On Fault Identification in Networks by Passive Testing

Raymond E. Miller[1]
Department of Computer Science
University of Maryland,
College Park, MD 20742
miller@cs.umd.edu

Khaled A. Arisha
Department of Computer Science
University of Maryland,
College Park, MD 20742
arisha@cs.umd.edu

Abstract

In this paper, we employ the finite state machine (FSM) model for networks to investigate fault identification using passive testing. First, we introduce the concept of passive testing. Then, we introduce the FSM model with necessary assumptions and justification. We introduce the fault model and the fault detection algorithm using passive testing. Extending this result, we develop the theorems and algorithms for fault identification. An example is given illustrating our approach. Then, extensions to our approach are introduced to achieve better fault identification. We then illustrate our technique through a simulation of a practical X.25 example. Finally future extensions and potential trends are discussed.

1 Introduction

Due to the rapid growth in Telecommunication networks and the fast evolution in technology, the need for a more efficient and effective network management approach becomes more urgent. The International Standard Organization (ISO) has defined network management for the Open System Interconnection (OSI) seven-layer model in terms of five functional areas: fault management, configuration management, accounting management, performance management, and security management [7]. A considerable effort has been made to standardize network management protocols and develop network management systems, such as the Simple Network Management Protocol (SNMP) and the Common Management Information Protocol (CMIP) [6]. However, there is much to be done towards formally specifying problems in network management and developing formal techniques to solve these problems. Our work models the network using the formal approaches of Finite State Machine (FSM) as is done in [1], however we study the fault identification problem rather than simply the fault detection.

The work presented here focuses on one critical functional area of network management; namely fault management. There are two approaches to test a network for fault management: active testing and passive testing. The most commonly used approach for fault management is active testing, which gathers information actively. By "actively" we mean injecting test messages into the network to aid in finding network faults. Active testing has techniques in common with conformance testing of protocols. Conformance testing is used to test protocols off-line to insure that a protocol implementation conforms to its specification. Test sequences are generated from the specification. These input sequences are applied to the implementation to see whether the produced output sequence matches the expected one given by the specification. In contrast, fault management for networks takes place while the network is in use. Because of this, it is desired to keep testing traffic overhead to a minimum. Passive testing simply observes the normal traffic of the network, without adding any test messages. Thus using passive testing enables examining the input-output behavior without forcing the network to any test input sequences. As will be discussed here, quite a bit of fault management can be accomplished using passive testing.

Fault management usually covers the following aspects: detection, location, identification, coverage and correction. The main objective of this research is to see how much fault management information we can obtain using passive testing only. The simplest approaches to passive testing use a FSM specification to model the behavior of the network. Given an implementation of the network under test, it is viewed as a black box where only the input-output behavior is observable. The problem is to determine whether the behavior of the implementation conforms to the behavior of the specification. If it does not conform, this implies the existence of a fault.

Lee et al [1] apply passive testing on a FSM model of the network for fault detection. Their paper demonstrates effective fault detection capabilities of passive testing

[1] Some of this work was done while consulting for Bell Laboratories, Lucent Technologies.

based on observation of the input/output sequence of the implementation. However, due to the limitation of the single FSM model, no fault isolation or fault location was possible. In [2][3] Miller presents a variant of the Communicating Finite State Machines (CFSM) model to specify a network. Using this model he showed that some fault location information could be deduced. As noted above, using passive testing to detect faults eliminates testing overhead normally encountered by other methods that inject special test messages into a network. Of course fault detection is not sufficient. Once a fault is detected, other remedial steps are required to eliminate the fault. Fault location helps by isolating the corrective actions to only a portion of the network. Thus, additional fault location capability by passive testing would be very useful if faults could be isolated to ever-smaller regions. Additionally, if the exact fault that occurred could be limited to a small set of possibilities, this would further simplify the corrective activities. It is precisely this greater fault location and identification capability by passive testing that is being considered here. Miller and Arisha [4][5] demonstrated that better fault location is possible. Here we consider fault identification. Referring to the previous experience of using the FSM model [1] and extending the results to our CFSM model [2][3][4][5], fault identification is based on FSM model in this paper.

We describe the FSM based specification model in section 2 of this paper, with necessary assumptions and justifications. In section 3, we introduce the fault model and give a brief description of the fault detection algorithm using passive testing. Section 4 describes the theorems and algorithms developed for fault identification. It also illustrates the algorithms through an example. In section 5, extensions to our approach are given to give further fault identification. An experiment modeling a practical network is presented in section 6 along with simulation results to demonstrate the effectiveness of our approach. Finally, conclusions and possible extensions are discussed in section 7.

2 The Specification Model

In this section we introduce the FSM model for network specification. The model is defined in [1] where the system is modeled as one Finite State Machine (FSM). In this paper we restrict our attention to deterministic machines.

2.1 The FSM model

A deterministic finite state machine (DFSM) M is a six-tuple: $M = (I, O, S, s_0, \delta, \lambda)$ where:

- I, O, and S are finite non-empty sets of input symbols, output symbols, and states respectively.
- s_0 is a designated initial state.
- 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 state s in S and receives an input a in 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. Also the definition for the transition function δ and the output function λ can be extended from input symbols to strings as well. Starting from initial state s_0, an input sequence $x = a_0,a_1,\ldots,a_k$ takes the machine successively to states $s_{i+1} = \delta(s_i, a_i)$, $i=0,1,\ldots,k$, with the final state $\delta(s_0, x) = s_{k+1}$, and produces an output sequence $y=\lambda(s_0, x)=b_0,\ldots,b_k$, where $b_i=\lambda(s_i,a_i)$, $i=0,1,\ldots,k$.

