Conformance Testing

David Lee and Mihalis Yannakakis

Bell Laboratories, Lucent Technologies

600 Mountain Avenue, RM 2C-423

Murray Hill, NJ 07974

System reliability can not be overemphasized in software engineering as large and complex systems are being built to fulfill complicated tasks. Consequently, testing is an indispensable part of system design and implementation; yet it has proved to be a formidable task for complex systems. Testing software contains very wide fields with an extensive literature. See the articles in this volume. We discuss testing of software systems that can be modeled by finite state machines or their extensions to ensure that the implementation conforms to the design.

A finite state machine contains a finite number of states and produces outputs on state transitions after receiving inputs. Finite state machines are widely used to model software systems such as communication protocols. In a testing problem we have a specification machine, which is a design of a system, and an implementation machine, which is a ‘‘black box’’ for which we can only observe its I/O behavior. The task is to test whether the implementation conforms to the specification. This is called the *conformance testing* or *fault detection* problem. A test sequence that solves this problem is called a *checking sequence*.

Testing finite state machines has been studied for a very long time starting with Moore's seminal 1956 paper on ‘‘gedanken-experiments’’ (31), which introduced the basic framework for testing problems. Among other fundamental problems, Moore posed the conformance testing problem, proposed an approach, and asked for a better solution. A partial answer was offered by Hennie in an influential paper (14) in 1964: he showed that if the specification machine has a distinguishing sequence of length L then one can construct a checking sequence of length polynomial in L and the size of the machine. Unfortunately, not every machine has a distinguishing sequence. Hennie also gave another nontrivial construction of checking sequences in case a machine does not have a distinguishing sequence; in general however, his checking sequences are exponentially long. Several papers were published in the 60's on testing problems, motivated mainly by automata theory and testing switching circuits. Kohavi's book gives a good exposition of the major results (18), see also (12). During the late 60's and early 70's there were a lot of activities in the Soviet literature, which are apparently not well known in the West. An important paper on fault detection was by Vasilevskii (37) who proved polynomial upper and lower bounds on the length of checking sequences. However, the upper bound was obtained by an existence proof, and he did not present an algorithm for constructing efficiently checking sequences. For machines with a reliable reset, i.e., at any moment the machine can be taken to an initial state, Chow developed a method that constructs a checking sequence in polynomial time (9). There was very little activity subsequently until the late 80's when the fault detection problem was resurrected and is now being studied extensively anew due to its applications in testing communications protocol software systems (see (25) for a detailed survey and references).

After introducing some basic concepts of finite state machine, we discuss various techniques of conformance testing. In the first part of this article, we describe several test generation methods based on status messages, reliable reset, distinguishing sequences, identifying sequences, characterization sets, transition tours and UIO sequences, and finally a randomized polynomial time algorithm. Finite state machines model well some software systems and control portions of protocols. However, often in practice systems contain variables and their operations depend on variable values; finite state machines are not powerful enough to model in a succinct way such systems. Extended finite state machines, which are finite state machines extended with variables, have emerged from the design and analysis of both circuits and communication protocols as a more convenient model. We discuss conformance testing of extended finite state machines in the second part of this article.

Background

Finite state systems can usually be modeled by *Mealy* machines that produce outputs on their state transitions after receiving inputs.

Definition 1. A finite state machine (FSM) M is a quintuple $M = (I, O, S, \delta, \lambda)$ where I, O, and S are finite and nonempty sets of input symbols, output symbols, and states, respectively. δ: $S \times I \to S$ is the state transition function; and λ: $S \times I \to O$ is the output function. When the machine is in a current state s in S and receives an input a from I it moves to the next state specified by $\delta(s, a)$ and produces an output given by $\lambda(s, a)$. □

There is a variant of the model in which outputs are associated with the states (instead of the transitions); the following theory and testing methods apply also to this model. An FSM can be represented by a *state transition diagram*, a directed graph whose vertices correspond to the states of the machine and whose edges correspond to the state transitions; each edge is labeled with the input and output associated with the transition. For the FSM in Fig. 1, suppose that the machine is currently in state s_1. Upon input *b*, the machine moves to state s_2 and outputs 1. We denote the number of states, inputs, and outputs by $n = |S|$, $p = |I|$, and $q = |O|$, respectively. We extend the transition function δ and output function λ from input symbols to strings as follows: for an initial state s_1, an input sequence $x = a_1,...,a_k$ takes the machine successively to states $s_{i+1} = \delta(s_i, a_i)$, $i = 1, \cdots, k$, with the final state $\delta(s_1, x) = s_{k+1}$, and produces an output sequence $\lambda(s_1, x) = b_1,...,b_k$, where $b_i = \lambda(s_i, a_i)$, $i = 1, \cdots, k$. Suppose that the machine in Fig. 1 is in state s_1. Input sequence *abb* takes the machine through states s_1, s_2, and s_3, and outputs 011.

Two states s_i and s_j are *equivalent* if and only if for every input sequence the machine will produce the same output sequence regardless of whether s_i or s_j is the initial state; i.e., for an arbitrary input sequence *x*, $\lambda(s_i, x) = \lambda(s_j, x)$. Otherwise, the two states are *inequivalent*, and there exists an input sequence *x* such that $\lambda(s_i, x) \neq \lambda(s_j, x)$; in this case, such an input sequence is called a *separating* sequence of the two inequivalent states. For two states in different machines with the same input and output sets, equivalence is defined similarly. Two machines *M* and *M′* are *equivalent* if and only for every state in *M* there is a corresponding equivalent state in *M′*, and vice versa. Two machines are *isomorphic* if they are identical except for a renaming of states. Note that any two isomorphic machines are equivalent, but not necessarily vice-versa. Given a machine, we can ‘‘merge’’ equivalent states and construct a *minimized (reduced)* machine which is equivalent to the given machine and no two states are equivalent. The minimized machine is unique up to isomorphism. We can construct in polynomial time a minimized machine and also obtain separating sequences for each pair of states (18).

