23

Modeling and Testing of Protocol Systems

David Lee
Bell Laboratories, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974
Tel: (908) 582-5872; *Fax:* (908) 582-5857
E-mail: lee@research.bell-labs.com

David Su
National Institute of Standards and Technology
Bldg 820, W. Diamond, Gaithersburg, MD 20899
Tel: (301) 975 6194; *Fax:* (301) 926-9675
E-mail: dsu@nist.gov

Abstract

Presented are conformance testing of the control portions of protocol systems, which can be modeled by finite state machines and checking sequences that can be used to verify the structural isomorphism of an implementation of the protocol system with its finite state machine-based specification. However, finite state machines are not powerful enough to model data portions associated with many real systems such as Personal HandyPhone Systems and 5*ESS*

Testing of Communicating Systems, M. Kim, S. Kang & K. Hong (Eds)
Published by Chapman & Hall

Intelligent Network Application Protocols. Extended finite state machines with variables and means to test them are also presented. Practical systems like ATM Traffic Management Protocols often contain parameters in the input/output cells; they increase the observability of the system but complicate their testing. Our model is further extended to communicating parameterized extended finite state machines for the test generation.

Keywords

Network protocol, finite state machine, extended finite state machine, conformance testing, checking sequence, complete test set, directed graph, covering path

1 INTRODUCTION

A finite state machine contains a finite number of states and produces outputs on state transitions after receiving inputs. Finite state machines have been widely used to model systems in diverse areas such as sequential circuits, some types of programs, and more recently, network protocols. This motivated early on research into the problem of testing finite state machines to discover aspects of their behavior and to ensure their correct functioning. 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 input/output (I/O) behavior, we want to test whether the implementation conforms to the specification. This is called *conformance testing* or *fault detection.* A test sequence that solves this problem is called a *checking sequence.*

There is an extensive literature on testing finite state machines, the fault detection problem in particular, dating back to the 50's. Moore's seminal 1956 paper on "gedanken-experiments" [Moore, 1956]. introduced the framework for testing problems. Among other fundamental problems, he 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 [Hennie, 1964] in 1964: he showed that if the 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 pa-

pers 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 [Kohavi, 1978]. 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 [Vasilevskii, 1973] 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 [Chow, 1978]. For machines without reset, a randomized polynomial time algorithm was reported in [Yannakakis and Lee, 1995]. Yet deterministic polynomial time algorithms remain open.

After introducing some basic concepts of finite state machine, we discuss various techniques for constructing checking sequences, using 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 control portions of protocols. However, practice systems often contain variables and their operations depend on variable values; finite state machines are not powerful enough to model in a succinct way such physical systems. In the second part of the paper, we use extended finite state machines, which are finite state machines extended with variables, to model systems, including Personal HandyPhone System (PHS) and Intelligent Network Application Protocols (INAP), and to generate tests.

Finally, we further extend the model to communicating parameterized extended finite state machines to discuss testing of ATM Traffic Management Protocols.

2 FINITE STATE MACHINES

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. $\delta : S \times I \rightarrow S$ is the state*

transition function; and $\lambda : S \times I \rightarrow O$ *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* $\lambda(s, a)$.

We denote the number of states, inputs, and outputs by $n = |S|$, $p = |I|$, and $q = |O|$, respectively. 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 Figure 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 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 $x = a_1, \cdots, a_k$ 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 $\lambda(s_1, x) = b_1, \cdots, b_k$, where $b_i = \lambda(s_i, a_i)$, $i = 1, \cdots, k$. Suppose that the machine in Figure 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. 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. We can construct in polynomial time a minimized machine and also obtain separating sequences for each pair of states [Kohavi, 1978]. A *separating family* of sequences for a machine of n states is a collection of n sets Z_i, $i = 1, \cdots, n$, 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., $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$; 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. Each Z_i has no more than $n - 1$ sequences and of length no more than $n - 1$ [Lee and Yannakakis, 1996a].

Given an FSM A of n states and separating families of sequences Z_i for each state s_i and an FSM B of 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 of no more than n states 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, and since B has at most n states, there is a one-to-one correspondence between similar states of A and B. Furthermore, two machines with the same input and output sets are *isomorphic* if they are identical except for a renaming of states. The ultimate goal of testing of systems modeled by finite state machines is to check if an implementation machine B is isomorphic to a specification machine A. Often we first check their similarity and then isomorphism.

