Fault Coverage Evaluation of Protocol Test Sequences

by
Jinsong Zhu and Samuel T. Chanson

Technical Report 93-19
June 1993

Department of Computer Science
The University of British Columbia
Vancouver, B. C. V6T 1Z2
Canada

email: jzhu@cs.ubc.ca, chanson@cs.ubc.ca

Fault Coverage Evaluation of Protocol Test Sequences*

Jinsong Zhu† and Samuel T. Chanson
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z2
Email: (jzhu, chanson)@cs.ubc.ca

Abstract

In this paper, we investigate the quality of a given protocol test sequence in detecting faulty implementations of the specification. The underlying model is a deterministic finite state machine (FSM). The basic idea is to construct all FSMs having $n + i$ states (where n is the number of states in the specification and i a small integer) that will accept the test sequence but do not conform to the specification. It differs from the conventional simulation method in that it is not necessary to consider various forms of fault combinations and is guaranteed to identify any faulty machines. Preprocessing and backjumping techniques are used to reduce the computational complexity. We have constructed a tool based on the model and used it in assessing several UIO-based optimization techniques. We observed that the use of multiple UIO sequences and overlaps can sometimes weaken the fault coverage of the test sequence. The choice of the transfer sequences and order of the test subsequences during optimization may also affect fault coverage. Other observations and analysis on the properties of test sequences are also made.

1 Introduction

Test sequence generation for communication protocols has been an active research area. However, the evaluation of the fault coverage for a given test sequence remains an open problem. Only a few methods such as the W method [4] and DS method [8] have been proven to generate test sequences that can uniquely identify a finite state machine under test. The problem with the W method is that it usually generates rather long test sequences, while the applicability of the DS method is limited as few FSMs possess a distinguishing sequence [10]. Recently, a new method called the UIO method [12] has become popular because of its more extensive applicability and generally short sequence length produced. Some optimization techniques based on the UIO have been proposed [1, 3, 11, 13] to further shorten the length of a test sequence. However, since the UIO method itself does not guarantee to produce a checking experiment [17], the UIO-based optimizations could further reduce the fault detection power. This is especially true when the effect of combined faults in a faulty implementation is not well understood. An accurate evaluation of the fault coverage of

*This work was partially supported by a grant from the Canadian Institute for Telecommunications Research under the NCE program of the government of Canada.

†On leave from the Department of Computer Science, Tsinghua University, Beijing, China.

these methods, as well as the influence of optimization on fault coverage, is therefore important both in practice and in theory.

In [5, 15], a simulation method was proposed to estimate the fault coverage of a given test sequence. Based on a classification of faults, it randomly generates faulty machines to see if they can be defeated by the test sequence. The number of machines that cannot be defeated is used as a measure for fault coverage. A severe limitation of the method is that because of the complicated fault behaviors, it is difficult if not impossible to come up with a complete fault classification. The sampling of a small fraction of all cases also means the results must be interpreted with some degree of uncertainty.

An argument for this method is that since the number of distinguishable machines with n states, m inputs, and p outputs is enormous (asymptotically approaching $(np)^{mn}$ [9]), it is apparently infeasible to perform a brute force examination of all of them to decide the exact fault coverage. This is true; however, we can attack the problem in another way: instead of randomly generating FSMs and then checking for their conformance, we can construct only those FSMs that will accept the given test sequence and generate the same outputs but do not conform to the specification. Such FSMs will be referred to as *indistinguishable* FSMs with respect to the specification and a given test sequence. They are the faulty FSMs that cannot be detected by the test sequence. For a good test sequence, the number of indistinguishable FSMs, say k, should be much smaller than the number of all possible machines, and thus it should be easier to generate and examine all of them. The fault coverage can then be measured by k like this: The smaller the value of k, the better the fault coverage. If $k = 0$, then the test sequence has full fault coverage. This technique will uncover all faults and eliminates the need to explicitly consider various fault combinations.

The idea of generating machines satisfying a particular sequence was also proposed in [18] for test sequence generation. Their method is to use the constraint satisfaction techniques in AI to generate the machines. However, the method did not take full advantage of the properties of the deterministic FSM, and was low in efficiency. In comparison, although the generation procedure is in essence a large combinatorial problem, by employing some techniques to reduce the computational complexity, our method has exhibited impressive performance for machines with a reliable reset capability (see Section 3). We have used our tool in assessing several test sequence generation methods including some optimization techniques. A motivation for evaluating optimization techniques was that we were suspicious of the fault coverage of some of the optimization methods. The observations obtained with the tool confirmed our suspicion. This should alert us of the possible coverage reduction when applying optimization techniques to reduce test sequences.

In the following sections, we give a description of the underlying model used in our study in Section 2, and then detail our coverage evaluation methodology in Section 3. Section 4 illustrates the method with an example and gives some empirical performance data. In Section 5, four optimization techniques to be assessed are briefly summarized. Section 6 presents the results of the assessment, and offers some observations on the properties of test sequences. Finally, in Section 7, we close the paper by highlighting some future work.

2 Underlying Model

The underlying model in our study is a deterministic FSM. It is used in modeling the control part of a protocol (other techniques have been proposed for testing the data part [16, 2]). A deterministic FSM can be represented by a quintuple $M =< Q, X, Y, \delta, \lambda >$, where Q, X, Y are the internal states, input alphabet and output alphabet respectively. δ (the *next state function*) is a mapping of $Q \times X$

into Q, and λ (the *output function*) is a mapping of $Q \times X$ into Y. The functions δ and λ can be extended for an input sequence $\sigma = x_1 x_2 ... x_k$ as usual: $\delta(q_1, \sigma)$ is the final state after σ is applied to state q_1, and $\lambda(q_1, \sigma)$ denotes the corresponding output sequence. That is, $\lambda(q_1, \sigma) = y_1 y_2 ... y_k$ where $y_i = \lambda(q_i, x_i)$ and $q_{i+1} = \delta(q_i, x_i)$ for $i = 1, ..., k$, and $\delta(q_1, \sigma) = q_{k+1}$.

If a state of an FSM is designated as the initial state, denoted as q_0, the FSM is said to be *initialized*. We will use initialized FSMs as examples in this paper. The method also applies to uninitialized FSMs as long as a test sequence for each possible initial state is available.

An FSM is deterministic if any input symbol fed to the FSM causes a unique transition, i.e.,

$$\forall q_i, q_j, q_k \in Q \ \forall x \in X \ \ (\delta(q_i, x) = q_j \wedge \delta(q_i, x) = q_k \Leftrightarrow q_j = q_k). \qquad \text{(P1)}$$

It can be derived from P1 that if two states produce different outputs for the same input, then the two states must be distinct, i.e.,

$$\forall q_i, q_j \in Q \ \forall x \in X \ \ (\lambda(q_i, x) \neq \lambda(q_j, x) \Rightarrow q_i \neq q_j). \qquad \text{(P2)}$$

A simple extension of P2 is that if two states produce different output sequences under the same input sequence, then the two states must be different. Let X^* denote the set of finite-length input sequences, then

$$\forall q_i, q_j \in Q \ \forall \sigma \in X^* \ \ (\lambda(q_i, \sigma) \neq \lambda(q_j, \sigma) \Rightarrow q_i \neq q_j). \qquad \text{(P3)}$$

For a given input output sequence, if we do not limit the number of states and the input alphabet, there can be an infinite number of automata that "implement" the sequence and the number of indistinguishable FSMs can be infinite. We therefore assume the number of states in an indistinguishable FSM to be no more than the number of states in the specification plus a small integer, say i. We will study the cases $i = 0$ and $i = 1$ as examples. For $i = 0$, conformance checking can use the usual machine equivalence algorithm [5, 15] (note that the V-equivalence [4] is presumed in this study as the equivalence relation between two automata), because in this case, conformance means equivalence, and vice versa. For $i > 0$, the generated FSM is first minimized, and then checked using the same algorithm. Other faults considered in this study are the usual ones, i.e., output faults, transfer faults, and their combinations.

