PASSIVE TESTING AND APPLICATIONS TO NETWORK MANAGEMENT

David Lee Arun N. Netravali Krishan K. Sabnani Binay Sugla Ajita John

Bell Laboratories, Lucent Technologies

ABSTRACT

An important aspect of network management is fault management - determining, locating, isolating and correcting faults in the network. This paper deals with the algorithms for detecting faults, i.e., behavior of the network different from specifications. It is important for communication networks to detect faults "in-process" i.e., while the network is in its normal operation. Thus, we detect faults by examining the input-output behavior without forcing the system to specialized inputs explicitly for testing. Such testing is commonly called passive testing. We model the network as a finite state machine and develop procedures for passive testing including the required data structure, efficient implementations and the complexity of our procedures. We start with fully observable and deterministic machines and then study more realistic models: partially observable and nondeterministic machines. We also discuss extensions to communicating finite state machines and machines extended with parameters and variables. We apply our techniques to management of a signaling network operating under the Signaling System 7 (SS7) and report experimental results, which show the feasibility of applying passive testing to practical systems.

1. INTRODUCTION

Today's networks are becoming larger, more sophisticated, heterogeneous, and geographically dispersed. In addition, networks are put together by integrating equipment from multiple vendors. Consequently, management of such networks is becoming an important but more difficult task. Various things can go wrong, disabling the network or a portion of the network or degrading the performance to an unacceptable level. In many applications, where the end-to-end network performance needs to be guaranteed, many network elements need to be managed at the same time. A large network cannot be managed by human efforts alone. The complexity of such a network dictates the use of automated network management tools [1]. Fortunately, standards (for example, Simple Network Management Protocol - SNMP) have been proposed and are becoming popular for communicating status information using a protocol, a database structure specification and a set of data objects.

To maintain proper operation of a sophisticated network, the system as a whole and each individual component must be in proper working order. A fault is an abnormal condition or behavior that is different from how the system is either specified or expected to behave. Managing faults in a system is one of the key requirements for network management [2] [3]. It includes: (1) decide if a fault has occurred, (2) determine the location of the fault, (3) isolate the rest of the network so that it can continue to function, (4) reconfigure the network to minimize the impact of the fault, and (5) repair the fault.

We model communication networks by machines with a finite number of states and transitions between the states. A transition may be associated with certain input/output behavior, which may not be observable as in the case of Internal transitions or networks in which we can only observe packets but not interactions. Due to limited observability, a network often exhibits certain nondeterministic behavior, which is typically modeled by a nondeterministic finite state machine. Additionally, the interactions of different entities in a network can be better modeled by communicating finite state machines. Furthermore, since control variables and parameters are usually embedded in the network, we use extended finite state machines. All these concepts will be defined formally in later sections.

We have a *specification* machine A, which models the design or desired behavior of a network. We have an *implementation* machine B, which is the network under test and is a "black-box"; we can only observe its input/output behavior. We want to determine whether B has faults, i.e., is it different from A ?

The problem of fault detection has been studied extensively in the late sixties and early seventies motivated by testing of sequential circuits and more recently by testing of network protocols. A variety of methods have been proposed [4] [5] [6] [7]

0-8186-8061-X/97 $10.00 © 1997 IEEE

[8] [9] [10] [11] [12] [13] A test is designed based on the structure of the finite state machine that models the system. It is appropriate for testing an isolated machine of small to medium size. However, for network protocols modeled as communicating finite state machines, due to the interactions of component machines and variables, the size of the composite and/or expanded machine is formidable. This makes structured testing impractical. To cope with the complexity, unstructured testing has been proposed as well[14], [15]

Almost all the techniques for fault detection in the published literature involve *active testing;* a tester has complete control over the inputs and devises a test sequence to reveal possible faults of the system. Obviously, this approach is less applicable to network management; we have no control over the system inputs, and we can only *passively* observe the input/output behavior. Specifically, in operational networks it is frequently difficult to insert *arbitrary* inputs without affecting the service or the operation of the network. This leads naturally to *passive testing;* we observe the input/output behavior of a system in its normal operation for the purpose of detecting faults[16].