We define now within a uniform framework some important types of sequences. A *separating family* of sequences for a FSM *A* is a collection of *n* sets Z_i, $i = 1,...,n$, of sequences (one set for each state) such that for every pair of states s_i, s_j there is an input string α that: (1) separates them, i.e., $\lambda_A(s_i,\alpha) \neq \lambda_A(s_j,\alpha)$; and (2) α is a prefix of some sequence in Z_i and some sequence in Z_j. We call Z_i the *separating* set of state s_i, and the elements of Z_i its separating sequences. If a separating family has the same set *Z* for all the states (i.e. $Z = Z_i$ for all *i*), then the set *Z* is called a *characterizing set*. Every reduced FSM has a *characterizing set* containing at most $n - 1$ sequences each of length no more than $n - 1$. The same is true for separating families, although they provide more flexibility (since one can use a different set for each state) and thus may have fewer and shorter sequences. If there is a characterizing set *Z* that contains only one sequence *x*, then *x* is called a *(preset) distinguishing sequence*. Note that if we input the sequence *x* to the machine, then every state gives a different output; hence a distinguishing sequence allows us to identify the initial state of a machine. Unfortunately, not every reduced machine has a distinguishing sequence; furthermore it is possible that there is such a sequence but only of exponential length, and it is a computationally intractable problem to determine if a given machine has a preset distinguishing sequence (24). A separating family in which all sets Z_i are singletons (though possibly distinct for different states) forms what is called an *adaptive distinguishing sequence*; it provides a way for identifying the initial state of a machine using an adaptive test, i.e., a test in which the input symbol that is applied at each step may depend on the previously observed output symbols. Again, not every reduced machine has an adaptive distinguishing sequence, but unlike the preset case, we can determine efficiently if there exists an adaptive distinguishing sequence, and if so, we can construct one of polynomial length (24).

Given a reduced FSM *A* with *n* states, a separating family of sequences Z_i for each state s_i, and an FSM *B* with the same input and output symbols, we say that a state q_i of *B* is *similar* to a state s_i of *A* if it agrees (gives the same output) on all sequences in the separating set Z_i of s_i. A key property is that q_i can be similar to at most one state of *A*. Let us say that an FSM *B* is *similar* to *A*, if for each state s_i of *A*, the machine *B* has a corresponding state q_i similar to it. Note that then all the q_i's must be distinct. If we know that *B* has at most *n* states, then there is a one-to-one correspondence between similar states of *A* and *B*. For *B* to be equivalent to *A*, it needs to be isomorphic (since *A* is reduced). That is, the ultimate goal is to check if an implementation machine *B* is isomorphic to a specification machine *A*. Often we first check their similarity and then isomorphism.

Systems modeled by finite state machines

Given a complete description of a *specification* machine *A*, we want to determine whether an *implementation* machine *B*, which is a ‘‘black-box’’, is isomorphic to *A*. Obviously, without any assumptions the problem is impossible to solve; for any test sequence we can easily construct a machine *B*, which is not equivalent to *A* but produces the same outputs as *A* for the given test sequence. There is a number of natural assumptions that are usually made in the literature in order for the test to be at all possible. (1) Specification machine *A* is strongly connected, i.e, there is a path between every pair of states; otherwise, during a test some states may not be reachable. (2) Machine *A* is reduced; otherwise, we can always minimize it first. (3) Implementation machine *B* does not change during the experiment and has the same input alphabet as *A*. (4) Machine *B* has no more states than *A*. Assumption (4) deserves a comment. An upper bound must be placed on the number of states of *B*; otherwise, no matter how long our test is, it is possible that it does not reach the ‘‘bad’’ part of *B*. The usual assumption made in the literature, and which we will also adopt is that the faults do not increase the number of states of the machine. In other words, under this assumption, the faults are of two types: *output faults*, i.e., one or more transitions may produce wrong outputs, and *transfer faults*, i.e., transitions may go to wrong next states. Under these assumptions, we want to design an experiment that tests whether *B* is isomorphic to *A*. From assumptions (2) and (4), *B* is isomorphic to *A* if and only if *B* is equivalent to *A*.

Suppose that the implementation machine *B* starts from an unknown state and that we want to check whether it is isomorphic to *A*. We first apply a sequence that is supposed to bring *B* (if it is correct) to a known state s_1 that is the initial state for the main part of the test; such a sequence is called a *homing sequence* (18). Then we verify that *B* is isomorphic to *A* using a *checking sequence*, which is to be defined in the sequel. However, if *B* is not isomorphic to *A*, then the homing sequence may or may not bring *B* to s_1; in either case, a checking sequence will detect faults: a discrepancy between the outputs from *B* and the expected outputs from *A* will be observed. From now on we assume that a homing sequence has taken the implementation machine *B* to a supposedly initial state s_1 before we conduct a conformance test.

Definition 2. Let *A* be a specification FSM with *n* states and initial state s_1. A *checking sequence* for *A* is an input sequence *x* that distinguishes *A* from all other machines with *n* states; i.e., every (implementation) machine *B* with at most *n* states that is not isomorphic to *A* produces on input *x* a different output than that produced by *A* starting from s_1. □

All the proposed methods for checking experiments have the same basic structure. We want to make sure that every transition of the specification FSM *A* is correctly implemented in FSM *B*; so for every transition of *A*, say from state s_i to state s_j on input *a*, we want to apply an input sequence that transfers the machine to s_i, apply input *a*, and then verify that the end state is s_j by applying appropriate inputs. The methods differ by the types of subsequences they use to verify that the machine is in a right state. This can be accomplished by status messages, separating family of sequences, characterizing sequences, distinguishing sequences, UIO sequences, and identifying sequences, depending on what types of sequences the given specification machine possesses.

Status messages

A *status message* tells us the current state of a machine. Conceptually, we can imagine that there is a special input *status*, and upon receiving this input, the machine outputs its current state and stays there. Such status messages do exist in practice. In protocol testing, one might be able to dump and observe variable values which represent the states of a protocol machine.

With a status message, the machine is highly observable at any moment. We say that the status message is *reliable* if it is guaranteed to work reliably in the implementation machine *B*; i.e., it outputs the current state without changing it. Suppose the status message is reliable. Then a checking sequence can be easily obtained by simply constructing a covering path of the transition diagram of the specification machine *A*, and applying the status message at each state visited (32) (36). Since each state is checked with its status message, we verify whether *B* is similar to *A*. Furthermore, every transition is tested because its output is observed explicitly, and its start and end state are verified by their status messages; thus such a covering path provides a checking sequence. If the status message is not reliable, then we can still obtain a checking sequence by applying the status message twice in a row for each state s_i at some point during the experiment when the covering path visits s_i; we only need to have this double application of the status message once for each state and have a single application in the rest of the visits. The double application of the status message ensures that it works properly for every state.

For example, consider the specification machine *A* in Fig. 1, starting at state s_1. We have a covering path from input sequence $x = ababab$. Let *s* denote the status message. If it is reliable, then we obtain the checking sequence *sasbsasbsasbs*. If it is unreliable, then we have the sequence *ssasbssasbssasbs*.

