On Testing Hierarchies for Protocols

Deepinder P. Sidhu, *Senior Member, IEEE,* Howard Motteler, *Member, IEEE,* and Raghu Vallurupalli

***Abstract*— Consider a protocol specification represented as a fully specified Mealy automata, and the problem of testing an implementation for conformance to such a specification. No single sequence-based test can be completely reliable, if we allow for the possibility of an implementation with an unknown number of extra states. We define a hierarchy of test sequences, parameterized by length of behaviors under test. For the reset method of conformance testing, we prove that the hierarchy has the property that any fault detected by test i is also detected by test $i+1$, and show that this sequence of tests converges to a reliable conformance test. For certain bridge sequence methods for constructing test sequences, this result does not always hold. In experiments with several specifications, we observe that given a small number of extra states in an implementation, our sequence of tests converge to total fault coverage for small values of i, for both reset and bridge sequence methods. We also observe that choice of characterizing sequence has less effect on fault coverage than choice of behavior length or number of extra states in the implementation.**

***Index Terms*— Protocols, conformance testing, testing hierarchies, optimal length test sequences, faults, fault coverage**

I. Introduction

PROTOCOLS are rules and conventions by which network entities communicate. Formal description techniques have been used successfully in eliminating ambiguity and incompleteness in protocol specifications. Formal specifications can sometimes still be subject to errors of interpretation, and implementations may contain programming errors. This leads to the need for a way to test a protocol implementation for conformance to its specification.

The control structure of a communications protocol can be described by a deterministic Mealy automaton, or finite state machine (FSM). This is an automaton where state transitions have both an input and an output. In a *completely specified* FSM, at each state there is an out edge for each symbol of the input alphabet; a *partially specified* FSM is an FSM that is not completely specified. The completely specified FSM acts as a total function from strings of its input alphabet to strings of its output alphabet, while the partially specified FSM acts as a partial function.

Manuscript received November 26, 1991; revised August 2, 1993; transfered from IEEE Transactions on Software Engineering by IEEE/ACM Transactions on Networking Editor Simon S. Lam. This research was supported in part by the Department of Defense at the University of Maryland Baltimore County.

The authors are with the Maryland Center for Telecommunications Research, Department of Computer Science, University of Maryland-Baltimore County, Baltimore, MD 21228; and the Institute for Advanced Computer Studies, University of Maryland-College Park, College Park, MD 20742 (email: dsidhu@umbc3.umbc.edu) (email: motteler@umbc.edu) (email: raghu@cs.umbc.edu).

IEEE Log Number 9214670.

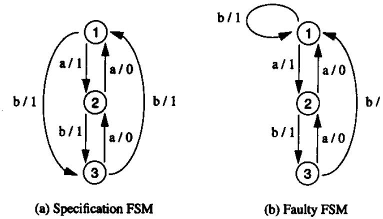


Fig. 1. An example FSM.

In this paper, we consider only completely specified machines. A partial specification can easily be extended to a complete specification: for each unspecified input at a state, we add a self-loop with that input and the empty string as output. Alternatively, an unspecified input can lead to an error state.

Protocol testing is usually carried out by applying a test sequence to the implementation under test (IUT) [5], [7]. The test sequence is constructed in some fashion from the specification of the protocol. The construction of a protocol test sequence involves many choices, many of which are not guided by any formal theory [8].

A *fault* is a difference between the behavior of the implementation and the behavior of the specification. If a test finds a fault we know the implementation is faulty. However, a particular test sequence may not find a particular fault. When this happens we say the fault is *masked.* The *fault coverage* of a test sequence is a measure of its expected reliability in detecting faults, and is often expressed as a percentage of faulty machines that are detected.

Fig. 1 gives an example of a completely specified FSM. This machine has three states $\{1,2,3\}$, two inputs $\{a,b\}$ and two outputs $\{0,1\}$. In later discussion of fault coverage, we also consider a corresponding faulty FSM shown in Fig. 1. The faulty FSM has an error in the next state of a transition, i.e., the transition (1,3;b/1) has become (1,1;b/1).

Much recent research has focused on "optimizing" protocol test sequences; i.e., finding a shortest test sequence that checks a given set of partial behaviors. Such optimization methods save at most about a factor of two in test sequence length, while sometimes paying a considerable price in fault coverage [9]. In this paper we reverse this emphasis, and consider how long (both in theory and in practice) a test sequence needs to be to give complete fault coverage.

In much of the testing literature, it is assumed that the implementation has the same number of states as the specification [10], [11]. We consider a more realistic version of the conformance testing problem, where we allow for the possibility of an unspecified number of extra states in an

1063-6692/93$03.00 © 1993 IEEE

implementation. An implementation can easily have extra states. For example, a specification state might be implemented as an equivalence class of internal states, one for each value of a local variable. Such a nonminimal implementation might still be correct, but if there are errors, then they must be found in the context of the larger state set. In general, extra states allow more forms of fault masking [12], i.e., more ways in which faults can escape detection.

The time complexity of determining conformance is exponential in the number of states in the implementation. If we do not have a bound on this number, then the problem is undecidable. Because of this any testing method using a single test sequence is at best an approximation. We define a hierarchy $(\beta_0, \beta_1, \beta_2, \ldots)$ of test sequences, parameterized by length of behaviors under test. For the reset method (defined in Section II), we prove that any fault detected by a β_i test is also detected by a β_{i+1} test. This hierarchy of tests defines a corresponding chain of implementations which converges to a set of machines that are equivalent to the specification. For bridge sequence methods (also defined in Section II) this is not always true.

In experiments, we observe that given a small number of extra states in an implementation, for both RCP and reset methods, our sequence of tests converge to total fault coverage for small values of i. We also observe that choice of characterizing sequence has less effect on fault coverage than choice of behavior length or number of extra states in the implementation.

The paper is organized as follows. Section II gives background and definitions concerning test methods based on *characterizing sequences*, and considers the relationship between fault coverage of test sequences and subsequences. In Section III, properties of the testing hierarchy are analyzed, and in Section IV, we present experimental measurements of the fault coverage of a number of test sequences. Section V contains a brief summary and conclusions.

II. CS-BASED TEST METHODS

Our emphasis is on characterizing sequence-based test methods. A *characterizing sequence* (CS) for a protocol FSM is a sequence of inputs and outputs which exhibit some distinctive signature for each state of the FSM. In CS-based test methods, a test sequence is formed by joining test subsequences with some form of bridging sequences. The individual subsequences consist of an edge or sequence of edges under test, followed by a characterizing sequence for the tail state of the last edge in the sequence of edge(s) under test. The CS is used to check that the behavior sequence is being applied at the intended place in the implementation.

A. Characterizing Sequences

A number of different characterizing sequences have been proposed. In this section we consider *unique input output* sequences (UIO's), and *distinguishing sequences* (DS). Many other sorts of characterizing sequences have been proposed, including and W-sets [17], pairwise distinguishing sequences [13], and the characterizing sequences of the UIOv [14] and BUIO [15] test methods.