There are various ways in which passive testing can be used to manage networks. For example, Figure 1 shows a schematic diagram where managed sub-systems send relevant status information to a remote fault detection unit through local I/O filters. In Figure 2, using RMON standards[17], each managed device filters packets to feed inputs/outputs to a local fault detection unit.

Many network problems that occur due to intrusions and security violations can be addressed by the passive testing approach as well. This is clear from the observation that unwanted intrusions matter only if they are successful in changing the input/output behavior of the implemented machine. Thus, many security attacks may be treated as intentional and subtle modifications to the behavior of implemented machine that are manifested only under the presence of certain inputs and when the machine is in a certain state[18].

We have developed methods for passive fault detection of deterministic and nondeterministic finite state machines. Due to the nature of passive testing, our methods are readily extendible to communicating finite state machines with unobservable transitions and extended with variables. For clarity, we shall be mainly focused on finite state machine testing and then describe the extendibility to more general models. We report our algorithms, data structures, efficient implementations, and application to managing a real network system SS7.

In Section 2, we study passive testing of deterministic and fully-observable machines. In Section 3, we discuss a more realistic model of nondeterministic machines with unobservable transitions. In Section 4, we further extend our results to communicating and extended finite state machines. In Section 5, we report the results of applying our testing methods to a signaling network operating under the Signaling System 7 (SS7). We conclude our paper with a few possible future research directions in Section 6.

2. COMPLETELY OBSERVABLE AND DETERMINISTIC MACHINES

In this section, we assume that the network is modeled as a deterministic finite state machine. Although this assumption is not realistic, this exercise will help us in designing passive testing procedures for more complex models.

Finite state systems can usually be modeled by *Mealy* machines that produce outputs on their state transitions after receiving inputs. We start with deterministic finite state machines with fully observable transitions.

Definition 1 A deterministic finite state machine (DFSM) 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: $S \times I \rightarrow S$ is the state transition function;

λ: $S \times I \rightarrow O$ is the output function.

When the machine is in a 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 $\lambda(s, a)$.

□

For simplicity, we use FSM for DFSM unless otherwise stated. An FSM can be represented by [19] 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.

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 usual: 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 $y = \lambda(s_1, x) = b_1,...,b_k$, where $b_i = \lambda(s_i, a_i), i = 1, \cdots, k$.

2.1 Fault Detection

We are going to passively test a network B modeled as an FSM A. We assume that B has the same input/output symbols as A and that B is a ''black-box'' and we can only observe its input/output (I/O) behavior.

We want to passively observe the I/O behavior of B and determine whether it has faults that make it ''different'' from A. ''Different'' can have different implications; it can be structurally not isomorphic, inequivalent in bisimulation, or observationally inequivalent [4] [20]. Due to the nature of passive testing, which is based on an observation of I/O behavior of the implementation machine B, we only check for observational equivalence. If B has an observational behavior that is different from that of A, then B is faulty. Specifically, suppose that we have observed an I/O sequence from B x/y, i.e., $y = \lambda_B(s, x)$ from a certain (unknown) state s. If there is no state s' in A with $y = \lambda_A(s', x)$, then B is not equivalent to A and, therefore, B is faulty. On the other hand, we have to figure out from which state we have started checking and, consequently, the passive testing process contains the following two steps.

(1) Home onto the state of network B. Suppose that machine A has n states. If machine B is equivalent to A, then initially it can be in any of the n states. After we start passive testing, we have to precisely find out the current state of B. For doing this, we use the I/O observations to keep on narrowing down the states B can be in. Essentially, this is a *passive homing sequence.*[19] [21]. We call this procedure "passive homing." When the set of states B can be in is down to one state, then we can start checking if B has a fault. It is also possible that the set of states B can become empty after a passive homing process; in this case the machine is faulty and we can terminate the test.

(2) After homing, keep comparing the observed behavior with the expected behavior. While we are tracing the machine unambiguously, if there is a discrepancy between the expected output from the specification machine A and that from the observed output, we have detected faults. We only detect fault(s), not locate faults. The next subsection has details of this procedure.

2.2 An Algorithm

We keep a linked list L of current possible states. The algorithm contains two parts: a homing process followed by a fault detection process. Recall that the next state and output functions of machine A are denoted by δ and λ, respectively. Let n_k be the number of possible current states after observing k I/O pairs with $n_0 = n$.