Reset

We say that machine *A* has a *reset capability* if there is an initial state s_1 and an input symbol *r* that takes the machine from any state back to s_1, i.e., $\delta_A(s_i, r) = s_1$ for all states s_i. We say that the reset is *reliable* if it is guaranteed to work properly in the implementation machine *B*, i.e., $\delta_B(s_i, r) = s_1$ for all s_i; otherwise it is *unreliable*.

For machines with a reliable reset, there is a polynomial time algorithm for constructing a checking sequence (6) (9) (37). Let Z_i, $i = 1,...,n$ be a family of separating sets; as a special case the sets could all be identical (i.e., a characterizing set). We first construct a breadth-first-search tree (or any spanning tree) of the transition diagram of the specification machine *A* and verify that *B* is similar to *A*; we check states according to the breadth-first-search order and tree edges (transitions) leading to the nodes (states) as follows. For every state s_i, we have a part of the checking sequence that does the following for every member of Z_i: first it resets the machine to s_1 by input *r*, then it applies the input sequence (say p_i) corresponding to the path of the tree from the root s_1 to s_i and then applies a separating sequence in Z_i. If the implementation machine *B* passes this test for all members of Z_i, then we know that it has a state similar to s_i, namely the state that is obtained by applying the input sequence p_i starting from the reset state s_1. If *B* passes this test for all states s_i, then we know that *B* is similar to *A*. This portion of the test also verifies all the transitions of the tree. Finally, we check nontree transitions as follows. For every transition, say from state s_i to state s_j on input *a*, we do the following for every member of Z_j: reset the machine, apply the input sequence p_i taking it to the start node s_i of the transition along tree edges, apply the input *a* of the transition, and then apply a separating sequence in Z_j. If the implementation machine *B* passes this test for all members of Z_j then we know that the transition on input *a* of the state of *B* that is similar to s_i gives the correct output and goes to the state that is similar to state s_j. If *B* passes the test for all the transitions, then we can conclude that it is isomorphic to *A*.

For the machine in Fig. 1, a family of separating sets is: $Z_1 = \{ a, b \}$, $Z_2 = \{ a \}$, and $Z_3 = \{ a, b \}$. A spanning tree is shown in Fig. 2 with thick tree edges. Sequences *ra* and *rb* verify state s_1. Sequence *rba* verifies state s_2 and transition (s_1, s_2): after resetting, input *b* verifies the tree edge transition from s_1 to s_2 and separating sequence *a* of Z_2 verifies the end state s_2. The following two sequences verify state s_3 and the tree edge transition from s_2 to s_3: *rbba* and *rbbb* where the prefix *rbb* resets the machine to s_1 and takes it to state s_3 along verified tree edges, and the two suffixes *a* and *b* are the separating sequences of s_3. Finally, we test nontree edges in the same way. For instance, the self-loop at s_2 is checked by the sequence *rbaa*.

With reliable reset the total cost is $O(pn^3)$ to construct a checking sequence of length $O(pn^3)$. This bound on the length of the checking sequence is in general best possible (up to a constant factor); there are specification machines *A* with reliable reset such that any checking sequence requires length $\Omega(pn^3)$ (37). For machines with unreliable reset, only randomized polynomial time algorithms are known (41); we can construct with high probability in randomized polynomial time a checking sequence of length $O(pn^3 + n^4 \log n)$.

Distinguishing sequences

For specification machines with a *distinguishing* sequence there is a deterministic polynomial time algorithm to construct a checking sequence (14) (18). of length polynomial in the length of the distinguishing sequence. A distinguishing sequence is similar to an unreliable status message in that it gives a different output for each state, except that it changes the state. For example, for the machine in Fig. 1, *ab* is a distinguishing sequence, since $\lambda(s_1, ab) = 01$, $\lambda(s_2, ab) = 11$, and $\lambda(s_3, ab) = 00$.

Given a distinguishing sequence x_0, first check the similarity of the implementation machine by examining the response of each state to the distinguishing sequence, then check each transition by exercising it and verifying the ending state, also using the distinguishing sequence. A *transfer* sequence $\tau(s_i, s_j)$ is a sequence that takes the machine from state s_i to s_j. Such a sequence always exists for any two states since the machine is strongly connected. Obviously, it is not unique and a shortest path (3) (10) from s_i to s_j in the transition diagram is often preferable. Suppose that the machine is in state s_i and that distinguishing sequence x_0 takes the machine from state s_i to t_i, i.e., $t_i = \delta(s_i, x_0)$, $i = 1, \cdots, n$. For the machine in the initial state s_1, the following test sequence takes the machine through each of its states and displays each of the *n* different responses to the distinguishing sequence:

$$x_0\tau(t_1, s_2)x_0\tau(t_2, s_3)x_0 \cdots x_0\tau(t_n, s_1)x_0 \ . \qquad (1)$$

Starting in state s_1, x_0 takes the machine to state t_1 and then $\tau(t_1, s_2)$ transfers it to state s_2 for its response to x_0. At the end the machine responds to $x_0\tau(t_n, s_1)$. If it operates correctly, it will be in state s_1, and this is verified by its response to the final x_0. During the test we should observe *n* different responses to the distinguishing sequence x_0 from *n* different states, and this verifies that the implementation machine *B* is similar to the specification machine *A*.

We then establish every state transition. Suppose that we want to check transition from state s_i to s_j with input/output *a/o* when the machine is currently in state t_k. We would first take the machine from t_k to s_i, apply input *a*, observe output *o*, and verify the ending state s_j. We cannot simply use $\tau(t_k, s_i)$ to take the machine to state s_i, since faults may alter the ending state. Instead, we apply the following input sequence: $\tau(t_k, s_{i-1})x_0\tau(t_{i-1}, s_i)$. The first transfer sequence is supposed to take the machine to state s_{i-1}, which is verified by its response to x_0, and as has been verified by (1), $x_0\tau(t_{i-1}, s_i)$ definitely takes the machine to state s_i. We then test the transition by input *a* and verify the ending state by x_0. Therefore, the following sequence tests for a transition from s_i to s_j:

$$\tau(t_k, s_{i-1})x_0\tau(t_{i-1}, s_i)ax_0 \qquad (2)$$

After this sequence the machine is in state t_j. We repeat the same process for each state transition and obtain a checking sequence. Observe that the length of the checking sequence is polynomial in the size of the machine *A* and the length of the distinguishing sequence x_0.