TABLE I
CHARACTERIZING SEQUENCES

State	UIO_1	UIO_2	DS
1	a/1	a/1	a/1.a/0
2	a/0.a/1	a/0.a/1	a/0.a/1
3	a/0.a/0	b/1.a/1	a/0.a/0

1. UIO's: A UIO sequence for a state of an FSM M is an input/output (I/O) behavior that is not exhibited by any other state of M [5]. It is usually possible to generate a UIO sequence for a strongly-connected, minimal and completely specified Mealy machine. A set of UIO's are produced, for the states of M, by constructing multiple trees rooted at each state using a breadth-first procedure. The specification FSM of Fig. 1 has two sets of minimum length UIO sequences, shown in Table I.

2. Distinguishing Sequences: An input sequence is a DS of an FSM M if the output string produced by M, in response to the input string, is different for each state of M. It is not always possible to find a DS for a particular FSM. A DS can be generated by constructing a *distinguishing tree* [16]. For the example FSM of Fig. 1, the shortest distinguishing sequence (DS) is aa, as shown in Table I.

B. Test Subsequences

A test subsequence can be defined most generally as

$$\mathcal{L}_{p,q} \cdot \mathrm{CS}(q)$$

where $\mathcal{L}_{p,q} = L_1 \cdot L_2 \cdot \ldots \cdot L_\lambda$ is a sequence of λ input/output labels L_i that take the protocol FSM from state p to state q. (We use $\cdot$ to denote string concatenation.) This is the "$\mathcal{L}$-sequence" or behavior sequence to be tested. $\mathrm{CS}(q)$ is the characterizing sequence for state q, for example a UIO or DS.

We classify test subsequences based on the length of the $\mathcal{L}$-sequence. A β_λ test subsequence for a given specification and set of characterizing sequences is an $\mathcal{L}$-sequence of length λ derived from the specification, concatenated with the characterizing sequence for the tail state of the $\mathcal{L}$-sequence. The β_0 subsequences have an empty $\mathcal{L}$-sequence, and test the states of an implementation. The β_1 subsequences test a single transition in an implementation, the β_2 subsequences test pairs of transitions of an implementation, and so on. For an FSM with n states and k inputs, there are nk, λ behaviors of length λ.

If a specification has at least one input transition for each state, then for a given CS-method and for all $i \geq 0$, every β_i test subsequence is a proper subsequence of some β_{i+1} test subsequence. This follows since the assumed input transition allows any β_i subsequence to be extended one edge to the left, forming a β_{i+1} test subsequence.

Before we consider methods for joining test sequences, it is interesting to note that a sufficiently long single behavior can be treated as a test sequence. In [13] a test method is proposed, which we would characterize as a single β_λ test subsequence, with $\lambda = |E|$ and no edges repeated along the behavior.

TABLE II

β_1 TEST SUBSEQUENCES AND SEQUENCES BUILT USING RCP METHOD FOR FSM IN FIG. 1 USING UIO_1, UIO_2, AND DS AS CHARACTERIZING SEQUENCES

CS	β_1 Subsequences	β_1 Test Sequence	Length
UIO_1	$[a/1.a/0.a/1], [b/1.a/0.a/0]$ $[a/0.a/1], [b/1.a/0.a/0]$ $[b/1.a/1], [a/0.a/0.a/1]$	$b/1.a/0.a/0.a/1.a/0.a/1.b/1$ $b/1.a/1.b/1.a/0.a/0.a/1.b/1.a/0.a/0$	18
UIO_2	$[a/1.a/0.a/1], [b/1.b/1.a/1]$ $[a/0.a/1], [b/1.b/1.a/1]$ $[b/1.a/1], [a/0.a/0.a/1]$	$b/1.b/1.a/1.a/0.a/1.a/0a/1.a/0.a/1.b/1$ $b/1.a/1.b/1.b/1.a/1.b/1.a/0.a/0.a/1.a/0$	20
DS	$[a/1.a/0.a/1], [b/1.a/0.a/0]$ $[a/0.a/1.a/0], [b/1.a/0.a/0]$ $[b/1.a/1.a/0], [a/0.a/0.a/1]$	$b/1.a/0.a/0.a/1.a/0.a/1.a/0.a/1.a/0.b/1$ $a/0.a/0.a/1.b/1.a/0.a/0.b/1.b/1.a/1.a/0$	20

C. Test Sequences

Fixing a set of characterizing sequences and an $\mathcal{L}$-sequence length defines a set of test subsequences. These must still be combined into a single test sequence. Two schemes for constructing test sequences are discussed in the following sections.

1. Reset Transition Method: This method is applicable when the protocol FSM has a reset transition [5], [7]. The method constructs a test subsequence for each $\mathcal{L}$-sequence by concatenating a shortest path from the initial state of the FSM to the start state of $\mathcal{L}$-sequence, the $\mathcal{L}$-sequence itself, and a characterizing sequence for the tail state of $\mathcal{L}$-sequence. Any test subsequence which is a prefix of another is omitted, and the resulting set of subsequences is concatenated with reset transitions to form a test sequence.

2. Bridge Sequence Methods: Bridge sequence methods construct an executable test sequence by concatenating a set of test subsequences using bridge sequences. A *bridge sequence* is a sequence of inputs and outputs along a path in the protocol FSM between a pair of states. The bridge sequences are used to provide a connection from the tail state of one test subsequence to the head state of another test subsequence. The optimization procedures for this method lead to solving the Rural Chinese Postman (RCP) problem and bipartite matching problems.

The *RCP method* involves finding a minimum cost tour of a graph involving a selected set of edges. A protocol FSM can be represented by a directed graph $G = (V, E)$ where V is the set of vertices representing the states of the FSM and E is the set of edges representing the transitions of the FSM. An edge from vertex v_i to vertex v_j labeled with L_l is represented by $(v_i, v_j; L_l)$ where L_l is the input/output label associated with the transition. The characterizing sequence applied to state v_i is indicated by $\text{CS}(v_i)$ and the tail (last) state of $\text{CS}(v_i)$ by $\text{TAIL}(\text{CS}(v_i))$. From the directed graph $G = (V, E)$ construct a new directed graph $G'_{\lambda,\text{CS}} = (V', E')$ such that $V' \equiv V$ and $E' \equiv E_C$, where E_C corresponds to a set of edges which correspond to test subsequences, defined as follows:

$$\begin{aligned} E_C = \{&(v_i, v_k; \mathcal{L}_{i,j} \cdot \text{CS}(v_j) : \\ &\mathcal{L}_{i,j} = L_{l_1} \cdot L_{l_2} \cdot \ldots L_{l_\lambda}, \\ &\exists v_{p_1}, \ldots, v_{p_{\lambda-1}} \in V \text{ such that} \\ &(v_i, v_{p_1}, L_{l_1}), (v_{p_1}, v_{p_2}, L_{l_2}), \ldots, (v_{p_{\lambda-1}}, v_j, L_{l_\lambda}) \in E \\ &\quad \text{and } \text{TAIL}(\text{CS}(v_j)) = v_k\}. \end{aligned}$$

The Rural Chinese Postman Problem involves finding a minimum cost tour on graph $G'_{\lambda,\text{CS}}$ such that each edge in E_C is visited at least once. This problem has a polynomial-time solution if the edge-induced subgraph $G[E_C]$ forms a weakly-connected spanning graph of $G'_{\lambda,\text{CS}}$.

