FAULT DETECTION OF FSM BASED SYSTEM

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Computer Science & Engineering

Thapar University, Patiala

By:

Pradeep Kumar
(80632017)

Under the supervision of:

Mr. Ajay Kumar
Lecturer, CSED

JULY 2008

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "**Fault Detection of FSM Based System**", in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mr. Ajay Kumar and refers other researcher's works which are duly listed in the reference section.
The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Pradeep Kumar

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Mr. Ajay Kumar
Lecturer
Computer Science and Engineering Department
Thapar University , Patiala

Countersigned by

Dr. (Mrs) Seema Bawa
Professor & Head
Computer Science & Engineering Department
Thapar University
Patiala

Dr. R.K. Sharma
Dean, Academic Affaris
Thapar University,
Patiala.

Acknowledgement

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. This thesis is the result of work carried out during the final year of my course whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

No amount of words can adequately express the debt, I owe to ***Mr. Ajay Kumar, Lecturer***, CSED, Thapar University, for his kind support, motivation and inspiration that triggered me for the thesis work. I owe him lots of gratitude for having me shown this way of research. He could not even realize how much I have learned from him.

I wish to express my gratitude to ***Dr. (Mrs.) Seema Bawa,*** Head of Department, CSED and ***Mrs. Damandeep Kaur,*** P.G. Coordinator, for their excellent guidance and encouragement right from beginning of this course. I am also thankful to all the faculty and staff members of the Computer Science & Engineering Department for providing me all the facilities required for the completion of this work.

Most importantly, I would like to give God the glory for all of the efforts I have put into this report.

Pradeep Kumar
Roll no. 80632017

Abstract

Passive testing is very important to guarantee the correctness of the fault detection. Passive Testing is a new concept of fault detection of any network in recent years. Fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the input/output behaviour of the SUT without interfering its normal operations. A new approach to Finite State Machine (FSM) based passive fault detection which improves the performance and gathers information during testing compared with the approach.

The results of theoretical and implementation evaluations are reported. Testing is a trade-off between increased confidence in the correctness of the system under test and constraints on the amount of time and effort that can be spent in testing.

The presented thesis provided overview of passive fault detection algorithms which determine system under test (SUT) is faulty by observing the input/output behaviour. We are able to take output of different fault detection algorithms and Compare these fault detection algorithms theoretical and implementation based upon system under test. The experimental and implementation results of the passive testing on fault detection algorithms are also shown and analyzed. We employ the finite state machine (FSM) model for networks to investigate fault identification using passive testing.

Keywords: system under test, fault detection, passive testing.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Finite state machines have been widely used to model systems in diverse areas, including sequential circuits (in lexical analysis, pattern matching etc.) and more recently, communication protocols .The demand of system reliability motivates research into the problem of testing finite state machines to ensure their correct functioning and to discover aspects of their behavior.

The theory is very similar for the two types. We discuss the following two types of testing problems. In the first type of problems, we have the transition diagram of a finite state machine but we do not know in which state it is. We apply an input sequence to the machine so that from its input/output (I/O) behavior we can deduce desired information about its state. Specifically, in the state identification problem we wish to identify the initial state of the machine, a test sequence that solves this problem is called a distinguishing sequence. Here we will focus on the basic problems of testing finite state machines and present the general principles and methods.

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 introduced the framework for testing problems. Moore studied the related, but harder problem of machine identification: given a machine with a known number of states, determine its state diagram. He provided an exponential algorithm and proved an exponential lower bound for this problem. He also posed the conformance testing problem and asked whether there is a better method than using machine identification. Furthermore, only exponential algorithms were known for determining the existence and for constructing such sequences. Hennie also gave another nontrivial construction of checking sequences in case a machine does not have a distinguishing sequence. 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 research on fault detection was by Vasilevskii, who proved polynomial upper and lower bounds on the length of checking sequences.

Several researches were published in the 60's on testing problems, motivated mainly by automata theory. 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 discover on fault detection was by Vasilevskii, who proved polynomial upper and lower bounds on the length of checking sequences Machines, such as nondeterministic and probabilistic finite state machines.

1.2 Background

Finite state systems can usually be modeled by *Mealy* machines that produce outputs on

Their state transitions after receiving inputs.

Definition 1.2.1: A finite state machine (FSM) M is a quintuple M (I, O, S, δ, λ)

Where I, O, and S are finite and nonempty sets of input symbols, output symbols, and states, respectively.

δ: $S*I \rightarrow$ δ is the state transition function;

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

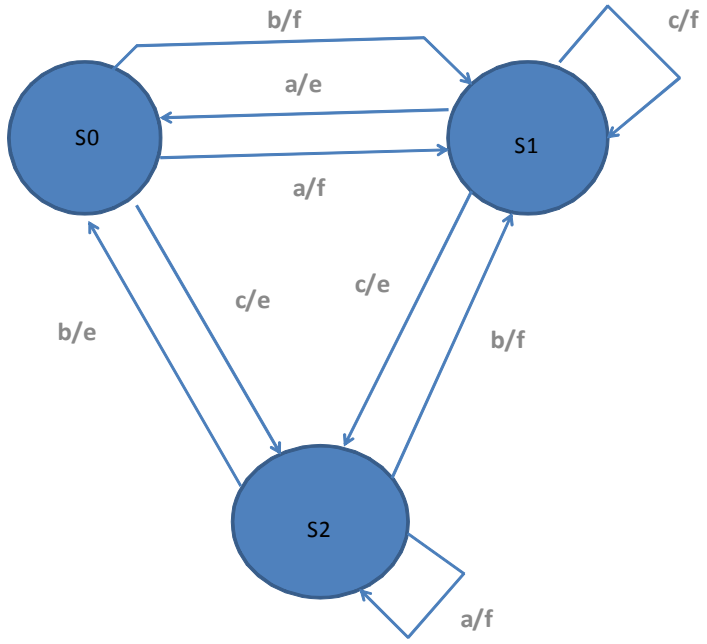


Figure 1.1: Specification of FSM M[27]

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 δ (s, a) and produces an output given by λ (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 d and the output function l can be extended from input symbols to strings as well. Starting from initial state $s0$, an input sequence $x = a0,a1,\ldots,ak$ takes the machine successively to states $si+1$= d (si, ai), $i=0,1,\ldots,k$, with the final state d$(s0, x) = sk+1$, and produces an output sequence y= λ $(s0, x)=b0,\ldots,bk$, where bi= δ (si,ai), $i=0,1,\ldots,k$.

Let M be the FSM shown in Figure 1.1 with X = {a, b, c} and Y = {e, f}.

The specification has the following characterization set

For state : a/e, b/f

For state : a/f, b/f

For state : a/f, b/e

1.3 Assumptions and Justifications for The FSM Model

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.4 The Fault Model

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 briefly covers the passive testing algorithm.

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

1.4.1 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.

1.4.2 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.

Chapter 2

Literature Survey

2.1 Development Cycle Using FSM Tools

A set of software tools that have been developed to create, manipulate, verify and execute logic control systems written using finite state machines. Modular Finite State Machines (MFSMs) were developed as a new framework for logic control for large systems. They are used for developing, verifying and executing discrete event control systems.

The main advantage of MFSMs is their modular structure which facilitates the creation of large systems. A designer would perform the following steps while using the software tools to aid in the development of a MFSM system.

1. Create text files defining the system. This includes the overall system file as well as the modules and filters. The same module or filter file can be used in several places within a system or in different systems.
2. Load the files into the software tools. The software tools check the files for errors and consistency. If errors exist the software tools report the nature of the error and the line on which it was encountered.
3. Verify the system. The verification tools are defined for single modules. For small systems, the entire system can be composed into a single module. For larger systems, the designer would create subsystems bounded by filters, and verify each one by composing it into a module and checking the module.
4. Create a text file describing the IO for the system**.** A MFSM system interacts with its environment through external filters, such as input bits, output bits and timers. The software tools use FSM definitions of these external filters to perform the verification.
5. Load the IO description into the software tools. As with the system, module and filter files, the software tools check the IO description file for errors.
6. Run the system in the execution environment. the execution environment will attempt to attach the external filters to the environment using the information contained within the IO description. Once all errors are eliminated, the execution environment

2.2 Application to Real Protocols

2.2.1 Extended Finite State Machine

In order to specify a real protocol, we must use the Extended Finite State Machine (EFSM) formalism there may be a predicate on variables associated with each transition of an EFSM. The use of extended finite states machines raises a new issue. In active testing, it is easy to know the value of the variables, because we know the initial state.

In passive testing, we do not know their value, since the machine can be in any state at the beginning of the trace. We propose several approaches.

• generate the reachability graph

• transform the extended finite state machine into a nondeterministic finite state machine

2.2.1.1 Reachability Graph

The simplest approach is to generate the reachability graph, which is tantamount to unfolding the EFSM to its equivalent FSM. The reachability graph we obtain is an unfolding of the EFSM specification, which includes instantiations of the variables and parameters used in the original specification.