Algorithm 1. (Fault detection of deterministic machines)

input. A specification FSM A and an I/O sequence from an implementation FSM B: $a_1/o_1 \cdots a_k/o_k \cdots$;
output. Whether B is not observationally equivalent to A.

```
begin
    L = { s_1, ..., s_n };  /* possible current states */
    for (k = 1 to ∞) do /* check each I/O pair */
        for (i = 1 to n_{k-1}) do /* check each state in L */
            if (o_k ≡ λ(s_i, a_k))
                replace s_i by δ(s_i, a_k) in L;
            else
                remove s_i;
        end
        remove redundant states in L;
        if (L ≡ ∅)
            return FAULT; /* B is faulty */
        else if (|L| ≡ 1)
            goto fault_detection;  /* homing sequence found */
        else
            ;
    end
/* tracing unambiguously - fault detection */
fault_detection:
    let L = { s };
```

```
        r = 1;
        while (o_{k+r} ≡ λ(s, a_{k+r})) do {
            s = δ(s, a_{k+r});
            r++; /* no faults detected yet; check
            next I/O pair */
            }
        end
        return FAULT;
end
```

After detecting faults, we want to locate and correct them. We shall not digress here. See for instance [22].

2.3 Efficient Implementations and Complexity

In the homing process, after observing an I/O pair, we examine each state in L, find the outgoing transition with the observed input, and check the expected output. If there is a discrepancy, then the item (state) is removed from the list, and the counter of possible current states is decreased by 1. If the expected output is the same as the observed one then the item is replaced by the next state. When the list L becomes empty there is a discrepancy between the expected and the observed outputs no matter what the initial state of the implementation machine was and we declare faults. If the list L becomes a singleton set then we have observed a homing sequence and we move on to the fault detection process. Otherwise, we keep tracing and updating L. In the fault detection process either we can determine a unique next state or there is a discrepancy in the expected and observed outputs. In the former case we keep tracing and in the latter case we declare faults.

The cost of checking redundant states in L is also proportional to n_k using a bit vector, for instance. Processing of each I/O pair a_k/o_k takes a constant time for each state under consideration. There are n_k states on the list L when we process the $(k+1)$st I/O pair a_{k+1}/o_{k+1}. Suppose that we detect faults at the Nth I/O pair from observing a discrepancy between the specification machine A and the implementation machine B. The total processing time is then $O(\sum_{k=0}^{N-1} n_k)$. Note that n_k can only decrease with increasing k and after having observed a homing sequence n_k stays at 1 until a fault is detected.

Theorem 1. It takes time $O(\sum_{k=0}^{N-1} n_k)$ for a passive test sequence to detect faults of an implementation machine where N is the total number of observed I/O's up to a discrepancy and n_k is the number of possible current states after the observation of the kth I/O pair.

□

2.4 Missing Observations

Suppose that during our passive testing, we missed the observation of k pairs of I/O's and we want to resume passive testing afterwards. A naive approach is to start from scratch, i.e., from all possible n states, and to use the previously described algorithm. However, this is unnecessary. We describe two procedures next, based on what we know about how much we have missed; either we know the exact number of I/O pairs missed or a bound on the number of I/O pairs missed. For simplicity, we first discuss the case when there is only one I/O pair missed and then the general scenario of missing an arbitrary number of I/O pairs.

During the observation, we have missed one I/O pair. We update the active states list L as follows. From the current states in L, we find all the possible next states from those in L and replace all the states in L. Meanwhile we eliminate the redundant states. The total processing cost is proportional to pn_k where p is the number of input symbols.

We now consider the general case where we have missed k I/O pairs in a row at step i with n_i states in L. We repeat the above procedure k times. Suppose that the resulting list L has n_{i+j} states, $j = 1, \cdots, k$. The total cost is proportional to $p\sum_{j=1}^{k} n_{i+j}$. The worst case cost is $O(pkn)$. As a matter of fact, we can stop the processing when any of the $n_{i+j} = n$; we have no information as to where we are and we have to start from scratch.

As k increases, the uncertainty grows exponentially. If k is very large, then we should just restart passive homing from scratch. Otherwise, we use the following procedure described here to narrow search for passive homing. There are two cases.