An optimal length test sequence can be constructed for a protocol FSM, modeled as a directed graph G, from a set of test subsequences using the Rural Chinese Postman tour [18] as follows:

1) For every test subsequence, identify the starting vertex v_j and ending vertex v_k in graph G.
2) Construct a graph $G'_{\lambda,\text{CS}}$ from graph G by adding an edge $(v_i, v_k; \mathcal{L}_{i,j} \cdot \text{CS}(v_j))$ from vertex v_i to vertex v_k for each test subsequence where v_k is the state reached on applying characterizing sequence CS to state v_j.
3) Construct an augmented graph $\hat{G}^*_{\lambda,\text{CS}} = (\hat{V}^*, \hat{E}^*)$ from graph $G'_{\lambda,\text{CS}}$ (where $\hat{V}^* = V'$ and $\hat{E}^*$ is initialized to E') by adding edges from E to $\hat{E}^*$ such that every vertex in $\hat{V}^*$ is symmetric, i.e., a vertex in-degree is equal to its out-degree.

This symmetric augmentation can be reduced to a polynomial-time min-cost/max-flow problem if the edge-induced graph $G[E_C]$ is weakly-connected. The cost associated with an edge $(v_i, v_k; \mathcal{L}_{i,j} \cdot \text{CS}(v_j)) \in E_C$ is the sum of the costs of edges labeled $L_{l_1}, L_{l_2}, \ldots, L_{l_\lambda}$, and the costs of all edges in $\text{CS}(v_j)$. Traversing an edge $(v_i, v_k; \mathcal{L}_{i,j} \cdot \text{CS}(v_j)) \in E_C$ corresponds to traversing a β_λ subsequence $\mathcal{L}_{i,j}$. Therefore, the minimum-cost test sequence, which contains all test subsequences such that no two subsequences are overlapped, corresponds to a minimum-cost tour of $G'_{\lambda,\text{CS}}$ such that each edge in E_C is traversed at least once. An optimal length test sequence (or a Rural Chinese Postman tour) corresponds to basically an Euler tour of the augmented graph $\hat{G}^*_{\lambda,\text{CS}}$.

In [18], it is demonstrated that if certain sufficient conditions are met for a protocol FSM, an efficient method exists for constructing optimal length test sequences based on the Rural Chinese Postman Problem. In [19], it is shown that an efficient method exists for test generation under more general (i.e., weaker) sufficient conditions on a protocol FSM. A wide class of protocols satisfy these new sufficient conditions.

Table II shows optimal length test sequences constructed from β_1 subsequences using the RCP method for the FSM in Fig. 1. The table shows only one optimal test sequence constructed from a set of test subsequences; several test sequences of the same length are possible.

D. Fault Detection in Test Sequences and Subsequences

A test sequence is constructed from a set of test subsequences concatenated with bridging sequences. Faults detected in a test sequence may or may not be detected in test subsequences, and conversely, faults detected in subsequences may or may not be detected in test sequences. We have found instances of each of the following:

1) *Test subsequences detect a fault and the test sequence detects a fault.* The β_1 subsequences using DS as Characterizing Sequences, shown in Table II detect the faulty FSM shown in Fig. 1 and the corresponding test sequence shown in Table II also detect the same fault.
2) *Test subsequences detect a fault but the test sequence does not detect a fault.* An example of this using the RCP method is given in [12].
3) *Test subsequences do not detect a fault and the test sequence does not detect a fault.* Neither the β_1 subsequences using UIO_2 as Characterizing Sequences, shown in Table II detect the faulty FSM shown in Fig. 1 nor the corresponding test sequence shown in Table II detect the same fault.
4) *Test subsequences do not detect a fault but the test sequence detects a fault.* The following test sequence detects the fault in the faulty FSM shown in Fig. 1. This test sequence is built using β_1 subsequences using UIO_2 as Characterizing Sequences, shown in Table II which does not detect the fault.

$$[b/1.b/1.a/1][a/0.a/1][b/1.b/1.a/1][b/1][b/1.a/1]$$
$$[a/0][a/1.a/0.a/1][a/0][b/1][a/0.a/0.a/1][a/0]$$

This test sequence is obtained by a nonoptimal symmetric augmentation of $G'_{\lambda,\mathrm{CS}}$ as explained in Step 3 of the RCP method of constructing test sequence described in Section II-C-2.

III. Testing Hierarchies

As noted, determining conformance is a formally intractable problem, and a test sequence can be viewed as an approximation to a complete conformance test. Our goal in this section is to define a sequence of tests of increasing length that converge to a complete conformance test. Ordering tests based on the length of behaviors to be tested (β_0, β_1, etc.) gives a hierarchy of partial behaviors, but does not necessarily give a hierarchy of test sequences. In [12] an example is given of a test sequence generated with the RCP method, where a fault is detected in β_1 testing but not in β_2 testing.

A Testing Hierarchy for the Reset Method

We introduce a slight variation of reset method testing, with the property that a β_{i+1} test sequence detects at least as many faults as a β_i test sequence, and where there exists a value j such that a β_j test sequence detects all faults.

In the context of testing with the reset method, we have the following definitions. A β_i *reset test subsequence* is the concatenation of a *preamble* (a shortest path from the initial state to state p), an $\mathcal{L}$-sequence $\mathcal{L}_{p,q}$ of length i, and a characterizing sequence for state q. Fig. 2 shows a typical reset test subsequence. A β_i *reset test sequence* is formed by omitting subsequences that are proper prefixes of others, and concatenating the resulting set of sequences with reset transitions.

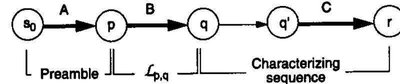


Fig. 2. A test subsequence for reset method. A, B, and C are label sequences of length 0 or more.

In our variation of the reset method, we assume that the IUT can be reliably returned to the start state from any state. This assumption can be satisfied in several ways: (1) We could assume the implementation of reset is correct. (2) The testing environment could have has some sort of 'meta-reset' (i.e., some way to return the IUT to its start state, without sending it a reset input). (3) The tester could perform a separate test for each reset test subsequence.

There are several consequences to assuming forced resets. One consequence is that the order in which test subsequences are presented does not matter. Another consequence of enforced resets is that the tester has more confidence that the edge sequence being traversed in the implementation corresponds to the intended sequence in the specification. This is of particular importance when the implementation may have extra states.