To avoid equivalent states, the FSM is assumed to be minimal. This is justifiable because we can reduce an FSM to its minimal form [7], and testing can only determine an implementation's conformance up to the level of equivalence. The machine should also be strongly connected and fully specified. Strong connectivity ensures each state can be reached from any other state. For partially specified machines, we use the completeness assumption that the machine will remain in the present state without producing any output (or *null* output) for any unspecified input.

In the following sections, we shall also use the graph representation of an FSM. This is a directed graph $G = (V, E)$, where the vertex set V denotes the set of states, and the edge set E represents the transitions, *i.e.*, $V = \{q_0, ..., q_{n-1}\}$, $E = \{(q_i, q_j) | i, j \leq n-1$ and there is a transition from q_i to $q_j\}$. An edge from q_i to q_j, which receives input a_k and produces output o_l, is labeled by $(q_i, q_j; L)$ where $L \equiv a_k / o_l$, the input part of L is denoted $L^{(i)} \equiv a_k$, and the output part of L is $L^{(o)} \equiv o_l$. This representation is useful when graph algorithms are used to derive test sequences.

3 Coverage Evaluation Methodology

Generating indistinguishable FSMs can be viewed as the reverse procedure of test sequence generation. In this procedure, a test sequence and a specification FSM are used as inputs for constructing indistinguishable FSMs. The test sequence can be thought of as an unfolding of the FSM. The idea is to "collapse" the test sequence back to one or more FSMs which may or may not conform to the original specification. If only one FSM is obtained, then it must conform to the specification and the number of indistinguishable FSM is zero, or, the coverage is 100%.

An FSM is said to have a *reset capability* (or *resettable*) if a special input signal ri always correctly sets the machine to its initial state q_0 from any state. Otherwise it is called a *resetless* machine. In the following procedure we will study resettable machines only. For resetless machines, the entire test sequence can be considered as a single subsequence and thus is a special case of the first situation.

For a resettable machine, the test sequence consists of test subsequences which start from q_0. Based on the property of a deterministic FSM, we can construct a *test tree* with these subsequences. The root of the tree is q_0. Each node corresponds to a state in the specification. The edges from a node to its children represent outgoing transitions from the corresponding state for each input symbol. Input symbols are arranged in a fixed order for every node to avoid isomorphic trees. The depth of the tree is the length of the longest subsequence. For a resetless machine, the test tree degenerates to a simple path. Figure 1 shows a sample FSM, its UIO test sequence, and the corresponding test tree. Clearly, this tree can be constructed efficiently (in polynomial time) from the test sequence. The properties P1 and P2 guarantee a unique tree for a given test sequence.

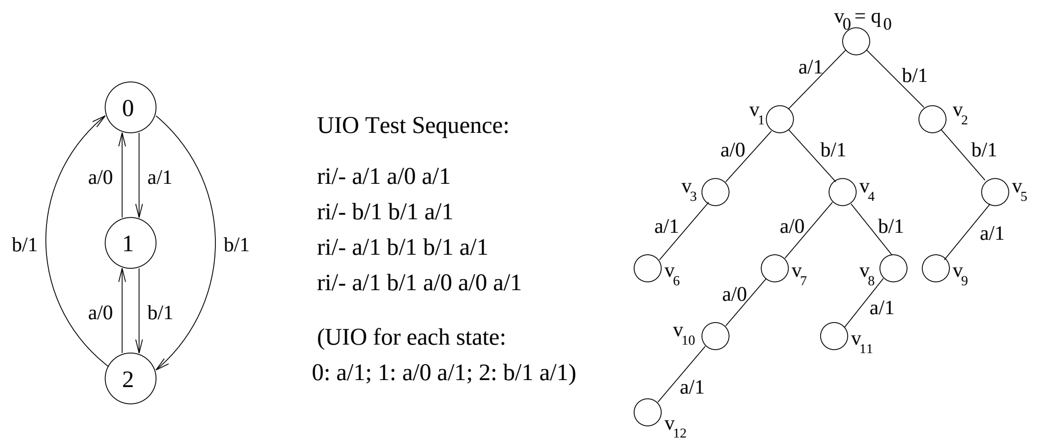


Figure 1: Test tree corresponding to the UIO test sequence

Now we assign each node a variable v_i, with i numbered according to the node's breadth-first traversal order in the tree (see Figure 1). v_0 is always assigned to the root q_0. Where there is no ambiguity, we shall use the term node and variable interchangeably. The number of variables l (other than v_0) can be determined by a breadth-first traversal of the tree. For example, in Figure 1, $l = 12$. Initially, each variable can represent any state. Suppose the specification FSM has n states: $Q = \{q_0, q_1, ..., q_{n-1}\}$, then the domain of each variable is Q. A consistent instantiation of all variables constitutes a solution FSM, which is either the specification FSM or an indistinguishable machine. If we search all variables to generate the solutions as was done in [18], the computational complexity would be $O(n^l)$. The following techniques are used to reduce the computational complexity.

(1) Preprocessing: Although the domain of each variable is Q, the deterministic property will often restrict the values that a variable can assume. For example, according to P2, a variable cannot take on the value (i.e., a state) of a previous variable which it is not equal to. In the extreme case, each variable may only have one value, representing the given FSM. We reduce the domains of the

variables in the breadth-first order. For each variable v_i, we can obtain a set of variables, called unequal variables of v_i, whose indices are smaller than i and whose values are different from that of v_i. Variables that can only have one value constitute the unique set. Initially, the unique set contains v_0 only. The uniqueness of a variable v_i can be determined by examining its set of unequal variables. If this set contains the current unique set, then v_i must itself be a uniquely determined variable and is added to the unique set. To prevent isomorphic solutions, the unique state q_j assigned to v_i is chosen such that j is the smallest index not yet assigned. For example, if v_i is the first one that differs from v_0, then it is assigned q_1. The first variable that is distinct from v_0 and v_i is then assigned the next unassigned state, q_2. A variable that is not in the unique set but contains some unique states in its set of unequal variables can have these unique states removed from its domain. This is because it is not possible for the variable to assume any of these values. This procedure is performed until all variables have been processed. The results are a reduced domain and a set of unequal variables for each variable. The preprocessing phase often prunes the search space greatly and saves considerable time in the subsequent searches. Furthermore, for any variable v_j which is not uniquely determined, its set of unequal variables can help to reduce the search space dynamically, since it will not be necessary to assign v_j a value which has already been assigned to any of its unequal variables.

(2) Backjumping: During searching, when a variable cannot be assigned any value which is consistent with the previous assignments (a *dead-end* situation), we can jump back to the variable which causes the inconsistency rather than backtracking one step at a time as is usually done. This idea is widely used in solving search problems [6]. The point is to go back to the source of failure as far as possible. In our problem, when a variable is instantiated, it may be forced to take a value in two ways. First, it may only take a single value if its domain size is one. Second, the assignment of a previous variable which has the same input symbol may force it to assume the same value in order to be consistent with the properties of a deterministic FSM. Such value-forced variables cannot be the source of failures, so when a dead-end is encountered, they need not be reconsidered in selecting candidates. When the test sequence contains many identical transitions, this situation will occur very frequently.

The algorithms for preprocessing and backjumping search are given below.

Algorithm-PREP: Domain reduction of node variables

Input: Node variables $v_i, i = 1, .., l$
Output: Reduced domain D_i for each v_i and its unequal variables NEQ_i.

Step 1: Generate unequal variables: Initially, the set of unequal variables for each v_i, $NEQ_i = \emptyset$.

for every $v_i (1 \leq i \leq l)$ **do**
  for every v_j with $j < i$ **do**
    if $(v_i \neq v_j)$ **then**
      add v_j to NEQ_i.

