing entities (UIO or U-method, DS for D-method, and W set for W-method) are used for recognizing states, they are expected to give the same performance.

4) Lengths of Test Sequences: The lengths of test sequences generated by the four methods, in the worst case, may differ to a large extent from each other. Based upon our modifications to the original D- and W-methods for generating β-sequence, the major factor contributing to the length of each test sequence lies in the choice of those characterizing entities (distinguishing sequence for D-method and W set for W-method). For example, if the length of a distinguishing sequence from the D-method is greater than the maximal length of UIO sequences from the U-method, then the test sequence generated from the D-method will be longer than the one from the U-method. Similar discussions hold for other cases. On the average, T-method will produce the shortest test sequence and W-method the longest test sequence among the four meth-

ods, while D- and U-methods generate test sequences of comparable lengths.

It is important to compare test methods in terms of the length of test sequences for some real protocols [25]. For the NBS TP4 subset discussed in Section IV, the approximate length of the strong conformance test sequences are:

T-Method	47 lines
U-Method	391 lines
D-Method	406 lines
W-Method	3240 lines.

These test sequences are too long to be included in this paper (see [26]). It is clear that the W-method gives an unacceptably long test sequence.

5) Nonuniqueness of Test Sequences: The test sequences generated by the four methods are not guaranteed to be unique. This is clearly true for the T-method since the test sequence, which is obtained from a transition tour through the states of an FSM, is generated randomly. The other three (U-, D-, and W-) methods also produce test sequences which are not necessarily unique. It is easy to see for the W-method since, in general, more than one W-set of a given size may exist for an FSM. Similar arguments apply to the U- and D-methods.

6) Building Test Sequence from Test Subsequences: The test sequence generated by the T-method is obtained by applying random inputs to a fault-free machine until the machine has traversed every transition at least once. Since edge(s) may have been added to an FSM to satisfy the strong connectivity assumption, a test sequence will test all the transitions in the original machine as well as those corresponding to added edge(s).

The test sequence generated by the U-method consists of a β-sequence which includes the reset inputs which enable the machine to go back to the start state to begin another test after testing a state or a transition. The reset provides a convenient way to reset the machine to the start state without turning off the machine or finding a lengthy input sequence to go back to the start state. An optimization procedure can be applied to the test sequence formed from test subsequences by eliminating test subsequences which are completely contained in some other test subsequences.

A test sequence generated by the D-method consists of a β-sequence. It differs from the U-method in that every test subsequence ends with a distinguishing sequence (DS) instead of a UIO sequence. A reset input begins every test subsequence and a similar optimization procedure can be applied to generate an optimized test sequence.

A test sequence generated by the W-method also consists of a β-sequence. It differs from the U- and D-methods in that a W-set is used in place of a UIO sequence or a DS. A reset input begins every subsequence. A similar optimization procedure is used to generate an optimized test sequence.

As a general evaluation of the four methods, we can make the following comments: T-method is simple but may not capture all single faults; D-method is more involved in its implementation and requires the existence of a distinguishing sequence, which may not exist for an FSM; U-method is easy to comprehend; and W-method, in general, will produce longer test sequences than others.

VI. Conclusions

In this paper, we discussed four formal methods (T-, U-, D-, and W-methods) for generating protocol test sequences from their specification. All of these four methods assume minimal, strongly connected, and fully specified Mealy machine models of protocol entities. These three conditions are not always true for real protocols. A fully or completely specified FSM is a necessary and sufficient condition for generating a distinguishing sequence (if it exists) and a W set for the D- and W-methods, respectively. But it is only a sufficient condition for producing a set of UIO sequences for the U-method. On the other hand, strong connectivity for a protocol FSM is a necessary condition for the T-method for generating a test sequence.

All of these methods except the T-method employ the same basic idea in generating test sequences: 1) use a characterizing entity (UIO sequences for U-method, distinguishing sequence for D-method, and W set for W-method) as a means for identifying the states of an FSM, and 2) apply the characterizing entity in generating a β-sequence for testing an FSM. For the T-method, a test sequence for an FSM is generated by visiting all the transitions of the FSM. The lengths of test sequences generated by the four methods, in the worst case, may differ to a large extent from each other. On the average, T-method will produce the shortest test sequence and W-method the longest test sequence among the four methods, while D- and U-methods generate test sequences of comparable lengths.