At Most k I/O Pairs Missed

First assume that we know we have missed no more than k I/O pairs. From all the recorded possible states, we want to determine all the possible current states. This can be done by a

modified breadth-first-search (BFS) as follows.

We first add an auxiliary source state s', which has an edge leading to all the previously recorded states. We then conduct a BFS for $k+1$ steps from s'. The set of all the visited states are the possible current states. We need certain modifications to the data structures for BFS to implement this algorithm. Informally, we have to keep track of the distance (level) from all the frontier nodes to the source s'. The total cost is $O(pn)$ where p and n are the number of inputs and states, respectively.

Exactly k I/O pairs Missed

Now assume that we have missed exactly k pairs of I/O's and that we want to know what are the possible current states. It turns out that the more we know the more we pay to keep track of them. However, in this case, the possible number of current states may be less than the previous case.

We need a further modification of the BFS. Similarly, we create a new source node s' and edges from the source to all the previously recorded states. We then conduct a BFS from s' using a queue to keep track of the frontier nodes. *We have to explore and expand along cross and back edges,* which is different than conventional BFS. We perform the search $k+1$ steps; the frontier nodes are the possible current states. The total cost is $O(kpn)$ where p and n are the number of inputs and states, respectively. Note that the time complexity is not exponential and not even multiplicative; it is additive in terms of steps k.

3. UNOBSERVABLE TRANSITIONS AND NONDETERMINISTIC MACHINES

In many networks some subset of inputs and outputs may be unobservable. This results in unobservable transitions, which have not been considered in Section 2. We model them by a τ move [23]. Taking the transitive closure [24] with respect to the τ moves, we obtain a nondeterministic machine *without* τ moves, which is equivalent to the original machine[23]. The problem is then reduced to passive testing of nondeterministic machines:

Definition 2 A Nondeterministic finite state machine (NFSM) 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.

$\delta \subseteq S \times I \times S$ is the state transition relation, and

λ is the output function mapping from δ to 2^O, the power set of the output set O.

Furthermore, for each state s in S and input a in I there exists a state t in S such that the transition (s, a, t) is in δ. □

Different from DFSM's, upon the same input, an NFSM can move to more than one state and produce outputs accordingly. The output function value is determined by the current state, input, and next state. Suppose that $(s, a, t) \in \delta$, i.e., upon input a, the machine moves from the current state s to a next state t. There can be more than one possible output associated with the transition and we denote them by $\lambda(s, a, t)$, which is a subset of the output set O. The last property in Definition 2 is not essential; it makes the machine completely specified and consistent with Definition 1.

An NFSM 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 outputs associated with the transition. From a state, there can be more than one outgoing edge labeled with the same input symbol and going to the same state or different states.

3.1 Fault Detection Algorithm

Similar to DFSM's in Section 2, initially, the implementation machine can be in any of the n states. While observing its I/O behavior, we keep track of a list of all its possible current states. If this list becomes empty then we have detected faults. However, different from the case of DFSM, the total number of possible current states may not be monotonically decreasing; from a single state an input may take the machine to more than one next state.

The data structures are similar to that for Algorithm 1. The only difference is that when processing an I/O pair from a current state, there may be more than one possible next state. Specifically, at a current state s_i and an I/O pair a_k/o_k, denote the set of all possible next states by $\Delta(s_i, a_k, o_k) = \{ t \in S\colon (s_i, a_k, t) \in \delta$ and $o_k \in \lambda(s_i, a_k, t) \}$. We have

Algorithm 2. (Fault detection of nondeterministic machines)

input. A specification NFSM A, and an I/O sequence from an implementation NFSM B: $a_1/o_1 \cdots a_k/o_k \cdots$;

output. Whether B is not observationally equivalent to A.

begin

$L = \{ s_1, \cdots, s_n \}$; /* possible current states */

for (k = 1 to ∞) **do** /* check each I/O pair */

for (i = 1 to n_{k-1}) **do** { /* check each state in L */

delete s_i from L;

for each $s' \in \Delta(s_i, a_k, o_k)$ **do**

insert s' in L;

end

remove redundant states in L;

if ($L \equiv \emptyset$)

return FAULT; /* B is faulty */

}

end

end

end

□

Similar to DFSM, we can consider missing observations. The approaches are essentially the same and we omit the details here.