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. 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 the test sequence is, it is possible that the test does not reach the faulty states or transitions in B, and this condition will not be detected. 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. With the above four assumptions, it is well known [Moore, 1956] that we only have to check 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, and such a sequence is called

a *homing sequence* [Kohavi, 1978]. 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. Furthermore, these sequences can be generated deterministically or randomly. The following subsections illustrate various test generation techniques.

2.1 Status messages and reset

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 [Naito and Tsunoyama, 1981; Uyar and Dahbura, 1986]. 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 Figure 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 $ssasbssasbssasbs$.

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 [Chow 1978; Chan, Vuong and Ito, 1989; Vasilevskii, 1973]. 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). 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

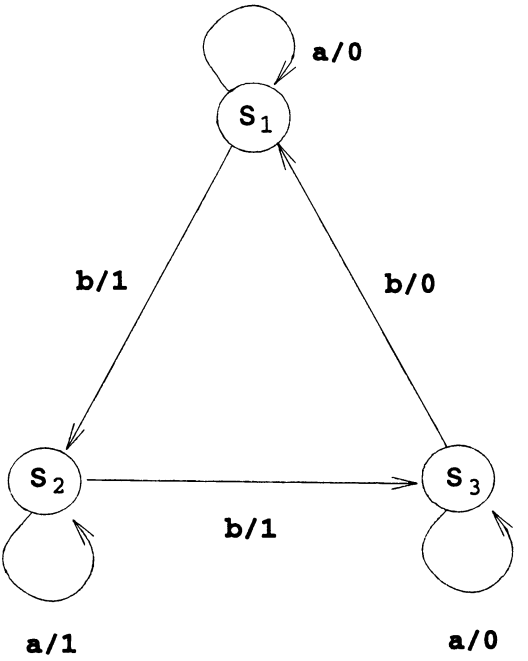

Fig. 1. Transition diagram of a finite state machine

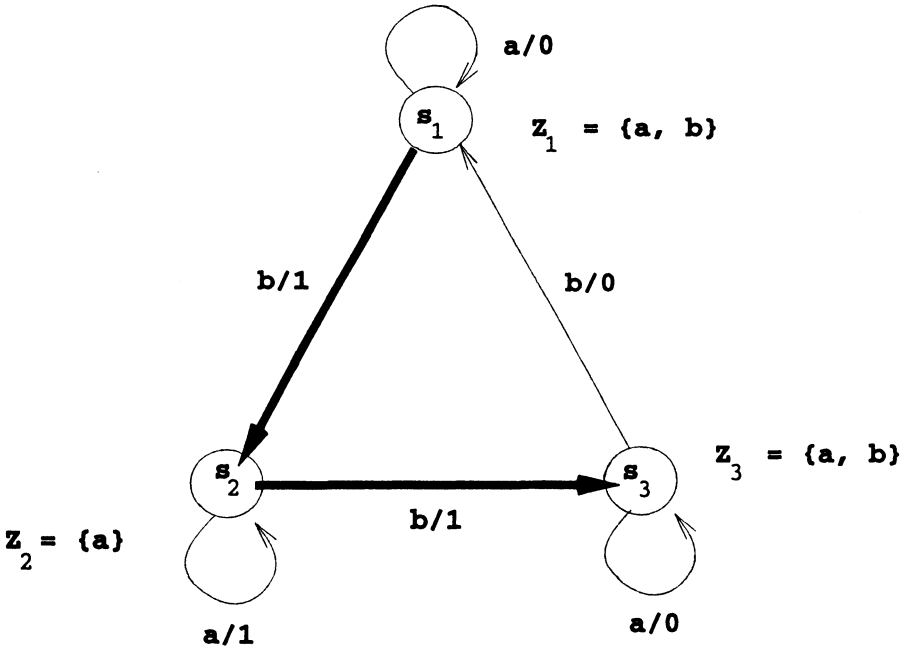

Fig. 2. A Spanning tree of machine in Fig. 1

of the tree. Finally, we check nontree transitions. 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 Figure 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 Figure 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 $\Omega(pn^3)$ [Vasilevskii, 1973]. For machines with unreliable reset, only randomized polynomial time algorithms are known [Yannakakis and Lee, 1995; Lee and Yannakakis, 1996a]; we can construct with high probability in randomized polynomial time a checking sequence of length $O(pn^3 + n^4 logn)$.