Recall that a distinguishing sequence for the machine in Fig. 1 is $x_0 = ab$. The transfer sequences are straightforward, for example, $\tau(s_1, s_2) = b$. The sequence in (1) for checking states is *abababab*. Suppose that the machine is in state s_3. Then the following sequence *babbab* tests for the transition from s_2 to s_3: *b* takes the machine to state s_1, *ab* definitely takes the machine to state s_2 if it produces outputs 01, which we have observed during state testing, and, finally, *bab* tests the transition on input *b* and the end state s_3. Other transitions can be tested similarly.

We can use *adaptive* distinguishing sequences to construct a checking sequence. An adaptive distinguishing sequence is not really a sequence but an adaptive experiment (i.e. a decision tree) that specifies how to choose inputs adaptively based on observed outputs to identify the initial state. An adaptive distinguishing sequence corresponds to a separating family in which each state s_i has only one separating sequence x_i in its set, i.e., $Z_i = \{x_i\}$. We can construct a checking sequence using the same construction as above with the following difference: at each step where we are supposed to apply the distinguishing sequence x_0, we apply instead the separating sequence x_i for the current state s_i. An adaptive distinguishing sequence has length $O(n^2)$, and, consequently, a checking sequence of length $O(pn^3)$ can be constructed in time $O(pn^3)$ (see

(24) for the details).

Identifying sequences

The previous three methods are based on knowing where we are during the experiment, using status messages, reset, and distinguishing sequences, respectively. However, these sequences may not exist in general. A method was proposed by Hennie that works for general machines, although it may yield exponentially long checking sequences. It is based on certain sequences, called *identifying sequences* in (18) (*locating sequences* in (14)) that identify a state in the *middle* of the execution. Identifying sequences always exist and checking sequences can be derived from them (14) (18).

Similar to checking sequences from distinguishing sequences, the main idea is to display the responses of each state to its separating family of sequences instead of one distinguishing sequence. We use an example to explain the display technique. The checking sequence generation procedure is similar to that from the distinguishing sequences and we omit the detail.

Consider machine *A* in Fig. 1. We want to display the responses of state s_1 to separating sequences *a* and *b*. Suppose that we first take the machine to s_1 by a transfer sequence, apply the first separating sequence *a*, and observe output 0. Due to faults, there is no guarantee that the implementation machine was transferred to state s_1 in the first place. Assume instead that we transfer the machine (supposedly) to s_1 and then apply *aaa* which produces output 000. The transfer sequence takes the machine *B* to state q_0 and then *aaa* takes it through states q_1, q_2, and q_3, and produces outputs 000 (if not, then *B* must be faulty). The four states q_0 to q_3 cannot be distinct since *B* has at most three states. Note that if two states q_i, q_j are equal, then their respective following states q_{i+1}, q_{j+1} (and so on) are also equal because we apply the same input *a*. Hence q_3 must be one of the states q_0, q_1, or q_2, and thus we know that it will output 0 on input *a*; hence we do not need to apply *a*. Instead we apply input *b* and must observe output 1. Therefore, we have identified a state of *B* (namely q_3); that responds to the two separating sequences *a* and *b* by producing 0 and 1 respectively, and thus is similar to state s_1 of *A*.

The length of an identifying sequence in the above construction grows exponentially with the number of separating sequences of a state and the resulting checking sequence is of exponential length in general.

A Polynomial time randomized algorithm

With status messages, reset, or short distinguishing sequences (of at most polynomial length), we can find in polynomial time checking sequences of polynomial length. In the general case without such information, Hennie's algorithm constructs an exponential length checking sequence. The reason of the exponential growth of the length of the test sequence is that it deterministically displays the response of each state to its separating family of sequences. Randomization can avoid this exponential "blow-up"; we now describe a polynomial time randomized algorithm that constructs with high probability a polynomial length checking sequence (41). The probabilities are with respect to the random decisions of the algorithm; we do not make any probabilistic assumptions on the specification *A* or the implementation *B*. For a test sequence to be considered ‘‘good’’ (a checking sequence), it must be able to uncover *all* faulty machines *B*. As usual, ‘‘high probability’’ means that we can make the probability of error arbitrarily small by repeating the test enough times (doubling the length of the test squares the probability that it is not a checking sequence).

We break the checking experiment into two tests. The first test ensures with high probability that the implementation machine *B* is similar to *A*. The second test ensures with high probability that all the transitions are correct: they give the correct output and go to the correct next state.

```
Test 1. (Similarity)
    For $i = 1$ to $n$ do
        Repeat the following $k_i$ times:
            Apply an input sequence that takes $A$ from its current state to state $s_i$;
            Choose a separating sequence from $Z_i$ uniformly at random and apply it. □
```

We assume that for every pair of states we have chosen a fixed transfer sequence from one state to the other. Assume that z_i is the number of separating sequences in Z_i for state s_i. Let x be the random input string formed by running Test 1 with $k_i = O(nz_i \min(p, z_i) \log n)$ for each $i = 1,...,n$. It can be shown that, with high probability, every FSM B (with at most n states) that is not similar to A produces a different output than A on input x.

```
Test 2. (Transitions)
    For each transition of the specification FSM $A$, say $\delta_A(s_i, a) = s_j$, do
        Repeat the following $k_{ij}$ times:
            Take the specification machine $A$ from its current state to state $s_i$;
            Flip a fair coin to decide whether to check the current state or the transition;
            In the first case, choose (uniformly) at random a sequence from $Z_i$ and apply it;
            In the second case, apply input $a$ followed by a randomly selected sequence from $Z_j$. □
```

Let x be the random input string formed by running Test 2 with $k_{ij} = O(\max(z_i, z_j) \log(pn))$ for all i, j. It can be shown that, with high probability, every FSM B (with at most n states) that is similar but not isomorphic to A produces a different output than A on input x.

Combining the two tests, we obtain a checking sequence with a high probability (41). Specifically, given a specification machine A with n states and input alphabet of size p, the randomized algorithm constructs with high probability a checking sequence for A of length $O(pn^3 + p'n^4 \log n)$ where $p' = \min(p, n)$.

In our exposition we have assumed that the specification is a completely specified FSM. Similar methods apply to partially specified machines, as long as the relevant sequences exist. The methods can be also extended to the case of faults that introduce additional states, although in this case the tests become inherently longer (see (25) for further discussion).