3.2 Efficient Implementations and Complexity

Note that in Algorithm 1 there is a homing procedure, since the machine is deterministic and the number of states in L can only decrease as k increases. However, Algorithm 2 deals with nondeterministic machines and there are no homing procedures in general, since the number of states in L may increase with k.

The data structures and implementations for Algorithm 2 are similar to those of Algorithm 1. The main difference is that at a current state, upon an input, we have a set of possible next states, and, furthermore, for each possible transition, we have a set of possible outputs. We omit the details and discuss the cost next.

The total number of the outgoing transitions and their corresponding outputs examined is determined by the current state and observed I/O pair. Suppose that at the kth I/O pair, the total number of transitions and outputs processed is m_k, then

Theorem 2. It takes time $O(\sum_{k=1}^{N} m_k)$ for a passive test sequence to detect faults of an implementation machine where N is the total number of observed I/O's up to an observed discrepancy and m_k is the total number of possible transitions and outputs processed after the observation of the kth I/O pair.

4. EXTENSIONS

In principle, FSM appropriately models *control portion* of communication protocols. However, in practice the usual specifications of protocols include variables and operations based on variable values; ordinary FSM's are not powerful enough to model in a succinct way the physical systems any more. *Extended Finite State Machines* (EFSM), which are finite state machines extended with variables, are used to model the protocol data portions as well. In an EFSM, associated with each transition, there is a predicate on the current variable values and an action (assignment) of the variable values. A transition is executed if the associated predicate is TRUE for the current variable values, which are then updated by the associated actions during the execution.

Early results of (active) test generation for EFSM's have been reported. For instance, see [25] [26] [27] [21]. The difficulties are from the predicates and actions associated with transitions; a path in the control portion of the corresponding FSM may not be feasible due to the predicates associated with the transition. As a matter of fact, even the reachability problem is undecidable if the variable values are infinite and PSPACE-complete otherwise.

However, for passive testing of control portions of network protocols, we can ignore the variables, predicates, and actions in the specification machines; this is one of the major difference between active and passive testing. The reason is: the I/O sequences we observe are feasible since they are from a physical system, the implementation machine, whereas in active testing we have to satisfy the constraints from the predicates and actions on current variable values when we generate test sequences. If we observe a discrepancy (in the control portions) of the specification and the implementation machines then definitely we have detected faults. As in the case of testing for FSM's, no discrepancy does not necessarily imply that the implementation machine conforms to the specification machine. This issue will be further addressed in Section 6.

In practice, network entities interact, for example, by message exchanges through communications channels. Such systems can be modeled by *communicating finite state machines* (CFSM), which is a collection of finite state machines interacting with each other [28].

If we can passively observe each component machine then the problem is reduced to that of FSM's. However, this is not always the case in practice; we can only observe part of the I/O activities from an output port of the *composite machine* [28]. We could construct the composite machine and apply the passive testing procedure of Algorithm 1 and 2. However, initially we may be in any of the global states of the composite machine, and they are often impossible to construct; this is the well-known state explosion problem. Practical constraints are to be further explored to reduce the number of possible initial states. A *reset* to a unique initial state or a *partial reset* to a small set of possible initial states might be a possibility.

5. APPLICATIONS TO MANAGING SIGNALING SYSTEM SS7

In this section we discuss applications of the passive testing approach to fault detection in a signaling network operating under the Signaling System 7 (SS7) protocol mechanisms. The system is chosen to illustrate the applicability of the passive testing approach to a real problem. We report preliminary experimental results.

5.1 SS7 Background

The CCITT SS7 signaling set of protocol standards enable the switches, databases and other intelligent nodes (called the Signaling End Points or SEPs) of a modern telecommunication network to exchange messages related to call processing, queries and management information [29]. A typical signaling network is configured as shown in Figure 3. Each switch is connected to a signaling region that consists of two mated routers (called the Signaling Transfer Points or STPs) for fault redundancy. A switch is usually connected to both the STPs of the region via access links (A -link). The two STPs that form a region are interconnected by cross links (C-links). There may be many regions in the signaling network. All the routers in one region may be connected to all other routers in another region through links that are called bridge links (B-links). Figure 3 describes the nomenclature used to denote the various components of the SS7 based signaling network.