2.2.1.2 Transforming the EFSM into an NDFSM

Figure 2.1 a/Observable[23]

EFSM can be transformed into an NDFSM by removing the predicates enabling the transitions

Figure 2.2 b/ Unobservable[23]

This method, the detection power is lessened. Faults caused by bad parameters values or by control variables will not be detected.

2.3 Fault Location in SDH/WDM Networks

The components of an optical network, in the presence of a fault, aiming to find which network component is causing the failure. The algorithm showed, named Correlated Fault location Algorithm, CFLA, uses the alarms' correlation in order to reduce the list of suspected components shown to the network operators. In this algorithm operators analyze the alarms and based on them, they make decisions in order to solve the problem. So having the alarms information, the operator has to locate the fault. Its main advantage over other existing algorithms is the low number of information necessaries in order to give the fault location.

2.3.1 Network Components

An optical network is basically composed by nodes linked by optical fibers. The node is the place where the main optical devices as transmitter, receiver, add/drop filters and switch, among others, are located channel is established when one node communicates to another node.

Following are the elements considered in our network model are defined.

1. Network's components
2. Interconnection
3. Alarms
4. Channel

Every time a fault occurs an alarm storm arrives at the management room and it is analyzed by the network manager. Aiming to improve the quality of the input data and the answers of the network's operators, to reduce the redundancy of operations, to diminish the risks in the decisions and to increase the velocity of answer when a fault occurs, an algorithm for fault location was developed. The ability to find damaged passive components gives more power and efficiency to the algorithm proposed. In order

to realize the location of the damaged components, the algorithm needs only the identification of the elements that are sending alarms to the management room. The automatic processing of the fault location relation realized by the algorithm proposed reduces also the necessary time required by the complete solution of the problem.

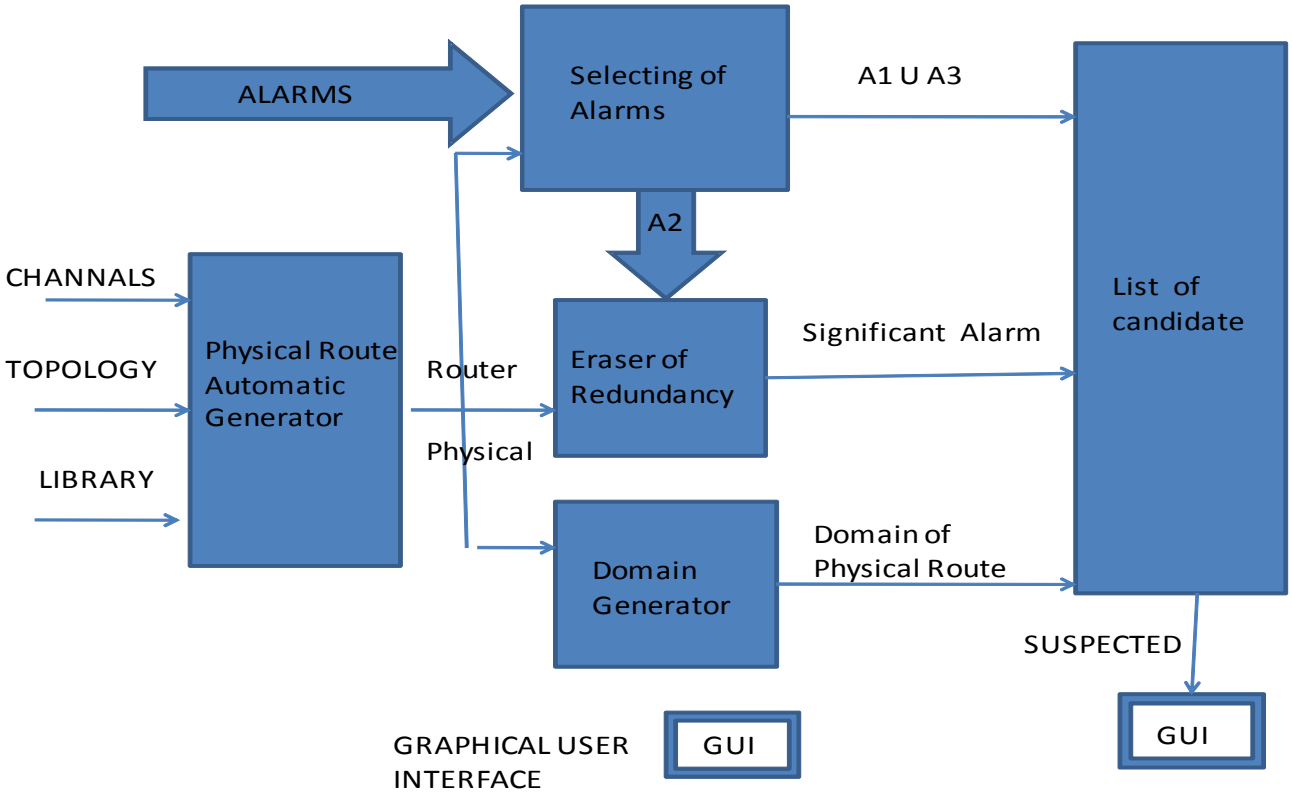


Figure 2.3 The graphic representation of the fault location process[26]

2.4 Fault Coverage of Finite State Specifications

Underlying the techniques for fault coverage analysis and assurance mainly developed in the context of protocol conformance testing based on finite state models. Special attention is paid to parameters which determine the testability of a given specification and influence the length of a test suite which guarantees complete fault coverage.

The essential idea of this correctness-proving viewpoint is that an execution scenario in which no error is detected should ensure that the implementation is free of faults. A coverage criterion guides a testing strategy and sets requirements for a test suite based upon coverage of certain characteristics associated with the given specification. The list of characteristics considered for defining the test coverage criteria

includes the following.

- Conformance requirements
- Specification structure
- input domains

In the case of an FSM specification, a branch cover constitutes a transition tour which leaves many control errors untested.

The coverage criteria described may be used with two different testing paradigms.

1. Exhibiting correct behavior concerning the criteria, or
2. Discovering any implementation faults in relation to the criteria.

2.4.1 Fault models

The formidable obstacle in constructing or analyzing a test suite is that it must verify whether a given conformance relation defined on an infinite set of input sequences, holds between two machines. There are at least two other techniques to describe this set.

1. limiting the number of states
2. The use of fault functions.

A mutant is an FSM obtained by applying to MS (which might be a partial machine) each of the following four types of operations.

1. Alter the tail state of a transition (a transfer fault)
2. Alter the output of a transition (an output fault)
3. Add a transition
4. Add an extra state.

As an example, consider the FSM MS with two states {1,2}, two inputs {*a*,*b*} and two outputs {*y*,*z*} shown in Figure 2.4. Assume that the transition 1-a/y-->2 is correctly implemented, and the remaining are suspicious transitions.

In particular, transition 1-*b*/*z*-->1 can have only output faults, 2-*b*/*y*-->1 can have only transfer faults, and transition 2-*a*/*y*-->1 can have any fault. The corresponding fault function is represented by Table 2.1. Here, the entries shown in bold represent the state table of the specification FSM in Figure 2.1.

Table 2.1 can be interpreted as a state table of a nondeterministic FSM. It represents 1 * 2 *2 * 4 = 16 mutants that are deterministic submachine of this nondeterministic machine, in other words, |Impl(MS)| = 16. Note that there exist 128 different FSM with two states, two inputs and two outputs, i.e. |Impl (2, MS)| = 128.

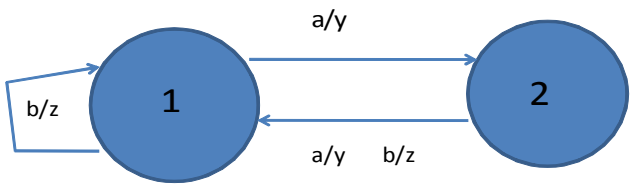


Figure 2.4 : An FSM[28]

	a	**b**
1	2/y	1/z, 1/y
2	1/y, 1/z ,2/y, 2/z	1/y,2/y

Table 2.1: A fault function

2.4.2 Fault coverage

To define the fault coverage, we need to use the following notations.

Nt (m, MS) - the total number of machines in Impl (m, MS).

Nc (*m, MS*) - the number of machines in Impl (m, MS) which conform to MS.

Np (m, MS, TS) - The number of machines in Impl (m, MS) which can pass the given test suite *TS*.

Nt (m, MS) - Nc (m, MS) is the number of machines in Impl (m, MS) which do not conform to MS.

Nt (m, MS) - Np (m, MS, TS) is the number of machines in Impl (m, MS) which cannot pass the given test suite TS (and therefore do not conform to MS).

The fault coverage of a test suite TS with respect to MS, denoted as FC (m, MS, TS), is

FC (m, MS, TS) =Nt (m, MS) - Np(m, MS, TS) / Nt(m, MS) - Nc(m, MS).