2.2 Assumptions and Justifications for The FSM Model

A) **Single Fault**. We assume that if a fault occurs, only one fault occurs during a test cycle. This assumption is important since multiple faults can cause complications such as the hiding of faults. These complications make fault detection by passive testing either more complicated or even impossible. The single fault assumption can be justified however, since we are not assuming a system has frequent faults. If this is so, an effective approach would be to have the one fault in the network detected and corrected before the second fault occurs.

B) **Complete Machines**. If we define for each state all the possible combinations of inputs on incoming channels, this may lead to a combinatorial explosion. Instead, we show only transitions that would actually take place during correctly specified operation. For all those not specified, a fault should be detected. So all unspecified transitions will lead to an implicitly defined additional fault state with a new output called "*f*" to indicate "fault." This fault state is not an "original state" in the specification; it is used only to allow us to assume that the machines are completely specified. However, in the actual passive testing process, if this faulty state is reached, this would imply that the implementation machine could not produce the observed input/output sequence if it is a correct implementation of the specification.

C) **Deterministic Machines**. Non-determinism sometimes comes from the lack of complete information during the specification phase. We are assuming here that all "necessary" information is available to indicate deterministically the behavior of the machine. Sometimes non-determinism is introduced in the specification machine to allow

different options to be chosen during implementation. We are assuming specific options are chosen in our FSM specification insuring that it is a DFSM.

3 The Fault Model

In this section we introduce the fault model together with a broad overview of the fault detection procedure. The fault model describes the assumption about fault types expected in the network. The fault detection procedure discussed in this section briefly covers the algorithm mentioned in [1].

3.1 Fault Types

Due to our assumptions of the FSM model used in passive testing, the two types of faults that we can investigate, in terms of the FSM specification, are:

1) **Output Fault**: This occurs when a transition has the same head and tail states and the same input as in the specification FSM, but the output is altered.
2) **Tail State Fault**: This occurs when a transition has the same head state and input/output symbols as specified, but the tail state is altered.

3.2 Fault Detection Overview

As described in [1], the fault detection capability of passive testing can be summarized as follows:

We compare the observed input/output sequence of the implementation machine *B* with the expected behavior of the specification machine *A*. *B* is considered "faulty" if its behavior is *different* than that of *A*. That is, there is no state in *A* that would display the input/output sequence observed from *B*. Here we consider only Observational equivalence. We do not consider any structural isomorphism or equivalence by bi-simulation since all we can do is to observe the input/output sequence from *B* and we do not know the structure of *B*.

We define the observer as the entity possessing passive testing fault management functionality. The fault detection procedure can be described as follows:

a) Since we do not know the state of *B* when the observation starts. We assume that *B* could be in any state representing a state of *A*. Let L^0 designate this initial set of possible states.
b) Once the observer observes the first input (i_1) together with the corresponding output (o_1), transitions in *A* lead from states of L^0 to another set of possible states L^1.
c) As each input/output pair (i_j/o_j) is observed, a new set L^j is produced similarly from the set L^{j-1}. The sequence $L^0, L^1, \ldots, L^{j-1}, L^j$ is formed such that, with every observation (i_j/o_j), only the subset of states from L^{j-1} that can still accept an input i_j and produce an output o_j will survive, and they are going to be replaced by their successor states after applying the transition (i_j/o_j). The $\{L^i\}$ sequence is monotonically non-increasing in size, due to the assumption that the machine is deterministic.
d) As the *i/o* sequence progresses, more observed input/outputs will join the observation sequence $e=i_1/o_1, i_2/o_2, \ldots i_{j-1}/o_{j-1}, i_j/o_j$. Consequently, the sequence of sets of possible states $L=L^0, L^1,\ldots, L^{j-1}, L^j$ is obtained
e) At some point of observing the input/output sequence, the set of possible current states can lead to a singleton or to an empty set. In case of a singleton set L^k, all subsequent L^j, where $j>k$, will be a singleton sets, or become empty. This process of obtaining a singleton set is called the *passive homing sequence.* And from that point on the current state is determined exactly.
f) As long as L^k is not an empty set, the observation is not contradicting the expected behavior of the specification machine. In this case passive testing concludes that there is no fault yet observed. The implementation machine behavior conforms to that of the specification up to that point.
g) If set L^k becomes empty, this indicates a fault in the implementation. There is no state in *A* that could produce the observed *i/o* sequence *e*. Assuming that the implementation *B* is deterministic and complete, it will end up in some state and be able to continue, but no such state continuation is specified in *A*.

An example of a FSM model and the passive testing fault detection algorithm is shown in Figure 1, where *x* is the observed input/output sequence.