2.2 Distinguishing sequences

For machines with a *distinguishing* sequence there is a deterministic polynomial time algorithm to construct a checking sequence [Hennie, 1964; Kohavi, 1978] of polynomial length. 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 Figure 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 implementation machines 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 [Aho, Hopcroft and Ullman, 1974] 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:

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

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 sate 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 I/O pair 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:

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

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 Figure 1 is: $x_0 = ab$. The transfer sequences are, 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 a decision tree that specifies how to choose inputs adaptively based on observed outputs to identify the initial state. An adaptive distinguishing sequence has length $O(n^2)$, and, consequently, a checking sequence of length $O(pn^3)$ can be constructed in time $O(pn^3)$ [Lee and Yannakakis, 1994].

2.3 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 [Kohavi, 1978] (*locating sequences* in [Hennie, 1964]) that identify a state in the *middle* of the execution. Identifying sequences always exist and checking sequences can be derived from them [Hennie, 1964; Kohavi, 1978].

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 Figure 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.

2.4 A Polynomial time randomized algorithm

With status messages, reset, or distinguishing sequences, 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 [Yannakakis and Lee, 1995; Lee and Yannakakis, 1996a]. As is often used in theoretical computer science, "high probability" means that we can make the probability of error arbitrarily small by repeating the test enough times; specifically, the probability that it is not a checking sequence is squared if the length of the testing sequence is doubled. Note that 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.

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.

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 $k_i = O(nz_i \min(p, z_i)logn)$ for each $i = 1, \cdots, n$. 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.

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 [Yannakakis and Lee, 1995; Lee and Yannakakis, 1996a]. 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 $O(pn^3 + p'n^4logn)$ where $p' = \min(p, n)$.

2.5 Heuristic procedures and optimization

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. Without fault models, covering paths are often used in both circuit testing 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 [Aho, Dahbura, Lee and Uyar, 1991; Naito and Tsunoyamma, 1981; Uyar and Dahbura, 1986].

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* [Sabnani and Dahbura, 1988]. A UIO sequence for a state s_i is an input sequence x_i that distinguishes s_i from any other states, i.e., for any state $s_j \neq s_i$, $\lambda(s_i, x_i) \neq \lambda(s_j, x_i)$. 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 [Chan, Vuong and Ito, 1989]. We would like to find a shortest path that covers each red edge at least once. This is a *Rural Postman Tour* [Garey and Johnson, 1979], 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 [Aho, Dahbura, Lee and Uyar, 1991].

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 [Lee, Sabnani, Kristol and Paul, 1996; West 1986]. Basically, we only keep track of the current state and determine the next input on-line; 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 [Lee, Sabnani, Kristol and Paul, 1996] 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 [Sidhu and Leung, 1989; Yang and Ural, 1990].

3 EXTENDED FINITE STATE MACHINES

For testing of data portions of protocol systems finite state machines are not powerful enough to model in a succinct way the physical systems any more. Extended finite state machines, which are finite state machines extended with variables, have emerged from the design and testing of such systems. For instance, IEEE 802.2 LLC [ANSI, 1989] is specified by 14 control states, a number of variables, and a set of transitions (pp. 75-117). A typical transition is (p. 96):

current_state SETUP
input ACK_TIMER_EXPIRED
predicate S_FLAG = 1
outputCONNECT_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.

In our efforts in test generation for Personal HandyPhone Systems (PHS), a 5ESS based, ISDN wireless system [Lee and Yannakakis, 1996b] and for 5ESS Intelligent Network Application Protocols (INAP) [Huang, Lee and Staskauskas, 1996], we use the following model.