Problems with this formula:

- The fault coverage is determined, only if we make the additional assumption that the output alphabets of all the machines in Impl(m, MS) are subsets of Y - the output alphabet of MS (under

this assumption, Nt(m, MS) = (m|Y|)m|X|, where X is the input alphabet of MS), otherwise the cardinality of Impl(m, MS) would remain unknown. Thus, the "real" coverage is much higher, since a large number of implementations with a "foreign" output symbol are also detected by the test suite.

- The value of FC (m, MS, TS) is not equally distributed over [0,1]. A TS consisting of a single test event already has the coverage of more than (|Y|-1) / |Y|. If we are given an FSM with, for instance, just ten outputs, the fault coverage of a single test event is already over 90%. As the number of outputs in the MS increases, the fault coverage of a single test event approaches 100%.
- For real protocol machines, it often occurs that Nt(m, MS) >> Nc(m, MS) and Nt(m, MS) >> Np(m, MS, TS) and therefore FC(m, MS, TS) >> 100%. Thus calculations with a normal precision might not be sufficient to compare the test suites by their fault coverages.

2.4.3 Fault Coverage Analysis

The purpose of fault coverage analysis is to calculate the fault coverage for a given test suite TS, a specification MS and a fault model. In the following, we limit ourselves to the fault model Impl(m, MS) of all complete FSM specifications with a number of states less or equal to m.

1. Exhaustive mutation analysis
2. Monte-Carlo simulations
3. Structural analysis

Chapter 3

Testing

3.1 Testing

Development of software systems is comprised of three stages. In the first stage, developers of the system derive a set of requirements from their customers. These requirements are normally represented in a requirements specification. Then, in consultation with these requirements, a design is built. After that Coding or implementing takes place where the design is translated into code using some programming language. Errors might be introduced at this stage, but can be discovered by verification and testing. Testing is an integral and important part in the life cycle of software development. A testing process aims to check whether the implementation under test is functionally equivalent to its specification.

"Testing is the process of executing a program or system with the intent of finding errors or involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results".

The process of testing begins with test design. A set of tests is normally created through the analysis of the system under test. This set of tests is used to check whether the system has been correctly implemented. In the phase of test design, a test model is often required in order that the generation of tests is formalized. This model describes the system behavior with abstracted information, aiming to reduce the complexity of the description of the system being developed. The test model can be constructed by using either informal specification languages or formal specification languages. Due to the properties of imprecision and ambiguity, informal specifications often lead to misunderstandings and make testing difficult and unreliable. By contrast, formal specification languages are based upon mathematics and have a formally defined semantics.

A test strategy is an algorithm or heuristic to create test cases. Two measurements are applied for the evaluation of efficiency of a test. One of the measurements is test cost while the other is fault coverage. A good test strategy needs to embody the two measurements in two aspects:

1. Test cases generated with such a strategy should cover, as much as possible, all faults that the system under test may have.
2. Test cost associated with these test cases should be relatively low. Unfortunately, detecting all

faults is generally infeasible.

Howden suggests that there is algorithm to find consistent, reliable, valid and complete test criteria. Complete testing is in general a very difficult process. Instead, testing provides a level of confidence in the correctness of an implementation with regard to the constraint of some test criteria. Exhaustive testing, where the test cases consist of every possible set of input values, is the only way that will guarantee complete fault coverage.

This technique is size of the input domain makes exhaustive testing infeasible. Regardless of the limitations, testing is an expensive process, typically consuming at least 50 % of the total costs involved in the development while adding nothing to the functionality of the product. It has been suggested that manually generating test cases could be very difficult even for moderately sized systems. Although, for some systems, it is possible to generate test cases manually, the process tends to be costly and inefficient. Automation of the testing process is thus required, which could be desirable both to reduce development costs and to improve the quality of (or at least confidence in) software.

3.2 Passive testing

Today's networks are becoming larger, more heterogeneous and are assembled by integrating equipment from multiple vendors. Consequently, managing networks and network devices is becoming increasingly difficult, making the development of automated network management tools and techniques more important. A key requirement of network management systems is detecting faults, where a fault identifies abnormal behavior in a network or network device. Faults are detected by analyzing monitored network traffic or device state, the observed data is mid-stream in that the network may already have been in operation for some time and the fault detection process is passive in that it cannot affect the normal operation of the network or network device, such as by injecting arbitrary traffic for the purposes of testing.

3.2.1 The Basics of Passive Testing

In passive testing contrary to active testing, the tester does not control the implementation under test. The implementation is in operating condition, and the tester only observes the messages exchanged by the IUT and its environment, in order to check if they correspond to behavior compliant with the specification (see Fig. 3.1). The main difficulty in passive testing is that we have no knowledge of the state in which the implementation is at the beginning of the trace (no assumption is made about the moment when the recording of trace begins and therefore it is not necessarily the initial state). Each

input/output pair of the trace is assumed to represent a transition in the specification and our objective is to match the transitions of the trace with those of the specification.

The passive testing process can be decomposed into two steps; the passive homing sequence, in which the current state is found out and the fault detection phase, in which the trace is compared with the specification.

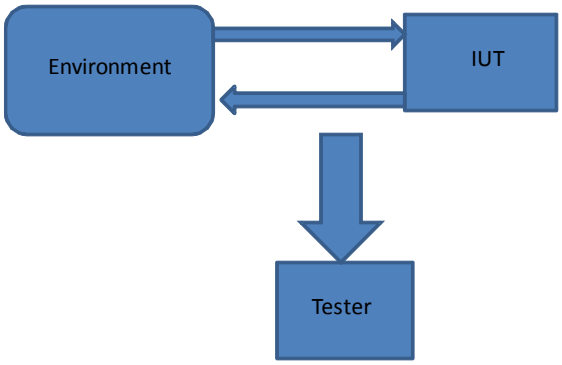


Figure 3.1 An architecture passive testing[23]

3.2.1.1 First Stage: Passive Homing Sequence

The current state is determined by elimination. Initially, all the states are candidates. The transitions of the trace are studied one after the other: the states which accept the input/output are replaced by the destination state of the corresponding transition (redundant states are eliminated), whereas the states which do not accept it are eliminated. After a number of iterations, there are two possible outcomes:

1. Either a single state is obtained: it corresponds to the current state and we proceed to the second stage.
2. An input/output pair that is not accepted by any candidate state is encountered in the trace; it indicates that the behavior observed does not correspond to the specification, and that a fault has been detected.

3.3.1.2 Second Stage: Fault Detection

From the current state, we follow the trace in the specification. If we reach a state which does not accept the following input/output pair of the trace, there is an error. If not (*i.e* the end of the trace is reached and there were no deadlocks), no error was detected.

Passive testing is weaker than other well-known validation methods such as verification and active testing in that system implementation details are unknown, inputs are not controllable and the system need not be in its initial state when testing begins. While several practical monitoring tools have been described, both for network management and for detecting security intrusions, the first formal and systematic treatment of passive testing modeled a network as a (possibly non-deterministic) finite state machine and gave algorithms for detecting if an implementation machine conforms to a given specification machine.

The fundamental idea underlying these passive testing algorithms is to perform a "homing" process which seeks to infer the current state of the specifications. In this passive testing algorithm is providing a formal setting in which the problems of fault identification and fault location are addressed by adapting the passive testing algorithm suitably. We precisely analyze the conformance relation passively tested by the homing algorithm. We first formulate general correctness conditions on any algorithm which seeks to passively test an implementation for some particular property. We call an algorithm (for testing) if, whenever it rejects an implementation, the implementation is indeed faulty. We term the algorithm complete if, in addition to being testing, it rejects a faulty implementation whenever observations are made that could not have been generated by a correct implementation.

Given that these conformance relations are strictly distinct, it is somewhat countintuitive that the same testing algorithm could be complete for passively testing all of these different conformance relations. For example, there are implementations that are trace-contained in a specification machine but not observationally equivalent to it. By the testing algorithm for trace-containment, it accepts such implementations that are faulty with respect to observational equivalence.

However, the completeness of the homing algorithm with respect to observational equivalence establishes that no other passive-testing algorithm could detect such implementations as faulty, either. Thus, the homing algorithm's failure to reject these faulty implementations is not as much a reflection on the homing algorithm as an intrinsic inability of the passive testing methodology to completely detect non-conformance according to observational equivalence.

We show that a passively testable property must be a safety property. This yields an alternate, more direct proof that observational equivalence and trace equivalence are not passively testable conformance relations. However, we exhibit an example of a safety property that is not passively

testable which shows that safety is a necessary but not sufficient condition. We develop an exact characterization of passively testable properties as being validity conditions on traces that are prefix and suffix closed.

Applying the conformance testing, we obtain that conformance is passively testable exactly when.