We make two further "technical" assumptions for the results of this section. The first assumption concerns shortest paths from the initial state to the first state q of an $\mathcal{L}$-sequence. Suppose p is a state on a preamble leading to state q. We assume the preamble leading to p is a prefix of the preamble leading to q. This would be the case, for example, if we chose as our preambles the lexically least shortest paths. The second assumption is that the start state has some self loop that, even though it is tested, is implemented correctly. This would be the case if, for example, we include reset transitions in our $\mathcal{L}$-sequences. We refer to reset testing that satisfies these two assumptions, together with reliable reset as *restricted reset testing*. In practice, we do not feel that these are significant restrictions.

State names from the specification are used to refer to points along a test sequence or subsequence. It is important to keep in mind that these states may or may not correspond to the same states in the implementation, when the label sequence from the specification is followed in the implementation. For example, two separate paths to the same state in the specification might lead to distinct states in an implementation; this is particularly likely if the implementation has extra states.

In subsequent discussion, the phrase "where the fault is detected" refers to the first place in a test subsequence where an edge output value is different from the expected value. Due to ways that faults can interact and appear at more than one place in a path, this may be later (perhaps considerably later) than the first instance of a faulty edge along a path [12].

We now prove three theorems that together show that given the above assumptions, the β_i reset test sequences form a hierarchy that converges in the limit to complete fault coverage.

Theorem 1: If a fault is detected by a β_i restricted reset test sequence, then it is detected by a β_{i+1} restricted reset test sequence.

Proof: Due to various forms of fault masking, it is not enough to show that every edge visited in a β_i test sequence will also be visited in a β_{i+1} reset test sequence, or even that every $\mathcal{L}$-sequence of a β_i test sequence is subsumed by an $\mathcal{L}$-sequence of a β_{i+1} test sequence. The key idea in the proof is that any path from the initial state in the implementation which is traversed in a β_i test subsequence will also be traversed in a β_{i+1} test subsequence, up to and including the edge where a fault is detected.

Suppose a β_i reset test sequence detects a fault. Then the fault is detected in at least one of the β_i reset test subsequences, illustrated in Fig. 2. Either the fault is detected in the preamble, from s_0 to p, the $\mathcal{L}$-sequence from p to q, or the characterizing sequence from q to r. Suppose the fault is detected in the preamble. Then the β_{i+1} test subsequence consisting of the original preamble, an $\mathcal{L}$-sequence $\mathcal{L}_{p,t}$ for any state t that is $i+1$ edges distant from p, and the characterizing sequence for t will detect the fault.

Suppose the fault is detected in the $\mathcal{L}$-sequence $\mathcal{L}_{p,q}$. The characterizing sequence has length at least 1, so let q' be the second state of this sequence. Then the β_{i+1} test subsequence consisting of the original preamble, the $\mathcal{L}$-sequence $\mathcal{L}_{p,q'}$, and the characterizing sequence for q' will detect the fault.

Suppose the fault is detected in the characterizing sequence for state q. We must guarantee that the entire path from s_0 through the old preamble and $\mathcal{L}$-sequence is traversed in the β_{i+1} test. Suppose the preamble has length at least 1. We have assumed that the preamble to a state along a preamble will always be a prefix of that preamble. Then we can form a β_{i+1} test subsequence from the β_i test subsequence: the last edge of the preamble becomes the first edge of the $\mathcal{L}$-sequence. Finally, if the preamble is empty, the $\mathcal{L}$-sequence can be extended to the left with the self-loop of the start state.

We have shown that in each case, a path from the start state in the implementation which is traversed in a β_i test sequence, and which uncovers a fault, will also be traversed in a β_{i+1} test sequence, up to and including the edge where a fault is detected. □

A slightly weaker version of the theorem holds if we relax the assumptions that the preamble to a state along a preamble will always be a prefix of that preamble and that the start state has a correctly implemented self loop. The weaker version may be stated as follows: If a fault is detected by a β_i reset test sequence, then for some $j > i$ it is detected by a β_j reset test sequence, where $j \le i + \ell$, where ℓ is the length of the longest characterizing sequence. This follows from extending the $\mathcal{L}$-sequence to the right rather than to the left, in the case where a fault is detected in the characterizing sequence.

Suppose we have a specification FSM, a particular set of characterizing sequences for that FSM, and a particular set of shortest paths to states. These choices uniquely define[1] a β_i reset test sequence, for each $i \ge 0$. For this specification FSM, let B_i be the set of all deterministic FSM's that pass the β_i reset test sequence. From Theorem 1, we have that any machine passing a β_j reset test will also pass a β_i reset test, for $0 \le i < j$. Thus we have a chain $B_0 \supseteq B_1 \supseteq B_2 \ldots$ of implementations. Steps in this chain may be either equality or proper containment. In the next theorem, we show that for any value of i, there is some specification and implementation such that the containment is proper.

[1] Unique up to order of test subsequences, which does not matter for the reset method considered here.

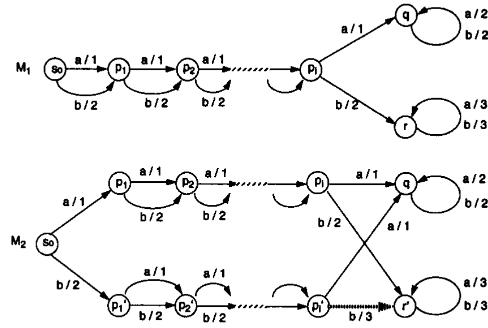


Fig. 3. Figure for Theorem 2.

Theorem 2: For any $i \ge 0$, we can find FSM's M_1 and M_2 such that the difference between the machines can be detected by a β_{i+1} reset test derived from M_1, but not by a β_i reset test derived from M_1.

Proof: The proof is by construction, which is given in Fig. 3. (Reset edges are omitted to simplify the illustration, and we assume that reset edges and the edge where the fault is introduced are not part of any characterizing sequence.) The idea is that M_1 and M_2 are *almost* equivalent as automata, with i states of M_1 being split into i pairs of states in M_2; a single edge label at the end of one of the paths in M_2 differs from the corresponding edge label in M_1. The only tests which can catch this fault are those that choose the b branch from s_0, while if we assume lexically least shortest paths, all preambles will travel on the a branch from s_0. This leaves only $\mathcal{L}$-sequences with empty preambles to catch the fault. The fault can be caught only by an $\mathcal{L}$-sequence long enough to reach from the initial state s_0 to the fault. □

In practice, given a specification FSM, for small values of i it is easy to find faulty implementations missed by a β_i test and caught by a β_{i+1} test. (Almost any row in the tables of Section V contains an example.) Theorem 2 shows that there are cases where this can hold for arbitrarily large values of i, and also illustrates how extra states can give rise to faults that are hard to detect.

In the next theorem, we show that for any faulty implementation, there is some i such that a β_i reset test sequence derived from the specification detects the fault. This implies that the chain $B_0 \supseteq B_1 \supseteq B_2 \ldots$ converges to a set of machines that are equivalent to the specification.

Theorem 3: Let M_1 and M_2 be any two completely specified FSM's, and let n be the larger of their number of states. Then a β_n reset test sequence derived from M_1 will detect if M_2 is equivalent to M_1.