For a finite set of variables $\vec{x}$, a predicate on variable values $P(\vec{x})$ returns FALSE or TRUE. Given a function $A(\vec{x})$, an action is an assignment: $\vec{x} := A(\vec{x})$. Informally, an *extended finite state machine* (EFSM) has a finite set of states, inputs, outputs, and transitions between states, which are associated with inputs and outputs. In addition each transition is also associated with a predicate $P(\vec{x})$ and an action $A(\vec{x})$; the transition is executable if the predicate returns TRUE for the current variable values and in this case the variable values are updated by an assignment: $\vec{x} := A(\vec{x})$. Initially, the machine is in an initial state s_1 initial variable values: $\vec{x}_{init}$. Suppose that the machine is at state s with the current variable values $\vec{x}$ and that $< a, P(\vec{x})/o, A(\vec{x}) >$ is an outgoing transition from state s to q. Upon input a, if the predicate $P(\vec{x})$ returns TRUE, then the machine follows the transition, outputs o, changes the current variable values by action $\vec{x} := A(\vec{x})$, and moves to state q. 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 becoming an important topic, especially in the network protocol area [Favreau and Linn, 1986; Miller and Paul, 1993; Koh and Liu, 1994; Huang, Lee, and Staskauskas, 1996; Lee and Yannakakis, 1996b]. An EFSM usually has an initial state s_1 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 [Yannakakis and Lee, 1995; Lee and Yannakakis, 1996a] while for EFSM's hundreds of thousands of

states (configurations) are typical and it is in general impossible to construct a checking sequence. A commonly used heuristic procedure in practice is: each transition in the specification EFSM has to be executed at least once. A *complete test set* for an EFSM is a set of test sequences such that each transition is tested at least once.

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. We shall not digress to this topic. 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 also with the initial control state.

Formally, we have a directed graph G with n nodes, m 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. * 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 graph covering paths that cover all the edges. The path (test) length makes little difference and we are interested in minimizing the number of paths. We shrink each strongly connected component [Aho, Hopcroft and Ullman, 1974] into a node, which contains all the colors of the nodes and

*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.

edges in the component. The problem then is reduced to that on a directed acyclic graph (DAG) [Aho, Hopcroft and Ullman, 1974]. From now on, unless otherwise stated, we assume that the graph is a DAG.

We need a complete test set - a set of paths from the initial node to the sink node that cover all the colors C. 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, the problem is NP-hard. We need to restrict ourselves to approximation algorithms. Similar to the standard approximation algorithm for Set Cover [Garey and Johnson, 1979], 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.

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. Find a longest 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 on a DAG [Aho, Hopcroft and Ullman, 1974]. 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. It can be really bad in the worst case; the coverage ratio can be $\Omega(k)$ where k is the maximal number of uncovered colors on a path.

We now discuss a greedy heuristic procedure. It takes linear time and works well in practice. We topologically sort the nodes and compute a desired path from each node to the sink in a reverse topological order as follows. 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. 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 $\Omega(k)$.

We now describe briefly 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: $O(\sqrt{k})$.

In spite of the negative results in the worst case, the greedy heuristic procedures were applied to real systems [Huang, Lee and Staskauskas, 1996; Lee and Yannakakis, 1996b] 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 execution is the second part. To reduce the number of tests as much as possible exact procedures for either maximal color paths or minimal complete test sets are needed. 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 $c << k$. The problem can be solved in time and space polynomial in the number of colors k and the size of the graph. The detailed algorithm is more involved and we refer the readers to [Lee and Yannakakis, 1996b].

4 PARAMETERIZED EFSM'S

Finally, we consider testing of parameterized EFSM's. As a case study we discuss modeling and test generation of ATM Traffic Management protocol for the ABR (Allowed Bit Rate) services [ATM, 1996]. A formal specification is given in [Lee, Ramakrishnan, Moh and Shankar, 1996], using *Communicating Parameterized Extended Finite State Machines with Timers*. Suppose that two end stations send data and Resource Management (RM) cells to each other via a virtual circuit. Each end station consists of three communicating EFSM's, sending cells to each other. Cells contain *parameters* for traffic monitoring and

rate control. Furthermore, there are timers to determine the transmission of data and RM cells. The following is a typical transition:

current_state: S_2
next_state: S_1
event: $Y \geq Crm \& T > Trm$
actions:
$ACR := ACR * (1 - CDF)$;
$CCR_{FRM} := ACR$;
send an FRM cell;
$Y := Y + 1$; $X_1 = 0$;
$t := 0$; $T := 0$

Here CCR_{FRM} is a parameter in the Forward RM (FRM) cell to be sent, ACR, Y and X_1 are variables, T and t are timers, and Crm, Trm and CDF are system parameters, which are constants determined at the connection set up. When the current variable Y and timer T values satisfy the conditions in **event**, the following **actions** are taken and the system moves from state S_2 to S_1: the allowed cell rate ACR is reduced multiplicatively and then copied to CCR_{FRM} parameter in the FRM cell to be sent next, and the involved variable and timer values are updated.