Heuristic procedures and optimizations

Checking sequences guarantee a complete fault coverage but sometimes could be too long for practical applications and heuristic procedures are used instead. For example, in circuit testing, test sequences are generated based on fault models that significantly limit the possible faults (1). Without fault models, covering paths are often used in both circuit testing (1) (12) and protocol testing where a test sequence exercises each transition of the specification machine at least once. A short test sequence is always preferred and a shortest covering path is desirable, resulting in a Postman Tour (2) (11) (20) (32) (36).

A covering path is easy to generate yet may not have a high fault coverage. Additional checking is needed to increase the fault coverage. For instance, suppose that each state has a *UIO sequence* (33). A UIO sequence for a state s_j is an input sequence x_j that distinguishes s_j from any other states, i.e., for any state $s_k \neq s_j$, $\lambda(s_j, x_j) \neq \lambda(s_k, x_j)$. To increase the coverage we may test a transition from state s_i to s_j by its I/O behavior and then apply a UIO sequence of s_j to verify that we end up in the right state. Suppose that such a sequence takes the machine to state t_j. Then a test of this transition is represented by a test sequence, which takes the machine from s_i to t_j. Imagine that all the edges of the transition diagram have a white color. For each transition from s_i to s_j, we add a red edge from s_i to t_j due to the additional checking of a UIO sequence of s_j. A test that checks each transition along with a UIO sequence of its end state requires that we find a path that exercises each red edge at least once. It provides a better fault coverage than a simple covering path, although such a path does not necessarily give a checking sequence (6). We would like to find a shortest path that covers each red edge at least once. This is a *Rural Postman Tour*

(13), and in general, it is an NP-hard problem. However, practical constraints are investigated and polynomial time algorithms can be obtained for a class of communication protocols (2).

Sometimes, the system is too large to construct and we cannot even afford a covering path. To save space and to avoid repeatedly testing the same portion of the system, a ‘‘random walk’’ could be used for test generation (22) (38). Basically, we only keep track of the current state and determine the next input online; for all the possible inputs with the current state, we choose one at random. Note that a pure random walk may not work well in general; as is well known, a random walk can easily get ‘‘trapped’’ in one part of the machine and fail to visit other states if there are ‘‘narrow passages’’. Consequently, it may take exponential time for a test to reach and uncover faulty parts of an implementation machine through a pure random walk. Indeed, this is very likely to happen for machines with low enough connectivity and few faults (single fault, for instance). To avoid such problems, a *guided random walk* was proposed (22) for protocol testing where partial information of a history of the tested portion is being recorded. Instead of a random selection of next input, priorities based on the past history are enforced; on the other hand, we make a random choice within each class of inputs of the same priority. Hence we call it a guided random walk; it may take the machine out of the ‘‘traps’’ and increase the fault coverage.

In the techniques discussed, a test sequence is formed by combining a number of subsequences, and often there is a lot of overlaps in the subsequences. There are several papers in the literature that propose heuristics for taking advantage of overlaps in order to reduce the total length of tests (8) (35) (39).

Systems modeled by extended finite state machines

In software applications such as feature testing of communication protocols, the pure finite state machine model is not powerful enough to model in a succinct way the actual systems any more. Extended finite state machines, which are finite state machines extended with variables, are commonly used to specify such systems. For instance, IEEE 802.2 LLC (5) is specified by 14 control states, a number of variables, and a set of transitions (pp. 75-117). For example, a typical transition is (p. 96):

current_state SETUP
input ACK_TIMER_EXPIRED
predicate S_FLAG = 1
output CONNECT_CONFIRM
action P_FLAG := 0; REMOTE_BUSY := 0
next_state NORMAL

In state SETUP and upon input ACK_TIMER_EXPIRED, if variable S_FLAG has value 1, then the machine outputs CONNECT_CONFIRM, sets variables P_FLAG and REMOTE_BUSY to 0, and moves to state NORMAL.

To model this and other protocols, including other ISO standards and complicated systems such as 5ESS (Lucent No. 5 Electronic Switching System) we extend finite state machines with variables as follows. We denote a finite set of variables by a vector: $\vec{x} = (x_1, \cdots, x_k)$. A predicate on variable values $P(\vec{x})$ returns FALSE or TRUE; a set of variable values $\vec{x}$ is valid for P if $P(\vec{x}) = \text{TRUE}$, and we denote the set of valid variable values by $X_P = \{\vec{x} : P(\vec{x}) = \text{TRUE}\}$. Given a function $A(\vec{x})$, an action is an assignment: $\vec{x} := A(\vec{x})$.

Definition 3. An *extended finite state machine* (EFSM) is a quintuple $M = (I, O, S, \vec{x}, T)$ where I, O, S, $\vec{x}$, and T are finite sets of input symbols, output symbols, states, variables, and transitions, respectively. Each transition t in the set T is a 6-tuple: $t = (s_t, q_t, a_t, o_t, P_t, A_t)$ where s_t, q_t, a_t, and o_t are the start (current) state, end (next) state, input, and output, respectively. $P_t(\vec{x})$ is a predicate on the current variable values and $A_t(\vec{x})$ defines an action on variable values.

Initially, the machine is in an initial state $s_0 \in S$ with initial variable values: $\vec{x}_{init}$. Suppose that the machine is at state s_t with the current variable values $\vec{x}$. Upon input a_t, if $\vec{x}$ is valid for P_t, i.e., $P_t(\vec{x}) = \text{TRUE}$, then the machine follows the transition t, outputs o_t, changes the current variable values by action $\vec{x} := A_t(\vec{x})$, and moves to state q_t.

For each state $s \in S$ and input $a \in I$, let all the transitions with start state s and input a be: $t_i = (s, q_i, a, o_i, P_i, A_i)$, $1 \le i \le r$. We assume that the sets of valid variable values of these r predicates are mutually disjoint, i.e., $X_{P_i} \cap X_{P_j} = \emptyset$, $1 \le i \ne j \le r$. □

Clearly, if the variable set is empty and all predicates $P \equiv$ TRUE then an EFSM becomes an ordinary FSM. Each combination of a state and variable values is called a *configuration*. Given an EFSM, if each variable has a finite number of values (Boolean variables for instance), then there is a finite number of configurations, and hence there is an equivalent (ordinary) FSM with configurations as states. Therefore, an EFSM with finite variable domains is a compact representation of an FSM.