Step 2: Reduce domains of variables: Initially, the set of uniquely determined variables $U = \{v_0\}$, state set $S = \{q_1, ..., q_{n-1}\}$, and the domain for $v_i (i = 1, ..., l)$ is $D_i = S \cup \{q_0\}$.

for every $v_i (1 \leq i \leq l)$ **do**
  if $U \subset NEQ_i$ **then begin**
    add v_i to U;
    $D_i \leftarrow \{q_k\}$, k = the smallest subscript in S ;
    $S \leftarrow S - \{q_k\}$;

for all $v_j \in NEQ_i$ **do**
  $D_j \leftarrow D_j - \{q_k\}$;
end else
  for all $v_j \in NEQ_i$ **do**
    if $v_j \in U$ **then**
      $D_i \leftarrow D_i - D_j$;

In Step 1, the set of unequal variables for each variable is generated. The **if** statement is executed $l(l-1)/2$ times. The evaluation of the condition $v_i \neq v_j$ is based on property P3. Since input sequences starting from a node constitute a subtree rooted at that node, the evaluation can be performed by comparing two subtrees rooted at v_i and v_j respectively. The comparison can be done by means of a breadth-first search algorithm for the subtree. When comparing two nodes, property P2 is used. When a pair of distinct nodes is found, the two subtrees are distinct (P3). However, when subtrees are incomplete, the node with an absent edge is not considered distinct from its corresponding node where the edge is present. For example, in Figure 1, although v_2 has no edge corresponding to a, we cannot conclude v_2 must be different from v_0. Similarly, v_5 may possibly be equal to v_2. The time complexity of tree comparison is at most $O(l)$. Hence, the complexity of Step 1 is $O(l^3)$.

The domains are reduced in Step 2 using the NEQ sets. If NEQ_i contains the current uniquely determined variables, v_i itself becomes a member of U. For each member v_j of U, the domains of v_j's NEQ variables can be reduced by removing v_j's corresponding value. The idea is that if a variable is not equal to a uniquely determined variable, it cannot assume the unique value of that variable. The complexity of this step is $O(l^2)$. Thus, the overall complexity of Algorithm-PREP is $O(l^3)$. Note that the test tree must have already been constructed from the test sequence before using this algorithm.

Algorithm-SEARCH: Backjumping search to find solutions

Input: Node variables with reduced domains and the original specification
Output: The set of indistinguishable FSMs

Initially, the index of variables $i = 1$. A stack is used to store intermediate steps.

1. $c_i \leftarrow$ get_candidate(v_i);
2. **if** c_i = NONE **then begin**
3.   /* dead-end encountered */
4. $i \leftarrow$ popstack;
5.   **if** stack is empty **then**
6.   exit with no more solutions;
7.   goto 1;
8. **end**
9. **else begin**
10.   **if** c_i is not a forced candidate **then**
11.   pushstack(i);
12. $i \leftarrow i + 1$;
13.   **if** $i = l + 1$ **then begin**
14.   /* a solution is found */
15.   record the solution;

16. check its conformance with the specification;
17. $i \leftarrow$ popstack;
18. **end**
19. goto 1; /* go on searching */
20. **end**

The procedure *get_candidate* in line 1 selects a consistent value for v_i from its domain (we will say that v_i is instantiated). The reduced domain and the set NEQ_i both help reduce the number of candidates. During the instantiation process, it builds up a partial solution FSM with the values of variables up to v_i, one by one. A consistent value of v_i adds a transition to the FSM when the corresponding state q_p of v_i's parent v_p has no outgoing transition with the label i/o from v_p to v_i; otherwise v_i is forced to $\delta(q_p, i)$ in the FSM. The partial FSM is constructed incrementally until all variables are assigned values, at which time the FSM becomes a final solution. To avoid isomorphic solutions, when instantiating v_i, only one value out of a set of equivalent candidates is selected. The candidates are equivalent in the sense that they represent equivalent states in the current partial FSM. If one of the equivalent candidates fails, the others will fail too. This also reduces the domains to be searched dynamically. *Get_candidate* returns NONE when no consistent value of v_i can be found. This is the dead-end situation and backjumping takes place. Since a forced variable cannot be the source of a failure as mentioned before, line 11 only pushes an unforced variable onto the stack for subsequent backtracking. In line 13, $i = l + 1$ means all variables have been successfully instantiated, thus a solution is found. Line 16 checks the solution's conformance with the specification using Dahbura and Sabnani's algorithm [5]. A non-conforming solution represents an indistinguishable FSM.

In the worst case when there is no reduction of domain for all variables and no forced variables, the complexity of Algorithm-SEARCH would remain $O(n^l)$. However, the complexity is far less in practice. The next section shows some examples.

4 Empirical Results

A tool based on the above model has been implemented in C and runs on a Sun 4/30 workstation under Sun OS 4.1.1. It accepts as input a test sequence and a specification FSM. The tool first constructs the test tree, reduces the domains using Algorithm-PREP, and then searches for all solutions using Algorithm-SEARCH. We will first illustrate the algorithms with an example, then use the tool to study some existing test sequences. Empirical results on the performance of the tool are also presented.

4.1 An Illustrative Example

We illustrate the algorithms with the example in Figure 1. Initially, $D_0 = \{q_0\}$, $D_i = \{q_0, q_1, q_2\}$ for $i = 1, ..., 12$. From the tree, it can be seen that v_1 must be different from v_0 as their outputs to a are different. At this point, only v_0 is in the unique set U, so v_1 is also a uniquely determined variable. v_1 is then assigned q_1, the next value in the state set. Next, v_2 is found to be distinct from v_1 because of their different outputs for ba. Therefore v_1's value q_1 is removed from v_2's domain. After all variables have been processed, their domains become:

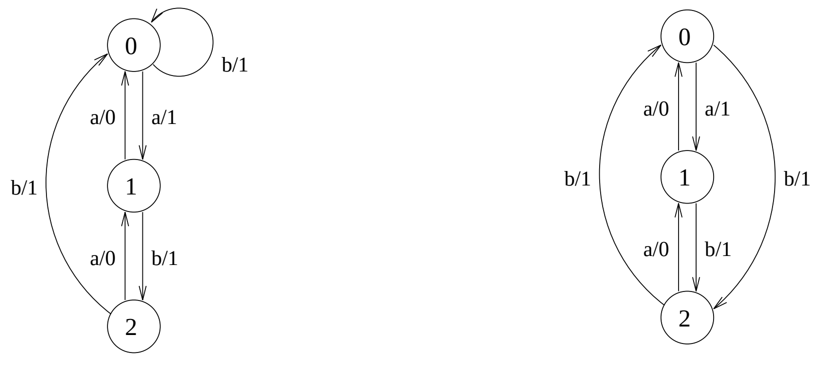
$D_1 = \{q_1\}, D_2 = \{q_0, q_2\}, D_3 = \{q_0\}, D_4 = \{q_2\},$
$D_5 = \{q_0\}, D_6 = \{q_0, q_1, q_2\}, D_7 = \{q_1\}, D_8 = \{q_0\},$

$D_9 = \{q_0, q_1, q_2\}, D_{10} = \{q_0\}, D_{11} = \{q_0, q_1, q_2\}, D_{12} = \{q_0, q_1, q_2\}$.
Note that the domains for the leaf nodes cannot be reduced because they have no outgoing edges. Now Algorithm-SEARCH is used to search for solutions. For the first five variables, only v_2 has two choices. We first choose q_0, which is consistent with $v_1 = q_1$. The remaining variables are forced to take the values q_1, q_1, q_0, q_1, q_0, q_1, and q_1 respectively. Thus, we obtain our first solution:

$v_0 = q_0, v_1 = q_1, v_2 = q_0, v_3 = q_0, v_4 = q_2, v_5 = q_0, v_6 = q_1,$
$v_7 = q_1, v_8 = q_0, v_9 = q_1, v_{10} = q_0, v_{11} = q_1, v_{12} = q_1$.