1. The conformance relation is trace containment.
2. The set of traces of the specification machine is suffix-closed.

Using this characterization of passively testable properties, we show that while a passive-testing algorithm for an arbitrary property must account for the unobserved initial behavior of the system, a passive-testing algorithm for a passively testable property can "pretend" as if the system is being observed from its initial state without sacrificing.

This allows us to derive complete passive testing algorithms for such properties that have O(1) running time, in contrast to the homing algorithm which has a running time of O(n) for each observation made, where n is the number of states in the specification machine.

There are two approaches to test for fault detection 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 application of 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. It is desired to keep testing traffic overhead to a minimum. Passive testing simply observes the normal traffic of the application 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. The simplest approaches to passive testing use a FSM specification to model the behavior of the application 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.

The effective fault detection capabilities of passive testing based on observation of the input/output sequence of the implementation. We apply passive testing of the fault detection algorithms, one of the most widely-deployed in application networks. In particular, the more complex and intricate core of the algorithms that is concerned with fault detection and an adaptive retrains mission is well amenable to passive testing.

3.3 Back-box testing

In back-box testing a system is tested against its requirements without having internal knowledge of how the system was implemented. Test cases are generated from the system specification. Only information about what inputs does the system expect and what are the specified outputs is available, without knowledge of how the system derives those results. Since this no knowledge of the implementation of the system is required in black box testing, test cases can be designed as soon as the system specifications are complete. They test not only individual system components but also the interaction between them. The test cases are implementation independent. Typical black box design techniques include are

1. Equivalence portioning
2. Boundary value analysis
3. Decision table testing
4. Pair wise testing
5. State transition tables
6. Use case testing
7. Cross function testing

3.4 White-box testing

White-box testing uses information from the internal structure of a system to devise tests to check the operation of individual components or the system as a whole. Black-box and white-box testing both choose test cases that investigate a particular characteristic of the system, however in white-box testing test cases can be generated to test some implementation specific aspects of the system. Typical white box design techniques includes

1. Control flow testing
2. Data flow testing

3.5 Conformance testing

When testing from an FSM model M it is assumed that the implementation under test (IUT) can be modeled by an unknown FSM M0 and thus that testing involves comparing the behavior of two FSMs. Verifying that M0 is equivalent to M by only observing the input/output behavior of M0 is known as conformance testing or fault detection.

Often a fault can be categorized as either an output fault or a state transfer fault. Output faults are those faults where the wrong output is produced by a transition and state transfer faults are those faults where the state after a transition is wrong. An output fault can be detected by executing a transition and observing its output. A state transfer fault can be detected by checking if the final state is correct after the transition is executed.

The first step is known as homing a machine to a desired initial state . It can be done by using a homing sequence which can be constructed in polynomial time . The second step, transition verification, is to check whether M0 produces a desired output sequence. The last step is to check whether M0 is in the expected state = (, x).

There are three main techniques that can be used in state verification:

• Distinguishing sequence (DS)

• Unique input/output sequence (UIO)

• Characterizing set (CS)

Chapter 4

Testing from finite state machines System

4.1 Testing from finite state machines

Finite state machines have been used to model systems in different areas like sequential circuits, software development, and network management and communication protocols. When testing against a finite state machine we can only test that the specification has been correctly implemented. This is usually done by observing the output behavior of the implementation and comparing that to the specified output behavior. This process is known as conformance testing

4.1.1 Testing Problems

In a testing problem we have a machine *M* about which we lack some information and we would like to deduce this information by its I/O behavior. We apply a sequence of input symbols to *M*, observe the output symbols produced and infer the needed information of the machine.

Problem-1 (Homing/Synchronizing Sequence) Determine the final state after the test.

Problem- 2 (State Identification) Identify the unknown initial state.

Problem- 3 (State Verification) The machine is supposed to be in a particular initial state. Verify that it is indeed in that state.

Problem-4 (Machine Verification/Fault Detection/Conformance Testing) We are given the complete description of another machine *A*, the ‘‘specification machine’’. Determine whether *M* is equivalent to *A*.

Problem- 5 (Machine Identification) Identify the unknown machine *M*.

4.2 Algorithm for Passive Fault Detection

We first present the algorithm for fault detection by Lee, then the new algorithms for FSM-based passive fault detection. In order to make the analysis and further comparisons of the algorithms, we consider the number of comparisons between the actual output y_j and the expected output $\lambda(s, x_j)$ as the measure of computational complexity, $1 \leq j \leq k$, $s \in S$.

4.2.1 The passive fault detection algorithms

We have rewritten the fault detection algorithm given as Algorithm 0 without changing its computational complexity. Consider the mealy machine represented by the following figure 4.1. We

convert the transition diagram into the following transition table.

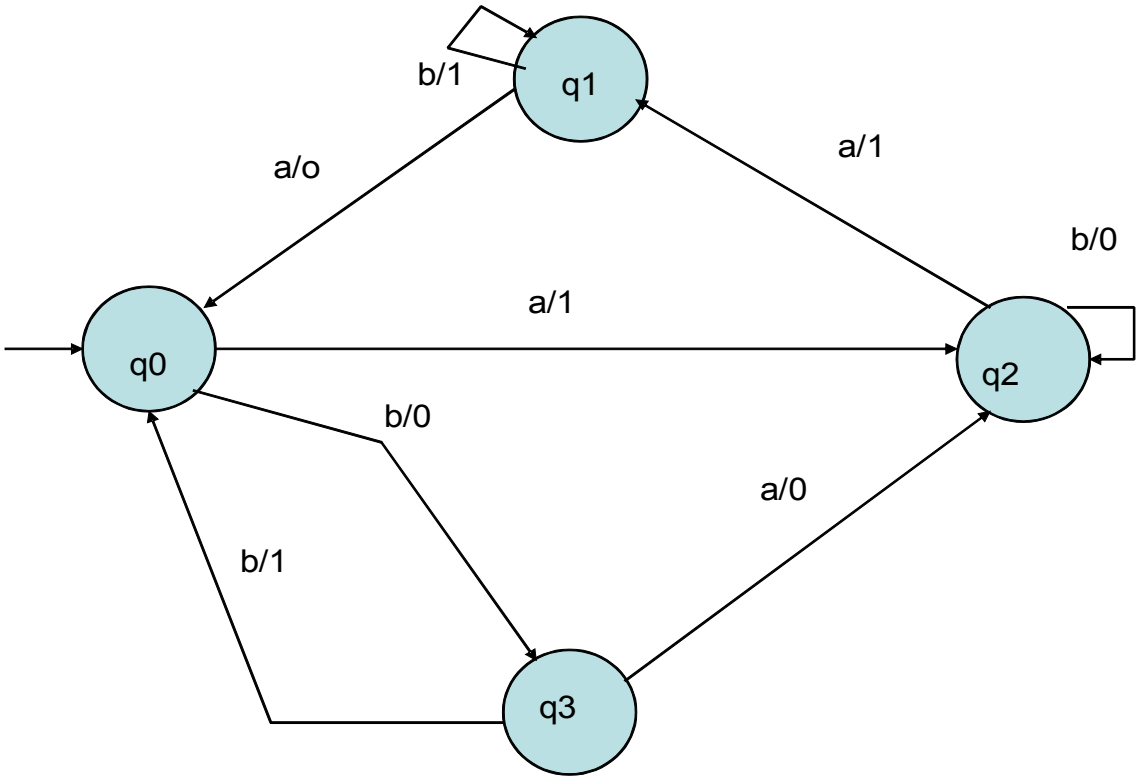


Figure 4.1: An FSM M

Present state	Next state Input a (state)	(output)	Next state Input b (state)	(output)
Initial q0	q2	1	q3	0
q1	q0	0	q1	1
q2	q1	1	q2	0
q3	q2	0	q0	1

Table 4.1: Transition table an FSM m

Algorithm- 0

Given: FSM $M = (S, X, Y, \lambda, \delta)$ $S_0 = S$,

I/O sequence $Q = (x_1/y_1)(x_2/y_2)\ldots(x_k/y_k)$

Begin:

$j \leftarrow 1$ /* j is the counter for I/O pairs */

$S' \leftarrow S_0$;

While ($j \leq k$)

If ($S' \neq \emptyset$) {

$S'' \leftarrow \emptyset$;

For (each $s \in S'$) /* check each state in S'*/

if ($= \lambda(s, x_j)$) $S'' \leftarrow S''$ U $\delta(s, x_j)$;

/*redundant states in S'' are removed*/

End for

$S' \leftarrow S''$;

$j \leftarrow j+1$;

}

Else /* $S' = \emptyset$*/

Return ("N is faulty");

End while

Return ("No fault is detected by Q and the set of possible current states is S"");

End

If the while loop terminates before the entire Q is checked, N is declared to be faulty. Otherwise, Q is declared to be insufficient to determine whether N is faulty. In this case, the possible current states are determined but the possible starting states (where Q starts) are unknown. In order to find the set of possible starting states, a post-processing will be needed.