We now discuss testing of EFSM's, which has become an important topic recently, especially in the network protocol area (19) (26) (29). An EFSM usually has an initial state s_0 and all the variables have an initial value $\vec{x}_{init}$, which consists of the *initial configuration*. A *test sequence* (or a *scenario*) is an input sequence that takes the machine from the initial configuration back to the initial state (possibly with different variable values). We want to construct a set of test sequences of a desirable *fault coverage*, which ensures that the implementation machine under test *conforms* to the specification.

The fault coverage is essential. However, it is often defined differently from different models and/or practical needs. For testing FSM's we have discussed checking sequences, which guarantee that the implementation machine is structurally isomorphic to the specification machine. However, even for medium size machines it is too long to be practical (41) while for EFSM's hundreds of thousands of states (configurations) are typical and it is in general impossible to apply a checking sequence. A commonly used heuristic procedure in practice is to try to make sure that each transition in the specification EFSM is executed at least once.
Definition 4. A *complete test set* for an EFSM is a set of test sequences such that each transition is tested at least once. □

Given the succinct representation of EFSM's, one might imagine that it is an *easy* problem. As a matter of fact, even an apparently easier problem, the reachability problem, is hard where we want to determine if a control state is reachable from the initial state. Specifically, it is undecidable if the variable domains are infinite and PSPACE-complete otherwise.

To find a complete test set, we first construct a *reachability graph G*, which consists of all the configurations and transitions that are reachable from the initial configuration. We obtain a directed graph where the nodes and edges are the reachable configurations and transitions, respectively. Obviously, a control state may have multiple appearances in the nodes (along with different variable values) and each transition may appear many times as edges in the reachability graph. In this reachability graph, any path from the initial node (configuration) corresponds to a feasible path (test sequence) in the EFSM, since there are no predicate or action restrictions anymore. Therefore, a set of such paths in G, which exercises each transition at least once, provides a complete test set for the EFSM. We thus reduce the testing problem to a graph path covering problem.

The construction of the reachability graph is often a formidable task; it has the well-known state explosion problem due to the large number of possible combinations of the control states and variable values. One approach to this problem is to apply an on-line minimization algorithm to construct an equivalent graph $G_{\min}$, which collapses all configurations of the reachability graph that are equivalent in terms of the transitions that they can perform. Such a minimized graph can be constructed efficiently directly from the EFSM {23); $G_{\min}$ could be much smaller than G and can be used in its place for generating test sequences. Furthermore, for the testing purpose, we do not need a complete reachability graph; we only need a subgraph that contains all the transitions so that a set of covering paths still provides a complete test set (16). We shall not digress to this topic further. From now on we assume that we have a graph G that contains all the transitions of a given EFSM and we want to construct a complete test set of a small size. For clarity, we assume that each path (test sequence) is from the initial node to a *sink* node, which is a configuration with the initial control state.

To summarize, we have a directed graph with an initial node and a sink node. The nodes are configurations, which correspond to combinations of control states and variable values, and a state may appear in more than one node. The edges correspond to transitions, and a same transition may appear many times in the graph as edges between different configurations. We want to find a complete test set: a set of paths from the initial node to the sink node such that each transition in the *original* EFSM is covered; specifically, among the multiple appearances of a transition, it is sufficient to cover any one of them. Therefore, the test generation is reduced to covering path problems on graphs.

Test sequence generation

Formally, we have a directed graph $G = <V, E>$ with $n = |V|$ nodes, $m = |E|$ edges, a *source* node s of in-degree 0, and a *sink* node t of out-degree 0. All edges are reachable from the source node and the sink node is reachable from all edges. There is a set C of $k = |C|$ distinct *colors*. Each node and edge is associated with a subset of colors from C. Each transition in the EFSM corresponds to a distinct color in C and may have multiple appearances in G. We consider a more general case here; each node and edge have a set of colors from C. A path from the source to sink is called a *test*.

We are interested in a set of tests that cover all the colors; they are not necessarily the conventional covering paths that cover all the edges. Formally, a *complete test set* covers all the colors in C. The path (test) length makes little difference and we are interested in minimizing the number of paths. We shrink each strongly connected component (3) (10) into a node, which contains all the colors of the nodes and edges in the component. The problem then is reduced to that on a directed acyclic graph (DAG) (10). From now on, unless otherwise stated, we assume that the graph $G = <V, E>$ is a DAG. We now describe different test generation techniques, which correspond to path construction problems on graphs. For details see (26).

Minimal complete test set

We need a complete test set - a set of paths from the initial node to the sink node that cover all the colors. On the other hand, in the feature testing of communication systems, setting up and running each test is time consuming and each test is costly to experiment. Consequently, we want to minimize the number of tests. Therefore, our goal is: *Find a complete test set of minimum cardinality.* However, it turns out that the problem is NP-hard. We discuss a greedy method next.

Maximal color paths

We need to restrict ourselves to approximation algorithms. Similar to the standard approximation algorithm for Set Cover (17) (27), we use the following procedure. We first find a path (test) that covers a maximum number of colors and delete the covered colors from C. We then repeat the same process until all the colors have been covered. Thus, we have the following problem: *Find a test that covers the maximum number of colors.* This problem is also NP-hard.

In view of the NP-hardness of the problem, we have to content ourselves with approximation algorithms again. We now describe some heuristic methods.

Longest path

Suppose that an edge (node) has c uncovered colors so far. We assign a weight c to that edge (node), and we have a weighted graph. Ech path has an associated weight, which is the sum of the weights of its edges and nodes. We find a longest (maximum weight) path from the source to sink; it is possible since the graph is a DAG. This may not provide a maximal color test due to the multiple appearances of colors on a path.

However, if there are no multiple appearances of colors on the path, then it is indeed a maximal color test.

There are known efficient ways of finding a longest path in a DAG. We can first topologically sort the nodes and then compute the longest paths from each node to the sink in the reverse topological order (10). Specifically, suppose that we are processing node u and examine all its outgoing edges (u, v) where v is a node of higher topological ordering and has its longest path to the sink computed. Suppose that (u, v) has weight $w_{u,v}$ and that a longest path from v to sink has weight w_v. Then a path from u to v and then following a longest path from v to the sink has a weight $w_{u,v} + w_v$. We can easily compare all the outgoing edges from u and choose a longest path from u to the sink node.

The time and space needed is $O(m)$ where m is the number of edges. How does this heuristic method compare with the optimal solution? An obvious criterion is the *coverage ratio*: the number of maximal number of colors on a path over the number of colors covered by the algorithm. In the worst case it can be k, the number of uncovered colors.