Since all variables after v_2 were forced, the algorithm goes back directly to v_2 to find the next solution. The next candidate for v_2 is q_2, which is also consistent with $v_1 = q_1$. The other variables are again forced to the same values, producing another solution:

$v_0 = q_0, v_1 = q_1, v_2 = q_2, v_3 = q_0, v_4 = q_2, v_5 = q_0, v_6 = q_1,$
$v_7 = q_1, v_8 = q_0, v_9 = q_1, v_{10} = q_0, v_{11} = q_1, v_{12} = q_1$.



(a) Solution 1: indistinguishable FSM

(b) Solution 2: conforming FSM

Figure 2: The two solutions for the UIO test sequence given in Figure 1

It can be seen there are no more candidates for each variable, and the algorithm terminates with two solutions. Figure 2 shows the two corresponding FSMs, with the first one an indistinguishable FSM, and the second one conforming to the specification. The two solutions are both obtained backtrack-free, which means only $O(l)$ time is needed to get a solution. This is the best result we can hope for. We also observed that although domains for the leaf nodes cannot be reduced, they are usually forced to some values. For example, v_6, v_9, v_{11}, and v_{12} were all forced variables.

4.2 Evaluation of Some Test Sequences

In [5, 15], some examples were used for fault coverage evaluation using the simulation method. We applied our tool to the examples, and derived the same conclusions. The W, DS and UIO methods are evaluated in [15]. The conclusion was that they all had the same coverage for strong conformance test. Nevertheless, since the number of all possible machines is $15^{10} (\approx 5.7 \cdot 10^{11})$ for the sample machine M (Fig. 1 in [15]), and only 10^6 randomly generated machines were examined, we were never completely sure about the conclusion. However, with our tool, it can be confirmed that the three test sequences given in the paper all have a unique conforming solution (see Table 1 for FSM 1). The example in [5] (FSM 2) for UIO method also has a unique solution. An encouraging observation is that all solutions were obtained backtrack-free.

To further evaluate our tool, we studied the W method with an extra state, using the same sample FSM 1. The original example for the DS method in [8] (FSM 3) was also studied for both the DS

Example	(n, m, p, L)	Method	No. of Solutions	No. of Backtrackings	No. of Forced Cases	Time (s)
FSM 1 [15]	(5,2,3,29)	UIO	1	(0, 0)	(9, 9)	0.04
	(5,2,3,35)	DS	1	(0, 0)	(12, 12)	0.04
	(5,2,3,58)	W	1	(0, 0)	(18, 18)	0.05
	(5,2,3,130)	W^{+1}	15	(0, 60)	(48, 558)	0.15
FSM 2[5]	(7,4,4,111)	UIO	1	(0, 15)	(38, 256)	0.13
FSM 3 [8]	(6,2,2,40)	UIO	1	(8, 59)	(35, 203)	0.09
	(6,2,2,62)	DS	1	(0, 0)	(25, 25)	0.07
ISDN BRI[1]	(8,14,12,425)	UIO	1	(0, 0)	(105, 105)	0.82
NBS TP4[14]	(15,27,26,4131)	DS	1	(105, 488)	(2874, 2909)	273.77

Table 1: Performance of our method on some examples

and UIO methods, assuming it is resettable. Furthermore, two real protocols, the ISDN BRI network layer protocol [1] and a subset of the NBS Class 4 transport protocol (TP4) [14], were studied for the UIO and DS methods respectively. The fault coverages for these examples were all found to be 100%.

Table 1 summarizes the results of our experiments. In the table, (n, m, p, L) is the number of states, inputs and outputs for the example, and the length of the test sequence, respectively. The solutions for all the examples are all unique and conforming to the original specification (no indistinguishable solutions). Two numbers are recorded in the column for "No. of Backtrackings" and also the column for "No. of Forced Cases". The first one is the number when the first solution is found, and the second when the algorithm stops. Forced cases occur when a variable is forced to a value during instantiation. The larger the number of forced cases, the better the gains from backjumping. The column "Time" is the number of seconds of CPU time consumed by the tool, including input and output processing. The results showed that the strategy of preprocessing and backjumping is very effective. Even a sizable real protocol like NBS TP4 can be handled efficiently. Considering the small number of backtrackings (many of which are backtrack-free) and the worst case complexity of n^L, the savings are substantial.

5 Test Sequence Optimization Techniques

A useful application of our tool is to evaluate some test generation methods whose fault coverage is still uncertain. If the fault coverage is not 100%, the tool may be able to help explain why. We chose to study the various optimization techniques in this paper. In this section, we give a cursory introduction to the four major optimization techniques which are evaluated in the next section.

The basic method of testing an FSM is to test each transition and identify the final state after the transition. A test subsequence, denoted as $TEST(q_i, q_j; L)$ [1] (also called *segment* in [3]), consists of the input for the testing edge $(q_i, q_j; L)$ followed by a characterizing sequence (CS) for state q_j, *i.e.*,

$$TEST(q_i, q_j; L) = L^{(i)} \cdot CS(q_j).$$

Each test segment is concatenated by the use of a reset signal and a shortest path $P(q_i)$ from q_0 to q_i to form the overall test sequence TS, *i.e.*,

$$TS = \sum_{(q_i,q_j)\in E} ri \cdot P(q_i) \cdot TEST(q_i, q_j; L).$$

This method was used in our previous examples. The optimization techniques focus on how to connect the test subsequences to minimize the length of the overall test sequence. The reset capability is not required. Subsequences can be started one after another, or even overlapped, without having to go back to the initial state. Since UIO sequences usually exist and multiple UIO sequences often provide more room for optimization, UIO is chosen as the CS in all the four optimization techniques. But where only single UIO is used, other characterizing sequences can also be used in principle.

5.1 Rural Chinese Postman Method

This method was proposed by Aho, Dahbura, Lee and Uyar [1]. It uses the *Rural Chinese Postman* (RCP) problem in graph theory to minimize the transfer sequence between subsequences. It opened a new direction for optimization research. Other optimization methods are basically extensions of this method. In the paper, the authors formulate the optimization problem as follows. The specification FSM is represented as a graph $G = (V, E)$. First, a new graph $G' = (V', E')$ is constructed such that $V' \equiv V$ and $E' \equiv E \cup E_C$, where

$$E_C = \{(q_i, q_k; L_l \cdot UIO_j) | (q_i, q_j; L_l) \in E \text{ and } \delta(q_j, UIO_j) = q_k\}.$$

Edges in E_C are "pseudo-transitions" that represent all test subsequences. They have exactly the same number as the transitions in G, and contain all the vertices of V. Thus, the edge-induced subgraph $G[E_C] = (V, E_C)$ is a spanning subgraph of G'. The cost of edges in E_C is defined as the cost of $TEST(q_i, q_j; L_l)$ which is usually taken to be the total number of edges. Clearly, the objective becomes traversing each edge in E_C at least once with a minimum cost tour of G'. Such a tour is a Rural Chinese Postman tour. The RCP problem is NP-complete for the most general case, but when $G[E_C]$ is weakly connected, it can be solved in polynomial time. In [1], it is pointed out that if an FSM has the reset capability or has a self loop for each state, then $G[E_C]$ must be weakly connected. We have observed that even for machines without reset or self loops, most of them are still weakly connected. This convinces us that the technique has a wide applicability. When $G[E_C]$ is not weakly connected, heuristics will have to be used to find a sub-optimal solution.