Let s_j denote the set of possible current states right after the first j I/O pairs of Q, i.e.,$s_j = \lambda(s_0, x_1 x_2 \ldots x_j)$. The computational complexity of Algorithm 0 is $c_1 = (\text{sigma})^{k}_{j=1} | s_{j-1} |$

In Algorithm 0, every state in the set of possible current states will be checked to compare its related I/O pair with the current I/O pair in Q, i.e., for a state s in s_j , there will be one comparison between y_{j+1} and the expected output $\lambda(s, x_{j+1})$, and $| s_j |$ comparisons are needed to check the set . Thus, the total number of comparisons is $(\text{sigma})^{k}_{j=1} | s_{j+1} |$.

4.2.2 Passive fault detection algorithm-1

Algorithm 1 is based on proposed approach which checks, for each state $s \in S_0$ whether Q is a trace of M at s. It terminates when Q is verified to be a trace of M at a state $s \in s0$ or when all states in s_0 are checked and no state is found compatible with Q.

Algorithm -1
Given: FSM $M = (S, X, Y, \delta, \lambda)$ $s_0 \subseteq S$,
I/O sequence $Q = (x_1/y_1)\ (x_2/y_2)\ldots(x_k/y_k)$
Begin:
$i \leftarrow 1$; /* i is the state counter */
While ($i \leq n$)
$j \leftarrow 1$; /*$\leftarrow j$ is the counter for I/O pairs */
$s \leftarrow s_i$; /* s will represent , $x_1 \ldots x_{j-1}$) when $j > 1$ */
While ($j < k$ AND y_j ($\lambda = s, x_j$))
$s \leftarrow \delta(s, x_j)$
$j \leftarrow j+1$; /*s is updated as the current state*/
End while
If ($j = k$ AND $y_j = \lambda(s, x_j)$)
Return ("Q is a trace of M at state si and The possible current state is s");
Else
$i \leftarrow i+1$;
End while
Return ("N is faulty");
End

Algorithm 1 either declares N to be faulty or yields both the possible current state (s) and possible starting state (s_i) once a state compatible with Q is found.

For the given state s_i of M and I/O sequence $Q = x_1, x_2, ..x_k / y_1\ y_2 ...y_k$ let $c_i(Q)$ denote the largest number j ($1 \leq j \leq k$) such that

$y_1 y_2 ...y_{j-1} = \lambda(s_i, x_1 \ldots x_j)$ and $\lambda(\delta(s_i, x_1 \ldots x_j \ldots)) \neq y_j$

Let $c_{2worst}(M, s_0, Q) = (\text{sigma})_{i=1}^{n} c_i(Q)$. If s_r is the first state of M such that Q is a trace of M at , then $(M, s_0, Q) = (\text{sigma})_{i=1}^{n} c_i(Q)$.

If N is faulty, then $(M, s_0, Q) = c_{2worst}(M, s_0, Q) = (\text{sigma})_{i=1}^{n} c_i(Q)$.

In Algorithm 1, each state in s_0 is checked to determine whether it is compatible with Q. The checking procedure for a state s_i will not stop until it confronts a mismatch (then the next state s_{i+1} will be selected to check) or the entire sequence Q has been checked and no mismatch found (then is reported to be compatible with Q). The whole checking procedure will terminate when a state compatible with Q is found or when all the states have been checked and no state is found to be compatible with Q.

Assume $y_1\ y_2 ...y_{j-1} = \lambda(s_i, x_1\ x_2 .. x_{j-1})$ but $y_j = \lambda(\delta(s_i, x_1\ x_2 .. x_{j-1})\ x_j)$, it means j comparisons

(denoted by $c_i(Q)$) are needed to determine that Q is not a trace of M at s_i. If $s_r \in s_0$ is the first state of M such that Q is the trace of M at s_r, Algorithm 1 will detect mismatch in checking s1 s2 .. s r-1and stop after checking . Thus, the total number of comparisons needed is $(\text{sigma})_{i=1}^{n} c_i(Q)$.

It encounters the redundant checking problem which is: two traces starting from different states converge to the same state after applying Q_j^p . In Algorithm 1, the common part Q_j^s will be rechecked redundantly.

Algorithm -2

Given: FSM $M = (S, X, Y, \delta, \lambda)$ $s_0 \subseteq S$,
I/O sequence $Q = (x_1 / y_1)(x_2/y_2)\ldots(x_k/y_k)$
Begin:
$F1\ldots k$-$\emptyset$;
i ← *1;*
While ($i \leq n$)
$j \leftarrow 1$; /*← j is the counter for I/O pairs */
$s \leftarrow s_i$; /* s (will represent $\delta(s_i, x_1 \ldots x_{j-1})$ when $j > 1$ */
While ($j < k$ AND $= y_j(s, x_1 x_2 .x_j)$)
$s \leftarrow \delta(s, x_j)$; /* s is updated as the current state */
If ($s \in Fj$) /* to eliminate *redundant checking problem* */
Break; /* state s has already been checked. Thus, end this trace */
Else
$j \leftarrow j + 1$;
End while
If ($j = k$ AND $y_j = \lambda(s, x_j)$)
Return ("Q is a trace of M at state s*i* and the possible current state is s");
Else
$i \leftarrow i + 1$; /*record the trace*/
If ($j > 1$) **add** $\delta(s_i, x_1 \ldots x_l)$ **to** , l=1,…,j-1;
End while
Return ("N is faulty");
End

For a given state of M and an I/O sequence $Q = x_1 \ldots x_k/y_1 \ldots y_k$, let $c'_i(Q)$ denote the largest number j ($1 \leq j \leq k$) such that

(1) $y_1 \ldots y_j = \lambda(s_i, x_1 \ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1 \ldots x_{j-1}) x_j) \neq y_j$;

(2) for every l ($1 \leq l \leq j$-1), $\delta(s_i, x_1 \ldots x_l) \notin$. F_l

If the rth state checked, s_r is the first state of M such that Q is a trace of M at s_r, then the computational complexity of Algorithm 2

$c_3(M, s_0, Q) = (\text{sigma})_{i=0}^{n}(Q)$

If N is faulty, then $(M, s_0, Q) = (\text{sigma})_{i=0}^{n}(Q)$;

If $r = 1$, $c_{3\,best}(M, s_0, Q) = c'_{i'}(Q)$.

Compared to Algorithm-1 the checking procedure of Algorithm 3 on state s_i will stop when it encounters a mismatch with Q, the whole Q has been checked compatible, or $\delta(s_i, x1x2\ldots xj) \in Fj$. Similar as for previous algorithm let $c'i(Q)$ denote the largest number j ($1 \leq j \leq k$) before checking on s_i terminates. If s_r is the first state of M such that Q is the trace of M at s_r , then the total number of comparisons needed is $\text{sigma}_{i=1}^{n}(Q)$.

4.3 Comparison of the fault detection Algorithms

We consider the number of comparisons between the actual output yj and the expected output $\lambda(s, xj)$ as the measure of computational complexity

The computational complexities of the three algorithms given in the previous subsections are summarized in Table 4.2

Table 4.2 Computational complexity

Type of algorithm	Computational complexity
Algorithm0	$C_1 = \text{sigma}_{j=1}^{k} \lvert s_{j-1} \rvert$
A lgorithm1	$C_2 (M, S_0, Q) = (\text{sigma})\, r_{j=1}\, c'_1 (Q)$
Algorithm2	$C_3 (M, S_0, , Q) = (\text{sigma})\, r_{j=1}\, c'_i (Q)$

1. k is the length of Q, $\lvert s_j \rvert$ is the number of states in the set of possible current states,
2. r is the number of states checked before a state compatible with Q is found,
3. $c'_i(Q)$ is the largest number j ($1 \leq j \leq k$) such that $y_1\ldots = \lambda(, x_1\ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j-1}),) \neq$
4. $c'_i(Q)$ is the largest number j ($1 \leq j \leq k$) such that $y_1\ldots y_{j-1} = \lambda(, x_1\ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j-1}),) \neq y_j$

Thus, Algorithm-2 always performs at least as well as Algorithm-0. The equality in their computational complexities occurs when $r = n$. Based on the computational complexities of the algorithms presented above, several assertions can be made on their performance in different conditions. When N is not determined to be faulty:

1. If there is no redundant checking problem, the performance of Algorithm -1 will be the same as that Algorithm -2 and be at least equal to that of Algorithm -0.
2. If there is redundant checking problem, the performance of Algorithm -2 will be at least equal to those of Algorithms-0 and-1 and it is not possible to compare the performances of Algorithm -0, Algorithm -1.

When *N* is determined to be faulty:

3. If there is no redundant checking problem, the performances of all the algorithms will be the same.
4. If there is redundant checking problem, the performance of Algorithm -2 will be equal to that of Algorithm-0 the performances of Algorithms-1 and it is not possible to compare the performances of Algorithms-0 .