To assess the fault detection capabilities of these test methods, we use Monte-Carlo simulation to estimate the classes of implementation machines which, upon the application of the test sequence, produce the same outputs as the specification machine but are not equivalent to it. Our results show that T-method weak conformance test sequences are able to detect faults in output labels but not in tail states of transition edges. For U-method weak conformance test sequences, they can detect both kinds of single faults but not combinations of faults in some cases. In particular, they cannot detect double faults as demonstrated in Section VI. For strong conformance testing, T-method test sequences show the same behavior as its weak conformance counterpart while U-, D-, and W-method test sequences are capable of detecting all kinds of faults and give the same performance.

Since most real communication protocols are incompletely specified, only T- and U-methods are able to generate weak conformance test sequences. But these test sequences do not possess reliable fault detection capabilities as the results in Sections III and IV demonstrate. One so-

lution is to complete the protocols by adding self-loops for the missing edges. Then the strong conformance test sequences for U-, D-, and W-methods are able to detect all kinds of faults. However, this creates a large number of artificial edges, which, in turn, produces test sequences of intolerable lengths.

In this paper, we have not considered synchronization problem [21] and optimality of a test in generating test sequences from protocol specifications by the four methods. Both problems are important and deserve consideration. In connection with the optimal length test sequence, a method has been proposed in [22] which is based on the solution to the Chinese Postman Problem. The method generates a transition tour for an FSM like in the T-method.

For further details about this paper, see [26].

Appendix A
Weak Conformance Test Sequence for NBS TP4 Using the T-Method

The NBS Class 4 transport protocol subset discussed in Section IV was used to generate a weak conformance test sequence given below using the T-method. The test sequence is a concatenation of the following lines.

N_CR U_cr N_DR N_DR N_di S_rt U_cq N_CC N_AK' N_GR
N_AK' N_GR" U_kr N_AK' N_AK N_GR" S_rt N_CR' N_ci N_CR
U_dr N_DR S_rt U_cq' N_di N_CR U_cr N_AK' N_GR' N_GR"
U_kr N_AK' N_AK N_AK' N_GR S_rt U_cq N_DR S_rt N_ci
N_di U_cq' N_cc S_it S_gt S_rt N_CR U_dr N_DC S_rt
U_cq S_it' N_CC' N_di S_rt N_ci N_CR' S_ct N_CR U_cr
S_gt S_rt U_cq N_CC N_GR U_kr N_GR" N_AK S_rt N_ci
N_CR U_dr S_gt S_rt N_CR U_cr S_xt S_xt' U_kr N_DR
S_rt N_CR U_dr S_tt N_DC S_rt U_cq' N_cc N_CC U_kr
N_GR N_AK' N_AK S_rt U_cq N_CC U_kr N_GR' N_GR" N_AK
N_GR' N_GR" N_GR S_rt N_CR U_cr U_kr N_AK' N_GR N_GR"
N_AK S_rt N_CR U_cr U_kr S_gt S_rt N_CR U_cr U_kr
S_xt S_xt'

Appendix B
Weak Conformance Test Sequence for NBS TP4 Using the U-Method

The NBS Class 4 transport protocol subset discussed in Section IV was used to generate a weak conformance test sequence using the U-Method. The UIO sequences for the states of the TP4 subset discovered by this method are:

State	UIO	State	UIO
Closed	U_cq/16	Cr_Sent	N_CC/20
Cr_Revd	U_cr/22	Ack_Wait	N_AK'/15
Estab	N_GR/33	Closing	N_DR/23
Ref_Wait	N_DR/24	Calling	N_cc/16
Called	S_ct/26	C_Ack_Wait	N_AK'/34
G_Closing_A	N_AK/31	Gcls1	N_GR/32
G_Closing_P	U_kr/50 N_AK/23	Gcls3	N_AK/23
Gcls2	N_AK/36		

where the numerical output labels in the UIO sequence for states in the above table stand for following strings of outputs:

16 : CR S_IT
20 : AK S_IAT cc C_IT C_GT
22 : CC S_RTT
23 : C_AT S_RT
24 : DC S_IT C_RT
31 : S_IAT C_IAT
32 : AK kc C_AT S_RT
33 : C_IAT S_IAT AK kc
34 : C_RTT C_GT S_IAT S_FCT S_WT
36 : C_AT S_RT kc

An optimized weak conformance test sequence derived using the U-method is given below.