To solve the RCP problem, a rural symmetric augmentation graph $\hat{G}^* = (\hat{V}^*, \hat{E}^*)$ of G' is constructed such that $\hat{V}^* \equiv V'$ and $\hat{E}^*$ contains all edges in E_C, and possibly some edges in E. The idea is to minimize the number of edges chosen from E, and at the same time make the augmented graph symmetric, *i.e.*, for every vertex in $\hat{V}^*$ the in-degree equals the out-degree. Algorithms for minimum-cost and maximum-flow in graph theory can be used to find such a minimal augmentation. The edges in $\hat{G}^*$ can be covered by an Euler tour which can be computed efficiently. The tour is then the overall test sequence.

In this method, only one UIO sequence is used for each state. The following method which is an extension of this one, employs multiple UIOs for a state to achieve further reduction of test sequence length.

5.2 Multiple UIO Method

Multiple UIO sequences are a set of minimal length UIOs for a state. It was found in [13] that using different UIOs for identifying a state in different subsequences can reduce the length of the overall test sequence. This is because by selecting the appropriate UIOs, the graph $G[E_C]$ can be made closer to symmetry, *i.e.*, the difference of in-degrees and out-degrees for a vertex may be smaller, thereby fewer edges from E are needed to augment it. The problem is also translated into a minimum-cost maximum-flow problem, and heuristics may be needed when the problem is intractable in rare cases. Different UIOs can then be selected for testing different transitions with the same final state.

It was shown in [13] that up to 30% reduction of test sequence length could be achieved, depending on the properties of the FSM. This is a substantial saving over the single UIO RCP method.

5.3 Overlaps Method

Another factor, the overlapping between subsequences, is considered in [3] to further minimize the test sequence. Single UIO sequence is used. The idea is that if two subsequences S_1 and S_2 are overlapped, *i.e.*, the last part of S_1 coincides with the first part of S_2, then they can be merged with the overlapping part serving both S_1 and S_2. If S_2 is completely contained in S_1, then S_2 disappears after the merge. The problem is how to maximally exploit the overlapping. In [3], a technique is presented to transform the problem into a minimum cost maximum cardinality matching problem in a bipartite graph. In general, the solution to the problem can be a number of disconnected circuits in the transformed graph. Some heuristics are then used to connect them. Empirical study in [3] shows that this technique achieves a significant reduction in the test sequence length.

5.4 Multiple UIO and Overlaps Method

This technique combines multiple UIO sequences and overlaps to fully exploit the properties of the subsequences and yield the shortest test sequence. Two such methods were proposed in [3] and [11] respectively. The method in [11] is used in our study as it produces shorter test sequences in general.

In [11], a machine is called *definitely diagnosable* if it has no converging edges, *i.e.*, no two edges going into the same state with the same input output label. In this case, the test sequence is simply an Euler tour of the FSM graph G (minimally augmented if G has no Euler tour) plus the UIO sequence for the last state of the tour. The rationale is that for such machines, the test sequence not only tests each transition but also serves as the characterizing sequence for each state visited. If converging edges exist, a graph G' is constructed by removing the converging edges. Then a set of disjoint paths in G' that covers all edges (only one path if G' contains Euler tours) is computed. The problem becomes how to join these disjoint paths and the converging edges such that the total length is minimal. UIO sequences are used both for joining the paths and also identifying the states along the paths. It turns out that such a problem can be converted to a maximum cardinality minimum cost matching problem for a bipartite graph. If the solution is an Euler tour (or path), then the UIO of the last state is appended to the tour to form a test sequence. Otherwise some heuristics are used to connect the disconnected parts. The overall test sequence is shown to be a combination of multiple UIOs and overlaps.

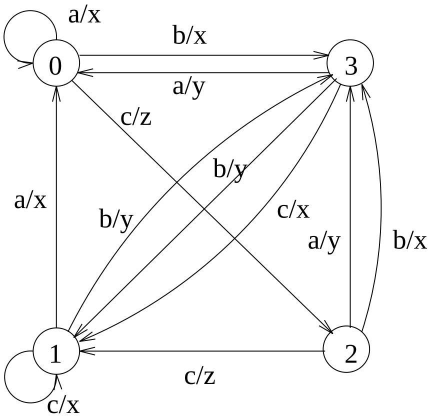
The examples in [11] as well as our experiments indicate that this is another leap in the optimization. The length of the test sequence given in [11] is surprisingly short, typically in the order of the number of transitions in the FSM.

6 Coverage Evaluation Results

The tool presented in Section 3 was used to conduct an empirical study of the four optimization methods. It is found that optimization can sometimes reduce the fault detection capability of a test sequence compared to an unoptimized one. The tool was also employed to explain how this could have happened.

6.1 An Example

Figure 6.1 is the FSM used in our study. It was generated quite arbitrarily. The minimal UIO sequences for each state with different tail states are also given. The first UIO is the one used when single UIO is required.



STATE	UIO SEQUENCE	TAIL
q_0	c/z a/y	q_3
	c/z c/z	q_1
q_1	b/y a/y	q_0
q_2	a/y a/y	q_0
	a/y b/y	q_1
	c/z b/y	q_3
q_3	a/y a/x	q_0
	a/y b/x	q_3
	a/y c/z	q_2

Figure 3: A sample FSM and UIOs for each state

The test sequences for the above four optimization methods are listed below:

(T1) RCP with single UIO: Length = 45
c/z a/y a/y b/x a/y a/x b/x b/y b/y a/y a/x b/x b/y c/x b/y a/y b/x b/y a/x c/z a/y a/y c/z a/y b/y b/y a/y a/x c/z a/y c/x b/y a/y c/z a/y a/y a/x c/z b/x a/y a/x c/z c/z b/y a/y

(T2) RCP with multiple UIO: Length = 38
c/z c/z b/y a/y c/z c/z b/y a/y c/z a/y a/y c/z b/x a/y c/z c/z b/y a/y a/x c/z a/y b/y b/y a/y c/z c/z a/x c/z c/z c/x b/y a/y b/x a/y b/x c/x b/y a/y

(T3) Overlaps with single UIO: Length = 34
c/z c/z b/y a/y a/x c/z b/x a/y a/x b/x a/y a/x b/x b/y c/x b/y a/y a/x c/z a/y a/y a/x b/x b/y a/x c/z a/y b/y b/y a/y b/x c/x b/y a/y

(T4) Multiple UIO and overlaps: Length = 27
c/z a/y a/y c/z c/z b/y b/y b/y a/y b/x c/x b/y a/y c/z b/x a/y b/x a/y a/x c/z c/z a/x c/z c/z c/x b/y a/y

It can be seen that T2 and T3 each gains about 16% and 24% reduction in length respectively, with respect to T1, and this is consistent with that reported in [3, 13]. T4's reduction is 40%, approximately the sum of the effects of multiple UIOs and overlaps.

To evaluate the coverage, each test sequence is fed to our tool. The number of solutions is the number of indistinguishable FSMs and is used as the measure of fault coverage. Note that in our study, no reset capability is assumed. The results are listed in Table 2 (See Appendix for all solution FSMs).

This, together with some other experiments, led to the following observations.

Observation 1: Multiple UIOs, as well as overlaps, can *sometimes* cause loss of fault coverage. The two optimization methods are comparable in length reduction as well as in coverage loss. The

	T1	T2	T3	T4
Sequence Length	45	38	34	27
No. of indistinguishable FSMs	0	9	8	26

Table 2: Fault coverage results for T1-T4

use of both techniques causes a higher coverage loss than when either one is used alone. Let L_{T_i} and C_{T_i} represent the sequence length and fault coverage for each method respectively, we have (based on a finite number of experiments):

$$L_{T1} \geq L_{T2} \approx L_{T3} > L_{T4}, \text{ and } C_{T1} \geq C_{T2} \approx C_{T3} > C_{T4}.$$

There were some experiments where multiple UIO or overlaps did not reduce the original fault coverage. This implies that the property of an FSM also plays a role in determining coverage. For T4, however, all examples we used showed a coverage loss.

As can be seen, during test sequence generation, different test sequences (with the same length) can be produced because of the multiplicity of Euler tours in a graph. The maximum flow minimum cost may also have different edge sets. This led to our next observation.