Proof: Let T_i be the tree of labeled paths from the start state of M_i, for $i \in \{1,2\}$. Nodes at depth d in T_i are exactly

the states reachable from the start state in M_i in d steps. For completely specified machines (which we are assuming) T_i is infinite; its nodes have state labels, and each node has one descendant for each input symbol.

Let n_i be the number of states in M_i. Then T_i is completely determined by that finite portion no deeper than n_i, as every state (together with its connections to its neighbors) appears in T_i. Then M_1 and M_2 are equivalent iff T_1 and T_2 are equivalent up to depth $n = \max(n_1, n_2)$. Those β_n reset test subsequences with empty preamble will travel every path in the larger of the two trees, and so will determine equivalence. □

The theorem depends only on those test subsequences with empty preamble. This suggests yet another test method: for specification M_i, simply test behaviors corresponding to successive levels of T_i. This gives an exponential time bound, and an immediate hierarchy theorem: faults detected at level j will be caught at all levels greater than j. The difficulty with this as a practical test method is that its behavior would normally be very bad for small values of j: until j was equal to n, some edges might not be tested at all. The reset testing hierarchy performs much better for small values of j.

Although we have an exponential bound on the complexity of testing, the bound is in terms of the number of states of the implementation, which in general may not be known. In much of the testing literature, it has been assumed that the implementation has the same number of states as the specification. Only if we have a bound on the the number of states in the implementation do we have any bound on the length of a test sequence needed for complete fault coverage.

The testing hierarchy we have defined reflects a direct tradeoff between reliability and run time of a test. In Section V, we will consider some particular specification machines, and see that in practice, for small numbers of extra states, β_i testing for small values of i gives complete fault coverage.

B. Testing Hierarchies and Other Test Methods

We briefly consider why the theorems of the previous section do not seem to hold for bridge sequence methods. Temporarily define a "β_i bridge test sequence" as a test sequence derived from some fully specified bridge sequence test method.

With the reset test method, we showed that any path from the initial state in the implementation that is traversed in a β_i test sequence is also traversed in a β_{i+1} test sequence, up to and including the edge where a fault is detected. With a bridge sequence method, the corresponding assertion would be that the entire β_i test sequence, beginning at the initial state, is also traversed in a β_{i+1} test sequence, up to and including the edge where a fault is detected. The only way this can be guaranteed in general is if the β_i bridge test sequence is a prefix of the β_{i+1} bridge test sequence, and this is not true in general of bridge sequence methods.

Of course, one could simply *define* a class of bridge sequence test methods with the desired property, by concatenating a β_{i+1} test sequence to the end of a β_i test sequence. The problem with this is that the whole point of bridge sequence methods is to reduce overall test sequence length, and concatenating a β_{i+1} test with β_0 through β_i tests defeats this purpose.

IV. Fault Coverage Of Test Sequences

The *fault coverage* of a test sequence is a measure of its expected ability to detect faults. Although we can give an exact measure of fault coverage for a particular specification and set of faulty implementations, a more general notion of fault coverage must remain informal, as the space of all "reasonable" specifications and all "reasonable" faulty implementations is not well defined. In this section we use several representative specifications and what are intended to be plausible faulty implementations, and measure the fault coverage for a number of the test methods we have discussed.

A. Estimating Fault Coverage

We generate a set of plausible faulty implementations with a simple mutation scheme. We define a group of machines that are "close" in some sense to the specification, but are still faulty. First, as an optional step, one or more randomly selected states are split into two states. In-edges of the original state are divided randomly between the two new states, and both new states have the same set of out-edges. The state split results in a machine that is equivalent but not minimal.

After the optional state split, one or more edges of the specification are selected at random, and changes are made in accordance with one of the 10 following mutation types [7]. This set of fault types is intended to capture the necessarily informal notion of implementation that are "slightly wrong."

1) The output of an edge is changed.
2) The tail state of as edge is changed.
3) The outputs of two different edges are changed.
4) The tail states of two different edges are changed.
5) The output one edge and tail state of another edge are changed.
6) The output and tail state of an edge are changed.
7) The outputs and tail states of two edges are changed.
8) The outputs of two edges and tail state of another edge are changed.
9) The outputs of two edges and the tail states of another two edges are changed.
10) The outputs of three edges and the tail states of another two edges are changed.

Given this mutation scheme, fault coverage for a test sequence is estimated as follows. The test sequence is applied to a large number of mutated implementations from a particular fault class. If no faults are detected, an additional automata equivalence test is applied, to determine if the mutation is in fact equivalent to the specification. The extra test is needed, as it is possible that a pair of mutations might cancel, or might result in an implementation that is equivalent as an automata. (The automata equivalence algorithm cannot be used for conformance testing, because we cannot assume that the state transition diagram of the implementation is available.)

If r is the total number of machines generated, d is the number found faulty with the test sequence, and c is the

number of machines found equivalent with the automata equivalence algorithm, then our measured fault coverage is $d/(r-c)$.

B. Experiments

In this section, we measure the fault coverage of a number of test sequences on the example machine of Fig. 1 and on the NBS TP4 and IEEE 802.2 LLC protocols. We also consider test sequence lengths for several larger protocol FSM's.

1. Experiment 1: For this experiment we take the specification FSM of Fig. 1, and generate a large number of faulty machines, including machines with extra states. We then measure fault coverage of test sequences formed with behaviors ranging from β_0 to β_4, using a DS characterizing sequence and also UIO_1 and UIO_2. In this experiment, the RCP method is used for joining test subsequences.

We generate faulty machines with 0, 1, and 2 extra states. Extra states are generated by splitting states to get an equivalent machine. A state is selected at random, and then the in-edges to the state are divided randomly between the two split states. Both the states retain the out-edges of the original state. A set of 100 000 faulty machines is generated with zero, one, and two extra states, for each of our ten fault classes. (In those cases where the number of mutants is small, the large number of mutants simply guarantee that each mutant appears approximately the same number of times in the sample set.) In the experiments with extra states, we first generate 100 FSM's, with either one or two extra states, equivalent to the specification. Each of these machines is then mutated 1000 times, in accordance with our ten fault classes.

In this particular experiment, the following results were observed. In general, extra states in the implementation decrease fault coverage, and longer behaviors increase fault coverage. In almost all cases, total fault coverage is above 99% when the behavior length is greater than the number of extra states. Choice of behavior length has a greater effect on fault coverage than choice of characterizing sequence; however characterizing sequence has some effect. The DS had consistently better fault coverage than the UIO's for short behaviors (lengths 1 and 2), with less differences noticeable for longer behaviors.

In a number of cases, fault coverage temporarily decreases with increasing behavior length.

For example, using a DS characterizing sequence, and considering implementations with one extra state, fault coverage either remains the same or becomes worse as we go from behavior sequences of length 2 to length 3, for every fault class. Also, for class 3 faults with two extra states, fault coverage decreases as we go from behavior sequences of length 3 to length 4. Using UIO_1 as a characterizing sequence for every fault class except 3, and for both one and two extra states, fault coverage decreases as we go from behaviors of length 3 to behaviors of length 4. In the experiment with UIO_2, fault coverage never decreases with increasing behavior length.

