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 automaton, 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 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 output 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.

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
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 \(\beta_i\) test is also detected by a \(\beta_{i+1}\) 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 BUO [15] test methods.

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.

<table>
<thead>
<tr>
<th>State</th>
<th>UIO(_1)</th>
<th>UIO(_2)</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a/a</td>
<td>a/a</td>
<td>a/a</td>
</tr>
<tr>
<td>2</td>
<td>a0/a1</td>
<td>a0/a1</td>
<td>a0/a1</td>
</tr>
<tr>
<td>3</td>
<td>a0/a0</td>
<td>b/a1/a1</td>
<td>a0/a0</td>
</tr>
</tbody>
</table>

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 sequence can be defined most generally as

\[
\mathcal{L}_{p,q} \cdot CS(q)
\]

where \(\mathcal{L}_{p,q} = L_1 \cdot L_2 \cdots \cdot L_\lambda\) is a sequence of \(\lambda\) 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. \(CS(q)\) 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 \(\beta_\lambda\) test subsequence for a given specification and set of characterizing sequences is an \(\mathcal{L}\)-sequence of length \(\lambda\) derived from the specification, concatenated with the characterizing sequence for the tail state of the \(\mathcal{L}\)-sequence. The \(\beta_0\) subsequences have an empty \(\mathcal{L}\)-sequence, and test the states of an implementation. The \(\beta_1\) subsequences test a single transition in an implementation, the \(\beta_2\) subsequences test pairs of transitions of an implementation, and so on. For an FSM with \(n\) states and \(k\) inputs, there are \(nk\lambda\) behaviors of length \(\lambda\.

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 \(\beta_i\) test subsequence is a proper subsequence of some \(\beta_{i+1}\) test subsequence. This follows since the assumed input transition allows any \(\beta_i\) subsequence to be extended one edge to the left, forming a \(\beta_{i+1}\) 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 \(\beta_0\) test subsequence, with \(\lambda = |E|\) and no edges repeated along the behavior.
TABLE II

<table>
<thead>
<tr>
<th>CS</th>
<th>$\beta_1$ Subsequences</th>
<th>$\beta_1$ Test Sequence</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>UO_1</td>
<td>$[a/1.a/0.a/1], [b/1.a/0.a/0]$</td>
<td>$b/1.a/0.a/0.a/0.a/1.b/1/b/1$</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>$[a/0.a/1], [b/1.a/0.a/0] [b/1.a/1], [a/0.a/0.a/1]$</td>
<td>$b/1.a/0.a/0.a/0.a/1.b/1/b/1$</td>
<td></td>
</tr>
<tr>
<td>UO_2</td>
<td>$[a/1.a/0.a/1], [b/1.b/1.a/1]$</td>
<td>$b/1.b/1.a/0.a/0.a/1.a/0.a/0.a/1.b/1/b/1$</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>$[a/0.a/1], [b/1.b/1.a/1]$</td>
<td>$b/1.a/0.a/0.a/0.a/1.b/1/b/1$</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>$[a/1.a/0.a/1], [b/1.a/0.a/0]$</td>
<td>$b/1.a/0.a/0.a/0.a/0.a/0.a/1.b/1/b/1$</td>
<td>20</td>
</tr>
</tbody>
</table>

C. Test Sequences

Fixing a set of characterizing sequences and an $L$-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 sequence for each $L$-sequence by concatenating a shortest path from the initial state of the FSM to the start state of $L$-sequence, the $L$-sequence itself, and a characterizing sequence for the tail state of $L$-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 $L_i$ is represented by $(v_i, v_j; L_i)$, where $L_i$ is the input/output label associated with the transition. The characterizing sequence applied to state $v_i$ is indicated by $CS(v_i)$ and the tail (last) state of $CS(v_i)$ by $TAIL(CS(v_i))$. From the directed graph $G = (V, E)$ construct a new directed graph $G'_{\Lambda, CS} = (V', E')$ such that $V' = V$ and $E' \subseteq E_C$, where $E_C$ corresponds to a set of edges which correspond to test subsequences, defined as follows:

$$E_C = \{(v_i, v_k; L_{i,j} \cdot CS(v_j)) :$$

$$L_{i,j} = L_{i_1}, L_{i_2}, \ldots, L_{i_k},$$

$$\exists p_1, \ldots, p_{k-1} \in V \text{ such that}$$

$$(v_{p_1}, v_{p_2}, L_{i_1}), (v_{p_2}, v_{p_3}, L_{i_2}), \ldots, (v_{p_{k-1}}, v_j, L_{i_k}) \in E$$

and $TAIL(CS(v_i)) = v_k).$

The Rural Chinese Postman Problem involves finding a minimum cost tour on graph $G'_{\Lambda, CS}$ 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 $G'_{\Lambda, CS}$.