Rset N_CR U_cr N_DR N_DR
Rset N_CR U_cr N_AK' N_GR
Rset N_CR U_cr S_gt N_DR
Rset N_CR U_cr S_xt N_AK'
Rset N_CR U_cr S_xt' N_AK'
Rset N_CR U_cr U_kr N_DR N_DR
Rset N_CR U_cr U_kr N_AK' N_AK
Rset N_CR U_cr U_kr S_gt N_DR
Rset N_CR U_cr U_kr S_xt N_AK'
Rset N_CR U_cr U_kr S_xt' N_AK'
Rset N_CR U_dr N_DR
Rset U_cq N_CC N_AK' N_GR
Rset U_cq N_CC N_GR N_AK' U_kr N_AK
Rset U_cq N_CC N_GR N_GR" U_kr N_AK
Rset U_cq N_CC N_GR U_kr N_AK' N_AK
Rset U_cq N_CC N_GR U_kr N_AK N_DR
Rset U_cq N_CC N_GR U_kr N_GR" N_AK
Rset U_cq N_CC N_GR' N_GR
Rset U_cq N_CC N_GR" N_GR
Rset U_cq N_CC U_kr N_AK' N_AK
Rset U_cq N_CC U_kr N_AK N_AK' N_GR
Rset U_cq N_CC U_kr N_AK N_GR N_DR
Rset U_cq N_CC U_kr N_AK N_GR' N_GR
Rset U_cq N_CC U_kr N_AK N_GR" N_GR
Rset U_cq N_CC U_kr N_GR N_AK' N_AK
Rset U_cq N_CC U_kr N_GR N_AK N_DR
Rset U_cq N_CC U_kr N_GR N_GR" N_AK
Rset U_cq N_CC U_kr N_GR' N_AK
Rset U_cq N_CC U_kr N_GR" N_AK
Rset U_cq N_DR N_DR N_DR
Rset U_cq N_DR N_di N_DR
Rset U_cq N_DR S_rt U_cq
Rset U_cq N_DR N_GR" N_DR
Rset U_cq S_it N_CC
Rset U_cq S_gt N_DR
Rset U_cq S_it' N_CC
Rset U_cq N_CC' N_DR N_DR
Rset U_cq N_CC' N_DC N_DR
Rset U_cq N_CC' N_di N_DR
Rset U_cq N_CC' S_gt N_DR
Rset U_cq N_CC' S_tt N_DR
Rset N_CR' U_cq
Rset N_ci N_CR U_cr
Rset N_ci N_di U_cq
Rset N_ci N_CR' S_ct
Rset N_ci S_ct U_cq
Rset U_cq' N_di U_cq
Rset U_cq' N_cc N_CC

Acknowledgment

The authors wish to thank Dr. K. Sabnani of AT&T Bell Laboratories for useful discussions and to the anonymous referees for careful reading of the manuscript and for suggesting improvements. They are also grateful to Mr. K. W. Chan and Mr. T. T. Shih for valuable comments and help in implementing some software tools used in this study.

REFERENCES

[1] H. Zimmerman, "OSI reference model—The ISO model of architecture for open systems interconnection," *IEEE Trans. Commun.*, vol. COM-28, pp. 425-432, Apr. 1980.

[2] R. J. Linn and W. H. McCoy, "Producing tests for implementations of OSI protocols," in *Protocol Specification, Testing, and Verification, III*, H. Rudin and C. H. West, Eds. Amsterdam, The Netherlands: North-Holland, 1983, pp. 505-520.

[3] D. P. Sidhu, "Protocol verification via executable logic specifications," in *Protocol Specification, Testing, and Verification, III*, H. Rudin and C. H. West, Eds. Amsterdam, The Netherlands: North-Holland, 1983, pp. 237-248.

[4] D. P. Sidhu and C. S. Crall, "Executable logic specifications for protocol service interfaces," *IEEE Trans. Software Eng.*, vol. 14, Jan. 1988.

[5] D. Rayner, "Standardizing conformance testing for OSI," in *Protocol Specification, Testing, and Verification, V*, M. Diaz, Ed. Amsterdam, The Netherlands: North-Holland, 1986.

[6] D. P. Sidhu, "Protocol verification using prolog," ISU Preprint, 1985.

[7] S. Naito and M. Tsunoyama, "Fault detection for sequential machines by transition tours," in *Proc. IEEE Fault Tolerant Comput. Conf.*, 1981.

[8] K. Sabnani, and A. Dahbura, "A protocol test generation procedure," *Comput. Networks ISDN Syst.*, vol. 15, pp. 285-297, 1988.

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Reading, MA: Addison-Wesley, 1983.

[10] G. Gonenc, "A method for the design of fault detection experiments," *IEEE Trans. Comput.*, vol. C-19, pp. 551-558, June 1970.

[11] Z. Kohavi, *Switching and Finite Automata Theory*. New York: McGraw-Hill, 1978.

[12] T. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 178-187, Mar. 1978.

[13] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[14] A. D. Friedman and P. R. Menon, *Fault Detection in Digital Circuits*. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[15] M. A. Harrison, "On asymptotic estimates in switching theory and automata theory," *J. ACM*, vol. 13, pp. 151-157, 1966.