Observation 2: The choice of different solutions for the transformed graph problem could lead to test sequences with different fault coverage. In other words, the order of connecting subsequences in forming the test sequence has an effect on fault coverage.

For example, the following test sequence (T2a), which was obtained using a different Euler tour for T2, has only 4 indistinguishable FSMs instead of 9. In other words, it has a better fault coverage than T2.

(T2a) $c/z\ c/z\ b/y\ b/y\ b/y\ a/y\ a/x\ c/z\ a/y\ c/x\ b/y\ a/y\ b/x\ a/y\ b/x\ a/y\ c/z\ c/z\ a/x$
$c/z\ c/z\ b/y\ a/y\ c/z\ a/y\ a/y\ c/z\ b/x\ a/y\ c/z\ c/z\ b/y\ a/y\ c/z\ c/z\ c/x\ b/y\ a/y$

In contrast, in the unoptimized method of constructing the test sequence when reset is available (see Section 5), the order of the subsequences $ri \cdot P(q_i) \cdot TEST(q_i, q_j; L)$ is irrelevant with respect to the fault coverage of the overall test sequence. This is because the test tree is uniquely determined by the subsequences independent of their orders.

Observation 3: For FSMs with a reliable reset capability, the execution order of test subsequences does not affect the fault coverage of the test sequence.

However, the choice of $P(q_i)$ does influence fault coverage. For example, in Figure 1, if we had chosen $b/1$ as the preamble to state 2, the final test sequence would have no indistinguishable solution. This behavior is consistent with Observation 2.

Observation 4: For FSMs with a reliable reset capability, the choice of preambles in the subsequences could affect the fault coverage of the test sequence.

6.2 Analysis

To understand why some faults successfully escaped detection by a test sequence, we examined some of the indistinguishable FSMs. Figure 4 shows three such FSMs with respect to T2.

Note that in Figure 4(b), there are two transfer faults: (F1) $(q_1, q_3; c/x)$, and (F2) $(q_3, q_3; b/y)$. Figure 4(c) has three faults: a transfer fault (F3) $(q_1, q_2; a/x)$, and an output fault concurrent with a transfer fault in one edge: (F4) $(q_1, q_3; c/z)$. Figure 4(d) is a weird mutant with transitions radically different from the original FSM. It would be hard to construct this indistinguishable FSM without the tool.

In Figure 4(b), when the original edge $(q_1, q_1; c/x)$ is tested, the UIO sequence $b/y\ a/y$ intended

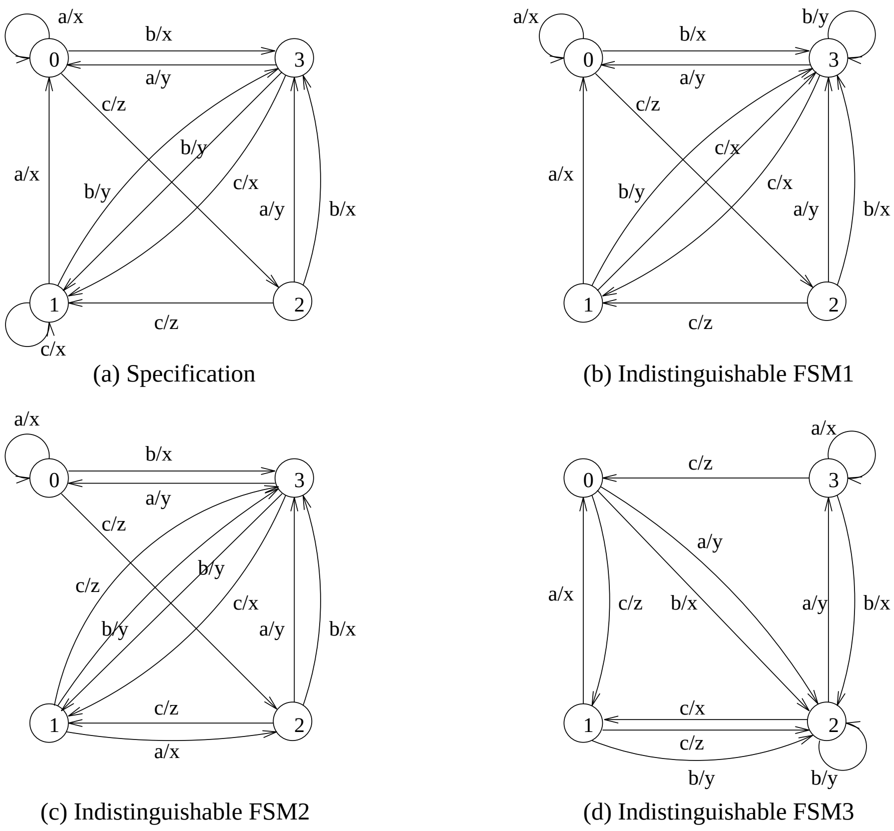


Figure 4: Three indistinguishable FSMs with respect to T2

to identify q_1 is also acceptable by q_3, the tail state of F1, due to F2. When F2 is tested by applying b/y b/y a/y at q_3, it again fools us by giving the same outputs. As the tail state for this UIO is not changed by the faults, other subsequences are not able to exercise the faulty edge. T2, which connects subsequences with two more edges that are not the faulty ones, thus fails to expose the faults. Clearly, this type of faults may also escape detection by the single UIO RCP method. T1 happens to be able to defeat this faulty machine because it has the right transfer sequences. Interestingly, if we had chosen a different transfer sequence by purposely evading faulty edges, T1 would also fail. This indicates that although the use of multiple UIO sequences reduces the length of transfer sequences, it also decreases the probability of capturing some faults. The same is true for overlaps.

Figure 4(c) shows another way we could be cheated by a faulty machine. When testing F3, the subsequence a/x c/z c/z brings the machine to q_3. T2 happens to use c/x b/y a/y, which is acceptable by q_3, to test the outgoing transition c/x from q_1 right after a/x c/z c/z. Instead of testing the transition c/x from q_1, the transition c/x from q_3 is tested. Other subsequences again do not touch the faulty edges. We call this phenomenon the *fault cancellation* effect, where several faults are combined to cancel each other. It can be seen that different ordering of the subsequences could lead to different ways of fault cancellation, hence possible different fault coverages. The FSM shown in Figure 4(d) is difficult to interpret as the transition pattern differs greatly from the original specification. The simple patterns of fault cancellation shown in Figure 4(b) and 4(c) as explained above are unable to unravel how the faults escaped detection. It seems to suggest that no general patterns of fault cancellation could be found to explain all undetected faults.

One may argue that the coverage loss may not be due to the optimization techniques but caused by the UIO sequence itself. This is possible, but as we observed, optimization could make the situation worse, especially when multiple UIO sequences and overlaps are used together. We attribute it to the increased fault cancellation effect caused by complex combinations of the test segments when optimization techniques are used. The arbitrary choice of preambles and transfer sequences as well as the ordering of the subsequences, may also result in a test sequence with damaged fault coverage. This is possible because all test sequence generation methods are based on a correct specification while the UIO sequence and an optimized sequence, unlike a DS or a W set, cannot guarantee an equivalence mapping between the specification and a machine that accepts the sequence.

7 Conclusions

We have examined the issue of fault coverage evaluation of a given test sequence for a deterministic FSM. Our metric for fault coverage is the number of indistinguishable machines, and a tool has been constructed based on this model. The computational complexity of finding indistinguishable FSMs has been substantially reduced by using pre-processing and backjumping techniques. The tool has been used in evaluating the quality of some optimization techniques, and a number of interesting observations were obtained. The most important ones are that the order of transfer subsequences and choices for preambles could affect fault coverage of the sequence. By generating indistinguishable machines, we were also able to examine them to see how they have outflanked a test sequence. Fault cancellation is an interesting phenomenon that causes many faults to go undetected. Optimization techniques, especially when both multiple UIO sequences and overlaps are used, increase the chance of fault cancellation, and thus increase the possibility of losing coverage. However, a faulty machine could exhibit various unexpected behaviors. A general pattern of fault cancellation that can explain all faults seems unlikely.