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 $v_i$ and ending vertex $v_k$ in graph $G$.

2) Construct a graph $G'_{\Lambda, CS}$ from graph $G$ by adding an edge $(v_i, v_k; L_{i,j} \cdot CS(v))$ from vertex $v_i$ to vertex $v_k$ for each test subsequence where $v_k$ is the state reached on applying characterizing sequence $CS(v)$ at state $v_j$.

3) Construct an augmented graph $G'_{\Lambda, CS} = (V', E')$ from graph $G'_{\Lambda, CS}$ (where $V' = V'$ and $E'$ is initialized to $E'$) by adding edges from $E$ to $E'$ such that every vertex in $V'$ 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 $E_C$ is weakly-connected. The cost associated with an edge $(v_i, v_k; L_{i,j} \cdot CS(v)) \in E_C$ is the sum of the costs of edges labeled $L_{i_1}, L_{i_2}, \ldots, L_{i_k}$, and the costs of all edges in $CS(v_j)$. Traversing an edge $(v_i, v_k; L_{i,j} \cdot CS(v_j)) \in E_C$ corresponds to traversing a $\beta_1$ subsequence $L_{i,j}$. 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 $G'_{\Lambda, CS}$ 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 $G'_{\Lambda, CS}$.

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 sequence constructed from $\beta_1$ 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 \( \beta_1 \) 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 \( \beta_1 \) subsequences using \( UO_2 \) 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 \( \beta_1 \) subsequences using \( UO_2 \) 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 \( G'_{\delta,CS} \) 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 (\( \beta_0, \beta_1, \) 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 \( \beta_1 \) testing but not in \( \beta_2 \) testing.

A Testing Hierarchy for the Reset Method

We introduce a slight variation of reset method testing, with the property that a \( \beta_j \) test sequence detects at least as many faults as a \( \beta_j \) test sequence, and where there exists a value \( j \) such that a \( \beta_j \) test sequence detects all faults.

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

In our variant 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 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 is that forced 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 \( L \)-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 \( L \)-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 \( \beta_j \) reset test sequences form a hierarchy that converges in the limit to complete fault coverage.
Theorem 1: If a fault is detected by a $\beta_i$ restricted reset test sequence, then it is detected by a $\beta_{i+1}$ restricted reset test sequence.

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

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

Suppose the fault is detected in the $L$-sequence $L_{p,q}$. The characterizing sequence has length at least 1, so let $q'$ be the second state of this sequence. Then the $\beta_{i+1}$ test subsequence consisting of the original preamble, the $L$-sequence $L_{p,q'}$, 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 preamble and $L$-sequence is traversed in the $\beta_{i+1}$ 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 $\beta_{i+1}$ test subsequence from the $\beta_i$ test subsequence: the last edge of the preamble becomes the first edge of the $L$-sequence. Finally, if the preamble is empty, the $L$-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 $\beta_i$ test sequence, and which uncovers a fault, will also be traversed in a $\beta_{i+1}$ 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 $\beta_i$ reset test sequence, then for some $j > i$ it is detected by a $\beta_j$ reset test sequence, where $j \leq i + \ell$, where $\ell$ is the length of the longest characterizing sequence. This follows from extending the $L$-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 a $\beta_i$ reset test sequence, for each $i \geq 0$. For this specification FSM, let $B_i$ be the set of all deterministic FSM’s that pass

the $\beta_i$ reset test sequence. From Theorem 1, we have that any machine passing a $\beta_i$ reset test will also pass a $\beta_i$ reset test, for $0 \leq i < j$. 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.