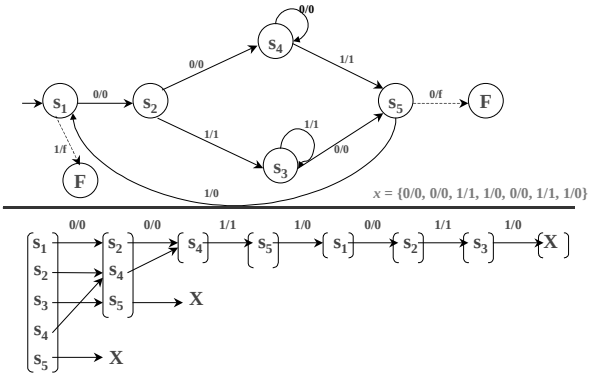

Figure 1. An example FSM for the specification

The detailed algorithm that describes the above procedure is in [1] together with extensions to the algorithm to recover from missing at most *k* observations. Other extensions dealing with unobservable transitions and non-deterministic machine specifications are also provided in [1].

4 Fault Identification

This section covers the theorems and algorithms for the fault identification process. It also gives an illustrative example demonstrating the proposed technique.

4.1 The Backward Trace

In section 3.2 we discussed how we obtained the sequence of sets L^0, L^1, …, L^{j-1}, L^j from the observed input/output sequence. Now, let us assume we have an observed input/output sequence i_1/o_1, i_2/o_2,…i_{k-1}/o_{k-1}, i_k/o_k and the resulting sequence of sets L^0, L^1, …, L^{k-1}, L^k where $L^k=\phi$ and $L^{k-1}\neq\phi$. That is, at step k we have just detected that a fault has occurred. We will call this process "forward trace" since it can be computed step-by-step as each input/output pair is observed. Now, for fault identification purposes we analyze this input/output sequence, in terms of the specification, by another process that we call the "backward trace", to produce a second sequence of sets of states.

1) We let $(L^k)^R$ be the set of all states of A.
2) In a backward manner we form set $(L^{j-1})^R$ from $(L^j)^R$ as follows: $(L^{j-1})^R$ contains all states that are head states of transitions with input/output i_j/o_j with tail states being members of $(L^j)^R$.

The following relationship between the sequences of forward and backward trace sets exists.

Theorem 1: The corresponding backward and forward sets are disjoint, i.e. $L^j \cap (L^j)^R = \phi$.

Proof: Assume the forward set L^j and the backward set $(L^j)^R$ are not disjoint, then we have at least one common state between the forward trace and the backward trace. So, we can follow –using the "faulty" input/output sequence- a valid path from some state in L^0 down to L^j and then using the common state between it and the $(L^j)^R$, we can continue in the backward trace from $(L^j)^R$ to $(L^k)^R$. And this contradicts with the criteria of fault detection in passive testing, i.e. $L^k = \varphi$.

Corollary 1: The backward set $(L^j)^R$ is empty for some the $j \geq 0$.

This is an immediate result of the previous theorem, since L^0 contains all states of the FSM and since $L^0 \cap (L^0)^R=\phi$, $(L^0)^R$ will contain no state.

4.2 Fault Identification

Output Fault identification:

Theorem 2: If L^j has a state s_p that under i_{j+1} has an output $\neq o_{j+1}$ and $(L^{j+1})^R$ has $\delta(s_p, i_{j+1})$ as an element, then the output fault $s_p \text{—}(i_{j+1}/o_{j+1})\rightarrow \delta(s_p, i_{j+1})$ could have occurred.

Proof: By definition of the output fault only the output symbol is altered. For a transition $s_p \text{—}(\alpha/\beta)\rightarrow s_q$ to be an output fault, there should be a transition in the specification FSM $s_p \text{—}(\alpha/\gamma)\rightarrow s_q$, where $\gamma \neq \beta$. So, s_q should be $\delta(s_p, i_{j+1})$, assuming that $\delta(s_p, i_{j+1}) \neq F$ (the Faulty state), i.e. s_p has such a transition in the specification going to some valid state. The specification should also satisfy the condition $o_{j+1} \neq \lambda(s_p, i_{j+1})$. Thus, we can consider the output fault $s_p \text{—}(i_{j+1}/o_{j+1})\rightarrow \delta(s_p, i_{j+1})$ to occur if the state s_p is in L^j, $\delta(s_p, i_{j+1})$ is in $(L^{j+1})^R$, $o_{j+1}\neq \lambda(s_j, i_{j+1})$ and the observed input/output tuple is (i_{j+1}/o_{j+1}).

Tail State Fault Identification:

Theorem 3: If L^j has a state s_p with transition $s_p \text{—}(i_{j+1}/o_{j+1})\rightarrow s_q$ and there is a s_r in $(L^{j+1})^R$, then $s_p \text{—}(i_{j+1}/o_{j+1})\rightarrow s_r$ is a tail state fault that could have occurred.

Proof: By definition the tail state fault alters only the tail state of the transition from the specification. Thus for a transition $s_p \text{—}(\alpha/\beta)\rightarrow s_q$ in the specification to have a tail state fault, there should be a transition in the implementation FSM $s_p \text{—}(\alpha/\beta)\rightarrow s_r$, where $s_q \neq s_r$. Now, for any $s_r \in (L^{j+1})^R$, changing the specified transition $s_p \text{—}(\alpha/\beta)\rightarrow s_q$ to $s_p \text{—}(\alpha/\beta)\rightarrow s_r$ is a tail state fault which creates a state sequence for the observed *i/o* sequence. Thus this tail state fault could have occurred.

We now describe our "forward-backward-crossover" algorithm that identifies some of the faults that could have caused the observed input/output sequence to be produced. To introduce this we first look at an example.