A Greedy heuristic

We now discuss a greedy heuristic procedure. It takes linear time and works well in practice. We again topologically sort the nodes and compute a desired path from each node to the sink in a reverse topological order as follows. Instead of keeping the color sets of all the paths from a node to the sink, we only keep the one with a supposedly ‘‘maximum number’’ of colors. Specifically, when we process a node u and consider all the outgoing edges (u, v) where v has a higher topological order and has been processed, we take the union of the colors of node u, edge (u, v), and node v. We compare the resulting color sets from all the outgoing edges from u and keep one with the largest cardinality. This procedure is well defined since G is a DAG. However, it may not provide a maximum color coverage test; when we choose the outgoing edge from u, we do not incorporate information of the colors from the source to u.

Since we take unions of and compare color sets of no more than k colors, the time and space complexity of this approach is $O(km)$. where k is the number of uncovered colors and m is the number of edges. Although the second method seems to be better in many cases, its worst case coverage ratio is also $\Omega(k)$.

A Transitive greedy heuristic

We now discuss an improved procedure. This is similar to the greedy heuristic, except that when we process a node u, we do not consider only its immediate successors but all its descendants. Specifically, for each outgoing edge (u, v) and descendant v' of v (possibly $v = v'$), we take the union of the colors of node u, edge (u, v), and node v'. We compare the resulting color sets from all the outgoing edges from u and descendants v' and keep one with the largest cardinality.

The time complexity of this algorithm is $O(knm)$, since we may examine on the order of n descendants when we process a node. The worst case coverage ratio of this method is somewhat better: $O(\sqrt{k})$.

More on complexity of test generation

We now come back to the original minimum complete test set problem. Suppose that we successfully find a maximum color test repeatedly until we obtain a complete test set in N steps while the minimum complete test set contains N^* tests. How far is N from N^*? Is there a better algorithm? It follows from the results on the Set Cover problem that $N = \Theta(N^* \log k)$ (17) (27). That is, on the one hand, for any instance, if we can find repeatedly maximum color tests, then the complete test set will contain at most $N^* \log k$ tests; moreover, an approximation within factor r for maximal color paths will yield a test set of size at most $N^* r \log k$. Conversely, there are instances in which even if we could find repeatedly paths that cover the maximum number of colors, the resulting test set contains $N^* \log_e k$ test (where $\log_e$ denotes the natural logarithm).

Moreover, the negative results on the approximation of the Set Cover problem (28) imply that we cannot do better than a logarithmic factor in polynomial time. That is, for any polynomial time algorithm which constructs a complete test set of cardinality N, there are cases such that $N = \Omega(N^* \log k)$.

Paths with a constant bound on the number of colors covered

In spite of the negative results in the worst case, the longest path and greedy heuristic procedures were applied to real systems (26) and proved to be surprisingly efficient; a few tests cover a large number of colors and, afterwards, each test covers a *very small* number of colors. A typical situation is that the first 20% tests cover more than 70% of the colors. Afterwards, 80% of the tests cover the remaining 30% of the colors, and each test covers 1 to 3 colors. Consequently, the costly part of the test generation is the second part. Under these circumstances, exact procedures for either maximal color paths or minimal complete test sets are needed to reduce the number of tests as much as possible. The question is, can we obtain more efficient algorithms if we know that there is a bound on the maximum number of colors on any path that is a small constant $c << k$. We consider the following problems.

Suppose that a maximum color test covers no more than $c << k$ colors where c is a small constant. (1) Find a minimum complete test set; and (2) Find a maximum color test.

First, let us discuss Problem (1). We can find the different color sets of all the source-to-sink paths, in time that depends on the number of the color sets (instead of the potentially much larger number of paths) by a bottom-up processing of the DAG in reverse topological order. At each node we compute a family F_u of the color sets of the paths that start at u. At the source node we need to solve the Set Cover problem to find a subset of minimum cardinality that covers all the k colors. The complexity varies with the constant c. For $c = 1$, the problem is trivial: since a color set (path) contains at most one color, we can simply take k distinct color sets, which provides a minimum complete test set. On the other hand, at each node we can use a bit map to record the color sets and it takes time $O(k)$ to process each outgoing edge from a node. Therefore, the total time and space complexity is $O(km)$. For $c = 2$, Problem (1) can still be solved in polynomial time using graph matching techniques. For $c \geq 3$, the problem is NP-hard.

Problem (2) can be solved in time and space polynomial in the number of colors k and the size of the graph. The basic ideas are as follows. If all we want to do is to find a path that covers c colors (rather than all paths), then in the bottom-up computation we do not need to keep all the color sets but only a sufficient number of them. That is, at each node u, instead of the complete family F_u of color sets of the paths starting at u, we need keep only a subfamily L_u such that if the DAG contains a path through u that covers c colors, then there is such a path whose suffix from u to the sink t uses only colors from some member of L_u. That avoids keeping track all the subsets of colors; there are exponentially many of them. The detailed algorithm is more involved and we refer the readers to (26).

Conclusion

We have studied various techniques for conformance testing of software systems that can be modeled by finite state machines or their extensions. For finite state machines, we described several test generation methods based on status messages, reliable reset, distinguishing sequences, identifying sequences, characterization sets, transition tours and UIO sequences, and a randomized polynomial time algorithm. For extended finite state machines, the problem can be reduced to a graph path covering, and we presented several approaches to ensure the fault coverage and to reduce the number of tests.

While testing of software systems modeled by finite state machines is a well studied problem, testing of extended finite state machines is still at an early stage; the difficulties arise because of the state explosion due to the large number of combinations of variable values. Furthermore, software systems such as communication protocols usually contain timers and testing of the temporal properties is necessary. However, timers have an infinite range of values and their behaviors are difficult to test. Preliminary works have been done for the system reduction (4) (23), yet efficient test generation methods remain to be explored.