4.4 Implementation of fault detection algorithms Evaluation

An implementation evaluation is made to compare the fault detection passive computational complexity of the algorithms and to verify the validity of the assertions drawn cases when $m = 0$ or 1.

In Case I, called correct implementation, there is exactly only one state in $S0$ that is compatible with Q ($m = 1$).

In Case II, called faulty implementation, there is no state in $S0$ that is compatible with Q ($m = 0$) and "faulty" is expected to be reported.

We apply all three algorithms to the FSMs in these two cases and record the results.

Experimental results confirm the assertion the output of theses Algorithms let M be as in figure 4.2 and transition table. We check for output result the above Algorithms, let Q(sequence)= "abbbab /100011" thus Q is declared to be sufficient, Q is trace of M , reported as "not faulty" .

Figure 4.2 Represent the not faulty implementation of FSM

If Q = “abbbab/100001”is insufficient at M, Q is not trace of M , reported as “faulty”

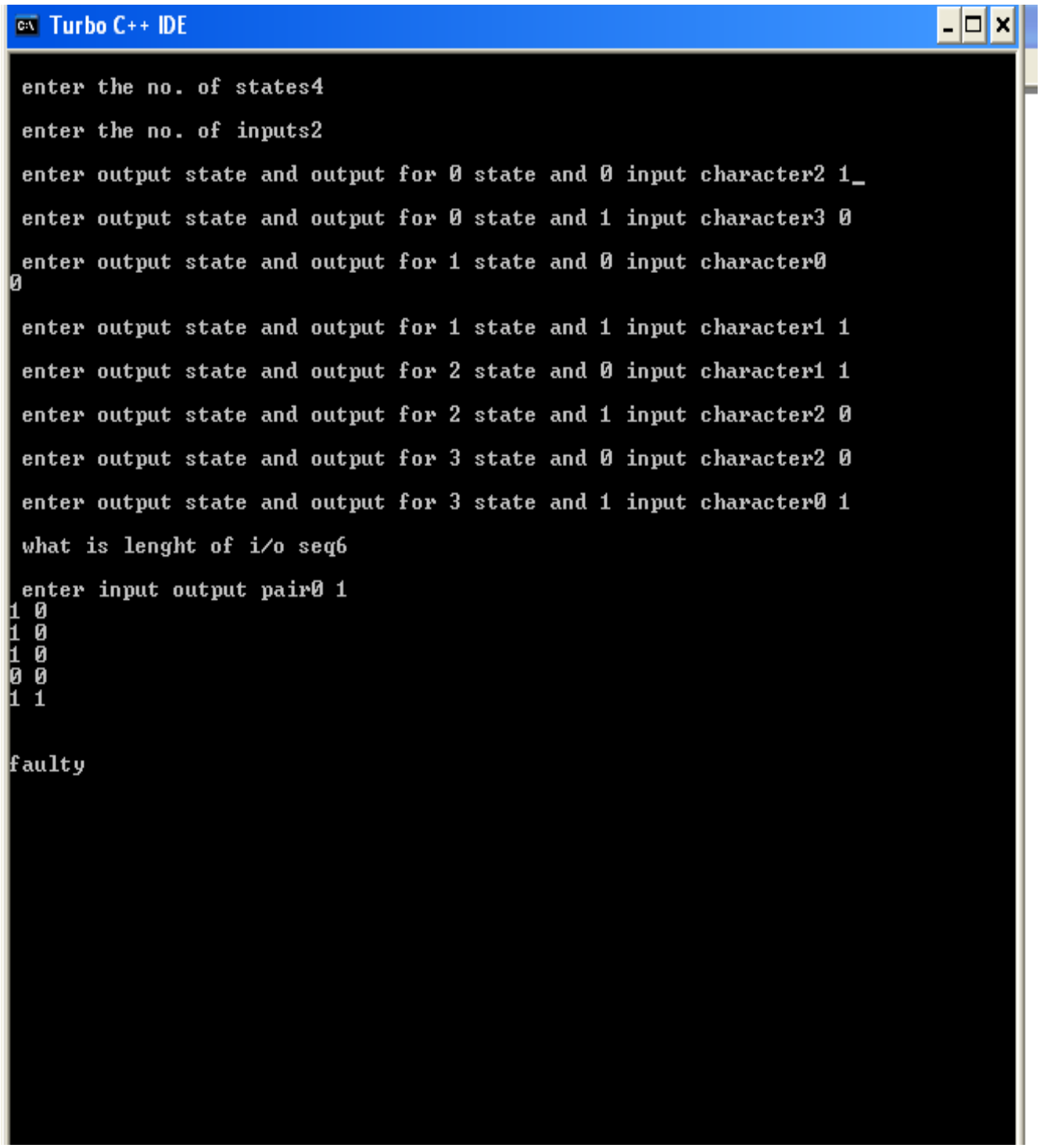


Figure 4.3 Represent the faulty implementation of FSM

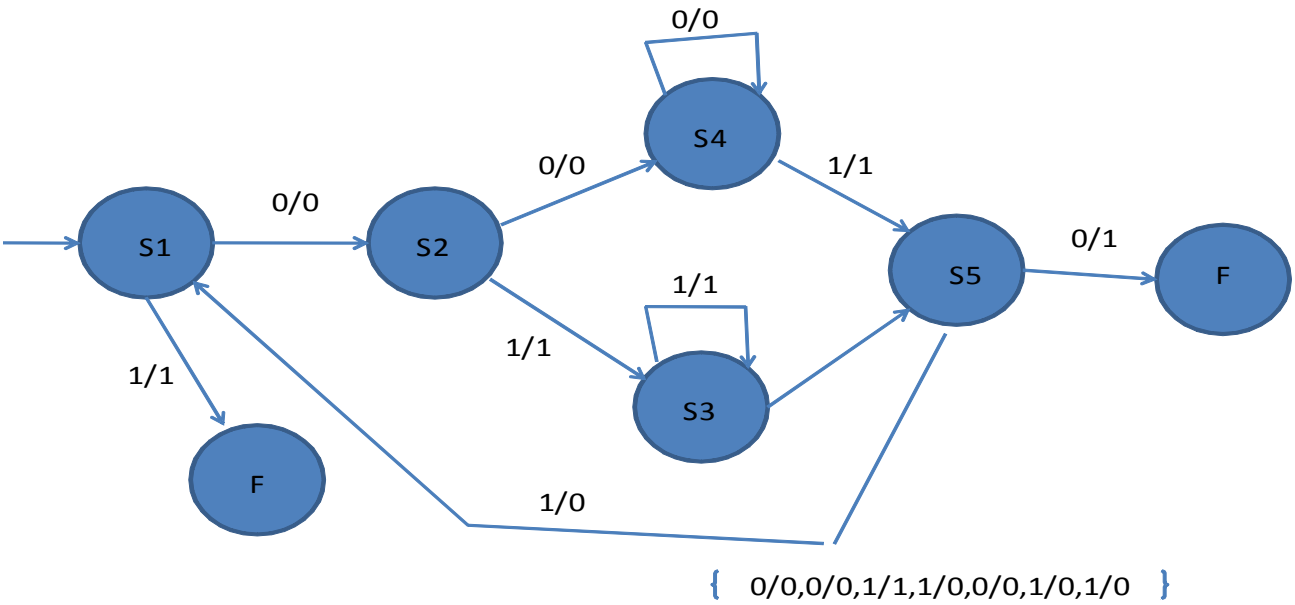
4.5 Fault Detection algorithms 2's Overview

We compare the observed input/output sequence of the implementation machine *M* with the expected behavior of the specification machine *M'*. *M* is considered "faulty" if its behavior is different than that

of *M'*. That is, there is no state in *M'* that would display the input/output sequence observed from *M*. 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 *M*. We define the observer as the entity possessing passive testing fault management functionality. The fault detection procedure can be described as follows.

1. Since we do not know the state of *M* when the observation starts. We assume that *M* could be in any state representing a state of *M'*. Let L^0 designate this initial set of possible states.
2. Once the observer observes the first input (i_1) together with the corresponding output (o_1), transitions in *M'* lead from states of L^0 to another set of possible states .
3. As each input/output pair (I_1 / o_1) is observed, a new set L^j is produced similarly from the set L^{j-1}. The sequence $L^0, L^1, ...L^{j-1}, L^j$ is formed such that, with every observation (I_1 /o_1), only the subset of states from that can still accept an input and produce an output will survive, and they are going to be replaced by their successor states after applying the transition (I_1 /o_1) . The $\{L^i\}$ sequence is monotonically non increasing in size, due to the assumption that the machine is deterministic.
4. As the *i/o* sequence progresses, more observed input/outputs will join the observation sequence $e= I_1 /o_1 ,I_2 /o_2, ...I_k /o_k$. Consequently, the sequence of sets of possible states $L= L^0, L^1, ...L^{j-1}$ is obtained.
5. 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.
6. 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.
7. 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 *M* is deterministic and complete, it will end up in some state and be able to continue, but no such state continuation is specified in *M'*.