The routing algorithm in a SS7 network attempts to distribute the packet traffic uniformly between every set of links between two communicating nodes. It is designed to maintain packet sequencing and achieve low latency. Since the message traffic on the signaling network is critical to the completion of calls, the SS7 network places a very strong emphasis on high reliability and availability. Thus arbitrary failures must be managed to provide continued end to end delivery of packets.

In response to this need, a set of *automatic* network management controls have been put in place to enable reroute of packet traffic under *arbitrary* failure conditions. The logic of these controls is complex because of the asymmetric nature of the *links* in the signaling network. For example, in order to provide overload protection to the C-link, one of the heuristics recommends that the use of the C-link be limited to situations where other situations are more undesirable. Given a number of similar constraints, the network management controls operate as follows. Each router in this network, the STP, maintains a status of its neighboring nodes and links. Based on this knowledge it attempts to find a route for the packets it receives, maintaining packet sequencing and keeping end to end routing delays within specified limits.

5.2 SS7 Route Management Controls

The messages that are exchanged between the routers and the SEPs of the signaling network to reroute packet traffic constitute the automatic route management control section of SS7 standard. The typical messages that are exchanged between adjacent nodes are of the form that recommend to the receiving node if a particular route may be used (Transfer Allowed, TFA), may not be used (Transfer Prohibited, TFP) or *should* not be used (Transfer Restricted, TFR).

Since large networks will typically have some failures in a given period of time, the correct operation of these controls is imperative. Indeed, some of the recent spectacular failures in operations of telecommunication networks have their roots in incorrect specification or erroneous implementation of these network management controls. In the examples given below, we address the problem of detecting faults in operations of

these network management controls where the implementation is inconsistent with the specification. Another point to note is that it is easier to observe these control messages as they are distinguishable from the general packet traffic and typically require far less bandwidth than the actual data traffic.

Figure 4 describes a small state machine for illustrative purposes. For a given STP there are two routes to a given switch. The direct route is through the A-link connecting the STP to the switch. The indirect route may be via the C-link to its mate STP and then the A-link to the switch. The two routes from a particular STP to a switch are then governed by the failures of the A-links and the C-link.

5.3 Experiments

Extensive experiments were conducted on passive fault detection of SS7 using our techniques. The average sequence length to detect any of the 1266 faults over 1000 randomly generated I/O sequences turned out to be between 700 and 4000. The average time a sequence of length 1000 took on a single processor of a Sparc 1000 was 0.2 seconds. About 0.82% of the faults generated in our experiments were found to be undetectable using the procedure described in Section 3.2.

The results show that passive testing is feasible for the SS7 network. In all our experiments, more than 90% of the faults were detected within 7000 I/O pairs and less than 4% of the faults were undetectable within 10000 pairs. Due to the space limit we refer the interested readers to [30]. for details.

6. CONCLUSION

The concept of passive testing was first described in [16] without any proposed methods for solving the problem. The idea seemed to be forgotten due to the lack of applications. We believe that the techniques we have developed can be useful not only for network management but also for other application areas when we have no control over the system behavior and yet we are to analyze the system.

We only discussed fault detection; we conclude that there are faults in the implementation machine (the network in our case) but we are not able to locate the faults for the purpose of correcting them. While active fault location has been studied; heuristic procedures and polynomial time algorithms for single fault location have been obtained [31] [32] [22] passive fault locating remains to be explored.

The above process only detects faults in case of an observation of a discrepancy in terms of the output between a specification machine and an implementation machine; they are observationally inequivalent. However, if there is no discrepancy so far, how can we determine whether they are observationally equivalent or not? A general question is as follows. Suppose that upon an input sequence x both the specification and implementation machines produce the same output sequence y. Can we conclude that the two machines are structurally isomorphic or observationally equivalent? An input sequence that provides an answer is called a *checking sequence*. It has been shown that there are always checking sequences of polynomial length for DFSM's [12]. However, it is only for active testing. For passive testing, a possible approach is to construct a minimized machine that produces the observed I/O sequence x/y. If the machine is the same as the specification machine than we are sure that the implementation machine conforms. Otherwise, it is faulty. However, to construct the minimized machine is in general an NP-hard problem [19]. As a matter of fact, it can be shown that to determine whether a given input sequence is a checking experiment is NP-hard [33].