Bibliography

1. V. D. Agrawal and S. C. Seth, *Test Generation for VLSI Chips,* Computer Society Press, 1988.
2. 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 Trans. on Communication*, vol. 39, no. 11, pp. 1604-15, 1991.
3. A.V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
4. R. Alur, C. Courcoubetis, and D. Dill, ‘‘Model checking for real-timed systems,’’ in *Proc. 5th IEEE Symp. on Logic in Computer Science*, pp. 414-425, 1990.
5. International standard ISO 8802-2, ANSI/IEEE std 802.2, 1989.
6. W. Y. L. Chan, S. T. Vuong, and M. R. Ito, ‘‘An improved protocol test generation procedure based on UIOs,’’ *Proc. SIGCOM*, pp. 283-294, 1989.
7. S. T. Chanson and J. Zhu, ‘‘A unified approach to protocol test sequence generation’’, *Proc. INFOCOM*, pp. 106-14, 1993.
8. M.-S. Chen Y. Choi, and A. Kershenbaum, ‘‘Approaches utilizing segment overlap to minimize test sequences,’’ *Proc. IFIP WG6.1 10th Intl. Symp. on Protocol Specification, Testing, and Verification*, North-Holland, L. Logrippo, R. L. Probert, and H. Ural Ed. pp. 85-98, 1990.
9. T. S. Chow, ‘‘Testing software design modeled by finite-state machines,’’ *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, pp. 178-87, 1978.
10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw Hill, 1989.
11. J. Edmonds and E.L. Johnson, ‘‘Matching, Euler tours and the Chinese postman,’’ *Mathematical Programming*, vol. 5, pp. 88-124, 1973.
12. A. D. Friedman and P. R. Menon, *Fault Detection in Digital Circuits*, Prentice-Hall, 1971.
13. M. R. Garey and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
14. F. C. Hennie, ‘‘Fault detecting experiments for sequential circuits,’’ *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pp. 95-110, 1964.
15. G. J. Holzmann, *Design and Validation of Computer Protocols,* Prentice-Hall, 1991.
16. S. Huang, D. Lee, and M. Staskauskas, ‘‘Validation-based Test Sequence Generation for Networks of Extended Finite State Machines,’’ *Proc. FORTE/PSTV*, North Holland, R. Gotzhein Ed. 1996.
17. D. S. Johnson. ‘‘Approximation algorithms for combinatorial problems,’’ *J. of Computer and System Sciences*, vol. 9, pp. 256--278, 1974.
18. Z. Kohavi, *Switching and Finite Automata Theory,* 2nd Ed., McGraw-Hill, 1978.
19. L.-S. Koh and M. T. Liu, ‘‘Test path selection based on effective domains,’’ *Proc. of ICNP*, pp. 64-71, 1994.
20. M.-K. Kuan, ‘‘Graphic programming using odd or even points,’’ *Chinese Math.*, vol. 1, pp. 273-277, 1962.
21. R. P. Kurshan, *Computer-aided Verification of Coordinating Processes*, Princeton University Press, Princeton, New Jersey, 1995.
22. D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul, ‘‘Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach,’’ *IEEE Trans. on Communications*, vol. 44, no. 5, pp. 631-640, 1996.
23. D. Lee and M. Yannakakis, ‘‘On-line minimization of transition systems,’’ *Proc. 24th Ann. ACM Symp. on Theory of Computing*, pp. 264-274, 1992.
24. D. Lee and M. Yannakakis, ‘‘Testing finite state machines: state identification and verification,’’ *IEEE Trans. on Computers*, vol. 43, no. 3, pp. 306-320, 1994.

25. D. Lee and M. Yannakakis, ‘‘Principles and Methods of Testing Finite State Machines - a Survey,’’ *The Proceedings of IEEE*, Vol. 84, No. 8, pp. 1089-1123, August 1996.

26. D. Lee and M. Yannakakis, ‘‘Optimization Problems from Feature Testing of Communication Protocols’’, *The Proc. of ICNP*, pp. 66-75,1996.

27. L. Lovasz. ‘‘On the ratio of optimal integral and fractional covers,’’ *Discrete Mathematics*, vol. 13, pp. 383--390, 1975.

28. C. Lund and M. Yannakakis, ‘‘On the Hardness of Approximating Minimization Problems,’’ *J. ACM* 41(5), pp. 960-981, 1994.

29. R. E. Miller and S. Paul, ‘‘Generating conformance test sequences for combined control and data of communication protocols,’’ *IFIP Protocol Specification, Testing, and Verification*, XII, pp. 1-15, 1992.

30. R. E. Miller and S. Paul, ‘‘On the generation of minimal length test sequences for conformance testing of communication protocols,’’ *IEEE/ACM Trans. on Networking,*, Vol. 1, No. 1, pp. 116-129, 1993.

31. E. F. Moore, ‘‘Gedanken-experiments on sequential machines,’’ *Automata Studies*, Annals of Mathematics Studies, Princeton University Press, no. 34, pp. 129-153, 1956.

32. S. Naito and M. Tsunoyama, ‘‘Fault detection for sequential machines by transitions tours,’’ *Proc. IEEE Fault Tolerant Comput. Symp.*, IEEE Computer Society Press, pp. 238-43, 1981.

33. K. K. Sabnani and A. T. Dahbura, ‘‘A protocol test generation procedure,’’ *Computer Networks and ISDN Systems*, vol. 15, no. 4, pp. 285-97, 1988.

34. B. Sarikaya and G.v. Bochmann, ‘‘Synchronization and specification issues in protocol testing,’’ *IEEE Trans. on Commun.*, vol. COM-32, no. 4, pp. 389-395, 1984.

35. D. P. Sidhu and T.-K. Leung, ‘‘Formal methods for protocol testing: a detailed study,’’ *IEEE Trans. Soft. Eng.*, vol. 15, no. 4, pp. 413-26, April 1989.

36. M.U. Uyar and A.T. Dahbura, ‘‘Optimal test sequence generation for protocols: the Chinese postman algorithm applied to Q.931,’’ *Proc. IEEE Global Telecommunications Conference*, 1986.

37. M. P. Vasilevskii, ‘‘Failure diagnosis of automata,’’ *Kibernetika*, no. 4, pp. 98-108, 1973.

38. C. West, ‘‘Protocol validation by random state exploration,’’ *Proc. IFIP WG6.1 6th Intl. Symp. on Protocol Specification, Testing, and Verification*, North-Holland, B. Sarikaya and G. Bochmann, Ed. 1986.

39. B. Yang and H. Ural, ‘‘Protocol conformance test generation using multiple UIO sequences with overlapping,’’ *Proc. SIGCOM*, pp. 118-125, 1990.

40. M. Yannakakis and D. Lee, ‘‘An efficient algorithm for minimizing real-time transition systems,’’ *Proc. CAV*, pp. 210-224, 1993.

41. M. Yannakakis and D. Lee, ‘‘Testing finite state machines: fault detection,’’ *J. of Computer and System Sciences*, *Vol. 50, No. 2, pp. 209-227, 1995.*

Figure 1. Transition diagram of a finite state machine.

Figure 2. A Spanning tree of machine in Figure 1.