Test sequences for this experiment are formed using the RCP method, and the "bumps" (temporary worsening of fault coverage with increasing behavior lengths) show that Theorem 1 does not hold for the RCP method. Theorem 1 guarantees that such bumps will not occur when using the reset method.

2. Experiment 2: In this experiment, we use the FSM derived from modeling the control portion of the IEEE 802.2 Logical Link Control (LLC) protocol [20]. This FSM has six states, seven inputs, and 16 outputs. We assume the FSM has a reset transition from every state to the initial state, and that it is extended to a completely specified FSM, by adding self loops with null outputs for unspecified transitions.

A set of 1 000 000 faulty machines is generated with zero, one, two, and three extra states, for each of our ten fault classes. In the experiments with extra states, we first generate 100 FSM's with extra states, which are equivalent to the specification FSM. Each of these machines is then mutated 10 000 times, in accordance with our ten fault classes. The test sequences are generated using the reset transition method, with a UIO characterizing sequence, and testing behaviors of lengths 1, 2, and 3.

Results from this experiment are given in Table III. As before, rows represent a particular set of 1 000 000 faulty machines, and columns a particular test sequence, listed by behavior length. Entries of the table are fault coverage, represented both as a percentage and with the actual number of faulty machines that escape detection.

In this experiment, we see that as in experiment 1, extra states in the implementation decrease fault coverage, and longer behaviors increase fault coverage. As predicted by Theorem 1, there is no case where fault coverage decreases with increasing behavior length. In this experiment, all faulty machines are detected when the behavior length is one more than the number of extra states. In Theorem 3, we showed that for any particular faulty implementation M, there is some i such that a β_i reset test will detect the fault, where i is the larger of the number of states in the specification and faulty implementation. Thus if we have any set S of faulty machines, where no machine in S has more than j states, then a β_j reset test will detect all faulty machines in S. Theorem 3 guarantees that all faulty machines we generate will be detected for j equal to 6, 7, 8, or 9, depending on whether the set of faulty machines has zero, one, two, or three extra states. This is an upper bound, and in this experiment we are seeing all faulty machines detected with much shorter test sequences.

3. Experiment 3: In this experiment, we use the FSM derived from modeling the control portion of NBS TP4 protocol [21]. This FSM has seven states, 14 inputs and 24 outputs. We assume the FSM has a reset transition from every state to the initial state, and that it is extended to a completely specified FSM, by adding self loops with null outputs for unspecified transitions. A set of 1 000 000 faulty machines is generated with zero, one, two, and three extra states, for each of our ten fault classes. In the experiments with extra states, we first generate 1000 FSM's with extra states, equivalent to the specification. Each of these machines is then mutated 1000 times, in accordance with our ten fault classes. The test sequences are generated using the reset transition method, with a UIO characterizing sequence, and with behaviors of lengths 1, 2, and 3.

TABLE III
FAULT COVERAGE FOR FSM FOR IEEE 802.2 LLC PROTOCOL, FOR TEST SEQUENCES USING A UIO CHARACTERIZING SEQUENCE

Fault class	Extra states	T_{β_1} %, count	T_{β_2} %, count	T_{β_3} %, count	T_{β_4} %, count
1	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	90.4, 89 990	100.0, 0	100.0, 0	100.0, 0
	2	81.2, 174 311	99.3, 6408	100.0, 0	100.0, 0
	3	73.4, 248 101	98.3, 16 142	99.9, 927	100.0, 0
2	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	89.3, 90 857	100.0, 0	100.0, 0	100.0, 0
	2	79.0, 174 838	99.3, 6100	100.0, 0	100.0, 0
	3	70.0, 249 351	98.3, 14 265	99.9, 699	100.0, 0
3	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	98.1, 18 802	100.0, 0	100.0, 0	100.0, 0
	2	94.5, 54 100	99.9, 1158	100.0, 0	100.0, 0
	3	90.5, 94 720	99.7, 3317	≈ 100.0, 193	100.0, 0
4	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	96.4, 35 396	100.0, 0	100.0, 0	100.0, 0
	2	90.9, 88 504	99.8, 2352	100.0, 0	100.0, 0
	3	85.0, 145 162	99.4, 5972	≈ 100.0, 286	100.0, 0
5	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	97.2, 27 296	100.0, 0	100.0, 0	100.0, 0
	2	92.7, 71 735	99.8, 1823	100.0, 0	100.0, 0
	3	87.8, 120 413	99.5, 4892	≈ 100.0, 238	100.0, 0
6	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	90.4, 95 033	100.0, 0	100.0, 0	100.0, 0
	2	81.1, 184 050	99.3, 6766	100.0, 0	100.0, 0
	3	73.2, 261 927	98.3, 16 869	99.9, 965	100.0, 0
7	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	98.8, 11 540	100.0, 0	100.0, 0	100.0, 0
	2	95.8, 42 116	99.9, 704	100.0, 0	100.0, 0
	3	92.0, 79 737	99.8, 2073	≈ 100.0, 116	100.0, 0
8	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	99.5, 5201	100.0, 0	100.0, 0	100.0, 0
	2	98.0, 20 302	≈ 100.0, 326	100.0, 0	100.0, 0
	3	95.8, 41 481	99.9, 945	≈ 100.0, 47	100.0, 0
9	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	99.9, 1489	100.0, 0	100.0, 0	100.0, 0
	2	99.2, 7544	≈ 100.0, 77	100.0, 0	100.0, 0
	3	98.2, 18 203	≈ 100.0, 254	≈ 100.0, 14	100.0, 0
10	0	100.0, 0	100.0, 0	100.0, 0	100.0, 0
	1	≈ 100.0, 409	100.0, 0	100.0, 0	100.0, 0
	2	99.7, 2820	≈ 100.0, 22	100.0, 0	100.0, 0
	3	99.2, 7994	≈ 100.0, 64	≈ 100.0, 1	100.0, 0

Test sequences are built using the reset method. Each row of a table represents a particular set of 1 000 000 faulty machines, listed by fault class and number of extra states. For implementations with extra states, 10 000 faulty machines are generated for each equivalent machine generated with extra states. Columns of the table represent a particular test sequence, and are listed by behavior length. Entries of the table are fault coverage, represented both as a percentage and with the actual number of faulty machines that escape detection.

Results from this experiment are given in Table IV. As before, rows represent a particular set of 1 000 000 faulty machines, and columns a particular test sequence, listed by behavior length. Entries of the table are fault coverage, represented both as a percentage and with the actual number of faulty machines that escape detection.

As in Experiments 1 and 2, extra states in the implementation decrease fault coverage, and longer behaviors increase fault coverage. As predicted by Theorem 1, there is no case where fault coverage decreases with increasing behavior length. As in experiment 2, we are seeing all faulty machines detected using a behavior length of one more than the number of extra states.