   An example of a FSM model and the passive testing fault detection algorithm is shown in Figure 4.4, where *x* is the observed input/output sequence.



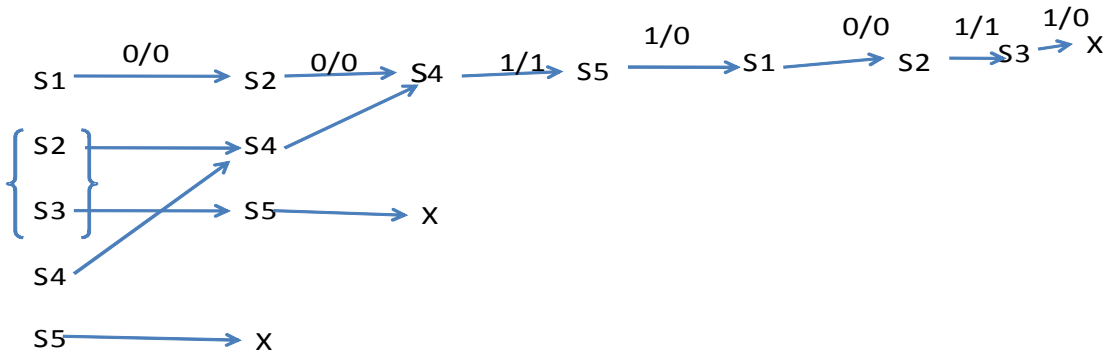


Figure 4.4 An example FSM for the specification [10,11]

4.6 Fault Identification

Fault Identification covers the theorems and algorithms for the fault identification process.

4.6.1 The Backward Trace and Forward Trace

In backward trace we obtained the sequence of sets, $L^0, L^1, \ldots L^{j-1}$ from the observed input/output sequence. Now, let us we have an observed input/output sequence $i_1/o_1, i_2/o_2, \ldots i_{k-1}/o_{k-1}, (i_k/o_k)$ and the resulting sequence of sets $L^0, L^1, \ldots L^{j-1}, L^j$ 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 as follows:

$(L^{J-1})^R$ Contains all states that are head states of transitions with input/output with tail states being members of $(L^k)^R$.

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

Theorem 1: The corresponding backward and forward sets are disjoint, i.e.$(L^J) \cap (L^J)^R = \Phi$[10]

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 down (L^J) to and then using the common state between it and the$(L^k)^R$, 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. $(L^k) = \Phi$.

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 $L^0 \cap (L^0)^R = \Phi$, will contain no state.

4.6.2 Output Fault identification

Theorem 2: If L^J has a state 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 $--(i_{j+1}/o_{j+1}) \rightarrow \delta(s_p, i_{j+1})$ could have occurred.

Proof: By definition of the output fault only the output symbol is altered. For a transition s_p $--(\alpha/\beta) \rightarrow$ s_q to be an output fault, there should be a transition in the specification FSM s_p $--(\alpha/\gamma) \rightarrow$ s_q where . So, s_q should be (s_p, i_{j+1}) assuming that $(s_p, i_{j+1}) \neq F$ (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 $sp \text{ --}(i_{j+1} / o_{j+1}) \rightarrow \delta(i_{j+1}/ o_{j+1})$ to occur if the state s p is in L_j, $\delta(s_p, i_{j+1})$ is in $(L_j)^R, \neq \lambda(s_j, i_{j+1})$ and the observed input/output tuple is (I_{j+1} / o_{j+1}).

4.5.3 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: 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 $s_r \varepsilon (L_{j+1})^R$, 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. We now describe our "forward-backward-crossover" algorithm fault could have occurred.

4.7 A Case study of fault identification

Using the same FSM specification as shown in figure 4.4, 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 4.5 along with "crossovers".

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

Applying theorem 2 we see that this crossover depicts an output fault of transition *s3--(1/1)→ s3* changing to *s3-- (1/0)→ s3*.

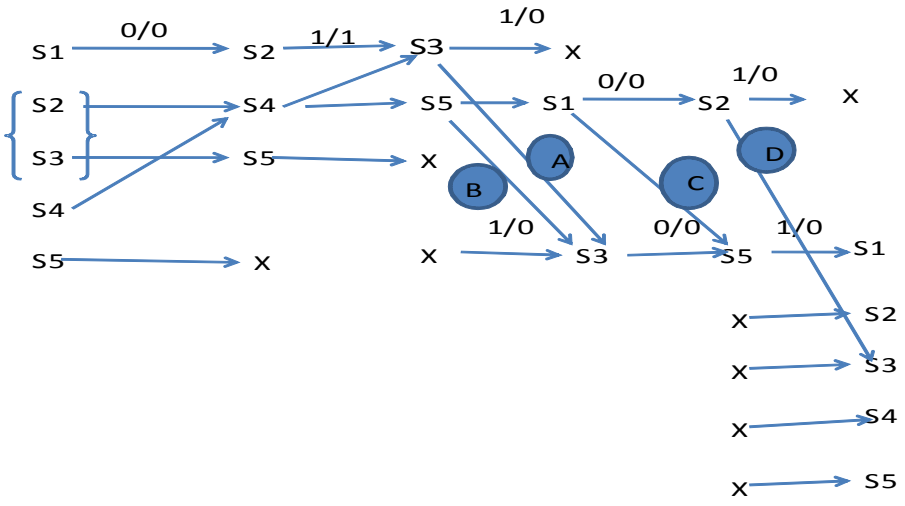


Figure 4.5 The forward and backward traces [11]

2. Applying theorem 3 we see that this crossover depicts a tail-state fault of transition *s5-- (1/0)→ s1* changing to *s5-- (1/0)→ s3*.
3. Applying theorem 3 we see that this crossover depicts a tail-state fault of transition *s1-- (0/0)→ s2* changing to *s1--(0/0)→ s5*.
4. Applying theorem 2 we see that this crossover depicts an output fault of transition *s2-- (1/1)→ s3* changing to *s2 --(1/0)→s3*

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.

4.7.1 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 in the forward trace analysis.

In this 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 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.

The above figure 4.6 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 *s2-- (0/0)→ s4* to *s2 --(0/0)→ s1* then the forward analysis, assuming this fault,

1. *s1--(0/0)→ s2--(0/0)→ s1 --(0/0)→ s2-- (0/0)→ s1 --(0/0)→s2 --(1/1)→ s3 --(0/0)→ s5*
2. *s2--(0/0)→s1 --(0/0)→ s2-- (0/0)→ s1--(0/0)→ s2-- (0/0)→ s1--(1/1)→ X*
3. *s3-- (0/0)→s5 --(0/0)→X*
4. *s4 --(0/0)→s4--(0/0)→ s4--(0/0)→ s4--(0/0)→ s4 --(0/0)→ s4--(1/1)→ s5--(0/0)→X.*
5. *s5 --(0/0)→X.*

Thus, as seen by this analysis, if initially in state , the observed input/output sequence would indeed be produced.

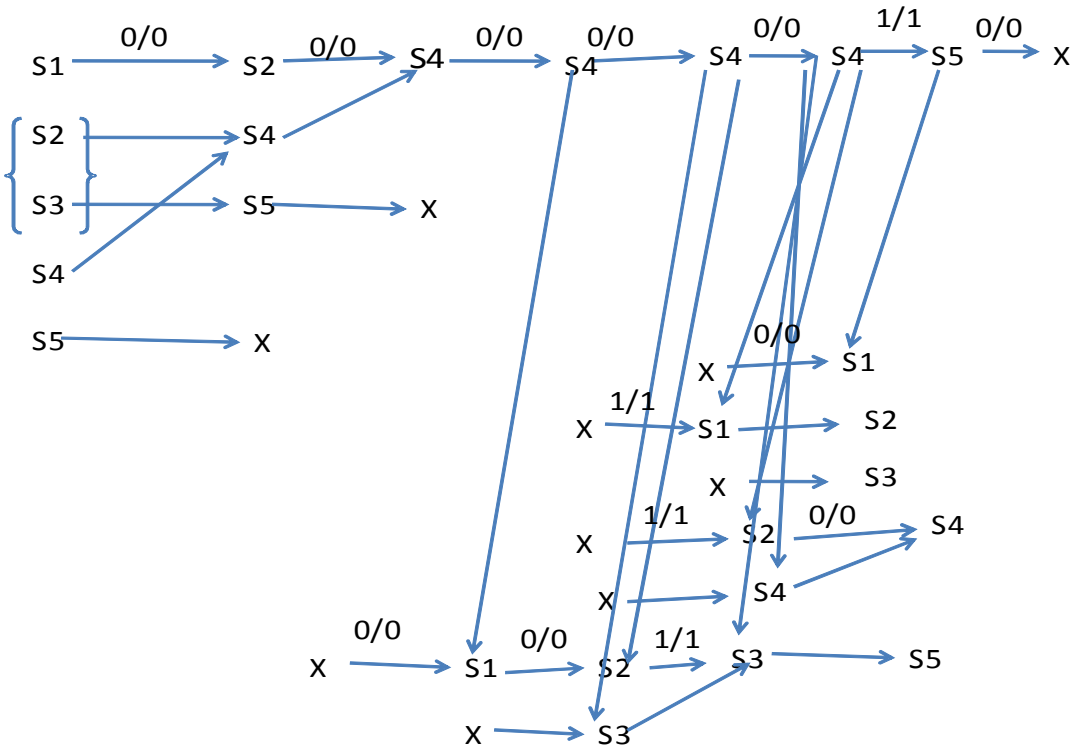


Figure 4.6 show the Recurrent Fault [10,11,16]

This *s2--(0/0)→ s1* 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 *s2--(0/0)→s4* to *s2 --(0/0)→ s2* and *s3 --(0/0)→ s5* to *s3 --(0/0)→ s3*.