4.3 Example

Using the same FSM specification as shown in figure 1, we assume an observed input/output sequence: {*0/0, 1/1, 1/0, 0/0, 1/0*}. The forward and backward traces are shown in figure 2 along with "crossovers" shown by dotted arrows.

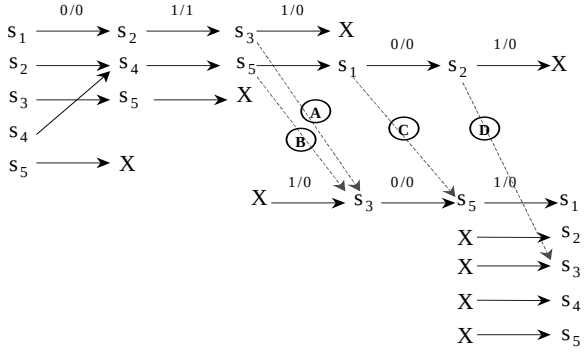

Figure 2. Example of the forward and backward traces

The four "crossovers" arrows are applications of theorems 2 and 3 as described below:

A) Applying theorem 2 we see that this crossover depicts an output fault of transition s_3 —(1/1)→ s_3 changing to s_3 —(1/0)→ s_3.

B) Applying theorem 3 we see that this crossover depicts a tail-state fault of transition s_5 —(1/0)→ s_1 changing to s_5 —(1/0)→ s_3.

C) Applying theorem 3 we see that this crossover depicts a tail-state fault of transition s_1 —(0/0)→ s_2 changing to s_1 —(0/0)→ s_5.

D) Applying theorem 2 we see that this crossover depicts an output fault of transition s_2 —(1/1)→ s_3 changing to s_2 —(1/0)→ s_3.

This example should provide insight over how single faults that could possibly occur, and would cause the implementation to produce that observed input/output sequence, can be found using the forward and backward traces along with crossovers. The algorithm is thus:

Forward/Backward Crossover Algorithm

1. Do the forward trace analysis for the observed input/output sequence, letting k be the least k such that $L^k = \phi$.
2. Do the backward trace analysis for the observed input/output sequence. Note: This can only be done after the complete input/output sequence has occurred.
3. Add crossover arrows by applying theorems 2 and 3.

Output faults (theorem 2) can arise in this analysis from states, that under some observed input/output have no next state (i.e. an X) in the forward trace analysis. In our example we found two such cases where the current tail state for the transition appeared in the backward analysis at the next step in the input/output sequence. On the other hand, tail state faults (Theorem 3) can arise from states in the forward trace analysis that have next states in the forward trace, but whose faulty next states appear in the next step of the backward analysis.

Although we can identify some of the possible faults (both output and tail-state) that could have caused the observed input/output sequence to be produced by an implementation, this procedure does not identify all such output or tail-state faults. The basic problem is that the forward/backward crossover analysis only finds those faults for which the faulty transition was traversed only once (i.e. that was shown by the crossover) during the observed input/output sequence. It can leave unidentified those possible faults in which a faulty transition is traversed more than once during the observed input/output sequence. We demonstrate this problem for the same machine as in figure 1 but with a different input/output sequence.

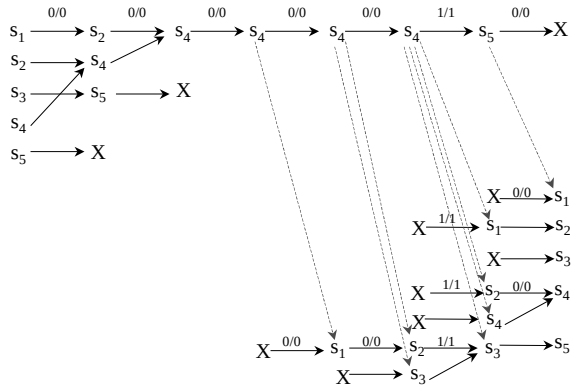

Figure 3: Example of the Recurrent Fault

The above figure shows standard forward/backward crossover analysis for the input/output sequence {*0/0, 0/0, 0/0, 0/0, 0/0, 1/1, 0/0*}. In this case seven tail-state faults and one output fault are identified by the analysis. Yet, if we consider the fault s_2 —(0/0)→ s_4 to s_2 —(0/0)→ s_1 then the forward analysis, assuming this fault, would give

s_1 —(0/0)→ s_2 —(0/0)→ s_1 —(0/0)→ s_2 —(0/0)→ s_1 —(0/0)→ s_2 —(1/1)→ s_3 —(0/0)→ s_5

s_2 —(0/0)→ s_1 —(0/0)→ s_2 —(0/0)→ s_1 —(0/0)→ s_2 —(0/0)→ s_1 —(1/1)→ X

s_3 —(0/0)→ s_5 —(0/0)→ X

s_4 —(0/0)→ s_4 —(0/0)→ s_4 —(0/0)→ s_4 —(0/0)→ s_4 —(0/0)→ s_4 —(1/1)→ s_5 —(0/0)→ X

s_5 —(0/0)→ X

Thus, as seen by this analysis, if initially in state s_1, the observed input/output sequence would indeed be produced. Note, however, that this s_2 —(0/0)→ s_1 has been traversed multiple times in the sequence and this fault was thereby missed from being identified in the forward/backward crossover procedure. As it turns out two other "recurrent faults" are also possible for this input/output sequence. They are s_2 —(0/0)→ s_4 to s_2 —(0/0)→ s_2 and s_3 —(0/0)→ s_5 to s_3 —(0/0)→ s_3. The reader may verify this. The question then arises as to how to find (or identify) these "recurrent faults". This is the topic of the next section.