Although the optimization techniques may weaken the fault detection power of the test sequence, they are still useful in practice because they can still catch most faults. They would be particularly useful in the initial testing phase where coverage is not the main concern. One problem might be that such a testing is not incremental in the sense that it is hard for a subsequent test with a better sequence to take advantage of the result of a previous test. It would be useful if an incremental hierarchy of optimization techniques could be established.

Another interesting issue is to determine the conditions under which the UIO-method or an optimization technique is guaranteed to generate a checking experiment. The conditions are related to both the property of the FSM and the choice of preambles and transfer sequences. Before this is done, the only way may be to run the tool presented in this paper to check the coverage of a test sequence. A potential limitation of this method is the tool's computational complexity for very large problems. Techniques such as learning while searching and parallel algorithms could be developed to further improve the performance of the tool. It is fortunate, however, that checking needs to be done only once for each test sequence, rather than once for each implementation under test.

References

[1] A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Transactions on Communications*, 39(11), November 1991.

[2] S.T. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Proc. IEEE INFOCOM*, San Francisco, March 1993.

[3] M.S. Chen, Y. Choi, and A. Kershenbaum. Approaches utilizing segment overlap to minimize test sequences. In *Proc. IFIP 10th Int. Symp. on Protocol Specification, Testing, and Verification*, 1990.

[4] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, May 1978.

[5] A. Dahbura and K. Sabnani. An experience in estimating fault coverage of a protocol test. In *Proc. IEEE INFOCOM*, 1988.

[6] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

[7] A. Gill. *Introduction to the Theory of Finite State Machines.* McGraw-Hill Book Company, Inc., 1962.

[8] G. Gonenc. A method for the design of fault detection experiments. *IEEE Transactions on Computer*, June 1970.

[9] M. A. Harrison. On asymptotic estimates in switching theory and automata theory. *Journal of ACM*, 13:151–157, 1966.

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

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

[12] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[13] Y.-N Shen, F. Lombardi, and A.T. Dahbura. Protocol conformance testing using multiple UIO sequences. In *Proc. IFIP 9th Int. Symp. on Protocol Specification, Testing, and Verification*, 1989.

[14] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. Technical Report 86–23, Dept of Computer Science, Iowa State Univ., 1986.

[15] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, April 1989.

[16] H. Ural and B. Yang. A test sequence selection method for protocol testing. *IEEE Transactions on Communication*, April 1991.

[17] S.T. Vuong and W.Y.L Chan. The UIOv-method for protocol test sequence generation. In *Proc. 2nd Int. Workshop on Protocol Testing System*, October 1989.

[18] S.T. Vuong and K.C. Ko. A novel approach to protocol test sequence generation. In *Proc. GLOBECOM'90*, December 1990.

Appendix: All Indistinguishable FSMs for the Sample

Target Machine (same for all test sequences):

```
      c    a    b
S0 : z/2, x/0, x/3,
S1 : x/1, x/0, y/3,
S2 : z/1, y/3, x/3,
S3 : x/1, y/0, y/1,
```

(1) T1:

```
===== SOLUTION 1 =====
      c    a    b
S0 : z/2, x/0, x/3,
S1 : x/1, x/0, y/3,
S2 : z/1, y/3, x/3,
S3 : x/1, y/0, y/1,
Conforming solution!

total solutions = 1, non-conforming solutions = 0
```

(2) T2:

```
===== SOLUTION 1 =====
      c    b    a
S0 : z/1, x/2, y/2,
S1 : z/2, y/2, x/0,
S2 : x/1, y/2, y/3,
S3 : z/0, x/2, x/3,
Non-conforming solution!
```

```
===== SOLUTION 2 =====
      c    b    a
S0 : z/1, x/2, y/2,
S1 : z/2, y/2, x/0,
S2 : x/2, y/2, y/3,
S3 : z/0, x/2, x/3,
Non-conforming solution!
```

```
===== SOLUTION 3 =====
      c    b    a
S0 : z/1, x/3, x/0,
S1 : z/2, x/3, y/3,
S2 : x/2, y/3, x/0,
S3 : x/2, y/2, y/0,
Conforming solution!
```

```
===== SOLUTION 4 =====
      c    b    a
S0 : z/1, x/3, x/0,
S1 : z/2, x/3, y/3,
S2 : x/2, y/3, x/0,
S3 : x/2, y/3, y/0,
Non-conforming solution!
```

```
===== SOLUTION 5 =====
      c    b    a
S0 : z/1, x/3, x/0,
S1 : z/2, x/3, y/3,
S2 : x/2, y/3, x/0,
S3 : x/3, y/3, y/0,
Non-conforming solution!
```

```
===== SOLUTION 6 =====
      c    b    a
S0 : z/1, x/3, x/0,
S1 : z/2, x/3, y/3,
S2 : x/3, y/3, x/0,
S3 : x/2, y/3, y/0,
Non-conforming solution!
```

```
===== SOLUTION 7 =====
      c    b    a
S0 : z/1, x/3, x/0,
S1 : z/2, x/3, y/3,
S2 : x/3, y/3, x/0,
S3 : x/3, y/3, y/0,
Non-conforming solution!
```

```
===== SOLUTION 8 =====
      c    b    a
S0 : z/3, x/2, x/0,
S1 : z/2, y/2, x/3,
S2 : x/1, y/1, y/0,
S3 : z/1, x/2, y/2,
Non-conforming solution!
```

```
===== SOLUTION 9 =====
      c    b    a
S0 : z/3, x/2, x/0,
S1 : z/2, y/2, x/3,
S2 : x/1, y/2, y/0,
S3 : z/1, x/2, y/2,
Non-conforming solution!
```

```
===== SOLUTION 10 =====
      c    b    a
S0 : z/3, x/2, x/0,
S1 : z/2, y/2, x/3,
S2 : x/2, y/2, y/0,
S3 : z/1, x/2, y/2,
Non-conforming solution!
```

```
total solutions = 10, non-conforming solutions = 9
```

(3) T3:

```
===== SOLUTION 1 =====
      c    b    a
S0 : z/0, y/0, y/1,
S1 : x/0, x/1, x/2,
S2 : z/2, x/3, y/0,
S3 : -/-, y/1, y/1,
Non-conforming solution!
```

```
===== SOLUTION 2 =====
      c    b    a
S0 : z/0, y/0, y/1,
S1 : x/0, x/3, x/2,
S2 : z/2, x/3, y/0,
S3 : x/0, y/1, y/1,
Non-conforming solution!
```

```
===== SOLUTION 3 =====
      c    b    a
S0 : z/0, y/2, y/3,
S1 : x/0, y/0, x/3,
S2 : x/0, y/1, y/1,
S3 : z/3, x/2, y/2,
Non-conforming solution!
```

```
===== SOLUTION 4 =====
      c    b    a
S0 : z/0, y/2, y/3,
S1 : x/0, y/0, x/3,
S2 : x/1, y/1, y/1,
S3 : z/3, x/2, y/2,
Non-conforming solution!
```

```
===== SOLUTION 5 =====
      c    b    a
S0 : z/2, y/0, y/1,
S1 : x/0, x/1, x/3,
S2 : z/0, y/1, y/1,
S3 : z/3, x/2, y/0,
Non-conforming solution!

===== SOLUTION 6 =====
      c    b    a
S0 : z/2, y/1, y/2,
S1 : x/1, y/0, x/3,
S2 : z/1, x/1, x/3,
S3 : z/3, x/0, y/0,
Non-conforming solution!

===== SOLUTION 7 =====
      c    b    a
S0 : z/2, x/1, x/3,
S1 : x/1, y/2, x/3,
S2 : z/1, y/1, y/0,
S3 : z/3, x/2, y/2,
Non-conforming solution!

===== SOLUTION 8 =====
      c    b    a
S0 : z/2, x/3, x/0,
S1 : x/1, y/3, x/0,
S2 : z/1, x/3, y/3,
S3 : x/1, y/1, y/0,
Conforming solution!

===== SOLUTION 9 =====
      c    b    a
S0 : z/2, y/1, y/1,
S1 : x/2, x/1, x/3,
S2 : z/2, y/2, y/1,
S3 : z/3, x/0, y/2,
Non-conforming solution!

total solutions = 9, non-conforming solutions = 8
```