4. *Experiment 4:* In this experiment, we consider a number of FSM's, including some much larger machines. For each machine, we build test sequences with behaviors of length one and two, using the RCP method, with UIO characterizing sequences. The lengths of these test sequences are tabulated in Table V. The first machine is that of Fig. 1, while second (Example 2 in the table) is an example used as Example 1 in [9]. The last four machines are randomly generated, and are considerably larger than the other machines. A rough upper bound for the length of a test sequence built with the RCP method with behaviors of length λ can be given as follows. Let u be the length of the longest characterizing sequence, N the number of specification states, and I the size of the input alphabet. Then there are at most NI^{λ} behaviors of length λ. Using N as a bound on bridge sequence length, we have a bound of $NI^{\lambda}(N+\lambda+u)$ on test sequence length. Actual test sequence lengths range from about half the upper bound, for the smaller machines, to two orders of magnitude smaller than the upper bound, for the larger randomly generated machines.

The time for execution of a conformance test is proportional to the length of the test sequence. Since each step of this test should involve at most a small number of machine instructions, in practice, even the the 64 401 step test sequence is not prohibitively long.

C. Summary of Experimental Results

In general, in the examples we have considered, extra states in the implementation decrease fault coverage of a test sequence, and longer behaviors increase fault coverage. In Experiment 1 fault coverage is above 99% when the behavior length is greater than the number of extra states, and in Experiments 2 and 3, fault coverage is complete when behavior length is greater than the number of extra states. In Experiment 1, there were a few cases where fault coverage temporarily decreases with increasing behavior length. In Experiments 2 and 3, as predicted by Theorem 2, there is no case where fault coverage decreases with increasing behavior length.

In Experiment 1, choice of a DS or UIO as characterizing sequence has a small effect on fault coverage, in comparison with choice of behavior length. For short behaviors, DS had consistently better fault coverage than the UIO.

In Experiments 2 and 3, using the reset method, even though the test sequence length increases exponentially in behavior lengths, in practice, for small behavior lengths, the resulting test sequences are not too long to be useful. In Experiment 4, using the RCP method and examining only test sequence lengths, these test sequence lengths were much less than the upper bound.

V. SUMMARY AND CONCLUSIONS

The goal in conformance testing is to determine if a particular protocol implementation meets its formal specification. As a formal problem, the time complexity of determining conformance is exponential in the number of states in the implementation. If we do not have a bound on this number, then the problem is undecidable. The intractability of determining

TABLE IV
FAULT COVERAGE FOR FSM FOR NBS TP4 PROTOCOL, FOR TEST SEQUENCES USING A UIO CHARACTERIZING SEQUENCE

Fault class	Extra states	T_{β_1} %, count	T_{β_2} %, count	T_{β_3} %, count
1	0	100.0, 0	100.0, 0	100.0, 0
	1	90.0, 97 643	100.0, 0	100.0, 0
	2	80.0, 194 987	99.6, 3874	100.0, 0
	3	72.8, 264 344	98.9, 11 041	100.0, 0
2	0	100.0, 0	100.0, 0	100.0, 0
	1	89.7, 87 510	100.0, 0	100.0, 0
	2	79.6, 171 256	99.6, 3173	100.0, 0
	3	72.3, 232 173	98.9, 9163	100.0, 0
3	0	100.0, 0	100.0, 0	100.0, 0
	1	98.5, 15 304	100.0, 0	100.0, 0
	2	95.1, 48 803	99.9, 560	100.0, 0
	3	91.4, 85 546	99.8, 1817	100.0, 0
4	0	100.0, 0	100.0, 0	100.0, 0
	1	96.5, 34 264	100.0, 0	100.0, 0
	2	91.4, 83 355	99.9, 1251	100.0, 0
	3	86.7, 129 435	99.6, 3703	100.0, 0
5	0	100.0, 0	100.0, 0	100.0, 0
	1	97.4, 26 154	100.0, 0	100.0, 0
	2	93.1, 69 126	99.9, 1006	100.0, 0
	3	88.8, 111 709	99.7, 3012	100.0, 0
6	0	100.0, 0	100.0, 0	100.0, 0
	1	89.9, 100 074	100.0, 0	100.0, 0
	2	79.9, 199 906	99.6, 3963	100.0, 0
	3	72.8, 270 781	98.9, 11 314	100.0, 0
7	0	100.0, 0	100.0, 0	100.0, 0
	1	98.9, 11 310	100.0, 0	100.0, 0
	2	95.8, 41 882	≈ 100.0, 411	100.0, 0
	3	92.3, 77 190	99.9, 1330	100.0, 0
8	0	100.0, 0	100.0, 0	100.0, 0
	1	99.6, 4112	100.0, 0	100.0, 0
	2	98.3, 16 629	≈ 100.0, 135	100.0, 0
	3	96.5, 34 794	≈ 100.0, 459	100.0, 0
9	0	100.0, 0	100.0, 0	100.0, 0
	1	99.9, 1135	100.0, 0	100.0, 0
	2	99.4, 5742	≈ 100.0, 38	100.0, 0
	3	98.6, 13 710	≈ 100.0, 108	100.0, 0
10	0	100.0, 0	100.0, 0	100.0, 0
	1	≈ 100.0, 288	100.0, 0	100.0, 0
	2	99.8, 2013	≈ 100.0, 13	100.0, 0
	3	99.4, 5818	≈ 100.0, 28	100.0, 0

Test sequences are built using the reset method. Each row of a table represents a particular set of 1 000 000 faulty machines, listed by fault class and number of extra states. For implementations with extra states, 10 000 faulty machines are generated for each equivalent machine generated with extra states. Columns of the table represent a particular test sequence, and are listed by behavior length. Entries of the table are fault coverage, represented both as a percentage and with the actual number of faulty machines that escape detection.

conformance has lead to a broad range of approaches based on testing.

If we know that an implementation has the same number of states as a specification, and can find distinguishing sequences for all states, we can use a method such as that proposed in [22], to detect all faults. The method proposed there is similar to a β_0 test followed by a β_1 test, with the β_0 phase determining a unique set of state names and bridge sequences for the β_1 phase. The difficulty with this approach is that we may not be able to find distinguishing sequences, and more seriously, in practice we may not know the actual number of states in an implementation.

TABLE V
TEST SEQUENCE LENGTHS FOR RCP METHOD USING UIO'S AS CS

FSM	(N, I, O, T)	$\lambda = 1$	$\lambda = 2$
Example 1	(3, 2, 2, 6)	18 (36)	48 (84)
Example 2	(5, 4, 4, 20)	97 (160)	471 (720)
802.2 LLC	(6, 7, 16, 42)	112 (378)	1071 (2940)
NBS TP4	(7, 14, 17, 98)	227 (882)	4825 (13 720)
FSM.4	(50, 10, 5, 195)	769 (27 000)	3780 (275 000)
FSM.2	(20, 50, 50, 620)	1474 (22 000)	64 401 (1 150 000)
FSM.3	(50, 20, 20, 771)	2503 (53 000)	50 338 (1 080 000)
FSM.1	(50, 50, 50, 916)	2281 (130 000)	35 534 (6 625 000)

N = number of states; I = number of inputs; O = number of outputs; T = number of transitions; λ = length of behaviors tested. The number in the parenthesis is an upper bound on the test sequence length.