5 Recurrent Faults

This section covers extensions to our approach to improve fault identification. As discussed at the end of the last section, recurrent faults can arise and may go unidentified using our standard forward/backward crossover algorithm. Some extensions to our approach need to be devised to improve its fault identification capability. First, we need to characterize recurrent faults. Then, we describe our extension to our standard fault identification approach to enable the identification of

recurrent faults. Finally, we present some correction and complexity analysis for this extension.

For the time being, we assume that the recurrent fault occurs after the passive homing. Later, this restriction will be relaxed to generalize the algorithm. From the previous discussion in section 4, a recurrent fault may occur if it occurs more than once before being detected ($L^k=\phi$). It can be easily proven that an output fault cannot be a recurrent fault since the occurrence of an output fault is followed by an "X" –i.e. impossible next state, so we cannot proceed to have such a fault occur more than once. Thus, recurrent faults are only tail-state faults.

We now return to the tail state fault s_2 —$(0/0)$→ s_4 to s_2 —$(0/0)$→ s_1 that we discussed in section 4.3. We notice that before the passive testing detects this fault, the fault occurred again in the implementation before ($L^k=\phi$). This repetition of fault occurrence causes the fault to be "hidden". The forward/backward crossover analysis does not identify this fault. Other potential faults have been identified, but not this recurrent fault. Next, an explanation is given about why the forward/backward crossover analysis cannot identify recurrent faults, in general, by developing some recurrent fault characterizations.

To characterize the recurrent fault in an abstract way, the following figure shows the general recurrent fault case:

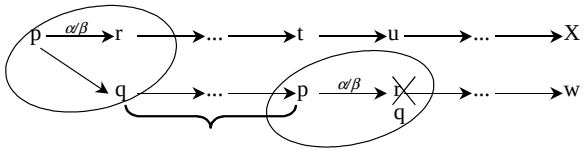

Figure 4: Recurrent Fault Characterization

As shown in Figure 4, we have the tail-state fault p —(α/β)→ r to p —(α/β)→ q. This fault reoccurs before it is detected ($L^k=\phi$). The upper trace shows the forward trace produced by passive testing from the specification, while the lower trace shows how the actual trace of states is produced by the faulty implementation. Notice that before the passive testing detects that there is a fault ($L^k=\phi$), the tail-state fault occurs again as in the lower trace. The forward/backward crossover analysis cannot identify such a fault, because the backward analysis – starting from $(L^k)^R$ containing all possible states including *w*- can trace backward using the observed input/output sequence up to the last state *q*. From that state *q*, the backward analysis cannot trace it back to state *p* using the observed input/output (α/β). Using the forward/backward crossover technique it will rather identify the tail-state fault t—(α/β)→ u to t—(α/β)→ q if the upper forward trace actually continues to a state *t*. And since the backward trace will not be able to reach the first occurrence of the recurrent fault, it may not be able to identify the recurrent fault.

This fault case characterized above is what we define as a "recurrent fault". Generally, recurrent faults do not occur that frequently. Our approach to identify recurrent faults is to pre-compute the set of all the potential tail-state faults off-line, using the FSM specification model only, without using any input/output sequence. Then we delete any faults that can be identified by the forward/backward crossover analysis to obtain a much smaller set of potential recurrent faults. Then, we use this smaller set of faults on-line to verify whether any of these faults can produce the observed input/output sequence as a recurrent fault. Although the set is potentially very large, some significant pruning can be achieved if we take into consideration the nature of the FSM model and interleave the analysis of both traces.

To calculate these faults off-line, for the specification FSM we exhaustively check every possible tail-state fault to see if it can produce the above characterization using every possible input/output sequence. Using some Graph's Depth First Search (DFS) based procedure, the set of recurrent faults can be calculated by trying all paths of input/output sequences after the fault splits the sequence into a specification sequence and a faulty implementation sequence. The main idea is to try to find another occurrence of the recurrent fault (last) starting from some (first) occurrence of that fault. The search for matching between such first and last occurrences of the recurrent fault can lead to an exponentially growing tree of possibilities. Some special cases which can help prune the search tree considerably, are as follows:

Immediate recurrence: This means that the next state in the faulty implementation is the same as the head-state, as shown below.

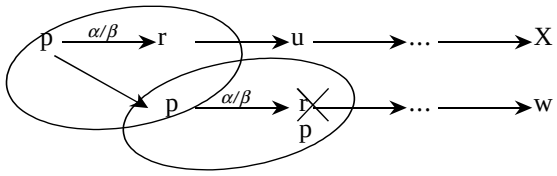

Figure 5: Immediate Recurrent Fault Case

In this case, we have the tail-state fault under test as p —(α/β)→ r to p —(α/β)→ p. The recurrent fault can obviously occurs again since the recurrent fault next state matches its head state (self-loop tail-state fault). The algorithm needs to test whether it can distinguish between the specification and the faulty implementation, i.e. whether there exists a state $u = \delta(r, \alpha/\beta)$ or there is an "X" state. For such a fault to be recurrent there should be such a state *u*, otherwise the standard forward/backward crossover algorithm can easily identify this recurrent fault.