Similar to testing of EFSM's we want to generate tests such that each transition is exercised at least once. Furthermore, we want to exercise the boundary values of the variables and parameters. The timers complicate the test generation; the timer expiration may take a long time and that makes the execution of some test sequences substantially more expensive than others. Furthermore, it takes significantly longer to make some events happen than others. For instance, a large number of data cells have to be sent to enforce the transmission of an RM cell. We need a different optimization criterion than that in the previous section; we want to minimize the test execution time rather than the number of tests. This can be formulated as follows. Each edge and node has a weight - the execution time. We want to generate tests such that each transition is executed at least once, and, furthermore, each boundary value of the variables and parameters is also exercised. On the other hand, we want to minimize the test execution time.

Similar to EFSM testing, we first generate a reachability graph of one end station and then assign to each transition a distinct color, which appears in the corresponding edges in the graph. Furthermore, each boundary value of a

variable and parameter is also assigned a distinct color, which appears in the corresponding nodes in the graph. We want to find a shortest tour (test) of the graph such that each color is covered at least once. It can be easily shown that the problem is NP-hard.

We use the following heuristic method. While deciding a tour on the graph from the current node we find a shortest path to a node, which is closest to the current node and contains an uncovered boundary value color. Using this technique, for the ABR protocol, 13 tests cover all the boundary values and transitions in the original specification [Lee, Su, Collica and Golmie, 1997].

The following is a sample test, which is a repeated execution of the sample transition in this section. It verifies that the Implementation Under Test (IUT) reduces its ACR by $ACR * CDF$ but not lower than MCR when the number of outstanding FRM cells is larger than CRM where UT is the Upper Tester and LT is the Lower Tester.

(1) Have UT send Mrm data cells to the IUT.

(2) LT waits for an FRM from the IUT.

The value of the CCR_{FRM} in the received FRM cell must satisfy:

$MCR \leq CCR \leq previous\_CCR * (1 - CDF)$.

(3) Set $previous\_CCR := CCR$.

(4) Repeat (1) to (3) until $CCR_{FRM} = MCR$ twice consecutively.

Obviously, the parameter CCR_{FRM} in the output cell FRM complicates the testing process. However, it adds to the observability of the system behavior; in this case we can read ACR variable values from this parameter.

5 CONCLUSION

We have studied various techniques for conformance testing of protocol 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 machine testing, it can be reduced to a graph path covering problem, and we present several approaches to ensure the fault coverage, to reduce the number of tests and to minimize the execution time.

We have discussed testing of *deterministic* machines. A different notion of

testing of *nondeterministic* machines is studied in several papers [Brinksma, 1988] and an elegant theory is developed. In this framework, the tester is allowed to be nondeterministic. A test case is an (in general nondeterministic) machine T, the implementation under test B is composed with the tester machine T, and the definition of B failing the test T is essentially that there exists a run of the composition of B and T that behaves differently than the composition of the specification A and T. It is shown in [Brinksma, 1988] that every specification can be tested in this sense, and there is a "canonical" tester. However, it is not clear how to use this machine T to choose test sequences to apply to an implementation. It has been shown [Alur, Courcoubetis, and Yannakakis, 1995] that testing of nondeterministic machines is in general a hard problem.

Acknowledgement. We are deeply indebted to the insightful and constructive comments from Jerry Linn.

6 REFERENCES

A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar (1991) 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.

A.V. Aho, J. E. Hopcroft, and J. D. Ullman (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

R. Alur, C. Courcoubetis, and M. Yannakakis (1995) Distinguishing tests for nondeterministic and probabilistic machines", *Proc. 27th Ann. ACM Symp. on Theory of Computing*, pp. 363-372.

ANSI (1989) International standard ISO 8802-2, ANSI/IEEE std 802.2.

ATM (1996) ATM Forum Traffic Management Specification Version 4.0, ATM Forum/96, af-tm-0056.000, April.

E. Brinksma (1988) A theory for the derivation of tests, *Proc. IFIP WG6.1 8th Intl. Symp. on Protocol Specification, Testing, and Verification*, North-Holland, S. Aggarwal and K. Sabnani Ed. pp. 63-74.