We allow for the possibility of an unspecified number of extra states in an implementation and define a hierarchy (β_0, β_1, $\beta_2, \ldots$) of test sequences, parameterized by length of behaviors under test. For the reset method, the hierarchy has the property that any fault detected by β_i reset testing is also detected by β_{i+1} reset testing, and that this sequence of tests converges to total fault coverage. This hierarchy of tests defines a corresponding chain of implementations which converges to a set of machines that are equivalent to the specification. The corresponding result does not hold in general for bridge sequence methods, as was shown by a counterexample given in [12].

In experiments, we have observed that for a small number of extra states in an implementation, our sequence of tests converge to total fault coverage for small values of λ. The value of λ giving convergence is roughly proportional to the number of extra states. This held for both the RCP and reset methods. However, convergence in the case of the reset method was always smooth (as implied by the hierarchy theorems) while in the case of the RCP method, we observed occasional temporary decreases in fault coverage, as the length parameter increased. We have also observed that choice of characterizing sequence had less effect on fault coverage than choice of behavior length or the number of extra states in the implementation. Finally, although overall test sequence length increases as the subsequence length parameter increases, for actual protocols, this growth is considerably less than the upper bound.

ACKNOWLEDGMENT

The authors wish to thank the reviewers for their patience and for their many helpful suggestions.

REFERENCES

[1] International Organization for Standardization (ISO), "Information processing systems—Open systems interconnections—Estelle—A formal description technique based on an extended state transition model," ISO 9074, 1989.

[2] International Organization for Standardization (ISO), "Information processing systems—Open systems interconnections—LOTOS—A formal description technique based on the temporal ordering of observational behaviour," ISO 8807, 1988.

[3] D. Rayner, "OSI conformance testing," *Computer Networks & ISDN Systems*, vol. 14, 1987.

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

[5] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks & ISDN Systems*, vol. 15, 1988.

[6] E. Brinksma, "A theory for the derivation of tests," in *Protocol Specification, Testing, and Verification*, vol. VIII, A. Aggarwal and K. Sabnani, ed. 1988.

[7] D. P. Sidhu and T. K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Trans. Software Engineering*, vol. 15, Apr. 1989.

[8] D. P. Sidhu, "Protocol testing: The first ten years, the next ten years," in *Proc. 10th Int. IFIP Symp. Protocol Specification, Testing and Verification*, Ottawa, Canada, June 1990.

[9] A. Chung, H. Motteler, and D. P. Sidhu, "Effects of optimization on length and fault coverage of protocol test sequences," to appear in *J. High Speed Networks.*

[10] Y.-N. Shen and F. Lombardi. On the detection of faulty extra states in protocol testing by UIO sequences. Submitted for publication, 1991.

[11] M. Yannakakis and D. Lee, "Testing finite state machines," in *Proc. 23rd Ann. ACM Symp. Theory of Computing*, 1991, pp. 476–485.

[12] H. Motteler, A. Chung, and D. P. Sidhu, "Undetected faults in protocol testing," to appear in *IEEE/ACM Trans. Networking.*

[13] R. E. Miller and S. Paul, "Generating minimal length test sequences for conformance testing of communication protocols," in *Proc. IEEE INFOCOM '91*, 1991.

[14] W. Y. L. Chan, S. T. Vuong, and M. Robert Ito, "An improved protocol test generation procedure based on UIO's," in *Proc. ACM SIGCOMM '89 Symp.: Communication Architectures and Protocols*, 1989.

[15] X. Shen, S. Scoggins, and A. Tang, "An improved RCP-method for protocol test generation using backward UIO sequences," Submitted for publication, 1991.

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

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

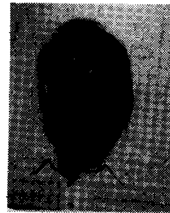
[18] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese Postman Tours," in*Protocol Specification, testing, and Verification*, vol. VIII, A. Aggarwal and K. Sabnani, ed. 1988.

[19] D. P. Sidhu and A. Chung, "On sufficient conditions for an efficient protocol conformance test generation technique based on rural chinese postman problem," Submitted for publication, 1990.

[20] IEEE Standards for Local Area Networks: Logical Link Control, ANSI/IEEE Standard 802.2, 1985.

[21] *Specification of a Transport Protocol for Computer Communications, Vol. 4: Service Specification.* Washington, DCL National Bureau of Standards, 1983, Rep. ICST/HLNP-83-4.

[22] F. C. Hennie, "Fault detecting experiments for sequential circuits," in *Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design*, Princeton, NJ, 1964, pp. 95–110.



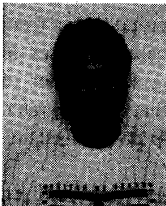
Deepinder P. Sidhu (SM'84/ACM'84) received the B. S. degree in electrical engineering from the University of Kansas, and the M. S. and Ph. D. degrees in computer science and theoretical physics respectively from the State University of New York, Stony Brook.

He is a Professor of Computer Science with the Computer Science Department of the University of Maryland-Baltimore County (UMBC) and the University of Maryland Institute for Advanced Computer Studies (UMIACS) at College Park. He is the Director of the Maryland Center for Telecommunications Research (MCTR) at UMBC. His current research interests are in the areas of computer networks and distributed systems. He has published over 100 papers in theoretical physics and computer science. His contributions to the determination of weak neutral current couplings of quarks, in gauge theories of fundamental particles, were cited by Steve Weinberg in his 1979 Nobel Prize acceptance speech.

Dr. Sidhu is the Editor-in-Chief of *Journal of High Speed Networks.* He was General Chair for SIGCOMM '92 and Program Chair for SIGCOMM '93. He is ACM/SIGCOMM National Lecturer for 1992–95. He is a co-founder of Protocol Development Corp. (acquired by Phoenix Techn. Ltd.) and TeleniX Corp. He managed the Secure and Distributed Systems Department within the R&D Division of SDC-Burroughs (now Unisys). He is the author of a graduate level text, *OSI Conformance Testing*, (Englewood Cliffs, NJ: Prentice Hall, 1994).

Howard Motteler (M'86/ACM'82) received the B. S. from the University of Puget Sound, his M. S. from Purdue University, West Lafayette, IN, and his Ph. D. from the University of Maryland, College Park.

He is an Assistant Professor of Computer Science at University of Maryland Baltimore County, and spent the 1992–1993 academic year at NASA/GSFC as an NRC research associate. His research interests center on parallel and distributed systems, and include scheduling, conformance testing, and scientific computation.



Raghu Vallurupalli received the B. S. degree from Andhra University, India. He also received the M. S. degree in digital communications from Indian Institute of Technology, India and the M. S. degree in computer science from University of Maryland, Baltimore County. He is currently pursuing the Ph. D. degree in computer science at University of Maryland, Baltimore County.

His research areas include protocols, conformance testing, high speed networks and congestion control.