Merged traces: This means that the two sequences of the specification and the faulty implementation merge into one state before detecting the fault ($L^k=\phi$). This indicates that if this fault is a recurrent fault, a later traversal of this fault will be able to be used to check for recurrence. Thus, we prune the tree by cutting off all paths due to this occurrence of the fault.

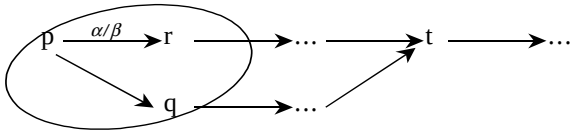

Figure 6: Merged Traces Case

Repeated patterns: This means that the same pattern of corresponding states has been revisited in the specification and the faulty implementation sequence. In this case we prune the tree at this second occurrence, since all possible situations will be checked by paths from the first occurrence.

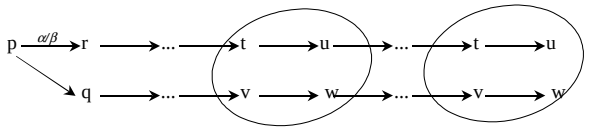

Figure 7: Repeated Trace Patterns Case

The algorithm to compute these recurrent faults can be described as follows:

∀ possible tail state fault for the specification FSM
(p—(α/β)→r to be *p—(α/β)→q)*
***visit**(r, q);*
end ∀
***Procedure visit**(spec_state, impl_state)*
∀ possible input tuple α_i in the implementation FSM
if (impl_state == p)
*if(∃ **no** δ(spec_state, α_i)*
return "Recurrent fault"
else return "Can be distinguished"
endif
*if(∃ **no** δ(spec_state, α_i) loop back and try next α_i "Distinguished"*
if(next_impl_state = next_spec_state) return "Merged"
if(repeated_pattern of impl_states, spec_states) return"Repeated Pattern"
***visit**(next_spec_state, next_impl_state)*
endproc

Correctness:

Termination: since the procedure visit is DFS-like and since the FSM has a finite number of states (*n*) and transitions, the recursive call cannot exceed *n* in any path it follows.

Coverage: The procedure guarantees complete checking since it tries out every possible tail state fault.

According to our characterization and definition of the "Recurrent" fault, the procedure successfully identifies these faults.

Complexity: The algorithm checks every tail state fault:
Total number of tail state faults is *(n-1)*|transitions per state| = O(n*I)*

For each tail state, procedure visit() will exhaustively test every possible input existing in the transitions of the implementation state *|transitions|=O(I)*, and this recursion worst case depth is as long as the number states (largest cycle in the FSM sequence w/o revisiting similar pattern, merging traces, or distinguishing between paths) = *O(n)*. Thus, the worst case recursion depth of visit() = $O(I^n)$.

Total complexity of the algorithm in worst case is $O(n*I*I^n) = O(I^{n+1})$.

Although the worst case complexity of the algorithm is exponential, there are still attractive features of the algorithm that justify using it:

1) It is done as pre-computation once in a lifetime for the specification FSM.
2) The number of recurrent faults is very small *O(1)*, especially with practical network protocols where a bigger input/output alphabet is used which decreases ambiguity and easily distinguishes different states.
3) It will simplify checking them in the on-line passive testing as explained below.

For recurrent faults that may occur before passive homing, we need to generalize the off-line pre-computation to be able to identify such. The algorithm is similar to the one described above except generalizing to consider different starting initial states for both specification and implementation. Other than that, the generalization is straightforward.

Although this approach looks like an exhaustive search, it is still attractive since it is done only once and hardly affects the efficiency of the on-line fault identification.

Example Going back to the 5-state example shown in Figure 1, applying the algorithm for pre-computation of recurrent faults, we get the following possible recurrent faults:

Table 1: Results of recurrent fault algorithm

Original transition	Faulty transition
s_1—(0/0)→s_2	s_1—(0/0)→s_1
s_2—(0/0)→s_4	s_2—(0/0)→s_1
s_2—(0/0)→s_4	s_2—(0/0)→s_2
s_2—(1/1)→s_3	s_2—(1/1)→s_2
s_3—(0/0)→s_5	s_3—(0/0)→s_3
s_4—(1/1)→s_5	s_4—(1/1)→s_4

On-line Checking of Recurrent faults During the forward processing (on-line) of the observed i/o sequence, another parallel procedure will be executed for each faulty implementation FSM with each of the recurrent faults.

At any point if the sequence of states fails to continue, the procedure for that recurrent fault is terminated.

If one of these sequences succeed to reach step *k*, with its $L^k \neq \phi$, we include this recurrent fault in the set of identified faults (set of Tail-State faults that may occur).

Some significant improvement of the performance of the on-line checking can be achieved especially when the observed input/output sequence is long. First, We can check the input/output sequence for repeated input/output pairs. Since recurrent faults can never happen if the associated input/output pair is not repeated in the input/output sequence. So we can eliminate checking any recurrent fault whose input/output pair does not appear more than once in the input/output sequence.

Another improvement can be achieved to shorten the on-line checking. We can search the forward and backward traces for the first and the last occurrence of the recurrent fault. In the forward trace we search for head-state and the input/output pair of the recurrent fault, while in the backward trace we search for the faulty next state and the input/output pair of the recurrent fault. IF there is no such "match", the fault cannot be a recurrent fault and it can be eliminated from consideration. If there is a match, the recurrent fault needs to be checked as described above.