W. Y. L. Chan, S. T. Vuong, and M. R. Ito (1989) An improved protocol test generation procedure based on UIOs, *Proc. SIGCOM*, pp. 283-294.

S. T. Chanson and J. Zhu (1993) A unified approach to protocol test sequence generation, *Proc. INFOCOM,* pp. 106-14.

M.-S. Chen Y. Choi, and A. Kershenbaum (1990) Approaches utilizing seg-

ment 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.

T. S. Chow (1978) Testing software design modeled by finite-state machines," *IEEE Trans. on Software Engineering,* vol. SE-4, no. 3, pp. 178-87.

A. D. Friedman and P. R. Menon (1971) *Fault Detection in Digital Circuits,* Prentice-Hall.

J.-P. Favreau and R. J. Linn, Jr. (1986) Automatic generation of test scenario skeletons from protocol specifications written in Estelle, *Proc. PSTV,* pp. 191-202. North Holland, B. Sarikaya and G. v. Bochmann Ed.

M. R. Garey and D. S. Johnson (1979) *Computers and Intractability: a Guide to the Theory of NP-Completeness,* W. H. Freeman.

F. C. Hennie (1964) Fault detecting experiments for sequential circuits, *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design,* pp. 95-110.

G. J. Holzmann (1991) *Design and Validation of Computer Protocols,* Prentice-Hall.

J. E. Hopcroft and J. D. Ullman (1979) *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley.

S. Huang, D. Lee, and M. Staskauskas (1996) Validation-based Test Sequence Generation for Networks of Extended Finite State Machines, *Proc. FORTE/PSTV,* North Holland, R. Gotzhein Ed.

L.-S. Koh and M. T. Liu (1994) Test path selection based on effective domains, *Proc. of ICNP,* pp. 64-71.

Z. Kohavi (1978) *Switching and Finite Automata Theory,* 2nd Ed., McGraw-Hill.

M.-K. Kuan (1962) Graphic programming using odd or even points, *Chinese Math.*, vol. 1, pp. 273-277.

David Lee, K. K. Ramakrishnan, W. Melody Moh, and A. Udaya Shankar (1996) Protocol Specification Using Parameterized Communicating Extended Finite State Machines - A Case Study of the ATM ABR Rate Control Scheme, em Proc. of ICNP'96, October.

D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul (1996) 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.

D. Lee, D. Su, L. Collica and N. Golmie (1997) Conformance Test Suite for the ABR Rate Control Scheme in TM v4.0, *ATM Forum/97-0034,* February.

D. Lee and M. Yannakakis (1994) Testing finite state machines: state identification and verification, *IEEE Trans. on Computers,* Vol. 43, No. 3, pp.

306-320.

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

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

R. E. Miller and S. Paul (1993) 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.

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

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

K. K. Sabnani and A. T. Dahbura (1988) A protocol test generation procedure, *Computer Networks and ISDN Systems,* vol. 15, no. 4, pp. 285-97.

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

D. P. Sidhu and T.-K. Leung (1989) Formal methods for protocol testing: a detailed study, *IEEE Trans. Soft. Eng.*, vol. 15, no. 4, pp. 413-26, April.

M.U. Uyar and A.T. Dahbura (1986) Optimal test sequence generation for protocols: the Chinese postman algorithm applied to Q.931, *Proc. IEEE Global Telecommunications Conference.*

M. P. Vasilevskii (1973) Failure diagnosis of automata, *Kibernetika*, no. 4, pp. 98-108.

C. West (1986) 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.

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

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

7 BIOGRAPHIES

David Lee is a Distinguished Member of Technical Staff of Bell Laboratories Research and also an adjunct professor at Columbia University and a Visiting Faculty of National Institute of Standards and Technology. His current research interests are communication protocols, complexity theory, and image processing. David Lee is a Fellow of The IEEE, Editor of IEEE/ACM Transactions on Networking and and an associate editor of Journal of Complexity.

David H. Su is the manager of the High Speed Network Technologies group at the National Institute of Standards and Technology. His main research interests are in modeling, testing, and performance measurement of communications protocols. Prior to joining NIST in 1988, he was with GE Information Service Company as the manager of inter-networking software for support of customers' hosts on GEIS's data network. He received his Ph.D. degree in Computer Science from the Ohio State University in 1974.