We have studied passive conformance testing of an implementation machine to a specification machine. In practice, often we do not know the specification and we want to *identify* an implementation machine; we want to identify the structure of a ''black-box'' from passively observing its I/O behavior. Even an active system identification problem is known to be inherently exponential [4]. The passive system identification problem is quite a challenge, especially, when a system has nondeterministic behavior.

ACKNOWLEDGEMENTS

We are indebted to Ray E. Miller and Yow-Jian Lin for the insightful comments and stimulating discussions. Constructive comments from the anonymous reviewers of ICNP'97 are deeply appreciated.

1. D. M. Tow, "Network Management - Recent Advances and future trends," *IEEE Journal on Selected Area in Communications*, vol. 6, pp. 732-741, May 1988.

2. P. Travis, "Why the AT&T Network Crashed," *Telephony*, vol. 218, no.4, pp 11, Jan 22, 1990.

3. E. C. Rosen, "Vulnerabilities of network control protocols: An example," *ACM SIGSOFT Software Engineering Notes*, Vol. 6, no. 1, pp 6-8, Jan 1981.

4. E.F. Moore, ''Gedanken-experiments on sequential machines,'' *Automata Studies, Annals of Mathematical Studies,* No. 34, Princeton Univ. Press, Princeton, NJ, pp. 129-153, 1956.

5. F.C. Hennie, ''Fault-detecting Experiments for Sequential Circuits,'' *Proc. 5th Ann. Symp. on Switching Circuit Theory and Logical Design,* pp. 95-110, 1964.

6. K.K. Sabnani and A.T. Dahbura, ''A Protocol Testing Procedure,'' *Computer Networks,* Vol. 15, No. 4, pp. 285-97, 1988.

7. D. Sidhu and T. Leung, ''Fault Coverage of Protocol Test Methods,'' *Proc. IEEE INFOCOM '88,* pp. 80-85, 1988.

8. 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.

9. S.C. Boyd and H. Ural, ''On the Complexity of Generating Optimal Test Sequences,'' *IEEE Transactions on Software Engineering,* Vol. 17, No. 9, pp. 976-978, 1991.

10. 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.

11. R. E. Miller and S. Paul, ''Structural analysis of protocol specifications and generation of maximal fault coverage conformance test sequences,'' *IEEE/ACM Trans. on Networking,*, Vol. 2, No. 5, pp. 457-470, 1994.

12. 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.

13. D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - a Survey," *Proceedings of The IEEE,* Vol. 84, No. 8, pp. 1090-1123, August 1996.

14. C. West, ''Protocol Validation by Random State Exploration,'' *Protocol Specification, Testing, and Verification, VI,* (eds.) B. Sarikaya and G.v. Bochmann, North-Holland, 1986.

15. 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.

16. C. L. Seitz, ''An approach to designing checking experiments based on a dynamic model,'' in *Theory of Machines and Computations,* Z. Kohavi and A. Paz Ed. Academic Press, 1972, pp. 341-9.

17. S. Waldbusser, "Remote Network Monitoring Management Information Base", *RFC*, 1271, November 1991.

18. C. Wang and Mischa Schwartz, "Fault Detection with Multiple Observers", *IEEE/ACM Transactions on Networking,* Vol. 1, No. 1, Feb 1993.

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

20. R. Milner, *Communication and Concurrency,* New York: Prentice Hall, 1989.

21. D. Lee and M. Yannakakis, ''Optimization Problems from Feature Testing of Communication Protocols,'' *Proc. of ICNP,* pp. 66-75, October, 1996.

22. David Lee and Krishan Sabnani, "Reverse-Engineering of Communication Protocols," *Proc. of ICNP*, pp. 208-216, October, 1993.

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

24. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Reading, Mass.: Addison-Wesley,

1974.

25. H. Ural and B. Yang, "A test sequence selection method for protocol testing," *IEEE Trans on Communications,* vol. 39, no. 4, pp. 514-523, 1991.

26. 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.

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

28. D. Brand and P. Zafiropulo, "On communicating finite-state machines," *JACM*, vol. 30, no. 2, pp. 323-42, April 1983.