[16] W. Cheney and D. Kincaid, *Numerical Mathematics and Computing*. Monterey CA: Brooks/Cole, 1980.

[17] "Specification of a transport protocol for computer communications, Volume 1: Overview and services," Nat. Bureau Standards, Washington, DC, Rep. ICST/HLNP-83-1, Jan. 1983.

[18] "Specification of a transport protocol for computer communications, Volume 3: Class 4 protocol," Nat. Bureau Standards, Washington, DC, Rep. ICST/HLNP-83-1, Jan. 1983.

[19] "Specification of a transport protocol for computer communication, Volume 4: Service specifications," Nat. Bureau Standards, Washington, DC, Rep. ICST/HLNP-83-4, Jan. 1983.

[20] D. P. Sidhu and T. P. Blumer, "Verification of NBS Class 4 transport protocol," *IEEE Trans. Software Eng.*, vol. COM-34, pp. 781-789, Aug. 1986.

[21] B. Sarikaya and G. V. Bochmann, "Synchronization and specification issues in protocol testing," *IEEE Trans. Commun.*, vol. COM-32, pp. 389-395, Apr. 1984.

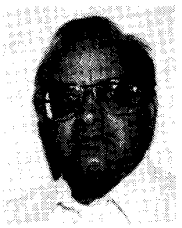
[22] M. U. Uyar and A. T. Dahbura, "Optimal test sequence generation for protocols: The Chinese postman algorithm applied to Q.931," in *Proc. IEEE Global Telecommun. Conf.*, 1986.

[23] A. Dahbura and K. Sabnani, "Experience in estimating fault coverage of a protocol test," in *Proc. IEEE INFOCOM'88*, 1988, pp. 71-79.

[24] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[25] D. P. Sidhu and T. K. Leung, "Experience with test generation for real protocols," in *Proc. SIGCOMM '88 Symp.: Communication Architectures and Protocols*, 1988, pp. 257-261.

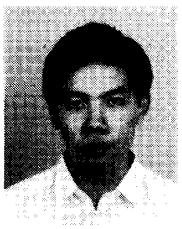
[26] ——, "Formal methods for protocol testing—A detailed study," ISU Preprint, Tech. Rep. 86-23, 1986.



Deepinder P. Sidhu (SM'84) received the B.S. degree in electrical engineering from the University of Kansas, Lawrence, and the M.S. degree in computer sciences and the Ph.D. degree in theoretical physics, both from the State University of New York, Stony Brook.

From 1973 to 1975, he was with Rutgers University as Research Associate in the Department of Physics. From 1975 to 1980, he was with the Brookhaven National Laboratory, Upton, NY, as Assistant and Associate Physicist where he worked in the areas of gauge theories of fundamental particles and computer networks and communication protocols. His contributions to the determination of weak neutral current couplings of quarks were cited by Steve Weinberg in his 1979 Nobel Prize acceptance speech. From 1980 to 1982, he was a member of Technical Staff with The Mitre Corporation, Bedford, MA, where he worked on problems in the areas of computer and communication security, operating systems, communication protocols, and architecture of distributed systems. From 1982 to 1984, he was manager of the Secure and Distributed Systems department within the Research and Development Division of SDC-Burroughs in Paoli, PA, where he had responsibility for the hardware development for SDC's MIL/INT local area network and implementation of NBS Class 4 transport protocol, DoD TCP/IP, IEEE 802.2 Logical Level Control Protocol (LLC). While with SDC-Burroughs, he was an adjunct Professor with Villanova University and Drexel University and taught courses at these universities. From 1984 to 1988, he was with Iowa State University as Associate Professor and then as Full Professor of Computer Science. In 1988, he joined the Department of Computer Science of the University of Maryland—Baltimore County and the University of Maryland Institute for Advanced Computer Studies at College Park as Professor of Computer Science. His research interests include computer networks, distributed computing, software engineering, artificial intelligence, and computer and communication security. He has authored or coauthored over 75 papers in the areas of theoretical physics and computer science. He is a co-founder of Protocol Development Corporation (PDC) which created an automated system for the implementation, verification, and testing of communication protocols based on the ISO standard Estelle specification technique. PDC is now a subsidiary of Pheonix Technologies.

Dr. Sidhu is a member of the Association for Computing Machinery and the IEEE Computer Society. He has chaired sessions at several conferences and workshops and served on program committees for conferences.



Ting-Kau Leung received the B.S. and M.S. degrees in computer science from Iowa State University in 1984 and 1986, respectively.

In 1987, he joined Nixdorf Computer Limited, Hong Kong, where he is currently working as a System Engineer.

Mr. Leung is a member of Upsilon Pi Epsilon.