6 Experiment and Practical Example

In this section we describe the generation and simulation for our experiment. Then, we apply our simulation to a practical example, X.25 network layer protocol.

6.1 Generation and Simulation

To evaluate the performance of the passive fault identification techniques we have just discussed, a simulator was constructed to determine how well the fault identification performed using a random injection of faults.

Working with a practical example; the X.25 communication protocol, we focus on the ISO-8208 based network packet layer protocol. We implemented in full detail the call establishment, call termination and a more abstract version of the data transfer. We show the resulting FSM model in Figure 10.

This diagram is a FSM as defined in section 2. Here, however, the symbols on each transition express the idea of receiving that symbol as input from an external entity and sending the same symbol to another external entity. For example, the symbol "CR" means receiving a "Call Request" input symbol from the upper layer protocol user entity, and it means also forwarding the same "Call Request" output symbol to the peer network layer protocol entity which it is communicating with. We simply left off the double marking, like CR/CR, for simplicity.

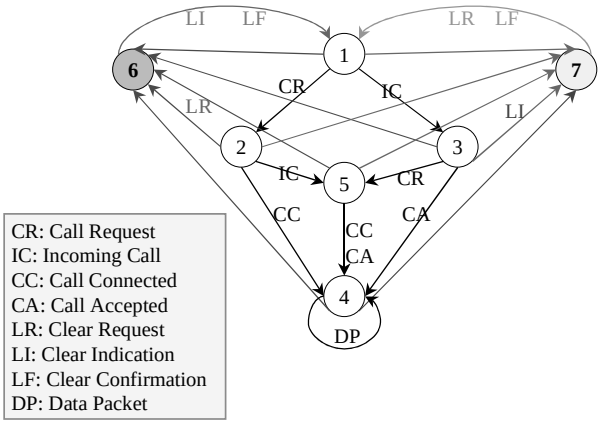

Figure 10. Model for the practical Experiment

First of all, we run our pre-computation algorithm to identify the recurrent tail state faults. Out of the 49 potential tail state faults, we computed only 2 recurrent ones exist, and are listed below:

Table 2: Recurrent fault list

Original transition	Faulty transition
s_6 —(LI/LI)→ s_1	s_6 —(LI/LI)→ s_6
s_7 —(LR/LR)→ s_1	s_7 —(LR/LR)→ s_7

The simulator consists of the following modules:

- The generator module generates input/output sequence with a fault randomly chosen and randomly injected in time.
- The simulator follows our approach to identify possible output faults and tail state faults.

Input/output sequence generator module: This module generates input/output sequences that have a fault injected as follows:

- Initially, the generator generates all possible Output and Tail-State faults for that FSM, and stores them for later random selection of a fault. It randomly selects at which step the fault injection is to occur as well.
- During simulating the forward trace $\{L^i\}$, the generator calculates the set of all valid and possible input/output symbols for the current set of states L^i. Then, it selects randomly an i/o pair and generates the corresponding set of possible states L^{i+1}.
- The generator keeps track of the current step in terms of observed transitions. When the fault injection step comes, the fault is in effect for the implementation FSM.
- Then the generator resumes simulating using the faulty implementation. It stops when L^k is empty.
- The generator logs the input/output sequence to the test-bed file.

Fault Identification module: This module is the main simulator module.

- It reads the specification FSM, the test file containing the input/output sequences, and the recurrent fault file.
- During the forward trace: it applies the input/output sequence until L^i becomes empty.
- Then, during the backward trace at each step it identifies output and tail-state faults.
- For each recurrent fault, which is undetected by the backward phase, the simulator checks whether that fault can generate the observed input/output sequence using a simple forward checking procedure.

6.2 Results and Comments

- The experiment has been run for 50,000 random input/output sequences on the FSM shown above.
- For each of these sequences, both forward and backward analyses have been done to identify output and tail-state faults.

The following diagram shows the distribution of the sequence length:

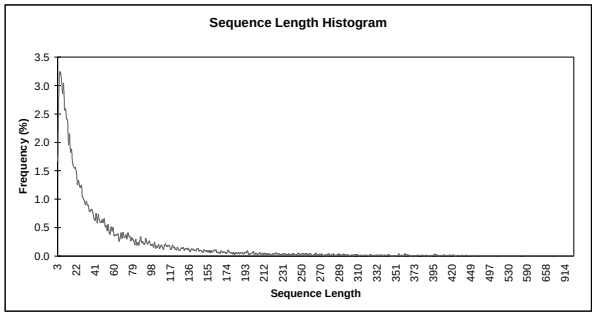

Figure 11. Histogram of test sequence length

- It can be seen that most of the sequence lengths lie between 3 and 13.
- From the above diagram, the following can be observed:
- The machine does not take long (many input/output symbols in passive testing) to detect the injected fault.
- To keep our next metrics (Mostly averages) statistically significant, we truncated all results for sequence lengths longer than 40 symbols.

- The Data below shows the following metrics distributed over different sequence lengths:
    - Average back steps taken in the backward processing,
    - Average count of the faults, both output and tail-state respectively,

  The average back-steps can be demonstrated below:

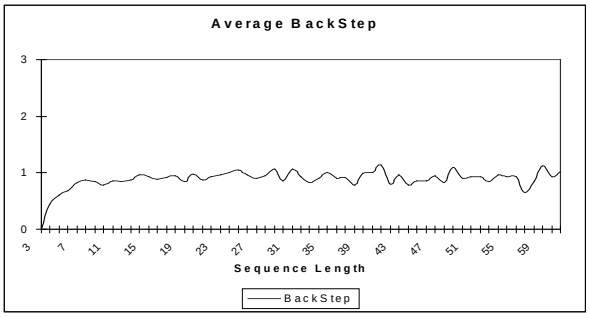

Figure 12. Average Backward Trace Length

- It can be noticed that average number of back-steps needed for backward analysis (post observation analysis) is small (around 1), which demonstrates that this approach is quite efficient.