(4) T4:

```
===== SOLUTION 1 =====
      c    a    b
S0 : z/0, y/0, y/3,
S1 : x/0, x/3, x/2,
S2 : z/1, y/1, x/1,
S3 : z/2, y/2, y/0,
Non-conforming solution!

===== SOLUTION 2 =====
      c    a    b
S0 : z/0, y/0, y/3,
S1 : x/0, x/3, x/2,
S2 : z/1, y/1, x/1,
S3 : z/2, y/2, y/3,
Non-conforming solution!

===== SOLUTION 3 =====
      c    a    b
S0 : z/0, y/0, y/3,
S1 : x/3, x/3, x/2,
S2 : z/1, y/1, x/1,
S3 : z/2, y/2, y/3,
Non-conforming solution!

===== SOLUTION 4 =====
      c    a    b
S0 : z/0, y/2, y/0,
S1 : x/0, x/3, x/3,
S2 : z/1, y/0, x/1,
S3 : z/2, y/1, */*,
Non-conforming solution!
```

```
===== SOLUTION 5 =====
      c    a    b
S0 : z/0, y/2, y/0,
S1 : x/3, x/3, x/3,
S2 : z/1, y/0, x/1,
S3 : z/2, y/1, y/0,
Non-conforming solution!
```

```
===== SOLUTION 6 =====
      c    a    b
S0 : z/0, y/3, y/0,
S1 : x/0, x/3, x/2,
S2 : z/1, y/1, x/2,
S3 : z/2, y/0, x/1,
Non-conforming solution!
```

```
===== SOLUTION 7 =====
      c    a    b
S0 : z/1, x/2, y/1,
S1 : x/0, y/3, y/1,
S2 : z/0, x/3, x/3,
S3 : z/2, y/2, x/1,
Non-conforming solution!
```

```
===== SOLUTION 8 =====
      c    a    b
S0 : z/1, x/2, */*,
S1 : x/1, y/3, y/1,
S2 : z/0, x/3, x/3,
S3 : z/2, y/2, x/1,
Non-conforming solution!
```

```
===== SOLUTION 9 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/1, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : x/1, x/0, y/1,
Non-conforming solution!
```

```
===== SOLUTION 10 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/1, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : x/3, x/0, y/1,
Non-conforming solution!
```

```
===== SOLUTION 11 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/1, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : z/1, x/2, y/1,
Non-conforming solution!
```

```
===== SOLUTION 12 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : x/1, x/0, y/1,
Non-conforming solution!
```

```
===== SOLUTION 13 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : x/3, x/0, y/1,
Non-conforming solution!
```

```
===== SOLUTION 14 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/1,
S2 : z/3, y/1, x/1,
S3 : z/1, x/2, y/1,
Non-conforming solution!
```

```
===== SOLUTION 15 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/3,
S2 : z/3, y/1, x/1,
S3 : x/3, x/0, y/1,
Conforming solution!
```

```
===== SOLUTION 16 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/3,
S2 : z/3, y/1, x/1,
S3 : z/1, x/2, y/1,
Non-conforming solution!
```

```
===== SOLUTION 17 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/1, y/0, y/1,
S2 : z/3, y/2, x/1,
S3 : z/1, x/2, */*,
Non-conforming solution!
```

```
===== SOLUTION 18 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/2, y/0, y/1,
S2 : z/3, y/2, y/2,
S3 : z/1, x/2, x/1,
Non-conforming solution!
```

```
===== SOLUTION 19 =====
      c    a    b
S0 : z/2, x/0, x/1,
S1 : x/3, y/0, y/1,
S2 : z/3, y/2, x/1,
S3 : z/1, x/2, y/1,
Non-conforming solution!
```

```
===== SOLUTION 20 =====
      c    a    b
S0 : z/2, x/0, x/3,
S1 : x/1, x/0, y/2,
S2 : z/1, y/3, y/1,
S3 : z/0, y/0, x/1,
Non-conforming solution!
```

```
===== SOLUTION 21 =====
      c    a    b
S0 : z/2, x/0, x/3,
S1 : x/1, x/0, y/2,
S2 : z/1, y/3, y/2,
S3 : z/0, y/0, x/1,
Non-conforming solution!
```

```
===== SOLUTION 22 =====
      c    a    b
S0 : z/2, x/0, x/3,
S1 : x/2, x/0, y/2,
S2 : z/1, y/3, y/2,
S3 : z/0, y/0, x/1,
Non-conforming solution!
```

```
===== SOLUTION 23 =====
      c    a    b
S0 : z/2, y/1, x/0,
S1 : x/2, x/0, x/0,
S2 : z/1, y/3, y/2,
S3 : z/0, y/3, x/1,
Non-conforming solution!
```

```
===== SOLUTION 24 =====
      c    a    b
S0 : z/3, x/2, x/1,
S1 : x/1, y/0, y/1,
S2 : z/2, x/0, */*,
S3 : z/1, y/1, x/1,
Non-conforming solution!
```

```
===== SOLUTION 25 =====
      c    a    b
S0 : z/3, x/2, x/1,
S1 : x/2, y/0, y/1,
S2 : z/2, x/0, y/1,
S3 : z/1, y/1, x/1,
Non-conforming solution!
```

```
===== SOLUTION 26 =====
      c    a    b
S0 : z/3, y/1, x/1,
S1 : x/2, x/3, x/0,
S2 : z/1, y/0, y/2,
S3 : z/2, y/2, x/0,
Non-conforming solution!
```

```
===== SOLUTION 27 =====
      c    a    b
S0 : z/3, y/1, x/1,
S1 : x/3, x/3, x/0,
S2 : z/1, y/0, y/2,
S3 : z/2, y/2, y/3,
Non-conforming solution!

total solutions = 27, non-conforming solutions = 26
```

(5) T2a:

```
===== SOLUTION 1 =====
      c    b    a
S0 : z/2, x/1, x/0,
S1 : x/1, y/1, y/0,
S2 : z/3, x/1, y/1,
S3 : x/1, y/1, x/0,
Non-conforming solution!
```

```
===== SOLUTION 2 =====
      c    b    a
S0 : z/2, x/1, x/0,
S1 : x/1, y/1, y/0,
S2 : z/3, x/1, y/1,
S3 : x/3, y/1, x/0,
Non-conforming solution!
```

```
===== SOLUTION 3 =====
      c    b    a
S0 : z/2, x/1, x/0,
S1 : x/3, y/1, y/0,
S2 : z/3, x/1, y/1,
S3 : x/1, y/1, x/0,
Non-conforming solution!
```

```
===== SOLUTION 4 =====
      c    b    a
S0 : z/2, x/1, x/0,
S1 : x/3, y/1, y/0,
S2 : z/3, x/1, y/1,
S3 : x/3, y/1, x/0,
Non-conforming solution!
```

```
===== SOLUTION 5 =====
      c    b    a
S0 : z/2, x/1, x/0,
S1 : x/3, y/3, y/0,
S2 : z/3, x/1, y/1,
S3 : x/3, y/1, x/0,
Conforming solution!

total solutions = 5, non-conforming solutions = 4
```