Theorem 2: For any $i \geq 0$, we can find FSM’s $M_1$ and $M_2$ such that the difference between the machines can be detected by a $\beta_{i+1}$ reset test derived from $M_1$, but not by a $\beta_i$ 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 $L$-sequences with empty preambles to catch the fault. The fault can be caught only by an $L$-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 $\beta_i$ test and caught by a $\beta_{i+1}$ 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 $\beta_i$ 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 $\beta_n$ 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 $M_2$ 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 $\beta_i$ reset test subsequences with empty preambles will travel every path in the larger of the two trees, and so will determine equivalence.

\[\square\]

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 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, $\beta_i$ 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 "$\beta_i$ 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 $\beta_i$ test sequence is also traversed in a $\beta_{i+1}$ 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 $\beta_i$ test sequence, beginning at the initial state, is also traversed in a $\beta_{i+1}$ 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 $\beta_i$ bridge test sequence is a prefix of the $\beta_{i+1}$ 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 $\beta_{i+1}$ test sequence to the end of a $\beta_i$ 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 $\beta_{i+1}$ test with $\beta_i$ through $\beta_1$ 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 $e$ is the
number of machines found equivalent with the automata equivalence algorithm, then our measured fault coverage is \( \frac{d}{(r - c)} \).

### 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 \( \beta_0 \) to \( \beta_4 \), 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. 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 \( \beta_i \) 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 \( \beta_j \) 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.

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 100 000 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 \( \beta_i \) 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 \( \beta_j \) 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

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Extra states</th>
<th>( T_{D1} )</th>
<th>( T_{D2} )</th>
<th>( T_{D3} )</th>
<th>( T_{D4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>90.4, 89,990</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>81.2, 174,311</td>
<td>99.3, 6408</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>4</td>
<td>73.4, 248,101</td>
<td>98.3, 1642</td>
<td>99.9, 927</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
</tbody>
</table>

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 \( \lambda \) 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 \( N^I \) behaviors of length \( \lambda \). Using \( N \) as a bound on bridge sequence length, we have a bound of \( N^I(N + I + u) \) 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 95% 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 UO CHARACTERIZING SEQUENCE

<table>
<thead>
<tr>
<th>Fault class</th>
<th>Extra states</th>
<th>$T_{d1}$ %</th>
<th>$T_{d2}$ %</th>
<th>$T_{d3}$ %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>90.0, 97.643</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>80.0, 194.987</td>
<td>99.6, 3874</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>72.8, 264.344</td>
<td>98.9, 11041</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>89.7, 87.510</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>79.6, 171.256</td>
<td>99.6, 3173</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>72.3, 223.723</td>
<td>98.9, 9163</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>98.5, 15.304</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>95.1, 48.803</td>
<td>99.9, 560</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>91.4, 85.546</td>
<td>99.8, 1817</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>4</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>96.5, 34.264</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>91.4, 83.355</td>
<td>99.9, 1251</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>86.7, 129.435</td>
<td>99.6, 3703</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>5</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>97.4, 25.154</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>93.1, 69.126</td>
<td>99.9, 1006</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>88.8, 111.109</td>
<td>99.7, 3012</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>6</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>89.9, 100.74</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>79.9, 199.906</td>
<td>99.6, 3906</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>72.8, 270.781</td>
<td>98.9, 1134</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>7</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>98.9, 11.310</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>95.8, 41.882</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>92.3, 77.199</td>
<td>99.9, 1350</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>8</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>99.6, 41.112</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>98.3, 16.629</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>96.5, 34.794</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>9</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>1</td>
<td>99.9, 11.335</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>2</td>
<td>99.4, 57.472</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>3</td>
<td>98.6, 13.710</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
<tr>
<td>10</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
<td>100.0, 0</td>
</tr>
</tbody>
</table>

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.

We allow for the possibility of an unspecified number of extra states in an implementation and define a hierarchy $(\beta_0, \beta_1, \beta_2, \ldots)$ of test sequences, parameterized by length of behaviors under test. For the reset method, the hierarchy has the property that any fault detected by $\beta_i$ reset testing is also detected by $\beta_{i+1}$ 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 $\lambda$. The value of $\lambda$ 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