Below the average Output and Tail State faults are plotted against sequence length:

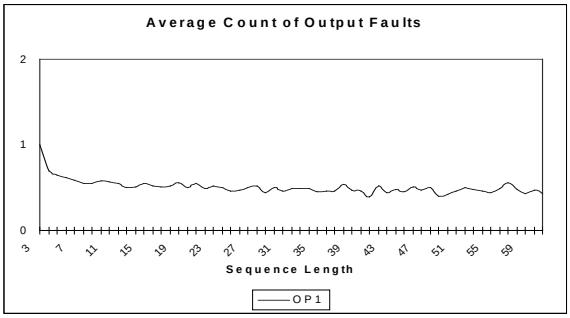

Figure 13. Average Count of Identified Output Faults

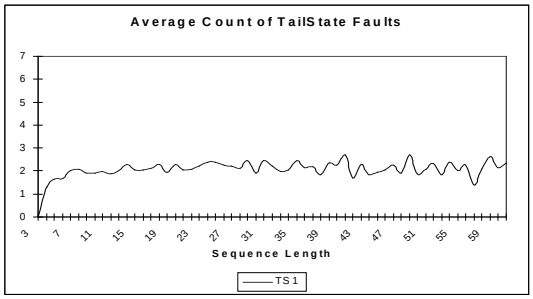

Figure 14. Average Count of Identified Tail-State Faults

- The fairly constant count of faults is a very good result since it shows how efficient we can determine the faults that may occur. Having output faults on average close to 1 and having tail state faults on average between 1 and 2, means that almost we can determine a small set of potential faults to be tested in later fault correction process.

7 Conclusions and Possible Extensions

In this paper we have shown how passive testing can be used to reduce the number of faults that could have caused a network implementation to display faulty behavior. The network specification was assumed to be a DFSM and the faults considered were output fault and tail-state fault deviations from the specification. Thus, once a fault is detected this fault identification approach can be used to narrow the possibilities of what fault occurred, thus simplifying the following tests of uniquely identifying and correcting the fault. The approach was demonstrated through several examples. An X.25 example was used to demonstrate the effectiveness of the approach on a practical example. Extensive simulation was done for the X.25 example over many simulation input/output sequences and many random injections of faults. This simulation demonstrated that the set of possible faults could be determined after only a very modest number of steps once a fault was injected, and also showed that considerable reduction in the number of possible faults giving rise to the observed input/output sequence was obtained by our approach. In fact in some cases this reduced to a unique fault.

Much remains to be done. First, the fault coverage attained by the observed input/output sequence can also be determined. We have shown that many of the approaches can be used to determine the fault coverage. These results are being included in a forthcoming technical report, and hopefully a future paper. There are other major challenges, however. It would be nice to have other practical examples for more up-to-date protocols and networks that would display the behavior of our fault identification procedure. Also, as was done in some previous works, showing how the passive fault detection techniques on the FSM model could be expanded to fault location using a CFSM specification, we expect that our fault identification approach for FSM's developed here could be extended to faults in a network specified by the CFSM model. If so, passive testing could be used first for fault detection, followed by fault location and then followed by fault identification in the small region of the network containing the fault. Finally, fault coverage results would provide some assurances as to how "good" the test was.

Another issue is whether further passive testing, beyond when a fault is detected, could be used to provide better fault identification and coverage. After all, the implementation would keep running even though it is known to be faulty, and thus some method of continuing the analysis might be possible and useful for gaining more information. Although we have some initial thoughts on how this might proceed we do not have any definite results to report.

The final challenge is to see how the techniques that have been developed for passive testing might be applied in the fault management systems of real network management tools. This somewhat formal approach and way of thinking seems to be quite distant from the techniques currently used in actual network management systems.

8 References

[1] D. Lee, A. Netravali, K. Sabnani, B. Sugla, and A. John, "Passive Testing and Applications to Network Management," *Proceedings of IEEE International Conference on Network Protocols*, pp. 113-122, October 1997.

[2] R. E. Miller, "Passive Testing of Networks using a CFSM Specification," *1998 IEEE International Performance Computing and Communications Conference*, pp. 111-116, February 1998.

[3] R. E. Miller, "Passive Testing of Networks Using a CFSM Specification," *Bell Labs Technical Memorandum*, BL011345-97-0522-03TM.

[4] R. E. Miller and K. Arisha, "On Fault Location in Networks by Passive Testing," *2000 IEEE International Performance Computing and Communications Conference*, February 2000.

[5] R. E. Miller and K. Arisha, Computer Science Dept., University of Maryland College Park, *Technical Report* #4044, August 1999.

[6] W. Stallings, "*SNMP, SNMPv2, and CMIP The Practical Guide to Network-Management Standards*," Addison-Wesley Publishing Company, 1993.

[7] ISO/IEC 7498-1: 1994 | ITU-T Recommendation X.200 (1994) Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model, 1994.