4.7.2 Faults

The passive fault is to improve fault identification. At the end of the case study, passive faults can arise and may go unidentified using our standard forward/backward crossover algorithm. Some extensions to this approach need to be devised to improve its fault identification capability.

First, we need to characterize passive faults. Then, we describe our extension to our standard fault identification approach to enable the identification of faults.

Finally, 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, 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 *s2 --(0/0)→ s4* to *s2 --(0/0)→ s1* . We notice that before the passive testing detects this fault, the fault occurred again in the implementation before *(=Φ)*. 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.

Chapter 5

Conclusions and Future work

Finite state machines have proved to be a useful model for systems in several different areas and undoubtedly will continue to find new applications. From the theoretical point of view, most of the fundamental problems have been resolved (for deterministic FSM) except that it is still not known how to construct checking sequences deterministically in polynomial time.

The theory of passive testing for other types of finite state systems, such as nondeterministic and probabilistic FSM, is less advanced. From the practical point of view, a lot of issues remain to be explored. Passive testing offers a challenge due to a large number of states and non determinism among others. In Finite state machine based passive fault detection algorithms Compared with all the former fault detection algorithms the algorithms 2 has better performance and provides more information during testing. The results of both theoretical and implementation evaluations confirm the improvement over the new approach of detection algorithms.

We have shown passive testing can be used to reduce the number of faults that could have caused a network implementation to display faulty behavior. 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. In fact in some cases this reduced to a unique fault. 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.

The 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. This somewhat formal approach and way of thinking seems to be quite distant from the techniques currently used in actual network management systems.

There are some research issues have been investigated. In each case, we have noted some problems. In the studies of the fault detection algorithms by applying network passive testing forced to form several fault detection, each of which aims to explore to passive testing algorithms. However, a problem was noted where the network implementation to display the faulty behavior.

In the future work, a new encoding approach might be considered to overcome such a problem. In the studies of fault detection based passive testing, detection fault location and fault overages. In the studies of fault diagnosis in finite state machine based passive testing, heuristics were defined that attempt to lead failures to be observed in some shorter test sequences, which helps to reduce the cost of

fault isolation and identification. The examples studied in the work are comparatively simple. These issues need to be investigated in the future work.

Annexure 1

Reference

1. W**.** Stallings, *"*SNMP, SNMPv2, and CMIP The Practical Guide to Network-Management Standards*,"*Addison-Wesley Publishing Company, 1993.
2. ISODEC 7498-1: 1994 I ITU-T Recommendation X.200 (1994) Information Technology – Open Systems Interconnection - Basic Reference Model: The Basic Model. 1994.
3. Manuscript received January 21, 2005. Aminadabe Barbosa de Sousa is with the Department of Teleinformatic Engineering, Federal University of Ceara Campus do PICI, Bloco 705, C. P.6007, 60455-760 – Fortaleza-CE, Brazil
4. José Neuman de Souza is with the Department of Teleinformatic Engineering, Federal University of Ceara Campus do PICI, Bloco 705, C. P. 6007, 60455-760 – Fortaleza-CE, Brazil
5. Carlos Delfino Carvalho Pinheiro is with Integrate College of Ceara ViscondedeMauá, 1040, 60125-160, Fortaleza-CE, Brazil
6. D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John (1997) "Passive Testing and Applications to Network Management," *Proc. Of ICNP'97*, pp.113-122.
7. R.E. Miller (1998) "Passive Testing of Networks Using a CFSM Specification," *Proc. of IPCCC'98*,pp.111-116.
8. M. Tabourier and A. Cavalli (1999) "Passive testing and application to the GSM-MAP protocol," Information and Software Technology, vol. 41, pp.813-821
9. R. E. Miller and K. Arisha, *Technical Report #4044,* Computer Science Dept., University of Maryland College Park, August 1999.
10. R. E. Miller and K. Arisha, "On Fault Location in Networks by Passive Testing," ZOO0 IEEE International Pegormance Computing and Communications Conference**,** February 2000.
11. R.E. Miller and K.A. Arisha (2001) "Fault Identification in Networks by Passive Testing," Proc. of 34th Annual Simulation Symposium, pp.277-284.
12. J. Wu, Y. Zhao, and X. Yin (2001) "From Active to Passive: Progress in Testing of Internet Routing Protocols," Proc. of FORTE' *01*, pp.101-118.
13. Y. Zhao, X. Yin, and J. Wu (2001) "OnLine Test System, an Application of Passive Testing in Routing Protocols," Proc. of ICN'01, pp.190-195.
14. Y. Zhao, X. Yin, and J. Wu (2001) "OnLine Test System, an Application of Passive Testing in

Routing Protocols," Proc. of ICN'01, pp.190-195.

15. D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin (2002) "A Formal Approach for Passive Testing of Protocol Data Portions," *Proc. Of ICNP'02*, pp.122-131.
16. R. Miller and K. Arisha, "On Fault identification in Networks by Passive Testing," Technical Report #4207NMIACS-TR#2001-03, Department of Computer Science, University of Maryland, College Park, January 2001.
17. M.H. DeGroot and M.J. Schervish. Probability and Statistics. Boston: Addison-Wesley, 2002.
18. D. Chen, J. Wu, and T.L. Chu (2003) "An Enhanced Passive Testing Tool for Network Protocols," Proc. of ICCNMC'03, pp.513-516.
19. R. Neapolitan and K. Naimipour. Foundations of Algorithms Using C++ Pseudocode, 3rd Edition. Sudbury, Mass.: Jones & Bartlett Publishers, 2003.
20. B. Alcalde, A. Cavalli, D. Chen, D. Khuu and D. Lee (2004) "Network Protocol System Passive Testing for Faulty Management - a Backward Checking Approach," *Proc. of IFIP FORTE'04*, *LNCS,* vol. 3235, pp.150-166.
21. D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin (2006) "Network Protocol System Monitoring – A Formal Approach with Passive Testing," IEEE/ACM Transactions on Networking, vol.14, pp.424-437.
22. R. E. Miller, "Passive Testing of Networks Using a CFSM Specification," Bell Labs Technical Memorandum**,** BLOll345-97-0522-03TM.
23. Marine Tabourier, Ana Cavalli and Melania Ionescu Institut National des Télécommunications, 9, rue Charles Fourier, 91011 Evry Cedex, France" A GSM-MAP Protocol Experiment Using Passive Testing" pp2
24. *David Lee Mihalis Yannakakis* AT&T Bell Laboratories Murray Hill, New Jersey "PRINCIPLES AND METHODS OF TESTING FINITE STATE MACHINES A SURVEY"
25. Hasan Ural, Zhi Xu and Fan Zhang SITE, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada "An Improved Approach to Passive Testing of FSM-based Systems"
26. Aminadabe B. de Sousa, José Neuman de Souza, José Everardo Bessa Maia, and Carlos Delfino C. Pinheiro An "Algorithm for Fault Location in SDH/WDM Networks"
27. S. Fujiwara* , G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi Departement d'informatique et de recherche operationnelle Universite de Montreal Montreal, Canada **"**Test selection based on finite state models**"**
28. A. Petrenko and G. v. Bochmann "On Fault Coverage of Tests for Finite State Specifications" pp 5

Annexure 2

Listof Abbreviations

FSM	Finite state machine
IUT	Implementation under test
SUT	System under test
DS	Distinguishing sequence
UIO	Unique input output
I/O	Input/output
CS	Characterizing set
MFSM	Modular Finite state machine
EFSM	Extended Finite state machine
NDFSM	Non deterministic Finite state machine
CFLA	Correlated Fault location Algorithms
IMP	Implementation

PAPER PUBLISHED

Pradeep Kumar and Ajay Kumar, "Comparative Study of FSM Based Fault detection Algorithms", Paper is communicated with National Conference of "Next Generation Computing and Information System", Department of Computer Science and Engineering Kot Bhalwal, Jammu on October 17 -18, 2008