29. Abdi R. Modarressi and Ronald A. Skoog, "Signaling System No. 7: A Tutorial," *IEEE Communications,* July 1990, pp. 19-35.

30. David Lee, Arun N. Netravali, Krishan K. Sabnani, Binay Sugla and Ajita John, "PASSIVE TESTING AND APPLICATIONS TO NETWORK MANAGEMENT" *Tech. Memo. Bell Laboratories,* 1997.

31. A. Ghedamsi and G. v. Bochmann, "Test result analysis and diagnostics for finite state machines," in *Proc. 12th Int. Conf. on Distributed Systems,* 1992.

32. A. Ghedamsi, G. v. Bochmann, and R. Dssouli, "Multiple fault diagnostics for finite state machines," in *Proc. INFOCOM 93*, pp. 782-91, 1993.

33. M. Yannakakis and D. Lee, "Test Selection and Passive Checking Sequences," paper in preparation.

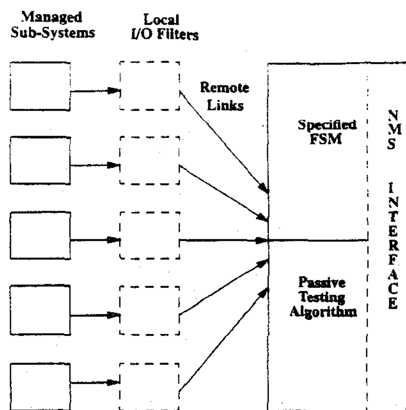

Figure 1. Use of Passive Testing Algorithm for Systems

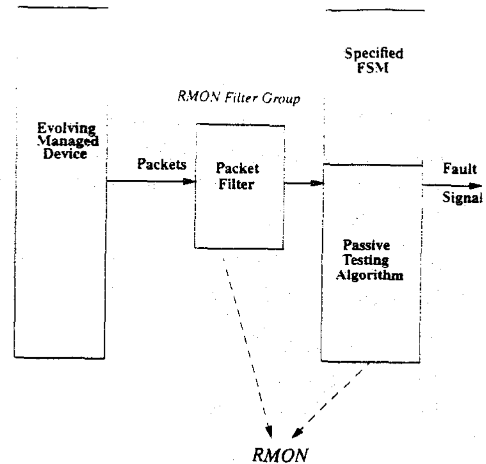

Figure 2. Use of Passive Testing Algorithm for Managing Simple Devices

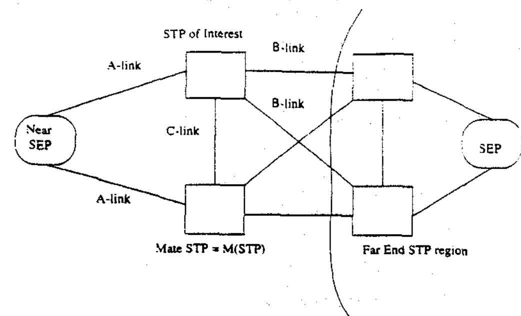

Figure 3: Structure of an SS7 Network

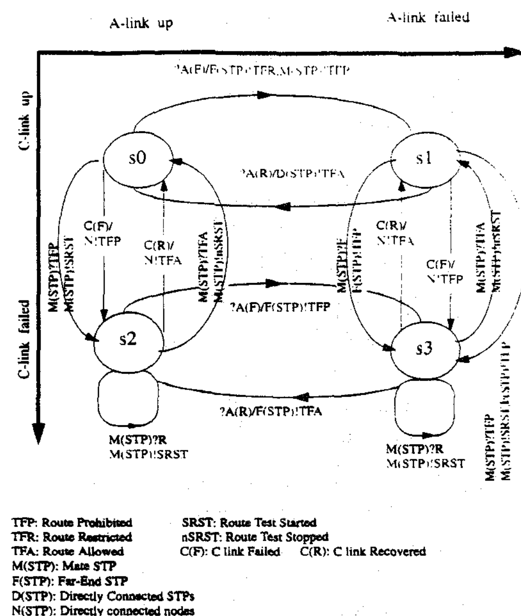

Figure 4: Example of a Partial FSM in STP of Interest